



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 на основе нативных потоков

Студент Гурова Н.А.

Группа ИУ7-54Б

Оценка (баллы)

Преподаватель Волкова Л.Л., Строганов Ю.В.

Оглавление

Введение	3
1 Аналитическая часть	4
1.1 Многопоточность	4
1.2 Алгоритм Z-буфера	5
2 Конструкторская часть	6
2.1 Требования к вводу	6
2.2 Требования к программе	6
2.3 Разработка алгоритмов	6
2.4 Работа с разделяемой памятью	6
3 Технологическая часть	11
3.1 Средства реализации	11
3.2 Сведения о модулях программы	11
3.3 Реализация алгоритмов	11
3.4 Функциональные тесты	15
4 Исследовательская часть	17
4.1 Технические характеристики	17
4.2 Время выполнения алгоритмов	17
4.2.1 Время выполнения однопоточной реализации	18
4.2.2 Время выполнения многопоточной реализации	18
Заключение	20
Список использованных источников	21

Введение

Существуют различные способы написания программ, одним из которых является использование параллельных вычислений. Параллельные вычисления – способ организации компьютерных вычислений, при котором программы разрабатываются как набор взаимодействующих вычислительных процессов, работающих параллельно (одновременно) [1].

Основная цель параллельных вычислений – уменьшение времени решения задачи. Многие необходимые для нужд практики задачи требуется решать в реальном времени или для их решения требуется очень большой объем вычислений. Таким трудоемким алгоритмом является, например, алгоритм Z-буфера - один из методов построения реалистичных сцен.

Целью данной работы является изучение организации параллельных вычислений на базе алгоритма Z-буфера.

В рамках выполнения лабораторной работы необходимо решить следующие задачи:

- 1) изучить основы параллельных вычислений;
- 2) изучить алгоритм Z-буфера;
- 3) разработать последовательную версию алгоритма Z-буфера;
- 4) разработать параллельную версию алгоритма Z-буфера;
- 5) провести сравнительный анализ времени работы реализаций.

1 Аналитическая часть

В этом разделе было рассмотрено понятие многопоточности и представлено описание алгоритма Z-буфера.

1.1 Многопоточность

Многопоточность [2] – способность центрального процессора или одного ядра в многоядерном процессоре одновременно выполнять несколько процессов или потоков, соответствующим образом поддерживаемых операционной системой.

Процесс – это программа в ходе своего выполнения. Когда мы выполняем программу или приложение, запускается процесс. Каждый процесс состоит из одного или нескольких потоков.

Поток – это сегмент процесса. Потоки представляют собой исполняемые сущности, которые выполняют задачи, стоящие перед исполняемым приложением. Процесс завершается, когда все потоки заканчивают выполнение.

Каждый поток в процессе – это задача, которую должен выполнить процессор. Большинство процессоров сегодня умеют выполнять одновременно две задачи на одном ядре, создавая дополнительное виртуальное ядро. Это называется одновременная многопоточность или многопоточность *Hyper – Threading*, если речь о процессоре от Intel.

Эти процессоры называются многоядерными процессорами. Таким образом, двухъядерный процессор имеет 4 ядра: два физических и два виртуальных. Каждое ядро может одновременно выполнять только один поток.

Как упоминалось выше, один процесс содержит несколько потоков, и одно ядро процессора может выполнять только один поток за единицу времени. Если мы пишем программу, которая запускает потоки последовательно, то есть передает выполнение в очередь одного конкретного ядра процессора, мы не раскрываем весь потенциал многоядерности. Остальные ядра просто стоят без дела, в то время как существуют задачи, которые необходимо выполнить. Если мы напишем программу таким образом, что она создаст несколько потоков для отнимающих много времени независи-

мых функций, то мы сможем использовать другие ядра процессора, которые в противном случае пылились бы без дела. Можно выполнять эти потоки параллельно, тем самым сократив общее время выполнения процесса.

