Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н. Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н. Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»
КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА К КУРСОВОЙ РАБОТЕ НА ТЕМУ:

«Визуализация лесистой местности»

Студент Княжев Алексей Викторович
фамилия, имя, отчество

Студент ИУ7-52Б _____ Княжев А. В.
группа *подпись, дата* *фамилия, и. о.*

Руководитель курсовой работы _____ Кострицкий А. С.
подпись, дата *фамилия, и. о.*

Москва — 2022 г.

Оглавление

Введение

Компьютерная графика является важной частью современного мира. Визуальное отображение информации помогает показывать объекты, которых не существует в реальном мире. Данная наука находит свое применение в сфере компьютерных игр, фильмов, бизнеса, развлечений, медицины. Благодаря достижениям компьютерной графики, сейчас становится возможным отображать объекты настолько реалистично, что становится трудно различить отрисованные компьютерами объекты от фотографий [1].

Одной из важных задач современной компьютерной графики является задача генерации местности: ландшафта и растительности.

Цель работы

Разработать программу для построения трёхмерного изображения лесистой местности.

Задачи работы

— изучить алгоритмы генерации местности;

— изучить методы преобразования сгенерированной местности в модель для построения;

— изучить и сравнить алгоритмы построения реалистичных изображений;

— формализовать объекты сцены;

— выбрать средства разработки, позволяющие решить поставленные задачи;

— реализовать данные алгоритмы.

1. Аналитическая часть

В данном разделе рассмотрены анализ задач отображения объектов, построения реалистичных изображений, генерации местности. Кроме того, рассматриваются методы решения вышеуказанных задач.

1.1. Описание объектов

1.1.1. Типы моделей

Существует несколько типов геометрической модели тела. Ниже представлены их описание и сравнение.

Каркасная модель

Каркасная модель является наиболее примитивным типом модели тела. Представляет собой набор вершин и ребер, без учета материала и характеристик граней/поверхностей тела. Получаемые в результате применения каркасных моделей изображения неоднозначны: нет возможности автоматизировать процессы удаления невидимых линий [10].

Поверхностная модель

Поверхностная модель, в отличие от каркасной, позволяет описывать поверхности тела, в том числе аналитическим способом, но не учитывает направление поверхности. Таким образом, не производится учет внешних/внутренних поверхностей тела.

Объемная модель

Объемная модель является модификацией поверхностной модели с тем отличием, что предоставляет информацию о направлении поверхности и расположении материала относительно поверхностей. Таким образом, становится возможным учет внутренней структуры тела. Объемные модели позволяют описывать объекты с большой степенью точности.

Выбор типа модели

В результате анализа и сравнения типов моделей, было решено использовать поверхностную модель, так как она позволяет оперировать поверхностями и текстурами объекта,

при этом в данной задаче нет необходимости в анализе внутренней структуры тел.

1.1.2. Способы задания модели

Аналитический

Модель задается с помощью аналитический выражений (уравнений) [4] вида

$$f(x, y, z) = 0. \tag{1.1}$$

Кроме явного уравнения, часто используется параметрическая форма вида $x = x(t)$.

Основными особенностями такого способа задания модели является высокая производительность по времени, отсутствие дополнительных затрат памяти, сложность преобразований, сложность задания высокодетализированных объектов в аналитической форме.

Аналитический способ используется в основном для задания простых объектов, например, ограниченных поверхностями второго порядка.

С использованием полигональной сетки

Полигональная сетка — это поверхность, заданная как множество полигонов [5]. Она представляет собой набор вершин, связей между ними (ребер) и граней. Эти составляющие и описывают форму объекта.

Среди особенностей полигональной сетки можно отметить то, что с помощью нее можно отображать любые поверхности — разбивая ее на маленькие объекты (полигоны). Благодаря этому, полигональная сетка используется в большинстве сфер применения компьютерной графики, например, в компьютерных играх. Кроме того, стоит отметить большой объем требуемой памяти из-за необходимости хранить все полигоны, и низкая производительность. Например, для хранения сферы аналитическим способом достаточно хранить центр и радиус, при этом при данном способе необходимо хранить информацию о всех полигонах сферы, которых может быть несколько тысяч.

Выбор способа задания модели

В рамках данной задачи, необходимо строить сложные объекты — деревья, а также генерировать их, то есть преобразование сгенерированных данных в уравнения не представляется возможным. Поэтому был выбран способ задания модели с использованием полигональной сетки, как полностью удовлетворяющий требованиям данной задачи.

1.2. Составляющие сцены

Сцена состоит из следующих составляющих.

- Модель, заданная с использованием полигональной сетки. Модель характеризуется следующими параметрами:
  - координатами вершин;
  - списком полигонов;
  - цветом материала;
  - цветом бликов;
  - коэффициентом диффузного отражения;
  - коэффициентом зеркального отражения;
  - коэффициентом фонового освещения.
- Камера, которая характеризуется следующими параметрами:
  - координатами;
  - направлением взгляда.
- Источник света, который характеризуется следующими параметрами:
  - координатами;
  - интенсивностью освещения.

1.3. Построение реалистичного изображения

Важнейшей частью работы является построение реалистичного изображения сгенерированной местности.

Основными этапами синтеза реалистичного изображения являются [7]:

- разработка трехмерной математической модели синтезируемой визуальной обстановки;
- определение направления линии визирования , положения картинной плоскости, размеров окна обзора, значений управляющих сигналов;
- формирование операторов, осуществляющих пространственное перемещение моделируемых объектов визуализации;

— преобразование модели, синтезируемой в пространстве, к координатам наблюдателя;

— отсечение объектов визуального пространства в пределах пирамиды видимости;

— вычисление двумерных перспективных проекций синтезируемых объектов видимости на картинную плоскость;

— исключение невидимых элементов синтезируемого пространства при данном положении наблюдателя, закрашивание и затенение видимых элементов объектов визуализации;

— вывод полутонового изображения синтезируемого визуального пространства на экран растрового дисплея.

1.3.1. Удаление невидимых линий и поверхностей

Существует множество алгоритмов удаления невидимых линий и поверхностей, которые разделяются на две категории [7]:

— Алгоритмы, работающие в объектном пространстве — имеют дело с физическими координатами, в которых описаны исходные объекты. Имеют высокую точность, однако неэффективны по времени, поэтому используются в основном для несложных сцен [7].

— Алгоритмы, работающие в пространстве изображения — имеют дело с координатами экрана. Точность ограничена разрешением дисплея. Но за счет большей эффективности используются для отрисовки более сложных сцен, чем алгоритмы, работающие в объектном пространстве.

В рамках данной задачи будет производится отрисовка сцен с большим количеством объектов, поэтому наиболее целесообразным является использование алгоритмов, работающих в пространстве изображения.

Алгоритмы разбиения на окна

Данные алгоритмы используют гипотезу о том, что на обработку областей, содержащих мало информации, мозгом человека тратится мало времени [8]. Далее дано описание алгоритма.

1. В пространстве изображения рассматривается окно, и решается вопрос о том, есть ли в этом окне объекты, и достаточно ли оно простое для визуализации. Если оно не

является достаточно простым, то происходит дальнейшее разбиение на подокна, пока они не станут достаточно простыми. Разбиение ограничено точностью растрового дисплея.

2. При достижении достаточно простого для визуализации окна, часто окна размером в один пиксел, необходимо выбрать один из вариантов алгоритмов: удаления невидимых линий и удаления невидимых поверхностей [8]:

   — при удалении невидимых линий, пиксел закрашивается цветом линии, если через него проходит какая-либо линия, иначе, закрашивается фоновым цветом;

   — при удалении невидимых поверхностей, происходит проверка охвата пиксела многоугольниками сцены. Если пиксел охвачен какими-либо многоугольниками, то выбирается ближайший относительно центра пиксела многоугольник, и пиксел закрашивается в его цвет. Иначе, если пиксел не охвачен каким-либо многоугольником, он закрашивается цветом фона.

Процесс разбиения на подокна образует древовидную структуру, показанную на рис. 1.1.

На сложных сценах алгоритмы производят большое количество разбиений [8]. Алгоритмы требуют модификации для предоставления средств учета теней и освещения.

Алгоритм, использующий z-буфер

Идея алгоритма заключается в использовании буфера кадра и буфера глубины. Буфер кадра — буфер, в котором хранится информация о каждом пикселе изображения. Буфер глубины — буфер, в котором хранится информация о максимальной величине координаты z отображаемого пиксела.

При обработке очередного многоугольника, для фиксированных x и y производится сравнение координаты z со значением в буфере глубины, если оно больше, чем в буфере, то происходит обновление буфера кадра и буфера глубины с соответствующими значениями для текущего многоугольника.

