Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования

ПЕРМСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

(ФГАОУ ВО ПНИПУ)

ЭЛЕКТРОТЕХНИЧЕСКИЙ ФАКУЛЬТЕТ

КАФЕДРА ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ И АВТОМАТИЗИРОВАННЫЕ СИСТЕМЫ

ОТЧЁТ

О ЛАБОРАТОРНОЙ РАБОТЕ

На тему: Финансовый калькулятор и задача коммивояжера

Выполнил:

Студент группы РИС-23-3б

Асташин В.А.

Проверил:

доц. кафедры ИТАС

Полякова О.А.

Пермь 2024

**Финансовый калькулятор**

**Постановка задачи для финансового калькулятора**

Создать калькулятор, который будет принимать на вход экономические данные и на их основе рассчитывать показатели.

**Анализ задачи и архитектура решения**

1) Инициализация графической среды

Приложение инициализирует графическую подсистему GLUT, создает главное окно с фиксированными размерами и устанавливает систему координат для корректного отображения элементов управления. Настраивается цветовая схема интерфейса с голубым фоном для обеспечения визуального комфорта пользователя.

2) Организация пользовательского интерфейса

Интерфейс строится на основе иерархической компоновки графических элементов, которые включают:

* Зоны ввода данных - текстовые поля для ввода финансовых показателей, включая выручку, себестоимость, коммерческие и управленческие расходы, прочие доходы и расходы, а также налог на прибыль;
* Элементы выбора - флажки для указания, какие именно показатели прибыли требуется рассчитать;
* Управляющие элементы - кнопки для выполнения расчета и очистки введенных данных;
* Область вывода - секция для отображения результатов вычислений
* Информационный блок - раздел с формулами расчета и тестовыми примерами для проверки правильности работы калькулятора.

3) Модуль расчета финансовых показателей

Расчетный модуль функционирует по следующему алгоритму:

* Определяет, какие показатели прибыли пользователь выбрал для расчета посредством проверки состояния соответствующих флажков;
* Извлекает числовые значения из полей ввода, преобразуя текстовые данные в числовой формат;
* Выполняет вычисления по установленным финансовым формулам для каждого выбранного показателя;
* Сохраняет полученные результаты в специализированных переменных для последующего отображения в интерфейсе.

4) Система управления данными

Реализован механизм сброса состояния калькулятора, который:

* Обнуляет все финансовые показатели и результаты предыдущих расчетов;
* Сбрасывает состояние элементов выбора (флажков);
* Деактивирует текущее поле ввода;
* Инициирует перерисовку интерфейса для отражения измененного состояния.

5) Подсистема обработки пользовательского ввода

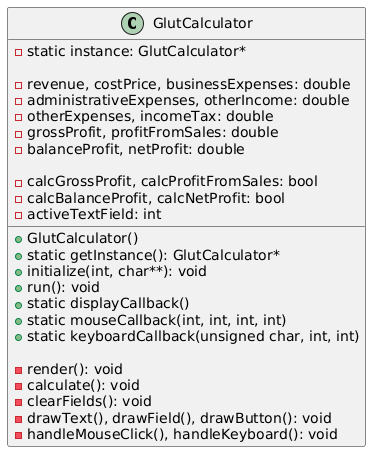
Обеспечивается двунаправленное взаимодействие с пользователем через:

* Манипулятор "мышь" - обработка кликов по элементам интерфейса, включая выбор полей ввода, переключение флажков и активацию кнопок управления;
* Клавиатурный ввод - возможность ввода цифровых значений в активные текстовые поля с поддержкой функции удаления символов.

6) Финансовые формулы расчета

1. Валовая прибыль = Выручка – Себестоимость;
2. Прибыль от продаж = Выручка - Себестоимость – Коммерческие расходы - Управленческие расходы;
3. Балансовая прибыль = Прибыль от продаж + Прочие доходы - Прочие расходы;
4. Чистая прибыль = Балансовая прибыль - Налог на прибыль.

**UML Диаграмма**



**Код программы**

**Файл calculator\_glut.h**

#pragma once

#include "freeglut.h"

#include <string>

#include <vector>

class GlutCalculator {

private:

static GlutCalculator\* instance;

// Данные калькулятора

double revenue, costPrice, businessExpenses, administrativeExpenses;

double otherIncome, otherExpenses, incomeTax;

double grossProfit, profitFromSales, balanceProfit, netProfit;

// Флаги расчета

bool calcGrossProfit, calcProfitFromSales, calcBalanceProfit, calcNetProfit;

// Активное текстовое поле

int activeTextField;

public:

GlutCalculator();

static GlutCalculator\* getInstance();

void initialize(int argc, char\*\* argv);

void run();

// Статические методы для callback'ов GLUT

static void displayCallback();

static void mouseCallback(int button, int state, int x, int y);

static void keyboardCallback(unsigned char key, int x, int y);

private:

void render();

void handleMouseClick(int x, int y);

void handleKeyboard(unsigned char key);

void calculate();

void clearFields();

// Методы отрисовки

void drawText(float x, float y, const std::string& text);

void drawRect(float x, float y, float width, float height);

void drawField(float x, float y, float width, float height, const std::string& value, bool active);

void drawCheckbox(float x, float y, const std::string& label, bool checked);

void drawButton(float x, float y, float width, float height, const std::string& label);

double getFieldValue(int index);

bool getCheckboxState(int index);

};

**Файл calculator\_glut.cpp**

#include "calculator\_glut.h"

#include <sstream>

#include <iomanip>

GlutCalculator\* GlutCalculator::instance = nullptr;

GlutCalculator::GlutCalculator() {

// Инициализация данных

revenue = costPrice = businessExpenses = administrativeExpenses = 0;

otherIncome = otherExpenses = incomeTax = 0;

grossProfit = profitFromSales = balanceProfit = netProfit = 0;

calcGrossProfit = calcProfitFromSales = calcBalanceProfit = calcNetProfit = false;

activeTextField = -1;

}

GlutCalculator\* GlutCalculator::getInstance() {

if (!instance) {

instance = new GlutCalculator();

}

return instance;

}

void GlutCalculator::initialize(int argc, char\*\* argv) {

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_DOUBLE | GLUT\_RGB);

glutInitWindowSize(1200, 700); // Фиксированный размер

glutInitWindowPosition(100, 100);

glutCreateWindow("Financial Calculator - GLUT");

// Устанавливаем фиксированную систему координат

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

gluOrtho2D(0, 1200, 700, 0); // (0,0) - левый верхний угол

glMatrixMode(GL\_MODELVIEW);

glClearColor(0.53f, 0.81f, 0.98f, 1.0f); // Цвет фона

// Регистрация callback'ов

glutDisplayFunc(displayCallback);

glutMouseFunc(mouseCallback);

glutKeyboardFunc(keyboardCallback);

}

void GlutCalculator::run() {

glutMainLoop();

}

// Static callback методы

void GlutCalculator::displayCallback() {

getInstance()->render();

