Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
«Санкт-Петербургский политехнический университет Петра Великого»
Институт компьютерных наук и кибербезопасности
Высшая школа кибербезопасности

К У Р С О В А Я Р А Б О Т А

Оптимизация программы для построения графиков функций

по дисциплине «Языки программирования»

Выполнил
студент гр. 435131001/30002 А.В, Костенников

Руководитель
Старший преподаватель Е.В. Малышев

«___» _____ 2025 г.

Санкт-Петербург

2025

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Целью данной курсовой работы является оптимизация уже разработанной на предыдущем курсе программы для построения графиков математических функций в 3D, так как на данный момент она работает не оптимально. Для достижения этой цели необходимо решить следующие задачи:

1. Добавить метрику для отслеживания скорости работы в виде времени выполнения расчетов координат точек графика.
2. Оценить алгоритмичесскую сложность немодифицированной программы, сделать замеры времени выполнения расчтов для типовых графов.
3. Внести алгоритмическую оптимизацию.
4. Провести повторные замеры производительности программы.
5. Внести не менее двух машинно-независимых оптимизаций.
6. Провести повторные замеры производительности программы.
7. Оценить алгоритмическую сложность оптимизированной программы.
8. Сопоставить полученные результаты профилироания, сделать выводы.

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

1.1. Метрика оценки неоптимизированного алгоритма.

В качестве метрики оценки алгоритма в нашем случае оптимален замер времени выполнения вычислений координат точек, на основе которых строится график, так как эта функция является самой «сложной».

После добавления метрики, были проведены замеры для выбранных графиков (рис. 1-3).

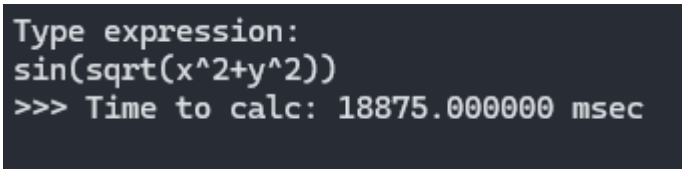

Рис. 1. Замер для «sin(sqrt(x^2+y^2))».

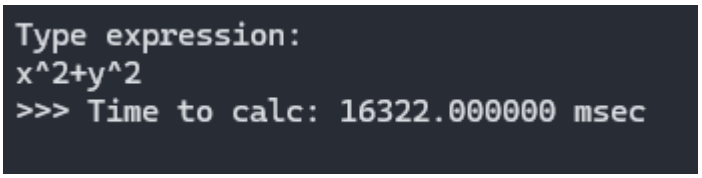

Рис. 2. Замер для «x^2+y^2».

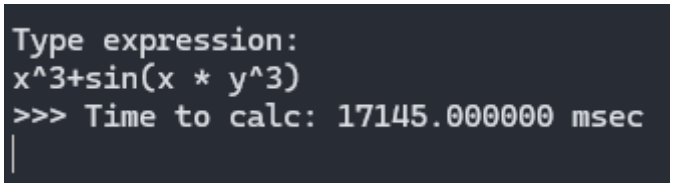

Рис. 3. Замер для «x^3+sin(x * y^3)».

Заметим, что на начальном этапе в среднем вычисление координат точек графика занимает 16 секунд.

1.2. Алгоритмическая сложность немоптимизированного алгоритма.

Если рассмотреть код функции расчета координат точек (рис. 4), можно отметить, что функция вычисления выражения вызывается постоянно и не зависит от условий. Так как не планируется ее изменение, обозначим сложность выполнения этой функции за “M”, чтобы более наглядно отобразить изменения.

```
void computePoints(char* expression) {

    int index = 0;

    for (double x = diapStart; x <= diapEnd; x += diapStep) {
        for (double y = diapStart; y <= diapEnd; y += diapStep) {
            double z = calculate_expression(expression, x, y);
            if (index == 0) {
                points = (struct Point*)malloc(sizeof(struct Point));
            }
            else {
                points = (struct Point*)realloc(points, (index + 1) * sizeof(struct Point));
            }
            struct Point* point = (struct Point*)malloc(sizeof(struct Point));
            point->x = x;
            point->y = y;
            point->z = z;
            points[index++] = *point;
        }
    }
}
```

Рис. 4. Код функции «computePoints» до оптимизации

Так, для каждой точки:

1. Код проходит диапазон значений “x” и “y” с шагом “diapStep”.
2. Для каждой пары (x, y) вызывается calculate_expression.
3. Оставшиеся операции имеют сложность O(1).

Получаем алгоритмическую сложность исходного алгоритма:

$$O(N^2 \cdot M)$$

1.3. Принцип алгоритмической оптимизации.

Так как программа подразумевает вычисление координат точек, лучшей алгоритмической оптимизацией является интерполяция.

Интерполяция – вычислительно-математические техники восстановления функций по имеющимуся дискретному набору ее известных значений, что подходит для нашего случая.

Остается выбрать ондин из существующих методов интерполяции для функции от двух переменных:

1. Билинейная интерполяция.
    - Преимущества: Простота реализации и высокая скорость.
    - Недостатки: Точность может быть недостаточной для функций с высокой кривизной.
2. Бикубическая интерполяция.
    - Преимущества: Высока точность, гладкий результат.
    - Недостатки: Большие вычислительные затраты, сложная реализация.
3. Интерполяция методом ближайшего соседа.
    - Преимущества: Очень большая скорость, минимальные вычислительные затраты.
    - Недостатки: Низкая точность, ступенчатые артефакты.
4. Радикальные базисные функции.
    - Преимущества: Гибкость, возможность работы с неравномерно распределенными данными.
    - Недостатки: Высокие вычислительные затраты, особенно для большого объема данных.