В качестве особенностей алгоритма можно отметить его простоту, работу со сценами любой сложности, большой объем требуемой памяти [11]. Алгоритм требует модификации для предоставления средств учета теней и освещения.

Алгоритмы построчного сканирования

Идеей алгоритмов построчного сканирования является обработка сцены в порядке прохождения сканирующей строки. Такой подход позволяет перейти от трехмерной задачи

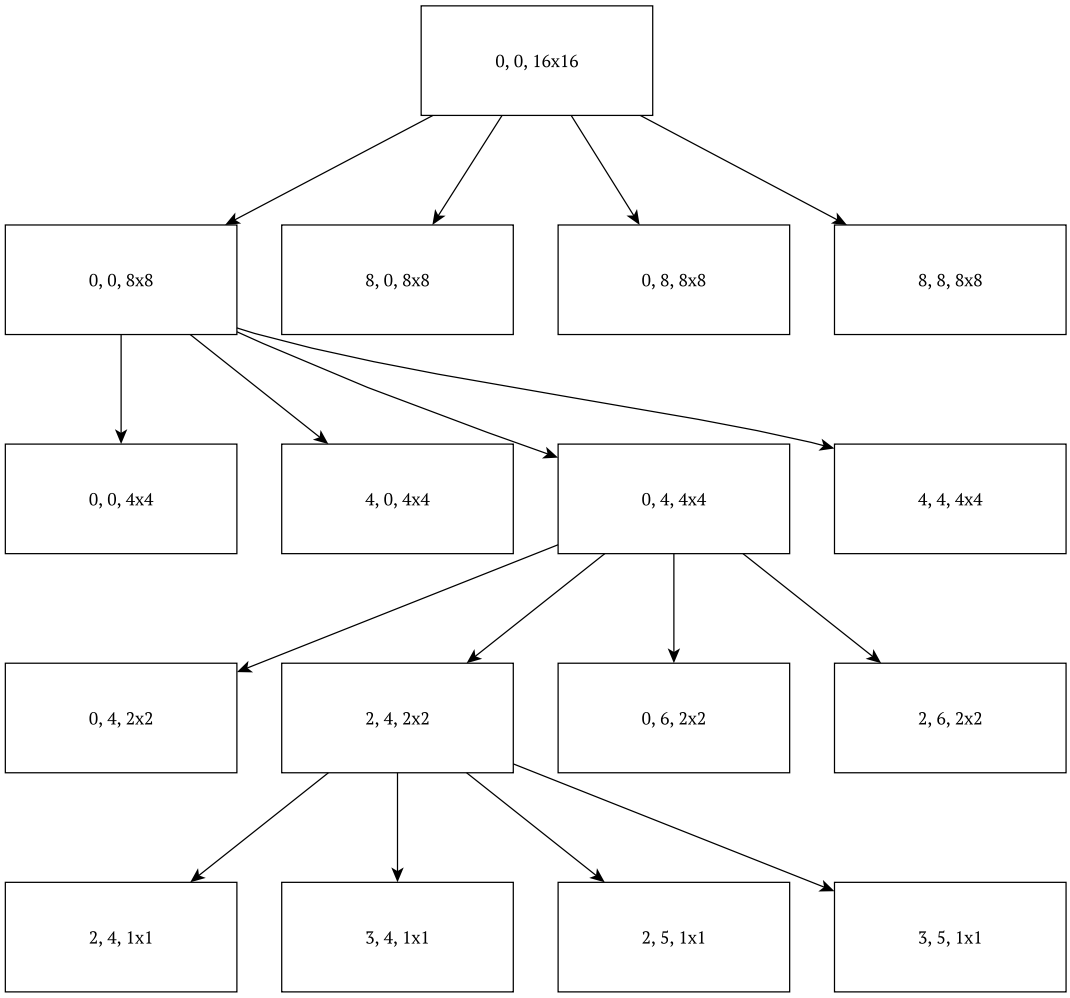

Рисунок 1.1 — Дерево структуры подокон

удаления поверхностей к двумерной задаче удаления линий [12].

Одним из вариантов алгоритма построчного сканирования является алгоритм построчного сканирования, использующий z-буфер. В отличие от классического алгоритма, при построчном сканировании нет необходимости хранить буфер кадра и буфер глубины для всего изображения, достаточно хранить буферы только для обрабатываемой строки.

Таким образом, алгоритм построчного сканирования, использующий z-буфер требует меньший объем памяти, чем классический алгоритм. Алгоритмы требуют модификации для предоставления средств учета теней и освещения.

Алгоритмы трассировки лучей

В отличие от остальных алгоритмов, алгоритмы трассировки лучей, являются методами грубой силы, так как не учитывают специфику обрабатываемых объектов [8]. Идея алгоритмов заключается в отслеживании световых лучей между источниками света, объ-

ектами и наблюдателем.

Алгоритм прямой трассировки лучей отслеживает лучи от источников света, с учетом их отражения от объектов сцены. Данный алгоритм является неэффективным [8], так как множество лучей не доходят до наблюдателя, но все равно оказываются обработаны.

Альтернативой методу прямой трассировки является алгоритм обратной трассировки лучей, в котором происходит отслеживание световых лучей в обратном направлении, то есть от наблюдателя к объектам, от объектов к источникам света. Это позволяет обрабатывать только видимые наблюдателем лучи. Визуализация алгоритма приведена на рис. 1.2.

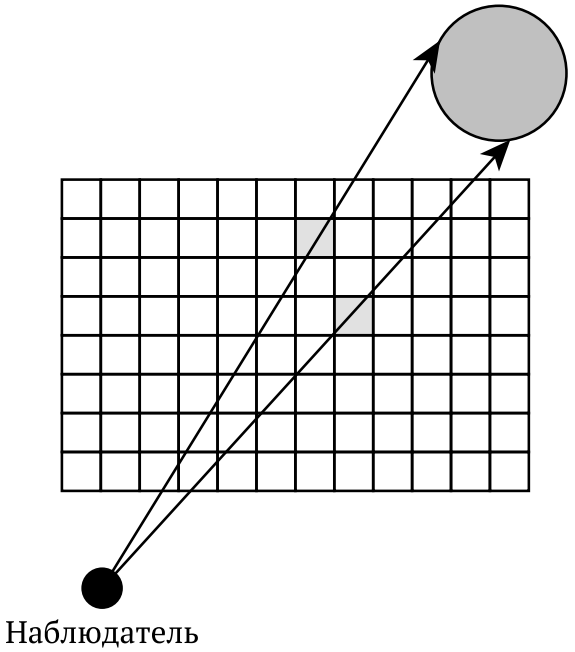

Рисунок 1.2 — Визуализация алгоритма обратной трассировки лучей

Алгоритмы позволяют учитывать тени и освещения, работают со сценами любой сложности. Кроме того, из-за присущей алгоритмам параллельности — лучи можно обрабатывать независимо друг от друга, алгоритм можно реализовывать с использованием методов параллельной обработки [9].

Выбор алгоритма удаления невидимых линий и поверхностей

В рамках данной работы важен учет теней и освещения, кроме того, производится работа со сложными объектами — деревьями, так что необходимым является способность работы алгоритма работать со сценами любой сложности.

Кроме того, рассматривается возможность обработки сцены с использованием методов параллельной обработки.

В соответствии с требованиями, для удаления невидимых линий и поверхностей был выбран алгоритм обратной трассировки лучей.

1.3.2. Учет теней и освещения

Так как для реализации удаления невидимых линий и поверхностей был выбран алгоритм обратной трассировки лучей, который позволяет учитывать тени и освещение, вопрос выбора методов учета теней и освещения в рамках данной работы не является актуальным.

1.4. Генерация дерева

Для генерации деревьев необходимо использовать следующее свойство: суммарная толщина веток, исходящих из некоторой другой ветки, равна толщине этой ветки [18]. Дерево в таком случае можно представить в виде графа [16], а точнее — двоичного дерева, так как чаще всего ветка дерева разделяется на две части.

При получении питательных веществ, ветка увеличивается в размерах, причем если у нее нет потомков, то она увеличивается как в длину, так и в толщину. Если же у нее есть потомки, то происходит увеличение только в толщину, остальные питательные вещества переходят к потомкам.

При достижении веткой определенной длины, происходит ее разделение на две части, и рост этой ветки в длину прекращается.

1.5. Выводы по аналитической части

В данном разделе были формализованы объекты сцены, рассмотрены методы построения реалистичного изображения, методы хранения объектов, произведено сравнение существующих подходов. Был рассмотрен метод генерации деревьев.