}

void GlutCalculator::mouseCallback(int button, int state, int x, int y) {

if (button == GLUT\_LEFT\_BUTTON && state == GLUT\_DOWN) {

getInstance()->handleMouseClick(x, y);

}

}

void GlutCalculator::keyboardCallback(unsigned char key, int x, int y) {

getInstance()->handleKeyboard(key);

}

void GlutCalculator::render() {

glClear(GL\_COLOR\_BUFFER\_BIT);

glLoadIdentity();

// ===== ЗАГОЛОВОК =====

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

drawText(500, 30, "FINANCIAL CALCULATOR");

// ===== ПОДПИСИ ПОЛЕЙ ВВОДА =====

drawText(50, 95, "Revenue");

drawText(50, 145, "Cost Price");

drawText(50, 195, "Business Expenses");

drawText(300, 95, "Admin Expenses");

drawText(300, 145, "Other Income");

drawText(300, 195, "Other Expenses");

drawText(550, 95, "Income Tax");

// ===== ПОЛЯ ВВОДА =====

drawField(50, 100, 200, 25, std::to\_string((int)revenue), activeTextField == 0);

drawField(50, 150, 200, 25, std::to\_string((int)costPrice), activeTextField == 1);

drawField(50, 200, 200, 25, std::to\_string((int)businessExpenses), activeTextField == 2);

drawField(300, 100, 200, 25, std::to\_string((int)administrativeExpenses), activeTextField == 3);

drawField(300, 150, 200, 25, std::to\_string((int)otherIncome), activeTextField == 4);

drawField(300, 200, 200, 25, std::to\_string((int)otherExpenses), activeTextField == 5);

drawField(550, 100, 200, 25, std::to\_string((int)incomeTax), activeTextField == 6);

// ===== ЧЕКБОКСЫ =====

drawCheckbox(50, 250, "Gross Profit", calcGrossProfit);

drawCheckbox(50, 300, "Profit from Sales", calcProfitFromSales);

drawCheckbox(50, 350, "Balance Profit", calcBalanceProfit);

drawCheckbox(50, 400, "Net Profit", calcNetProfit);

// ===== КНОПКИ =====

drawButton(550, 250, 150, 40, "CALCULATE");

drawButton(550, 310, 150, 40, "CLEAR");

// ===== РЕЗУЛЬТАТЫ =====

drawText(800, 100, "RESULTS:");

drawText(800, 130, "Gross Profit: " + std::to\_string((int)grossProfit));

drawText(800, 160, "Sales Profit: " + std::to\_string((int)profitFromSales));

drawText(800, 190, "Balance Profit: " + std::to\_string((int)balanceProfit));

drawText(800, 220, "Net Profit: " + std::to\_string((int)netProfit));

// ===== ФОРМУЛЫ РАСЧЕТА =====

glColor3f(0.3f, 0.3f, 0.3f); // Темно-серый цвет для формул

drawText(50, 500, "CALCULATION FORMULAS:");

drawText(50, 520, "Gross Profit = Revenue - Cost Price");

drawText(50, 540, "Profit from Sales = Revenue - Cost Price - Business Expenses - Admin Expenses");

drawText(50, 560, "Balance Profit = Profit from Sales + Other Income - Other Expenses");

drawText(50, 580, "Net Profit = Balance Profit - Income Tax");

glutSwapBuffers();

}

void GlutCalculator::handleMouseClick(int x, int y) {

// Проверка текстовых полей

int fieldsY[] = { 100, 150, 200, 100, 150, 200, 100 };

int fieldsX[] = { 50, 50, 50, 300, 300, 300, 550 };

for (int i = 0; i < 7; i++) {

if (x >= fieldsX[i] && x <= fieldsX[i] + 200 &&

y >= fieldsY[i] && y <= fieldsY[i] + 25) {

activeTextField = i;

glutPostRedisplay();

return;

}

}

// Проверка чекбоксов

int checkY[] = { 250, 300, 350, 400 };

for (int i = 0; i < 4; i++) {

if (x >= 50 && x <= 70 && y >= checkY[i] && y <= checkY[i] + 20) {

switch (i) {

case 0: calcGrossProfit = !calcGrossProfit; break;

case 1: calcProfitFromSales = !calcProfitFromSales; break;

case 2: calcBalanceProfit = !calcBalanceProfit; break;

case 3: calcNetProfit = !calcNetProfit; break;

}

glutPostRedisplay();

return;

}

}

// Проверка кнопок

if (x >= 550 && x <= 700 && y >= 250 && y <= 290) {

calculate();

glutPostRedisplay();

}

else if (x >= 550 && x <= 700 && y >= 310 && y <= 350) {

clearFields();

glutPostRedisplay();

}

}

void GlutCalculator::handleKeyboard(unsigned char key) {

if (activeTextField >= 0) {

double\* fields[] = { &revenue, &costPrice, &businessExpenses, &administrativeExpenses,

&otherIncome, &otherExpenses, &incomeTax };

if (key >= '0' && key <= '9') {

// Добавляем цифру

\*fields[activeTextField] = \*fields[activeTextField] \* 10 + (key - '0');

}

else if (key == 8 || key == 127) {

// Backspace - удаляем последнюю цифру

\*fields[activeTextField] = (int)(\*fields[activeTextField] / 10);

}

else if (key == 13 || key == 27) {

// Enter или Escape - снимаем фокус

activeTextField = -1;

}

glutPostRedisplay();

}

}

void GlutCalculator::calculate() {

if (calcGrossProfit) {

grossProfit = revenue - costPrice;

}

if (calcProfitFromSales) {

profitFromSales = revenue - costPrice - businessExpenses - administrativeExpenses;

}

if (calcBalanceProfit) {

balanceProfit = revenue - costPrice - businessExpenses - administrativeExpenses + otherIncome - otherExpenses;

}

if (calcNetProfit) {

netProfit = revenue - costPrice - businessExpenses - administrativeExpenses + otherIncome - otherExpenses - incomeTax;

}

}

void GlutCalculator::clearFields() {

revenue = costPrice = businessExpenses = administrativeExpenses = 0;

otherIncome = otherExpenses = incomeTax = 0;

grossProfit = profitFromSales = balanceProfit = netProfit = 0;

calcGrossProfit = calcProfitFromSales = calcBalanceProfit = calcNetProfit = false;

activeTextField = -1;

}

// ===== МЕТОДЫ ОТРИСОВКИ =====

void GlutCalculator::drawText(float x, float y, const std::string& text) {

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

glRasterPos2f(x, y);

for (char c : text) {

glutBitmapCharacter(GLUT\_BITMAP\_9\_BY\_15, c);

}

}

void GlutCalculator::drawRect(float x, float y, float width, float height) {

glBegin(GL\_QUADS);

glVertex2f(x, y);

glVertex2f(x + width, y);

glVertex2f(x + width, y + height);

glVertex2f(x, y + height);

glEnd();