Выбор метода зависит от поставленной задачи, в нашем конкретном случае билинейная интерполяция является оптимальной.

Функция билинейной интерполяции имеет вид:

$$F(x, y) = b_1 + b_2 x + b_3 y + b_4 xy$$

Эта функция интерполирует значения исходной функции двух переменных в произвольном прямоугольнике по четырём её значениям в вершинах прямоугольника.

1.4. Оценка алгоритма после введения машинно-независимых оптимизаций.

После добавления машинно-независимых оптимизаций были проведены замеры для выбранных графиков (рис. 4-6).

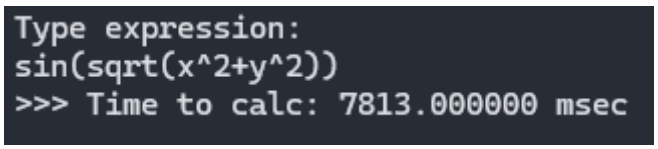

Рис. 4. Замер для «sin(sqrt(x^2+y^2))».

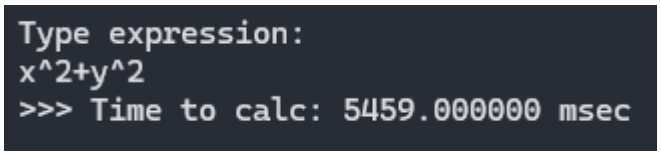

Рис. 5. Замер для «sin(sqrt(x^2+y^2))».

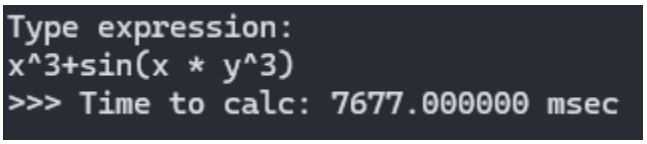

Рис. 6. Замер для «sin(sqrt(x^2+y^2))».

Заметим, что скорость вычисления координат точек после МНЗ оптимизаций в среднем возросло на 50-60%.

1.5. Оценка алгоритма после введения алгоритмической оптимизации.

После добавления алгоритмической оптимизации были проведены замеры для выбранных графиков (рис. 7-9).

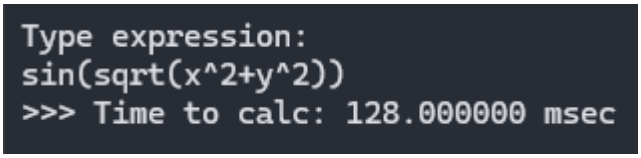

Рис. 7. Замер для «sin(sqrt(x^2+y^2))».

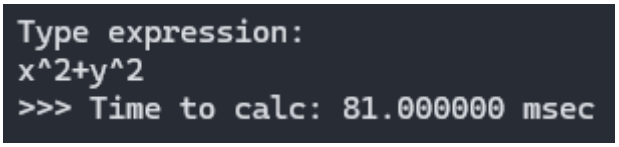

Рис. 8. Замер для «sin(sqrt(x^2+y^2))».

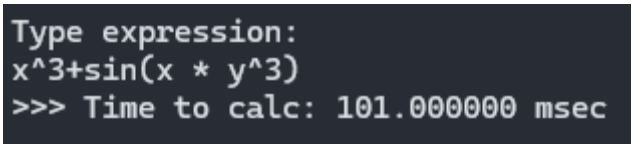

Рис. 9. Замер для «sin(sqrt(x^2+y^2))».

Заметим, что скорость вычисления координат точек после алгоритмической оптимизации в среднем возросло на 80-90% от значений после машинно-независимой оптимизации.

Так же, учитывая, особенности выбранной алгоритмической оптимизации, была проведена оценка точности результатов вычислений.

Если сравнить исходные графики и графики, полученные с помощью оптимизированной программы, можно сделать вывод, что для большинства функций точность не изменилась (рис. 10, 11), но в случае функций с сильной кривизной (рис. 12), появляются небольшие отклонения, которые можно исключить, уменьшив шаг для «грубой» сетки вычислений.

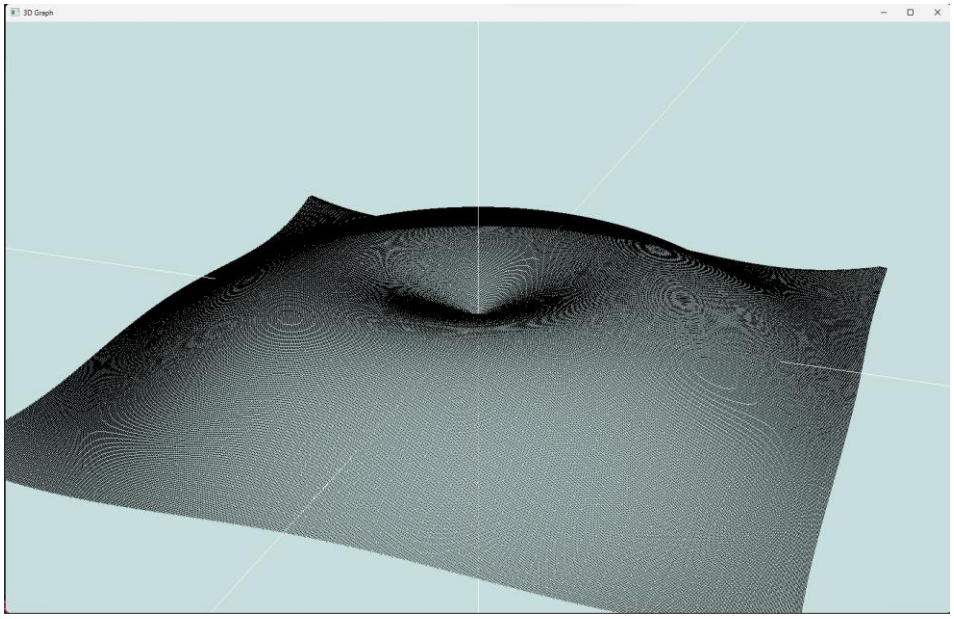

Рис. 10. График для «sin(sqrt(x^2+y^2))».

