## СОДЕРЖАНИЕ

Введение ..... 4

1 Аналитический раздел ..... 5

1.1 Описание и формализация объектов сцены ..... 5

1.1.1 Объекты сцены ..... 5

1.1.2 Выбор формы представления трехмерных объектов ..... 5

1.2 Выбор и анализ алгоритмов удаления невидимых ребер и поверхностей ..... 6

1.2.1 Алгоритм обратной трассировки ..... 6

1.2.2 Алгоритм Робертса ..... 7

1.2.3 Алгоритм, использующий Z-буфер ..... 8

1.3 Анализ методов закрашивания ..... 9

1.3.1 Простая закраска ..... 9

1.3.2 Закраска по Гуро ..... 10

1.3.3 Закраска по Фонгу ..... 11

1.4 Алгоритмы моделирования броуновского движения ..... 12

1.4.1 Классическое броуновское движение ..... 13

1.4.2 Алгоритм срединных смещений ..... 13

1.4.3 Фрактальное броуновское движение ..... 14

2 Конструкторский раздел ..... 17

2.1 Алгоритм срединных смещений ..... 17

2.2 Алгоритмы отрисовки ..... 19

2.3 Диаграмма классов ..... 21

3 Технологический раздел ..... 22

3.1 Требования к программному обеспечению ..... 22

3.2 Средства реализации ..... 22

3.3 Модули программы ..... 22

3.4 Реализация алгоритмов ..... 23

3.5 Тестирование алгоритмов ..... 25

4 Исследовательский раздел ..... 27

4.1 Демонстрация работы программы ..... 27

4.2 Измерение времени работы реализаций алгоритмов . . . . . 28
Заключение . . . . . 32
Список использованных источников . . . . . 33

# ВВЕДЕНИЕ

С развитием компьютерных технологий компьютерная графика приобрела совершенно новый статус, поэтому сегодня она является основной технологией в цифровой фотографии, кино, видеоиграх, а также во многих специализированных приложениях. Было разработано большое количество алгоритмов отображения. Главными критериями, которые к ним предъявляются, являются реалистичность изображения и скорость отрисовки. Однако зачастую чем выше реалистичность, тем больше времени и памяти требуется для работы алгоритма.

Одним из направлений моделирования является моделирование движения частиц. Имеется огромная потребность в качественной и эффективной отрисовке распространения частиц вируса. Особенно эта тема стала актуальной после начала пандемии короновируса. Пандемия COVID-19 повлияла на жизнь миллионов людей по всему миру. Помимо серьезных последствий для здоровья, пандемия также изменила нашу повседневную жизнь, перевернула рынок вакансий и подорвала экономическую стабильность. В данном курсовом проекте речь пойдет о моделировании распростарнения частиц вирусной инфекции.

Цели данной курсовой работы — разработать программу с пользовательским интерфейсом, которая предоставит функционал для моделирования броуновского движения частиц короновирусной инфекции в помещении с учетом скорости их распространения и времени жизни на разных поверхностях.

Задачи, которые необходимо выполнить для достижения поставленной цели:

— изучить алгоритмы удаления невидимых линий и поверхностей и методы закраски;

— проанализировать алгоритмы моделирования броуновского движения;

— выбрать подходящие для решения поставленной задачи алгоритмы и реализовать их;

— формализовать модель и описать выбранные типы и структуры данных;

— провести анализ производительности программного обеспечения.

# 1 Аналитический раздел

В данном разделе представлено описание объектов сцены, а также обоснован выбор алгоритмов, которые будут использованы для ее визуализации.

## 1.1 Описание и формализация объектов сцены

### 1.1.1 Объекты сцены

Объекты сцены:

— абстрактная фигура человека;

— стены;

— пол;

— частицы вируса.

Стены и пол представляют собой параллелепипеды. Частицы вируса представлены в форме шаров.

### 1.1.2 Выбор формы представления трехмерных объектов

Отображением формы и размеров объектов являются модели. Обычно используются три формы задания моделей:

— каркасная;

— поверхностная;

— объемная.

Каркасная модель — одна из простейших форм задания модели, так как заключается в хранении информации только о вершинах и ребрах объекта.

Поверхностная модель объекта — это оболочка объекта, пустая внутри. Такая информационная модель содержит данные только о внешних геометрических параметрах объекта. Такой тип модели часто используется в компьютерной графике. При этом могут использоваться различные типы поверхностей, ограничивающих объект, такие как полигональные модели, поверхности второго порядка и другие.

При объемном моделировании учитывается материал, из которого изготовлен объект. Так, известно с какой стороны поверхности расположен материал. Это реализовывается с помощью указания направления внутренней нормали.

Для решения поставленной задачи будет использована поверхностная модель, так как каркасные модели могут привести к неправильному восприятию формы объекта, а

реализация объемной модели потребует большего количества ресурсов на отображение деталей, не влияющих на качество решения задачи в ее заданной формулировке.

В свою очередь поверхностная модель может задаваться параметрическим представлением или полигональной сеткой.

В случае полигональной сетки форма объекта задаётся некоторой совокупностью вершин, ребер и граней. Наиболее подходящим представлением сцены в условиях поставленной задачи будет представление в виде списка граней, так как оно позволяет проводить явный поиск вершин грани и самих граней, которые окружают вершину.

## 1.2 Выбор и анализ алгоритмов удаления невидимых ребер и поверхностей

### 1.2.1 Алгоритм обратной трассировки

Алгоритм обратной трассировки лучей отслеживает лучи в обратном направлении (от наблюдателя к объекту). Считается, что наблюдатель расположен на положительной полуоси z в бесконечности, поэтому все световые лучи параллельны оси z. В ходе работы испускаются лучи от наблюдателя и ищутся пересечения луча и всех объектов сцены. В результате, пересечение с максимальным значением z является видимой частью поверхности, и атрибуты данного объекта используются для определения характеристик пикселя, через центр которого проходит данный световой луч.

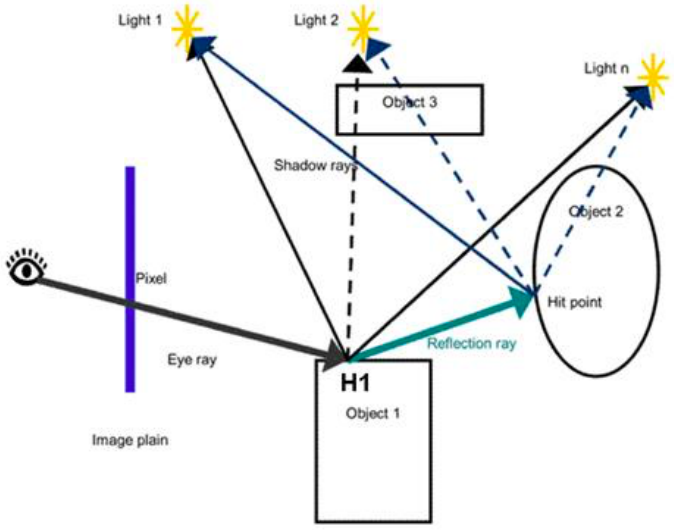


Рисунок 1.1 — Алгоритм обратной трассировки

Эффективность процедуры определения пересечений луча с поверхностью объекта оказывает самое большое влияние на эффективность всего алгоритма. Чтобы избавиться от ненужного поиска пересечений было придумано искать пересечение луча с объемной оболочкой рассматриваемого объекта. Под оболочкой понимается некоторый простой объект,

внутрь которого можно поместить рассматриваемый объект, к примеру параллелепипед или сферу.