}

void GlutCalculator::drawField(float x, float y, float width, float height, const std::string& value, bool active) {

// Фон поля

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

drawRect(x, y, width, height);

// Рамка

if (active) {

glColor3f(0.0f, 0.0f, 1.0f);

}

else {

glColor3f(0.5f, 0.5f, 0.5f);

}

glLineWidth(2.0f);

glBegin(GL\_LINE\_LOOP);

glVertex2f(x, y);

glVertex2f(x + width, y);

glVertex2f(x + width, y + height);

glVertex2f(x, y + height);

glEnd();

// Текст значения

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

drawText(x + 5, y + 15, value);

}

void GlutCalculator::drawCheckbox(float x, float y, const std::string& label, bool checked) {

// Квадрат чекбокса

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

drawRect(x, y, 20, 20);

// Рамка

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

glBegin(GL\_LINE\_LOOP);

glVertex2f(x, y);

glVertex2f(x + 20, y);

glVertex2f(x + 20, y + 20);

glVertex2f(x, y + 20);

glEnd();

// Галочка

if (checked) {

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

glBegin(GL\_LINES);

glVertex2f(x + 5, y + 10);

glVertex2f(x + 10, y + 15);

glVertex2f(x + 10, y + 15);

glVertex2f(x + 15, y + 5);

glEnd();

}

// Текст

drawText(x + 25, y + 5, label);

}

void GlutCalculator::drawButton(float x, float y, float width, float height, const std::string& label) {

// Фон кнопки

glColor3f(0.8f, 0.8f, 0.8f);

drawRect(x, y, width, height);

// Рамка

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

glBegin(GL\_LINE\_LOOP);

glVertex2f(x, y);

glVertex2f(x + width, y);

glVertex2f(x + width, y + height);

glVertex2f(x, y + height);

glEnd();

// Текст

drawText(x + 10, y + height / 2 + 5, label);

}

double GlutCalculator::getFieldValue(int index) {

double fields[] = { revenue, costPrice, businessExpenses, administrativeExpenses,

otherIncome, otherExpenses, incomeTax };

return fields[index];

}

bool GlutCalculator::getCheckboxState(int index) {

bool states[] = { calcGrossProfit, calcProfitFromSales, calcBalanceProfit, calcNetProfit };

return states[index];

}

**Файл main.cpp**

#include "calculator\_glut.h"

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

GlutCalculator\* calculator = GlutCalculator::getInstance();

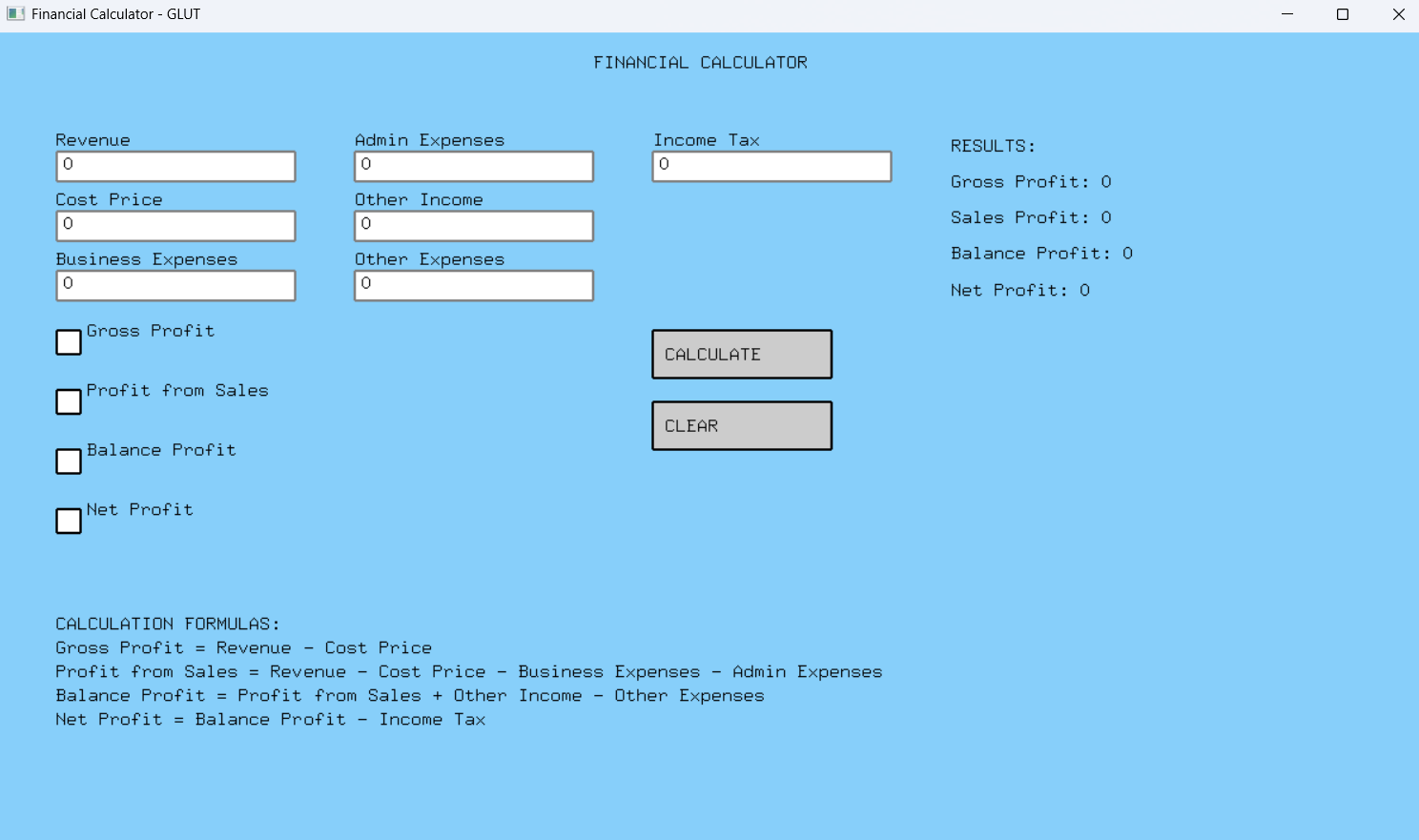
calculator->initialize(argc, argv);

