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»

ОТЧЕТ

По лабораторной работе №4

По курсу: «Анализ алгоритмов»

Тема: «Параллельные вычисления»

Студент: Пронин А. С.

Группа: ИУ7-52Б

Преподаватель: Волкова Л. Л.

Оценка: _____

Москва

2021

Содержание

Введение

Цель работы – изучение параллельных вычислений с помощью потоков. В данной лабораторной работе будет рассмотрена многопоточная работа (параллельные вычисления) для реализации алгоритма ZBuffer.

Задачи работы:

- изучить распараллеливание вычислений и работу с потоками;
- реализовать стандартный алгоритм ZBuffer, а также с применением параллеленных вычислений;
- экспериментально сравненить время работы алгоритма на разном количестве потоков.

1 Аналитический раздел

Многопоточность — способность центрального процессора (CPU) или одного ядра в многоядерном процессоре одновременно выполнять несколько процессов или потоков, соответствующим образом поддерживаемых операционной системой.

Этот подход отличается от многопроцессорности, так как многопоточность процессов и потоков совместно использует ресурсы одного или нескольких ядер: вычислительных блоков, кэш-памяти ЦПУ или буфера перевода с преобразованием (TLB).

В тех случаях, когда многопроцессорные системы включают в себя несколько полных блоков обработки, многопоточность направлена на максимизацию использования ресурсов одного ядра, используя параллелизм на уровне потоков, а также на уровне инструкций.

Поскольку эти два метода являются взаимодополняющими, их иногда объединяют в системах с несколькими многопоточными ЦП и в ЦП с несколькими многопоточными ядрами.

Многопоточная парадигма стала более популярной с конца 1990-х годов, поскольку усилия по дальнейшему использованию параллелизма на уровне инструкций застопорились.

Смысл многопоточности — квазимногозадачность на уровне одного исполняемого процесса.

Значит, все потоки процесса помимо общего адресного пространства имеют и общие дескрипторы файлов. Выполняющийся процесс имеет как минимум один (главный) поток.

Многопоточность (как доктрину программирования) не следует путать ни с многозадачностью, ни с многопроцессорностью, несмотря на то, что операционные системы, реализующие многозадачность, как правило, реализуют и многопоточность.

Достоинства:

- облегчение программы посредством использования общего адресного пространства;

- меньшие затраты на создание потока в сравнении с процессами;
- повышение производительности процесса за счёт распараллеливания процессорных вычислений;
- если поток часто теряет кэш, другие потоки могут продолжать использовать неиспользованные вычислительные ресурсы.

Недостатки:

- несколько потоков могут вмешиваться друг в друга при совместном использовании аппаратных ресурсов;
- с программной точки зрения аппаратная поддержка многопоточности более трудоемка для программного обеспечения;
- проблема планирования потоков;
- специфика использования. Вручную настроенные программы на ассемблере, использующие расширения MMX или AltiVec и выполняющие предварительные выборки данных, не страдают от потерь кэша или неиспользуемых вычислительных ресурсов. Таким образом, такие программы не выигрывают от аппаратной многопоточности и действительно могут видеть ухудшенную производительность из-за конкуренции за общие ресурсы.

Однако несмотря на количество недостатков, перечисленных выше, многопоточная парадигма имеет большой потенциал на сегодняшний день и при должном написании кода позволяет значительно ускорить однопоточные алгоритмы.

1.1 Описание задачи

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.

В данной лабораторной работе мы рассмотрим алгоритм ZBuffer.

1.2 Описание алгоритма

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).

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.

1.3 Вывод

Стандартный алгоритм ZBuffer обрабатывает объекты индивидуально, что предоставляет возможность для реализации алгоритма с применений параллельных вычислений.

2 Конструкторский раздел

В данном разделе представлены схемы алгоритма ZBuffer реализованного стандартным методом, а также с применением параллеленных вычислений.

2.1 Схемы алгоритмов

Ниже представлены схемы следующих алгоритмов:

- стандартный ZBuffer (рисунок 2.1);
- ZBuffer с применением параллеленных вычислений (рисунок 2.2);

2.2 Вывод

В данном разделе были разработаны схемы алгоритмов ZBuffer реализованного стандартным методом и с применением параллеленных вычислений.