2. Конструкторская часть

В данном разделе представлены используемые алгоритмы, а так же приведено обоснование выбора типов и структур данных.

2.1. Разработка алгоритма обратной трассировки лучей

2.1.1. Входные данные

— массив записей с информацией об объектах $Objects$, для каждого объекта сцены хранится информация о полигонах объекта;

— массив записей с информацией об источниках света $Lights$, для каждого источника света хранится информация о местоположении источника, а так же его интенсивности;

— целочисленные длина и ширина ожидаемого изображения;

— радиус-вектор положения наблюдателя S;

— координата z_m экрана.

2.1.2. Выходные данные

— растровое изображение заданной сцены $Image$.

2.1.3. Алгоритм

Общий алгоритм

Алгоритм обратной трассировки лучей можно разделить на следующие этапы.

1. Для каждого пиксела $Image$ с координатами x, y:

   (a) Сформировать вектор

$$V = \{x, y, z_m\} - S, \tag{2.1}$$

   где V — вектор направления взгляда.

(b) Найти пересечение с ближайшим объектом сцены.

(c) Найти освещенность данной точки с учетом света от источников.

(d) Установить цвет пиксела x, y изображения $Image$.

2. Вернуть $Image$.

Нахождение пересечения с ближайшим объектом сцены

1. Для каждого объекта сцены $Object$ из $Objects$:

   (a) Проверить, пересекает ли луч V сферическую оболочку объекта $Object$.

   Тест используется для оптимизации. Каждый объект сцены описывается в сферу. Соответственно, если луч не пересекает сферическую оболочку объекта, то оно точно не пересекает объект. При этом, проверка пересечения со сферой требует меньших вычислительных затрат, чем проверка пересечения со множеством полигонов.

   (b) Если не пересекает, то перейти на следующий шаг.

   (c) Найти пересечение луча с ближайшим полигоном объекта.

   (d) Поместить найденный полигон в промежуточную переменную $Nearest$.

2. Вернуть $Nearest$.

Нахождение пересечения с ближайшим полигоном объекта

1. Для каждого полигона $Polygon$ из объекта $Object$:

   (a) Найти точку пересечения луча V и полигона $Polygon$.

   (b) Если пересечения нет, то перейти на следующий шаг.

   (c) Если есть пересечение, то сравнить координату z точки пересечения с сохраненной координатой z_{max}.

   (d) Если больше, то сохранить точку пересечения и номер полигона в промежуточные переменные $Intersection_{nearest}$ и $Index_{nearest}$ соответственно.

2. Вернуть $Intersection_{nearest}$ и $Index_{nearest}$.

Нахождение точки пересечения луча и полигона

Пусть:

— уравнение плоскости, на котором лежит полигон имеет вид

$$a \cdot x + b \cdot y + c \cdot z + d = 0; \tag{2.2}$$

— вектор нормали к этой плоскости $n = \{a, b, c\}$ нормализован;

— луч представлен в виде

$$V(t) = S + V \cdot t, \tag{2.3}$$

где

— S — положение наблюдателя;

— V — вектор направления взгляда;

— t — параметр.

Тогда для нахождения точки пересечения луча и полигона надо произвести следующие действия.

1. Найти такой параметр t, при котором луч пересекает плоскость.

$$a \cdot (S_x + V_x \cdot t) + b \cdot (S_y + V_y \cdot t) + c \cdot (S_z + V_z \cdot t) + d = 0 \tag{2.4}$$

$$t = -\frac{a \cdot S_x + b \cdot S_y + c \cdot S_z + d}{a \cdot V_x + b \cdot V_y + c \cdot V_z} \tag{2.5}$$

2. Проверить, что точка пересечения лежит внутри полигона.

   Проверка пересечения производится с помощью следующего теста: вычисляется векторное произведение вектора, образованного стороной полигона, и вектора, образованного начало первого вектора и проверяемой точкой. Если для всех таких произведений, для каждой стороны, знаки векторного произведения совпадают, то точка лежит внутри полигона [13].

3. Если точка лежит внутри полигона, вернуть точку пересечения.

Проверка пересечения луча и сферической оболочки

Для проверки пересечения луча и полигона можно воспользоваться геометрическим методом. На рис. 2.1 изображен чертеж к нахождению пересечения луча и сферы.

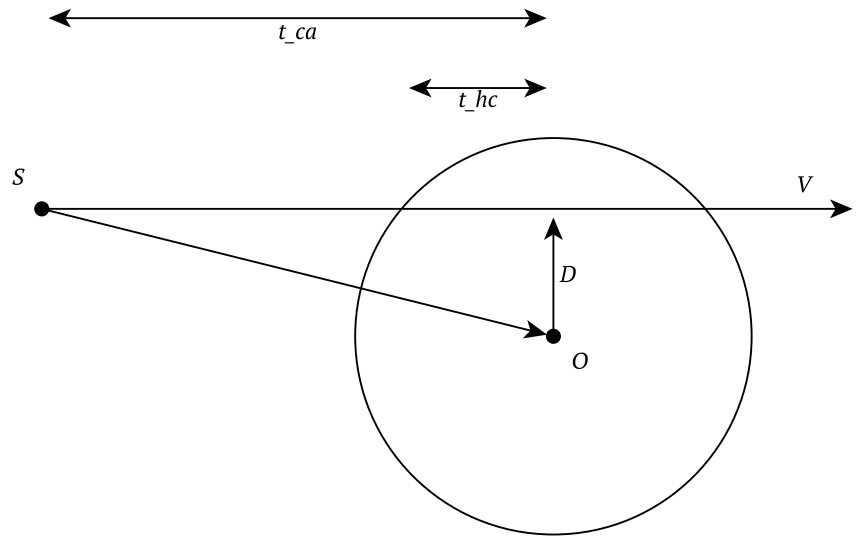

Рисунок 2.1 — Пересечение луча со сферой

1. Определить, лежит ли начало луча внутри сферы. Для этого необходимо вычислить вектор

$$SO = O - S. \tag{2.6}$$

   Если длина вектора SO меньше или равна радиусу окружности R, то начало луча лежит внутри сферы, то есть луч пересекает сферу. В этом случае вернуть «Да».

2. Найти ближайшую к центру точку луча. Расстояния до нее можно обозначить как D. Вычислить

$$t_{ca} = SO \cdot V, \tag{2.7}$$

   где t_{ca} — расстояние от начала луча до ближайшей к центру сферы точки.

3. Если $t_{ca} < 0$, вернуть «Нет».

4. Вычислить

$$D^2 = SO^2 - t_{ca}^2, \tag{2.8}$$

$$t_{hc}^2 = R^2 - D^2 = r^2 - SO^2 + t_{ca}^2. \tag{2.9}$$

5. Если $t_{hc}^2 < 0$, вернуть «Нет».

6. Вернуть «Да».

Нахождение освещенности точки с учетом света от источников света

1. Инициализировать освещенность $Intensity$ значением 0.

2. Для каждого источника света $Light$ из $Lights$:

   (a) Сформировать вектор

   $$V = \{x, y, z\} - Light, \tag{2.10}$$

   где x, y, z — координаты точки, для которой проверяется освещенность, V — вектор направления взгляда от источника света к точке.

   (b) Найти пересечение с ближайшим объектом сцены.

   (c) Если пересечение не совпадает с проверяемой точкой, перейти на следующий шаг.

   (d) Вычислить значение I — интенсивность света от одного источника в данной точке:

   $$I = K_a \cdot I_a + K_d \cdot (n \cdot V), \tag{2.11}$$

   где

   — K_a — коэффициент фонового освещения;
   — I_a — интенсивность фонового освещения;
   — K_d — коэффициент диффузного отражения;
   — n — вектор нормали к рассматриваемому полигону.

   (e) Сложить I к итоговому значению $Intencity$.

3. Вернуть $Intencity$.

2.2. Разработка алгоритма генерации лесистой местности

2.2.1. Входные данные

— точки (x_{min}, z_{min}), (x_{max}, z_{max}), задающие площадку, на которой будет расположен фрагмент лесистой местности;

— величина y — координата площадки по оси Y;

— число с плавающей точкой, соотношение питания одной ветки к другой $ratio$;

— коэффициент густоты кроны $spread$;

— длина ветки, при которой происходит разделение $splitSize$;

— количество деревьев на площадке N.

2.2.2. Выходные данные

— список описаний деревьев на площадке.

2.2.3. Алгоритм

Рост дерева

Пусть

— $feed$ — количество питательных веществ, полученной веткой.