В дальнейшем при рассмотрении пересечения луча и объемной оболочкой рассматриваемого объекта, если такого пересечения нет, то и соответственно пересечения луча и самого рассматриваемого объекта нет, и наоборот, пересечение найдено, то возможно, есть пересечение луча и рассматриваемого объекта.

Для расчета эффектов освещения сцены проводятся вторичные лучи от точек пересечения ко всем источникам света. Если на пути этих лучей встречается непрозрачное тело, значит данная точка находится в тени, иначе он влияет на освещение данной точки. Также для получения более реалистичного изображения сцены, нужно учитывать вклады отраженных и преломленных лучей.

Достоинства алгоритма:

— возможность использования алгоритма в параллельных вычислительных системах.

Недостаттки алгоритма:

— требуется большое количество высислений;

— производительность алгоритма.

### 1.2.2 Алгоритм Робертса

Алгоритм Робертса работает в объектном пространстве, кроме того работает только с выпуклыми телами. Если тело изначально является не выпуклым, то нужно его разбить на выпуклые составляющие.

Данный алгоритм состоит из следующих этапов:

— подготовка исходных данных;

— удаление линий, экранируемых самим телом;

— удаление линий, экранируемых другими телами.

Для определения, лежит ли точка в положительном подпространстве, используют проверку знака скалярного произведения $(l, n)$, где $l$ — вектор, направленный к наблюдателю, фактически определяет точку наблюдения; $n$ — вектор внешней нормали грани. Если $(l, n) > 0$, т. е. угол между векторами острый, то грань является лицевой. Если $(l, n) < 0$, т. е. угол между векторами тупой, то грань является нелицевой. В алгоритме Робертса требуется, чтобы все изображаемые тела или объекты были выпуклыми. Невыпуклые тела должны быть разбиты на выпуклые части. В этом алгоритме выпуклое многогранное тело с плоскими гранями должно представиться набором пересекающихся плоскостей. Уравнение произвольной плоскости в трехмерном пространстве имеет вид 1.1.

$$ax + by + cz + d = 0 \tag{1.1}$$

В матричной форме 1.1 выглядит как 1.2.

$$[x\ y\ z\ 1][P]^T = 0 \tag{1.2}$$

В формуле 1.2 выражение $[P]^T = [a\ b\ c\ d]$ представляет собой плоскость. Поэтому любое выпуклое твердое тело можно выразить матрицей тела, состоящей из коэффициентов уравнений плоскостей, т. е.

$$M = \begin{bmatrix} a_1 & a_2 & ... & a_n \\ b_1 & b_2 & ... & b_n \\ c_1 & c_2 & ... & c_n \\ d_1 & d_2 & ... & d_n \end{bmatrix}, \tag{1.3}$$

где каждый столбец содержит коэффициенты одной плоскости.

Любая точка пространства может быть представлена в однородных координатах вектором $[S] = [x\ y\ z\ 1]$. Более того, если точка $[S]$ лежит на плоскости, то $[S] * [P]^T = 0$. Если же $[S]$ не лежит на плоскости, то знак этого скалярного произведения показывает, по какую сторону от плоскости расположена точка. В алгоритме Робертса предполагается, что точки, лежащие внутри тела, дают отрицательное скалярное произведение, т. е. нормали направлены наружу.

Достоинства алгоритма:

— высокая точность вычислений.

Недостатки алгоритма:

— рост числа трудоемкости алгоритма, как квадрата числа объектов;

— работа только с выпуклыми телами.

### 1.2.3 Алгоритм, использующий Z-буфер

Алгоритм Z-буфера решает задачу в пространстве изображений.

В данном алгоритме рассматривается два буфера. Буфер кадра (регенерации) используется для заполнения атрибутов (интенсивности) каждого пикселя в пространстве изображения. В Z-буфер (буфер глубины) можно помещать информацию о координате z для каждого пикселя.

Для начала необходимо подготовить буферы. Для этого в Z-буфер заносятся максимально возможные значения z, а буфер кадра заполняется значениями пикселя, который описывает фон. Также нужно каждый многоугольник преобразовать в растровую форму и записать в буфер кадра. Сам процесс работы заключается в сравнении глубины каждого нового пикселя, который нужно занести в буфер кадра, с глубиной того пикселя, который уже занесён в Z-буфер. В зависимости от сравнения принимается решение, нужно ли заносить новый пиксель в буфер кадра и, если нужно, также корректируется Z-буфер (в него нужно занести глубину нового пикселя).

Достоинства алгоритма:

— элементы сцены заносятся в буфер кадра в произвольном порядке, поэтому в данном алгоритме не тратится время на выполнение сортировок;

— произвольная сложность сцены;

— поскольку размеры изображения ограничены размером экрана дисплея, трудоемкость алгоритма зависит линейно от числа рассматриваемых поверхностей.

Недостатки алгоритма:

— трудоемкость устранения лестничного эффекта;

— трудности реализации эффектов прозрачности;

— большой объем требуемой памяти.

## 1.3 Анализ методов закрашивания

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

Существует несколько основных методов закраски:

— простая закраска;

— закраска по Гуро, основанная на интерполяции значений интенсивности освещенности поверхности;

— закраска по Фонгу, основанная на интерполяции векторов нормалей к граням многогранника.

### 1.3.1 Простая закраска

Одной из самых простых моделей освещения является модель Ламберта. Она учитывает только идеальное диффузное отражение света от тела. Считается, что свет падающий

в точку, одинаково рассеивается по всем направлениям полупространства. Таким образом, освещенность в точке определяется только плотностью света в точке поверхности, а она линейно зависит от косинуса угла падения. При этом положение наблюдателя не имеет значение, так как диффузно отраженный свет рассеивается равномерно по всем направлениям.

Простая закраска используется при выполнении трех условий:

— предполагается, что источник находится в бесконечности;

— предполагается, что наблюдатель находится в бесконечности;

— закрашиваемая грань является реально существующей, а не полученной в результате аппроксимации поверхности.

Большим недостатком данной модели является то, что все точки грани будут иметь одинаковую интенсивность.

Рисунок 1.2 — Пример простой закраскии

### 1.3.2 Закраска по Гуро

Данный алгоритм предполагает следующие шаги:

— вычисление векторов нормалей к каждой грани;

— вычисление векторов нормали к каждой вершине грани путем усреднения нормалей к граням;

— вычисление интенсивности в вершинах грани;

— интерполяция интенсивности вдоль ребер грани;

— линейная интерполяция интенсивности вдоль сканирующей строки.

Достоинства:

— хорошо сочетается с диффузным отражением;

— изображение получается более реалистичным, чем при простой закраске.

Недостатки:

— данный метод интерполяции обеспечивает лишь непрерывность значений интенсивности вдоль границ многоугольников, но не обеспечивает непрерывность изменения интенсивности.

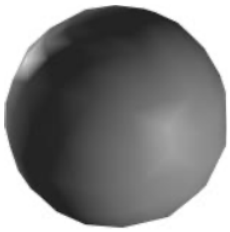

Рисунок 1.3 — Пример закраски по Гуро

### 1.3.3 Закраска по Фонгу

При такой закраске, в отличие от метода Гуро, вдоль сканирующей строки интерполируется значение вектора нормали, а не интенсивности.

Шаги алгоритма:

— вычисление векторов нормалей в каждой грани.

— вычисление векторов нормали к каждой вершине грани.

— интерполяция векторов нормалей вдоль ребер грани.

— линейная интерполяция векторов нормалей вдоль сканирующей строки.

— вычисление интенсивности в очередной точке сканирующей строки.

Достоинства:

— можно достичь лучшей локальной аппроксимации кривизны поверхности.