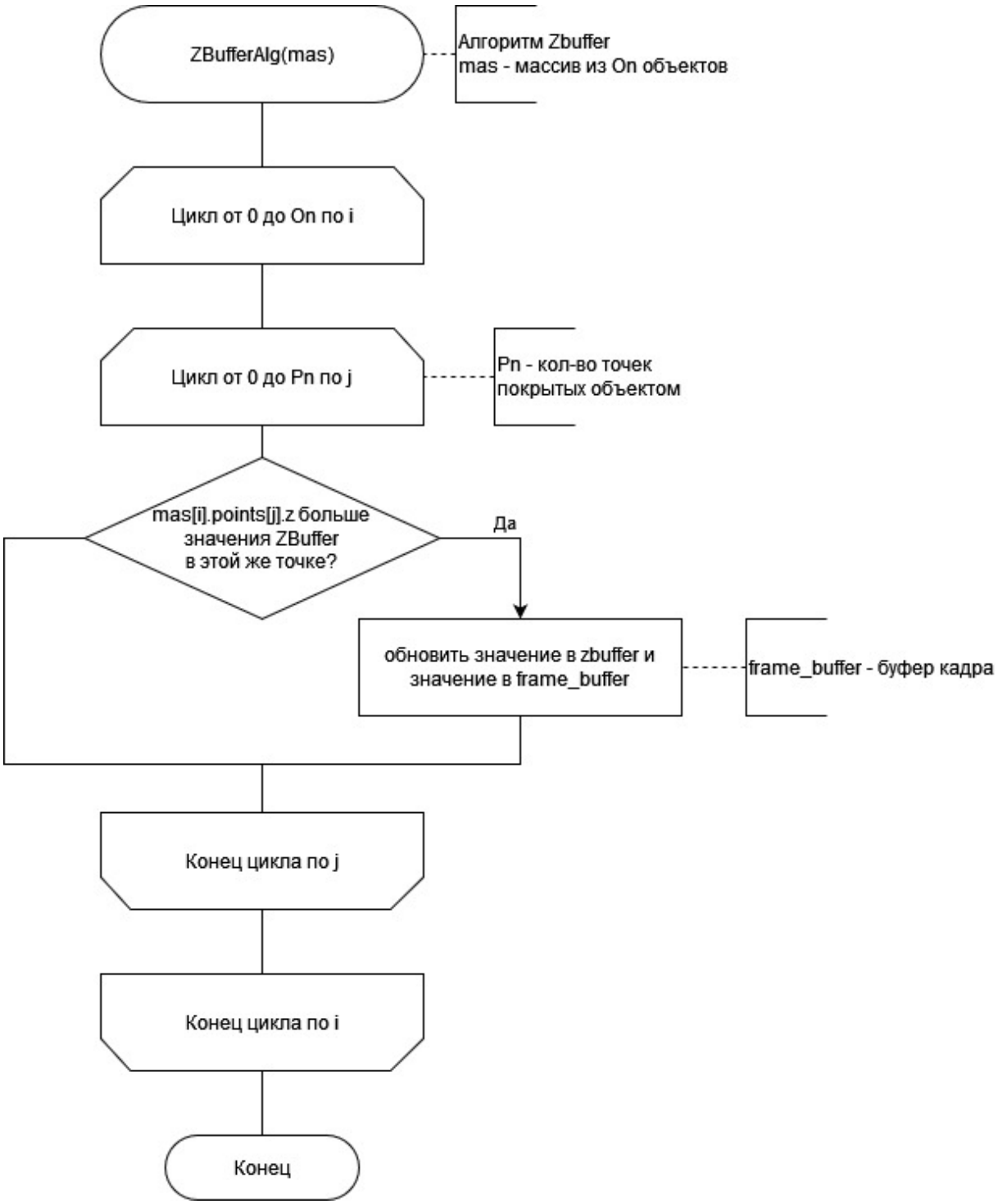

Рис. 2.1: Схема алгоритма ZBuffer

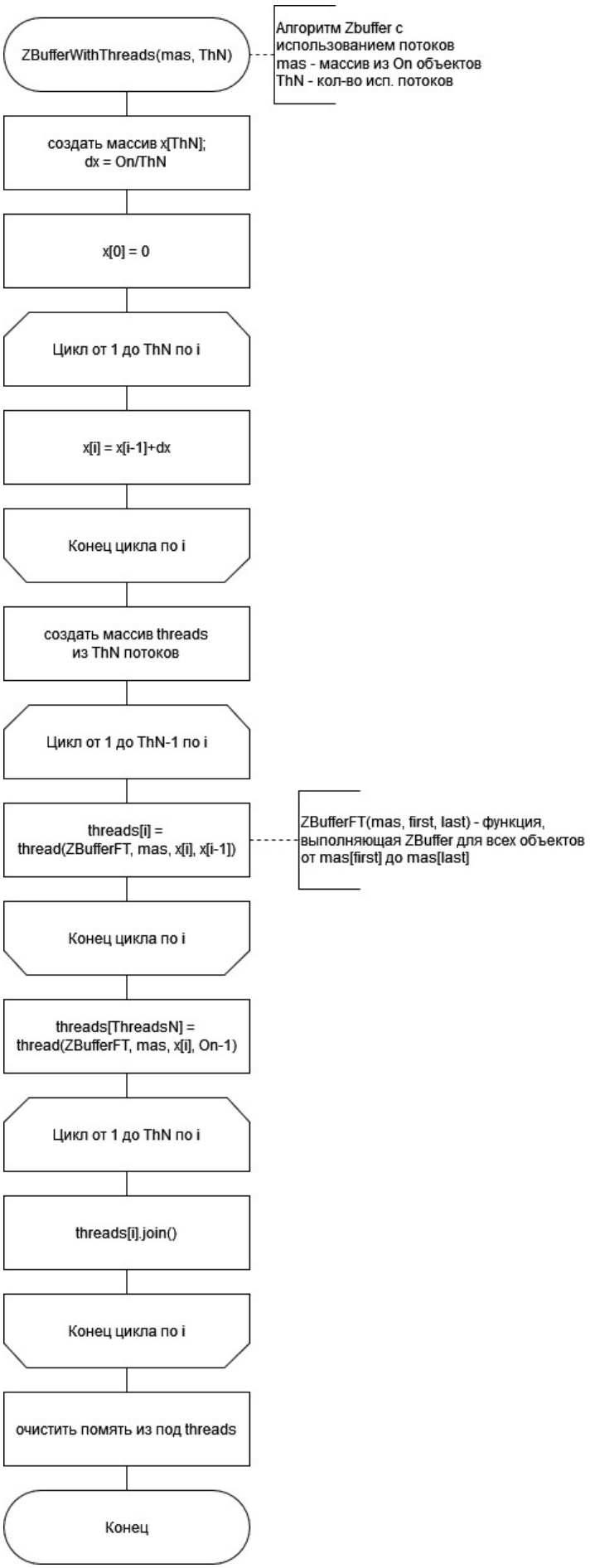

Рис. 2.2: Схема алгоритма ZBufferWithThreads

3 Технологический раздел

В данном разеделе представлены выбор инструментов для реализации и оценки алгоритмов, а также листинги полученного кода.

3.1 Требование к ПО

К программе предъявляется ряд требований:

- на вход подаётся объекта класса TriPolArray, состоящий из объектов (треугольных полигонов);
- на выходе — заполненные ZBuffer и FrameBuffer, содержащие соответственно глубины и цвета всех видимых точек на экране.

3.2 Выбор инструментов

По-скольку наиболее освоенным языком для разработчика является c++, для реализации алгоритмов был выбран именно он, т.к. таким образом работа будет проделана наиболее быстро и качественно.

Соответсвенно для компиляции кода будет использоваться компилятор G++.

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

Кроме этого, необходимо отключить оптимизации компилятора для более честного сравнения алгоритмов. В моём случае это делается с помощью ключа $-O0$ т.к. используется компилятор G++. [1]

3.3 Реализация алгоритмов

На листингах 3.1-3.2 представлены реализации алгоритма ZBuffer без и с использованием параллельных вычислений.

Листинг 3.1: Обычный алгоритм ZBuffer

```
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 3.2: Многопоточный алгоритм ZBuffer

```
void ZBufferAlg::executeWithThreads(TriPolArray &mas, int threadsN)
{
    int red = mas.getR();
    int green = mas.getG();
    int blue = mas.getB();
    zbuffer->reset();
    frame_buffer->reset();