1.2 Алгоритм Z-буфера

Суть данного алгоритма – это использование двух буферов: буфера кадра, в котором хранятся атрибуты каждого пикселя, и Z-буфера, в котором хранятся информация о координате Z для каждого пикселя.

Первоначально в Z-буфере находятся минимально возможные значения Z, а в буфере кадра располагаются пиксели, описывающие фон. Каждый многоугольник преобразуется в растровую форму и записывается в буфер кадра.

В процессе подсчета глубины нового пикселя, он сравнивается с тем значением, которое уже лежит в Z-буфере. Если новый пиксель расположен ближе к наблюдателю, чем предыдущий, то он заносится в буфер кадра и происходит корректировка Z-буфера.

Для решения задачи вычисления глубины Z каждый многоугольник описывается уравнением $ax + by + cz + d = 0$. При $c = 0$, многоугольник для наблюдателя вырождается в линию.

Преимуществами данного алгоритма являются простота реализации, а также линейная оценка трудоемкости.

Недостатки алгоритма - большой объем требуемой памяти и сложная реализация прозрачности.

Вывод

В данном разделе был рассмотрен алгоритм удаления невидимых граней, использующий Z-буфер, а также представлена информация по поводу многопоточности.

2 Конструкторская часть

В этом разделе была приведена схема последовательного алгоритма Z-буфера, а также схемы главного и рабочего потоков для параллельной реализации этого алгоритма. Кроме того, были описаны требования к вводу и программе.

2.1 Требования к вводу

На вход подается массив точек и связей между ними.

2.2 Требования к программе

Программа не должна аварийно завершаться при некорректном вводе. Вывод программы - матрица, в каждом узле которой находится значение цвета, соответствующее текущему состоянию сцены.

2.3 Разработка алгоритмов

На рисунках 2.1, 2.2 и 2.3 представлены схемы алгоритмов рабочего потока, потока диспетчера, и последовательного алгоритма удаления невидимых граней использующий Z-буфер.

2.4 Работа с разделяемой памятью

Алгоритм работает по принципу постолбцового развертывания изображения. Такая модификация гарантирует монополярный доступ к памяти при параллельном выполнении алгоритма, а также уменьшает количество требуемой для работы алгоритма памяти.