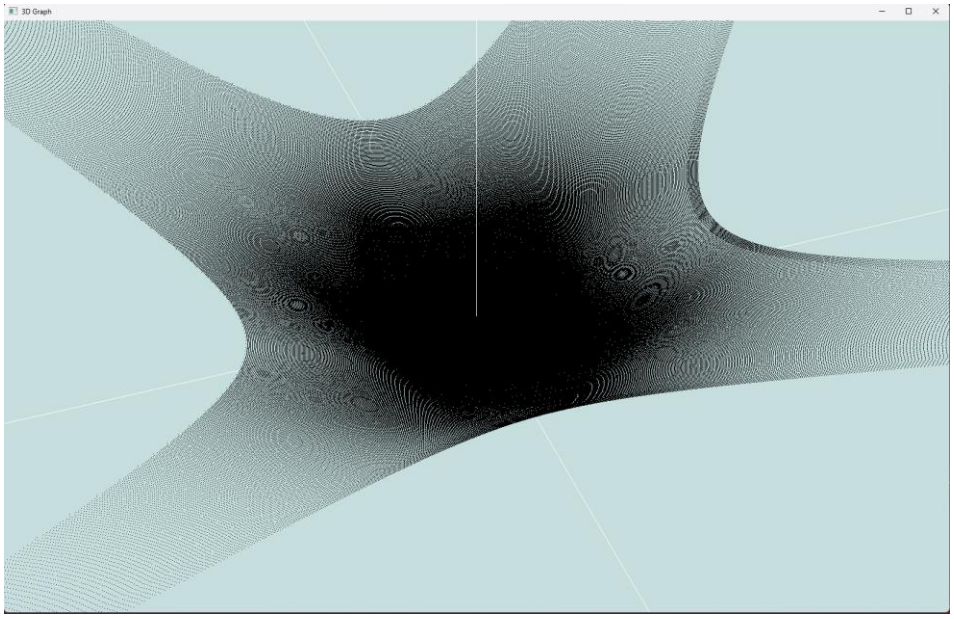

Рис. 11. График для «sin(sqrt(x^2+y^2))».

Рис. 12. График для «sin(sqrt(x^2+y^2))».

1.6. Алгоритмическая сложность оптимизированного алгоритма.

Если рассмотреть обновленный код функции расчета координат точек (рис. 13), можно оценить новую сложность алгоритма.

```
void computePoints(char* expression) {
    int index = 0;

    clock_t start = clock();

    int DimSize = (int)((diapEnd - diapStart) / diapStep) + 1;

    points = (struct Point*)calloc(DimSize * DimSize, sizeof(struct Point));

    for (double x = diapStart; x <= diapEnd; x += diapStep * 10) {
        for (double y = diapStart; y <= diapEnd; y += diapStep * 10) {
            double z = calculate_expression(expression, x, y);

            int cX = (int)((x - diapStart) / diapStep);
            int cY = (int)((y - diapStart) / diapStep);
            int cIndex = cX * DimSize + cY;

            points[cIndex].x = x;
            points[cIndex].y = y;
            points[cIndex].z = z;
        }
    }

    for (double x = diapStart; x <= diapEnd; x += diapStep) {
        for (double y = diapStart; y <= diapEnd; y += diapStep) {

            int fX = (int)((x - diapStart) / diapStep);
            int fY = (int)((y - diapStart) / diapStep);
            int fIndex = fX * DimSize + fY;

            if (fX % 10 != 0 || fY % 10 != 0) {

                int coarseX1 = (fX / 10) * 10;
                int coarseX2 = coarseX1 + 10;
                int coarseY1 = (fY / 10) * 10;
                int coarseY2 = coarseY1 + 10;

                int q11 = coarseX1 * DimSize + coarseY1;
                int q12 = coarseX1 * DimSize + coarseY2;
                int q21 = coarseX2 * DimSize + coarseY1;
                int q22 = coarseX2 * DimSize + coarseY2;

                double z = BilinearInterpolation(x, y, points[q11].x, points[q21].x, points[q11].y, points[q12].y, points[q11].z, points[q12].z, points[q21].z, points[q22].z);

                points[fIndex].x = x;
                points[fIndex].y = y;
                points[fIndex].z = z;
            }
        }
    }
```

Рис. 13. Код функции «computePoints» после оптимизации

Так, теперь функция имеет вид:

1. В первом цикле код проходит диапазон значений “x” и “y” с шагом “diapStep/10”.
2. Для каждой пары (x, y) вызывается calculate_expression.
3. Оставшиеся операции первого цикла имеют сложность O(1).
4. Во втором цикле код проходит диапазон значений “x” и “y” с шагом “diapStep”.

5. Для каждой пары (x, y) вызывает BilinearInterpolation, имеющую константную сложность.
6. Оставшиеся операции ьак же имеют сложность O(1).

