Министерство образования Республики Беларусь
Учреждение образования «Белорусский государственный университет информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Операционные среды и системное программирование

ОТЧЁТ
к лабораторной работе №2
на тему

Расширенное использование оконного интерфейса Win 32 и GDI. Формирование сложных изображений, создание и использование элементов управления, обработка различных сообщений, механизм перехвата сообщений (winhook)

Студент: гр.153502
Сачивко В.Г.

Проверил: Гриценко Н.Ю.

Минск 2023

СОДЕРЖАНИЕ

1 ЦЕЛЬ РАБОТЫ

Целью лабораторной работы является создание приложения для визуализации и анализа данных с использованием графиков и диаграмм.

2 ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Расширенное использование оконного интерфейса *Win32* и *GDI* (*Graphics Device Interface*) позволяет создавать более сложные приложения с графическим пользовательским интерфейсом и обработкой различных сообщений.

Win32 API - это набор функций и структур, предоставляемых *Microsoft* для создания *Windows*-приложений. Оконное приложение *Win32* состоит из главного окна (главного окна приложения) и дочерних окон (элементов управления и диалоговых окон). [1]

GDI - это компонент операционной системы *Windows*, отвечающий за рисование графики на экране и на печать. *GDI* предоставляет *API* для создания графических объектов, таких как кисти, шрифты, перья и регионы. Основные операции *GDI* включают рисование линий, текста, фигур, заливку и многие другие.

Обработка сообщений является центральным элементом программы *Win32*. Оконная процедура (*Window Procedure*) обрабатывает сообщения, отправленные окну. Некоторые распространенные сообщения включают *WM_PAINT* (отрисовка окна), *WM_COMMAND* (обработка команд от элементов управления), *WM_KEYDOWN* (обработка нажатий клавиш) и многие другие.

Расширенная графика и изображения могут быть созданы с использованием *GDI+* - более современного интерфейса для работы с графикой в *Windows*. *GDI+* предоставляет возможности для рисования и манипулирования изображениями, включая работу с изображениями в форматах *JPEG*, *PNG* и другими.

Интерфейс пользователя играет важную роль в проектировании приложений. Это включает в себя разработку удобного и интуитивно понятного интерфейса, а также обработку ввода и взаимодействие с пользователем.

Расширенное использование оконного интерфейса *Win32* и *GDI* открывает широкие возможности для создания сложных и интересных *Windows*-приложений с богатым графическим интерфейсом и функциональностью. Важно изучать документацию *Microsoft* и практиковаться в разработке приложений, чтобы успешно использовать эти инструменты и концепции.

3 РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ ПРОГРАММЫ

При запуске программы пользователю предлагается выбрать режим работы приложения (рисунок 1), после чего будет произведено считывание данных для визуализации из определенных заранее файлов. Оно происходит парами строк. Предполагается, что данные поступают вкорректном формате, в противном случае, при возникновении ошибок во время чтения оно будет остановлено.

В режиме работы «график» (рисунок 2) первая строка из пары является координатой x (целое число), вторая – y (целое число). Если приложение открыто в режиме «диаграмма» (рисунок 3), то первая строка является меткой, используемой для легенды, а вторая – непосредственно само значение (целое неотрицательное число).

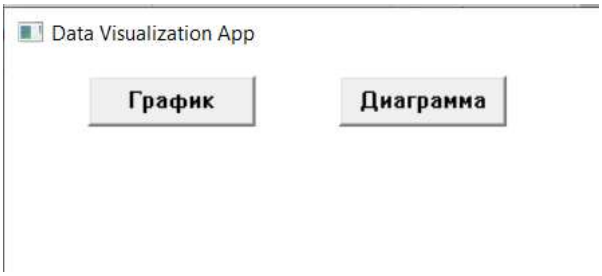

Рисунок 1 – Выбор режима работы

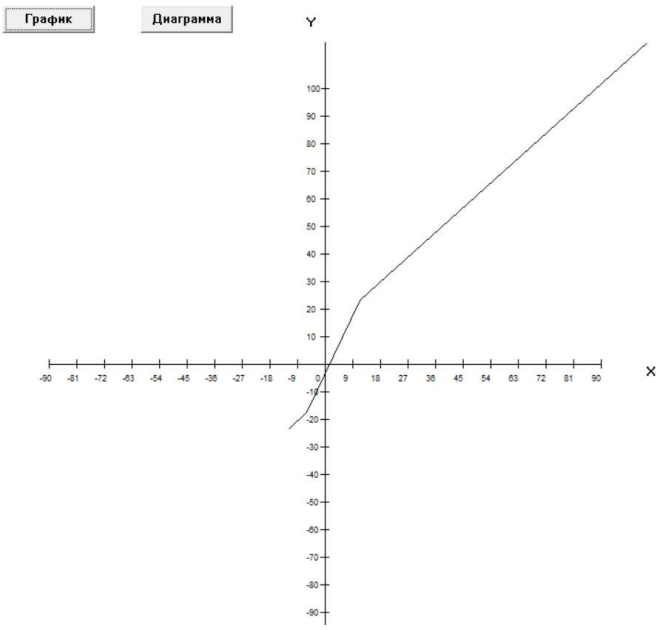

Рисунок 2 – Пример работы программы в режиме «график»