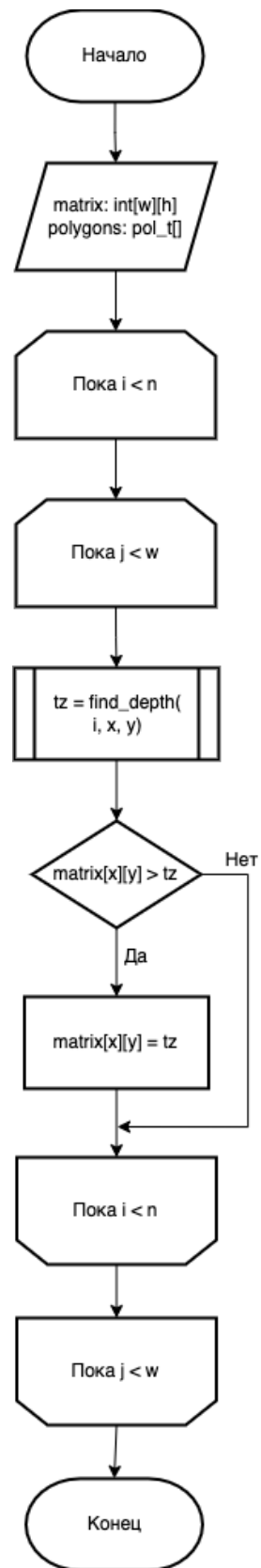


Рисунок 2.1 – Схема алгоритма рабочего потока

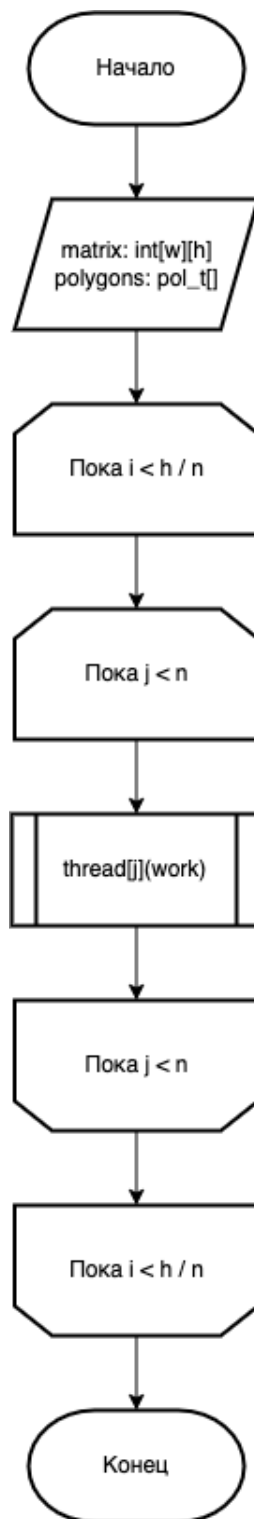


Рисунок 2.2 – Схема алгоритма потока диспетчера

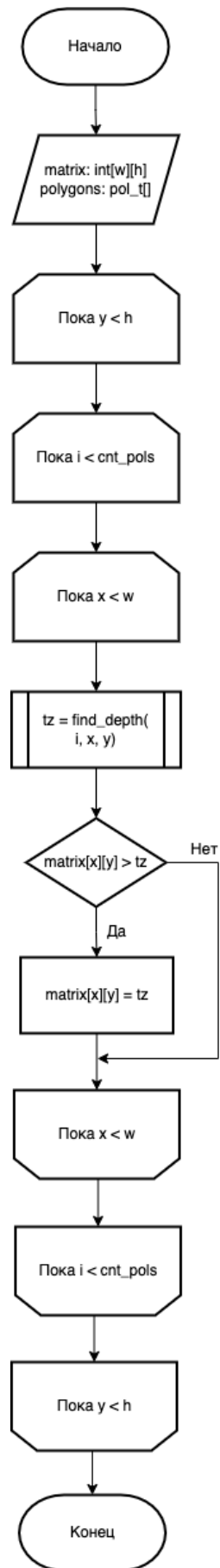


Рисунок 2.3 – Схема алгоритма линейной программы

Вывод

Была разработана схема последовательного алгоритма удаления невидимых граней. На ее основе была построена схема рабочего потока и схема потока диспетчера для многопоточной реализации алгоритма удаления невидимых граней.

3 Технологическая часть

В данном разделе были приведены средства реализации и листинги кода.

3.1 Средства реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран язык программирования C++ [3]. Данный язык имеет все необходимые инструменты для решения поставленной задачи.

3.2 Сведения о модулях программы

Программа состоит из трех модулей:

1. `z_buffer.cpp`, `z_buffer.h` – файлы, содержащие реализации алгоритма Z-буфера;
2. `tools.cpp`, `tools.h` – файлы, содержащие реализацию вспомогательных функций;
3. `server_test.cpp`, `server_test.h` – файлы, содержащие реализации тестовой системы.

3.3 Реализация алгоритмов

В листинге 3.1 представлена линейная реализация алгоритма удаления невидимых граней.