Получаем алгоритмическую сложность оптимизированного алгоритма:

$$O(N^2) + O\left(N^2 \cdot \frac{M}{100}\right) = O\left(N^2 + N^2 \cdot \frac{M}{100}\right)$$

Мы уже можем заметить уменьшение сложности алгоритма, но если дополнительно учесть, что “M/100” эквивалентно константе, получим:

$$O\left(N^2 + N^2 \cdot \frac{M}{100}\right) = O(N^2 \cdot 2) = O(N^2)$$

1.7. Анализ производительности на этапах оптимизации.

На основе полученного времени выполнения расчетов координат точек каждого из 3 графиков на каждом этапе курсовой работы былпостроен график, демонстрирующий результаты реализованных оптимизаций (рис. 14).

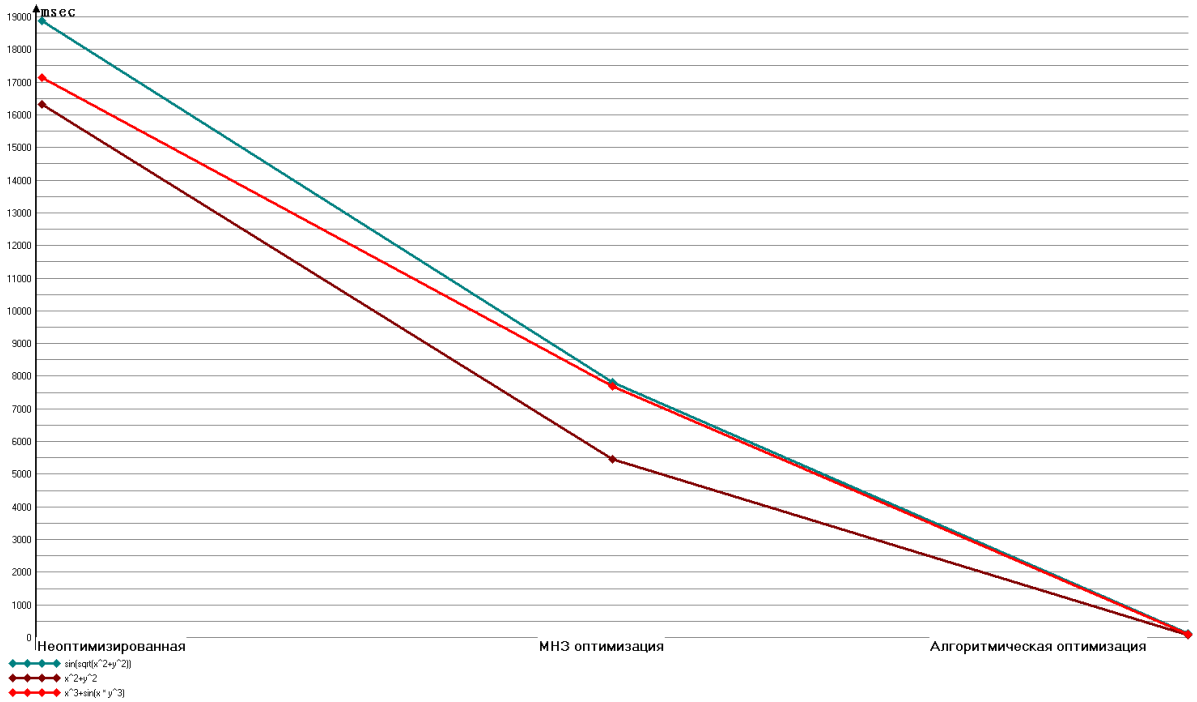

Рис. 14. График скорости вычислений на разных уровнях оптимизации.

ПРАКТИЧЕСКАЯ ЧАСТЬ

2.1. Введение метрик оценки алгоритма.

В функцию calculate_expression была добавлена метрика оценки, в частности вывод времени выполнения функции с помощью библиотеки time.h (рис. 15, 16).

```
void computePoints(char* expression) {
    int index = 0;

    clock_t start = clock();
```

Рис. 15. Введение метрик оценки в функцию.

```
    clock_t end = clock();

    printf(">>> Time to calc: %lf msec\n", (double)(end - start) / (CLOCKS_PER_SEC / 1000));
}
```

Рис. 16. Введение метрик оценки в функцию.

2.2. Машинно-независимые оптимизации.

В коде программы были реализованы следующие машинно-незавсимые оптимизации:

- Вынос из цикла расчета количесвта точек с учетом шага сетки и выделение под них памяти в функции вычисления координат точек (рисю 17) и функции обновления OpenGl (рис. 18).

```
clock_t start = clock();

int DimSize = (int)((diapEnd - diapStart) / diapStep) + 1;

points = (struct Point*)calloc(DimSize * DimSize, sizeof(struct Point));

for (double x = diapStart; x <= diapEnd; x += diapStep * 10) {
```

Рис. 17. Вынос из цикла в функции computePoints.

```
int DimPoints = (diapEnd - diapStart) / diapStep;

for (int i = 1; i < DimPoints * DimPoints; ++i) {
    glVertex3f(points[i].x, points[i].y, points[i].z);
}
```

Рис. 18. Вынос из цикла в функции DrawUpdated.

- В функции Dijkstra был вынесен в константу множественный вызов функции get (рис. 19).

```
void Dijkstra(char* operation, struct Dict* dict, struct Queue* queue, struct Stack** stack) {

    int value = get(dict, operation);

    if (value == -1) {
        char* temp;
        do {
            temp = pop(stack);
            if (get(dict, temp) != 1) {
                enqueue(queue, temp);
            }
        } while (get(dict, temp) != 1);
    }
    else if (value == 1) {
        push(stack, operation);
    }
    else {
        while (*stack != NULL && OperationWeight(value) <= OperationWeight(get(dict, (*stack)->data))) {
            enqueue(queue, pop(stack));
        }
        push(stack, operation);
    }
}
```

Рис. 19. Изменения в функции Dijkstra.