Недостатки:

— ресурсоемкость;

— вычислительная сложность.

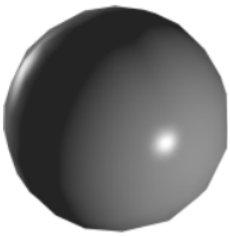

Рисунок 1.4 — Пример закраски по Фонгу

## 1.4 Алгоритмы моделирования броуновского движения

Броуновское движение — беспорядочное движение малых частиц, взвешенных в жидкости или газе, происходящее под действием молекул окружающей среды.

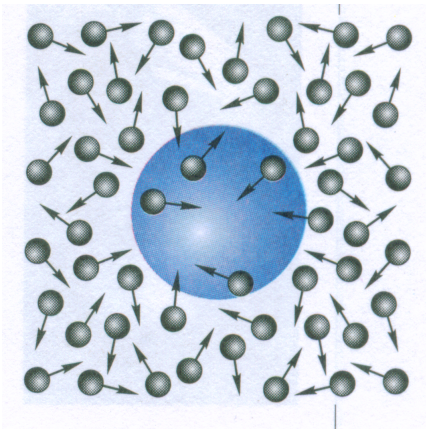

Рисунок 1.5 — Броуновское движение

Рассмотрим случайный процесс (случайную величину) $X(t)$, заданную на отрезке $[0,T]$.

Случайный процесс $X(t)$ называется одномерным броуновским движением на интервале $[0,T]$, если он обладает следущими свойствами:

— $X(0) = 0$ почти наверное и $X(t)$ - почти наверное непрерывная функция на $[0,T]$

— $X(t)$ - процесс с независимыми приращениями

— $X(t)$ - процесс с приращениями, распределёнными нормально.

Отметим следующие свойства броуновского движения:

— $X(t)$ почти наверное нигде не дифференцируем

— $X(t)$ - марковский процесс (не обладает памятью), т.е. если известна величина $X(t)$, то при $t_1 < t < t_2$ величины $X(t_1)$ и $X(t_2)$ независимы.

— Фрактальная размерность графика $X(t)$ равна 1.5

— Приращение $X(t)$ обладает свойством статистического самоподобия: для любого $r > 0$

$$X(t + \triangle t) = \frac{1}{\sqrt{r}}(X(t + r \triangle t) - X(t)) \tag{1.4}$$

— Стационарность приращений: дисперсия приращения зависит только от разности моментов времени

$$D(X(t_2) - X(t_1)) = \sigma^2 |t_2 - t_1| \tag{1.5}$$

— Математическое ожидание приращения равно

$$E(|X(t_2) - X(t_1)|) = \sqrt{\frac{2}{\pi}}\sigma\sqrt{|t_2 - t_1|} \tag{1.6}$$

Для моделрования броуновского движения можно воспользоваться разными алгоритмами. Рассмотрим несколько из них.

### 1.4.1 Классическое броуновское движение

Проще всего реализовать дискретную реализацию броуновского движения, рассмотрев последовательность $x_0 = 0$, $x_{n+1} = x_n + g_n$, где $g_n$ - случайная величина, имеющая нормальное распределение (например, $N(0,1)$).

1: $array[N]$
2: $array[0] \leftarrow 0$
3: **for** i = 1,..., $N$ **do**
4: $\quad array[i+1] \leftarrow array[i] + randomNormal(0,1)$
5: **end for**

### 1.4.2 Алгоритм срединных смещений

Метод случайного срединного смещения основан на работах Н.Виннера , он более сложен, чем метод из предыдущего параграфа, однако используется для конструктивного доказательства существования броуновского движения, а также для построения фрактальной интерполяции (когда необходимо чтобы кривая проходила через заданные точки интерполяции). Метод также может быть обобщен на случай $n$-мерных броуновских движений.

Алгоритм случайного срединного смещения вычисляет значения $X(t)$ в диадических рациональных точках вида $\frac{k}{2^n} \in [0,1]$. Последовательно вычисляются значения в середине отрезка [0,1], а затем в серединах отрезков $[0, \frac{1}{2}]$ и $[\frac{1}{2}, 1]$ и т.д. На каждом шаге итерации должен выполнятьяс закон дисперсии для приращения в вычисленных точках. Параметр $\sigma$ определяет масщтаб по вертикальной оси, не влияя на фрактальную размерность графика.

Вход: $N$, $\sigma$ // $N$ - число шагов алгоритма, при этом всего $2^N + 1$ точек интерполяции, $\sigma$ - параметр вертикального масштаба, коэффициент дисперсии

Выход: массив значений $\left\{X(\frac{k}{2^N})\right\}_{k=0}^{2^N}$ // реализация броуновского движения $X(t)$ на дискретном множестве точек вида $t_k = \frac{k}{2^N}$, k $\in [0, 2^N]$

1: $X(0) \leftarrow 0$
2: $X(1) \leftarrow \sigma g$ // g - случайная величина, распределенная нормально с параметрами $N(0,1)$
3: **for** j = 1,..., N **do**
4: $\quad$ **for** i = 1,..., $2^{N-1}$ **do**

5: $X((2i-1)2^{N-j}) \leftarrow X((i-1)2^{N-j+1}) + X(i2^{N-j+1}) + \frac{1}{2^{(j+1)/2}}\sigma$ g

6: **end for**

7: **end for**

### 1.4.3 Фрактальное броуновское движение

Фрактальное броуновское движение (ФБД) уже не является марковским процессом, а обладает некорой "памятью". Кроме того, вводя параметр $0 < H < 1$ можно получитьодномерное ФБД размерности $d = 2 - H$ и двумерное ФБД размерности $d = 3 - H$ . Заметим, что классическое броуновское движение получается как частный случай при $H = 0.5$. Для апроксимации ФБД нет простого метода, вроде суммирования нормальных случайных величин, как в случае классического броуновского движения. Для апроксимации ФБД наиболее удобно использовать преобразования Фурье.

Рассмотрим случайный процесс (случайную величину) $X(t)$, заданную на отрезке $[0, T]$.

*Случайный процесс* $X(t)$ называется одномерным фрактальным броуновским движением на интервале $[0, T]$, если он обладает следущими свойствами:

— $X(0) = 0$ почти наверное и $X(t)$ - почти наверне непрерывная функция на $[0, T]$

— $X(t)$ - процесс с приращениями, распределенными нормально

Отметим следующие свойства фрактального броуновского движения:

— $X(t)$ почти наверное нигде не дифференцируем

— Фрактальная размерность графика $X(t)$ равна $2 - H$

— Процесс $x(t)$ не обладает свойством независимости приращений

— Приращение *X(t)* обладает свойством статистического самоподобия: для любого $r > 0$

$$X(t + \triangle t) = \frac{1}{\sqrt{r}}(X(t + r \triangle t) - X(t)) \tag{1.7}$$

— Стационарность приращений: дисперсия приращения зависит только от разности моментов времени

$$D(X(t_2) - X(t_1)) = \sigma^2 |t_2 - t_1|^{2H} \tag{1.8}$$

— Математическое ожидание приращения равно

$$E(|X(t_2) - X(t_1)|) = \sqrt{\frac{2}{\pi}}\sigma |t_2 - t_1|^H \tag{1.9}$$

**Теорема 1** *Если $X(t)$ - ФБД с параметром $H$, то его спектральная плотность*

$$S(f) \propto \frac{1}{f^{2H+1}} \tag{1.10}$$

Идея метода состоит в следующем. Строится преобразование Фурье для искомого ФБД в частной области, задавая случайные фазы и подбирая амплитуды, удовлетворяющие свойству из Теоремы 1. Затем получаем ФБД во временной области с помощью обратного преобразования Фурье.