Листинг 3.1 – Алгоритм постолбцового Z-буфера

```
1 void process_level(triangle_t *triangle , screen_t *screen , int y ,  
  double *depth_arr) {  
2  
3     int max_x_ind = ind_of_max_of_axis(triangle , 0);  
4     int min_x_ind = ind_of_min_of_axis(triangle , 0);
```

```

5
6  int max_y_ind = ind_of_max_of_axis(triangle , 1);
7  int min_y_ind = ind_of_min_of_axis(triangle , 1);
8  int mid_y_ind = ind_of_mid_of_axis(triangle , 1);
9
10 double ymax = triangle->processed_vertexes[max_y_ind][1];
11 double ymin = triangle->processed_vertexes[min_y_ind][1];
12 double ymid = triangle->processed_vertexes[mid_y_ind][1];
13 double xmax = triangle->processed_vertexes[max_y_ind][0];
14 double xmin = triangle->processed_vertexes[min_y_ind][0];
15 double xmid = triangle->processed_vertexes[mid_y_ind][0];
16 double zmax = triangle->processed_vertexes[max_y_ind][2];
17 double zmin = triangle->processed_vertexes[min_y_ind][2];
18 double zmid = triangle->processed_vertexes[mid_y_ind][2];
19
20 if (y < triangle->processed_vertexes[min_y_ind][1] || y >
    triangle->processed_vertexes[max_y_ind][1]) {
21     return;
22 }
23
24 if (max_x_ind == min_x_ind || max_y_ind == min_y_ind) {
25     return;
26 }
27
28 if ((y == triangle->processed_vertexes[max_y_ind][1] || y ==
    triangle->processed_vertexes[min_y_ind][1]) &&
    count_value_with_axis(triangle , 1, y) == 1) {
29     int y_ind = index_with_axis(triangle , 1, y);
30     int x = ceil(triangle->processed_vertexes[y_ind][0]);
31     int z = ceil(triangle->processed_vertexes[y_ind][2]);
32
33     if (x >= 0 && x < screen->width) {
34         if (depth_arr[x] > z) {
35             depth_arr[x] = z;
36             color_pixel(screen , &(triangle->processed_color),
37                 x, y);
38         }
39     }
40     return;
41 }

```

```

42
43     double start_x;
44     double finish_x;
45     double start_z;
46     double finish_z;
47
48     if (y > ymid) {
49         double d_ya = ymax - ymin;
50         double d_xa = xmax - xmin;
51         double d_za = zmax - zmin;
52         start_x = xmin + d_xa * (y - ymin) / d_ya;
53         start_z = zmin + d_za * (y - ymin) / d_ya;
54
55         double d_yb = ymax - ymid;
56         double d_xb = xmax - xmid;
57         double d_zb = zmax - zmid;
58         finish_x = xmid + d_xb * (y - ymid) / d_yb;
59         finish_z = zmid + d_zb * (y - ymid) / d_yb;
60
61     } else if (y < ymid) {
62
63         double d_ya = ymax - ymin;
64         double d_xa = xmax - xmin;
65         double d_za = zmax - zmin;
66         start_x = xmin + d_xa * (y - ymin) / d_ya;
67         start_z = zmin + d_za * (y - ymin) / d_ya;
68
69         double d_yb = ymid - ymin;
70         double d_xb = xmid - xmin;
71         double d_zb = zmid - zmin;
72         finish_x = xmin + d_xb * (y - ymin) / d_yb;
73         finish_z = zmin + d_zb * (y - ymin) / d_yb;
74
75     } else {
76         double d_ya = ymax - ymin;
77         double d_xa = xmax - xmin;
78         double d_za = zmax - zmin;
79         start_x = xmin + d_xa * (y - ymin) / d_ya;
80         start_z = zmin + d_za * (y - ymin) / d_ya;
81
82         finish_x = xmid;

```

```

83         finish_z = zmid;
84     }
85
86     if (start_x > finish_x) {
87         double tmp = start_x;
88         start_x = finish_x;
89         finish_x = tmp;
90         tmp = start_z;
91         start_z = finish_z;
92         finish_z = tmp;
93     }
94
95     if (fabs(finish_x - start_x) < 1e-2) {
96         return;
97     }
98
99     double dz = (finish_z - start_z) / (finish_x - start_x);
100
101     for (int x = ceil(start_x - 1); x < ceil(finish_x + 1); x++) {
102         if (x >= 0 && x < screen->width) {
103             double z = start_z + dz * (x - start_x);
104
105             if (depth_arr[x] > z) {
106                 depth_arr[x] = z;
107                 color_pixel(screen, &(triangle->processed_color),
108                             x, y);
109             }
110         }
111     }
112
113 void complete_process_level(screen_t *screen,
114                             std::vector<triangle_t*> triangles, int y) {
115
116     if (y < screen->height && y >= 0) {
117         auto *depth_array = (double*) malloc(sizeof(double) *
118                                             screen->width);
119
120         for (int i = 0; i < screen->width; i++) {
121             depth_array[i] = 1e33;
122         }