- В функцию рботы со словарем get было добавлено условие совпадения первых букв, что в условиях программы позволяет избежать излишнего использования strcmp (рис. 20).

```
int get(struct Dict* dict, const char* key) {
    struct DictNode* curr = dict->head;
    while (curr != NULL) {
        if (curr->key[0] == key[0] && strcmp(curr->key, key) == 0) {
            return curr->value;
        }
        curr = curr->next;
    }
    return 0;
}
```

Рис. 20. Изменения в функции get.

- В функцию main из функции calculate_expression было перенесено заполнение словаря операций, за счет чего эта процедура была вынесена из цикла в функции computePoints.

2.3. Алгоритмическая оптимизация

Была выбрана билинейная интерполяция в качестве алгоритмической оптимизации. В ее рамках:

1. Была реализована функция BilinearInterpolation (рис. 21), вычисляющая значение координат точки «точной» сетки и основываюшаяся на формуле (рис. 22).

```
double BilinearInterpolation(double x, double y, double x1, double x2, d
    double s1 = (q11 * (x2 - x) * (y2 - y)) / ((x2 - x1) * (y2 - y1));
    double s2 = (q21 * (x - x1) * (y2 - y)) / ((x2 - x1) * (y2 - y1));
    double s3 = (q12 * (x2 - x) * (y - y1)) / ((x2 - x1) * (y2 - y1));
    double s4 = (q22 * (x - x1) * (y - y1)) / ((x2 - x1) * (y2 - y1));
    return s1 + s2 + s3 + s4;
}
```

Рис. 21. Функция BilinearInterpolation.

$$f(x,y) \approx F(x,y) = \frac{f(Q_{11})}{(x_2-x_1)(y_2-y_1)}(x_2-x)(y_2-y) + \\ + \frac{f(Q_{21})}{(x_2-x_1)(y_2-y_1)}(x-x_1)(y_2-y) + \\ + \frac{f(Q_{12})}{(x_2-x_1)(y_2-y_1)}(x_2-x)(y-y_1) + \\ + \frac{f(Q_{22})}{(x_2-x_1)(y_2-y_1)}(x-x_1)(y-y_1).$$

Рис. 22. Функция билинейной интерполяции.

2. Был реализован первый цикл формирования «грубой» сетки, т.е. вычисления каждой десятой координаты с помощью функции calculate_expression (рис. 23)

```
for (double x = diapStart; x <= diapEnd; x += diapStep * 10) {
    for (double y = diapStart; y <= diapEnd; y += diapStep * 10) {
        double z = calculate_expression(expression, x, y);

        int cX = (int)((x - diapStart) / diapStep);
        int cY = (int)((y - diapStart) / diapStep);
        int cIndex = cX * DimSize + cY;

        points[cIndex].x = x;
        points[cIndex].y = y;
        points[cIndex].z = z;
    }
}
```

Рис. 23. Формирование «грубой» сетки.

3. Был реализован второй цикл формирования «точной» сетки, т.е. интерполяции оставшихся точек на основе ближайших точно вычисленных (рис. 24)

```
for (double x = diapStart; x <= diapEnd; x += diapStep) {
    for (double y = diapStart; y <= diapEnd; y += diapStep) {

        int fX = (int)((x - diapStart) / diapStep);
        int fY = (int)((y - diapStart) / diapStep);
        int fIndex = fX * DimSize + fY;

        if (fX % 10 != 0 || fY % 10 != 0) {

            int coarseX1 = (fX / 10) * 10;
            int coarseX2 = coarseX1 + 10;
            int coarseY1 = (fY / 10) * 10;
            int coarseY2 = coarseY1 + 10;

            int q11 = coarseX1 * DimSize + coarseY1;
            int q12 = coarseX1 * DimSize + coarseY2;
            int q21 = coarseX2 * DimSize + coarseY1;
            int q22 = coarseX2 * DimSize + coarseY2;

            double z = BilinearInterpolation(x, y, points[q11].x, points[q21].x,

            points[fIndex].x = x;
            points[fIndex].y = y;
            points[fIndex].z = z;
        }
    }
}
```

Рис. 24. Формирование «точной» сетки.

ЗАКЛЮЧЕНИЕ

В ходе выполнения курсовой работы были внесены изменения в программу для построения и визуализации графиков математических функций с использованием языка программирования C и библиотеки OpenGL, оптимизирующие скорость ее выполнения.

Для достижения поставленной цели были решены следующие задачи:

Изучены теоретические основы типов оптимизаций программ, возможные способы машинно-независимых оптимизаций, а так же рассмотрены различные типы интерполяций.

Преобразован алгоритм вычисления координат точек графика путем внесения алгоритмической оптимизации в виде билинейной интерполяции.

Реализованы такие машинно-независимые оптимизации, как замена переменной цикла, вынос инварианта, правильный порядок логических операций в цикле.

Проведено тестирование программы для проверки корректности преобразования и вычислений, а также профилирование программы на каждом этапе оптимизации и анализ полученных значений.

В заключение можно отметить, что поставленные цели и задачи курсовой работы были успешно достигнуты, что подтверждается оптимизацией разработанного приложения и его успешным тестированием.