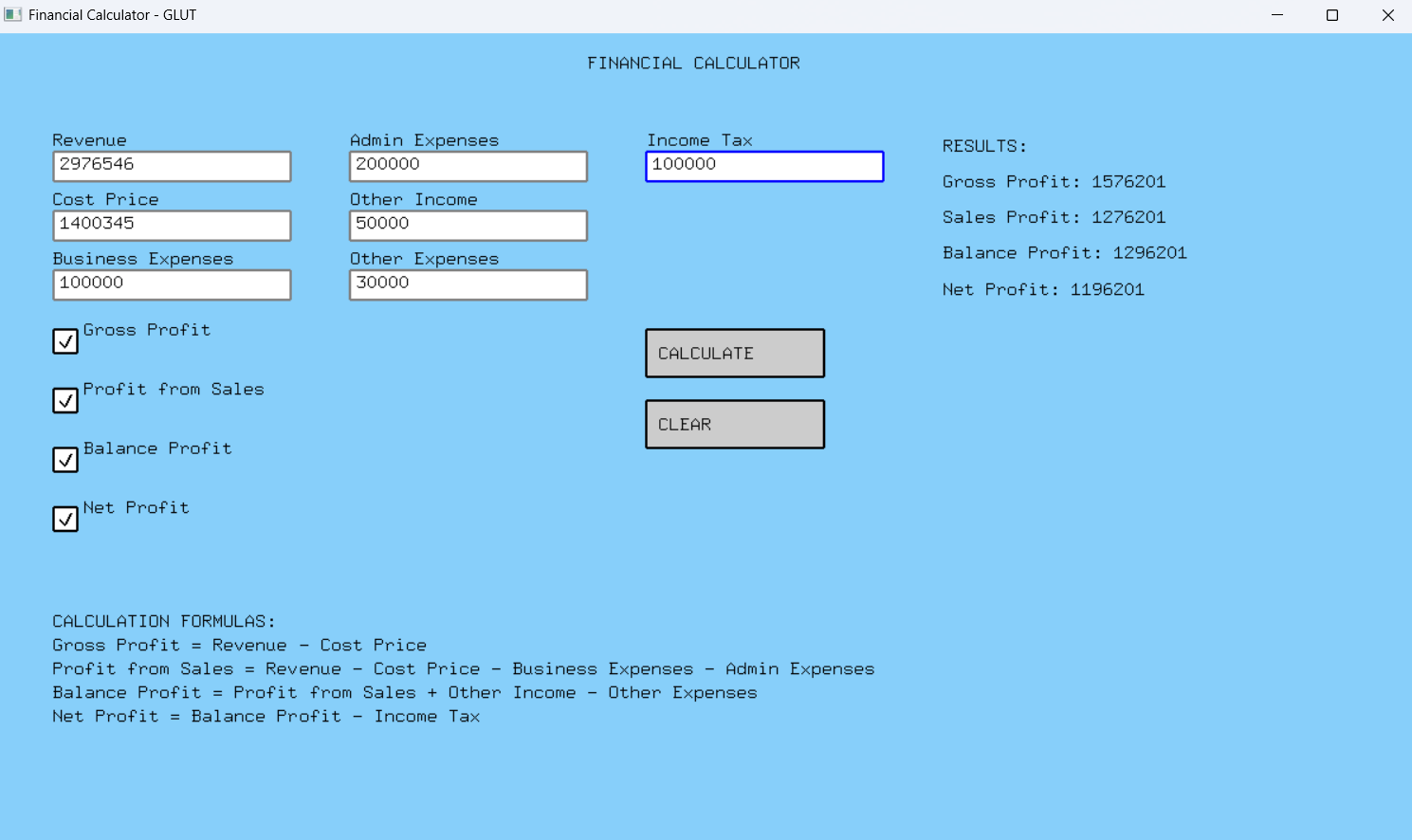
calculator->run();

return 0;

}

**Результаты работы программы**





**Задача коммивояжера**

**Постановка задачи**

Существует *N* городов, соединенных путями по принципу «каждый с каждым». Все пути имеют вес, расстояние между городами. Задача коммивояжера состоит в том, чтобы объехать все города, побывав в каждом лишь один раз, при этом требуется, чтобы сумма расстояний между городами была минимальной.

**Анализ задачи и архитектура решения**

1. Структура данных и глобальные компоненты

Класс GraphApp

Центральный класс приложения

Структуры данных:

Структура Node: Хранит координаты вершин (x, y) и их уникальные идентификаторы

Структура Edge: Содержит информацию о связях между вершинами (from, to) и весами рёбер

Основные коллекции:

Вектор nodes: Динамический массив всех вершин графа

Вектор edges: Динамический массив всех рёбер графа

2. Система управления графом

Методы манипуляции вершинами:

addNode(): Добавляет новую вершину в указанных координатах

removeNode(): Удаляет вершину по идентификатору и все связанные с ней рёбра

Методы манипуляции рёбрами:

addEdge(): Создает новое ребро между вершинами с указанным весом

removeEdge(): Удаляет ребро между заданными вершинами

Методы поиска и валидации:

findNodeAt(): Определяет вершину по координатам мыши

getEdgeWeight(): Возвращает вес ребра между двумя вершинами

getNodeIndex(): Находит индекс вершины в массиве по её идентификатору

3. Подсистема визуализации

Методы отрисовки графических элементов:

drawNode(): Отрисовывает вершину в виде круга с идентификатором

drawEdge(): Рисует линию ребра с отображением веса

drawButton(): Создает интерактивные кнопки управления

drawText(): Выводит текстовые надписи на экран

drawRect(): Рисует прямоугольные элементы интерфейса

Визуальные состояния:

Выделение активной вершины цветом

Отображение весов рёбер

Визуальный feedback при наведении и выборе

4. Модуль решения задачи коммивояжера (TSP)

Алгоритм ветвей и границ:

solveTSPBranchAndBound(): Основной метод решения TSP

bnbRecursive(): Рекурсивная функция поиска с возвратом

calculateLowerBound(): Вычисляет нижнюю границу для отсечения ветвей

Особенности реализации:

Поддержка выбора начальной вершины

Оптимизация через отсечение заведомо неперспективных путей

Гарантированное нахождение оптимального решения

5. Система взаимодействия с пользователем

Обработка событий мыши:

handleMouseClick(): Обрабатывает клики для создания вершин, выбора элементов и активации кнопок

handleMouseMotion(): Обеспечивает функциональность перетаскивания вершин

Обработка клавиатуры:

handleKeyboard(): Обеспечивает горячие клавиши для быстрого доступа к функциям

Консольный интерфейс:

consoleAddNode(), consoleRemoveNode(): Управление вершинами через консоль

consoleAddEdge(), consoleRemoveEdge(): Управление рёбрами через консоль

6. Вспомогательные сервисные функции

Утилиты отображения:

printMatrix(): Выводит матрицу смежности в консоль

render(): Основной метод рендеринга интерфейса

Функции инициализации:

initialize(): Настраивает графическое окружение и обработчики событий

run(): Запускает главный цикл приложения

7. Архитектурные особенности

Шаблон проектирования:

Реализация Singleton для управления состоянием приложения

Разделение ответственности между модулями

Управление состоянием:

nextNodeId: Счетчик для генерации уникальных идентификаторов вершин

dragging, selectedNode: Переменные для отслеживания взаимодействия

dragOffsetX, dragOffsetY: Смещения для плавного перетаскивания

Обработка граничных случаев:

Проверка существования вершин перед созданием рёбер

Валидация входных данных

Обработка отсутствия гамильтонова цикла

8. Интерфейс пользователя

Элементы управления:

Кнопки для операций с графом (добавление, удаление, расчет)

Визуальное представление матрицы смежности

Область отображения результатов TSP

Инструкции и справочная информация

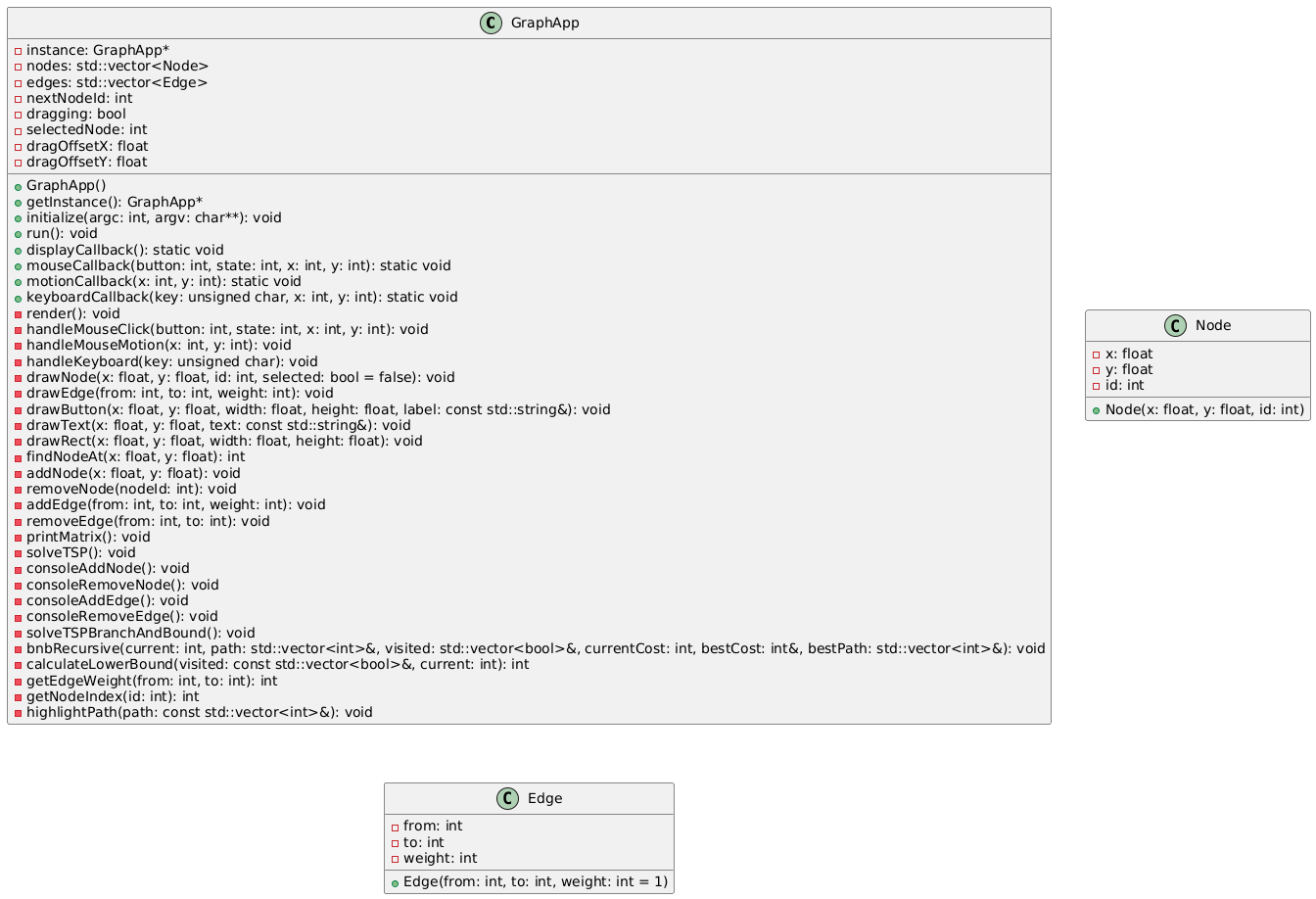
Интерактивные возможности:

Drag-and-drop для перемещения вершин

Визуальное выделение активных элементов

Немедленное обновление интерфейса при изменениях

**UML Диаграмма**

****

**Код программы**

**Файл graph\_app.h**

#pragma once

#include "freeglut.h"

#include <vector>

#include <string>

#include <algorithm>

class GraphApp {

private:

static GraphApp\* instance;

struct Node {

float x, y;

int id;

Node(float x, float y, int id) : x(x), y(y), id(id) {}

};

struct Edge {

int from, to, weight;

Edge(int from, int to, int weight = 1) : from(from), to(to), weight(weight) {}

};

std::vector<Node> nodes;

std::vector<Edge> edges;

int nextNodeId;

bool dragging;

int selectedNode;

float dragOffsetX, dragOffsetY;

public:

GraphApp();

static GraphApp\* getInstance();

void initialize(int argc, char\*\* argv);

void run();

static void displayCallback();

static void mouseCallback(int button, int state, int x, int y);

static void motionCallback(int x, int y);

static void keyboardCallback(unsigned char key, int x, int y);

private:

void render();

void handleMouseClick(int button, int state, int x, int y);

void handleMouseMotion(int x, int y);

void handleKeyboard(unsigned char key);

void drawNode(float x, float y, int id, bool selected = false);

void drawEdge(int from, int to, int weight);

void drawButton(float x, float y, float width, float height, const std::string& label);

void drawText(float x, float y, const std::string& text);

void drawRect(float x, float y, float width, float height);

int findNodeAt(float x, float y);

void addNode(float x, float y);

void removeNode(int nodeId);

void addEdge(int from, int to, int weigth);

void removeEdge(int from, int to);

void printMatrix();

void solveTSP();

void consoleAddNode();

void consoleRemoveNode();

void consoleAddEdge();

void consoleRemoveEdge();

void solveTSPBranchAndBound();

void bnbRecursive(int current, std::vector<int>& path, std::vector<bool>& visited,

int currentCost, int& bestCost, std::vector<int>& bestPath);

int calculateLowerBound(const std::vector<bool>& visited, int current);

int getEdgeWeight(int from, int to);

int getNodeIndex(int id);

void highlightPath(const std::vector<int>& path);

};

**Файл graph\_app.cpp**

#include "graph\_app.h"

#include <iostream>

#include <sstream>

#include <limits>

#include <algorithm>

GraphApp\* GraphApp::instance = nullptr;

GraphApp::GraphApp() : nextNodeId(1), dragging(false), selectedNode(-1) {

// Начальные вершины

addNode(200, 200);

addNode(400, 200);

addNode(300, 400);

}

GraphApp\* GraphApp::getInstance() {

if (!instance) {

instance = new GraphApp();

}

return instance;