```

```

121
122     for (int i = 0; i < triangles.size(); i++) {
123         process_level(triangles[i], screen, y, depth_array);
124     }
125
126     for (int x = 0; x < screen->width; x++) {
127         if (not screen->change[x][y] && depth_array[x] ==
128             INT_MAX) {
129             color_pixel(screen, &(screen->default_color), x,
130                 y);
131         }
132     }
133     free(depth_array);
134 }
135
136 void z_buffer_render(screen_t *screen, std::vector<triangle_t*>
137     triangles)
138 {
139     for (int y = screen->height - 1; y >= 0; y--){
140         complete_process_level(screen, triangles, y);
141     }
142 }

```

3.4 Функциональные тесты

Тестирование выполнено по методологии белого ящика. Были введены параметры фракталов (цвет, аксиома, набор правил и т.д.). Результат, преобразованный в изображение, приведен на рисунке 3.1.

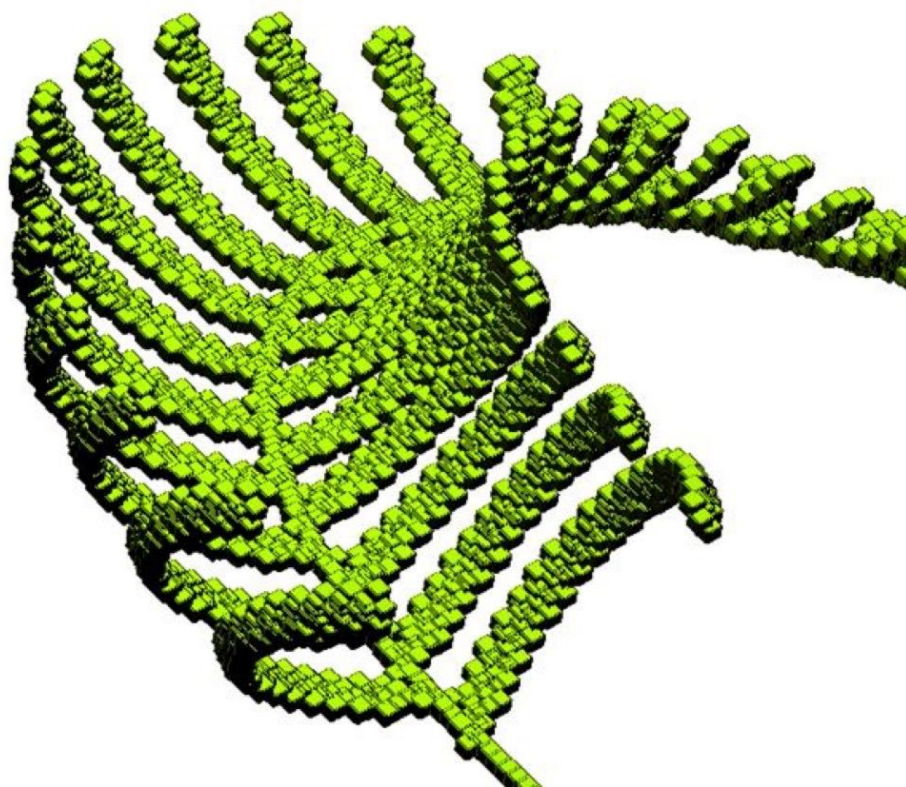


Рисунок 3.1 – Результат работы алгоритма удаления невидимых граней

Вывод

Был реализован алгоритм удаления невидимых граней, использующий Z-буфер. На его основе были реализованы: последовательная программа и многопоточная программа, которые строят изображение без невидимых граней. Программы были протестированы.