ПРИЛОЖЕНИЕ А

Листинг 1 - исходный код программы на языке C

```
#define _CRT_SECURE_NO_WARNINGS
#include <glew.h>
#include <glut.h>
#include <glfw3.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <time.h>

#define MAX_EXPRESSION_LENGTH 100

char userExpression[MAX_EXPRESSION_LENGTH];
float rotateX = 0.0;
float rotateY = 0.0;
float rotateZ = 0.0;
float translateZ = -30.0;
float translateX = 0.0;
float translateY = 0.0;
double diapStart = -4.0;
double diapEnd = 4.0;
double diapStep = 0.01;

struct Point {
    double x;
    double y;
    double z;
};

struct Point* points = NULL;

int IsSimpleOperation(char ch) {
    switch (ch) {
    case '+':
    case '-':
    case '*':
    case '/':
    case '^':
    case '!':
    case '(':
    case ')':
        return 1;
    default:
        return 0;
    }
}

int OperationWeight(int operation) {
    switch (operation) {
    case 1:
        return 0;
    case 2:
    case 3:
        return 1;
    case 4:
    case 5:
        return 2;
    case 6:
    case 7:
    case 8:
    case 9:
    case 10:
    case 11:
    case 19:
```

```
    case 110:
    case 111:
        return 3;
    default:
        return -1;
    }
}

struct QueueNode {
    char* data;
    struct QueueNode* next;
};

struct Queue {
    struct QueueNode* front;
    struct QueueNode* back;
};

struct Queue* createQueue() {
    struct Queue* queue = (struct Queue*)malloc(sizeof(struct Queue));
    queue->front = NULL;
    queue->back = NULL;
    return queue;
}

void enqueue(struct Queue* queue, char* data) {
    struct QueueNode* node = (struct QueueNode*)malloc(sizeof(struct QueueNode));
    node->data = (char*)malloc((strlen(data) + 1) * sizeof(char));
    strcpy(node->data, data);
    node->next = NULL;

    if (queue->back == NULL) {
        queue->front = node;
        queue->back = node;
    }
    else {
        queue->back->next = node;
        queue->back = node;
    }
}

char* dequeue(struct Queue* queue) {
    if (queue->front == NULL) {
        return NULL;
    }
    struct QueueNode* temp = queue->front;
    char* data = temp->data;
    queue->front = queue->front->next;

    if (queue->front == NULL) {
        queue->back = NULL;
    }

    free(temp);
    return data;
}

struct Stack {
    char* data;
    struct Stack* next;
};

void push(struct Stack** top, char* data) {
    struct Stack* stack = (struct Stack*)malloc(sizeof(struct Stack));
    stack->data = (char*)malloc((strlen(data) + 1) * sizeof(char));
    strcpy(stack->data, data);
    stack->next = *top;
    *top = stack;
}
```

```
char* pop(struct Stack** top) {
    if (*top == NULL) {
        return NULL;
    }
    struct Stack* temp = *top;
    *top = (*top)->next;
    char* data = temp->data;
    free(temp);
    return data;
}

struct ResStack {
    double data;
    struct ResStack* next;
};

void pushRes(struct ResStack** top, double data) {
    struct ResStack* stack = (struct ResStack*)malloc(sizeof(struct ResStack));
    stack->data = data;
    stack->next = *top;
    *top = stack;
}

double popRes(struct ResStack** top) {
    if (*top == NULL) {
        return 0;
    }
    struct ResStack* temp = *top;
    *top = (*top)->next;
    double data = temp->data;
    free(temp);
    return data;
}

struct DictNode {
    char* key;
    int value;
    struct DictNode* next;
};

struct Dict {
    struct DictNode* head;
};

struct Dict* createDict() {
    struct Dict* dict = (struct Dict*)malloc(sizeof(struct Dict));
    dict->head = NULL;
    return dict;
}

void add(struct Dict* dict, const char* key, int value) {
    struct DictNode* node = (struct DictNode*)malloc(sizeof(struct DictNode));
    node->key = (char*)malloc((strlen(key) + 1) * sizeof(char));
    strcpy(node->key, key);
    node->value = value;
    node->next = dict->head;
    dict->head = node;
}

int get(struct Dict* dict, const char* key) {
    struct DictNode* curr = dict->head;
    while (curr != NULL) {
        if (curr->key[0] == key[0] && strcmp(curr->key, key) == 0) {
            return curr->value;
        }
        curr = curr->next;
    }
    return 0;
```

```
}

void createDictOperations(struct Dict* dict) {
    add(dict, "(", 1);
    add(dict, ")", -1);
    add(dict, "+", 2);
    add(dict, "-", 3);
    add(dict, "*", 4);
    add(dict, "/", 5);
    add(dict, "^", 6);
    add(dict, "sqrt", 7);
    add(dict, "!", 8);
    add(dict, "sin", 9);
    add(dict, "cos", 10);
    add(dict, "tan", 11);
    add(dict, "arcsin", 19);
    add(dict, "arccos", 110);
    add(dict, "arctan", 111);
}

struct Dict* DictOperations;

void Dijkstra(char* operation, struct Dict* dict, struct Queue* queue, struct Stack** stack) {

    int value = get(dict, operation);

    if (value == -1) {
        char* temp;
        do {
            temp = pop(stack);
            if (get(dict, temp) != 1) {
                enqueue(queue, temp);
            }
        } while (get(dict, temp) != 1);
    }
    else if (value == 1) {
        push(stack, operation);
    }
    else {
        while (*stack != NULL && OperationWeight(value) <= OperationWeight(get(dict, (*stack)->data))) {
            enqueue(queue, pop(stack));
        }
        push(stack, operation);
    }
}

