

## 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 закрашка .....	9
1.3.2 Закрашка по Гуро .....	10
1.3.3 Закрашка по Фонгу .....	10
1.4 Алгоритмы моделирования броуновского движения .....	11
1.4.1 Классическое броуновское движение .....	12
1.4.2 Алгоритм срединных смещений .....	12
1.4.3 Фрактальное броуновское движение .....	13
2 Конструкторский раздел .....	15
2.1 Алгоритм срединных смещений .....	15
2.2 Алгоритмы отрисовки .....	16
2.3 Диаграмма классов .....	18
3 Технологический раздел .....	20
3.1 Требования к программному обеспечению .....	20
3.2 Средства реализации .....	20
3.3 Реализация алгоритмов .....	20
3.4 Интерфейс работы программного обеспечения .....	24
4 Исследовательский раздел .....	28
4.1 Цель эксперимента .....	28
4.2 План эксперимента .....	28

4.3 Результаты эксперимента .....	28
Заключение .....	30
Список использованных источников .....	31

## 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 коронавируса. 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 распространения частиц вирусной инфекции.

Цель данной курсовой работы — разработать программу с пользовательским интерфейсом, которая предоставит функционал для моделирования броуновского движения частиц коронавирусной 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 и пол;
- частицы вируса.

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

#### **1.1.2 Выбор формы представления трехмерных объектов**

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. Данный 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.

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.

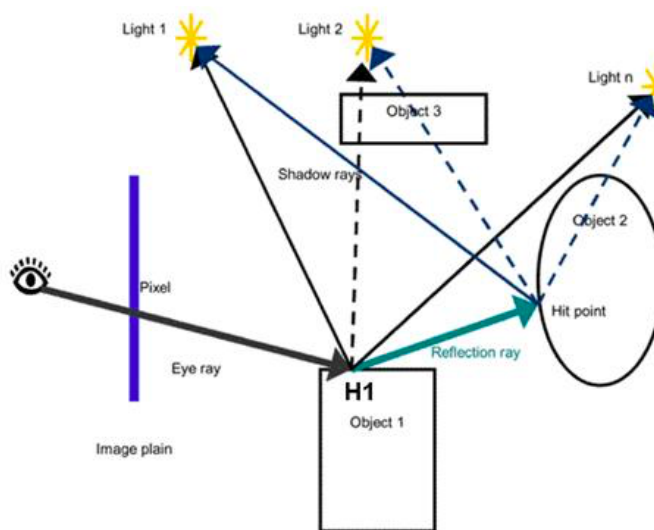


Рисунок 1.1 — Алгоритм обратной трассировки

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.

Преимущества алгоритма:

- возможность использования алгоритма в параллельных вычислительных системах.

Недостатки алгоритма:

- требуется большое количество вычислений;
- производительность алгоритма.

### 1.2.2 Алгоритм Робертса

Работа данного алгоритма проходит в 2 этапа:

- определение нелицевых граней для каждого тела отдельно;
- определение и удаление невидимых ребер [1].

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.

$$ax + by + cz + d = 0 \quad (1.1)$$

В матричной форме 1.1 выглядит как 1.2.

$$[x \ y \ z \ 1][P]^T = 0 \quad (1.2)$$

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.

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

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.

Преимущества алгоритма:

- высокая точность вычислений.

Недостатки алгоритма:

- рост числа трудоемкости алгоритма, как квадрата числа объектов [1];
- работа только с выпуклыми телами.

### 1.2.3 Алгоритм, использующий Z-буфер

В данном алгоритме рассматриваются два буфера. 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  $z(x, y)$  [1].

Преимущества 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;
- большой объем требуемой памяти.

### 1.3 Анализ методов закрашивания

Методы закрашивания используются для затенения полигонов (или поверхностей, 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

При однотонной закраске вычисляют один уровень интенсивности, который используется для закраски всего многоугольника. При этом предполагается, что:

- источник света расположен в бесконечности;
- наблюдатель находится в бесконечности;
- многоугольник представляет реальную моделируемую поверхность, а не является аппроксимацией криволинейной поверхности.

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



Рисунок 1.2 — Пример простой закраски



### 1.3.2 Закраска по Гуро

Данный алгоритм предполагает следующие шаги:

- вычисляются нормали ко всем полигонами;
- определяются нормали в вершинах путем усреднения нормалей по всем полигональным граням, которым принадлежит вершина;
- используя нормали в вершинах и применяя произвольный метод закрашки, вычисляются значения интенсивности в вершинах;
- каждый многоугольник закрашивается путем линейной интерполяции значений интенсивностей в вершинах сначала вдоль каждого ребра, а затем и между ребрами вдоль каждой сканирующей строки.

Достоинства:

- хорошо сочетается с диффузным отражением;
- изображение получается более реалистичным, чем при простой закрашке.

Недостатки:

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

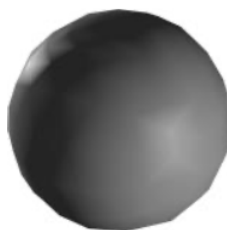


Рисунок 1.3 — Пример закрашки по Гуро

### 1.3.3 Закраска по Фонгу

При такой закрашке, в отличие от закрашки по Гуро, вдоль сканирующей строки интерполируется значение вектора нормали, а не интенсивности.

Шаги алгоритма:

- вычисление векторов нормалей в каждой грани и 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.

