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»

КУРСОВАЯ РАБОТА

НА ТЕМУ:

«Генерация ландшафта»

Студент ИУ7-51Б (группа) _____ (подпись, дата) Н. А. Баранов (И.О. Фамилия)

Руководитель курсового проекта _____ (подпись, дата) А. С. Кострицкий (И.О. Фамилия)

2024 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Цель курсовой работы – разработать программу для генерации ландшафта и его визуализации. В качестве параметров генерации принять значения высот в точках, размер ландшафта и шаг задания точек.

Для достижения поставленной цели необходимо решить следующие задачи:

1) формализовать задачу;
2) проанализировать и выбрать алгоритмы для создания изображения и генерации ландшафта;
3) спроектировать программу для генерации ландшафта и его визуализации;
4) выбрать средства реализации и реализовать спроектированную программу;
5) исследовать зависимость скорости генерации ландшафта от параметров генерации.

1 Аналитическая часть

В данном разделе приведено описание объектов сцены, а также анализ существующих алгоритмов для решения поставленных задач, в результате которого выбраны наиболее подходящие из них.

1.1 Формализация задачи

На рисунке 1.1 представлена формализованная задача.

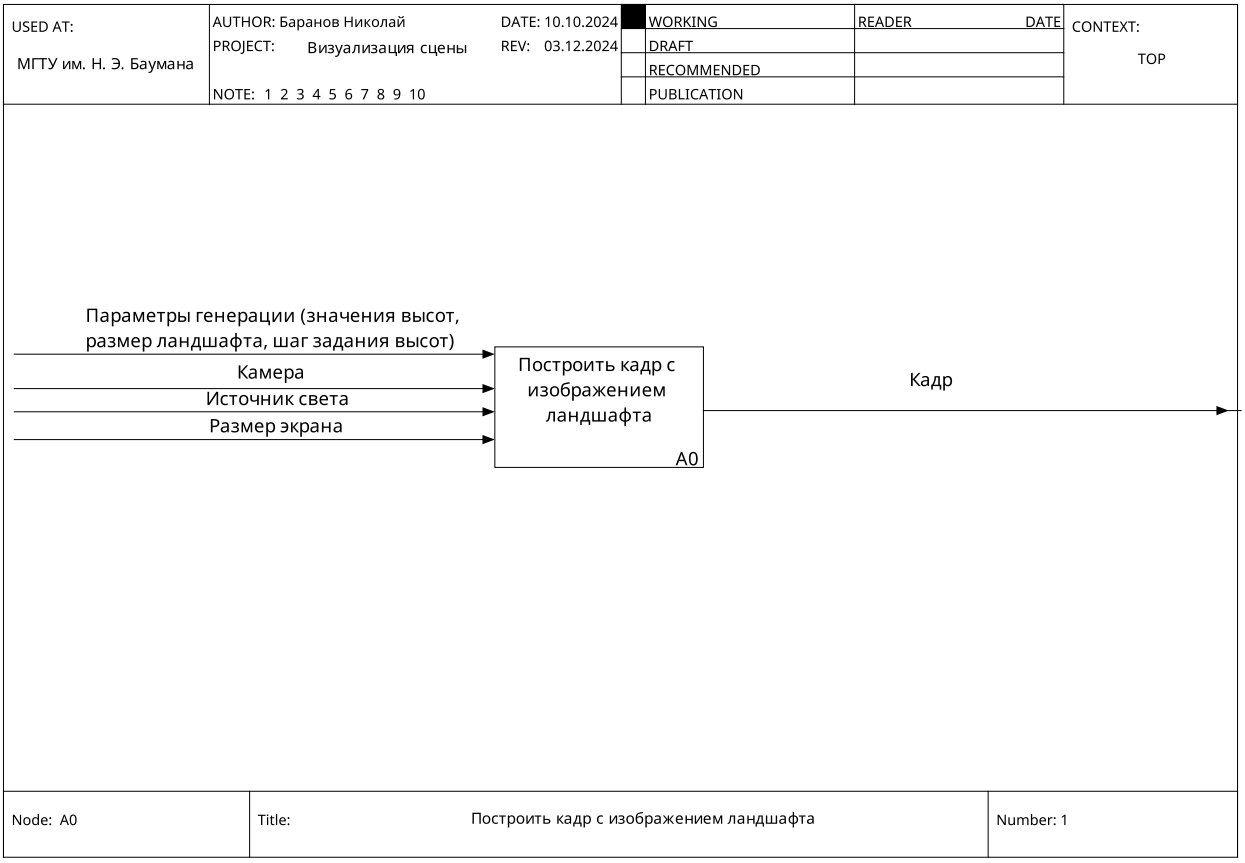

Рисунок 1.1 – Формализованная задача

1.2 Анализ объектов сцены

Сцена состоит из следующих объектов:

— источник света, расположенный на бесконечности – задаётся 2 углами;

— камера – задаётся положением в пространстве, фокусным расстоянием, дальностью видимости, а также 3 векторами, задающими систему координат камеры;

— ландшафт – задаётся регулярной картой высот, так как параметрами генерации являются значения высот в точках и шаг задания точек, и множеством полигонов.

1.3 Анализ алгоритмов генерации ландшафта

Существуют следующие алгоритмы генерации ландшафта [1][2][3][4][5][6]:

— шум Перлина;

— шум Симплекс;

— дробный шум Брауна;

— шум Value Noise;

— fault алгоритм;

— холмовой алгоритм;

— алгоритм diamond-square;

— алгоритм, использующий билинейную интерполяцию;

— алгоритм, использующий кригинг.

1.3.1 Алгоритмы, основанные на шумовых функциях

Алгоритмы, основанные на шумовых функциях, заполняют значения на карте высот результатами работы шумовых функций [2]. Однако шумовые функции никак не учитывают заранее заданные на ландшафте значения высот, поэтому не подходят для генерации ландшафта.

1.3.2 Fault алгоритм

Fault алгоритм заключается в выполнении следующих шагов в несколько итераций [3]:

1) разделение плоскости прямой на 2 части;

2) повышение значений высот в одной части и понижение в другой.

Данный алгоритм не подходит для генерации ландшафта с заранее заданными значениями высот.

1.3.3 Холмовой алгоритм

Холмовой алгоритм заключается в создании холмов на определённой области [4]. При этом он не предусматривает наличие заранее заданных значений высот, из-за чего не подходит для генерации ландшафта.

1.3.4 Алгоритм diamond-square

Алгоритм diamond-square представляет собой расширенную версию алгоритма midpoint displacement, заключающуюся в использовании двумерной плоскости и наличии 2 этапов [7]:

1) этап «square» – определение центральных точек для квадратов и присваивание им среднего значения из вершин со случайным смещением;

2) этап «diamond» – определение центральных точек для ромбов и присваивание им среднего значения из вершин со случайным смещением. Крайние случаи необходимо рассматривать отдельно, так как часть вершин может отсутствовать.

На выходе у него получается регулярная карта высот. Таким образом, выходит, что вся плоскость покрыта квадратами. Порядок генерации точек можно увидеть на рисунке 1.2.

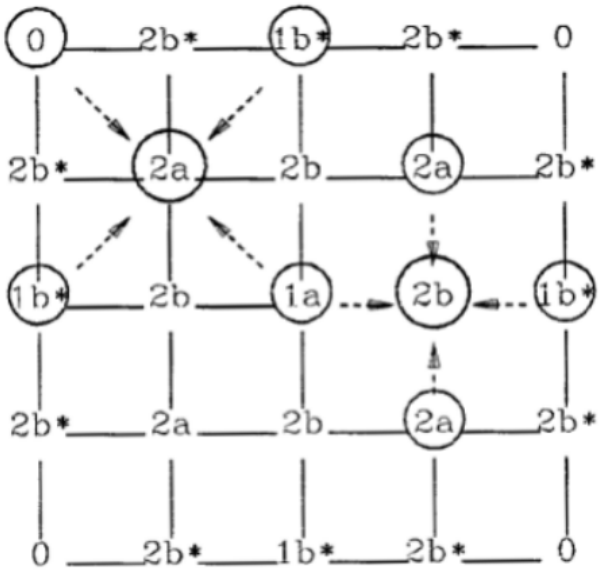

Рисунок 1.2 – Порядок генерации высот в алгоритме diamond-square [8]

Данный алгоритм хорошо подходит для генерации гор [1]. При этом у данного алгоритма есть сложности с генерацией значений на границах квадрата, так как на шаге square не хватает одной точки. Также сторона квадрата должна иметь размер $2^n + 1$.

1.3.5 Алгоритм, использующий билинейную интерполяцию

Алгоритм заключается в заполнении значений на карте высот результатом работы функции билинейной интерполяции $f(x, y)$ [5]. Значение $f(x, y)$ рассчитывается по формуле 1.1, где $f_1(x, y)$ рассчитывается по формуле 1.2, а $f_2(x, y)$ – по формуле 1.3, причём точка (x, y) находится внутри прямоугольника с вершинами в точках (x_1, y_1), (x_2, y_1), (x_1, y_2), (x_2, y_2), значения в которых равны h_{11}, h_{21}, h_{12} и h_{22} соответственно.

$$f(x, y) = f_1(x, y) + f_2(x, y) \tag{1.1}$$

$$f_1(x, y) = \frac{h_{11} \cdot (x_2 - x) \cdot (y_2 - y)}{(x_2 - x_1) \cdot (y_2 - y_1)} + \frac{h_{21} \cdot (x - x_1) \cdot (y_2 - y)}{(x_2 - x_1) \cdot (y_2 - y_1)} \tag{1.2}$$

$$f_2(x, y) = \frac{h_{12} \cdot (x_2 - x) \cdot (y - y_1)}{(x_2 - x_1) \cdot (y_2 - y_1)} + \frac{h_{22} \cdot (x - x_1) \cdot (y - y_1)}{(x_2 - x_1) \cdot (y_2 - y_1)} \tag{1.3}$$

1.3.6 Алгоритм, использующий кригинг

Цель кригинга – найти такую модель данных, среднее квадратичное значение ошибки которой будет минимальным [6]. Оценка значения в точке x_0 $z_k(x_0)$ рассчитывается по формуле 1.4. В ней n – число точек, от которых зависит значение в точке x_0, $z(x_i)$ – значения в этих точках, а λ_i – вес значения.

$$z_k(x_0) = \sum_{i=1}^{n} \lambda_i \cdot z(x_i) \tag{1.4}$$

Алгоритм состоит из следующих шагов:

1) составление ковариационной матрицы C расстояний между всеми известными точками. Значение в строке i столбца j C_{ij} рассчитывается по формуле 1.5. В ней h – расстояние между точками i и j, a – радиус вариограммы;

$$C_{ij} = Cov(h) = 1 - \exp^{-\frac{h}{a}} \tag{1.5}$$

2) решение системы уравнений 1.6 для каждой точки x_0, после чего вычисление значения по формуле 1.4.

$$\begin{cases} \lambda_1 \cdot C_{11} + \lambda_2 \cdot C_{12} + \ldots + \lambda_n \cdot C_{1n} - C_{10} = 0 \\ \lambda_1 \cdot C_{21} + \lambda_2 \cdot C_{22} + \ldots + \lambda_n \cdot C_{2n} - C_{20} = 0 \\ \ldots \\ \lambda_1 \cdot C_{n1} + \lambda_2 \cdot C_{n2} + \ldots + \lambda_n \cdot C_{nn} - C_{n0} = 0 \end{cases} \tag{1.6}$$

1.3.7 Сравнение алгоритмов

Из всех рассмотренных алгоритмов только алгоритм diamond-square, алгоритм, использующий билинейную интерполяцию, а также алгорит, использующий кригинг, подходят для генерации ландшафта с изначально заданными

точками. В таблице 1.1 представлено сравнение всех подходящих алгоритмов генерации ландшафта.

