Оглавление

Введение

Генерация ландшафта постоянно используется при моделировании трехмерных виртуальных сцен. Подобные сцены используются в фильмах с использованием компьютерной графики, трехмерных играх или в качестве демонстрационного материала для различных архитектурных объектов.

В кинематографе трехмерные ландшафты применяются, например, для создания сцен больших размеров, ручное моделирование которых оказалось бы нецелесообразно. Компьютерные игры используют методы генерации ландшафтов для создания уникальных игровых миров.

Инструменты для моделирования ландшафтов зачастую являются частью крупного программного продукта, установка и освоение подобных программ может быть довольно сложным процессом. Они позволяют сэкономить время для художника, способствуют быстрой итерации изменений.

Цель работы – разработать программу генерации и визуализации ландшафта. Для реализации данного проекта, необходимо решить ряд задач:

- проанализировать представления данных о ландшафте;
- проанализировать алгоритмы генерации ландшафта;
- проанализировать алгоритмы удаления невидимых поверхностей;
- выбрать необходимые структуры данных для изображения ландшафта;
- выбрать основные инструменты для разработки программы;
- разработать программу, реализующую поставленную задачу;
- провести сравнительный анализ времени выполнения алгоритма ZBuffer с использованием параллельных вычислений и без;
- оценить время выполнения алгоритма diamond-square в зависимости от размера карты высот.

1. Аналитическая часть

В данном разделе формализованы объекты синтезируемой сцены, а также рассмотрены принципы представления данных о ландшафте, алгоритмы генерации ландшафта и удаления невидимых поверхностей.

1.1 Формализация объектов синтезируемой сцены

На этапе конструирования потребуется формализовать структуру программы, поэтому следует обозначить объекты, которые составляют изображаемую сцену. Сцена состоит из следующих объектов, описанных ниже.

1) Ландшафт. Он представляет собой поверхность, состоящую из различных вершин. Чтобы изобразить поверхность ландшафта было решено использовать треугольные полигоны, т.к. на основе любых трёх точек может образовать поверхность, что очень удобно.
2) Статический источник света, находящийся в бесконечности над ландшафтом.

1.2 Представление данных о ландшафте

Существует несколько основных принципов представления данных для хранения информации о ландшафтах:

- первый – использование регулярной сетки высот (карты высот);
- второй – использование иррегулярной сетки вершин и связей, их соединяющих;
- третий – хранение карты ландшафта, но в данном случае хранятся не конкретные высоты, а информация об использованном блоке, в этом случае создается некоторое количество заранее построенных сегментов, а на карте указываются только индексы этих сегментов.

1.2.1 Регулярная сетка высот

В карте высот данные представлены в виде двухмерного массива, каждому элементу которого соответствует значение высоты в точке (x, y), где x и y – индексы матрицы.

С помощью этого способа можно представить достаточно обширные пространства. Но у него есть один существенный недостаток — слишком много описаний для точек, а также, в некоторых случаях, наблюдается избыточность данных (например, когда задается простая плоскость, то в этом случае, для построения простой плоскости будет использоваться множество точек, хотя можно было обойтись тремя). Но, с другой стороны, в некоторых ситуациях, данная избыточность может стать плюсом. В случае, если мы хотим изменить какую-то конкретную высоту или область ландшафта, мы легко можем это сделать. Кроме этого, для каждого элемента такой карты можно хранить не только значения высот, но и другие параметры, которые хранят информацию об особенностях ландшафта в конкретной точке (например, цвет). Также, поскольку вершины расположены регулярно и достаточно близко, можно более точно производить динамическое освещение [1].

Более того, благодаря регулярному расположению вершин, нет необходимости задумываться о проблеме, связанной с закраской невыпуклого многоугольника [2, с. 396].

1.2.2 Иррегулярная сетка

Еще один способ представления данных для ландшафтов — иррегулярная сетка вершин и связей их соединяющих.

По сравнению с картой высот используется значительно меньше информации для построения ландшафта. При данном подходе необходимо хранить значения высот вершин и информацию о связях между ними только для конкретного, небольшого количества точек. Это дает выигрыш в скорости при передаче огромных массивов информации, в процессе визуализации

ландшафта. Однако алгоритмы построения ландшафтов в основном предназначены для регулярных карт высот. Оптимизация таких алгоритмов под этот способ потребует значительных усилий. Также, поскольку вершины расположены достаточно далеко друг от друга и неравномерно, возникают сложности при динамическом освещении. Кроме того, хранение, просмотр и модификация такого ландшафта тоже предоставляет сложности [1].

1.2.3 Посегментная карта высот

В данном способе также используются карты высот. Только вместо высот в ней хранятся индексы *ландшафтных сегментов*. Как эти сегменты представлены, в принципе, роли не играет. Они могут быть и регулярными, и иррегулярными (причем можно использовать и те, и другие одновременно) [1].

Это дает нам следующие преимущества:

- возможность представления огромнейших открытых пространств;
- кроме самих ландшафтов в таких блоках можно хранить и информацию о зданиях, строениях, растениях, специфических ландшафтных решениях (например, пещеры или скалы, нависающие друг над другом);
- возможность создания нескольких вариантов одного и того же сегмента, но при разной степени детализации; в зависимости от скорости или загруженности компьютера можно выбирать более или менее детализованные варианты.

Но мы также сталкиваемся со следующими проблемами:

- сложность стыковки разных сегментов;
- данные, представляющие ландшафт не тривиальны (сложно представить, как это будет выглядеть).

1.3 Выбор алгоритма генерации ландшафта

Существует множество различных подходов к генерации ландшафтов. Большинство из них основано на шумовых функциях, полигональных картах или физическом моделировании.

1.3.1 «Простой» алгоритм

Данный алгоритм можно разделить на следующие шаги:

- заполнение карты высот случайными значениями в определённом диапазоне;
- усреднение значений высот – для каждой высоты берутся её собственное значение и значения всех соседних вершин, а затем их сумма делится на их количество и присваивается текущей вершине, максимальное количество вершин для вычисления – 9.

Плюсом данного алгоритма является максимальная лёгкость понимания и простота реализации. Но данный алгоритм не даёт реалистичных результатов. Ландшафт, полученный с помощью него, выглядит ломанной равниной при маленьком диапазоне случайных чисел и как очень скалистые горы при большом. Можно улучшить вид ландшафта, применив к нему шаг 2 несколько раз, но потребует лишнее время при генерации ландшафта [1].

1.3.2 Шум Перлина

В бытовом смысле, «шум» — это случайный мусор. Шум является белым если все значение вычисленные с его помощью независимы друг от друга. Например, если мы будем генерировать с его помощью строчку из нулей и единиц, то в среднем их количество должно совпадать. Шум полезен для генерации случайных шаблонов, особенно для непредсказуемых природных явлений. Однако большинство вещей не чисто случайны. Дым, облака, ландшафт могут иметь некий элемент случайности, но они были созданы в результате очень сложных взаимодействий множества крохотных частиц.

Белый шум содержит независимые частицы. Для генерации ландшафта белый шум не подойдёт.

Шум Перлина – это градиентный шум, состоящий из набора псевдослучайных единичных векторов (направлений градиента), расположенных в определенных точках пространства и интерполированных функцией сглаживания между этими точками. В отличии от белого шума, шум Перлина непрерывный и плавный – нету резких переходов от маленьких значений к большим. Для генерации шума Перлина в одномерном пространстве необходимо для каждой точки этого пространства вычислить значение шумовой функции, используя направление градиента (или наклон) в указанной точке. Алгоритм шума Перлина можно масштабировать одно-, двух- и трёхмерного вида. Более того, в алгоритм можно ввести четвёртое временное измерение, позволяя алгоритму динамически изменять текстуры во времени.

Шум Перлина широко используется в двухмерной и трёхмерной компьютерной графике для создания таких визуальных эффектов, как дым, облака, туман, огонь и т. д. Он также очень часто используется как простая текстура, покрывающая геометрическую модель. В отличие от растровых текстур, шум Перлина является процедурной текстурой, и поэтому он не занимает память, но вместе с тем исполнение алгоритма требует неких вычислительных ресурсов.

1.3.3 Холмовой алгоритм (Hill Algoritm)

Это простой итерационный алгоритм, основанный на нескольких входных параметрах. Алгоритм изложен в следующих шагах:

- создаем двухмерный массив и инициализируем его нулевым уровнем (заполняем все ячейки нолями);
- берем случайную точку на ландшафте или около его границ (за границами), а также берем случайный радиус в заранее заданных

пределах; выбор этих пределов влияет на вид ландшафта — либо он будет пологим, либо скалистым;

- в выбранной точке "поднимаем" холм заданного радиуса;
- возвращаемся ко второму шагу и так далее до выбранного количества шагов, от него потом будет зависеть внешний вид нашего ландшафта;
- проводим нормализацию ландшафта;
- проводим "долинизацию" ландшафта, делаем его склоны более пологими.

Для получения реалистичных результатов с помощью этого алгоритма потребуется много вычислительных ресурсов и большинство ландшафтов будут похожи на горы или холмистую равнину.

1.3.4 Diamond square

Самым же распространенным и дающим одни из самых реалистичных результатов является алгоритм diamond-square (или square-diamond), расширение одномерного алгоритма midpoint displacement на двумерную плоскость.

Алгоритм midpoint displacement – рекурсивный. Изначально любым образом задаётся высота на концах отрезка и разбивается точкой посередине на два под-отрезка. Эту точку смещают на случайную величину и повторяют разбиение и смещение для каждого из полученных под-отрезков. И так далее — пока отрезки не станут длиной в один пиксель. Случайные смещения должны быть пропорциональны длинам отрезков, на которых производятся разбиения. Данный алгоритм можно использовать, например, для генерации линии горизонта.

В случае же алгоритма diamond-square, вычисления производятся в двумерном пространстве – карте высот. На вход подаётся плоская поверхность,

высота вершин которых равна нулю. Затем присваиваются значения к угловым высотам. После этого алгоритм можно разбить на два шага:

- шаг diamond – в нём вычисляется срединная точка текущего квадрата для квадрата, путём усреднения значений угловых вершин и добавлением случайного числа;
- шаг square – в нём вычисляются средние точки рёбер для текущего квадрата, путём усреднения вершин слева, справа, сверху и снизу и добавлением случайного числа. Если же какая-то из вершин выходит за границу карты высот, то такую точку можно либо не учитывать, либо считать равной нулю, благодаря чему, ближе к краям ландшафта он будет снижаться.

Далее карта высот делится на 4 меньших квадрата и шаги diamond, square повторяются для них, пока квадраты не выродятся в точку.

1.4 Выбор алгоритма удаления невидимых поверхностей

Задача удаления невидимых линий и поверхностей является одной из наиболее сложных в компьютерной графике. Алгоритмы удаления невидимых линий и поверхностей служат для определения линий ребер, поверхностей или объемов, которые видимы или невидимы для наблюдателя, находящегося в заданной точке пространства.

Сложность задачи удаления невидимых линий и поверхностей привела к появлению большого числа различных способов ее решения. Многие из них ориентированы на специализированные приложения. Наилучшего решения обшей задачи удаления невидимых линий и поверхностей не существует. Поэтому необходимо выбрать алгоритм, наиболее подходящий для нашей цели – визуализации трёхмерного ландшафта [3, с. 123].

1.4.1 Алгоритм плавающего горизонта

Алгоритм плавающего горизонта можно отнести к классу алгоритмов, работающих в пространстве изображения. Алгоритм плавающего горизонта чаше всего используется для удаления невидимых линий трехмерного представления функций, описывающих поверхность в виде $F(x, y, z) = 0$. Подобные функции возникают во многих приложениях в математике, технике, естественных науках и других дисциплинах [3, с.125].

Главная идея данного метода заключается в сведении трехмерной задачи к двумерной путем пересечения исходной поверхности последовательностью параллельных секущих плоскостей, имеющих постоянные значения координат x, y или z.

Можно выделить 2 основных этапа данного алгоритма:

- рассматриваемая поверхность рассекается плоскостями, перпендикулярными оси Z; в каждом отсечении получается кривая; эта кривая описывается уравнением $y = F(x, z = const)$ или $x = Q(x, z = const)$;
- полученные кривые можно проецировать на плоскость Oxy (то есть на плоскость z=0) и изобразить видимые части каждой кривой.