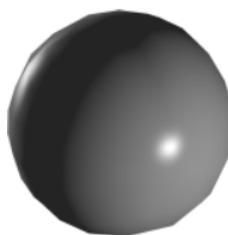


Рисунок 1.4 — Пример закраски по Фонгу

#### 1.4 Алгоритмы моделирования броуновского движения

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

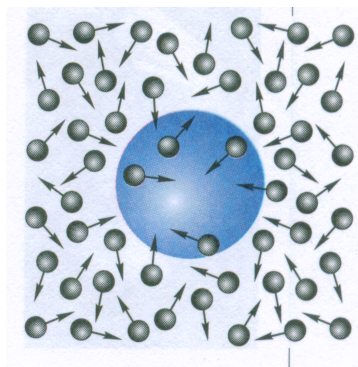


Рисунок 1.5 — Броуновское движение

Рассмотрим случайную величину  $X(t)$ , заданную на отрезке  $[0, T]$ . Случайный процесс  $X(t)$  называется одномерным броуновским движением на интервале  $[0, T]$ , если он обладает следующими 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.

Для моделирования броуновского движения можно воспользоваться различными алгоритмами.

### 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,  $N(0,1)$ ) [2].

```
1: array[N]
2: array[0] ← 0
3: for i = 1,..., N do
4:   array[i + 1] ← array[i] + randomNormal(0,1)
5: end for
```

### 1.4.2 Алгоритм срединных смещений

Метод случайного срединного смещения 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 [2].

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 выполняться закон дисперсии для приращения в вычисленных точках. Параметр  $\sigma$  определяет масштаб 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  $\{X(\frac{k}{2^N})\}_{k=0}^{2^N}$  (реализация броуновского движения  $X(t)$  на дискретном множестве точек вида  $t_k = \frac{k}{2^N}$ ,  $k \in [0, 2^N]$ ).

```
1: X(0) ← 0
2: X(1) ← σg // g - случайная величина, распределенная нормально с параметрами N(0,1)
3: for j = 1,..., N do
4:   for i = 1,..., 2N-1 do
5:     X((2i - 1)2N-j) ← X((i - 1)2N-j+1) + X(i2N-j+1) +  $\frac{1}{2^{(j+1)/2}}$ σ g
6:   end for
7: end for
```

### 1.4.3 Фрактальное броуновское движение

Фрактальное броуновское движение (ФБД) уже не является марковским процессом, а обладает некой "памятью". Для аппроксимации ФБД нет простого метода, вроде суммирования нормальных случайных величин, как в случае классического броуновского движения. Для аппроксимации ФБД наиболее удобно использовать преобразования Фурье.

Рассмотрим 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 следующими свойствами:

- $X(0) = 0$  почти наверное и  $X(t)$  - почти наверное непрерывная функция на  $[0, T]$ ;
- $X(t)$  - процесс с приращениями, распределенными нормально.

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

$$S(f) \propto \frac{1}{f^{2H+1}} \quad (1.4)$$

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 получается ФБД во временной области с помощью обратного преобразования Фурье.

Необходимо смоделировать дискретный аналог ФБД, то есть цель — получить величины  $\{X_n\}_{n=0}^{N-1}$ , аппроксимирующие ФБД в точках  $n$ . Для этого необходимо воспользоваться формулой дискретного преобразования Фурье

$$\hat{X}_n = \sum_{k=0}^{N-1} X_k e^{-2\pi kn/N} \quad (1.5)$$

и обратного дискретного преобразования Фурье

$$X_n = \sum_{k=0}^{N-1} \hat{X}_k e^{2\pi kn/N} \quad (1.6)$$

Далее будут рассмотрены только четные значения  $N$ , а для применения метода быстрого дискретного преобразования Фурье нужно, чтобы  $N = 2^M$ ,  $M \in \mathbb{N}$ .

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 аппроксимация ФБД в последовательные моменты времени  $n$ ).

1:  $\hat{X}_0 \leftarrow g$

```

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

```

## 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 позволяет 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 закрашка по Гуро, так как алгоритм достаточно быстр и хорошо сочетается с выбранным ранее z-буфером.

Для моделирования броуновского движения был выбран алгоритм срединных смещений, так как он позволяет реалистично изобразить броуновское движение, а также легко обобщается для случая  $n$ -мерных движений.