Таблица 1.1 – Сравнение алгоритмов генерации ландшафта

Критерий сравнения	Алгоритм diamond-square	Алгоритм, использующий билинейную интерполяцию	Алгоритм, использующий кригинг
Нужен генератор случайных чисел	Да	Нет	Нет
Проблемы с генерацией значений на границах	Да	Нет	Нет
Значение может зависеть от всех точек	Не все	Нет	Да

В результате сравнения был выбран алгоритм, использующий кригинг, так как ему не нужны дополнительные параметры генерации, а также при генерации значения в одной точке он может учесть все заданные значения высот.

1.4 Анализ алгоритмов удаления невидимых линий и поверхностей

Для получения изображения необходимо решить задачу удаления невидимых линий и поверхностей, а также выбрать метод закраски полигонов. Существуют следующие алгоритмы удаления невидимых линий и поверхностей [9][10][11]:

— алгоритм Робертса;

— алгоритм плавающего горизонта;

— алгоритм Варнока;

— алгоритм Вейлера-Азертона;

— алгоритм, использующий Z-буфер;

— алгоритм, использующий список приоритетов;

— алгоритм обратной трассировки лучей.

1.4.1 Алгоритм Робертса

Данный алгоритм работает в объектном пространстве [12]. Алгоритм состоит из следующих шагов:

1) удаление нелицевых граней, экранируемых самим телом;

2) удаление рёбер, экранируемых другими телами;

3) вычисление новых рёбер, полученных при «протыкании» друг друга.

Время работы алгоритма $O(N^2)$, где N – число рёбер.

Ландшафт не является выпуклым телом, поэтому для данного алгоритма его придётся изначально разделить на выпуклые тела.

1.4.2 Алгоритм плавающего горизонта

Алгоритм плавающего горизонта используется при представлении поверхности в виде функции $F(x, y, z) = 0$ [10]. Алгоритм состоит из следующих шагов:

1) пересечение поверхности параллельными плоскостями с постоянной координатой z (ось y направлена вверх);

2) построение кривой для каждой такой плоскости, начиная с ближайшей, лежащей на ней. Если же при заданном значении x значение y больше других найденных, то кривая видима в этой точке.

1.4.3 Алгоритм Варнока

Алгоритм Варнока состоит из следующих шагов [11]:

1) рассмотрение окна и решение вопроса о том, пусто ли оно или является его содержимое достаточно простым для визуализации;

2) если это не так, то окно разбивается на фрагменты прямыми, параллельными координатным осям, и идёт возврат к шагу 1. Иначе рисуется изображение.

Время работы алгоритма $O(CN)$, где C – количество пикселей в окне, а N – число полигонов на сцене. В худшем случае каждый пиксель будет рассматриваться как окно.

1.4.4 Алгоритм Вейлера-Азертона

Алгоритм Вейлера-Азертона является улучшенной версией алгоритма Варнока [11]. При разбиении окна прямые выбираются не параллельные координатным осям, а параллельные проекциям рёбер.

1.4.5 Алгоритм, использующий Z-буфер

Идея алгоритма состоит в том, чтобы для каждого пикселя дополнительно хранить ещё и координату Z или глубину [13]. Алгоритм состоит из следующих шагов:

1) заполнение Z-буфера значением $-\infty$;

2) при занесении очередного пикселя в буфер кадра значение его Z-координаты сравнивается с Z-координатой пикселя, который уже находится в буфере. Если Z-координата нового пикселя больше, чем координата старого, т. е. он ближе к наблюдателю, то атрибуты нового пикселя и его Z-координата заносятся в буфер, если нет, то ни чего не делается.

Одним из главных минусов является высокое потребление памяти за счёт создания Z-буфера.

1.4.6 Алгоритм, использующий список приоритетов

Данный алгоритм иногда называют алгоритмом художника [9]. Алгоритм состоит из следующих шагов:

1) сортировка полигонов по глубине или приоритету;

2) отрисовка полигонов, начиная с наиболее удалённых от точки наблюдения.

Более близкие к наблюдателю элементы будут «затирать» информацию о более далёких элементах в буфере кадра. Эффекты прозрачности можно включить в состав алгоритма путём не полной, а частичной корректировки содержимого буфера кадра с учётом атрибутов прозрачных элементов.

Проблема этого алгоритма заключается в том, что иногда приходится отдельно обрабатывать случаи частичного перекрытия интервалов глубины.

1.4.7 Алгоритм обратной трассировки лучей

Алгоритм состоит из следующих шагов [10]:

1. создание лучей с началом в точке наблюдения, каждый луч проходит через один из пикселей;

2. определение пересечения лучей с объектами. Для упрощения используется сферическая или прямоугольная оболочка;

3. поиск ближайших точек пересечения.

1.4.8 Сравнение алгоритмов

В таблице 1.2 представлено сравнение алгоритмов удаления невидимых линий и поверхностей и использованы следующие обозначения:

— Р – алгоритм Робертса;

— ПГ – алгоритм плавающего горизонта;

— В – алгоритм Варнока;

— ВА – алгоритм Вейлера-Азертона;

— Z – алгоритм, использующий Z-буфер;

— СП – алгоритм, использующий список приоритетов;

— ТЛ – алгоритм обратной трассировки лучей;

— О – объектное пространство;

— И – пространство изображения;

— А – аналитически заданный объект;

— ВО – выпуклые объекты;

— П – полигональный объект;

— Л – любой объект.

Таблица 1.2 – Сравнение алгоритмов генерации ландшафта

Критерий сравнения	Р	ПГ	В	ВА	Z	СП	ТЛ
Пространство работы	О	И	И	О	И	О	И
Сложность (N – число полигонов, C – число пикселей)	N^2	CN	CN	N^2	CN	CN	CN
Возможность вычислить интенсивность света	+	-	+	+	+	+	+
Обрабатываемые объекты	ВО	А	Л	П	Л	П	Л

В результате сравнения в качестве алгоритма удаления невидимых линий и поверхностей был выбран алгоритм, использующий Z-буфер, так как он подходит для решения поставленных задач.

1.5 Анализ методов закраски

Существуют следующие методы закраски [11][14]:

1. закраска Гуро;
2. закраска Фонга.

1.5.1 Закраска Гуро

Алгоритм состоит из следующих шагов [14]:

1. вычисление интенсивности в вершинах как результат усреднения нормалей ко всем полигональным граням, которым принадлежит вершина;
2. вычисление значение интенсивности в других пикселях при помощи интерполяции интенсивности.

У данного метода есть недостаток – в результате его применения может появиться эффект полос Маха [11].

1.5.2 Закраска Фонга

Алгоритм состоит из следующих шагов [14]:

1. вычисление интенсивности в вершинах как результат усреднения нормалей ко всем полигональным граням, которым принадлежит вершина;
2. вычисление значение интенсивности в других пикселях при помощи интерполяции векторов нормали.

Данный метод не устраняет эффект полос Маха, а в некоторых случаях он проявляется даже сильнее, чем для метода Гуро [11].

1.5.3 Сравнение алгоритмов

В качестве алгоритма закраски был выбран метод Гуро, так как интерполяция интенсивности выполняется быстрее интерполяции нормалей.

1.6 Формализация задачи с учётом выбранных алгоритмов

На рисунке 1.3 представлена формализованная задача с учётом выбранных алгоритмов.