Будем моделировать дискретный аналог ФБД, то есть наша цель- получить величины $\{X_n\}_{n=0}^{N-1}$, апроксимирующиеФБД в точках $n$. Воспользуемся формулой дискретного преобразования Фурье

$$\hat{X_n} = \sum_{k=0}^{N-1} X_k e^{-2\pi kn/N} \tag{1.11}$$

и обратного дискретного преобразования Фурье

$$X_n = \sum_{k=0}^{N-1} \hat{X_k} e^{2\pi kn/N} \tag{1.12}$$

Далее будем рассматривать только четные значения $N$, а для применения метода *быстрого дискретного преобразования Фурье* нужно, чтобы $N = 2^M$, $M \in \mathbb{N}$. Метод быстрого дискретного преобразования Фурье реализован во многих системах компьютерной алгебры. Он позволяет сократить вычисления в $\frac{2N}{\log_2 N}$ раз.

Для того, чтобы получающиеся величины $X_n$ были вещественными мы наложим условие сопряженной симметрии:

$$\hat{X}_0, \hat{X}_{N/2} \in \mathbb{R}, \hat{X}_n = \hat{X}_{N-n}, n = 1,...,N/2-1 \tag{1.13}$$

Фильтрация относится к той части моделирования, когда мы заставляем коэффициенты преобразования Фурье удовлетворять степенному закону из Теоремы 1:

$$|\hat{X}_n|^2 \propto \frac{1}{n^{2H+1}}, n = 1,...,N/2 \tag{1.14}$$

Для этого возьмем

$$\hat{X}_n = \frac{ge^{2\pi iu}}{n^{H+0.5}} \tag{1.15}$$

где $g$ - независимые значения нормально распределенной случайной величины с параметрами $N(0,1)$, а $u$ - независимые значения равномерно распределенной на отрезке [0,1] случайной величины. Оставшиеся коэффициенты вечислим из сотношений 1.15.

Для вычисления искомой аппроксимации ФБД $\{X_n\}_{n=0}^{N-1}$ применим обратное дискретное преобразование Фурье к набору $\left\{\hat{X}_n\right\}_{n=0}^{N-1}$.

Вход: $H \in (0,1)$, $N = 2^M$, $M \in \mathbb{N}$ // $H$ - параметр ФБД, размерность графика равна $d = 2 - H$, $N$ - параметр, определяющий количество точек дискретизации ФБД.

Выход: массив значений $\{X_n\}_{n=0}^{N-1}$ // дискретная апроксимация ФБД в последовательные моменты времени n.

1: $\hat{X}_0 \leftarrow g$
2: **for** j = 1,..., N/2-1 **do**
3: $\quad \hat{X}_j \leftarrow \frac{ge^{2\pi iu}}{j^{H+0.5}}$
4: **end for**
5: $\hat{X}_{N/2} \leftarrow \frac{g\cos(2\pi iu)}{(N/2)^{H+0.5}}$ // Здесь cos — вещественная часть комплексной экспоненты $e$
6: **for** j = N/2+1,..., N-1 **do**
7: $\quad \hat{X}_j \leftarrow \overline{\hat{X}_{N-j}}$
8: **end for**
9: $X \leftarrow convert(\hat{X})$ // Вектор $X = \{X_0,...,X_{N-1}\}$ получается обратным дискретным преобразованием Фурье из вектора $\hat{X} = \left\{\hat{X}_0,...,\hat{X}_{N-1}\right\}$.

### Вывод

В данном разделе было представлено описание объектов сцены, а также обоснован выбор алгоритмов, которые будут использованы для ее визуализации.

Для удаления невидимых линий и поверхностей выбран алгоритм Z-буфера, так как он обладает важными преимуществами — высокой скоростью работы и возможностью отображать сцены произвольной сложности. Также была выбрана локальная модель освещения Ламберта, так как программа не должна выводить реалистичного изображения и должна иметь наиболее высокую производительность, и закраска по Гуро, так как алгоритм достаточно быстр, и он хорошо сочетается с выбранным ранее Z-буфером.

Для моделирования броуновского движеняи был выбран алгоритм срединных смещений, так как он позволяет наиболее реалистично изобразить броуновское движение, а также легко обобщается для случая $n$-мерных движений. Также данный алгоритм содержит меньшее количество вычислений чем алгоритм Фурье-фильтрации, то есть вычсленяи будут выполняться быстрее.

# 2 Конструкторский раздел

В данном разделе будут представлены схемы алгоритмов, выбранных для решения задачи, и диаграмма классов.

## 2.1 Алгоритм срединных смещений

Схема алгоритма срединных смещений изображена на рисунке 2.1.

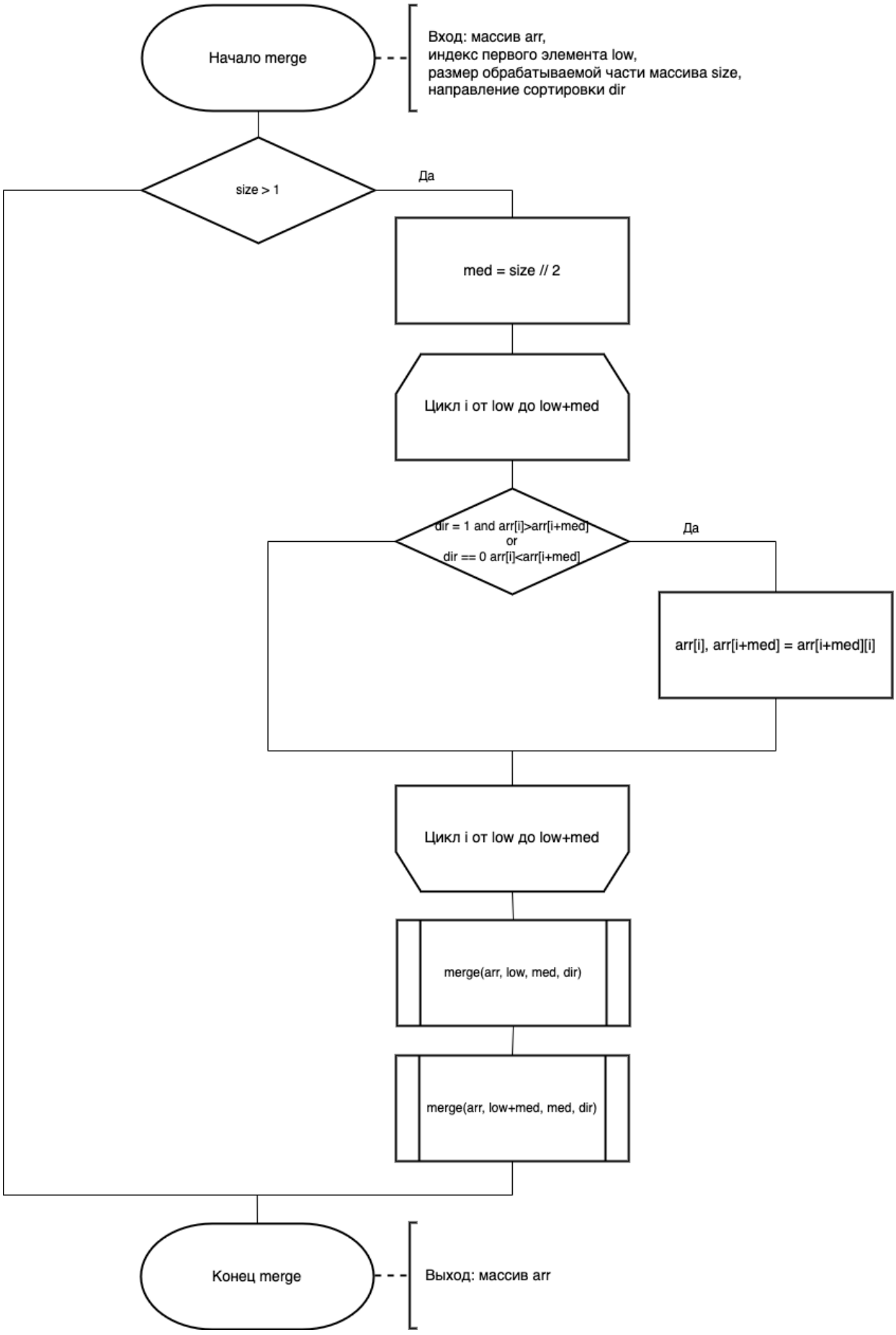


Рисунок 2.1 — Схема алгоритма срединных смещений

## 2.2 Алгоритмы отрисовки

Схема алгоритма Z-буфера изображена на рисунке 2.2.