4 Исследовательская часть

В данном разделе были приведены примеры работы программ, постановка эксперимента и сравнительный анализ алгоритмов на основе полученных данных.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

- Операционная система macOS Monterey 12.5.1
- Память 16 Гб.
- Процессор 2,3 ГГц 4-ядерный процессор Intel Core i5.

Во время тестирования устройство было подключено к сети электропитания, нагружено приложениями окружения и самой системой тестирования.

4.2 Время выполнения алгоритмов

Для замера процессорного времени использовалась функция `std::chrono::system_clock::now(...)` из библиотеки *chrono* [4] на C++. Функция возвращает процессорное время типа `float` в секундах.

Контрольная точка возвращаемого значения не определена, поэтому допустима только разница между результатами последовательных вызовов.

Замеры времени для каждой длины входного массива полигонов проводились 1000 раз. В качестве результата взято среднее время работы алгоритма на данной длине. При каждом запуске алгоритма, на вход подавались случайно сгенерированные массивы полигонов.

4.2.1 Время выполнения однопоточной реализации

Результаты замеров приведены в таблице 4.1.

Таблица 4.1 – Результаты замеров времени однопоточной и последовательной реализаций в микросекундах (N - количество полигонов)

N	Однопоточное выполнение	Последовательное выполнение
10	30	28
20	35	31
30	40	36
50	43	39
100	57	49
200	75	69

4.2.2 Время выполнения многопоточной реализации

Результаты замеров приведены в таблице 4.2.

Таблица 4.2 – Результаты замеров времени выполнения программы, реализующей многопоточный алгоритм удаления невидимых граней, использующий Z-буфер, для разного количества потоков в микросекундах.

Количество треугольников	Количество потоков					
	1	4	6	12	24	48
10	30	33	25	23	21	27
100	57	75	76	73	53	0,78
1000	245	143	118	103	99	128
10000	2098	845	688	603	561	642
100000	19935	5606	5231	4689	4487	4992

При 6 потоках достигается пик, при котором все логические ядра процессора одновременно выполняют параллельные ветки алгоритма. Далее при увеличении числа потоков производительность падает. Это объясняется тем, что создается очередь потоков, которая замедляет работу программы.

Вывод

В данном разделе было произведено сравнение времени выполнения реализации алгоритма удаления невидимых ребер, при последовательной реализации и многопоточной. Результат показал, что выгоднее всего по времени использовать столько потоков, сколько у процессора логических ядер.

Заключение

В результате выполнения лабораторной работы была достигнута поставленная цель: были изучены основы организации параллельных вычислений на базе алгоритма Z-буфера.

В ходе выполнения лабораторной работы были выполнены следующие задачи:

- 1) изучены основы параллельных вычислений;
- 2) изучен алгоритм Z-буфера;
- 3) разработан параллельный алгоритм Z-буфера;
- 4) реализованы последовательная и параллельная Z-буфера;
- 5) проведен сравнительный анализ времени работы реализаций.

Список использованных источников

- [1] Параллельные вычисления [Электронный ресурс]. Режим доступа: https://ru.bmstu.wiki/P§PөCГPөPьPьPҳPьCÆP,,CKPҳ_PўCKCЉPқCFPьPҳP,,PқCS (дата обращения: 05.10.2021).
- [2] Что такое многопоточность [Электронный ресурс]. Режим доступа: <https://ru.coursera.org/lecture/ios-multithreading/chto-takoie-mnoghopotochnost-4MMgN> (дата обращения: 23.10.2021).
- [3] Рэнди Дэвис Стефан. С++ для чайников. Для чайников. Вильямс, 2018. с. 400.
- [4] Date and time utilities [Электронный ресурс]. Режим доступа: <https://en.cppreference.com/w/cpp/chrono> (дата обращения: 23.10.2021).