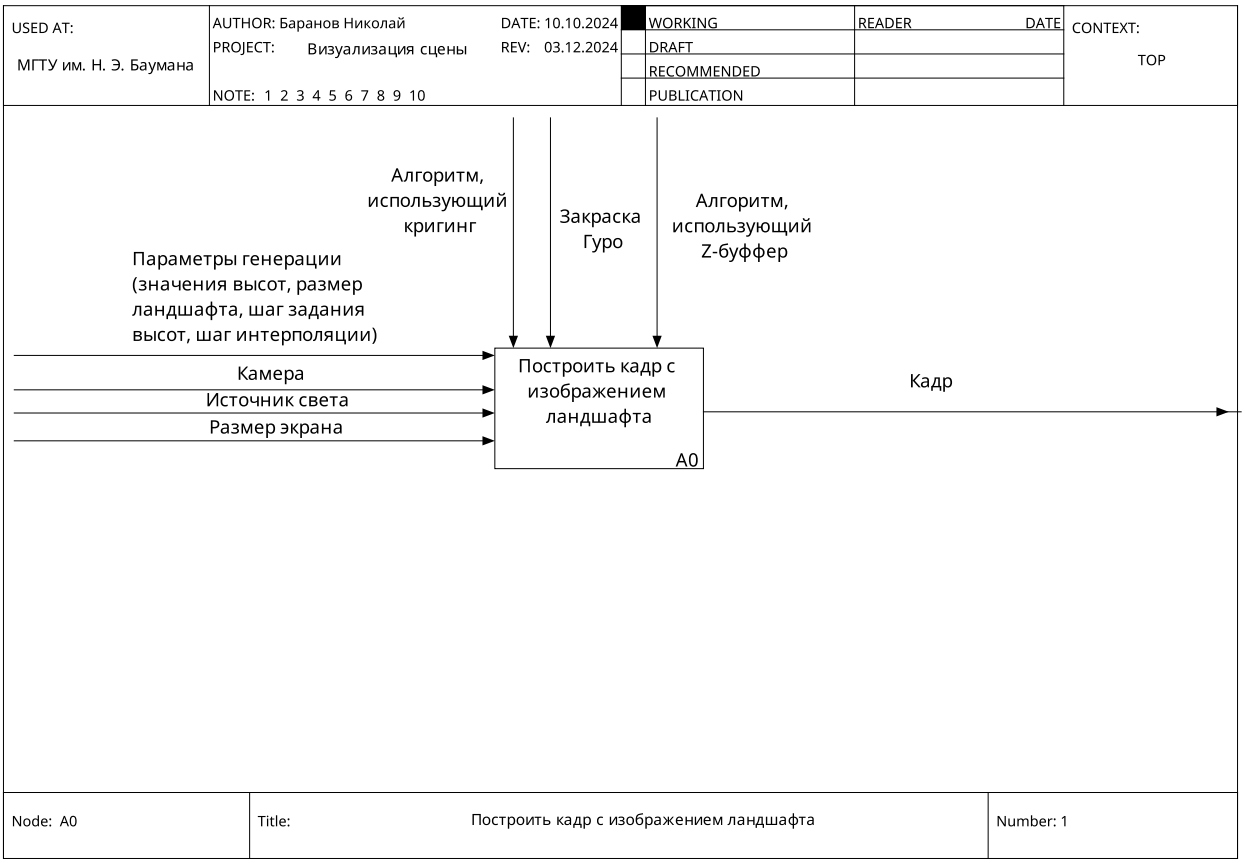

Рисунок 1.3 – Формализованная задача с учётом выбранных алгоритмов

Выводы

В данном разделе была формализована поставленная задача, формализованы объекты сцены, а также произведён поиск и выбор подходящих алгоритмов для решения поставленной задачи. Для генерации ландшафта было принято решение использовать алгоритм, использующий кригинг. Для удаления невидимых линий был выбран алгоритм, использующий Z-буфер, а для закраски граней – метод Гуро.

2 Конструкторская часть

2.1 Функциональная модель программы

На рисунке 2.1 представлена функциональная модель программы.

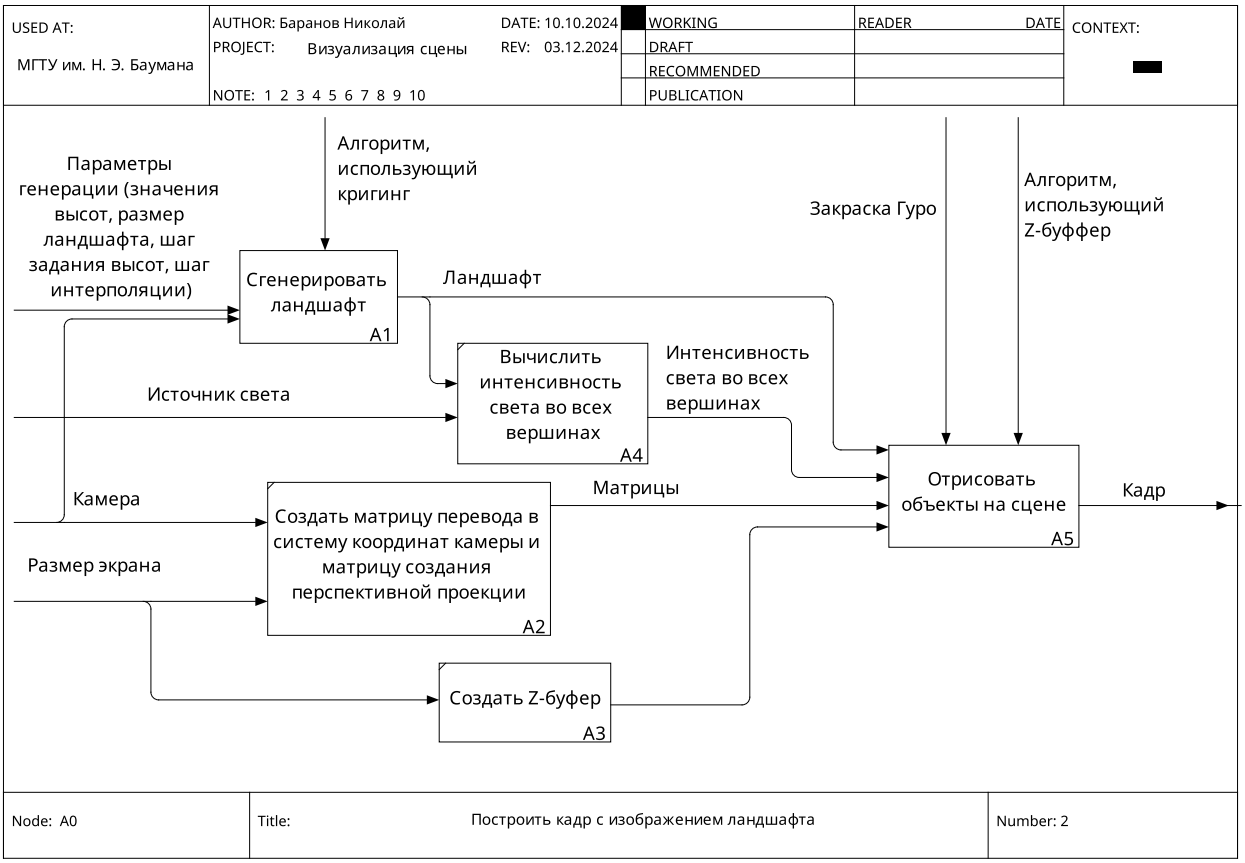

Рисунок 2.1 – Функциональная модель программы

2.2 Используемые типы и структуры данных

Для работы программы потребуются следующие структуры данных:

1) вершина – структура, состоящая из координат(трёхмерный вектор), нормали (трёхмерный вектор) и преобразованных координат (четырёхмерный вектор);

2) матрица размером 4 на 4, состоящая из 16 чисел с плавающей точкой. Требуется для перевода вершин в нужный базис;

3) камера – структура, состоящая из позиции (трёхмерный вектор), базиса (3 трёхмерных вектора), фокусного расстояния (число с плавающей точкой) и дальности прорисовки (число с плавающей точкой);

4) источник света – структура, состоящая из 2 углов (2 числа с плавающей точкой);

5) ландшафт – структура, состоящая из карты высот (двумерный словарь чисел с плавающей точкой) и списка полигонов;

6) сцена, содержащая в себе все остальные объекты.

2.3 Алгоритм, использующий кригинг

На рисунке 2.2 представлена схема алгоритма, использующего кригинг.

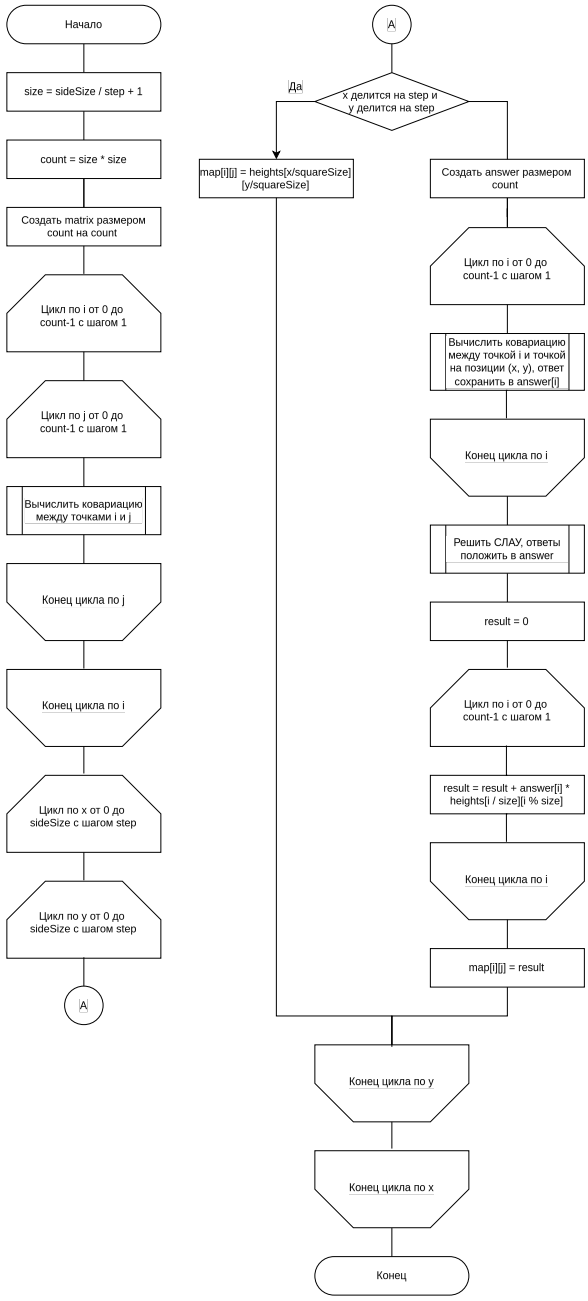

Рисунок 2.2 – Схема алгоритма, использующего кригинг

2.4 Построение кадра

Интенсивность света I вычисляется по закону Ламберта (формула 2.1):

$$I = I_0 \cos \phi \tag{2.1}$$

где I_0 – интенсивность освещения от источника света, а ϕ – угол между нормалью и вектором направления света.

На рисунке 2.3 изображена схема алгоритма преобразования координат вершин ландшафта и отсечения перед отрисовкой. На вход подаются матрицы преобразования в систему координат камеры $camera\_matrix$, матрица создания перспективной проекции $frustum\_matrix$, а также фокусное расстояние $focus$.

Матрица перевода в систему координат камеры $camera\_matrix$ получается по формуле 2.2:

$$camera\_matrix = \begin{pmatrix} VX_x & VX_y & VX_z & 0 \\ VY_x & VY_y & VY_z & 0 \\ VZ_x & VZ_y & VZ_z & focus \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.2}$$

где VX, VY и VZ – векторы базиса камеры, $focus$ - фокусное расстояние камеры.