1. Если ветка не имеет потомков (является листом):
   (a) Вычислить
   $$lenAddition = \sqrt[3]{feed} \tag{2.12}$$
   (b) Добавить $lenAddition$ к длине ветки.
   (c) Добавить остаток от $feed$ к площади ветки.
   (d) Если ветка имеет достаточную для разделения длину, разделить ее.
2. Если ветка имеет потомков:

(a) Вычислить

$$K = \frac{childArea}{childArea + area}, \tag{2.13}$$

где

— $childArea$ — суммарная площадь потомков данной ветки;

— $area$ — площадь данной ветки.

(b) Добавить к площади ветки значение $\frac{K \cdot feed}{len}$, где len — текущая длина ветки.

(c) Распределить остаток питания по потомкам в соотношении $ratio$.

Деление ветки

1. Создать ветки-потомки A и B.
2. Вычислить вектор нормали N к текущему направлению ветки.
3. Вычислить

$$Ns = N * randSign * spread, \tag{2.14}$$

где $randSign$ — случайный знак, $+1$ или -1.

4. Вычислить

$$Ms = -Ns. \tag{2.15}$$

5. Рассчитать направления A как линейную интерполяцию между векторами Ns и Dir для значения $ratio$.
6. Рассчитать направления B как линейную интерполяцию между векторами Ms и Dir для значения $1.0 - ratio$. Таким образом, чем больше $ratio$, тем больше ветка получает питания, следовательно растет, и тем меньше она будет отклоняться от направления родительской ветки.

Размещение деревьев по площадке

1. Вычислить S — площадь площадки.
2. Вычислить
3.

$$S_{one} = \frac{S}{N}, \tag{2.16}$$

где S_{one} — площадь, заметаемая одним деревом.

4. Вычислить

$$R = \sqrt{\frac{S_{one}}{\pi}}, \tag{2.17}$$

где R — радиус окружности, заметаемой одним деревом.

5. Создать пустой список деревьев $Trees$.

6. Создать список всех доступных точек на площадке $Points$.

7. Цикл N раз:

   (a) Выбрать случайную точку P из $Points$.

   (b) Добавить в $Trees$ дерево, начинающееся с выбранной точки P.

   (c) Удалить из $Points$ все точки, расстояние от которых до P меньше или равно R.

8. Вернуть $Trees$.

2.3. Обоснование типов и структур данных

В рамках данной работы использованы следующие сущности.

2.3.1. Материал

Запись, содержащая следующие поля:

— коэффициент фонового освещения — число с плавающей точкой;

— коэффициент диффузного отражения — число с плавающей точкой;

— цвет — вектор из трех значений целочисленного типа со значениями от 0 до 255 включительно.

2.3.2. Полигон

Запись, содержащая следующие поля:

— материал — запись типа «Материал»;

— координаты первой точки полигона — вектор из трех значений типа числа с плавающей точкой;

— координаты второй точки полигона — вектор из трех значений типа числа с плавающей точкой;

— координаты третьей точки полигона — вектор из трех значений типа числа с плавающей точкой;

— коэффициенты уравнения плоскости, проходящей через полигон, четыре числа с плавающей точкой.

2.3.3. Объект сцены

Запись, содержащая следующие поля:

— полигоны — массив, состоящий из элементов типа «Полигон». Выбор массива обусловлен возможностью произвольного доступа к элементам массива за константное время;

— координаты центра сферической оболочки — вектор из трех значений типа числа с плавающей точкой;

— радиус сферической оболочки — число с плавающей точкой.

2.3.4. Источник света

Запись, содержащая следующие поля:

— координаты источника света — вектор из трех значений типа числа с плавающей точкой;

— интенсивность — число с плавающей точкой.

2.3.5. Сцена

Запись, содержащая следующие поля:

— объекты — массив, состоящий из элементов типа «Объект сцены». Выбор массива обусловлен возможностью произвольного доступа к элементам массива за константное время;

— источники — массив, состоящий из элементов типа «Источник света». Выбор массива обусловлен возможностью произвольного доступа к элементам массива за константное время;

— положение камеры — вектор из трех значений типа числа с плавающей точкой;

— высота и ширина результирующего изображения — целые числа.

2.3.6. Дерево

Запись, содержащая следующие поля:

— длина ветки — число с плавающей точкой;

— направление — вектор из трех значений типа числа с плавающей точкой;

— потомки — два элемента типа «Дерево»;

— длина ветки, при которой произойдет разделение — число с плавающей точкой;

— соотношение количества питания между двумя ветками — число с плавающей точкой;

— коэффициент густоты кроны — число с плавающей точкой.

2.3.7. Лес

Запись, содержащая следующие поля:

— деревья — массив, состоящий из элементов типа «Дерево». Выбор массива обусловлен возможностью произвольного доступа к элементам массива за константное время.

2.4. Выводы по конструкторской части

В данном разделе были разработаны алгоритмы обратной трассировки лучей и генерации лесистой местности. Кроме того, были выбраны и обоснованы типы и структуры данных.

3. Технологическая часть

В данном разделе представлены требования к ПО, выбор средств реализации, реализации алгоритмов, описание интерфейса программы.

3.1. Требования к ПО

— программа выполняет генерацию лесистой местности на основе таких параметров, как количество деревьев, координаты местности, густота кроны, ассиметрия дерева;

— программа позволяет отображать сгенерированную местность в различные этапы жизненного цикла деревьев;

— программа создает изображение на основе сгенерированной модели лесистой местности;

— программа может сохранять полученное изображение в файл формата `png`;

— программа позволяет проводить модульное тестирование при запуске из командной строки;

— программа позволяет указывать количество потоков для работы при запуске из командной строки;

— программа выводит время, затраченное на построение изображения, при запуске из командной строки.

3.1.1. Функциональная модель

На рис. 3.1 – 3.2 представлена функциональная модель программы в нотации `IDEF0`.

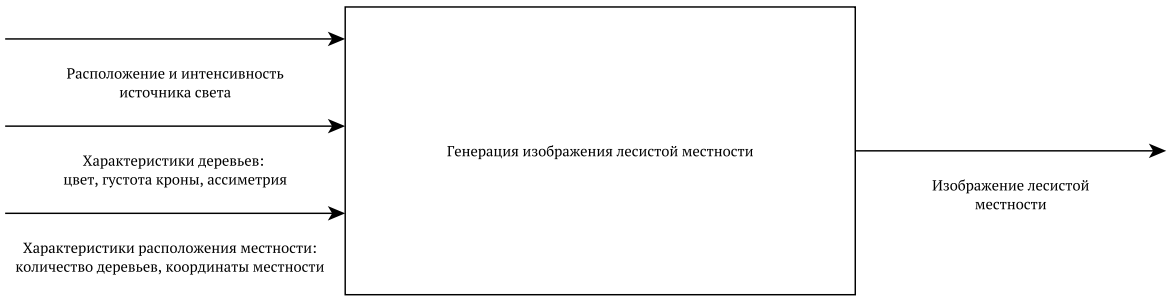

Рисунок 3.1 — Функциональная модель в нотации IDEF0

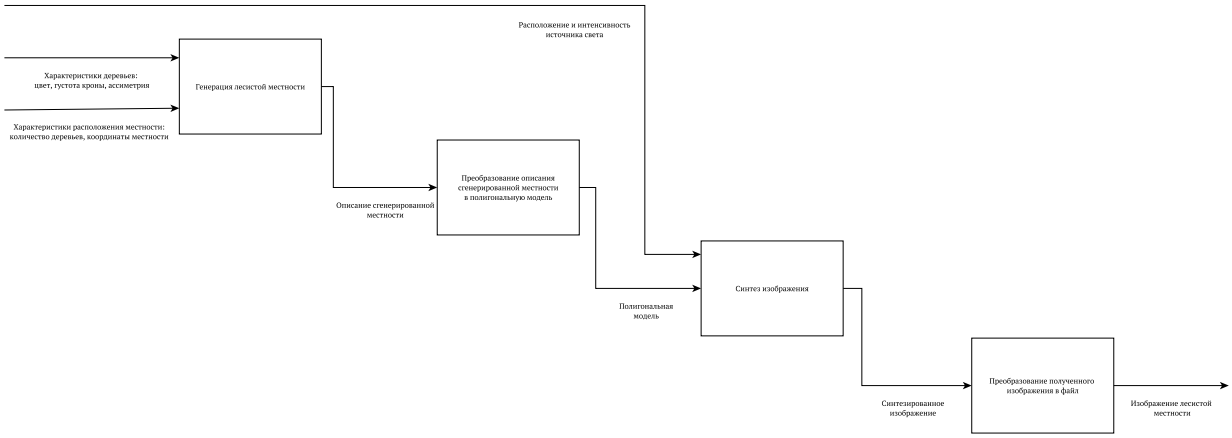

Рисунок 3.2 — Функциональная модель в нотации IDEF0 (продолжение)

3.2. Средства реализации

3.2.1. Язык программирования

В качестве языка программирования был выбран язык C++, так как он обладает достаточным инструментарием для работы с объектами, является компилируемым, имеет встроенные средства оценки времени работы программы.