Изображение строится, начиная с кривой, полученной в ближайшем к наблюдателю сечении. Кривая, полученная в сечении ближайшей плоскостью, является видимой. Кривая, полученная во втором сечении, тоже будет видима. Связано это с тем, что вторая кривая расположена либо выше первой, либо ниже первой. В частом случае, когда они будут совпадать, будет получена одна кривая. Начиная с третьей кривой понадобится решать задачу определения видимости точек кривой.

В изложенном алгоритме предполагается, что значение функции, т. е. *y*, известно для каждого значения *x* в пространстве изображения. Однако если для

каждого значения *x* нельзя указать (вычислить) соответствующее ему значение y, то невозможно поддерживать массивы верхнего и нижнего плавающих горизонтов. В таком случае используется линейная интерполяция значений *y* между известными значениями для того, чтобы заполнить массивы верхнего и нижнего плавающих горизонтов.

Из минусов можно отметить следующий: если функция содержит очень острые участки (пики), то приведенный алгоритм также может дать некорректные результаты. Этот эффект вызван вычислением значений функции и оценкой ее видимости на участках, меньших, чем разрешающая способность экрана, т. е. тем, что функция задана слишком малым количеством точек. Если встречаются узкие участки, то функцию следует вычислять в большем числе точек.

Кроме этого, данный алгоритм подходит только для визуализации горизонтов, потому что при некоторых трёхмерных преобразованиях он начинает изображать поверхность некорректно (например, при повороте вокруг оси, смотрящей на наблюдателя, будут появляться «белые пятна»).

1.4.2 Алгоритм Робертса

Алгоритм Робертса представляет собой первое известное решение задачи об удалении невидимых линий. Это математически элегантный метод, работающий в объектном пространстве. Алгоритм прежде всего удаляет из каждого тела те ребра или грани, которые экранируются самим телом. Затем каждое из видимых ребер каждого тела сравнивается с каждым из оставшихся тел для определения того, какая его часть или части, если таковые есть, экранируются этими телами. Поэтому вычислительная трудоемкость алгоритма Робертса растет теоретически, как квадрат числа объектов. Это в сочетании с ростом интереса к растровым дисплеям, работающим в пространстве изображения, привело к снижению интереса к алгоритму Робертса. Однако математические методы, используемые в этом алгоритме, просты, мощны и

точны. Кроме того, этот алгоритм можно использовать для иллюстрации некоторых важных концепций. Наконец, более поздние реализации алгоритма, использующие предварительную приоритетную сортировку вдоль оси z и простые габаритные или минимаксные тесты, демонстрируют почти линейную зависимость от числа объектов [3, с. 130].

Работа Алгоритм Робертса проходит в два этапа:

- определение не лицевых граней для каждого тела отдельно;
- определение и удаление невидимых ребер.

1.4.3 Алгоритм использующий z-буффер

Алгоритм, использующий z-буфер это один из простейших алгоритмов удаления невидимых поверхностей. Работает этот алгоритм в пространстве изображения. Идея z-буфера является простым обобщением идеи о буфере кадра. Буфер кадра используется для запоминания атрибутов (интенсивности) каждого пиксела в пространстве изображения, а z-буфер - это отдельный буфер глубины, используемый для запоминания координаты z или глубины каждого видимого пиксела в пространстве изображения. В процессе работы глубина или значение z каждого нового пиксела, который нужно занести в буфер кадра, сравнивается с глубиной того пиксела, который уже занесен в z-буфер. Если это сравнение показывает, что новый пиксел расположен впереди пиксела, находящегося в буфере кадра, то новый пиксел 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) [3, с.137].

Главное преимущество алгоритма – его простота. Кроме того, этот алгоритм решает задачу об удалении невидимых поверхностей и делает тривиальной визуализацию пересечений сложных поверхностей. Сцены могут быть любой сложности. Поскольку габариты пространства изображения

фиксированы, оценка вычислительной трудоемкости алгоритма не более чем линейна. Поскольку элементы сцены или картинки можно заносить в буфер кадра или в z-буфер в произвольном порядке, их не нужно предварительно сортировать по приоритету глубины. Поэтому экономится вычислительное время, затрачиваемое на сортировку по глубине.

Основной недостаток алгоритма - большой объем требуемой памяти. Кроме этого, сложно устранить лестничный эффект или реализовать эффект прозрачности.

Вывод из аналитической части

Сравним принципы представления данных о ландшафте: поскольку для достижения нашей цели нет необходимости в представлении огромных ландшафтов, можно откинуть вариант посегментной карты высот. А между регулярной и иррегулярной картой высот, логично выбрать первый вариант, т.к. в данном случае, у нас не будет проблем с генерацией ландшафта, его освещением и хранением данных о нём.

Сравним алгоритмы генерации ландшафта: «Простой» способ является слишком тривиальным решением поставленной задачи, которое не даёт удовлетворявших результатов, поэтому его можно сразу исключить. Холмовой алгоритм, требует слишком много вычислительных ресурсов и выдаёт однообразные ландшафты. Шум Перлина выглядит интересным решением, но алгоритм diamond-square наиболее распространён и даёт одни из самых реалистичных результатов. Кроме того, я лично заинтересовался в его реализации. Следовательно, для генерации ландшафта я выбрал алгоритм diamond-square.

Сравним алгоритмы удаления невидимых поверхностей: алгоритм плавающего горизонта можно сразу отбросить потому, что кроме визуализации ландшафта, нам будет необходимо реализовать для него трёхмерные преобразования и освещение. А если выбирать между алгоритмом Робертса и z-

буфером, для реализации поставленной цели, по моему мнению, разумно выбрать второй вариант по следующим причинам: во-первых, алгоритм использующий z-buffer имеет линейную зависимость вычислительной трудоёмкости от кол-ва объектов, а при использовании алгоритма Робертса, для достижения зависимости приближённой к линейной, необходимо произвести предварительную приоритетную сортировку вдоль оси z и использовать простые габаритные или минимаксные тесты. Во-вторых, алгоритму z-буфера безразлично на сколько сложна визуализируемая сцена. Кроме этого, для поставленной задачи, первый этап алгоритма Робертса оказывается бессмысленным, т.к. он работает только для выпуклых многогранников, а мы работаем со случайно сгенерированным ландшафтом. Следовательно, для решения задачи удаления невидимых поверхностей, был выбран алгоритм, использующий z-buffer.

Таким образом, в данном разделе были формализованы объекты синтезируемой сцены, рассмотрены принципы представления данных о ландшафте, алгоритмы генерации ландшафта и удаления невидимых поверхностей. По описанным выше причинам карта высот была выбрана в качестве принципа представления данных о ландшафте, алгоритм DiamondSquare в качестве алгоритма генерации ландшафта и алгоритм ZBuffer в качестве алгоритма удаления невидимых поверхностей.

2. Конструкторская часть

В данном разделе описаны алгоритмы, выбранные для реализации, представлены их схемы и выбраны структуры данных.

2.1 Diamond-square

Как уже упоминалось выше, алгоритм diamond-square является расширением одномерного алгоритма midpoint displacement на двумерную плоскость. Кроме этого, было сказано, что случайные смещения должны быть пропорциональны длинам отрезков, на которых производятся разбиения. Например, мы разбиваем отрезок длиной l — тогда точка посередине него должна иметь высоту $h = \frac{h_L + h_R}{2} + random(-R * l, R * l)$, где ($h_L$ и h_R — высоты на левом и правом конце отрезка, а константа R определяет «шероховатость» (roughness) получающейся ломаной и является главным параметром в данном алгоритме).

При реализации алгоритма diamond-square необходимо чтобы размерность карты высот была $2^n + 1$ (где n – натуральное число) для того, чтобы на каждом шаге diamond была центральная вершина. Кроме этого, для улучшения изображения крайние вершины будут приравнены к нулю и не будут подвергаться изменениям. Следовательно, ландшафт будет снижаться к краям. Кроме этого, упоминалось случайно число, которое добавлялось к результату усреднения угловых или боковых точек. Данное число будет уменьшаться в 2 раза с каждым рекурсивным вызовом нашей функции (каждый раз, когда мы делим ландшафт на 4 меньших квадрата), в последствии чего разность высот между высотами вершин будет тем меньше, чем ближе они находятся друг к другу в карте высот. Во всём остальном данный алгоритм аналогичен вышеописанному. Также, для улучшения вида ландшафта, можно после генерации, можно применить шаг 2 из «Простого» алгоритма, в результате чего, поверхность станет, более «плавной».

На рисунках 2.1-2.3 представлена схема алгоритма DiamondSquare

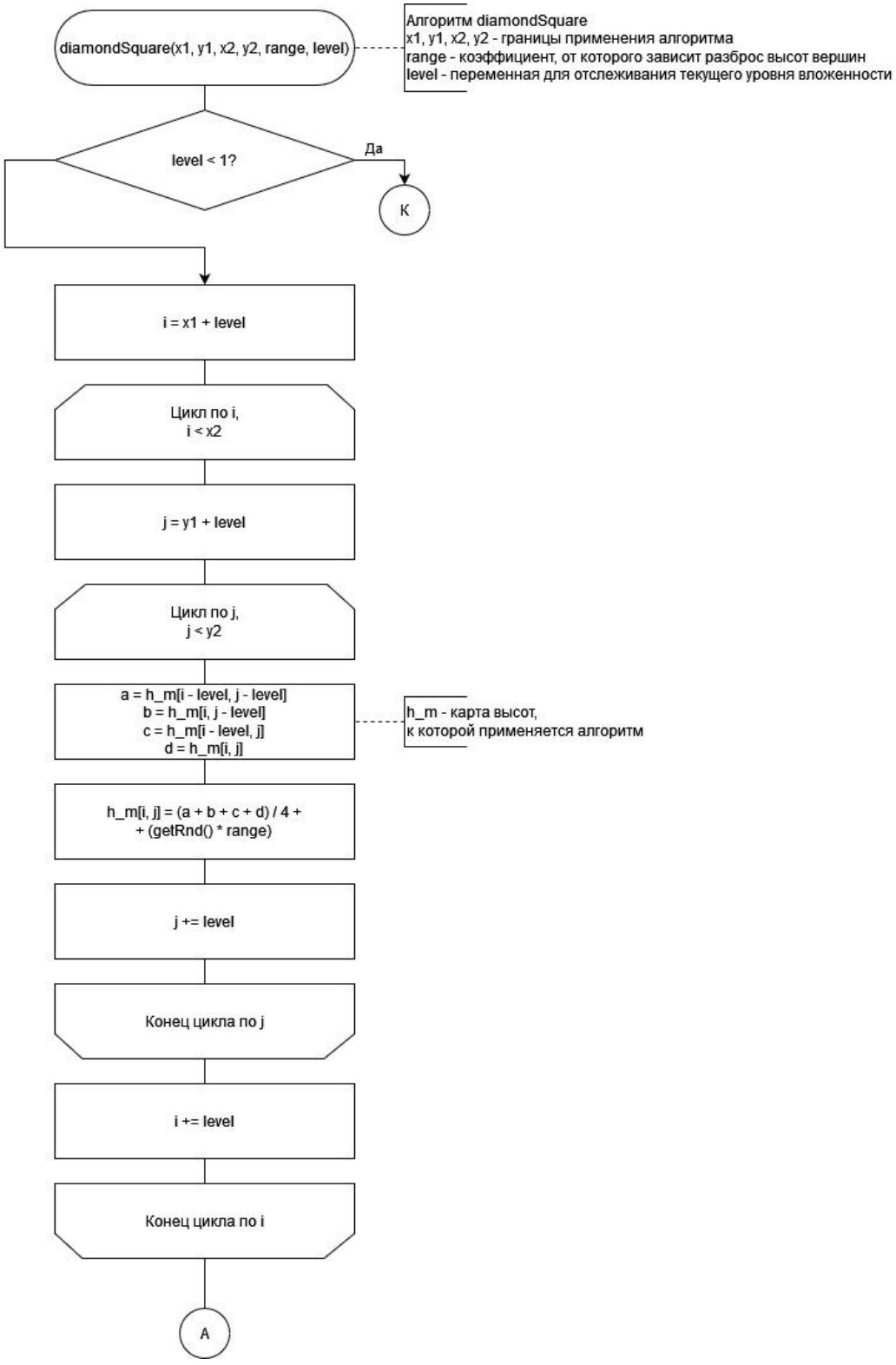

Рисунок 2.1 – Схема алгоритма DiamondSquare, часть 1 (Diamonds)