Матрица создания перспективной проекции $frustum\_matrix$ получается по формуле 2.3:

$$frustum\_matrix = \begin{pmatrix} -f & 0 & \frac{width}{2} & 0 \\ 0 & -f & \frac{height}{2} & 0 \\ 0 & 0 & \frac{v+f}{v} & -\frac{f(v+f)}{v} \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{2.3}$$

где f – фокусное расстояние камеры, v – дальность прорисовки, $width$ - ширина экрана, $height$ - высота экрана.

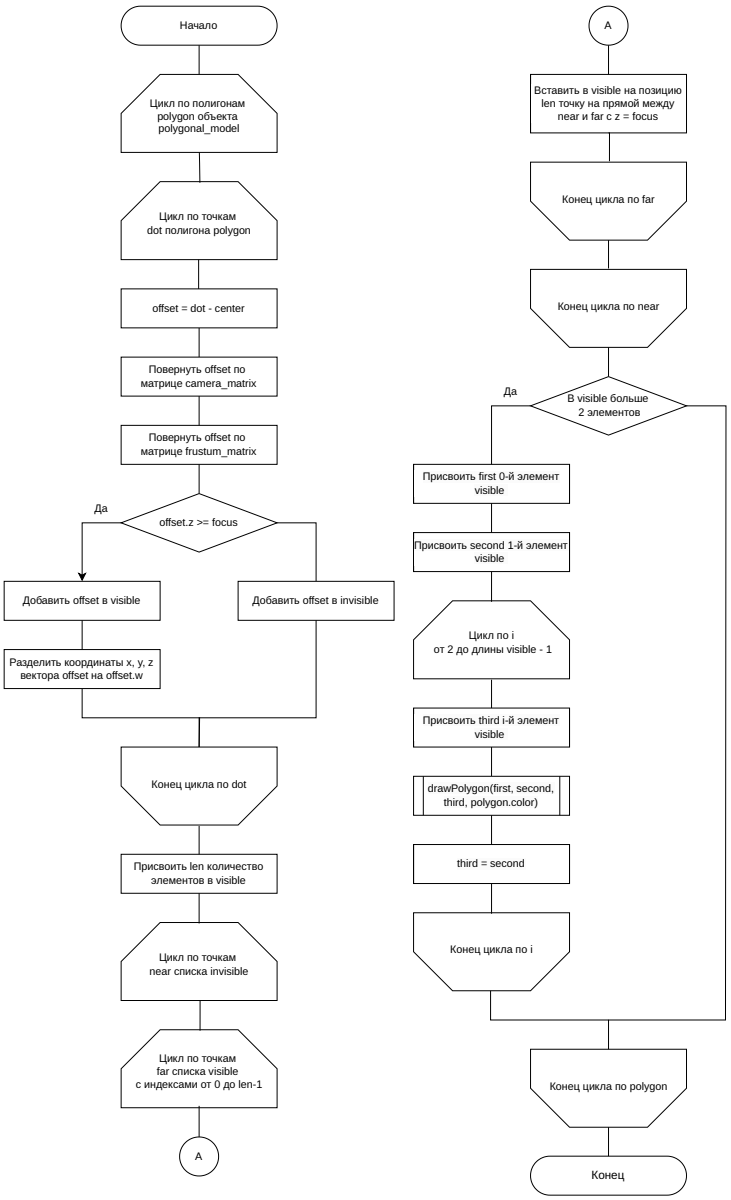

Рисунок 2.3 – Подготовка полигонов к отрисовке

На рисунке 2.4 изображена схема алгоритма отрисовки полигона. На вход алгоритму передаются 3 преобразованные вершины полигона A, B, и C, интенсивность света в них, цвет полигона $color$, Z-буфер $buffer$.

Координата z произвольной точки D на плоскости ABC вычисляется по формуле 2.4:

$$D_z = \alpha A_z + \beta B_z + \gamma C_z \tag{2.4}$$

где α, β и γ – барицентрические координаты точки D на плоскости ABC.

Интенсивность света I в точке D вычисляется по формуле 2.5:

$$D_I = \alpha A_I + \beta B_I + \gamma C_I \tag{2.5}$$

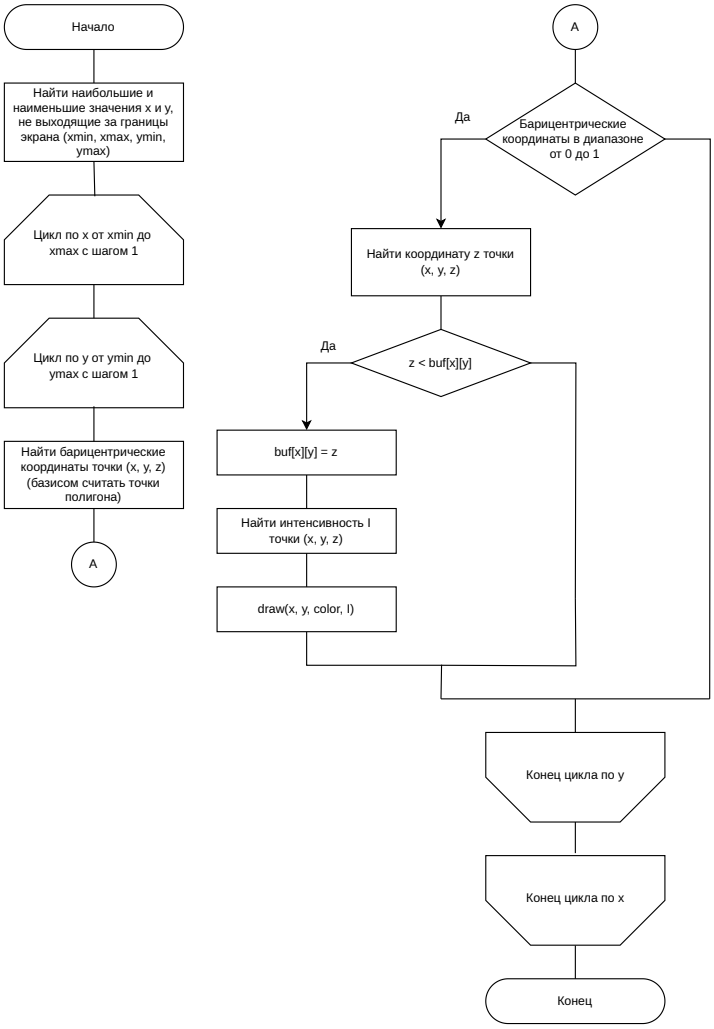

Рисунок 2.4 – Отрисовка треугольного полигона с помощью алгоритма, использующего Z-буфер, и закраски Гуро

Выводы

В данном разделе были выбраны структуры данных, а также построены схемы алгоритмов и разработана функциональная схема работы программы.

3 Технологическая часть

3.1 Средства реализации

Для реализации программы был выбран язык программирования Java, так как он содержит все необходимые средства для реализации выбранных в результате проектирования алгоритмов и структур данных.

Для тестирования был выбран фреймворк JUnit5, так как он имеет достаточный функционал для написания модульных тестов.

3.2 Примеры реализаций алгоритмов

В листингах 3.1-3.3 представлена реализация алгоритма преобразования координат вершин ландшафта и отсечения перед отрисовкой.

Листинг 3.1 – Реализация алгоритма подготовки полигонов к отрисовке (начало)

```
@Override
public void visit(@NotNull PolygonalModel polygonalModel) {
    for (var polygon : polygonalModel) {
        drawPolygon(polygon);
    }
}

private void drawPolygon(Polygon polygon) {
    for (var elem : polygon) {
        var dot = elem.drawDot;
        if (!dot.isUsed) {
            dot.set(center, elem.realDot);
            cameraMatrix.transformVector(dot);
            frustumMatrix.transformVector(dot);
            if (dot.w >= focus) {
                maybeVisible.add(dot);
                dot.to3D();
            } else {
                invalid.add(dot);
            }
            dot.isUsed = true;
        } else {
            if (dot.w >= focus) {
                maybeVisible.add(dot);
```

Листинг 3.2 – Реализация алгоритма подготовки полигонов к отрисовке (продолжение)

```
            } else {
                invalid.add(dot);
            }
        }
    }
    var len = maybeVisible.size();
    for (var incorrect : invalid) {
        for (var i = 0; i < len; ++i) {
            maybeVisible.add(len, findDot(incorrect, maybeVisible
               .get(i)));
        }
    }
    triangulate(maybeVisible, polygon.color);
    invalid.clear();
    maybeVisible.clear();
}

private DrawVector findDot(DrawVector a, DrawVector b) {
    var t = (b.w - focus) / (b.w - a.w);
    var result = new DrawVector();
    var x = b.x * b.w;
    result.x = x + (a.x - x) * t;
    var y = b.y * b.w;
    result.y = y + (a.y - y) * t;
    result.w = focus;
    result.to3D();
    result.isUsed = true;
    result.brightness = b.brightness + (a.brightness - b.
       brightness) * t;
    return result;
}

private void triangulate(List<DrawVector> polygon, Color color) {
    if (polygon.size() < 3) {
        return;
    }
    var first = polygon.getFirst();
    var second = polygon.get(1);
    for (var i = 2; i < polygon.size(); ++i) {
```

Листинг 3.3 – Реализация алгоритма подготовки полигонов к отрисовке (окончание)

```
        var third = polygon.get(i);
        drawStrategy.draw(first, second, third, color);
        second = third;
    }
}
```

В листингах 3.4-3.5 представлена реализация алгоритма, использующего Z-буффер, и закраски Гуро.