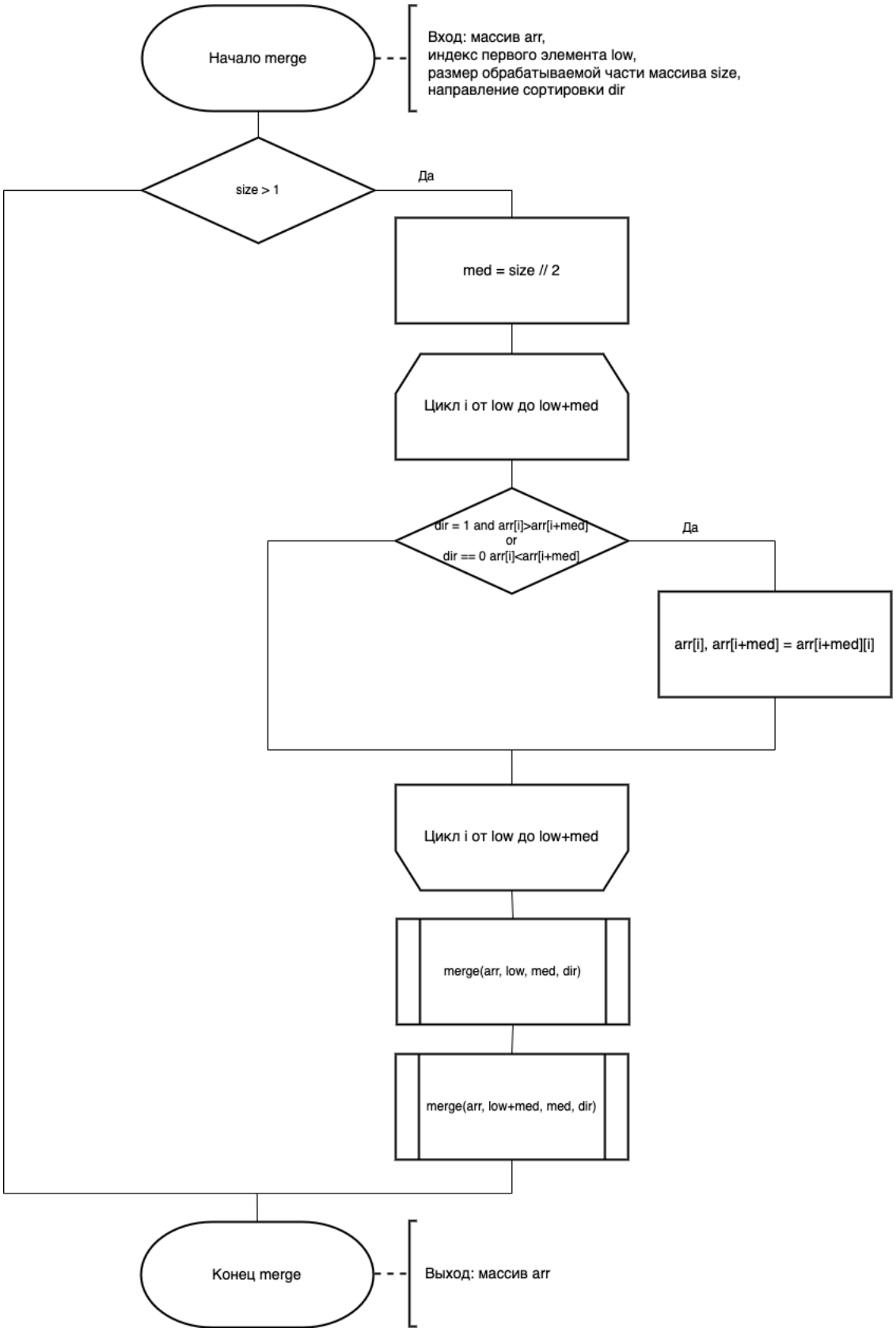


Рисунок 2.2 — Схема алгоритма Z-буфер

Схема алгоритма закраски по Гуро изображена на рисунке 2.3.