    //Sizes prep
    int size = mas.size();

    int x[threadsN];
    int dx = size/threadsN;
    x[0] = 0;
    for (int i = 1; i < threadsN; i++)
    {
        x[i] = x[i-1] + dx;
    }

    std::thread *th = new std::thread[threadsN];
    for (int i = 0; i < threadsN-1; i++)
    {
        th[i] = std::thread(&ZBufferAlg::executeFT, this, ref(mas), x[i], x[i+1], red,
            green, blue);
    }
    th[threadsN-1] = std::thread(&ZBufferAlg::executeFT, this, ref(mas),
        x[threadsN-1], size - 1, red, green, blue);

    for (int i = 0; i < threadsN; i++)
    {
        th[i].join();
    }
    delete[] th;
}
```

3.4 Тестирование

Для проверки написанных алгоритмов были подготовлены следующие тесты:

- проверка алгоритма ZBuffer
- проверка трёхмерного переноса

- проверка трёхмерного масштабирования
- проверка трёхмерного поворота

На рисунках 3.1-3.4 приведены результаты тестирования.

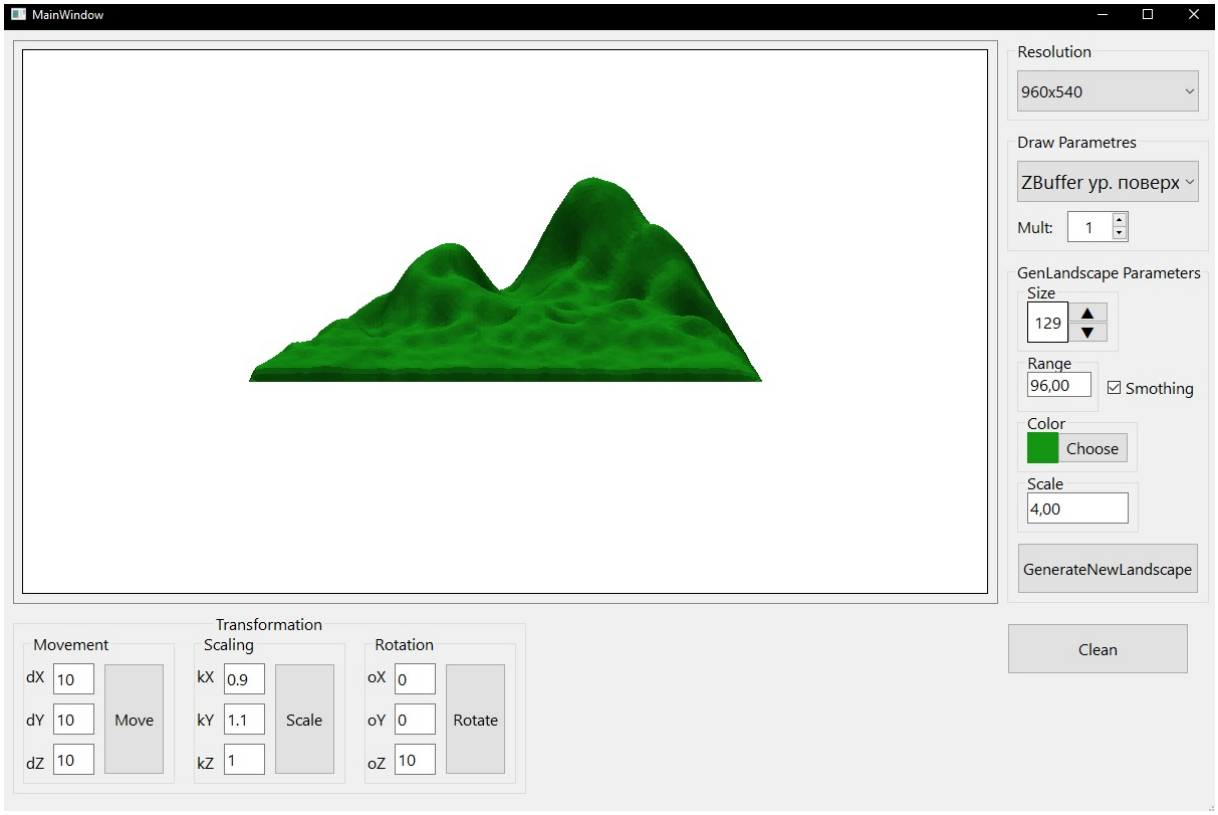

Рис. 3.1: Начальное положение ландшафта

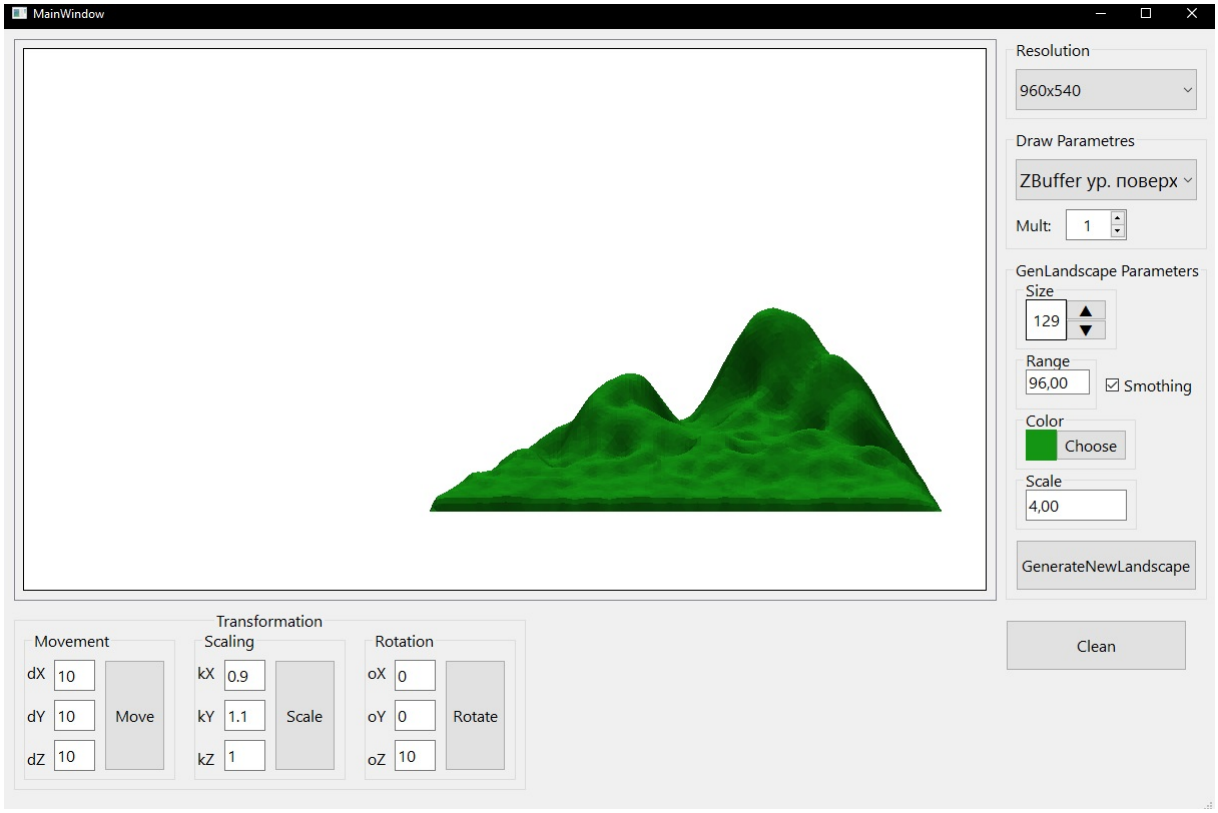

Рис. 3.2: Трёхмерный перенос ландшафта

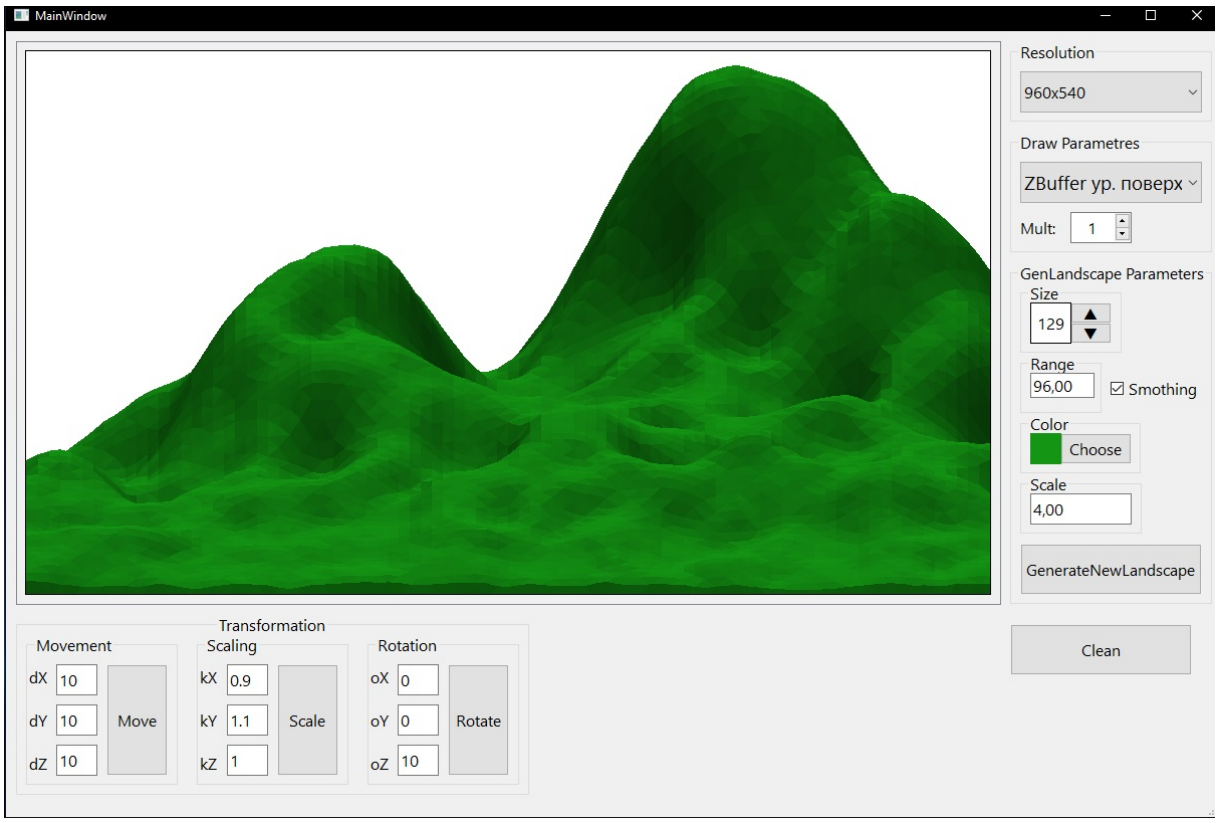

Рис. 3.3: Трёхмернон масштабирование ландшафта

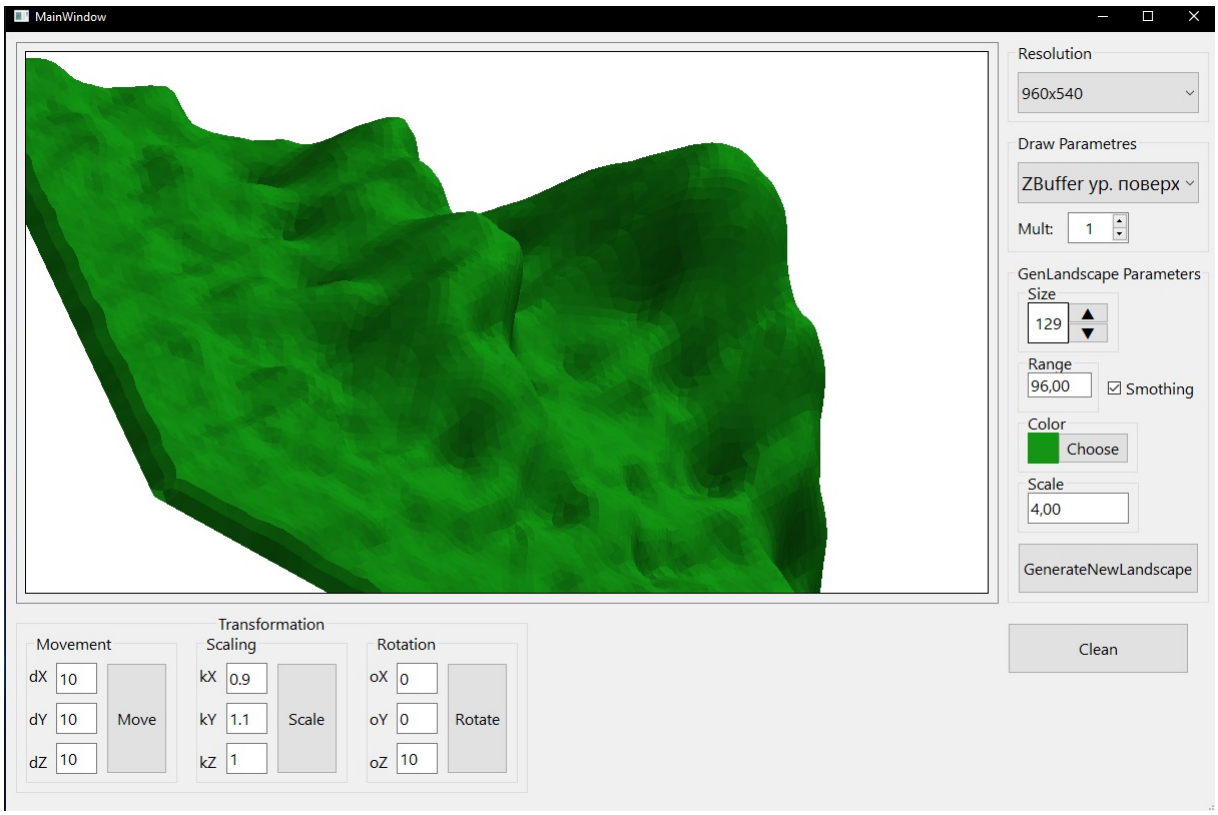