Листинг 3.4 – Реализация алгоритма отрисовки полигона (начало)

```
@Override
public void draw(@NotNull DrawVector d1, @NotNull DrawVector d2,
   @NotNull DrawVector d3, @NotNull Color color) {
    var x0 = Math.max(xMin, (int) Math.floor(Math.min(d1.x, Math.
       min(d2.x, d3.x))));
    var x1 = Math.min(xMax, (int) Math.ceil(Math.max(d1.x, Math.
       max(d2.x, d3.x))));
    var y0 = Math.max(yMin, (int) Math.floor(Math.min(d1.y, Math.
       min(d2.y, d3.y))));
    var y1 = Math.min(yMax, (int) Math.ceil(Math.max(d1.y, Math.
       max(d2.y, d3.y))));
    var divider = (d1.x - d2.x) * (d2.y - d3.y) - (d2.x - d3.x) *
        (d1.y - d2.y);
    for (var x = x0; x <= x1; ++x) {
        for (var y = y0; y <= y1; ++y) {
            var k1 = ((x - d2.x) * (y - d3.y) - (x - d3.x) * (y -
                d2.y)) / divider;
            var k2 = ((x - d3.x) * (y - d1.y) - (x - d1.x) * (y -
                d3.y)) / divider;
            var k3 = ((x - d1.x) * (y - d2.y) - (x - d2.x) * (y -
                d1.y)) / divider;
            if (k1 >= 0 && k1 <= 1 && k2 >= 0 && k2 <= 1 && k3 >=
                0 && k3 <= 1) {
                var z = d1.z * k1 + d2.z * k2 + d3.z * k3;
                if (buffer[x][y] > z) {
                    buffer[x][y] = z;
                    if (z > 0.9) {
                        color.setAlpha((int) ((1 - z) * 2550));
                    }
```

Листинг 3.5 – Реализация алгоритма отрисовки полигона (окончание)

```
                    var brightness = d1.brightness * k1 + d2.
                       brightness * k2 + d3.brightness * k3;
                    color.setBrightness(brightness);
                    image.setPixel(x, y, color.
                       getRGBWithBrightness());
                    color.setAlpha(255);
                }
            }
        }
    }
}
```

В листингах 3.6-3.9 представлена реализация алгоритма генерации ландшафта с помощью кригинга.

Листинг 3.6 – Реализация алгоритма генерации ландшафта (начало)

```
@Override
public void run() {
    synchronized (landscape) {
        needStop = false;
        var map = generateHeights();
        var polygons = generatePolygons(map);
        if (!needStop) {
            landscape.setPolygons(polygons);
            landscape.setInputHeightsMap(heights);
            landscape.setSideSize(sideSize);
            landscape.setSquareSize(squareSize);
            landscape.setStep(step);
            landscape.setNeedRecalculate(true);
        }
    }
}

private Map2D<Integer, PolygonVector> generateHeights() {
    var map = new HashMap2D<Integer, PolygonVector>();
    for (var i = 0; i <= sideSize && !needStop; i += step) {
        for (var j = 0; j <= sideSize && !needStop; j += step) {
            if (i % squareSize + j % squareSize > 0) {
                map.put(i, j, new PolygonVector(i, interpolate(i,
                    j), j));
            } else {
```

Листинг 3.7 – Реализация алгоритма генерации ландшафта (продолжение)

```
                map.put(i, j, new PolygonVector(i, fastHeights[i
                    / squareSize][j / squareSize], j));
            }
        }
    }
    return map;
}

private List<Polygon> generatePolygons(@NotNull Map2D<Integer,
    PolygonVector> heights) {
    var polygons = new LinkedList<Polygon>();
    for (var i = 0; i < sideSize && !needStop; i += step) {
        var near = heights.get(i, 0);
        var dx = heights.get(i + step, 0);
        var dz = near;
        var far = dx;
        for (var j = 0; j < sideSize && !needStop; j += step) {
            far = heights.get(i + step, j + step);
            dz = heights.get(i, j + step);
            var normal1 = Vector3D.getNormal(near, dx, far);
            if (normal1.y < 0) {
                normal1.inverse();
            }
            var normal2 = Vector3D.getNormal(near, dz, far);
            if (normal2.y < 0) {
                normal2.inverse();
            }
            normal1.normalize();
            normal2.normalize();
            polygons.add(new Polygon(near, dx, far, normal1, new
                Color(0, 255, 0)));
            polygons.add(new Polygon(near, dz, far, normal2, new
                Color(0, 255, 0)));
            near = dz;
            dx = far;
        }
    }
    return polygons;
}
```

Листинг 3.8 – Реализация алгоритма генерации ландшафта (продолжение)

```
private double interpolate(int x, int y) {
    var result = 0.0;
    if (x % squareSize == 0 || y % squareSize == 0) {
        result = fastHeights[x / squareSize][y / squareSize];
    } else {
        var matrix = copy(crigingMatrix);
        var answer = new double[matrix.length];
        var xi = 0;
        var yi = 0;
        for (var k = 0; k < matrix.length; ++k) {
            answer[k] = covariation(xi, yi, x, y);
            if (yi == sideSize) {
                yi = 0;
                xi += squareSize;
            } else {
                yi += squareSize;
            }
        }
        Gauss.solve(matrix, answer);
        var i = 0;
        var j = 0;
        for (var k = 0; k < matrix.length; ++k) {
            result += answer[k] * fastHeights[i][j];
            ++j;
            if (j == size) {
                j = 0;
                ++i;
            }
        }
    }
    return result;
}

private double covariation(int x0, int y0, int x1, int y1) {
    var h2 = (x1 - x0) * (x1 - x0) + (y1 - y0) * (y1 - y0);
    return 1 - Math.exp(-h2/radius2);
}

private double[][] copy(double[][] matrix) {
    var result = new double[matrix.length][matrix.length];
```

Листинг 3.9 – Реализация алгоритма генерации ландшафта (окончание)

```
        for (var i = 0; i < matrix.length; ++i) {
        System.arraycopy(matrix[i], 0, result[i], 0, matrix.
           length);
    }
    return result;
}
```

3.3 Интерфейс программы

На рисунке 3.1 представлен пример графического интерфейса программы.

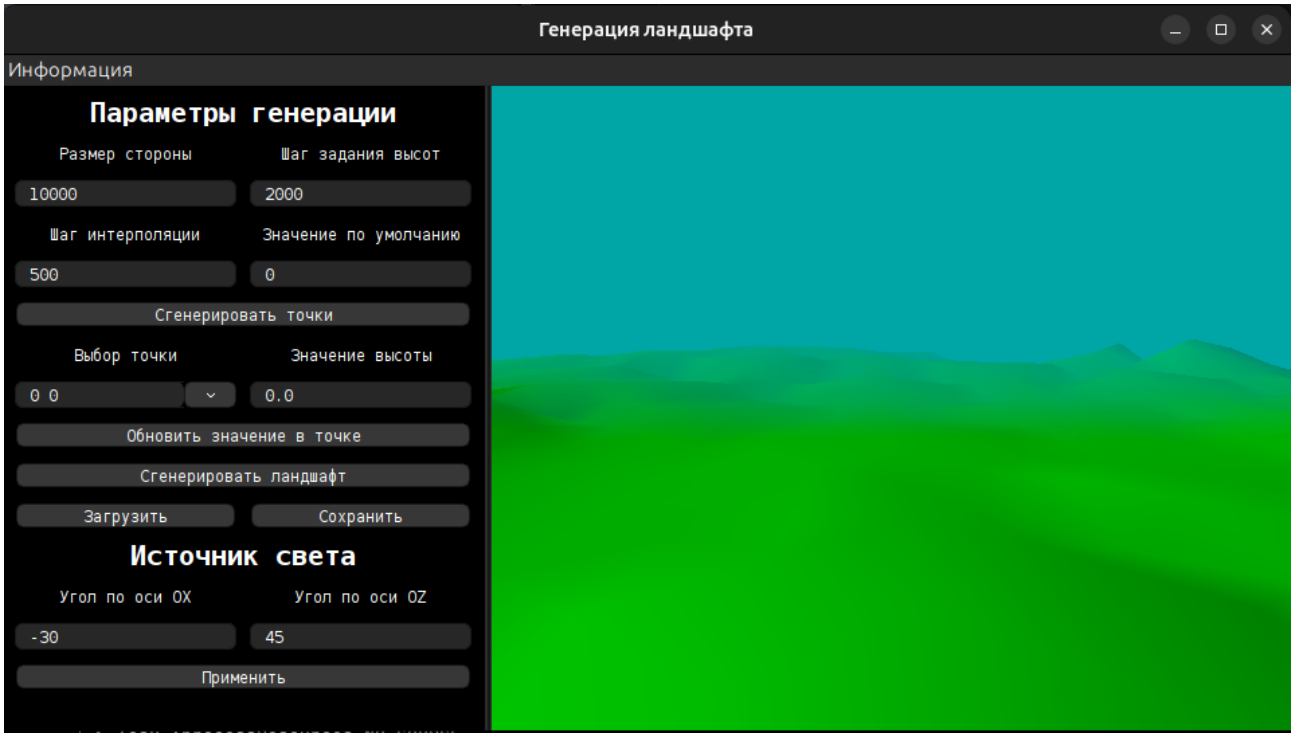

Рисунок 3.1 – Пример работы программы

3.4 Тестирование

3.4.1 Модульное тестирование

Было реализовано модульное тестирование с помощью фреймворка JUnit5. В качестве меры, используемой при тестировании программы, было выбрано покрытие кода, подсчёт которого производился автоматически средствами среды разработки IntelliJ Idea. Процент покрытия кода составил 18%.

Для создания теста использовалась аннотация @*Test*, а для проверки результата использовались статические методы класса *Assertions*. Пример модульного теста приведён в листинге 3.10. Результат выполнения модульных тестов представлен в листинге 3.11.

Листинг 3.10 – Пример модульного теста

```
@Test
void negativeHeights2() {
    try {
        heights.getLast().removeLast();
        new Generator(landscape, heights, sideSize, squareSize,
           step);
        fail();
    } catch (IllegalArgumentException ignored) {
    }
}
```

Листинг 3.11 – Результат выполнения модульных тестов

```
> Task :test

com.github.N1ckBaran0v.program.math.GaussTest

Test solve() PASSED
Test negative() PASSED

com.github.N1ckBaran0v.program.scene.GeneratorTest

Test negativeSquareSize() PASSED
Test run() PASSED
Test negativeHeights1() PASSED
Test negativeHeights2() PASSED
Test negativeHeights3() PASSED
Test negativeStep() PASSED
Test negativeSideSize() PASSED

SUCCESS: Executed 9 tests in 1s
```

3.4.2 Функциональное тестирование

Этапы функционального тестирования:

1) определение классов эквивалентности функционального тестирования;

2) составление входных данных для каждого класса эквивалентности;

3) генерация ландшафта и получение изображения для каждого тестового случая;

4) визуальная оценка результата.

Классы эквивалентности функционального тестирования:

1) все значения высот равны 0;

2) значения высот сгенерированы с использованием шума Перлина;

3) «квадратный» холм;

4) «круглый» холм;

5) на главных диагоналях значения высот равны 1000, в остальных вершинах – 0.

Примеры работы программы для каждого из тестовых случаев представлены на рисунках 3.2-3.6.