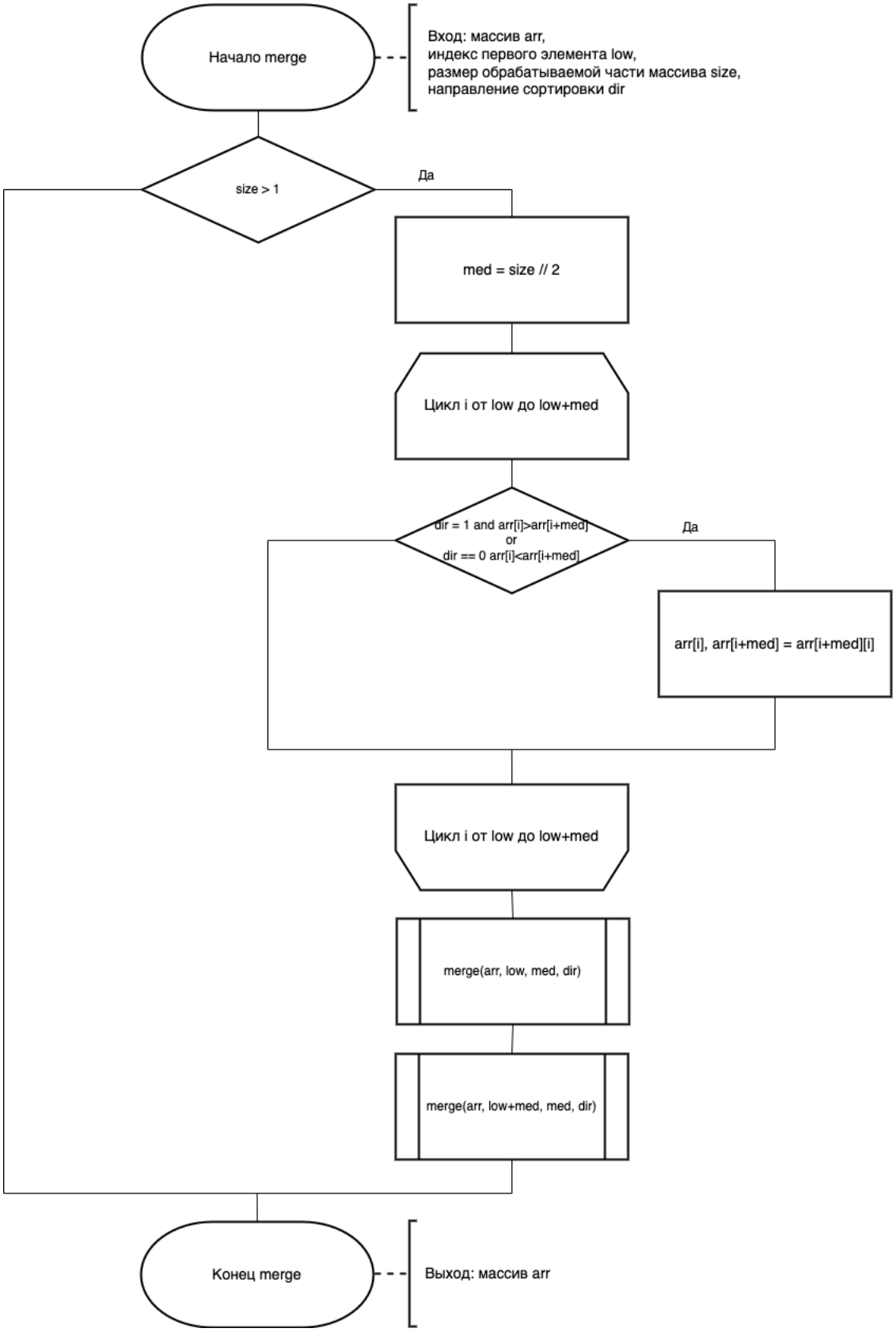


Рисунок 2.3 — Схема алгоритма закраски по Гур

## 2.3 Диаграмма классов

На рисунке 2.4 представлена диаграмма классов.

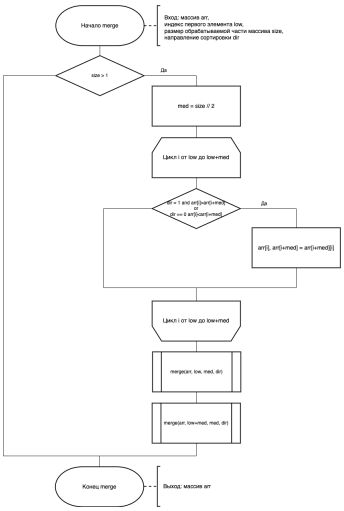


Рисунок 2.4 — Диаграмма классов

Классы объектов сцены:

— Model (класс трёхмерных объектов с возможностью перемещения, масштабирования и поворота вокруг собственного центра);

— Camera (класс камеры с возможностью перемещения);

— LightSourcePoint (класс источника освещения с возможностью) перемещения по сцене и изменения мощности.

Вспомогательные классы сцены:

— Scene (контейнер, который содержит в себе объекты сцены).

### Вывод

В данном разделе были рассмотрены схемы алгоритмов, использованных при отрисовке сцены, а также диаграмма классов.

# 3 Технологический раздел

В данном разделе будет обоснован выбор языка программирования и среды разработки, представлены реализации алгоритмов и рассмотрен интерфейс программы.

## 3.1 Требования к программному обеспечению

Программа должна предоставлять доступ к функционалу:

— возможность выбора материала покрытия пола и стен из предложенных вариантов (дерево, бумага(обои), керамика(плитка));

— изменение скорости движения частиц;

— изменение количества частиц инфекции;

— включение и выключение работы модели распространения частиц;

— вращение, перемещение и масштабирование модели.

## 3.2 Средства реализации

В качестве языка программирования для реализации курсовой работы использовался язык программирования C++ [3], т.к. он содержит возможности для работы с массивами и нативными потоками. В качестве среды разработки использовалась Visual Studio Code [4]. Для замеров времени использовалась функция time() из заголовочного файла time.h [3]. Для отображения результирующих данных в виде графиков была использована библиотека языка Python matplotlib [5].

## 3.3 Модули программы

Программа включает в себя следующие модули:

— main.cpp (главный файл программы, предоставляющий пользователю меню для выполнения основных функций);

— bitonic_sort.cpp и bitonic_sort.h (последовательный алгоритм битонной сортировки);

— parallel_bitonic_sort.cpp и parallel_bitonic_sort.h (параллельный алгоритм битонной сортировки);

— time_test.cpp и time_test.h (файлы, содержащие функции замеров времени работы указанных алгоритмов);

— constants.h (файл, содержащий все необходимые константы).

### 3.4 Реализация алгоритмов

В листингах 3.1 – 3.3 представлена реализация последовательного и параллельного алгоритмов битонной сортировки.

Листинг 3.1 — Реализация вспомогательной функции обмена значениями элементов в массиве

```
void comp_and_swap(vector<int>& arr, unsigned long i, unsigned long j,
int dir)
{
    if (dir == (arr[i] > arr[j]))
    {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }
}
```

Листинг 3.2 — Реализация последовательного алгоритма битонной сортировки

```
def compare_swap(arr, i, j, d):
    if (d == 1 and arr[i] > arr[j]) or (d == 0 and arr[i] < arr[j]):
        arr[i], arr[j] = arr[j], arr[i]

def merge(arr, low, size, d):
    if size > 1:
        high = size // 2
        for i in range(low, low + high):
            compare_swap(arr, i, i + high, d)
        merge(arr, low, high, d)
        merge(arr, low + high, high, d)

def bitonic_sort_rec(arr, low, size, d):
    if size > 1:
        high = int(size / 2)
        bitonic_sort_rec(arr, low, high, 1)
        bitonic_sort_rec(arr, low + high, high, 0)
        merge(arr, low, size, d)

def bitonic_sort(arr):
    bitonic_sort_rec(arr, 0, len(arr), 1)
    return arr
```

Листинг 3.3 — Реализация параллельного алгоритма битонной сортировки

```
void bitonic_merge_parallel(vector<int>& arr, unsigned long low, int size,
    unsigned long iter_count, int thread_number, int dir)
{
```

```
4       for (int i = 0; i < size * iter_count; i += size)
5       {
6           unsigned long med = size / 2;
7           unsigned long start = low + (thread_number * iter_count * size) + i;
8           for (unsigned long k = start; k < start + med; k++)
9           {
10              comp_and_swap(arr, k, k + med, dir);
11          }
12      }
13  }
14
15  void bitonic_sort_rec(vector<int>& arr, unsigned long low,
16      unsigned long size, int dir, int threads_count)
17  {
18      vector<thread> threads(threads_count);
19      if (size > 1)
20      {
21          unsigned long k = size / 2;
22          bitonic_sort_rec(arr, low, k, 1, threads_count);
23          bitonic_sort_rec(arr, low + k, k, 0, threads_count);
24          for (unsigned long j = size; j >= 2; j /= 2)
25          {
26              unsigned long for_count = size / j;
27              unsigned long real_threads_count = min(for_count,
                    threads_count);
28              unsigned long iter_count = for_count / real_threads_count;
29              for (int i = 0; i < real_threads_count; i++)
30              {
31                  threads[i] = thread(bitonic_merge_parallel, ref(arr), low,
                        j, iter_count, i, dir);
32                  threads[i].join();
33              }
34          }
35      }
36  }
37
38  void parallel_bitonic_sort(vector<int>& arr, int threads_count)
39  {
40      bitonic_sort_rec(arr, 0, arr.size(), 1, threads_count);
41  }
```

### 3.5 Тестирование алгоритмов

Было проведено модульное (компонентное) тестирование (unit testing) по позитивному сценарию.

Было реализовано несколько тестов в зависимости от входных данных (таблица 3.1):

— отсортированный массив;

— массив, отсортированный в обратном порядке;

— случайный массив;

— массив из одного элемента.

Все тесты пройдены успешно как для последовательного, так и для параллельного алгоритмов.