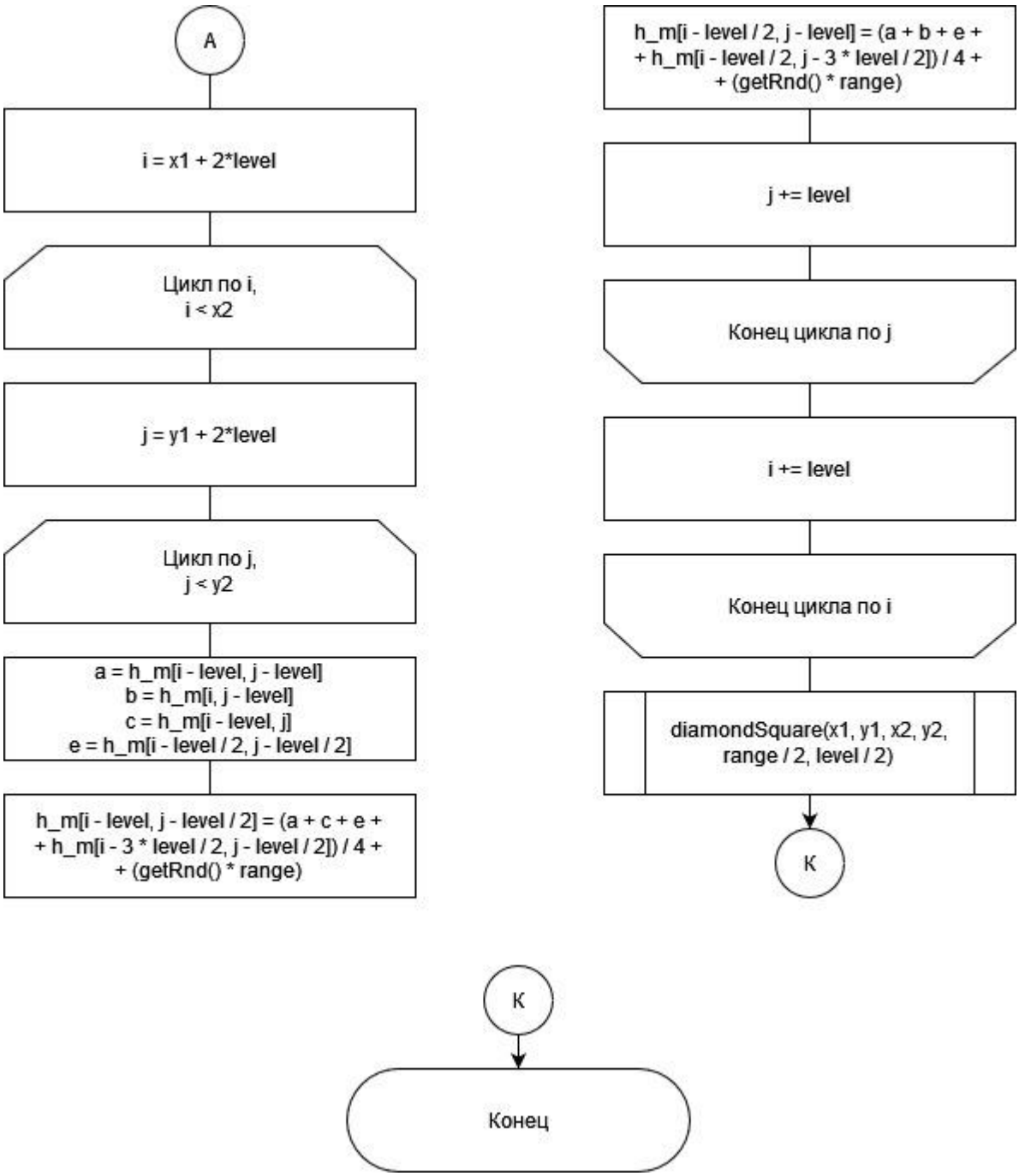

Рисунок 2.2 – Схема алгоритма DiamondSquare, часть 2 (Squares)

2.2 Z-буфер

Как уже упоминалось ранее, этот алгоритм работает в пространстве изображения. Поскольку для его реализации необходимо иметь возможность вычислять координату Z для каждой точки, было предложено следующее решение. Т.к. наш ландшафт состоит из треугольных полигонов, мы знаем только 3 его координаты. Для того чтобы вычислить Z для остальных точек, принадлежащих полигону, мы воспользуемся уравнением плоскости $Ax + By +$

$Cz + D = 0$. Для того, чтобы посчитать координату Z преобразуем данное уравнение к виду $Z = \frac{-Ax - Bx - D}{C}$. Если же коэффициент C окажется равным нулю, это означает что данная плоскость вырождается в линию для наблюдателя. В таком случае наша функция, возвращающая координату Z будет выдавать минимальное значение для используемого типа данных, чтобы не изображать таких полигонов. В любом случае, для того чтобы посчитать координату Z с помощью уравнения плоскости, необходимо сначала посчитать коэффициенты A, B, C и D для плоскости, в которой лежит данный треугольник. Для этого нам необходимо посчитать следующее равенство:

$$\begin{vmatrix} x - x_1 & y - y_1 & z - z_1 \\ x_1 - x_2 & y_1 - y_2 & z_1 - z_2 \\ x_2 - x_3 & y_2 - y_3 & z_2 - z_3 \end{vmatrix} = 0.$$

Оно, после вычисления определителя, представляет собой общее уравнение плоскости. После этого можно выразить все коэффициенты по отдельности:

$$A = y1 * (z2 - z3) + y2 * (z3 - z1) + y3 * (z1 - z2);$$

$$B = z1 * (x2 - x3) + z2 * (x3 - x1) + z3 * (x1 - x2);$$

$$C = x1 * (y2 - y3) + x2 * (y3 - y1) + x3 * (y1 - y2);$$

$$D = -\big(x1 * (y2 * z3 - y3 * z2) + x2 * (y3 * z1 - y1 * z3) + x3 * (y1 * z2 - y2 * z1)\big).$$

После вычисления уравнения плоскости, нужно определить находится данная точка в нашем треугольнике или нет. Данную задачу можно решать в двумерном пространстве, т.к. мы ищем зависимость $z(x, y)$. Чтобы избавиться от лишних вычислений, изначально поместим данный полигон в прямоугольник со сторонами $y_{max} - y_{min}$ и $x_{max} - x_{min}$ и будем проверять точки, только в этом прямоугольнике. Для того, чтобы определить находится ли точка внутри треугольника или нет, нужно определить видимость точки

относительно каждого ребра. Для этого оценивается знак скалярного произведения следующих векторов: внутренней нормали ребра (вектор направлен внутрь треугольника) и вектора который начинается в произвольной точке ребра, а заканчивается в рассматриваемой точке. Если результат больше или равен нулю, значит данная точка видима относительно текущего ребра. Если же скалярное произведение меньше нуля, то она находится снаружи. Соответственно для того, чтобы точка находилась внутри треугольника, условие видимости должно выполняться для каждого ребра.

Для данного подхода несложно вычислить вектор, который начинается в произвольной точке ребра, а заканчивается в рассматриваемой точке. Но необходимо также посчитать внутреннюю нормаль треугольника. Для этого воспользуемся свойством скалярного произведения – если скалярное произведение двух векторов равно нулю значит, что они перпендикулярны. Например, чтобы вычислить внутреннюю нормаль для очередной стороны возьмём вектор A, являющийся данной стороной => нам нужно найти вектор N перпендикулярный A. Для этого воспользуемся формулой скалярного произведения $A_x * N_x + A_y * N_y = 0$. Поскольку нам необходимо найти, только направление нормали, мы можем взять одну из проекций нормали равную 1. => $A_x + A_y * N_y = 0 => N_y = -\frac{A_x}{A_y}$. При этом не стоит пренебрегать перед вычислением выражения сверху проверкой на равенство A_y нулю, так как в таком случае требуется задать вектор $\{0, 1\}$ как вектор нормали заданной стороны. При этом, после выполнения вычислений, стоит проверить также, была определена внутренняя или внешняя нормаль (определяется по знаку скалярного произведения найденной нормали с вектором, заданным по следующей стороне), и во втором случае вернуть вектор, обратный найденному.

Также необходимо иметь возможность определять интенсивность конкретного полигона. Поскольку источник света всегда находится бесконечно

далеко над ландшафтом, а освещение считается диффузным, для вычисления интенсивности треугольного полигона достаточно посчитать косинус угла между лучом света и вектором нормали плоскости, в которой этот полигон находится. Значит, чем больше данный угол, тем меньше косинус, а, следовательно, и интенсивность. Вектор нормали можно получить, использовав коэффициенты уравнения плоскости, полученные ранее. Следовательно вектор нормали данной плоскости = $\overline{n}(A, B, C)$. А поскольку источник света находится бесконечно далеко, все лучи можно считать параллельными и направленными ровно вниз.

Теперь у нас есть все необходимые данные для определения координаты Z в точке (x, y) и её цвета. Полностью данный алгоритм записан ниже.

1) Посчитать для каждого треугольника y_{max}, y_{min}, x_{max} и x_{min}.
2) Посчитать для каждого треугольного полигона коэффициенты плоскости (A, B, C и D).
3) Посчитать для каждого полигона внутренние нормали, для определения видимости точек.
4) Заполнить буфер кадра фоновым значением цвета.
5) Заполнить z-буфер минимальным значением z.
6) Для каждого Пиксела в точке (x, y) в треугольнике вычислить его глубину z(x, y).
7) Сравнить глубину z(x, y) со значением Zбуфер(x, y), хранящимся в z-буфере в этой же позиции. Если z(x, y) > Zбуфер(x, y), то записать атрибут этого треугольного полигона (цвет) в буфер кадра и заменить Zбуфер(x, y) на z(x, y). В противном случае никаких действий не производить.