}

void GraphApp::initialize(int argc, char\*\* argv) {

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_DOUBLE | GLUT\_RGB);

glutInitWindowSize(1000, 700);

glutInitWindowPosition(100, 100);

glutCreateWindow("Graph Application - TSP Solver");

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

gluOrtho2D(0, 1000, 700, 0);

glMatrixMode(GL\_MODELVIEW);

glClearColor(0.9f, 0.95f, 1.0f, 1.0f);

glutDisplayFunc(displayCallback);

glutMouseFunc(mouseCallback);

glutMotionFunc(motionCallback);

glutKeyboardFunc(keyboardCallback);

}

void GraphApp::run() {

glutMainLoop();

}

// Static callback методы

void GraphApp::displayCallback() {

getInstance()->render();

}

void GraphApp::mouseCallback(int button, int state, int x, int y) {

getInstance()->handleMouseClick(button, state, x, y);

}

void GraphApp::motionCallback(int x, int y) {

getInstance()->handleMouseMotion(x, y);

}

void GraphApp::keyboardCallback(unsigned char key, int x, int y) {

getInstance()->handleKeyboard(key);

}

void GraphApp::render() {

glClear(GL\_COLOR\_BUFFER\_BIT);

glLoadIdentity();

// ===== ЗАГОЛОВОК =====

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

drawText(400, 30, "GRAPH APPLICATION - TSP SOLVER");

// ===== ОТРИСОВКА РЕБЕР =====

for (const auto& edge : edges) {

drawEdge(edge.from, edge.to, edge.weight);

}

// ===== ОТРИСОВКА ВЕРШИН =====

for (const auto& node : nodes) {

bool selected = (node.id == selectedNode);

drawNode(node.x, node.y, node.id, selected);

}

// ===== ПАНЕЛЬ КНОПОК =====

drawButton(50, 600, 150, 40, "PRINT MATRIX");

drawButton(220, 600, 150, 40, "ADD NODE");

drawButton(390, 600, 150, 40, "REMOVE NODE");

drawButton(560, 600, 150, 40, "ADD EDGE");

drawButton(730, 600, 150, 40, "REMOVE EDGE");

drawButton(400, 650, 200, 40, "SOLVE TSP");

// ===== ИНСТРУКЦИЯ =====

glColor3f(0.3f, 0.3f, 0.3f);

drawText(50, 500, "INSTRUCTIONS:");

drawText(50, 520, "Click empty space: Add node");

drawText(50, 540, "Click node + drag: Move node");

drawText(50, 560, "Press keys 1-6: Use buttons");

glutSwapBuffers();

}

void GraphApp::handleMouseClick(int button, int state, int x, int y) {

if (button == GLUT\_LEFT\_BUTTON && state == GLUT\_DOWN) {

// Проверка клика по кнопкам

if (y >= 600 && y <= 640) {

if (x >= 50 && x <= 200) { printMatrix(); return; }

if (x >= 220 && x <= 370) { consoleAddNode(); return; }

if (x >= 390 && x <= 540) { consoleRemoveNode(); return; }

if (x >= 560 && x <= 710) { consoleAddEdge(); return; }

if (x >= 730 && x <= 880) { consoleRemoveEdge(); return; }

}

if (y >= 650 && y <= 690 && x >= 400 && x <= 600) {

solveTSP(); return;

}

// Проверка клика по вершине

int clickedNode = findNodeAt(x, y);

if (clickedNode != -1) {

dragging = true;

selectedNode = clickedNode;

auto it = std::find\_if(nodes.begin(), nodes.end(),

[clickedNode](const Node& n) { return n.id == clickedNode; });

if (it != nodes.end()) {

dragOffsetX = x - it->x;

dragOffsetY = y - it->y;

}

}

else {

// Добавление новой вершины

addNode(x, y);

}

glutPostRedisplay();

}

else if (button == GLUT\_LEFT\_BUTTON && state == GLUT\_UP) {

dragging = false;

}

}

void GraphApp::handleMouseMotion(int x, int y) {

if (dragging && selectedNode != -1) {

auto it = std::find\_if(nodes.begin(), nodes.end(),

[this](const Node& n) { return n.id == selectedNode; });

if (it != nodes.end()) {

it->x = x - dragOffsetX;

it->y = y - dragOffsetY;

glutPostRedisplay();

}

}

}

void GraphApp::handleKeyboard(unsigned char key) {

switch (key) {

case '1': printMatrix(); break;

case '2': consoleAddNode(); break;

case '3': consoleRemoveNode(); break;

case '4': consoleAddEdge(); break;

case '5': consoleRemoveEdge(); break;

case '6': solveTSP(); break;

}

}

void GraphApp::drawNode(float x, float y, int id, bool selected) {

// Внешний круг

if (selected) {

glColor3f(1.0f, 0.5f, 0.0f); // Оранжевый для выделенной

}

else {

glColor3f(0.2f, 0.4f, 0.8f); // Синий для обычной

}

glBegin(GL\_TRIANGLE\_FAN);

for (int i = 0; i <= 360; i += 30) {

float angle = i \* 3.14159f / 180.0f;

glVertex2f(x + cos(angle) \* 20, y + sin(angle) \* 20);

}

glEnd();

// Внутренний круг

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

glBegin(GL\_TRIANGLE\_FAN);

for (int i = 0; i <= 360; i += 30) {

float angle = i \* 3.14159f / 180.0f;

glVertex2f(x + cos(angle) \* 15, y + sin(angle) \* 15);

}

glEnd();

// Номер вершины

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

std::string idStr = std::to\_string(id);

drawText(x - 5, y - 5, idStr);

}

void GraphApp::drawEdge(int from, int to, int weight) {

auto fromIt = std::find\_if(nodes.begin(), nodes.end(),

[from](const Node& n) { return n.id == from; });

auto toIt = std::find\_if(nodes.begin(), nodes.end(),

[to](const Node& n) { return n.id == to; });

if (fromIt == nodes.end() || toIt == nodes.end()) return;

// Линия ребра

glColor3f(0.3f, 0.3f, 0.3f);

glLineWidth(2.0f);

glBegin(GL\_LINES);

glVertex2f(fromIt->x, fromIt->y);

glVertex2f(toIt->x, toIt->y);

glEnd();

// Вес ребра (посередине)

float midX = (fromIt->x + toIt->x) / 2;

float midY = (fromIt->y + toIt->y) / 2;

glColor3f(0.8f, 0.0f, 0.0f);

std::string weightStr = std::to\_string(weight);

drawText(midX - 5, midY - 5, weightStr);

}

void GraphApp::drawButton(float x, float y, float width, float height, const std::string& label) {

// Фон кнопки

glColor3f(0.7f, 0.8f, 0.9f);

drawRect(x, y, width, height);

// Рамка

glColor3f(0.2f, 0.2f, 0.2f);

glLineWidth(2.0f);

glBegin(GL\_LINE\_LOOP);

glVertex2f(x, y);

glVertex2f(x + width, y);

glVertex2f(x + width, y + height);

glVertex2f(x, y + height);

glEnd();