## 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.

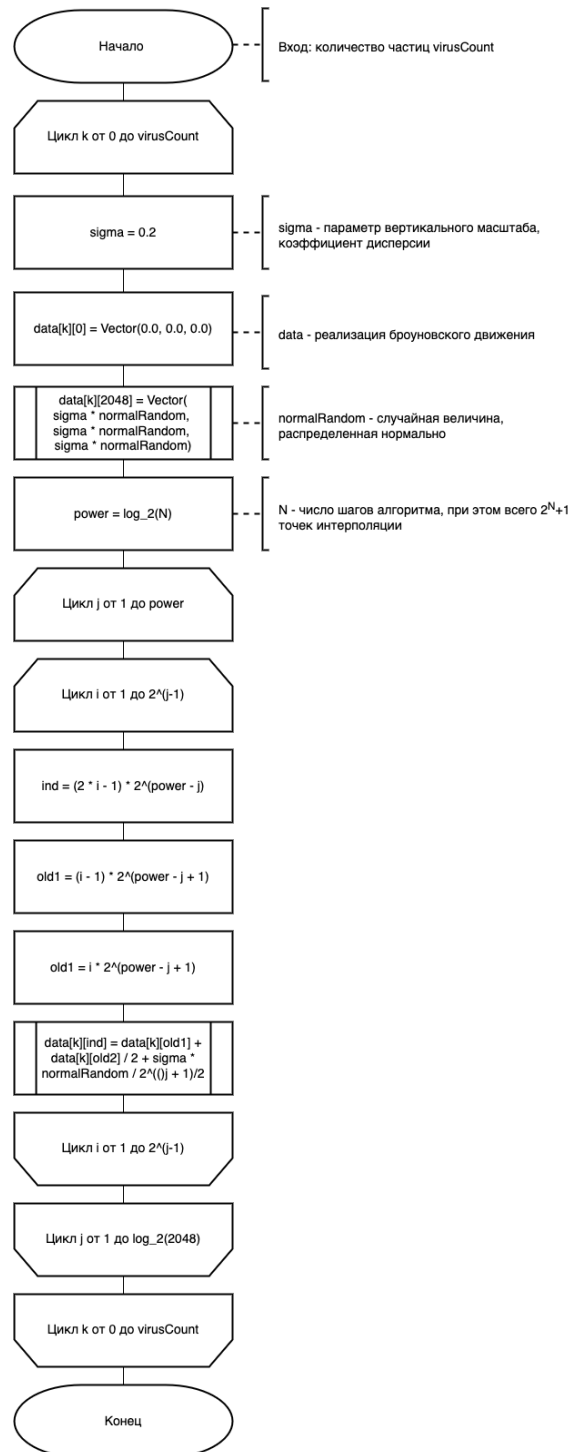


Рисунок 2.1 — Схема алгоритма срединных смещений

## 2.2 Алгоритмы отрисовки

Схема алгоритма Z-буфера изображена на рисунке 2.2.

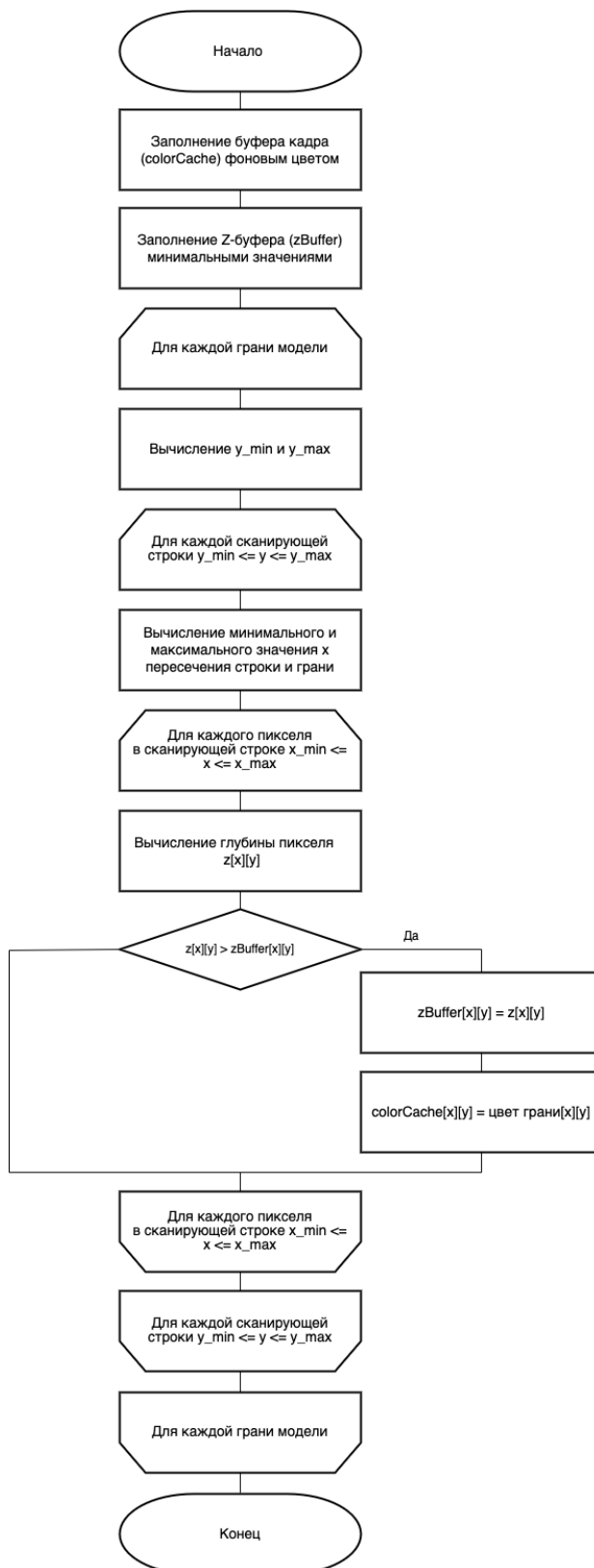


Рисунок 2.2 — Схема алгоритма Z-буфер

Схема алгоритма закрашки по Гуро изображена на рисунке 2.3.



Рисунок 2.3 — Схема алгоритма закрашки по Гуро



## 2.3 Диаграмма классов

На рисунке 2.4 представлена диаграмма классов.