Таблица 3.1 — Функциональные тесты

| Массив | Результат | Ожидаемый результат |
|---|---|---|
| $[1, 2, 3, 4, 5, 6, 7, 8]$ | $[1, 2, 3, 4, 5, 6, 7, 8]$ | $[1, 2, 3, 4, 5, 6, 7, 8]$ |
| $[8, 7, 6, 5, 4, 3, 2, 1]$ | $[1, 2, 3, 4, 5, 6, 7, 8]$ | $[1, 2, 3, 4, 5, 6, 7, 8]$ |
| $[5, 1, 8, 3, 2, 6, 4, 7]$ | $[1, 2, 3, 4, 5, 6, 7, 8]$ | $[1, 2, 3, 4, 5, 6, 7, 8]$ |
| $[1]$ | $[1]$ | $[1]$ |

**Вывод**

В данном разделе были перечислены требования к ПО, средства разработки, с помощью которых были реализованы последовательный и параллельный алгоритмы сортировки массивов, приведены листинги реализаций каждого из алгоритмов, а также тесты к ним.

# 4 Исследовательский раздел

В данном разделе приведено сравнение алгоритмов по времени работы. Все параметры замерялись на устройстве со следующими техническими характеристиками:

— операционная система macOS Monterey 12.6 [6];

— оперативная память 16 Гб;

— процессор 2,3 ГГц 8-ядерный Intel Core i9 9 поколения [7];

— 8 физических ядер, 16 логических ядер [7].

Во время замеров ноутбук был подключен к сети питания и нагружен только приложениями, использующимися при замерах.

## 4.1 Демонстрация работы программы

На рисунке 4.1 представлен пример работы программы.

```
МЕНЮ:
1. Последовательный алгоритм
2. Паралельный алгоритм
3. Замер времени
0. Выход
Выбор: 1
Введите количество элементов: 4
4 3 2 1
1 2 3 4
МЕНЮ:
1. Последовательный алгоритм
2. Паралельный алгоритм
3. Замер времени
0. Выход
Выбор: 2
Введите количество элементов: 4
4 3 2 1
Введите число потоков: 2
1 2 3 4
МЕНЮ:
1. Последовательный алгоритм
2. Паралельный алгоритм
3. Замер времени
0. Выход
Выбор: 0
```

Рисунок 4.1 — Пример работы программы

## 4.2 Измерение времени работы реализаций алгоритмов

Процессорное время замерялось при помощи функции time из заголовочного файла time.h [3]. Результаты сформированы в виде графиков при помощи библиотеки matplotlib [5]. Каждый алгоритм был выполнен 30 раз, а затем время усреднялось. Время было замерено на случайно сформированных массивах.

На рисунках 4.2 – 4.4 на осях X указано количество элементов во входном массиве. Время на осях Y указано в секундах.

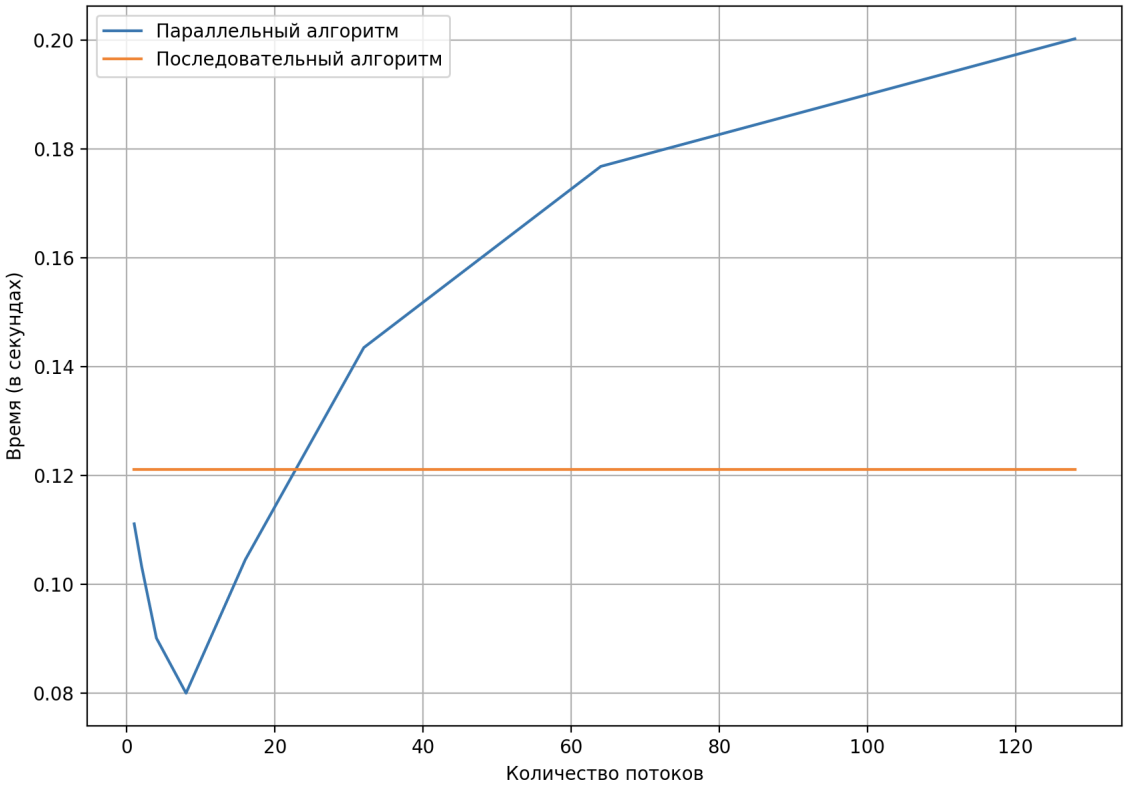


Рисунок 4.2 — Результаты замеров времени реализаций алгоритма битонной сортировки для разного количества потоков (количество элементов в массиве равно 1024)