3.2.2. Набор библиотек

В качестве набора библиотек был выбран Qt, так как он обладает достаточным инструменталом для работы с компьютерной графикой, а также имеет набор средств для создания кроссплатформенных графических приложений.

3.2.3. Формат выходных файлов

В качестве способа представления выходных данных был выбран формат `PNG`, так как он:

— является кроссплатформенным, его можно просмотреть на любой популярной платформе без использования дополнительного программного обеспечения;

— обеспечивает качество изображения не худшее, чем в конкурирующих форматах, при приемлимом потреблении памяти файлом [14];

— поддерживается фреймворком `Qt`.

3.3. Реализация алгоритма обратной трассировки лучей

В листингах 3.1 – 3.4 представлена реализация алгоритма обратной трассировки лучей.

Листинг 3.1 — Листинг функции, реализующей основной цикл прохода по пикселам

```
std::shared_ptr<QImage> Scene::paint() {
    polygons = forest.toPolygons();

    auto image = std::make_shared<QImage>(width, height,
    QImage::Format_RGB32);
    image->fill(Qt::black);

    QPainter painter(image.get());

    std::vector<std::vector<QColor>> colorBuffer(height,
    std::vector<QColor>(width, Qt::black));

    Ray ray = Ray(camera, QVector3D());

    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
        ray.direction = (QVector3D(x, y, 0) - camera).normalized();
        colorBuffer[y][x] = traceRay(ray);
        }
    }

    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            painter.setPen(colorBuffer[y][x]);
            painter.drawPoint(x, y);
        }
    }

    return image;
}
```

Листинг 3.2 — Листинг функции, реализующей трассировку одного луча

```
QColor Scene::traceRay(const Ray &ray) {
    TraceRayData data;
    size_t index = 0;
    bool first = true;

    for (size_t i = 0; i < polygons.size(); i++) {
        auto res = polygons[i].traceRay(ray);
        if (!res.ok) {
            continue;
        }

        if (first || res.t < data.t) {
            index = i;
            data = res;
            first = false;
        }
    }

    if (first) {
        return background;
    }

    auto col = getColor(data.point, ray, lights, polygons[index]);
    if (!col.haveColor) {
        return Qt::black;
    }

    return col.color;
}
```

Листинг 3.3 — Листинг функции, реализующей трассировку луча в рамках одного полигона

```
TraceRayData Polygon::traceRay(const Ray &ray) {
    if (!shell.traceRay(ray)) {
        return TraceRayData();
    }

    QVector3D dir = ray.direction;
    QVector3D start = ray.camera;

    double k = a * dir.x() + b * dir.y() + c * dir.z();
    if (qFuzzyIsNull(k))
        return TraceRayData();

    double t = -(a * start.x() + b * start.y() + c * start.z() + d) / k;
    if (t < 0)
        return TraceRayData();

    QVector3D isc = start + dir * t;
    if (!inside(isc))
        return TraceRayData();

    return TraceRayData(isc, QVector3D(a, b, c).normalized(), t);
}
```

Листинг 3.4 — Листинг функций, реализующих определение нахождения точки внутри полигона

```
inline int sign(double a) {
    if (qFuzzyIsNull(a)) {
        return 0;
    }

    if (a > 0) {
        return 1;
    }

    return -1;
}

inline bool equalSign(double v1, double v2) {
    return sign(v1) * sign(v2) >= 0;
}

inline bool equalSigns(const QVector3D &v1, const QVector3D &v2) {
    return equalSign(v1.x(), v2.x()) && equalSign(v1.y(), v2.y())
    && equalSign(v1.z(), v2.z());
}

inline bool Polygon::inside(const QVector3D &p) {
    auto ap = p - pointA;
    auto bp = p - pointB;
    auto cp = p - pointC;

    auto m1 = QVector3D::crossProduct(ab, ap);
    auto m2 = QVector3D::crossProduct(bc, bp);
    auto m3 = QVector3D::crossProduct(ca, cp);

    return equalSigns(m1, m2) && equalSigns(m2, m3) && equalSigns(m3, m1);
}
```

3.4. Реализация алгоритма генерации лесистой местности

В листингах 3.5 – 3.6 представлена реализация алгоритма генерации деревьев. В листингах 3.7 – 3.8 представлена реализация алгоритма перевода древовидной структуры сгенерированного дерева в полигональный формат. В листингах 3.9 – 3.10 представлена реализация алгоритма размещения деревьев на плоскости случайным образом.

Листинг 3.5 — Листинг функции, реализующей рост дерева

```
void Branch::grow(double feed) {
    radius = sqrt(area / M_PI);

    if (leaf) {
        double lenAddition = cbrt(feed);
        len += lenAddition;
        feed -= lenAddition * area;
        area += feed / len;
        if (canSplit()) {
            split();
        }
    } else {
        double childArea = childA->area + childB->area;
        double k = childArea / (childArea + area);
        area += k * feed / len;
        feed *= 1.0 - k;
        if (qFuzzyIsNull(k)) {
            return;
        }
        childA->grow(feed * ratio);
        childB->grow(feed * (1.0 - ratio));
    }
}
```

Листинг 3.6 — Листинг функций, реализующих разделение ветви дерева

```
double randSign() {
    return (rand() % 2) * 2 - 1;
}

QVector3D interpolate(QVector3D a, QVector3D b, double k) {
    return a * (1.0 - k) + b * k;
}

void Branch::split() {
    leaf = false;

    childA = new Branch(this, 0);
    childB = new Branch(this, 1);

    auto d = leafDensity(LOCAL_DEPTH);

    auto n = QVector3D::crossProduct(dir, d);
    n.normalize();

    auto m = -n;

    double flippedSpread = randSign() * spread;

    auto ns = n * flippedSpread;
    auto ms = m * flippedSpread;

    childA->dir = interpolate(ns, dir, ratio).normalized();
    childB->dir = interpolate(ms, dir, 1.0 - ratio).normalized();
}
```

Листинг 3.7 — Листинг функции, производящей перевод дерева в полигональный формат

```
std::vector<Polygon> Branch::toPolygons() {
    auto end = getEnd();
    auto x = dir + QVector3D(1.0, 1.0, 1.0);
    auto n = QVector3D::crossProduct(dir, x).normalized();
    double a = 180.0 / RING_POLYGONS;
    std::vector<Polygon> res;
    std::vector<QVector3D> positions;
    for (int i = 0; i < RING_POLYGONS; i++) {
        positions.push_back(start + radius * BRANCH_SCALE * n);
        n = rotateAboutVector(n, a, dir);
        positions.push_back(end + radius * BRANCH_SCALE * TAPER * n);
        n = rotateAboutVector(n, a, dir);
    }
    for (int i = 0; i < RING_POLYGONS; i++) {
        QVector3D a1 = positions[i * 2];
        QVector3D a2 = positions[(i * 2 + 2) % (2 * RING_POLYGONS)];
        QVector3D a3 = positions[i * 2 + 1];
        QVector3D b1 = positions[(i * 2 + 2) % (2 * RING_POLYGONS)];
        QVector3D b2 = positions[(i * 2 + 3) % (2 * RING_POLYGONS)];
        QVector3D b3 = positions[i * 2 + 1];
        res.push_back(Polygon(a1, a2, a3, color, Qt::white));
        res.push_back(Polygon(b1, b2, b3, color, Qt::white));
    }
    if (leaf && depth > 2) {
        QVector3D a1 = end + dir * 10;
        QVector3D a2 = end + n * 10;
        QVector3D a3 = end - n * 10;
        res.push_back(Polygon(a1, a2, a3, leafColor, Qt::white));
    }
```

Листинг 3.8 — Листинг функции, производящей перевод дерева в полигональный формат (продолжение листинга 3.7)

```
    if (leaf) {
        return res;
    }
    auto polygonsA = childA->toPolygons();
    auto polygonsB = childB->toPolygons();
    res.insert(res.end(), polygonsA.begin(), polygonsA.end());
    res.insert(res.end(), polygonsB.begin(), polygonsB.end());
    return res;
}
```