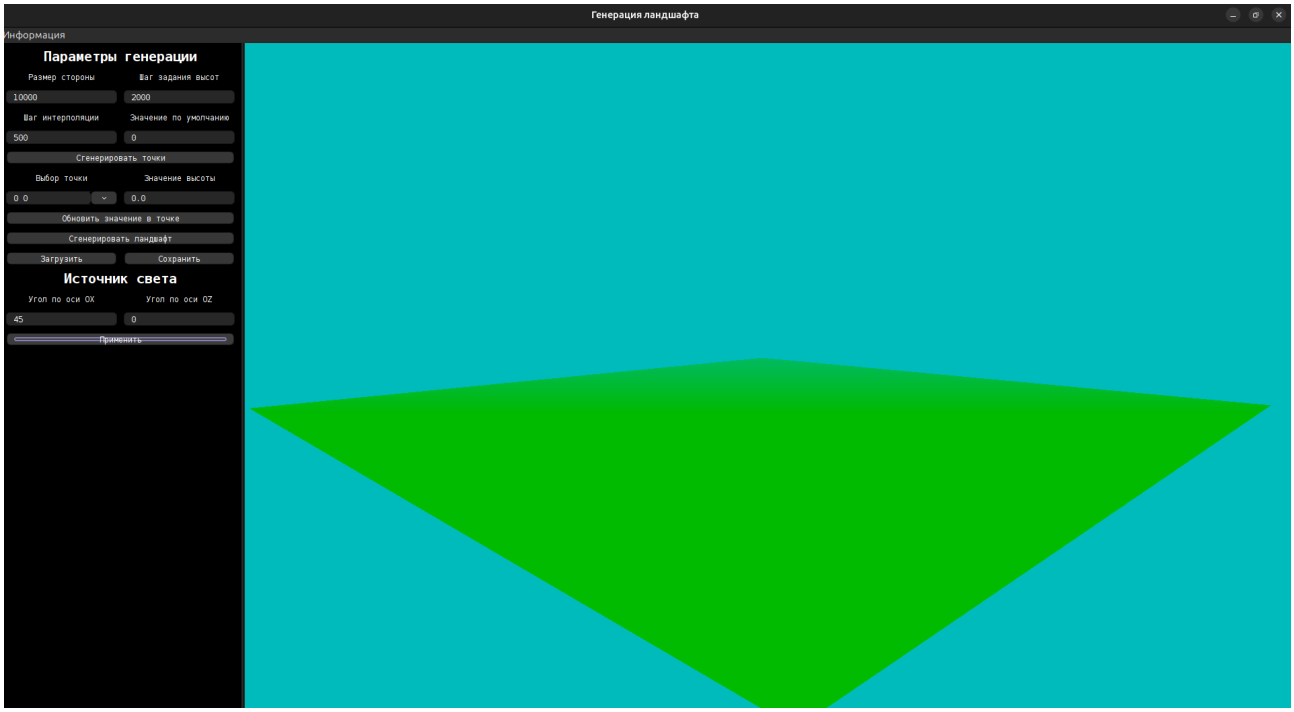

Рисунок 3.2 – Пример работы программы для теста 1

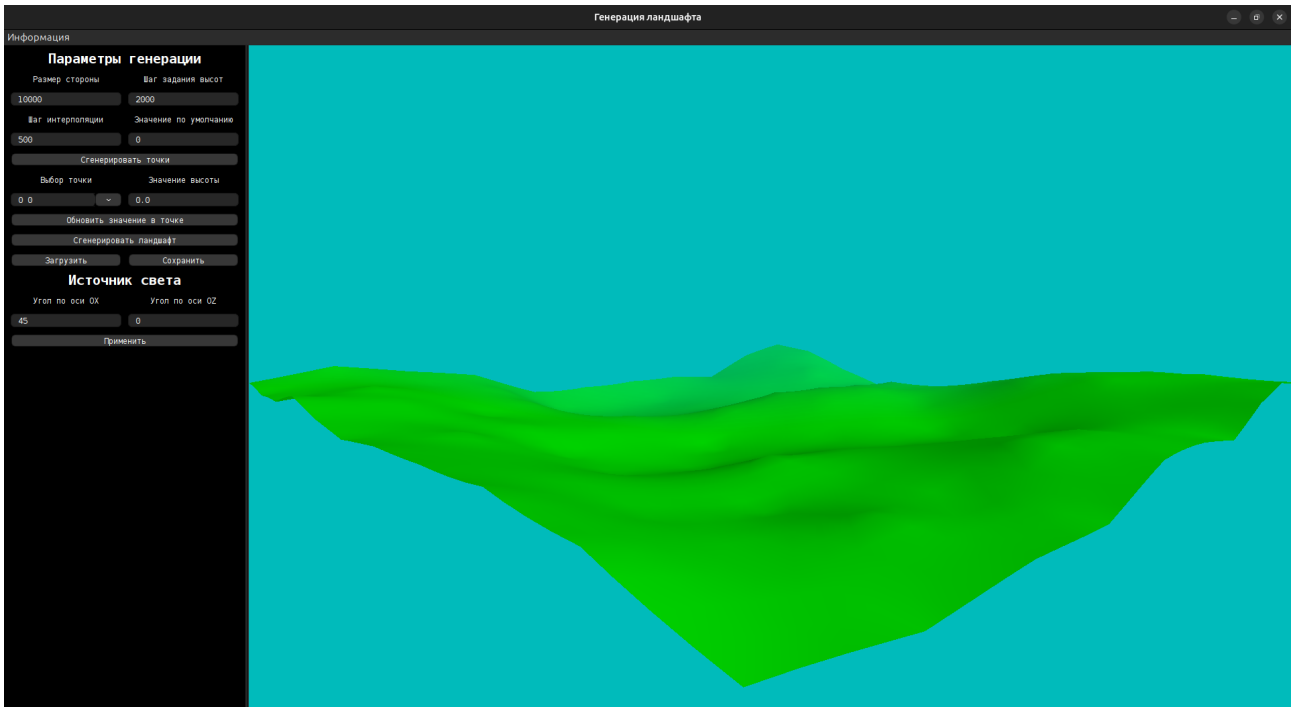

Рисунок 3.3 – Пример работы программы для теста 2

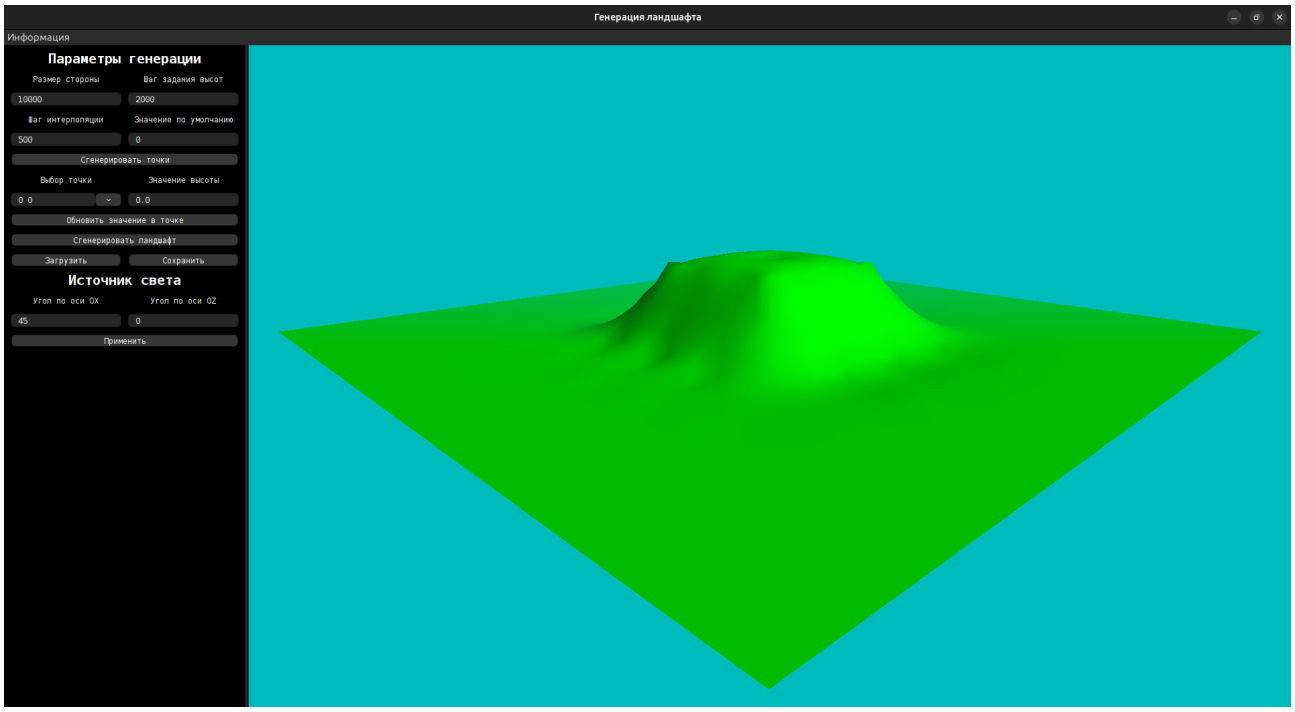

Рисунок 3.4 – Пример работы программы для теста 3

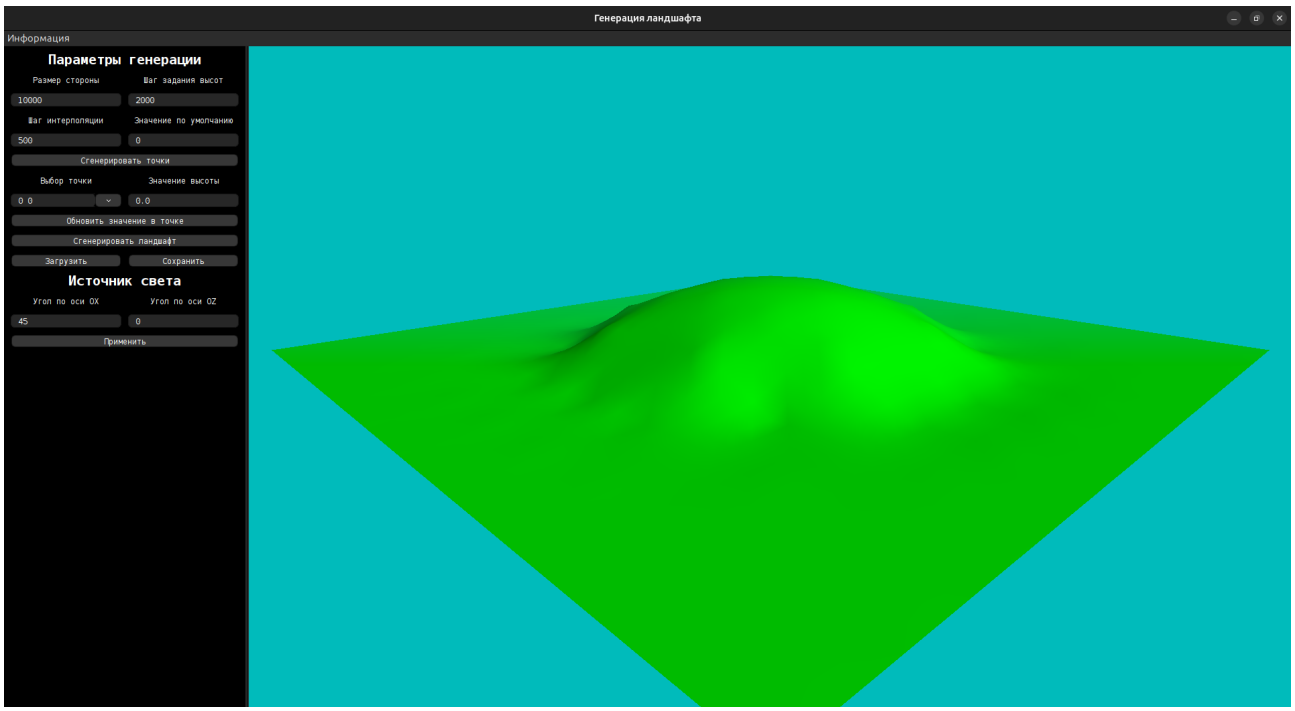

Рисунок 3.5 – Пример работы программы для теста 4

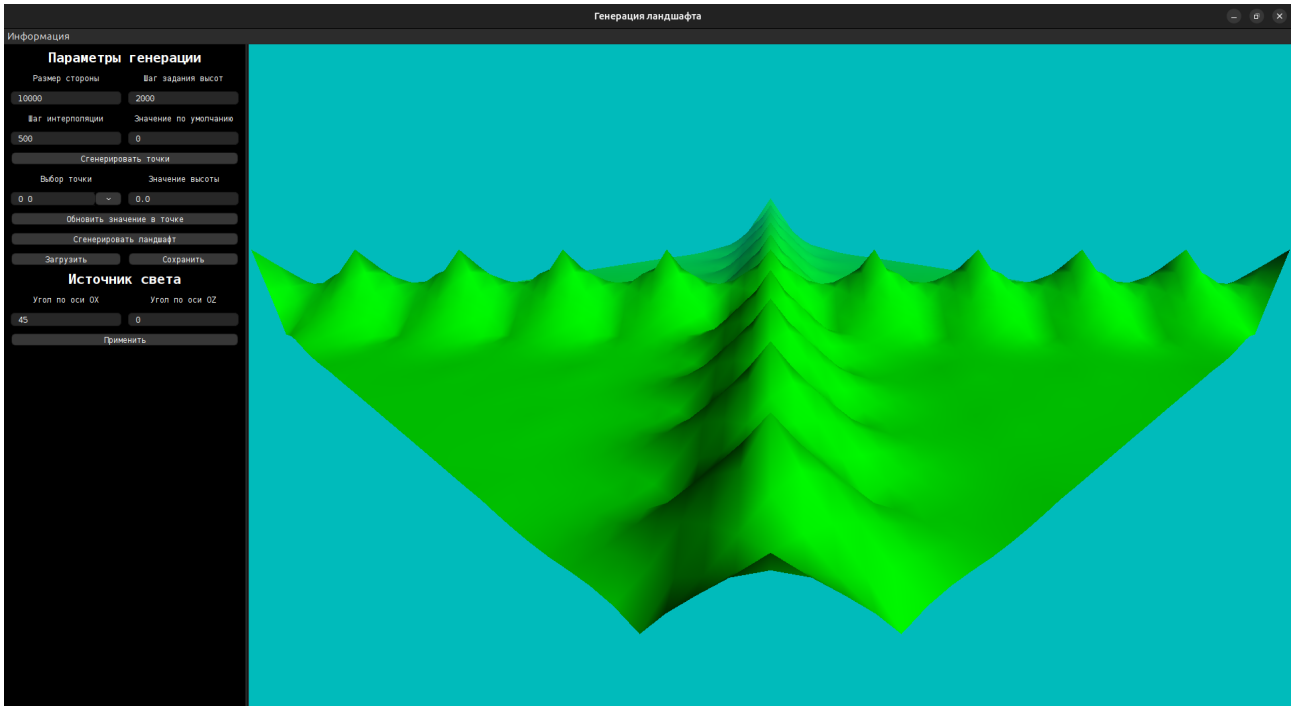

Рисунок 3.6 – Пример работы программы для теста 5

Выводы

В данном разделе были выбраны средства реализации, разработана спроектированная программа, представлен графический интерфейс и проведено тестирование.

4 Исследовательская часть

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось исследование:

— операционная система – Ubuntu 24.04.1 LTS;

— процессор – 1th Gen Intel® Core™ i7-1165G7;

— оперативная память – 16 Гб.

4.2 Зависимость скорости генерации ландшафта от параметров генерации

Для замера процессорного времени использовался метод *getCurrentThreadCpuTime* класса *ThreadMXBean* из пакета *java.lang.management*. При замерах зависимости скорости генерации от одного из параметров для других параметров выставлялись значения по умолчанию. Для размера ландшафта значением по умолчанию считалось 5000, для шага задания точек – 1000, для шага интерполяции – 100.