Рис. 3.4: Трёхмерный поворот ландшафта

Как видно по рисункам, все тесты пройдены.

3.5 Вывод

В данном разделе были выбраны инструменты для реализации выбранных алгоритмов, представлены листинги реализованных алгоритмов, а также проведено тестирование.

4 Исследовательский раздел

В данном разделе представлены технические характеристики и сравнительный анализ реализованных алгоритмов.

4.1 Технические характеристики

Ниже приведены технические характеристики устройства, на котором было проведено тестирование ПО:

- операционная система: Windows 10 (64-разрядная);
- оперативная память: 32 GB;
- Процессор: Intel(R) Core(TM) i7-7700K CPU @ 4.20GHz;
- кол-во логических процессоров: 4.

STOPED HERE!!

4.2 Сравнительный анализ времени выполнения алгоритмов

Чтобы провести сравнительный анализ времени выполнения алгоритмов замерялось процессорное время для массивов с 100, 200, ... 1000 элементами. Чтобы оценить время выполнения сортировки для массива размера N, он заполнялся числами от 0 $N^{10} - 1$, замерялось процессорное время для части кода, которая сортировала массивы 500000/N раз, после чего резултат делился на кол-во итераций.

Сравнительный анализ проводилось на компьютере с процессором AMD Ryzen 5 5600H.