double OperationResult(int operation, double temp1, double temp2) {
    switch (operation) {
    case 2:
        return temp1 + temp2;
    case 3:
        return temp1 - temp2;
    case 4:
        return temp1 * temp2;
    case 5:
        return temp1 / temp2;
    case 6:
        return pow(temp1, temp2);
    case 7:
        return sqrt(temp2);
    case 8:
        for (int i = (int)temp2 - 1; i > 1; i--) {
            temp2 *= i;
        }
        return temp2;
    case 9:
        return sin(temp2);
    case 10:
        return cos(temp2);
```

```
    case 11:
        return tan(temp2);
    case 19:
        return asin(temp2);
    case 110:
        return acos(temp2);
    case 111:
        return atan(temp2);
    default:
        return 0;
    }
}

double calculate_expression(char* expression, double x, double y) {
    struct Dict* DictOperations = createDict();
    createDictOperations(DictOperations);

    struct Queue* queue = createQueue();
    struct Stack* stack = NULL;
    struct ResStack* resstack = NULL;

    char* buffer = NULL;
    int bufferSize = 10;
    int bufferIndex = 0;
    char ch;

    buffer = (char*)malloc(bufferSize * sizeof(char));

    char* expr_ptr = expression;

    while ((ch = *expr_ptr++) != '\0') {
        if (bufferIndex == bufferSize) {
            bufferSize++;
            buffer = (char*)realloc(buffer, bufferSize * sizeof(char));
        }
        if (ch == ' ') {
            continue;
        }
        if (bufferIndex > 0) {
            //11
            if (buffer[bufferIndex - 1] > 47 && buffer[bufferIndex - 1] < 58 && ch > 47 && ch < 58) {
                buffer[bufferIndex++] = ch;
                continue;
            }
            //aa
            else if ((buffer[bufferIndex - 1] < 47 || buffer[bufferIndex - 1] > 58) && (ch < 47 || ch >
58)) {
                if (IsSimpleOperation(ch)) {
                    buffer[bufferIndex] = '\0';
                    bufferIndex = 0;
                    if (get(DictOperations, buffer) != 0) {
                        Dijkstra(buffer, DictOperations, queue, &stack);
                    }
                    else {
                        enqueue(queue, buffer);
                    }
                    buffer[bufferIndex++] = ch;
                    buffer[bufferIndex] = '\0';
                    bufferIndex = 0;
                    Dijkstra(buffer, DictOperations, queue, &stack);
                }
                else {
                    buffer[bufferIndex++] = ch;
                    continue;
                }
            }
            //1a
            else if (buffer[bufferIndex - 1] > 47 && buffer[bufferIndex - 1] < 58 && (ch < 47 || ch > 58))
{
```

```
                buffer[bufferIndex] = '\0';
                bufferIndex = 0;
                enqueue(queue, buffer);
                buffer[bufferIndex++] = ch;
                if (IsSimpleOperation(ch)) {
                    buffer[bufferIndex] = '\0';
                    bufferIndex = 0;
                    Dijkstra(buffer, DictOperations, queue, &stack);
                }
                continue;
            }
            //a1
            else if ((buffer[bufferIndex - 1] < 47 || buffer[bufferIndex - 1] > 58) && ch > 47 && ch < 58)
{
                buffer[bufferIndex] = '\0';
                bufferIndex = 0;
                Dijkstra(buffer, DictOperations, queue, &stack);
                buffer[bufferIndex++] = ch;
            }
        }
        else {
            buffer[bufferIndex++] = ch;
        }
    }
    if (bufferIndex > 0) {
        buffer[bufferIndex] = '\0';
        bufferIndex = 0;
        if (IsSimpleOperation(buffer[0])) {
            if (get(DictOperations, buffer) != 0) {
                Dijkstra(buffer, DictOperations, queue, &stack);
            }
            else {
                enqueue(queue, buffer);
            }
        }
        else {
            enqueue(queue, buffer);
        }
    }

    while (stack != NULL) {
        enqueue(queue, pop(&stack));
    }

    while (queue->front != NULL) {
        char* temp = dequeue(queue);

        if (temp[0] > 47 && temp[0] < 58) {
            pushRes(&resstack, strtod(temp, NULL));
        }
        else if (get(DictOperations, temp) != 0) {
            if (get(DictOperations, temp) > 6) {
                double temp_ = popRes(&resstack);
                double res = OperationResult(get(DictOperations, temp), 0, temp_);
                pushRes(&resstack, res);
            }
            else {
                double temp2 = popRes(&resstack);
                double temp1 = popRes(&resstack);
                double res = OperationResult(get(DictOperations, temp), temp1, temp2);
                pushRes(&resstack, res);
            }
        }
        else {
            if (temp[0] == 'x') {
                pushRes(&resstack, x);
            }
            if (temp[0] == 'y') {
                pushRes(&resstack, y);
```

```
            }
        }
        free(temp);
    }

    free(buffer);
    free(DictOperations);
    free(queue);

    return popRes(&resstack);
}

double BilinearInterpolation(double x, double y, double x1, double x2, double y1, double y2, double q11,
double q12, double q21, double q22) {