// Текст

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

drawText(x + 10, y + height / 2 + 5, label);

}

void GraphApp::drawText(float x, float y, const std::string& text) {

glRasterPos2f(x, y);

for (char c : text) {

glutBitmapCharacter(GLUT\_BITMAP\_9\_BY\_15, c);

}

}

void GraphApp::drawRect(float x, float y, float width, float height) {

glBegin(GL\_QUADS);

glVertex2f(x, y);

glVertex2f(x + width, y);

glVertex2f(x + width, y + height);

glVertex2f(x, y + height);

glEnd();

}

int GraphApp::findNodeAt(float x, float y) {

for (const auto& node : nodes) {

float dx = node.x - x;

float dy = node.y - y;

if (dx \* dx + dy \* dy <= 20 \* 20) {

return node.id;

}

}

return -1;

}

void GraphApp::addNode(float x, float y) {

nodes.emplace\_back(x, y, nextNodeId++);

}

void GraphApp::removeNode(int nodeId) {

// Удаляем вершину

nodes.erase(std::remove\_if(nodes.begin(), nodes.end(),

[nodeId](const Node& n) { return n.id == nodeId; }), nodes.end());

// Удаляем связанные ребра

edges.erase(std::remove\_if(edges.begin(), edges.end(),

[nodeId](const Edge& e) { return e.from == nodeId || e.to == nodeId; }), edges.end());

}

void GraphApp::addEdge(int from, int to, int weight) {

// Проверяем, существует ли уже такое ребро

for (const auto& edge : edges) {

if ((edge.from == from && edge.to == to) || (edge.from == to && edge.to == from)) {

std::cout << "Edge already exists!" << std::endl;

return;

}

}

// Проверяем, существуют ли вершины

bool fromExists = std::any\_of(nodes.begin(), nodes.end(),

[from](const Node& n) { return n.id == from; });

bool toExists = std::any\_of(nodes.begin(), nodes.end(),

[to](const Node& n) { return n.id == to; });

if (!fromExists || !toExists) {

std::cout << "One or both nodes don't exist!" << std::endl;

return;

}

edges.emplace\_back(from, to, weight);

}

void GraphApp::removeEdge(int from, int to) {

edges.erase(std::remove\_if(edges.begin(), edges.end(),

[from, to](const Edge& e) {

return (e.from == from && e.to == to) || (e.from == to && e.to == from);

}), edges.end());

}

void GraphApp::printMatrix() {

if (nodes.empty()) {

std::cout << "Graph is empty!" << std::endl;

return;

}

std::cout << "\n=== ADJACENCY MATRIX ===" << std::endl;

// Заголовок

std::cout << " ";

for (const auto& node : nodes) {

std::cout << node.id << " ";

}

std::cout << std::endl;

// Матрица

for (const auto& node1 : nodes) {

std::cout << node1.id << " ";

for (const auto& node2 : nodes) {

bool connected = false;

for (const auto& edge : edges) {

if ((edge.from == node1.id && edge.to == node2.id) ||

(edge.from == node2.id && edge.to == node1.id)) {

connected = true;

break;

}

}

std::cout << (connected ? "1 " : "0 ");

}

std::cout << std::endl;

}

std::cout << "========================" << std::endl;

}

void GraphApp::solveTSP() {

solveTSPBranchAndBound();

}

void GraphApp::consoleAddNode() {

std::cout << "Enter node coordinates (x y): ";

float x, y;

std::cin >> x >> y;

addNode(x, y);

glutPostRedisplay();

}

void GraphApp::consoleRemoveNode() {

std::cout << "Enter node ID to remove: ";

int id;

std::cin >> id;

removeNode(id);

glutPostRedisplay();

}

void GraphApp::consoleAddEdge() {

std::cout << "Enter edge (from to weight): ";

int from, to, weight;

std::cin >> from >> to >> weight;

if (weight <= 0) {

std::cout << "Weight must be positive!" << std::endl;

return;

}

addEdge(from, to, weight);

glutPostRedisplay();

}

void GraphApp::consoleRemoveEdge() {

std::cout << "Enter edge to remove (from to): ";

int from, to;

std::cin >> from >> to;

removeEdge(from, to);

glutPostRedisplay();

}

int GraphApp::getEdgeWeight(int from, int to) {

for (const auto& edge : edges) {

if ((edge.from == from && edge.to == to) || (edge.from == to && edge.to == from)) {

return edge.weight;

}

}

return -1; // Ребра не существует

}

int GraphApp::getNodeIndex(int id) {

for (size\_t i = 0; i < nodes.size(); i++) {

if (nodes[i].id == id) return i;

}

return -1;

}

void GraphApp::solveTSPBranchAndBound() {

if (nodes.size() < 2) {

std::cout << "Need at least 2 nodes for TSP!" << std::endl;

return;

}

// Показываем доступные вершины

std::cout << "\nAvailable nodes: ";

for (const auto& node : nodes) {

std::cout << node.id << " ";

}

std::cout << std::endl;

// Запрашиваем начальную вершину

std::cout << "Enter starting node ID: ";

int startNode;

std::cin >> startNode;

// Проверяем существование вершины

if (getNodeIndex(startNode) == -1) {

std::cout << "Node " << startNode << " doesn't exist!" << std::endl;

return;

}

std::cout << "\n=== TSP SOLUTION (Branch and Bound) ===" << std::endl;

std::cout << "Starting from node: " << startNode << std::endl;

std::vector<int> bestPath;

std::vector<bool> visited(nodes.size(), false);

int bestCost = std::numeric\_limits<int>::max();

// Начинаем рекурсию

std::vector<int> currentPath = { startNode };

visited[getNodeIndex(startNode)] = true;

bnbRecursive(startNode, currentPath, visited, 0, bestCost, bestPath);

// Выводим результат

if (bestCost != std::numeric\_limits<int>::max()) {

std::cout << "Optimal path: ";

for (size\_t i = 0; i < bestPath.size(); i++) {

std::cout << bestPath[i];

if (i < bestPath.size() - 1) std::cout << " -> ";

}

std::cout << " -> " << bestPath[0] << " (return to start)" << std::endl;

std::cout << "Total distance: " << bestCost << std::endl;

// Подсветка оптимального пути на графе

highlightPath(bestPath);

}

else {

std::cout << "No Hamiltonian cycle found!" << std::endl;

}

std::cout << "=====================================" << std::endl;

}

void GraphApp::bnbRecursive(int current, std::vector<int>& path,

std::vector<bool>& visited, int currentCost,

int& bestCost, std::vector<int>& bestPath) {

// Если все вершины посещены

if (path.size() == nodes.size()) {

// Пытаемся вернуться в начало

int returnWeight = getEdgeWeight(current, path[0]);

if (returnWeight != -1) {

int totalCost = currentCost + returnWeight;

if (totalCost < bestCost) {

bestCost = totalCost;

bestPath = path;

std::cout << "Found better solution: " << bestCost << std::endl;