Для сортировки пузырьком наихудшим случаем является массив отсортированный в обратном порядке. Наилучшим случаем является полностью отсортированный массив. На рисунке 4.4 изображены зависимости

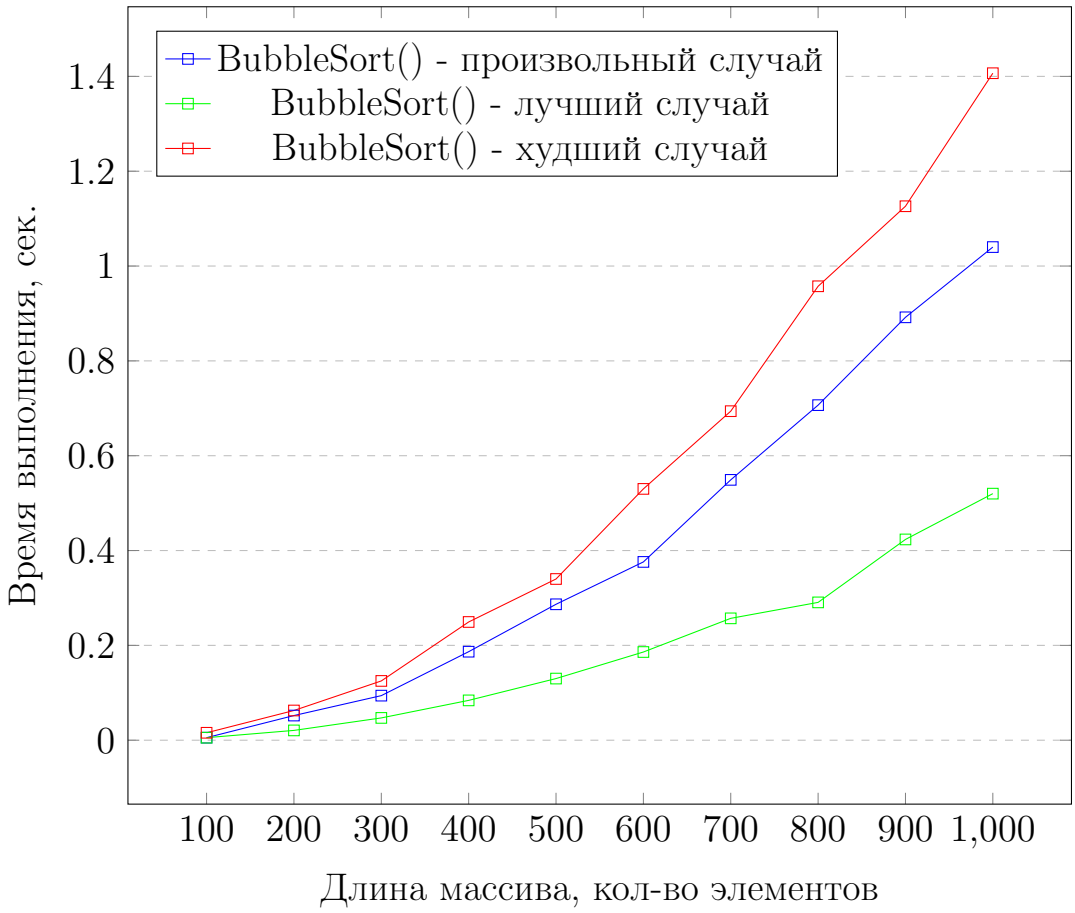

Рис. 4.1: Зависимость времени выполнения сортировки пузырьком от длины массива в разных случаях

времени выполнения сортировки от длины массива для произвольного, лучшего и худшего случаев. [2]

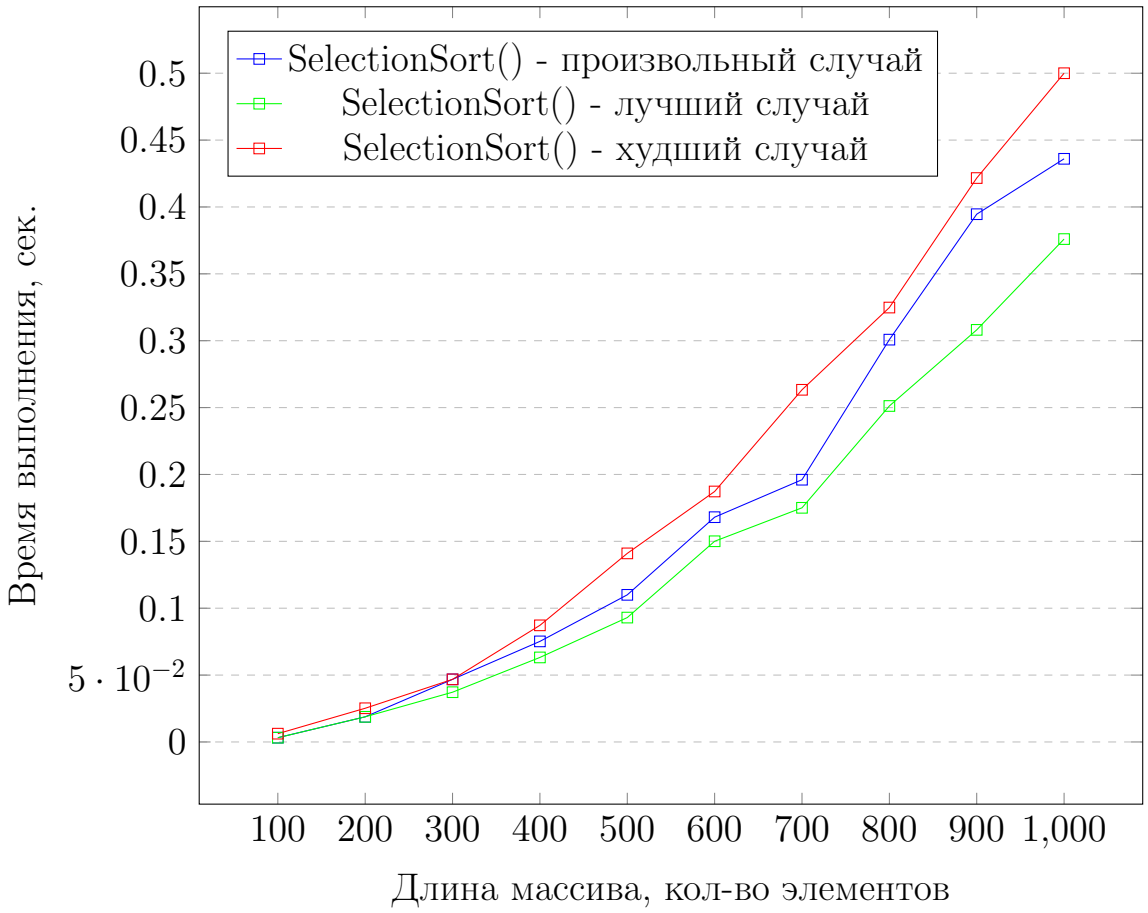

Рис. 4.2: Зависимость времени выполнения сортировки выбором от длины массива в разных случаях

Для сортировки выбором наихудшим случаем является массив отсортированный в обратном порядке. Наилучшим случаем является полностью отсортированный массив. На рисунке 4.5 изображены зависимости времени выполнения сортировки от длины массива для произвольного, лучшего и худшего случаев. [2]

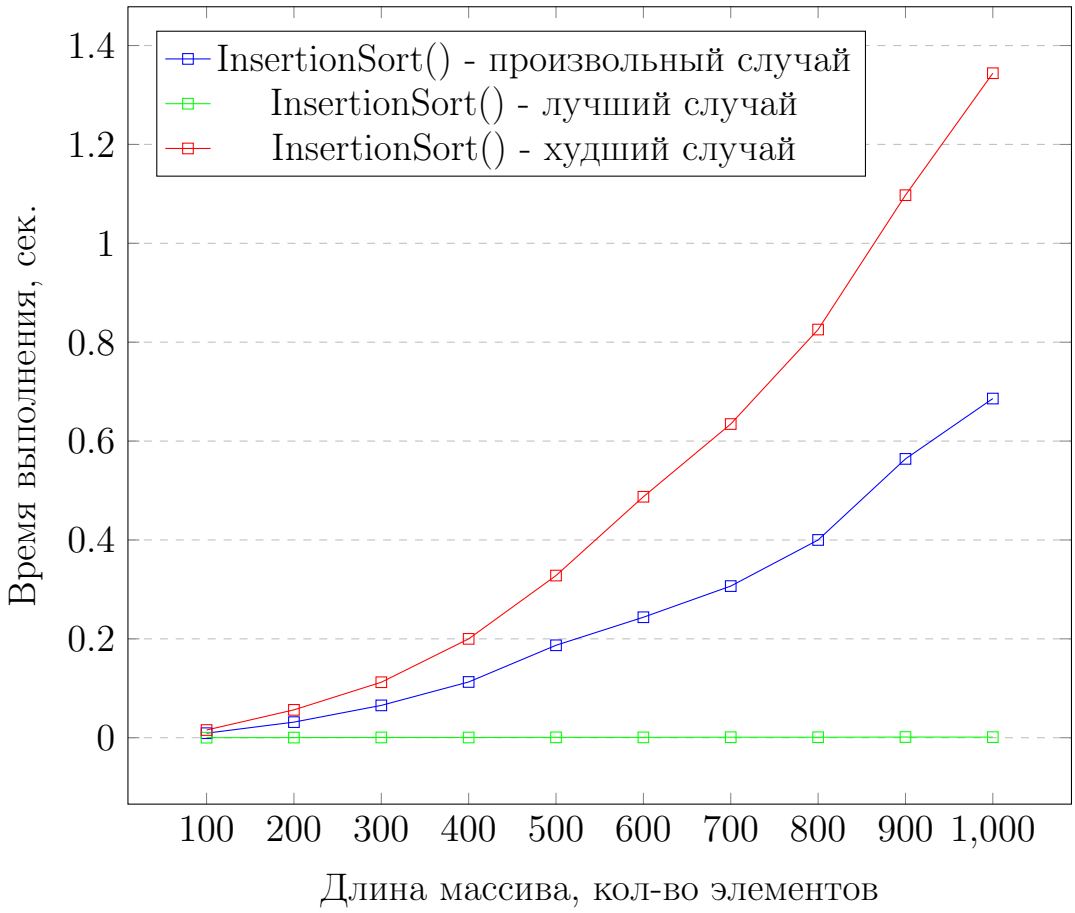

Рис. 4.3: Зависимость времени выполнения сортировки выбором от длины массива в разных случаях

Для сортировки вставками наихудшим случаем является массив отсортированный в обратном порядке. Наилучшим случаем является полностью отсортированный массив. На рисунке 4.6 изображены зависимости времени выполнения сортировки от длины массива для произвольного, лучшего и худшего случаев. [2]