Размеры ландшафта при получении зависимости скорости генерации от размера были взяты в диапазоне от 1000 до 10000 с шагом 1000. Результаты приведены в таблице 4.1. График зависимости времени генерации ландшафта от размера приведён на рисунке 4.1. Также на графике присутствует аппроксимация полинома 8 степени.

Для получения зависимости скорости генерации от шага задания точек были проведены замеры на значениях 100, 500, 1000, 2500 и 5000. Результаты приведены в таблице 4.2. График зависимости времени генерации ландшафта от шага задания точек приведён на рисунке 4.2. Также на графике присутствует аппроксимация функции, обратной к полиному 4 степени.

Для получения зависимости скорости генерации от шага интерполяции были проведены замеры на значениях 50, 100, 250, 500 и 1000. Результаты приведены в таблице 4.3. График зависимости времени генерации ландшафта от шага задания точек приведён на рисунке 4.3. Также на графике присутствует аппроксимация функции, обратной к полиному 4 степени.

Таблица 4.1 – Зависимость времени генерации ландшафта от размера

Размер стороны ландшафта	Время генерации, с
1000	0.000615656
2000	0.002232261
3000	0.007531867
4000	0.028810261
5000	0.092692385
6000	0.296540696
7000	0.717306372
8000	1.871987038
9000	3.840488489
10000	8.578590878

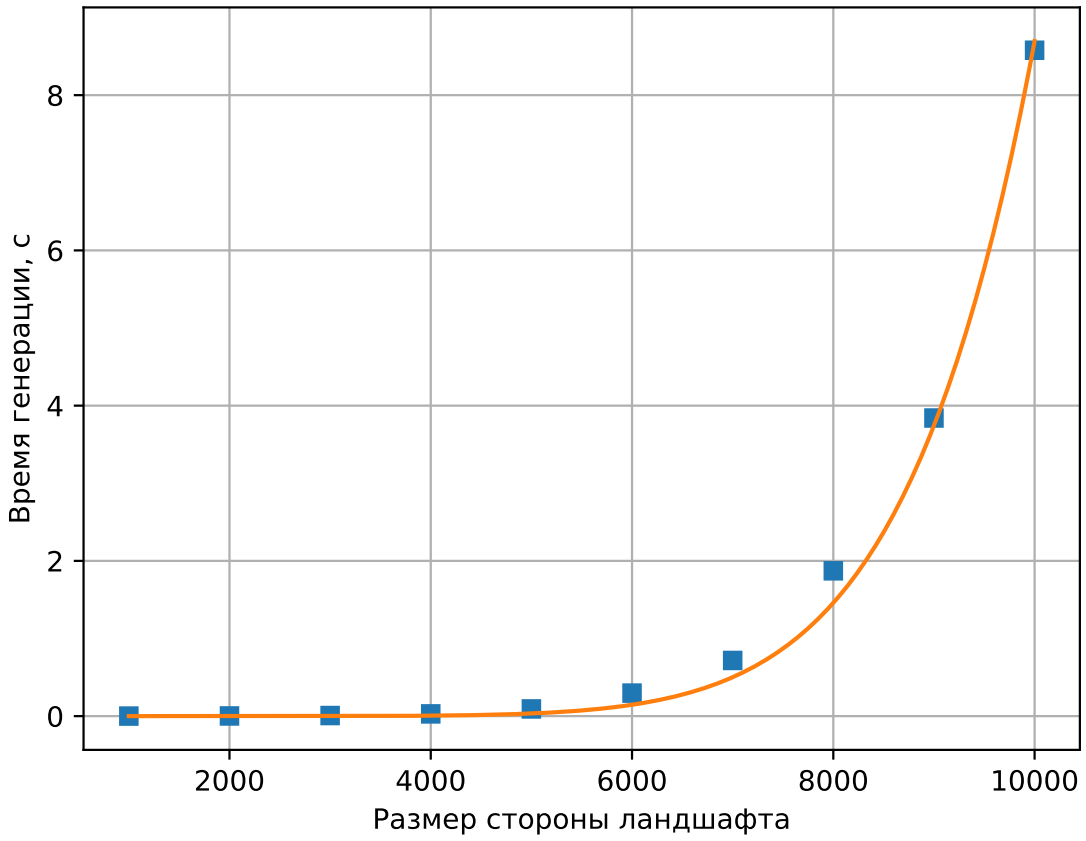

Рисунок 4.1 – Зависимость времени генерации ландшафта от размера

Таблица 4.2 – Зависимость времени генерации ландшафта от шага задания точек

Шаг задания точек	Время генерации, с
100	0.369764550
500	2.112369068
1000	0.091958011
2500	0.007408134
5000	0.003019618