}

}

return;

}

// Вычисляем нижнюю границу

int lowerBound = calculateLowerBound(visited, current);

if (currentCost + lowerBound >= bestCost) {

return; // Отсекаем ветвь

}

// Перебираем все возможные следующие вершины

for (size\_t i = 0; i < nodes.size(); i++) {

if (!visited[i]) {

int nextNode = nodes[i].id;

int edgeWeight = getEdgeWeight(current, nextNode);

if (edgeWeight != -1) { // Если ребро существует

visited[i] = true;

path.push\_back(nextNode);

bnbRecursive(nextNode, path, visited, currentCost + edgeWeight,

bestCost, bestPath);

// Backtrack

path.pop\_back();

visited[i] = false;

}

}

}

}

int GraphApp::calculateLowerBound(const std::vector<bool>& visited, int current) {

int bound = 0;

// Минимальное ребро из текущей вершины к непосещенным

int minFromCurrent = std::numeric\_limits<int>::max();

for (size\_t i = 0; i < nodes.size(); i++) {

if (!visited[i]) {

int weight = getEdgeWeight(current, nodes[i].id);

if (weight != -1 && weight < minFromCurrent) {

minFromCurrent = weight;

}

}

}

if (minFromCurrent != std::numeric\_limits<int>::max()) {

bound += minFromCurrent;

}

// Минимальные ребра для каждой непосещенной вершины

for (size\_t i = 0; i < nodes.size(); i++) {

if (!visited[i]) {

int minEdge1 = std::numeric\_limits<int>::max();

int minEdge2 = std::numeric\_limits<int>::max();

// Два минимальных ребра из этой вершины

for (size\_t j = 0; j < nodes.size(); j++) {

if (i != j) {

int weight = getEdgeWeight(nodes[i].id, nodes[j].id);

if (weight != -1) {

if (weight < minEdge1) {

minEdge2 = minEdge1;

minEdge1 = weight;

}

else if (weight < minEdge2) {

minEdge2 = weight;

}

}

}

}

if (minEdge1 != std::numeric\_limits<int>::max()) {

bound += minEdge1;

if (minEdge2 != std::numeric\_limits<int>::max()) {

bound += minEdge2;

}

}

}

}

return bound / 2; // Каждое ребро учтено дважды

}

void GraphApp::highlightPath(const std::vector<int>& path) {

std::cout << "Optimal path highlighted in console output" << std::endl;

}

**Файл main.cpp**

#include "graph\_app.h"

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

GraphApp\* app = GraphApp::getInstance();

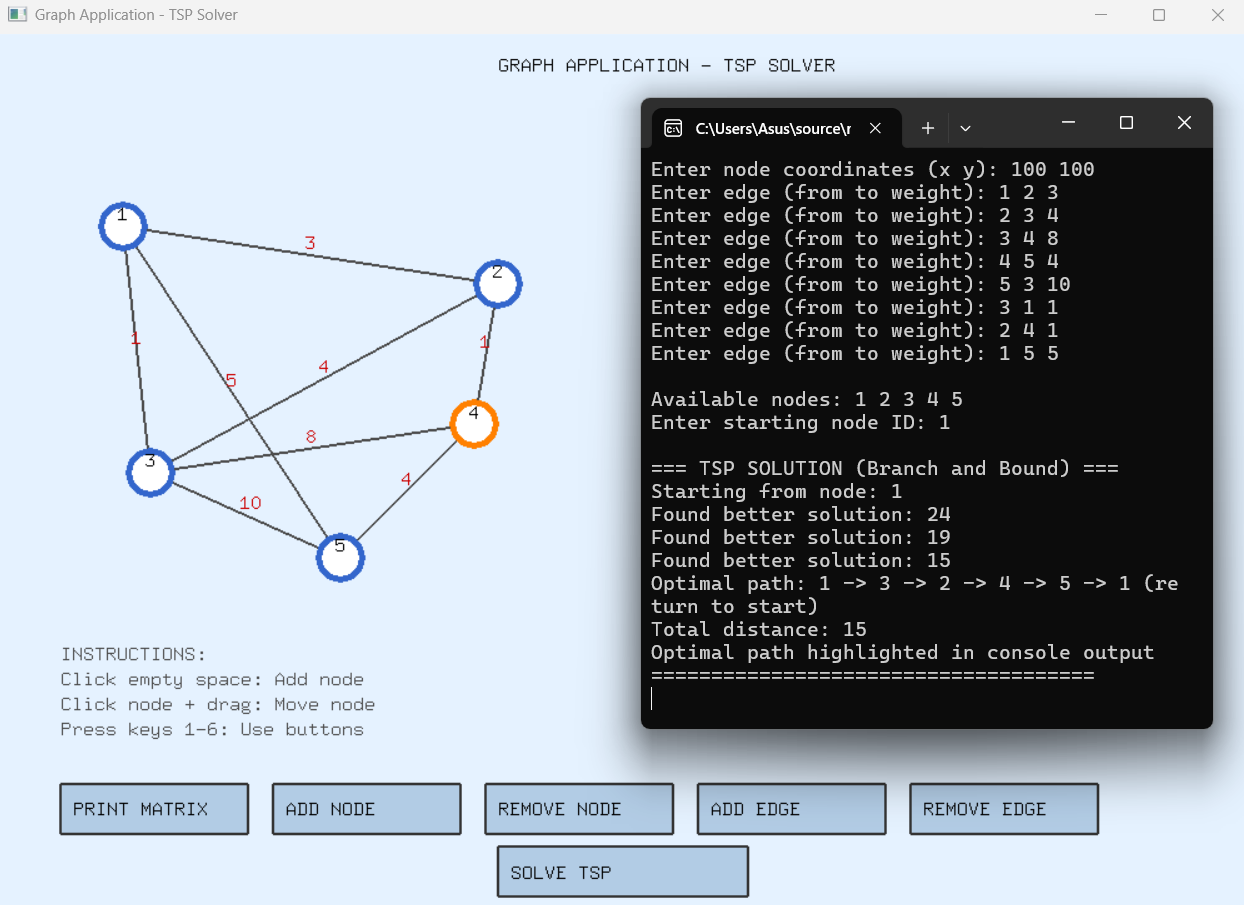
app->initialize(argc, argv);

app->run();

return 0;

}

**Результаты работы программы**

****

**Вывод**

Программа финансового калькулятора представляет собой

простое и удобное приложение, является полезным инструментом,

позволяющим быстро и точно рассчитать основные финансовые показатели.

Вторая программа представляет собой эффективный инструмент для

решения задачи коммивояжера с использованием метода ветвей и границ.

Она позволяет пользователю визуализировать графы, редактировать и

решать TSP.

**Ссылка GitHub**

https://github.com/astidii/PNRPU/tree/main/1%20semester/Creative%20work