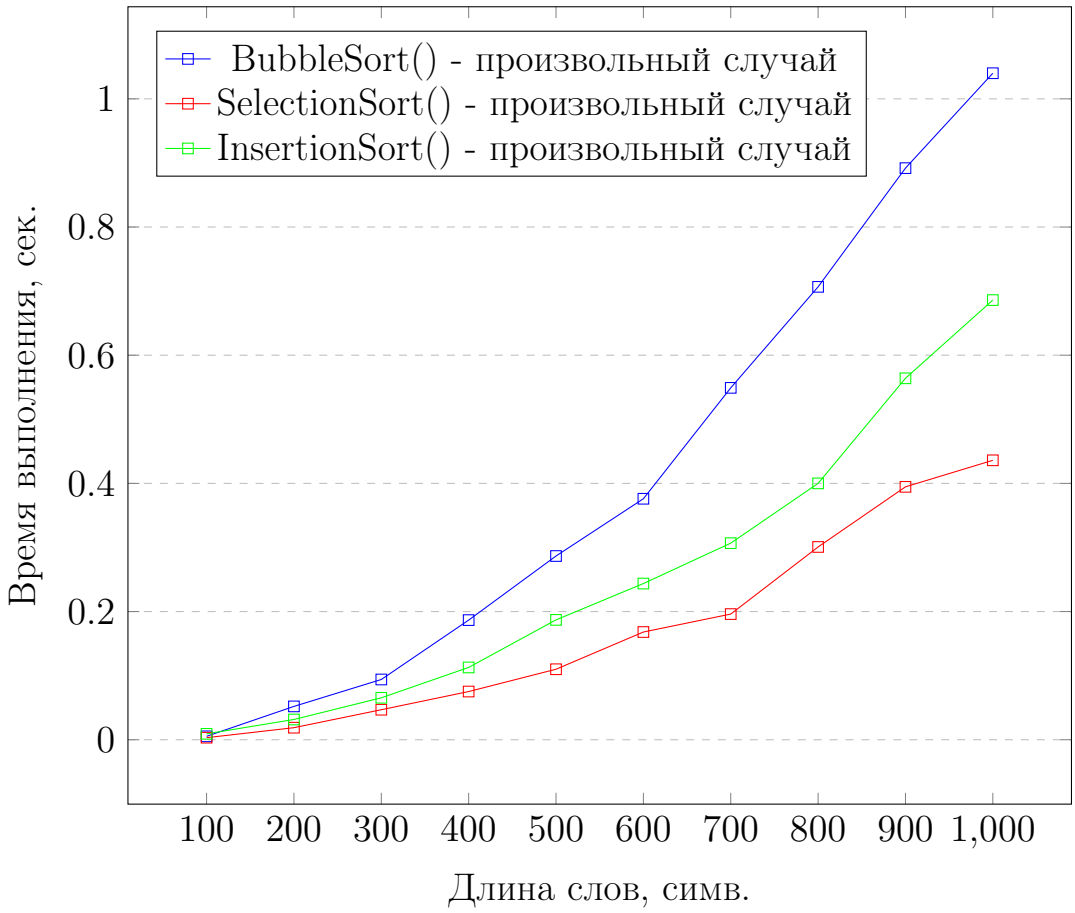

Рис. 4.4: Зависимость времени выполнения алгоритмов сортировок от длины массива в произвольном случае

Также приведены графики (рисунки 4.7-4.9) для сравнения алгоритмов сортировок между собой в произвольном, лучшем и худшем случаях.

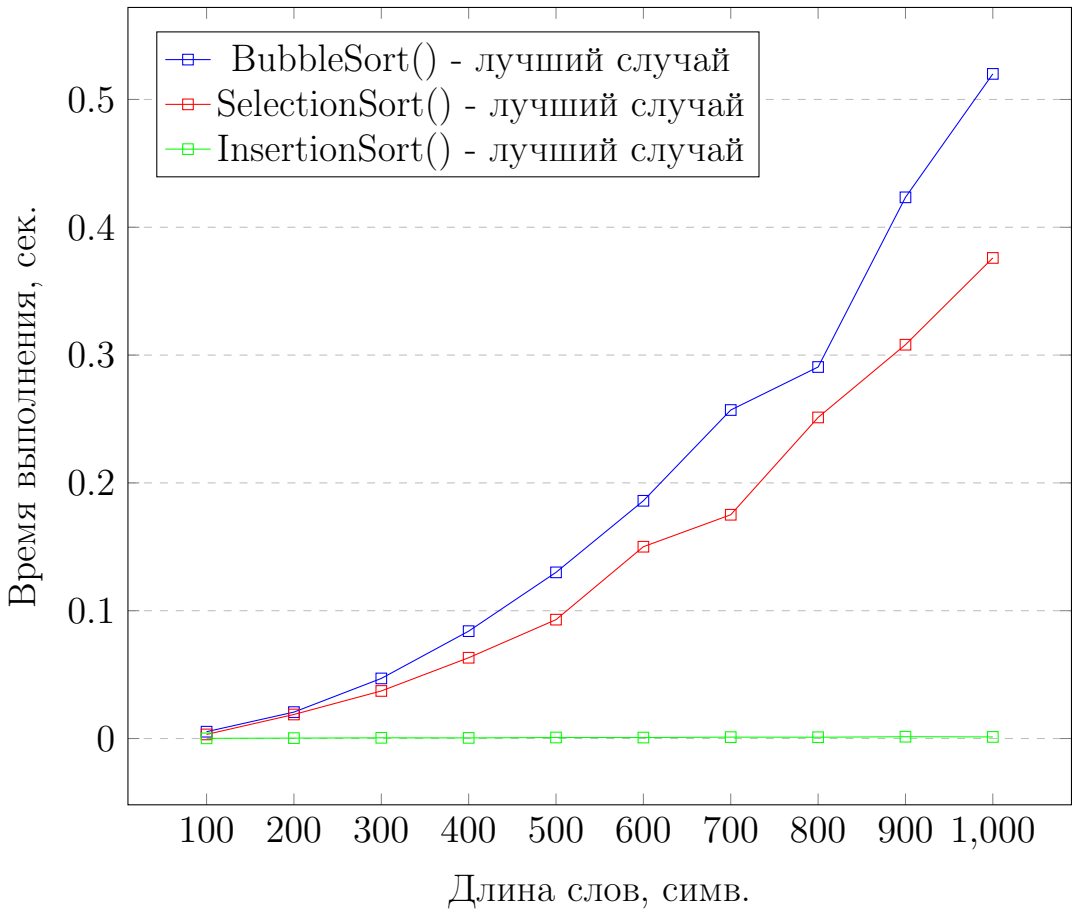

Рис. 4.5: Зависимость времени выполнения алгоритмов сортировок от длины массива в лучшем случае

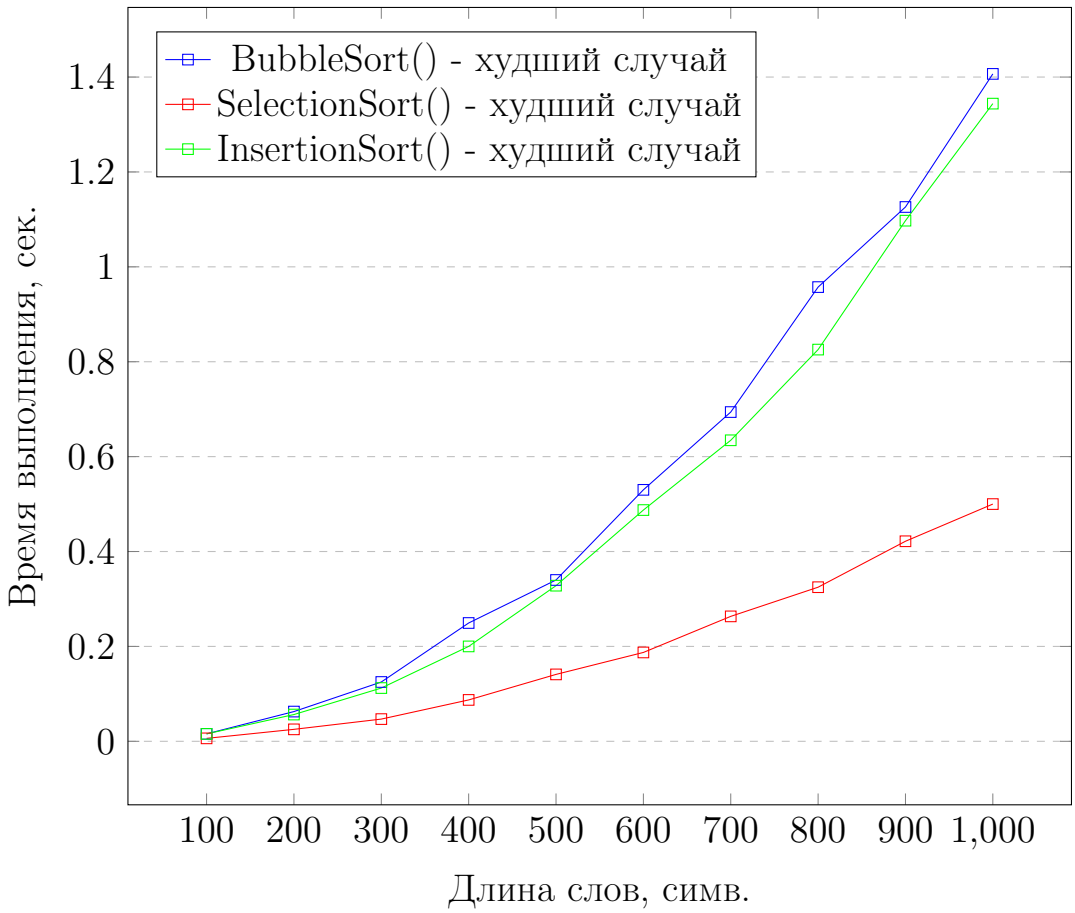

Рис. 4.6: Зависимость времени выполнения алгоритмов сортировок от длины массива в худшем случае

4.3 Оценка трудоёмкости

Для оценки трудоёмкости использовалась следующая модель вычислений: [3]

- Трудоёмкость следующих операций единична: +, -, =, +=, -=, ==, !=, <, >, -=, >=, «, », [];
- Трудоёмкость следующих операций = 2: *, /, %, /=, *=.

Трудоёмкость выбранных алгоритмов сортировок рассчитывалась по написанному коду.

На листинге 4.1 представлена программа для вычисления трудоёмкости алгоритма сортировки пузырьком для худшего случая.

Листинг 4.1: Вычисление трудоёмкости алгоритма сортировки пузырьком