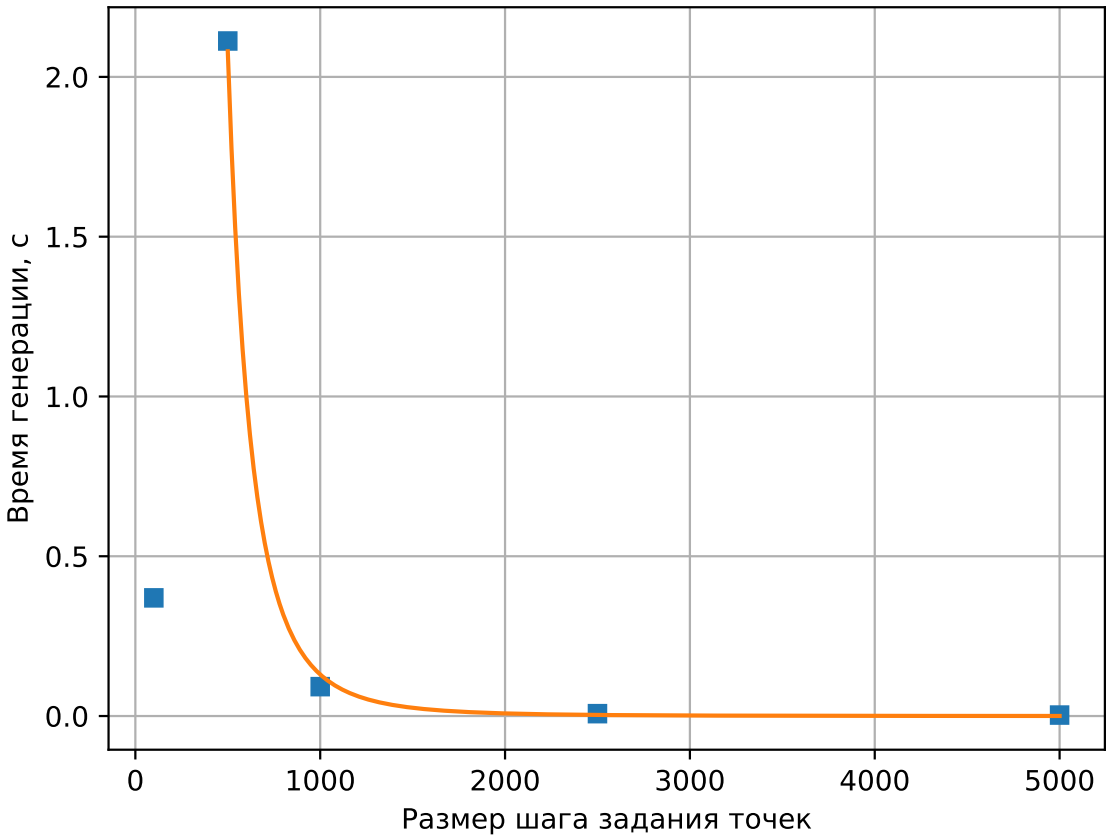

Рисунок 4.2 – Зависимость времени генерации ландшафта от шага задания точек

Таблица 4.3 – Зависимость времени генерации ландшафта от шага интерполяции

Шаг задания точек	Время генерации, с
50	0.358149701
100	0.091089610
250	0.014978900
500	0.003190569
1000	0.000084382

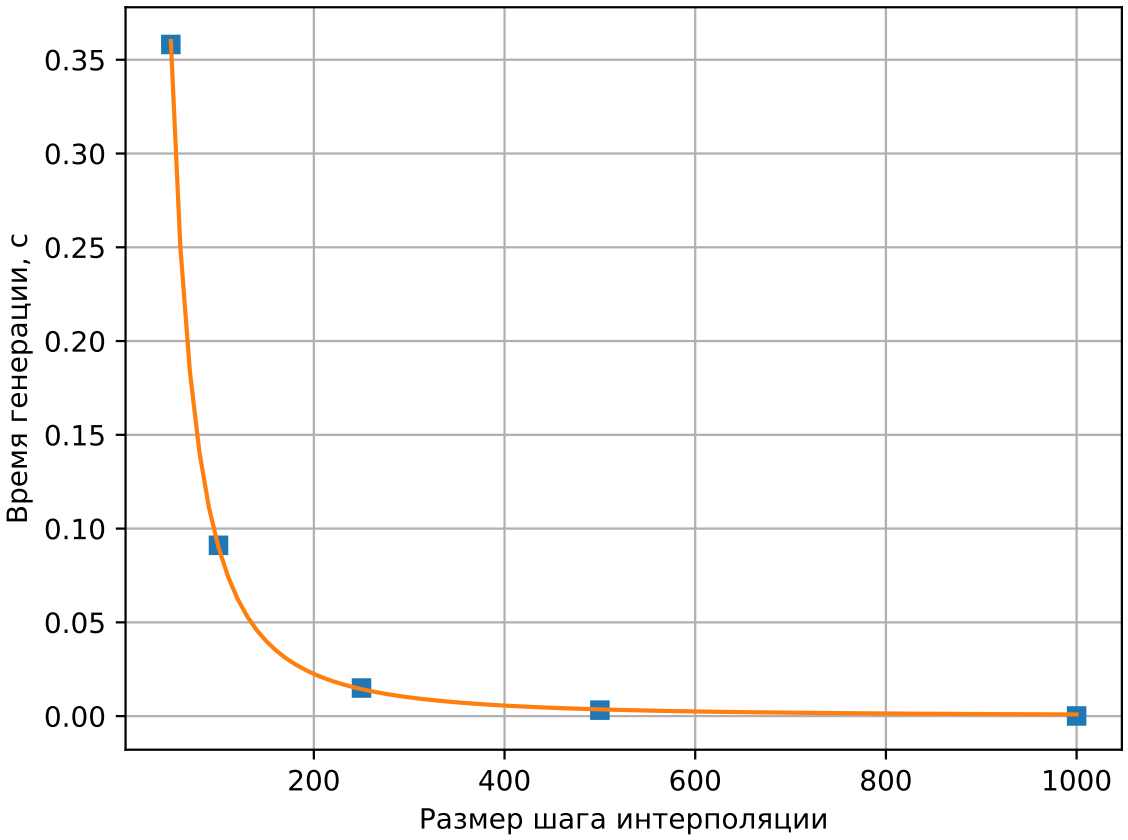

Рисунок 4.3 – Зависимость времени генерации ландшафта от шага интерполяции

Выводы

В данном разделе было проведено исследование зависимости скорости генерации ландшафта от параметров генерации. При увеличении размера ландшафта время его генерации растёт как полином 8 степени, а при увеличении шага задания точек и шага интерполяции - убывает как функция, обратная к полиному 4 степени. Это не относится к случаю, когда шаг задания точек и шаг интерполяции совпадают, так как в данном случае интерполяция не проводится.

ЗАКЛЮЧЕНИЕ

В ходе выполнения курсовой работы была достигнута цель – была разработана программу для генерации ландшафта и его визуализации. Были выполнены все поставленные цели:

1) была формализована задача;
2) были проанализированы и выбраны алгоритмы для создания изображения и генерации ландшафта;
3) была спроектирована программа для генерации ландшафта и его визуализации;
4) были выбраны средства реализации, а также была реализована спроектированная программа;
5) была исследована зависимость скорости генерации ландшафта от параметров генерации.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *Шишлянников И. В.* Анализ и применение алгоритмов генерации и моделирования ландшафта. — 78-я научная конференция аспирантов, магистрантов и студентов Белорусского государственного университета, 2021. — С. 107—111.

2. *Хисамов И. Р.* Обзор шумовых функций для генерации ландшафта. — 52 научные исследования учащихся XI международная научно-практическая конференция | МЦНС «Наука и просвещение», 2024. — С. 52—54.

3. Проектирование ландшафта с помощью алгоритмов генерации /. А. Ж. Кинтонова [и др.]. — Вестник КазАТК No 1 (124), 2023. — С. 182—194.

4. *Аникеев А. С.* Генерация и моделирование 2D ландшафта по контрольным значениям с использованием Unity на языке программирования C#. — Инженерный вестник Дона, 2020.

5. *Тимофеева. Н. Е.*, *Герасъкин. А. С.* Исследование возможности улучшения алгоритма билинейной интерполяции для корректировки цифровых изображений применением теории полей ориентации. — Вестник ВГУ. Серия: Системный анализ и информационные технологии, (1), 2018. — С. 119—125.

6. *Ремизов. А. Л.*, *Колединн. С. Н.* Решение задачи двумерной интерполяции исходных данных методом простого кригинга на языке программирования Python. — Математическое моделирование процессов и систем. Стерлитамак, 10–12 ноября 2021 года, 2021. — С. 119—123.

7. *Мультан. Р. И.*, *Мазур. А. Д.* Процедурная генерация текстур на основе алгоритма diamond-square. — 54-я научная конференция аспирантов, магистрантов и студентов БГУИР, 2018. — С. 66—67.

8. *Гимранова Р. М.* Методы генерации ландшафта карт в разработке игр на примере Unreal Engine 4. — 2020. — С. 21—22.

9. *Паршина К. С.* Алгоритмы удаления невидимых линий при построении изображений трехмерных тел. — Информационно-телекоммуникационные технологии и математическое моделирование высокотехнологичных систем : материалы Всероссийской конференции с

международным участием. Москва, РУДН, 16–20 апреля 2018 г., 2018. — С. 204—206.

10. *Головнин А. А.* Базовые алгоритмы компьютерной графики. — Проблемы качества графической подготовки студентов в техническом вузе: традиции и инновации, 2016. — С. 13—30.

11. *Митин. А. И.*, *Свертилова. Н. В.* Компьютерная графика: справочно-методическое пособие. — М.-Берлин: Директ-Медиа, 2016.

12. *Емельянова. Т. В.*, *Аминов. Л. А.*, *Емельянов. В. А.* Реализация алгоритма удаления невидимых граней. — Актуальные проблемы военно-научных исследований, 2021. — С. 37—44.

13. *Борисов. А. Н.*, *Сиек. Ю. Л.* Визуализация подводной сцены в параллельной вычислительной системе. — Морские интеллектуальные технологии, 2018. — С. 119—126.

14. *Сафина Д. Н.* Исследование методов закрашивания Гуро и Фонга. — Международная молодёжная научная конференция, посвященная 60-летию со дня осуществления Первого полета человека в космическое пространство и 90-летию Казанского национального исследовательского технического университета им. А.Н. Туполева-КАИ. Материалы конференции. Сборник докладов. В 6-ти томах. Том V. Казань, 2021. — С. 579—581.