На рисунке 2.4 представлена схема алгоритма ZBuffer

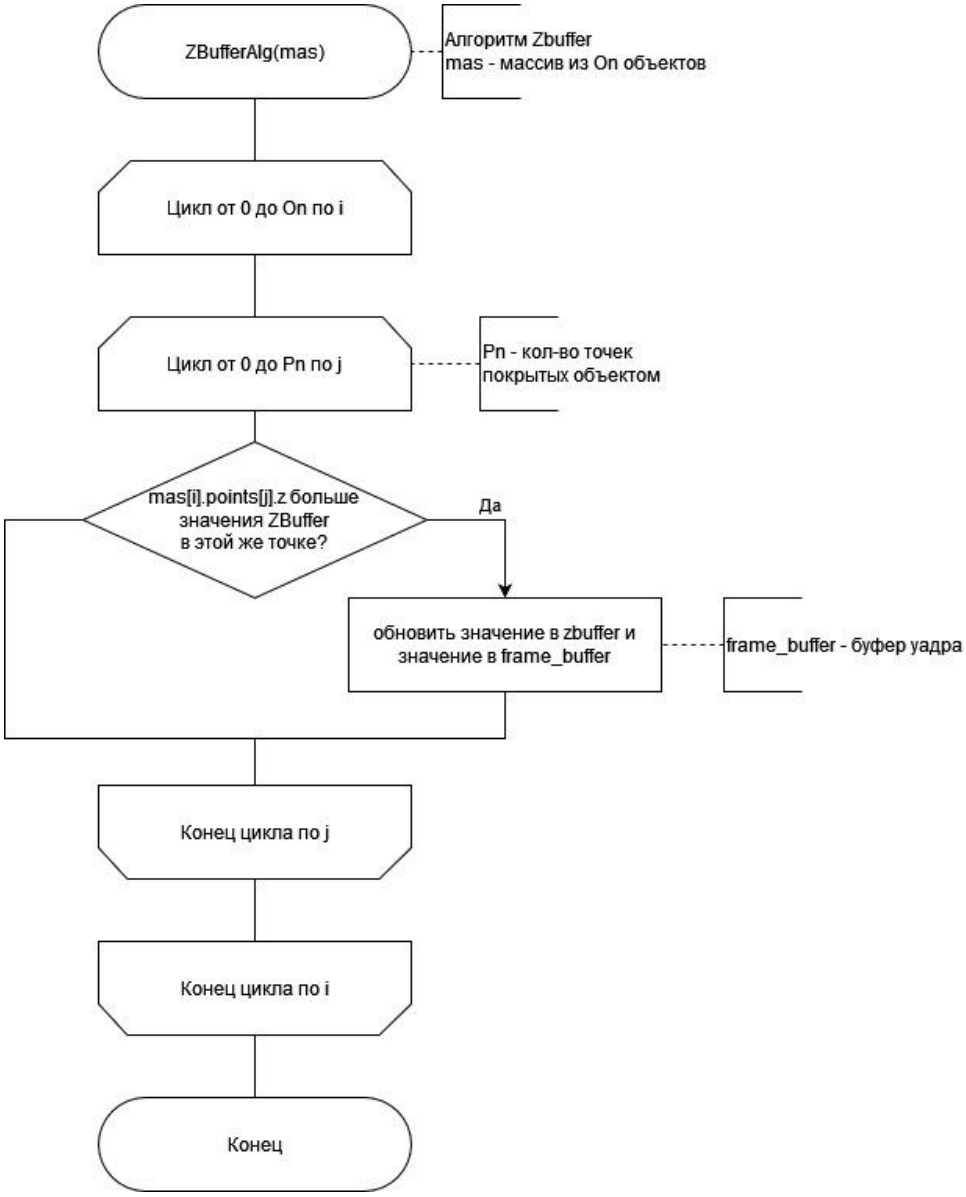

Рисунок 2.4 – Схема алгоритма ZBuffer

2.3 Выбор структур данных

Поскольку мы используем карту высот для представления ландшафта, разумно использовать матрицу со значениями каждой вершины, для генерации карты высот. Но поскольку нам необходимо реализовать трёхмерные преобразования для нашего ландшафта, нам нужно также знать не только высоту каждой вершины, но и две другие координаты, поэтому на основе сгенерированной карты высот будет создана матрица точек с координатами X, Y и Z.

Как было написано выше, для изображения ландшафта используются треугольные полигоны. Каждый полигон будет состоять из 3-ёх точек матрицы, которую мы получили ранее. Для того чтобы хранить эти треугольные полигоны используем массив.

На основе этих данных мы уже можем изобразить каркас нашего ландшафта, но нам необходимо также решить задачу удаления невидимых поверхностей. Поскольку мы выбрали алгоритм, использующий Z-buffer, нам необходимо создать еще две матрицы: буфер кадра и Z-буфер. Для реализации нашего алгоритма, также необходимо иметь возможность вычислять координаты Z для каждого X, Y наших полигонов, поэтому нужно вычислять коэффициенты плоскости для каждого полигона.

На рисунках 2.5-2.12 представлены диаграммы основных классов.