    double s1 = (q11 * (x2 - x) * (y2 - y)) / ((x2 - x1) * (y2 - y1));
    double s2 = (q21 * (x - x1) * (y2 - y)) / ((x2 - x1) * (y2 - y1));
    double s3 = (q12 * (x2 - x) * (y - y1)) / ((x2 - x1) * (y2 - y1));
    double s4 = (q22 * (x - x1) * (y - y1)) / ((x2 - x1) * (y2 - y1));
    return s1 + s2 + s3 + s4;
}

void computePoints(char* expression) {
    int index = 0;

    clock_t start = clock();

    int DimSize = (int)((diapEnd - diapStart) / diapStep) + 1;

    points = (struct Point*)calloc(DimSize * DimSize, sizeof(struct Point));

    for (double x = diapStart; x <= diapEnd; x += diapStep * 10) {
        for (double y = diapStart; y <= diapEnd; y += diapStep * 10) {
            double z = calculate_expression(expression, x, y);

            int cX = (int)((x - diapStart) / diapStep);
            int cY = (int)((y - diapStart) / diapStep);
            int cIndex = cX * DimSize + cY;

            points[cIndex].x = x;
            points[cIndex].y = y;
            points[cIndex].z = z;
        }
    }

    for (double x = diapStart; x <= diapEnd; x += diapStep) {
        for (double y = diapStart; y <= diapEnd; y += diapStep) {

            int fX = (int)((x - diapStart) / diapStep);
            int fY = (int)((y - diapStart) / diapStep);
            int fIndex = fX * DimSize + fY;

            if (fX % 10 != 0 || fY % 10 != 0) {

                int coarseX1 = (fX / 10) * 10;
                int coarseX2 = coarseX1 + 10;
                int coarseY1 = (fY / 10) * 10;
                int coarseY2 = coarseY1 + 10;

                int q11 = coarseX1 * DimSize + coarseY1;
                int q12 = coarseX1 * DimSize + coarseY2;
                int q21 = coarseX2 * DimSize + coarseY1;
                int q22 = coarseX2 * DimSize + coarseY2;

                double z = BilinearInterpolation(x, y, points[q11].x, points[q21].x, points[q11].y,
points[q12].y, points[q11].z, points[q12].z, points[q21].z, points[q22].z);

                points[fIndex].x = x;
                points[fIndex].y = y;
                points[fIndex].z = z;
```

```
            }
        }
    }

    clock_t end = clock();

    printf(">>> Time to calc: %lf msec\n", (double)(end - start) / (CLOCKS_PER_SEC / 1000));
}

void DrawUpdated() {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glPushMatrix();

    glTranslatef(translateX, translateY, translateZ);
    glRotatef(rotateX, 1.0, 0.0, 0.0);
    glRotatef(rotateY, 0.0, 1.0, 0.0);
    glRotatef(rotateZ, 0.0, 0.0, 1.0);

    // ��� x
    glBegin(GL_LINES);
    glVertex3f(-100.0, 0.0, 0.0);
    glVertex3f(100.0, 0.0, 0.0);
    glEnd();

    // ��� y
    glBegin(GL_LINES);
    glVertex3f(0.0, -100.0, 0.0);
    glVertex3f(0.0, 100.0, 0.0);
    glEnd();

    // ��� z
    glBegin(GL_LINES);
    glVertex3f(0.0, 0.0, -100.0);
    glVertex3f(0.0, 0.0, 100.0);
    glEnd();

    glColor3f(0.0f, 0.0f, 0.0f);

    glBegin(GL_POINTS);

    int DimPoints = (diapEnd - diapStart) / diapStep;

    for (int i = 1; i < DimPoints * DimPoints; ++i) {
        glVertex3f(points[i].x, points[i].y, points[i].z);
    }

    glEnd();

    glColor3f(1.0f, 1.0f, 1.0f);

    glPopMatrix();

    glutSwapBuffers();
}

void keyboard(unsigned char key, int x, int y) {
    switch (key) {
    case 27:  // Escape
        exit(0);
        break;
    case 'w':
        translateZ -= 1.0;
        break;
    case 's':
        translateZ += 1.0;
        break;
    case 'a':
        rotateY -= 5.0;
```

```
        break;
    case 'd':
        rotateY += 5.0;
        break;
    case 'q':
        translateX -= 1.0;
        break;
    case 'e':
        translateX += 1.0;
        break;
    case 'y':
        rotateX += 5.0;
        break;
    case 'h':
        rotateX -= 5.0;
        break;
    case 'g':
        rotateZ += 5.0;
        break;
    case 'j':
        rotateZ -= 5.0;
        break;
    case 'z':
        translateY -= 0.1;
        break;
    case 'x':
        translateY += 0.1;
        break;
    }

    glutPostRedisplay();
}

void specialKeys(int key, int x, int y) {
    switch (key) {
    case GLUT_KEY_UP:
        translateZ += 1.0;
        break;
    case GLUT_KEY_DOWN:
        translateZ -= 1.0;
        break;
    }
    glutPostRedisplay();
}

void init() {
    glClearColor(0.7764705882352941f, 0.8745098039215686f, 0.8705882352941177f, 1.0f);
    glMatrixMode(GL_PROJECTION);
    gluPerspective(45.0, 1.0, 1.0, 200.0);
    glMatrixMode(GL_MODELVIEW);
    gluLookAt(0.0, 0.0, 10.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
    glEnable(GL_DEPTH_TEST);
    glutKeyboardFunc(keyboard);
}

int main(int argc, char** argv) {

    printf("Type expression:\n");
    fgets(userExpression, MAX_EXPRESSION_LENGTH, stdin);
    userExpression[strcspn(userExpression, "\n")] = 0;

    DictOperations = createDict();
    createDictOperations(DictOperations);

    computePoints(userExpression);

    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(1600, 1000);
```

```
    glutCreateWindow("3D Graph");

    glewInit();

    init();

    glutDisplayFunc(DrawUpdated);
    glutKeyboardFunc(keyboard);
    glutSpecialFunc(specialKeys);

    glutMainLoop();

    free(points);
    return 0;
}
```