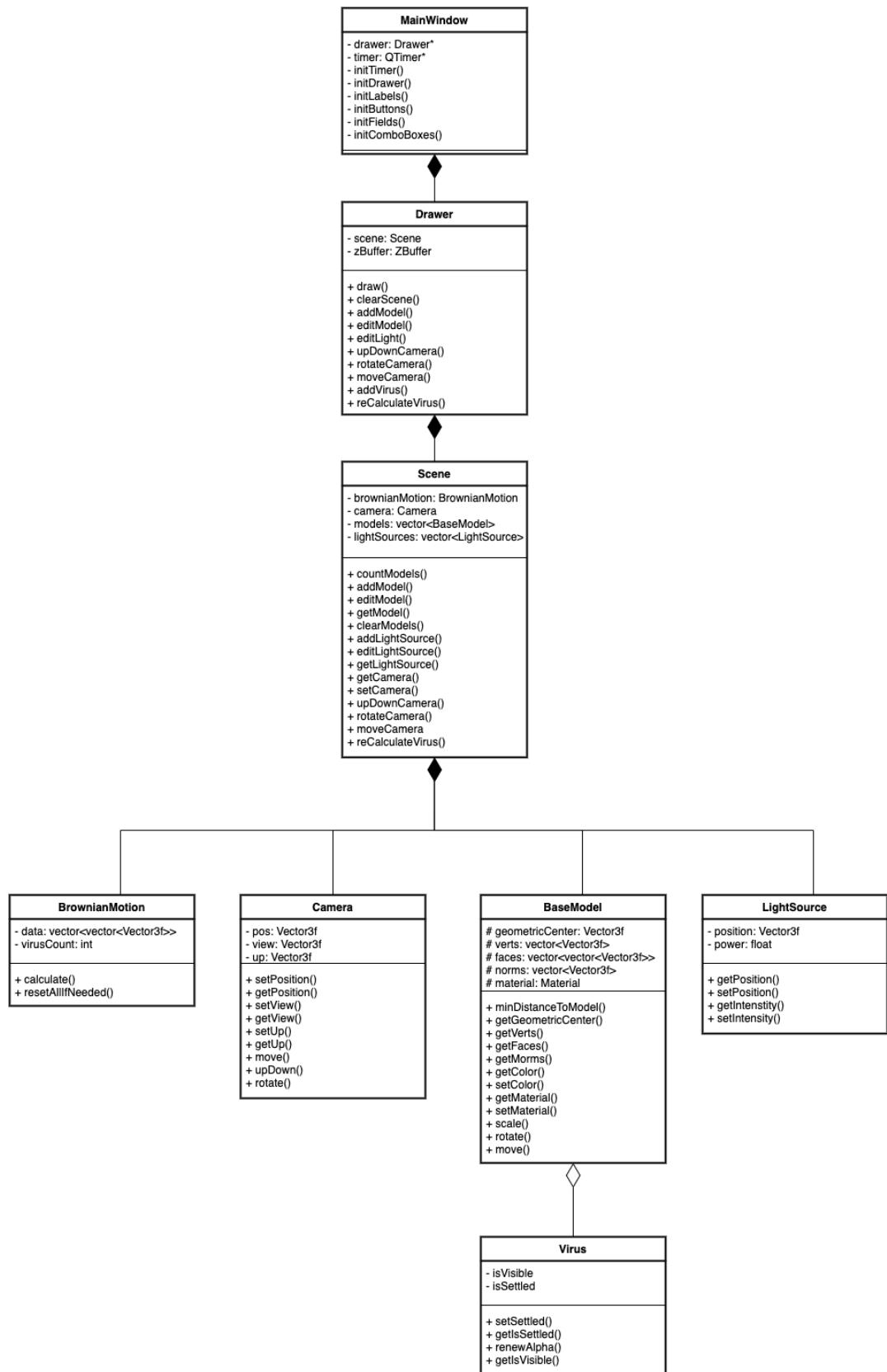


Рисунок 2.4 — Диаграмма классов

## **Вывод**

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

### 3 Технологический раздел

В данном разделе будут описаны средства реализации программного обеспечения, требования к нему, 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], так как он содержит возможности для создания программ с графическим интерфейсом. В качестве среды разработки использовался Qt Creator [4].

#### 3.3 Реализация алгоритмов

В листингах 3.1 – 3.2 представлена реализация алгоритмов моделирования броуновского движения и отрисовки сцены.

Листинг 3.1 — Реализация вспомогательной функции вычисления распределенной нормально случайной величины

```
1 float BrownianMotion::getNormalRandom()
2 {
3     static default_random_engine e;
4     static uniform_real_distribution<> dis(-step, step);
5     return dis(e);
6 }
```