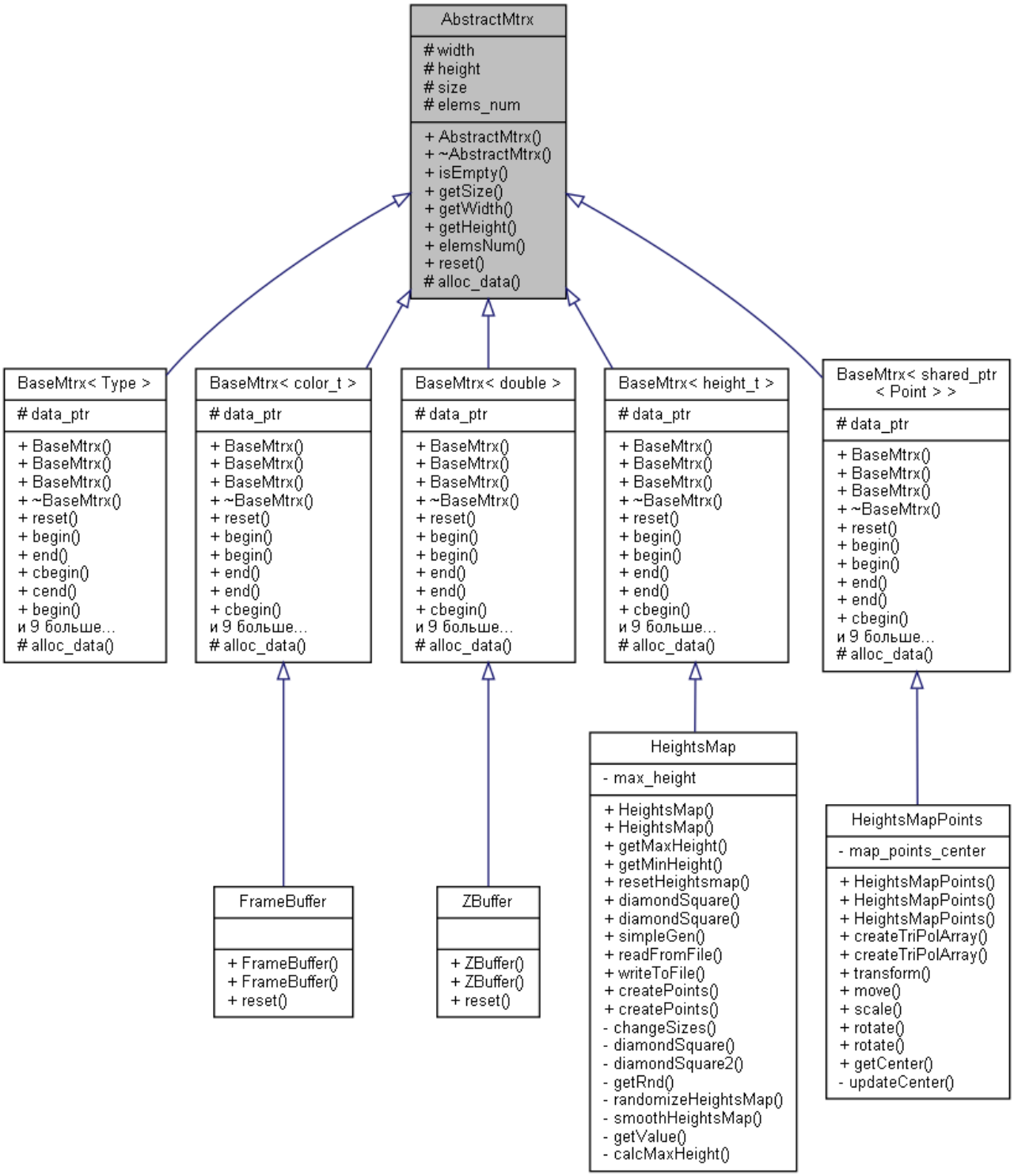

Рисунок 2.5 – Диаграмма класса AbstractMtrx и его потомков

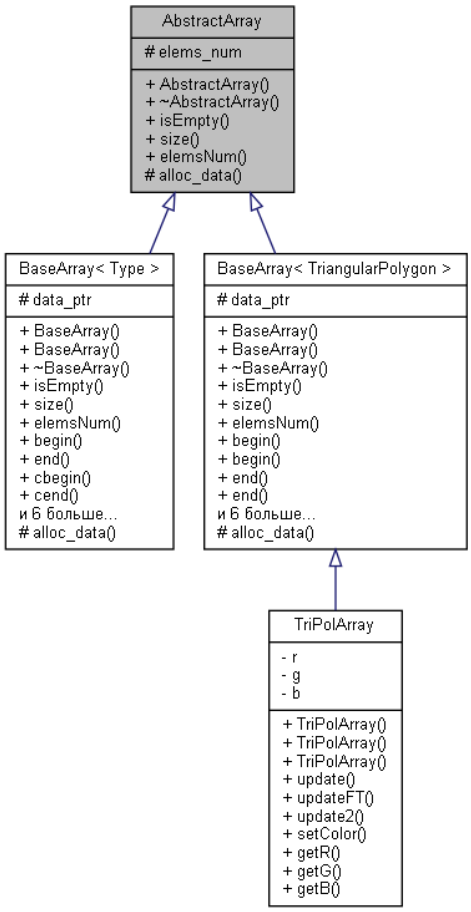

Рисунок 2.6 – Диаграмма класса AbstractArray и его потомков

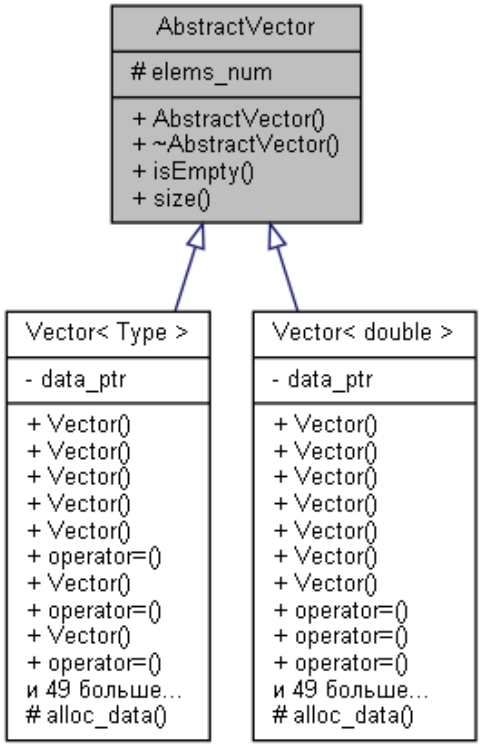

Рисунок 2.7 – Диаграмма класса AbstractVector и его потомков

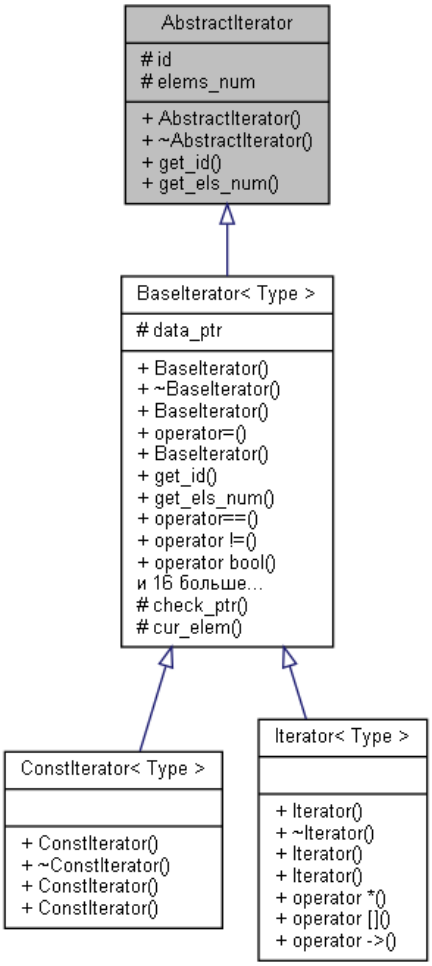

Рисунок 2.8 – Диаграмма класса AbstractIterator и его потомков

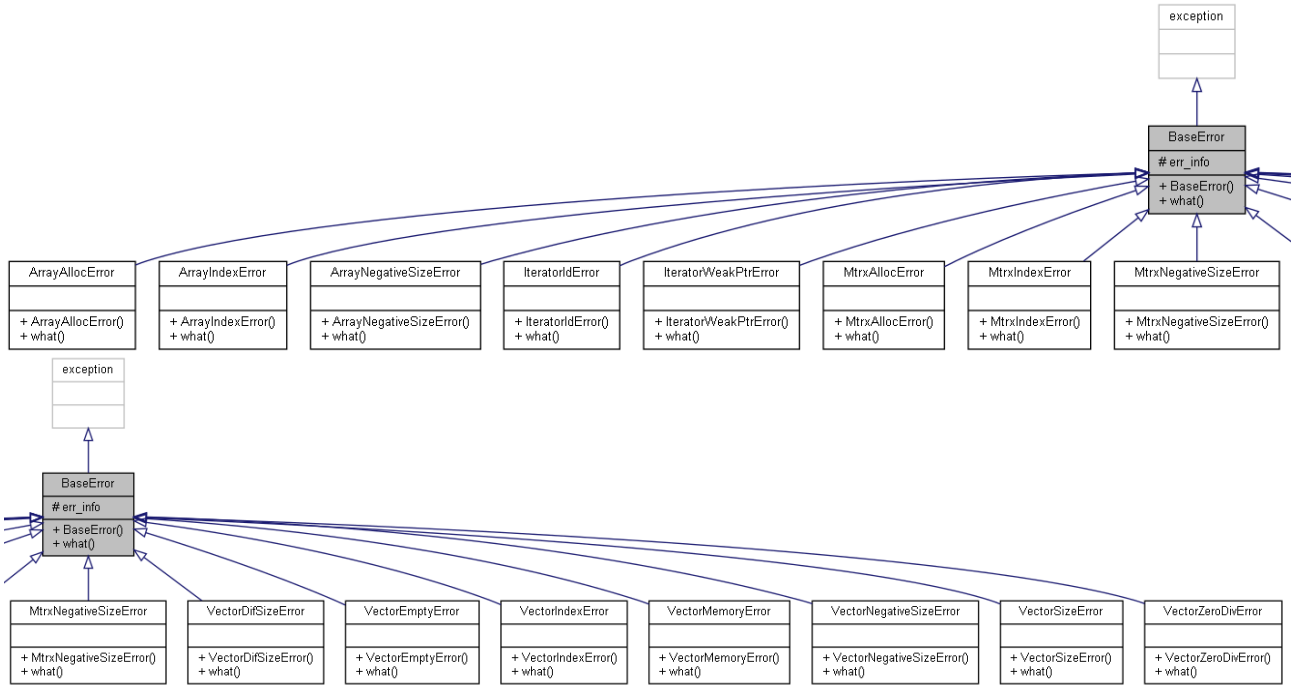

Рисунок 2.9 – Диаграмма класса BaseError и его потомков

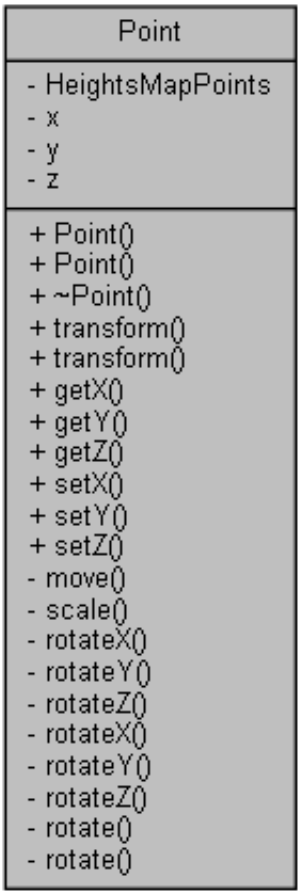

Рисунок 2.10 – Диаграмма класса Point

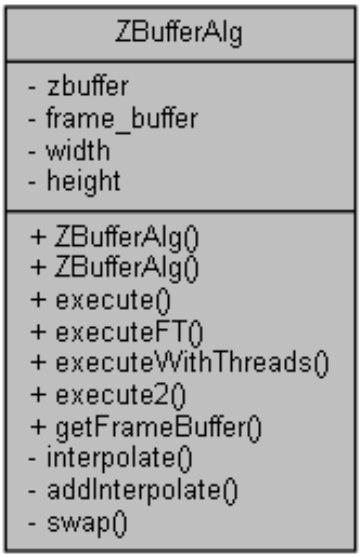

Рисунок 2.11 – Диаграмма класса ZBufferAlg

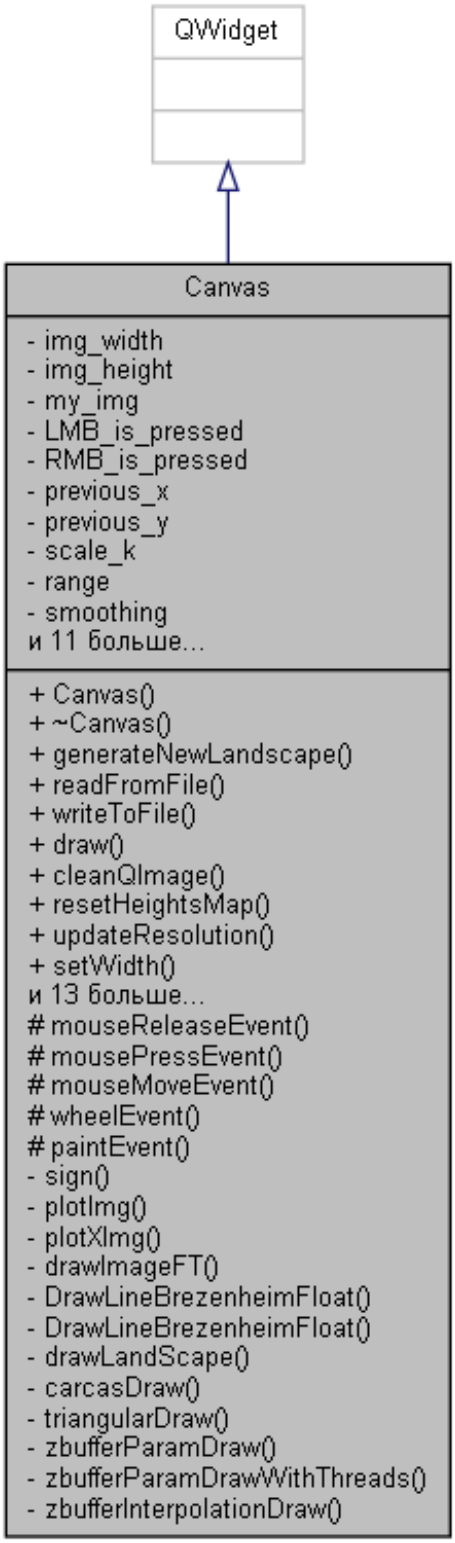

Рисунок 2.12 – Диаграмма класса Canvas

Вывод из конструкторской части

В данном разделе были описаны алгоритмы, выбранные для реализации, представлены их схемы и выбраны структуры данных.

3. Технологическая часть

В данном разделе выбраны инструменты для реализации проекта, представлены листинги кода, выбранных алгоритмов, показан интерфейс программы, и проведено тестирование разработанного ПО.

3.1 Основные инструменты, используемые для реализации и исследования

1. Язык программирования C++.

C++ – язык программирования общего назначения с уклоном в сторону системного программирования [4, с. 57].

Данный язык, достаточно популярен и широко распространён, кроме этого, он имеет ряд плюсов, описанных ниже.

1) Высокая производительность: язык спроектирован так, чтобы дать программисту максимальный контроль над всеми аспектами структуры и порядка исполнения программы. Один из базовых принципов C++ «не платишь за то, что не используешь» то есть ни одна из языковых возможностей, приводящая к дополнительным накладным расходам, не является обязательной для использования. Имеется возможность работы с памятью на низком уровне.

2) Кроссплатформенность: стандарт языка C++ накладывает минимальные требования на ЭВМ для запуска скомпилированных программ.

3) Поддержка различных стилей программирования: традиционное императивное программирование (структурное, объектно-ориентированное), обобщённое программирование, функциональное программирование, порождающее метапрограммирование.

Исходя из вышеперечисленных плюсов, очевиден выбор данного языка программирования для реализации поставленной задачи.

2. Кроссплатформенный фреймворк Qt.

Для реализации данного проекта, необходима была библиотека, упрощающая работу с графикой. На примете были Qt и MFC. Чтобы сделать выбор, пришлось их сравнить.

Qt – кроссплатформенный фреймворк для разработки программного обеспечения на языке программирования C++. Есть также «привязки» ко многим другим языкам программирования: Python — PyQt, PySide; Ruby — QtRuby; Java — Qt Jambi; PHP — PHP-Qt и другие. Поддерживаемые платформы включают Linux, OS X, Windows, VxWorks, QNX, Android, iOS, BlackBerry, ОС Sailfish и другие [5].

Qt позволяет запускать написанное с его помощью программное обеспечение в большинстве современных операционных систем путём простой компиляции программы для каждой системы без изменения исходного кода (кроссплатформенность). Включает в себя все основные классы, которые могут потребоваться при разработке прикладного программного обеспечения, начиная от элементов графического интерфейса и заканчивая классами для работы с сетью, базами данных и XML. Является полностью объектно-ориентированным, расширяемым и поддерживающим технику компонентного программирования.

Комплектуется визуальной средой разработки графического интерфейса Qt Designer, позволяющей создавать диалоги и формы.

Также существует возможность расширения привычной функциональности виджетов, связанной с размещением их на экране, отображением, перерисовкой при изменении размеров окна.

Мета-объектная система — часть ядра фреймворка для поддержки в C++ таких возможностей, как сигналы и слоты для коммуникации между объектами в режиме реального времени и динамических свойств системы

Одним из преимуществ проекта Qt является наличие качественной документации. Статьи документации снабжены большим количеством примеров. Исходный код самой библиотеки хорошо форматирован, подробно комментирован, что также упрощает изучение Qt.

Отличительная особенность — использование мета-объектного компилятора — предварительной системы обработки исходного кода. Расширение возможностей обеспечивается системой плагинов, которые возможно размещать непосредственно в панели визуального редактора. Но минусом получается то, что код написанный с помощью Qt нельзя скомпилировать на другом компьютере без установки фреймворка.

Microsoft Foundation Classes – библиотека на языке C++, разработанная Microsoft и призванная облегчить разработку GUI-приложений для Microsoft Windows путём использования богатого набора библиотечных классов.

Во-первых, если сравнивать только работу с GUI, то данная библиотека работает только под Windows, то есть ни о какой кроссплатформенности речи не идёт. Но не стоит забывать о том, что Qt в отличии от MFC имеет множество других полезных классов. Во-вторых, если же в MFC создать каркас приложения без дизайнера достаточно сложно, то в Qt это зачастую даже намного удобнее и проще.

Поскольку функционал Qt намного шире, то для реализации проекта был выбран именно фреймворк Qt.

3. Среда разработки Qt creator.

Qt Creator (ранее известная под кодовым названием Greenhouse) — кроссплатформенная свободная IDE для языков C, C++ и QML. Разработана

Trolltech (Digia) для работы с фреймворком Qt. Включает в себя графический интерфейс отладчика и визуальные средства разработки интерфейса как с использованием QtWidgets, так и QML. Поддерживаемые компиляторы: GCC, Clang, MinGW, MSVC, Linux ICC, GCCE, RVCT, WINSCW.

Основная задача Qt Creator — упростить разработку приложения с помощью фреймворка Qt на разных платформах. Поэтому для работы с данной библиотекой был выбран именно он.

4. Система версионного контроля git.

Для хранения исходников используется система Git (на портале github.com), т.к. это крупнейший веб-сервис для хостинга IT-проектов и их совместной разработки.

5. Библиотека clock().

Чтобы оценить время выполнения программы будет замеряться реальное время, т.к. таким образом можно будет сравнить реализации алгоритма ZBuffer с использованием параллельных вычислений и без. Для замера реального времени работы программы используется функция clock() т.к. программа тестируется на компьютере с установленной ОС Windows [6].

3.2 Реализация

На листингах 3.1-3.3 представлены реализации следующих алгоритмов:

- реализация алгоритма diamond-square на C++ (Листинг 1);
- реализация алгоритма, использующего z-буфер (Листинг 2);
- методы вычисление внутренних нормалей, коэффициентов плоскости и интенсивности цвета, для полигона (Листинг 3).