```
int getBubbleSort(int *l, int *r)
{
        int rez = 3; //init+srav
        for (int i = 0; i < r-l; i++)
        {
                rez += 3; //init+srav
                for (int *j = l; j < r-i; j++)
                {
                        if (*j > *(j+1))
                                swap(j, (j+1));
                        rez+=5; //telo j
                        rez++; //increment
                        rez+=2; //srav
                }
                rez++; //increment
                rez+=2; //srav
        }
        return rez;
}
```

Соответственно получается следующая формула трудоёмокости:

$Fbubblesort = 3 + (3 * (N - 1) + ((N - 1) * N/2) * 8) + 3 * (N - 1)$, где N - размер массива

В лучшем же случае, не надо будет менять элементы местами, а значит трудоёмкость тела цикла по j уменьшится на 3 и формула примет вид:

$Fbubble\_sort = 3 + (3 * (N - 1) + ((N - 1) * N/2) * 5) + 3 * (N - 1)$

На листинге 4.2 представлена программа для вычисления трудоёмкости алгоритма сортировки выбором для худшего случая.

Листинг 4.2: Вычисление трудоёмкости алгоритма сортировки выбором

```
int getSelectionSort(int *l, int *r)
{
        int rez = 2; //init+srav
        for (int *i = l; i <= r; i++)
        {
                rez += 2; //double = (assignment)
                int minz = *i, *ind = i;
                rez += 3; //init+srav
                for (int *j = i + 1; j <= r; j++)
                {
                        if (*j < minz)
                        {
                                minz = *j;
                                ind = j;
                        }
                        rez += 3; //telo j
                        rez++; //increment
                        rez++; //srav
                }
                rez+=3; //swap
                swap(i, ind);
                rez++; //increment
                rez++; //srav
        }
        return rez;
}
```

Следовательно формула трудоёмкости будет следующей:

$$Fselection\_sort = 2 + 10 * N + ((N - 1) * N/2) * 5$$

А в лучшем случае не будет выполнятся условие if и формула станет такой:

$$Fselection\_sort = 2 + 10 * N + ((N - 1) * N/2) * 3$$

На листинге 4.3 представлена программа для вычисления трудоёмкости алгоритма сортировки вставками для худшего случая.

Листинг 4.3: Вычисление трудоёмкости алгоритма сортировки вставками