Листинг 3.2 — Реализация вычисления изменения координат центров частиц (броуновское движение)

```

1  void BrownianMotion::setVirusCountAndResetAllIfNeeded(int newVirusCount)
2  {
3      if (virusCount == newVirusCount)
4      {
5          return;
6      }
7      virusCount = newVirusCount;
8      float sigma = 0.2;
9      data = std::vector<std::vector<Vector3f>>(virusCount,
10         std::vector<Vector3f>(SIZE + 1));
11  for (int k = 0; k < virusCount; k++)
12  {
13      data.at(k).at(0) = Vector3f(0.0, 0.0, 0.0);
14      data.at(k).at(SIZE) = Vector3f(sigma * getNormalRandom(), sigma *
15         getNormalRandom(), sigma * getNormalRandom());
16      for (int j = 1; j <= POWER; j++)
17      {
18          for (int i = 1; i <= pow(2, (j - 1)); i++)
19          {
20              int ind = (2 * i - 1) * pow(2, POWER - j);
21              int old1 = (i - 1) * pow(2, POWER - j + 1);
22              int old2 = i * pow(2, POWER - j + 1);
23              data.at(k).at(ind).x = (data.at(k).at(old1).x +
24                 data.at(k).at(old2).x) / 2 + sigma * getNormalRandom() /
25                 pow(2, (j + 1) / 2);
26              data.at(k).at(ind).y = (data.at(k).at(old1).y +
27                 data.at(k).at(old2).y) / 2 + sigma * getNormalRandom() /
28                 pow(2, (j + 1) / 2);
29              data.at(k).at(ind).z = (data.at(k).at(old1).z +
30                 data.at(k).at(old2).z) / 2 + sigma * getNormalRandom() /
31                 pow(2, (j + 1) / 2);
32          }
33      }
34  }
35  }
```

В листинге 3.3 представлена реализация вычисления интенсивности вершины от источников света на сцене.

Листинг 3.3 — Реализация вычисления интенсивности вершины

```

1  float Drawer::processLight(const Vector3f& vert, const Vector3f& norm)
2  {
3      float wholeIntensity = 0;
4      float intensity;
```

```

5
6     size_t lights = scene.getLightSourceCount();
7
8     for (size_t i = 0; i < lights; i++)
9     {
10         intensity = 0;
11         LightSourcePoint lsp = scene.getLightSource(i);
12
13         Vector3f lightDir = vert - lsp.getPosition();
14
15         intensity += lightDir * norm / pow(lightDir.norm(), 2.0);
16         intensity *= lsp.getIntensity() * LIGHT_REFLECT;
17
18         intensity = fmax(0.0, intensity);
19         intensity = fmin(1.0, intensity);
20
21         intensity = BG_LIGHT + intensity * (1 - BG_LIGHT);
22
23         wholeIntensity += intensity;
24     }
25
26     if (wholeIntensity == 0)
27         wholeIntensity = BG_LIGHT;
28     else
29         wholeIntensity /= lights;
30
31     return wholeIntensity;
32 }

```

В листинге 3.4 представлен метод, который осуществляет вычисление глубины каждой точки полигона и её интенсивности.

Листинг 3.4 — Реализация вычисления интенсивности вершины

```

1 void Drawer::processPoligon(Vector3i& t0, Vector3i& t1, Vector3i& t2,
2 const QColor& color, float& i0, float& i1, float& i2,
3 float modelAlpha)
4 {
5     if (t0.y == t1.y && t0.y == t2.y)
6         return;
7
8     if (t0.y > t1.y)
9     {
10         std::swap(t0, t1);
11         std::swap(i0, i1);
12     }
13     if (t0.y > t2.y)
14     {

```

```

15         std::swap(t0, t2);
16         std::swap(i0, i2);
17     }
18     if (t1.y > t2.y)
19     {
20         std::swap(t1, t2);
21         std::swap(i1, i2);
22     }
23
24     int total_height = t2.y - t0.y;
25
26     for (int i = 0; i < total_height; i++)
27     {
28         bool second_half = i > t1.y - t0.y || t1.y == t0.y;
29         int segment_height = second_half ? t2.y - t1.y : t1.y - t0.y;
30
31         float alpha = (float)i / total_height;
32         float betta = (float)(i - (second_half ? t1.y - t0.y : 0)) /
            segment_height;
33
34         Vector3i A = t0 + Vector3f(t2 - t0) * alpha;
35         Vector3i B = second_half ? t1 + Vector3f(t2 - t1) * betta : t0 +
            Vector3f(t1 - t0) * betta;
36
37         float iA = i0 + (i2 - i0) * alpha;
38         float iB = second_half ? i1 + (i2 - i1) * betta : i0 + (i1 - i0) *
            betta;
39
40         if (A.x > B.x)
41         {
42             std::swap(A, B);
43             std::swap(iA, iB);
44         }
45
46         A.x = std::max(A.x, 0);
47         B.x = std::min(B.x, w);
48
49         for (int j = A.x; j <= B.x; j++)
50         {
51             float phi = B.x == A.x ? 1. : (float)(j - A.x) / (float)(B.x -
                A.x);
52
53             Vector3i P = Vector3f(A) + Vector3f(B - A) * phi;
54             float iP = iA + (iB - iA) * phi;
55             if (P.x >= w || P.y >= h || P.x < 0 || P.y < 0) continue;
56
57             if (zBuffer.getDepth(P.x, P.y) < P.z)

```

```

58      {
59          zBuffer.setDepth(P.x, P.y, P.z);
60          QColor newColor = QColor(iColor(color.rgb(), iP));
61          if (fabs(modelAlpha - 1.0) > EPS)
62          {
63              newColor = calculateNewColor(newColor,
64                                             colorCache[P.x][P.y], modelAlpha);
65          }
66          colorCache[P.x][P.y] = newColor;
67      }
68  }
69  }

```

### 3.4 Интерфейс работы программного обеспечения

Окно управления представлено на рисунке 3.1. На рисунке 3.2 изображена сцена без частиц вируса. На рисунке 3.3 изображено движение частиц. На рисунке 3.4 видно, как частицы осели и начали тускнеть.