Листинг 3.1 – Алгоритм diamond-square

```
 1 void HeightsMap::diamondSquare(unsigned x1, unsigned y1, unsigned x2,
 2                                unsigned y2, float range, unsigned level)
 3 {
 4     if (level < 1)
 5         return;
 6
 7     // diamonds
 8     for (unsigned i = x1 + level; i < x2; i += level)
 9         for (unsigned j = y1 + level; j < y2; j += level)
10         {
11             float a = (*this)(i - level, j - level);
12             float b = (*this)(i, j - level);
13             float c = (*this)(i - level, j);
14             float d = (*this)(i, j);
15
16             float e = (*this)(i - level / 2, j - level / 2)
17                     = (a + b + c + d) / 4 + (getRnd() * range);
18         }
19
20
21
22
23     // squares
24     for (unsigned i = x1 + 2 * level; i < x2; i += level)
25         for (unsigned j = y1 + 2 * level; j < y2; j += level)
26         {
27             float a = (*this)(i - level, j - level);
28             float b = (*this)(i, j - level);
29             float c = (*this)(i - level, j);
30             float e = (*this)(i - level / 2, j - level / 2);
31
32             float f = (*this)(i - level, j - level / 2)
33                     = (a + c + e
34                        + (*this)(i - 3 * level / 2, j - level / 2)) / 4
35                        + (getRnd() * range);
36             float g = (*this)(i - level / 2, j - level)
37                     = (a + b + e
38                        + (*this)(i - level / 2, j - 3 * level / 2)) / 4
39                        + (getRnd() * range);
40         }
41
42     diamondSquare(x1, y1, x2, y2, range / 2, level / 2);
43 }
```

Листинг 3.2 – Алгоритм, использующий z-буфер

```
void ZBufferAlg::execute(TriPolArray &mas)
{
    int red = mas.getR();
    int green = mas.getG();
    int blue = mas.getB();
    zbuffer->reset();
    frame_buffer->reset();
    for (auto& elem : mas)
    {
        for (int i = max(elem.getMinX(), 0.);
             i < min(elem.getMaxX(), double(height));
             i++)
        {
            for (int j = max(elem.getMinY(), 0.);
                 j < min(elem.getMaxY(), double(width));
                 j++)
            {
                if (elem.isInTriangle(i, j))
                {
                    if ((*zbuffer)(i, j) < elem.getZ(i, j))
                    {
                        (*zbuffer)(i, j) = elem.getZ(i, j);
                        double intensivity = elem.getIntensity();

                        (*frame_buffer)(i, j)
                        = QColor(round(red * intensivity),
                                 round(green * intensivity),
                                 round(blue * intensivity));
                    }
                }
            }
        }
    }
}
```

Листинг 3.3 – Методы calcNormals(), calcSurface() и calcIntensity()

```
 1 void TriangularPolygon::calcNormals()
 2 {
 3     double vec1_x = p2->getX() - p1->getX(),
 4            vec1_y = p2->getY() - p1->getY();
 5     double vec2_x = p3->getX() - p2->getX(),
 6            vec2_y = p3->getY() - p2->getY();
 7     double vec3_x = p1->getX() - p3->getX(),
 8            vec3_y = p1->getY() - p3->getY();
 9     if (vec1_y != 0)
10         norm_vec1 = {1, -vec1_x/vec1_y};
11     else
12         norm_vec1 = {0, 1};
13     if (norm_vec1[0]*vec2_x + norm_vec1[1]*vec2_y < 0)
14         norm_vec1 = norm_vec1*(-1.);
15     if (vec2_y != 0)
16         norm_vec2 = {1, -vec2_x/vec2_y};
17     else
18         norm_vec2 = {0, 1};
19     if (norm_vec2[0]*vec3_x + norm_vec2[1]*vec3_y < 0)
20         norm_vec2 = norm_vec2*(-1.);
21     if (vec3_y != 0)
22         norm_vec3 = {1, -vec3_x/vec3_y};
23     else
24         norm_vec3 = {0, 1};
25     if (norm_vec3[0]*vec1_x + norm_vec3[1]*vec1_y < 0)
26         norm_vec3 = norm_vec3*(-1.);
27 }
28
29 void TriangularPolygon::calcSurface()
30 {
31     double x1 = p1->getX(), y1 = p1->getY(), z1 = p1->getZ();
32     double x2 = p2->getX(), y2 = p2->getY(), z2 = p2->getZ();
33     double x3 = p3->getX(), y3 = p3->getY(), z3 = p3->getZ();
34     A = y1 *(z2 - z3) + y2 *(z3 - z1) + y3 *(z1 - z2);
35     B = z1 *(x2 - x3) + z2 *(x3 - x1) + z3 *(x1 - x2);
36     C = x1 *(y2 - y3) + x2 *(y3 - y1) + x3 *(y1 - y2);
37     D = -(x1 * (y2 * z3 - y3 * z2)
38         + x2 * (y3 * z1 - y1 * z3)
39         + x3 * (y1 * z2 - y2 * z1));
40 }
41
42 void TriangularPolygon::calcIntensity()
43 {
44     if (A != 0 || B != 0 || C != 0)
45         intensity = fabs(B/sqrt(A*A+B*B+C*C));
46     else
47         intensity = 1;
48 }
```

3.3 Интерфейс программы

Интерфейс программы представлен на рисунке 3.1.

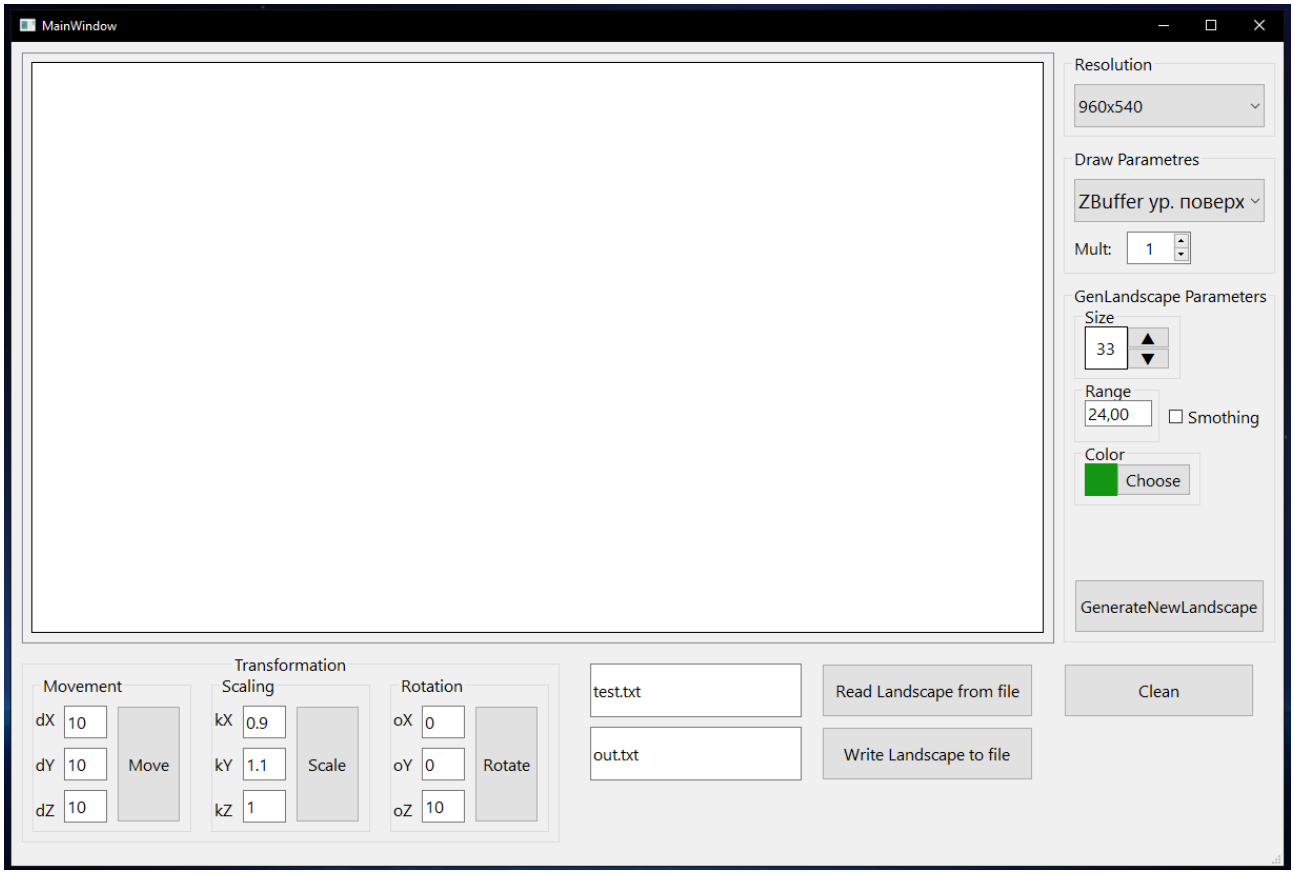

Рисунок 3.1 – Интерфейс

3.4 Тестирование

Для проверки написанных алгоритмов были подготовлены следующие тесты:

- проверка алгоритма DiamondSquare и ZBuffer;
- проверка трёхмерного переноса;
- проверка трёхмерного масштабирования;
- проверка трёхмерного поворота;
- проверка сохранения в файл;
- проверка загрузки из файла.

На рисунках 3.2-3.7 приведены результаты тестирования.