```
int getInsertionSort(int* l, int* r)
{
        int rez = 3; //init+srav
        for (int *i = l + 1; i <= r; i++)
        {
                rez++; //assigment
                int* j = i;
                rez+=3; //srav
                while (j > l && *(j - 1) > *j)
                {
                        rez+=4; //swap
                        swap((j - 1), j);
                        j--;
                        rez++; //deccrement
                        rez+=3; //srav
                }
        }
        return rez;
}
```

Формула трудоёмкости:

$Finsertion\_sort = 3 + 4 * (N - 1) + 8 * ((N - 1) * N/2)$

В лучшем случае полностью пропадает тело цикла while, а значит формула изменится на следующую:

$Finsertion\_sort = 3 + 4 * (N - 1)$

4.4 Вывод

По итогу иследования выяснилось, что разработанная программа работает верно, то-есть сортирует массивы по возрастанию. Кроме этого, смотря на время выполнения каждого алгоритма, логично сделать вывод, что наиболее быстрым в произвольном случае, является алгоритм сортировки выбором и судя по оценке трудоёмкости, наименее трудоёмким является также алгоритм сортировки выбором.

Заключение

По итогу проделанной работы была достигнута цель - изучены алгоритмы сортировки и получены навыки оценки трудоемкости алгоритмов.

Также были решены все поставленные задачи, а именно:

- реализованы 3 выбранных алгоритма сортировки;
- выполнена оценка времени выполнения алгоритмов сортировки;
- рассчитана трудоемкость каждого из алгоритма сортировки.

Список использованных источников

[1] Как применить настройки оптимизации gcc в qt? // URL: http://blog.kislenko.net/show.php?id=1991.

[2] Опанасенко М. Описание алгоритмов сортировки и сравнение их производительности [Электронный ресурс]. // URL: https://habr.com/ru/post/335920/.

[3] Ульянов М.В. *Ресурсно-эффективные компьютерные алгоритмы. Разработка и анализ.* Москва: ФИЗМАТЛИТ, 2008. 304 с.