Листинг 3.9 — Листинг функции, генерирующей случайное размещение деревьев

```
Forest::Forest(int n, int seed, QVector2D bl, QVector2D tr, double y,
 QColor tc, QColor lc, QColor pc, double r, double s, double ss):
    bottomLeft(bl),
    topRight(tr),
    treeColor(tc),
    leafColor(lc),
    planeColor(pc),
    ratio(r),
    spread(s),
    splitSize(ss),
    y(y)
{
    srand(seed);
    if (n > 0) {
        int trX = tr.x() - GAP;
        int trY = tr.y() - GAP;
        int blX = bl.x() + GAP;
        int blY = bl.y() + GAP;
```

Листинг 3.10 — Листинг функции, генерирующей случайное размещение деревьев (продолжение листинга 3.9)

```
        double shape = abs((trX - blX) * (trY - blY));
        double sOne = K * shape / n;
        double rad = sqrt(sOne / M_PI);
        std::vector<QVector2D> ps(shape);
        int i = 0;
        for (int y = blY; y < trY; y++) {
            for (int x = blX; x < trX; x++) {
                ps[i] = QVector2D(x, y);
                i++;
            }
        }
        std::default_random_engine e(seed);
        std::shuffle(ps.begin(), ps.end(), e);
        int size = ps.size();
        for (int i = 0; i < n && size > 0; i++) {
            auto pos = ps[rand() % size];
            trees.push_back(new Branch(QVector3D(pos.x(), y, pos.y()),
             ratio, spread, splitSize, treeColor, leafColor));
            for (int j = 0; j < size; j++) {
                if (ps[j].distanceToPoint(pos) <= rad) {
                    std::swap(ps[j], ps[size - 1]);
                    size--;
                }
            }
        }
    }
    bottomRight = QVector2D(topRight.x(), bottomLeft.y());
    topLeft = QVector2D(bottomLeft.x(), topRight.y());
}
```

3.5. Описание интерфейса программы

Программа запускается или из среды разработки `Qt Creator`, или с помощью командной строки. На рис. 3.3 представлен внешний вид основного окна программы.

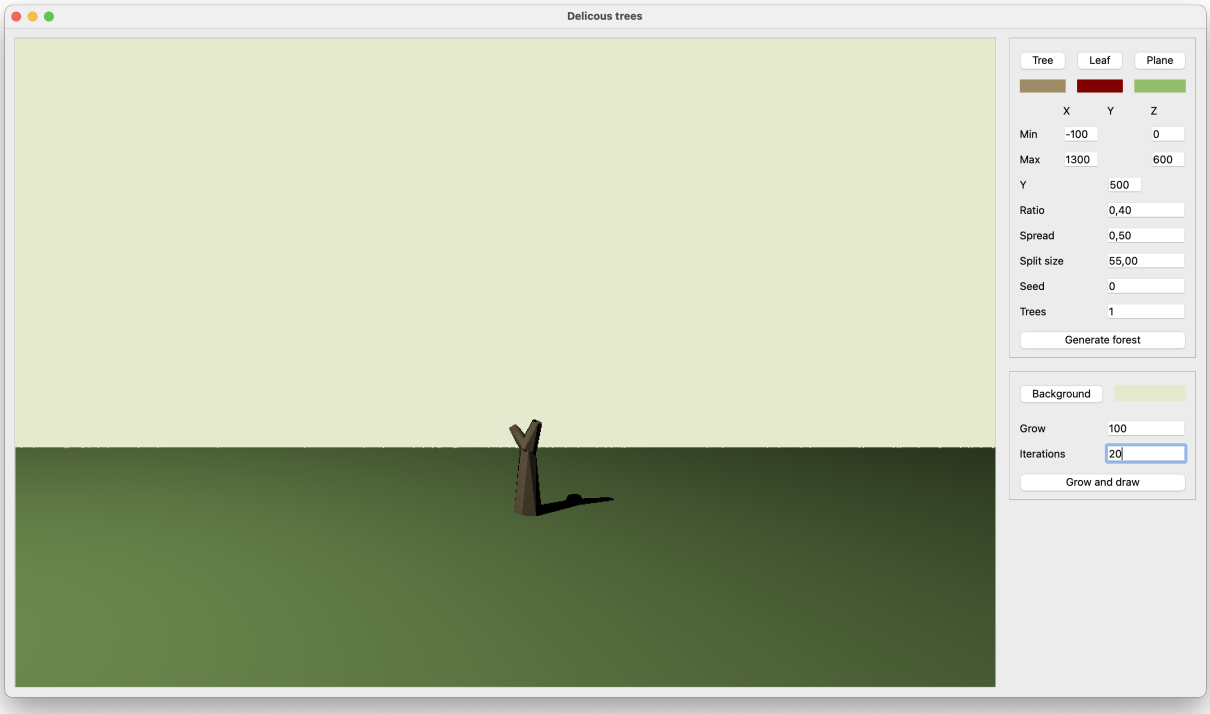

Рисунок 3.3 — Внешний вид основного окна программы

— `Tree` — кнопка для изменения цвета деревьев;

— `Leaf` — кнопка для изменения цвета листьев;

— `Plane` — кнопка для изменения цвета площадки под деревьями;

— `Min` — минимальные координаты плоскости по x и z;

— `Max` — максимальные координаты плоскости по x и z;

— `Y` — координата y плоскости;

— `Ratio` — соотношение объема питания одной ветки к другой;

— `Spread` — густота кроны;

— `Split size` — длина ствола, после которого он начнет разделяться;

— `Seed` — зерно случайной генерации;

— `Trees` — количество деревьев;

— `Generate forest` — кнопка для генерации нового леса;

— `Background` — кнопка для изменения цвета фона;

— `Grow` — количество питания, поставляемого деревьям;

— `Iterations` — количество итераций, в течение которых деревьям будет доставлено указанное количество питания;

— `Grow and draw` — кнопка для доставки питания к деревьям и отрисовки сцены.

3.6. Описание управления из командной строки

Программа поддерживает аргументы командной строки. Ниже представлено их описание.

— `-i` — количество итераций, в течение которых деревьям будет доставлено питание;

— `-o` — название выходного файла;

— `-u` — если указан, то проводится модульное тестирование;

— `-b` — если указан, то программа осуществляет замер времени;

— `-t` — количество потоков, используемых при синтезе изображения.

3.7. Тестирование

Тестирование программы проводилось с использованием модульных тестов. Для модульного тестирования использовалась библиотека `Qt Test`. Для каждого тестируемого класса `Class` создается класс `TestClass`, наследуемый от `QObject`. Конкретные тесты помещаются в приватные слоты этого класса.

Соответственно, для добавления новых тестов, необходимо создать класс тестирования (если такового нет), и добавить тесты в его приватные слоты.

В листингах 3.11 – 3.12 приведены примеры тестового класса и тестовой функции.

Листинг 3.11 — Класс тестирования полигона

```
class TestPolygon : public QObject
{
    Q_OBJECT
public:
    explicit TestPolygon(QObject *parent = nullptr);

private slots:
    void test_traceRay_usual();

signals:

};
```

Листинг 3.12 — Функция тестирования

```
void TestPolygon::test_traceRay_usual() {
    auto p = Polygon(QVector3D(0, 0, 0),
    QVector3D(100, 0, 0), QVector3D(0, 100, 0),
    Qt::black, Qt::black);

    Ray ray = Ray(QVector3D(0, 0, 100), QVector3D(0, 0, -1));
    auto resp = p.traceRay(ray);

    QCOMPARE(resp.ok, true);
    QCOMPARE(qFuzzyCompare(resp.t, 100), true);
}
```

3.8. Примеры работы программы

На рисунках 3.4 – 3.7 показаны примеры работы программы для различных входных параметров.