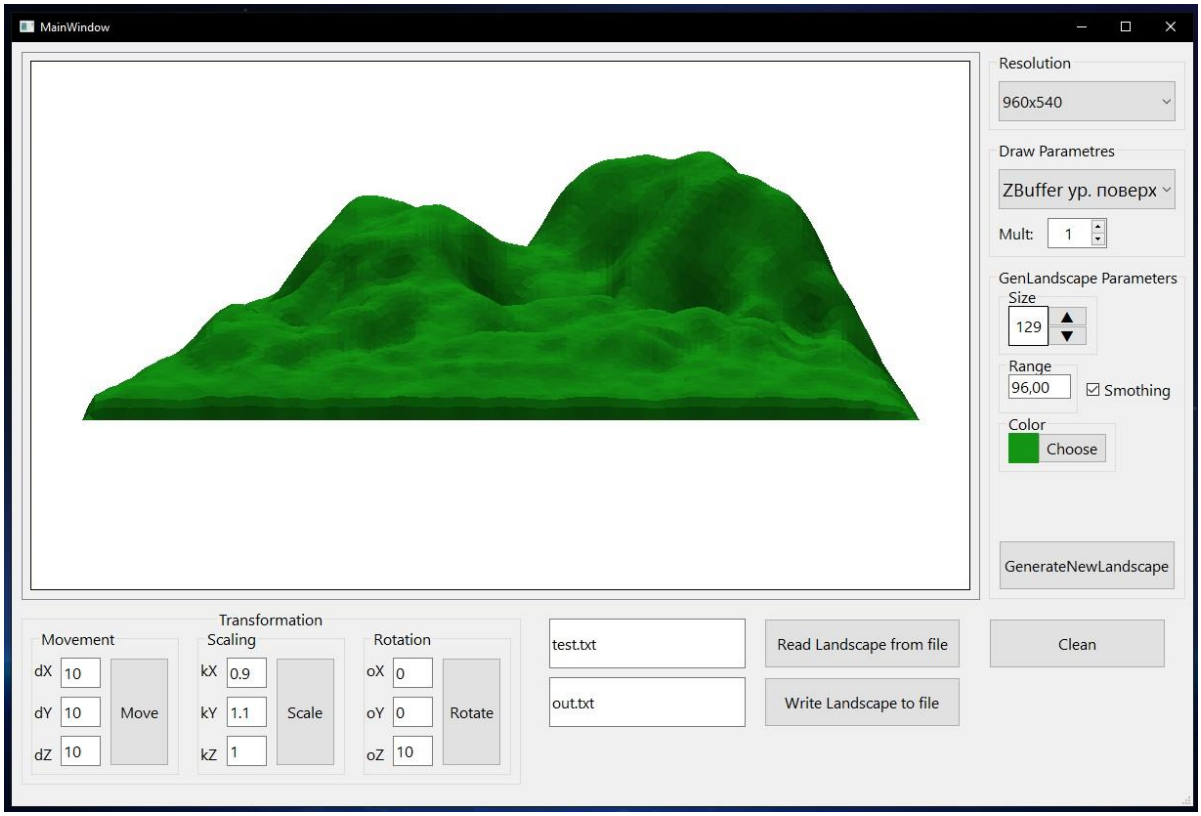

Рисунок 3.2 – Начальное положение ландшафта

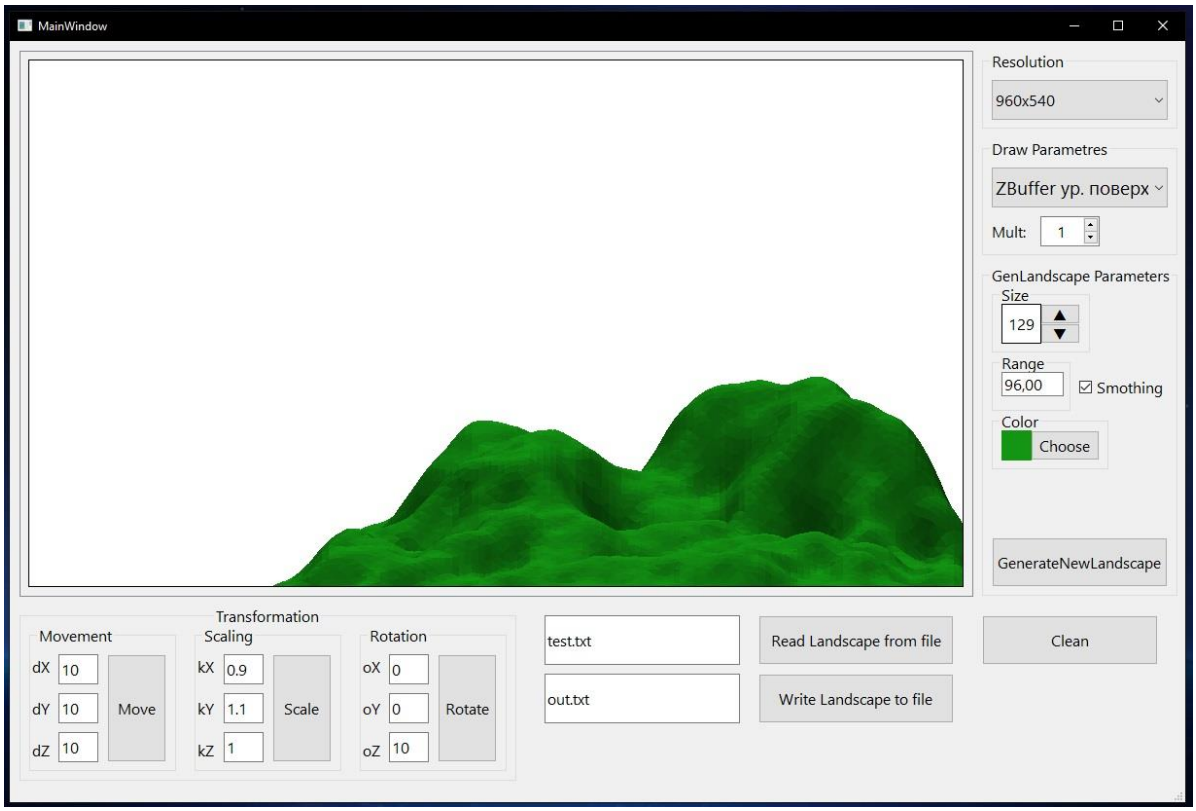

Рисунок 3.3 – Трёхмерный перенос ландшафта

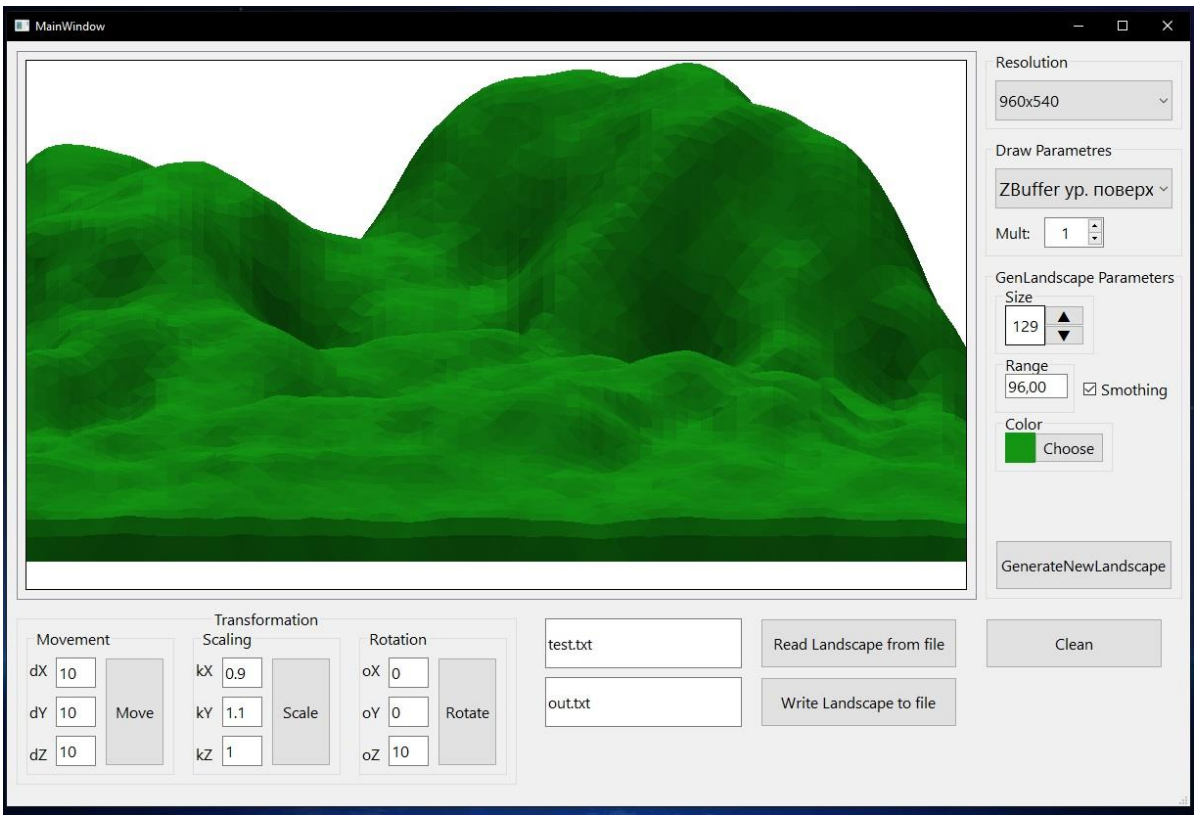

Рисунок 3.4 – Трёхмерное масштабирование ландшафта

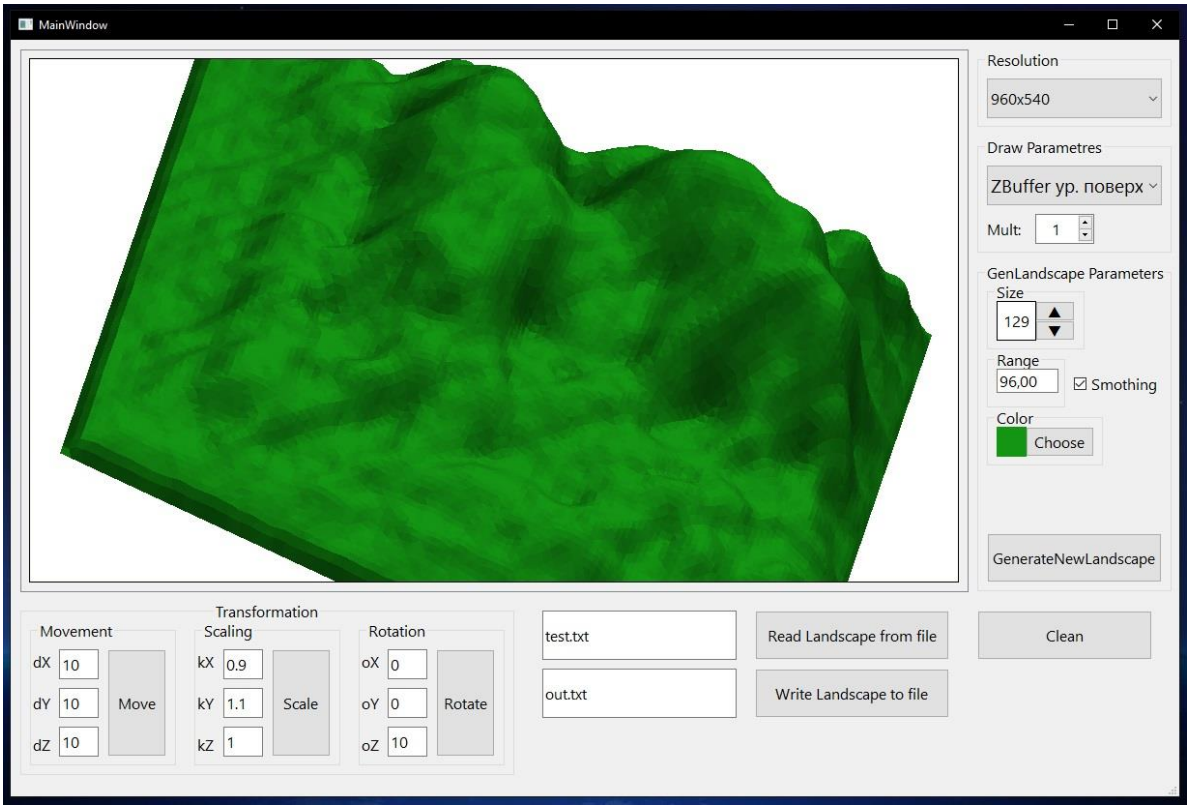

Рисунок 3.5 – Трёхмерный поворот ландшафта

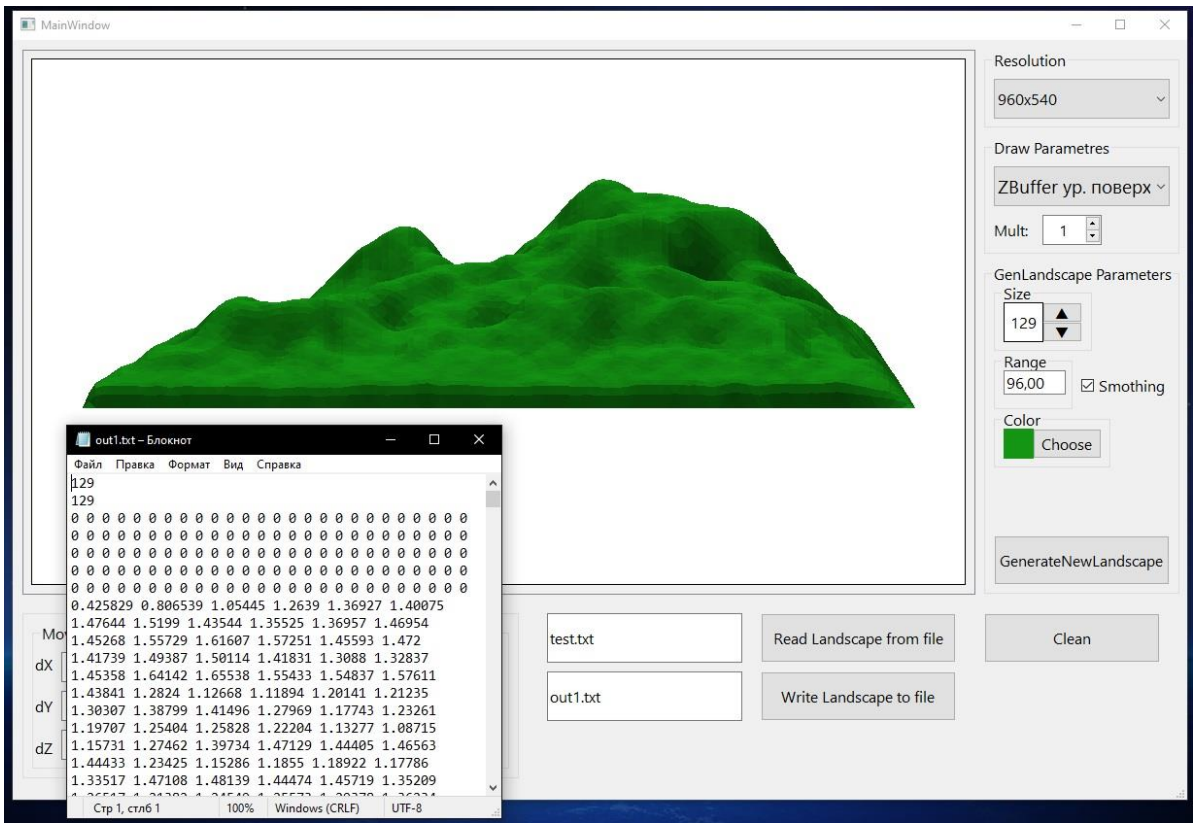

Рисунок 3.6 – Запись в файл

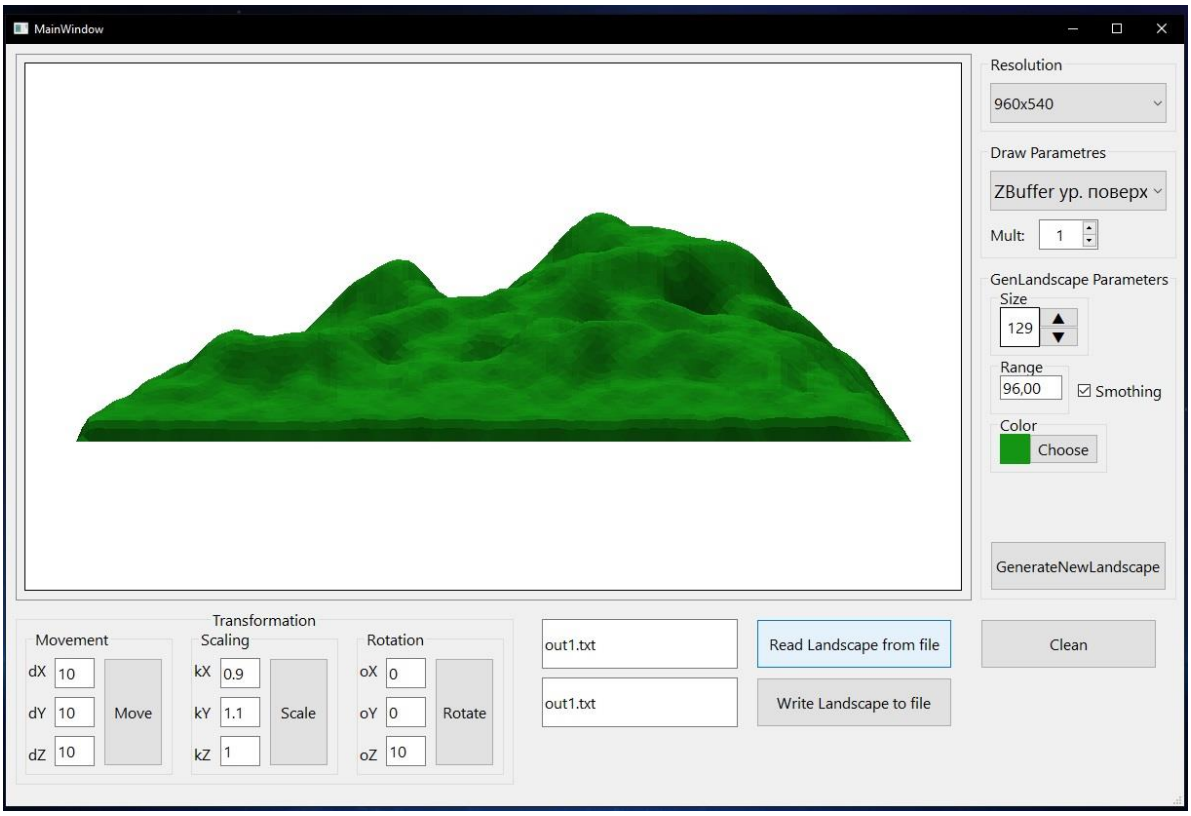

Рисунок 3.7 – Чтение из файла

Как видно по рисункам, все тесты пройдены.

Вывод из технологической части

В данном разделе были выбраны инструменты для реализации проекта, представлены листинги кода, выбранных алгоритмов, показан интерфейс программы, а также проведено тестирование разработанного ПО.

4. Исследовательская часть

В данном разделе представлены примеры работы программы, проведён сравнительный анализ времени выполнения алгоритма ZBuffer с использованием параллельных вычислений и без, а также произведена оценка времени выполнения алгоритма diamond-square в зависимости от размера карты высот.

4.1 Пример работы программы

Результат работы программы представлен на рисунках 4.1-4.2.

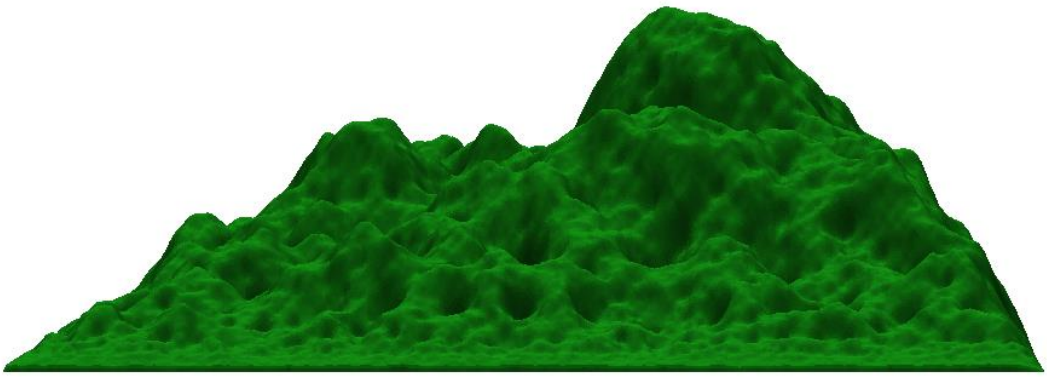

Рисунок 4.2 – Результат работы программы, часть 1

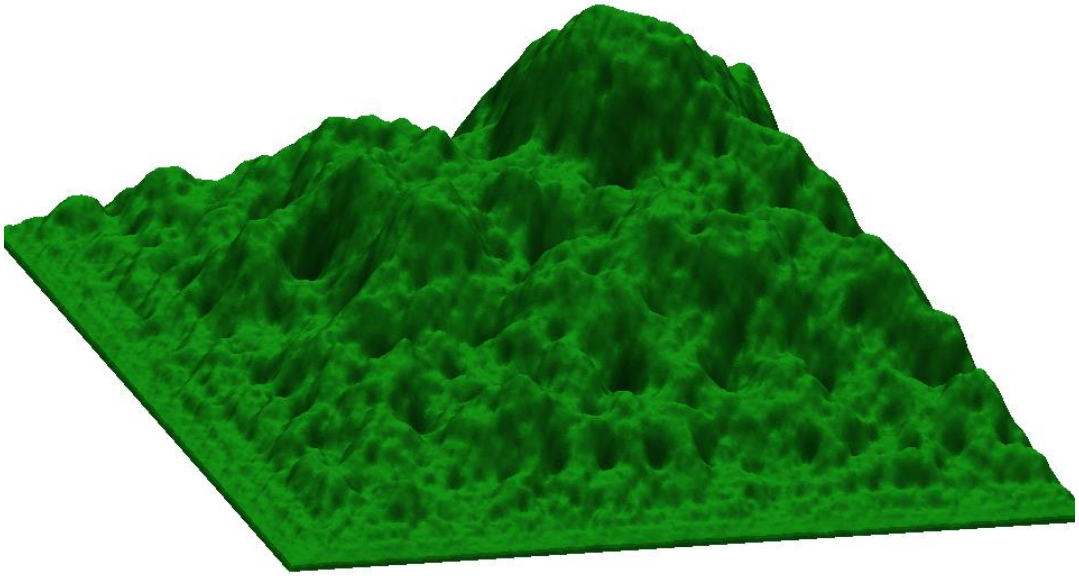

Рисунок 4.3 – Результат работы программы, часть 2

4.2 Технические характеристики

Ниже приведены технические характеристики устройства, на котором были проведены эксперименты при помощи разработанного ПО:

- операционная система: Windows 10 (64-разрядная);
- оперативная память: 32 GB;
- процессор: Intel(R) Core(TM) i7-7700K CPU @ 4.20GHz;
- количество ядер: 4;
- количество потоков: 8.

4.3 Сравнительный анализ времени выполнения алгоритма ZBuffer с использованием параллельных вычислений и без.

Чтобы провести сравнительный анализ времени выполнения алгоритмов, замерялось реальное время работы алгоритма ZBuffer 100 раз и делилось на кол-во итераций. В таблице 4.1 показаны результаты тестирования алгоритма, использующего параллельные вычисления, для разного кол-ва потоков.

Таблица 4.1 – Время выполнения алгоритма ZBuffer с использованием параллельных вычислений, при размере карты высот 33x33

Количество потоков, шт	Время выполнения, сек
1	0.12207
2	0.10223
4	0.08782
8	0.08062
16	0.08563
32	0.09113

Из таблицы выше видно, что наибольший выигрыш по времени даёт алгоритм, использующий 8 потоков, поэтому для сравнения с обычным

алгоритмом будем использовать именно его. В таблице 4.2 приведены результаты этого сравнения.

Таблица 4.2 – Время выполнения обычного ZBuffer и с использованием параллельных вычислений на 8 потоках

Размер карты высот, шт	Без потоков, сек	С 8ю потоками, сек
33x33	0.12056	0.08062
65x65	0.12524	0.08262
129x129	0.15733	0.08707
257x257	0.21563	0.09875
513x513	0.35673	0.12993

Смотря на результаты сравнительного анализа времени выполнения обычного и многопоточного алгоритмов ZBuffer, логично сделать вывод, что наиболее быстрым, является алгоритм, использующий параллельные вычисления.