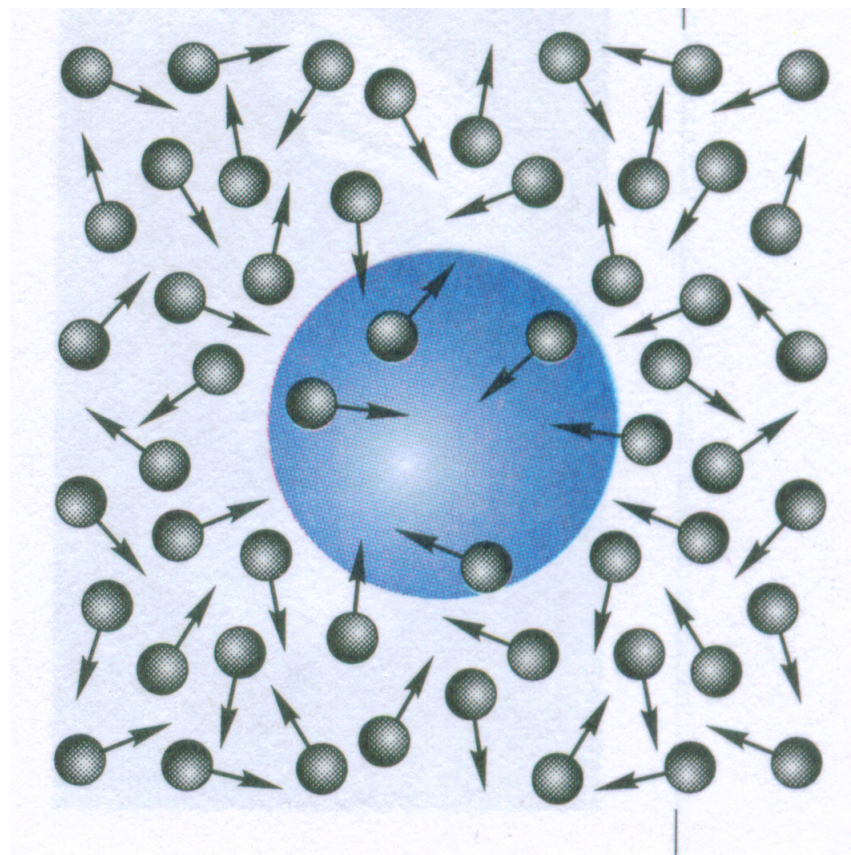


Рисунок 3.1 — Окно управления



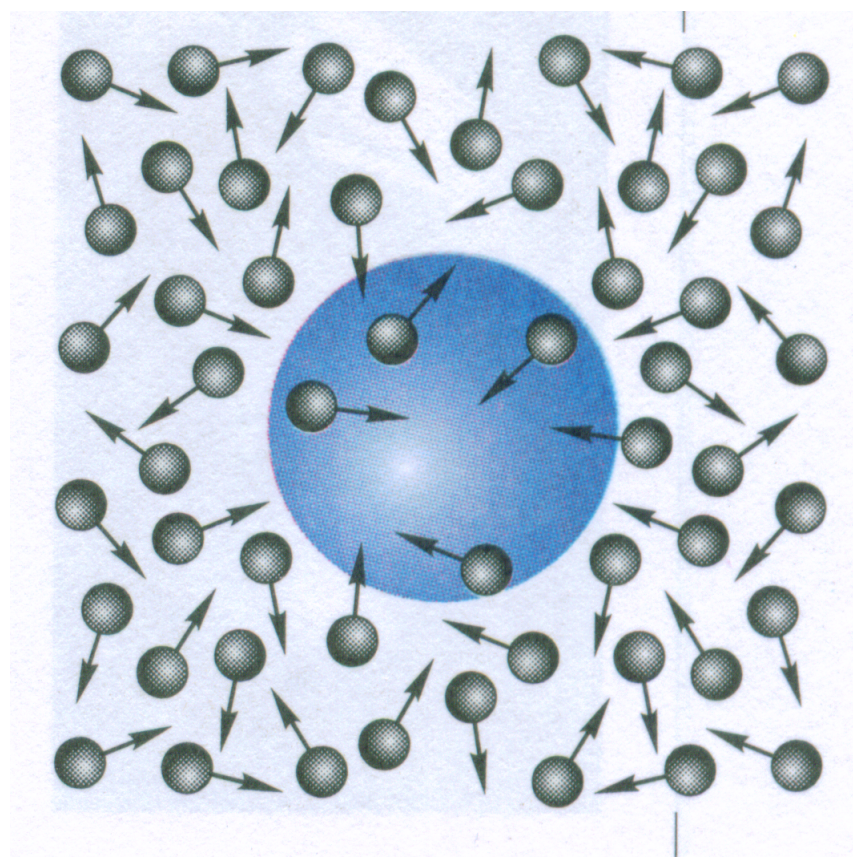


Рисунок 3.2 — Отображение сцены без частиц вируса

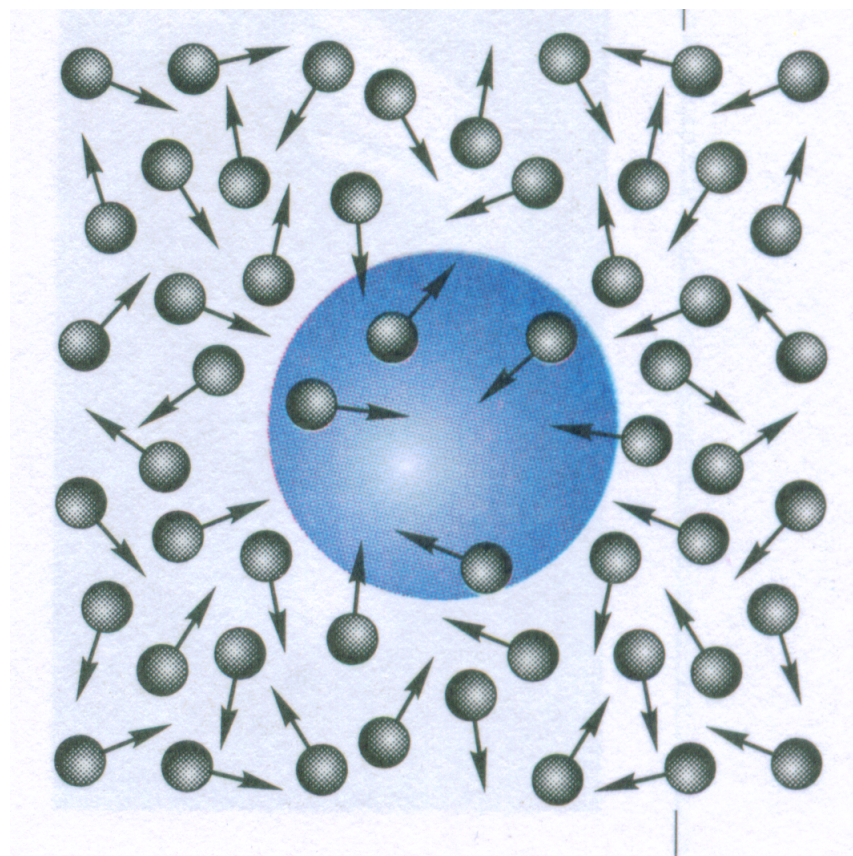


Рисунок 3.3 — Пример движения частиц



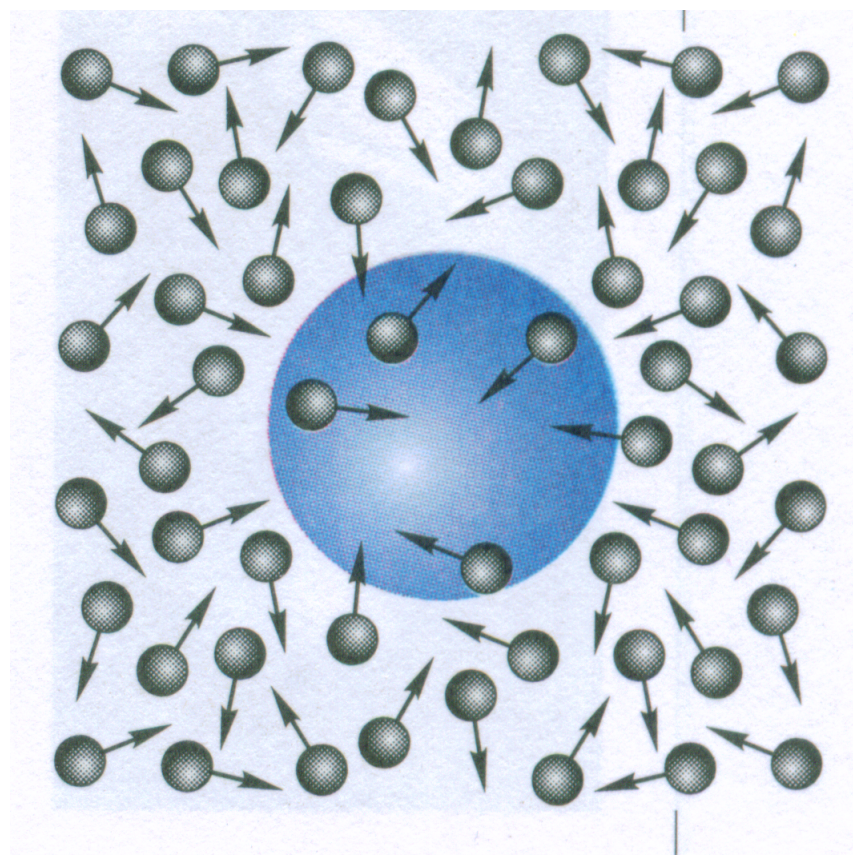


Рисунок 3.4 — Пример оседания и тускнения частиц

## **Вывод**

В данном разделе были перечислены требования к программному обеспечению, средства разработки, с помощью которых была реализована программа, приведены листинги реализаций алгоритмов вычисления необходимых параметров для отрисовки сцены, а также алгоритма моделирования броуновского движения.

## 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 [5];
- оперативная память 16 Гб;
- процессор 2,3 ГГц 8-ядерный Intel Core i9 9 поколения [6].

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

### 4.1 Цель эксперимента

### 4.2 План эксперимента

### 4.3 Результаты эксперимента

Процессорное время замерялось при помощи функции `std::chrono::system_clock::now()` из заголовочного файла `chrono` [3]. Результаты сформированы в виде графиков при помощи библиотеки `matplotlib` [7].

Результаты замеров представлены в таблице 4.1.