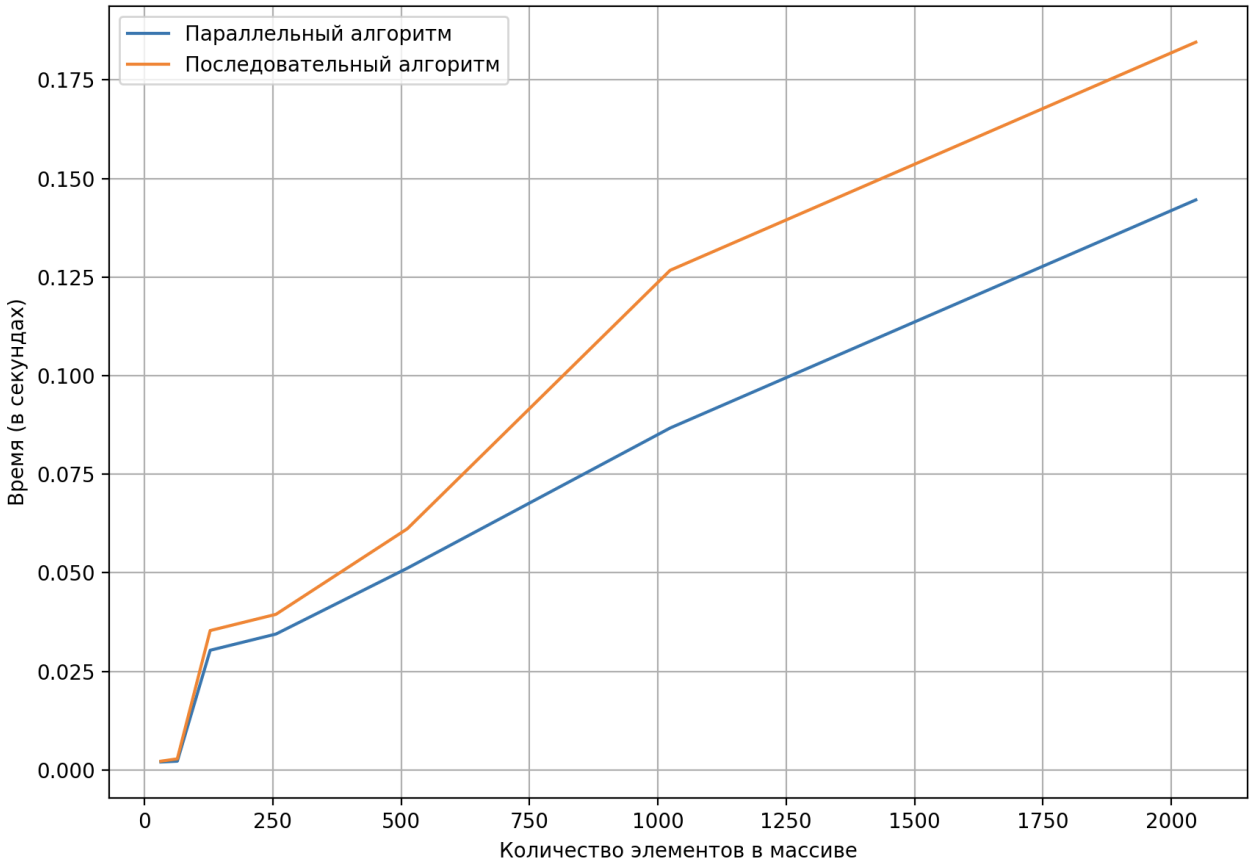


Рисунок 4.3 — Результаты замеров времени реализаций алгоритма битонной сортировки для 1 потока

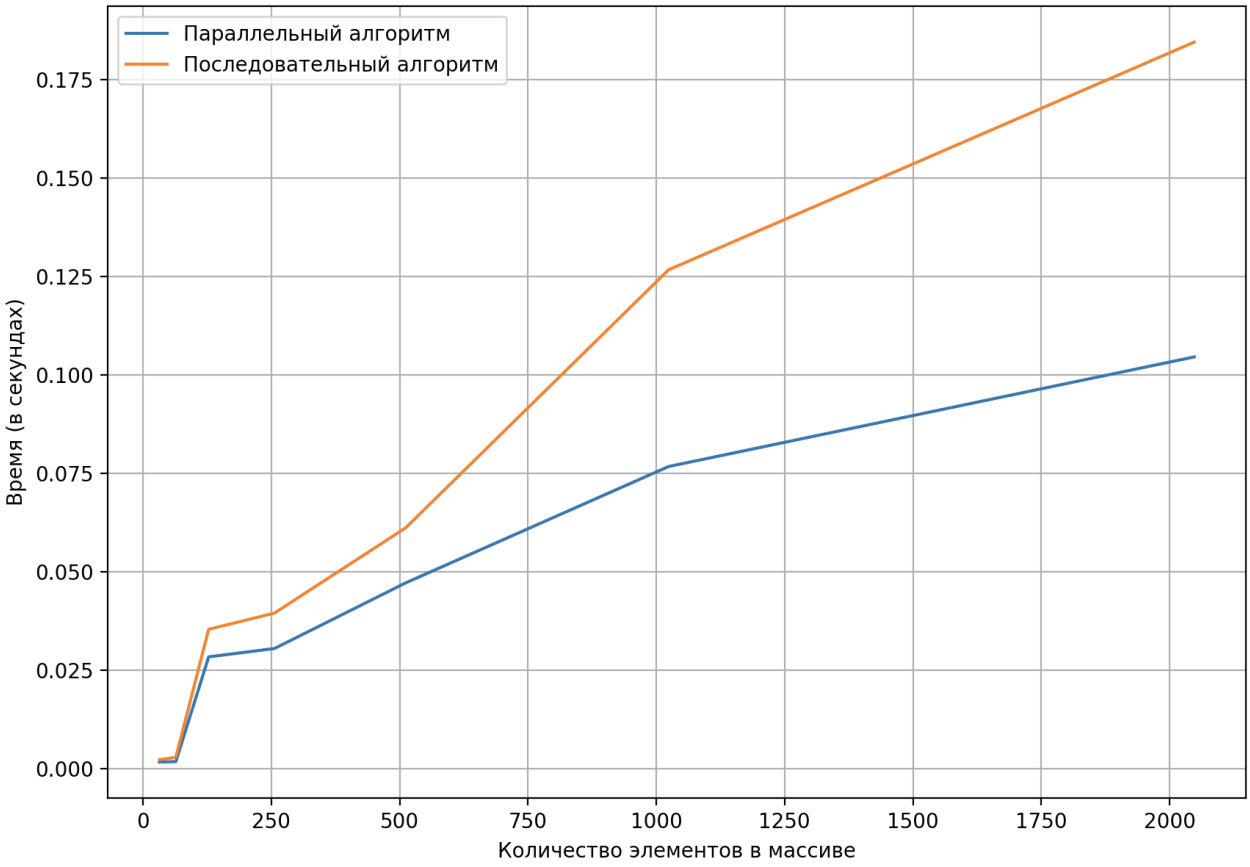


Рисунок 4.4 — Результаты замеров времени реализаций алгоритма битонной сортировки для 8 потоков

**Вывод**

В результате эксперимента можно сделать вывод, что при использовании 8 потоков, многопоточная реализация алгоритма битонной сортировки лучше реализации без многопоточности в среднем на 35% при количестве элементов в массиве равном 1024.

# ЗАКЛЮЧЕНИЕ

В результате выполнения данной лабораторной работы была достигнута цель работы: получены навыки организации параллельных вычислений на базе нативных потоков.

Были решены все задачи:

— изучены основы параллельных вычислений;

— реализован алгоритм битонной сортировки с использованием многопоточности и без нее;

— построены схемы алгоритмов;

— реализованы алгоритмы;

— проведен сравнительный анализ времени работы параллельного и последовательного алгоритмов.

В ходе сравнительного анализа самым эффективным по времени был признан параллельный алгоритм битонной сортировки. Так, при использовании 8 потоков, многопоточная реализация алгоритма битонной сортировки лучше реализации без многопоточности в среднем на 35% при количестве элементов в массиве равном 1024.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. П.М. Будагов. Проверка эффективности в сортировках // Современные технологии: актуальные вопросы, достижения и инновации. сборник статей XVII Международной научно-практической конференции. — 2018. — С. 101–103.

2. Osama Ahmed Abulnaja Muhammad Jawad Ikram. Analyzing Power and Energy Efficiency of Bitonic Mergesort Based on Performance Evaluation. — IEEE, 2017. — 5 с.

3. Документация языка C++ [Электронный ресурс]. — Режим доступа: `https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3690.pdf` (дата обращения: 13.11.2022).

4. Visual Studio Code [Электронный ресурс]. — Режим доступа: `https://code.visualstudio.com/docs` (дата обращения: 20.09.2022).

5. Документация библиотеки matplotlib [Электронный ресурс]. — Режим доступа: `https://matplotlib.org` (дата обращения: 25.09.2022).

6. macOS Monterey [Электронный ресурс]. — Режим доступа: `https://www.apple.com/macos/monterey/` (дата обращения: 30.09.2022).

7. Процессор Intel Core i9 9 поколения [Электронный ресурс]. — Режим доступа: `https://ark.intel.com/content/www/ru/ru/ark/products/97539/intel-core-i57260u-processor-4m-cache-up-to-3-40-ghz.html` (дата обращения: 13.11.2022).