Оценка времени выполнения алгоритма diamond-square в зависимости от размера.

Чтобы оценить время выполнения алгоритма diamond-square в зависимости от размера, замерялось реальное время для карт высот следующих размеров: 33x33, 65x65, 129x129, 257x257, 513x513, 1025x1025. Алгоритм выполнялся $1025^2 * 10/N^2$ раз, где N – размер стороны карты высот ландшафта. В таблице 4.3 показаны результаты эксперимента. Шаг “Smoothing” это шаг 2 из “Простого алгоритма”, описанный ранее.

Таблица 4.3 – Время выполнения алгоритма diamond-square

Размер карты высот, шт	Время выполнения без шага “Smoothing”, сек.	Время выполнения с шагом “Smoothing”, сек.
33x33	0.000564113	0.000614388
65x65	0.0023391	0.00258608
129x129	0.00980666	0.0109905
257x257	0.0398239	0.043239
513x513	0.157564	0.179538
1024x1024	0.6415	0.7194

Как и ожидалось, с ростом размера карты высот будет увеличиваться и время выполнения алгоритма diamond-square. Кроме этого, время выполнения алгоритма увеличивается на ~11% если выполнять шаг “Smoothing”.

Вывод из исследовательской части

В данном разделе были представлены примеры работы программы, проведён сравнительный анализ времени выполнения алгоритма ZBuffer с использованием параллельных вычислений и без, а также произведена оценка времени выполнения алгоритма diamond-square в зависимости от размера карты высот.

Заключение

По итогу проделанной работы была достигнута цель – разработана программа генерации и визуализации трехмерного изображения.

Также были решены все поставленные задачи, а именно:

- проанализированы представления данных о ландшафте;
- проанализированы алгоритмы генерации ландшафта;
- проанализированы алгоритмы удаления невидимых поверхностей;
- выбраны необходимые структуры данных для изображения ландшафта;
- выбраны основные инструменты для разработки программы;
- разработана программа, реализующая поставленную задачу;
- проведён сравнительный анализ времени выполнения алгоритма ZBuffer с использованием параллельных вычислений и без;
- произведена оценка времени выполнения алгоритма diamond-square в зависимости от размера карты высот.

Список Литературы

1. Кожухов Д. Генерация трехмерных ландшафтов: [Электронный ресурс] // URL: https://www.ixbt.com/video/3dterrains-generation.shtml (дата обращения: 12.09.21)
2. Роджерс Д. Алгоритмические основы машинной графики. Москва: Мир, 1989. 512 с.
3. Дёмин А. Ю., Кудинов А. В. Компьютерная графика: учеб. пособие / Том. политехн. ун-т. Томск, 2005. 160с.
4. Бьёрн Страуструп Язык программирования C++ - специальное издание. Москва: Бином, 2010. 1136 с.
5. Qt документация [Электронный ресурс]: http://doc.qt.io/ (дата обращения: 19.08.21)
6. MSDN – библиотека <clock> [Электронный ресурс] //Техническая документация Майкрософт. URL: https://docs.microsoft.com/ru-ru/cpp/c-runtime-library/reference/clock?view=msvc-160&viewFallbackFrom=vs-2019 (дата обращения: 20.10.21)