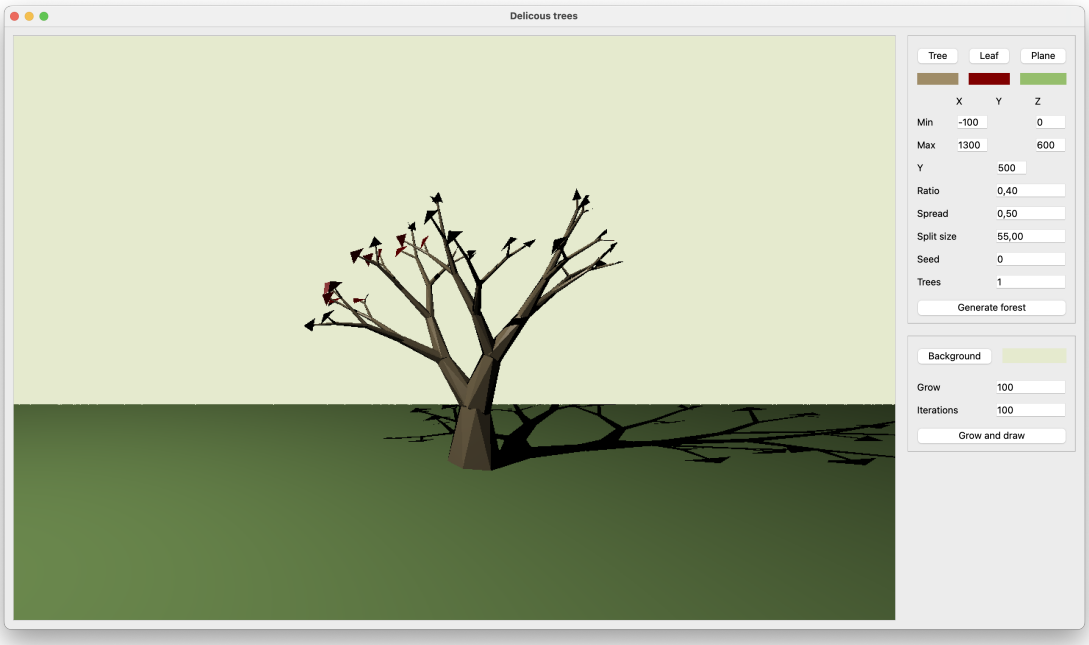

Рисунок 3.4 — Обычное дерево

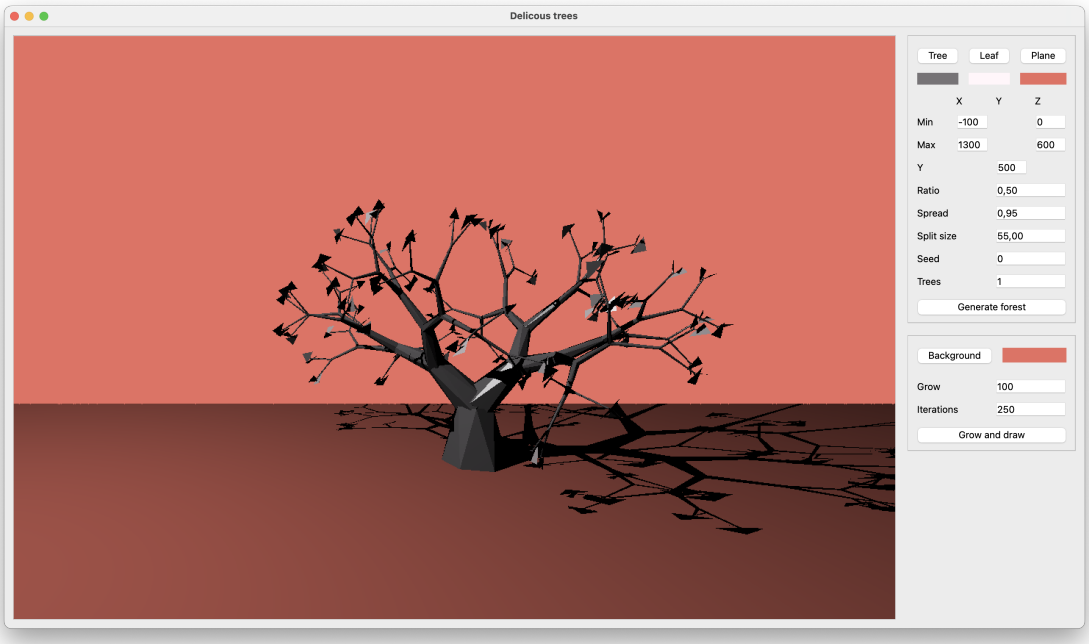

Рисунок 3.5 — Дерево в японском стиле

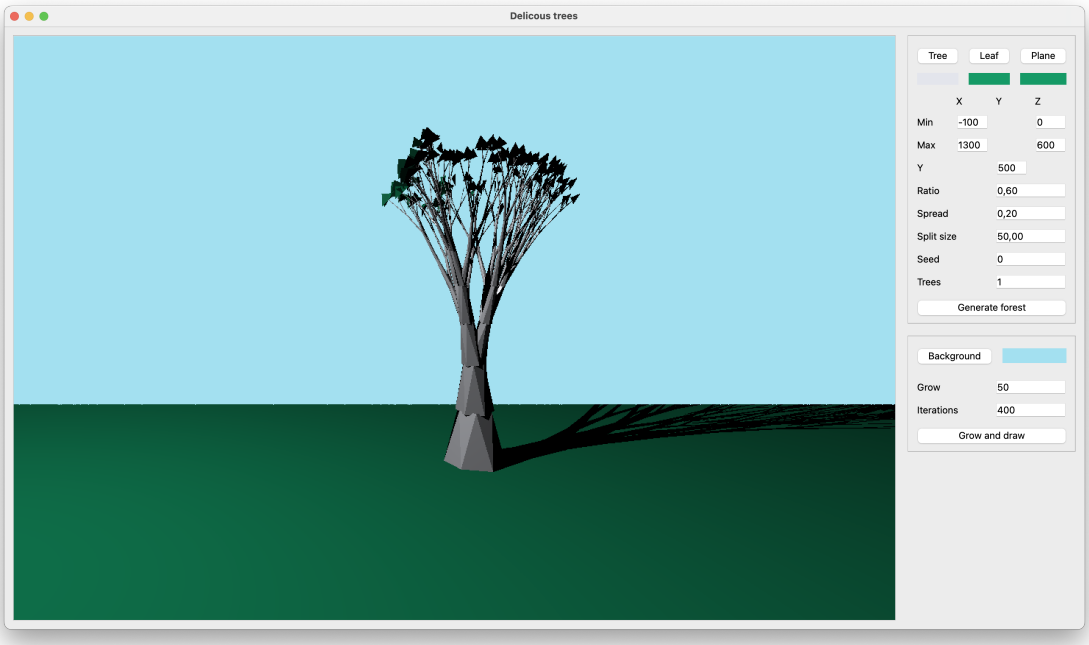

Рисунок 3.6 — Дерево в северном стиле

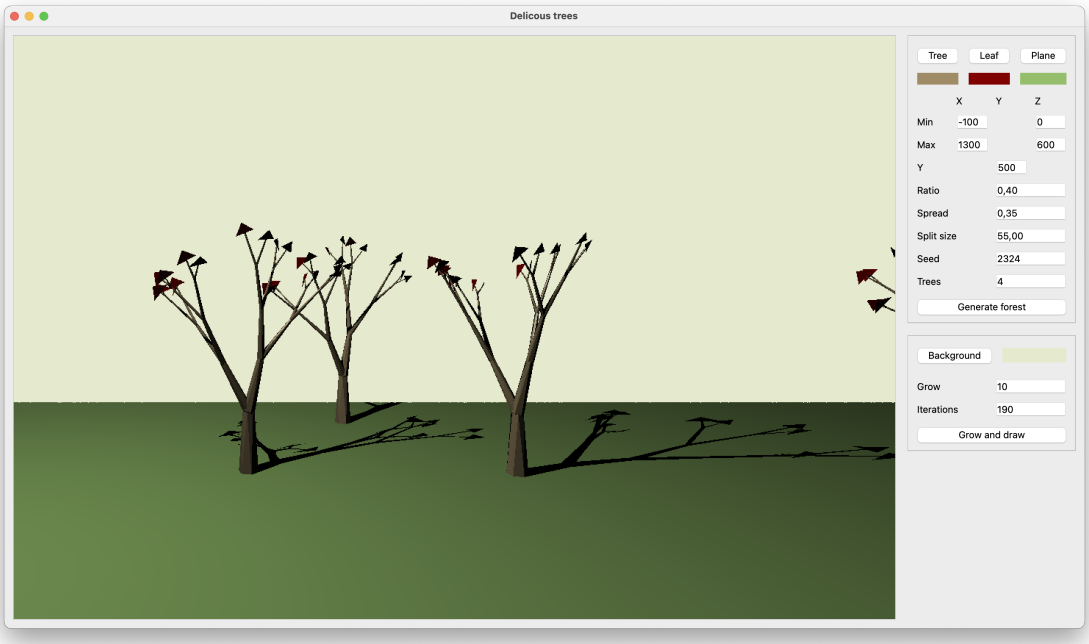

Рисунок 3.7 — Небольшой лес

3.9. Выводы по технологической части

В данном разделе была рассмотрена реализация алгоритмов обратной трассировки лучей и генерации лесистой местности. Был разобран интерфейс программы, флаги командной строки, модульное тестирование. Кроме того, были показаны примеры работы программы на различных входных данных.