Таблица 4.1 — Результаты замеров

Количество частиц вируса	Время отрисовки сцены (в миллисекундах)
1	235.0
10	255.0
20	265.0
40	295.0
60	330.0
100	390.0
150	460.0
200	550.0
250	620.0
300	680.0
400	830.0
500	1000.0
600	1130.0

На рисунках 4.1 на осях X указано количество частиц. Время на осях Y указано в миллисекундах.

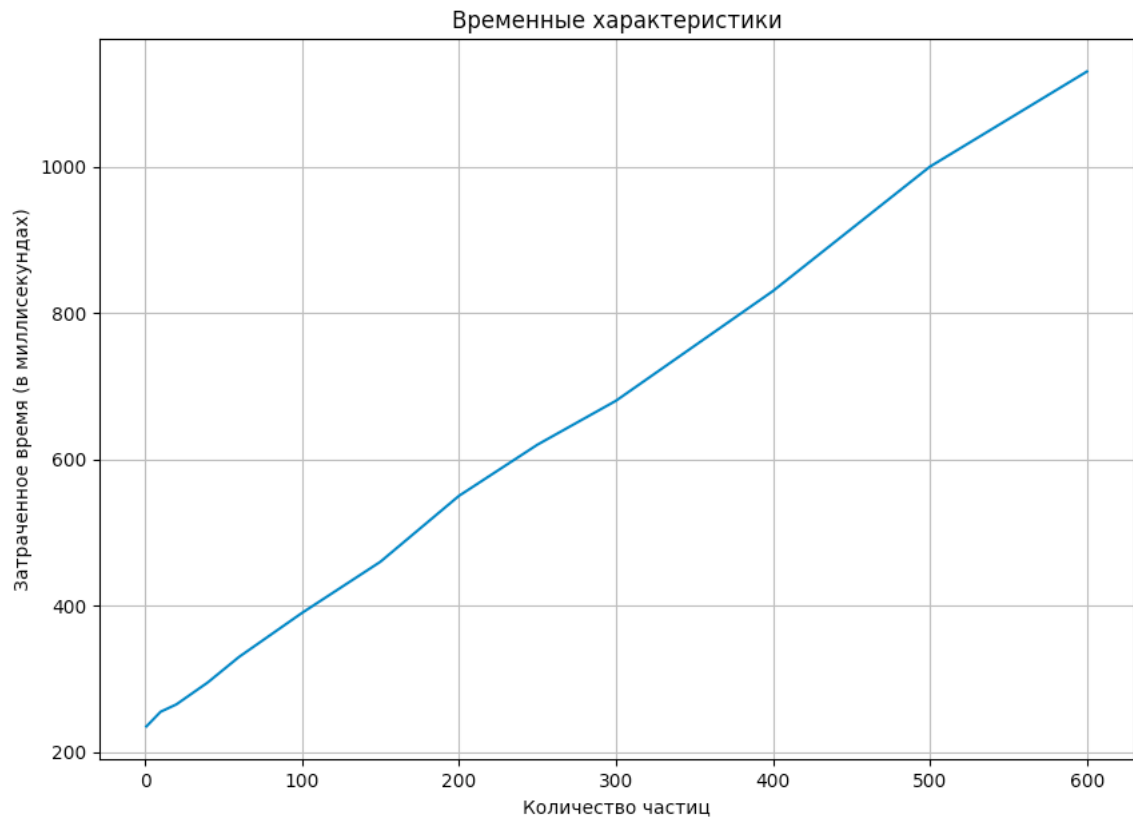


Рисунок 4.1 — Результаты замеров времени реализаций алгоритма битонной сортировки для разного количества потоков (количество элементов в массиве равно 1024)

## Вывод

В результате эксперимента можно сделать вывод, что увеличение количества частиц вируса значительно влияет на скорость отрисовки изображения. Причем время, затрачиваемое на отрисовку сцены, линейно зависит от количества частиц.

## ЗАКЛЮЧЕНИЕ

В результате выполнения курсовой работы была разработана программа с пользовательским интерфейсом, которая предоставляет функционал для моделирования броуновского движения частиц коронавирусной инфекции в помещении с учетом скорости их распространения и времени жизни на разных поверхностях. Были решены следующие задачи:

- рассмотрены алгоритмы удаления невидимых линий и поверхностей и методы закраски;
- проанализированы алгоритмы моделирования броуновского движения;
- выбраны и реализованы подходящие для решения поставленной задачи алгоритмы;
- формализована модель, представлена диаграмма классов;
- проведен анализ производительности программного обеспечения.

В ходе замеров было выяснено, что скорость отрисовки сцены линейно зависит от количества частиц вируса.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. А.Ю. Демин. Основы компьютерной графики: учебное пособие. — Изд-во Томского политехнического университета, 2011. — 191 с.
2. П.И. Трошин. Моделирование фракталов в среде Maxima. — Изд-во Казанского федерального университета, 2012. — 48 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. Qt Creator [Электронный ресурс]. — Режим доступа: <https://www.qt.io/product/development-tools> (дата обращения: 26.11.2022).
5. macOS Monterey [Электронный ресурс]. — Режим доступа: <https://www.apple.com/macos/monterey/> (дата обращения: 30.09.2022).
6. 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).
7. Документация библиотеки matplotlib [Электронный ресурс]. — Режим доступа: <https://matplotlib.org> (дата обращения: 25.09.2022).