4. Исследовательская часть

В данном разделе описаны замерные исследования и представлены их результаты.

4.1. Технические характеристики

Технические характеристики устройства, на котором выполнялся эксперимент [15]:

— 96 ГБ оперативной памяти;

— процессор Intel Ice Lake (96 ядер);

— операционная система Ubuntu 20.04.

4.2. Постановка исследования

Целью исследования является определение зависимости времени генерации изображения лесистой местности от количества потоков, выделенных на выполнение данной задачи. Исследование проводилось для количества потоков, не превышающих 400, так как уже на этом значении скорость работы алгоритма не увеличивается с увеличением числа потоков. Во время проведения исследования устройство было подключено к блоку питания, и не нагружено программами, кроме операционной системы и средств разработки.

4.3. Средства исследования

Время синтеза изображения измеряется с использованием библиотеки `chrono` [19]. Измеряется реальное время, что особо важно с учетом того, проводится исследования параллельных вычислений.

4.4. Результаты исследования

Полученные результаты измерения описаны в таблице 4.1. Нулевое количество потоков означает отсутствие создания дополнительных потоков.

Таблица 4.1 — Результаты замеров времени (в мс)

Количество потоков	Время
0	28288
1	28817
2	24448
5	11734
10	6135
20	3214
30	2185
40	1741
60	1349
80	1100
100	1058
120	1004
140	1048
160	981
190	925
230	899
260	869
290	848
320	889
350	877
400	872

На рис. 4.1 приведен график, показывающий зависимость времени выполнения программы от количества выделенных потоков.

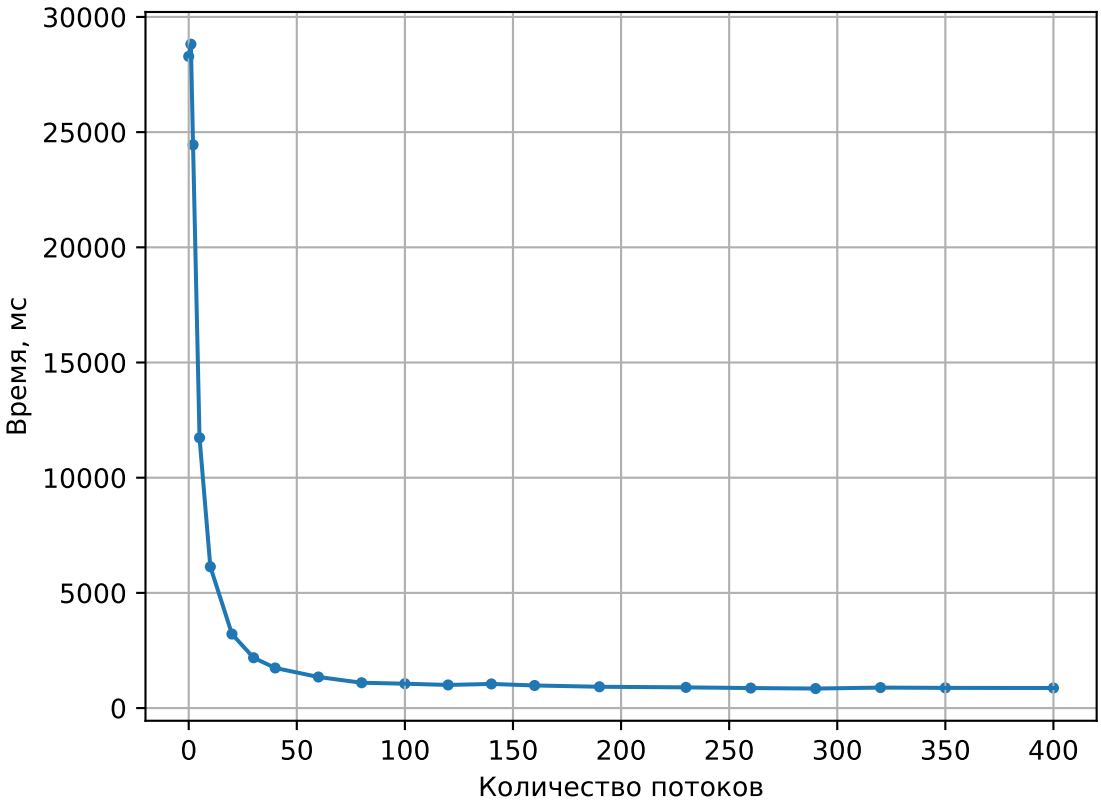

Рисунок 4.1 — График зависимости времени выполнения от количество потоков

4.5. Выводы по исследовательской части

В данном разделе было описано замерное исследование зависимости скорости работы алгоритма обратной трассировки лучей от количества потоков. По результатам замеров можно сказать, что улучшение скорости работы происходит при повышении количества потоков до 96, что равно количеству ядер на устройстве. Далее, улучшения производительности не наблюдается, так как часть потоков уже перестает выполняться параллельно.

Заключение

В ходе выполнения курсовой работы была достигнута поставленная цель: разработать программу для построения изображения лесистой местности.

Также реализованы все поставленные задачи:

— были изучены алгоритмы генерации местности;

— были изучены методы преобразования сгенерированной местности в модель для построения;

— были изучены алгоритмы построения реалистичных изображений;

— были формализованы объекты сцены;

— были выбраны средства разработки, позволяющие решить поставленные задачи;

— были реализованы алгоритмы обратной трассировки лучей и генерации леса.

Разработанную программу можно улучшить, добавив такой функционал, как генерация ландшафта, движение деревьев под действием ветра и прочее.

Список использованных источников

1. Боресков А. В., Шикин Е. В. Компьютерная графика. – 2017.
2. Бутенков С. А., Семерий О. С. Аналитический подход к решению задач компьютерной графики //Искусственный интеллект. – 2000. – Т. 3. – С. 428-437.
3. ОШАРОВСКАЯ Е. В., СОЛОДКА В. И. Синтез трехмерных объектов с помощью полигональных сеток //Сборник научных трудов «Цифровые технологии». – 2012. – №. 12.
4. Прилепко М. А. Математические модели представления компьютерной графики //Динамика систем, механизмов и машин. – 2012. – №. 5. – С. 306-312.
5. Царькова Ю. Р., Внукова О. В. ПОЛИГОНАЛЬНЯ СЕТКА В КОМПЬЮТЕРНОЙ ГРАФИКЕ //Студенчество России: век XXI. – 2020. – С. 394-398.
6. Ульянов А. Ю., Котюжанский Л. А., Рыжкова Н. Г. Метод трассировки лучей как основная технология фотореалистичного рендеринга //Фундаментальные исследования. – 2015. – №. 11-6. – С. 1124-1128.
7. Куров А. В. Конспект курса лекций по компьютерной графике, 2022.
8. Роджерс Д. Алгоритмические основы машинной графики. – Москва: Издательство «Мир». Редакция литературы по математическим наукам, 2001.
9. Ульянов А. Ю., Котюжанский Л. А., Рыжкова Н. Г. Метод трассировки лучей как основная технология фотореалистичного рендеринга //Фундаментальные исследования. – 2015. – №. 11-6. – С. 1124-1128.
10. Ваншина Е. А., Ваншин В. В. Типы моделей и принципы моделирования в преподавании графических дисциплин. – 2017.
11. Цапко И. В., Цапко С. Г. Алгоритмы и методы обработки информации в задачах трехмерного сканирования объектов //Известия Томского политехнического университета. Инжиниринг георесурсов. – 2010. – Т. 317. – №. 5.
12. Романюк А. Н. и др. Алгоритмы построения теней. – 2000.

13. Тюкачев Н. А. Алгоритм определения принадлежности точки многоугольнику общего вида или многограннику с треугольными гранями //Вестник Тамбовского государственного технического университета. – 2009. – Т. 15. – №. 3. – С. 638-652.

14. John Miano, Compressed Image File Formats.

15. Yandex Compute Cloud [Электронный ресурс]. Режим доступа: *https://cloud.yandex.ru/services/compute* (дата обращения: 04.11.2022).

16. Xu L., Mould D. A procedural method for irregular tree models //Computers & Graphics. – 2012. – Т. 36. – №. 8. – С. 1036-1047.

17. Yang Y., Wang R., Huo Y. Rule-based Procedural Tree Modeling Approach //arXiv preprint arXiv:2204.03237. – 2022.

18. Усольцев В. А. Биоэкологические аспекты таксации фитомассы деревьев. – 1997.

19. Date and time utilities [Электронный ресурс]. Режим доступа: *https://en.cppreference.com/w/cpp/chrono* (дата обращения: 06.11.2022).