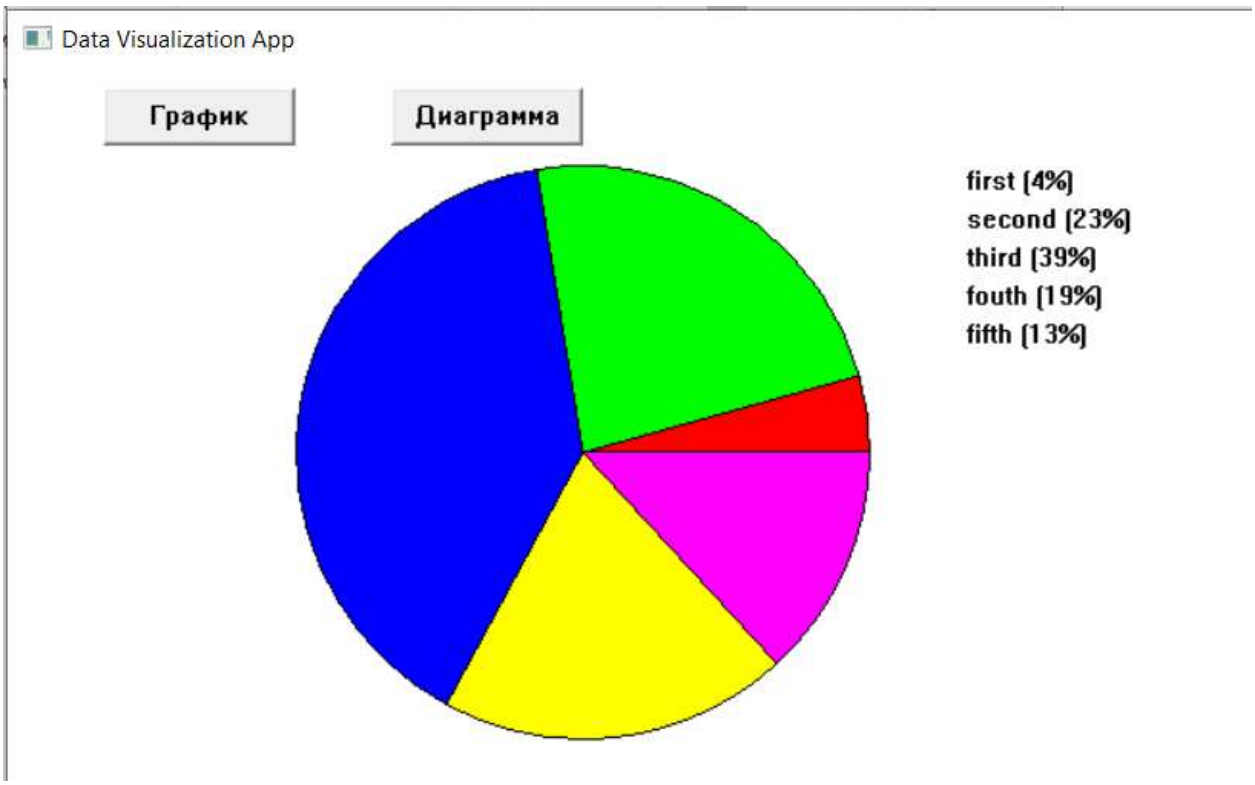

Рисунок 3 – Пример работы программы в режиме «диаграмма»

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Начало работы с классическими приложениями для Windows, которые используют API Win32 [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/windows/win32/desktop-programming

ПРИЛОЖЕНИЕ А
Исходный код программы
Файл Lab2.cpp

```
#define ID_GRAPH 101
#define ID_DIAGRAM 102

#include <Windows.h>
#include <fstream>
#include <vector>
#include <sstream>
#include <string>
#include <algorithm>
#include <map>

struct DataPoint {
    int x;
    int y;
    std::wstring label;
};

struct PieChartData {
    std::string label;
    double value;
};

std::vector<DataPoint> dataPoints;
std::vector<PieChartData> pieChartData;
bool isGraphSelected = false;
bool shouldDraw = false;

LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    switch (uMsg)
    {
    case WM_CLOSE:
        PostQuitMessage(0);
        break;
    case WM_PAINT:
    {
        PAINTSTRUCT ps;
        HDC hdc = BeginPaint(hwnd, &ps);
```

```
RECT clientRect;
GetClientRect(hwnd, &clientRect);
FillRect(hdc, &clientRect, (HBRUSH)(COLOR_WINDOW + 1));

if (shouldDraw)
{
   if (isGraphSelected)
   {
      int width = 700;
      int height = 700;
      int margin = 50;
      int centerX = 400;
      int centerY = 400;

      int minX = INT_MAX;
      int minY = INT_MAX;
      int maxX = INT_MIN;
      int maxY = INT_MIN;

      for (int i = 0; i < dataPoints.size(); i++)
      {
         minX = (dataPoints[i].x < minX) ? dataPoints[i].x : minX;
         minY = (dataPoints[i].y < minY) ? dataPoints[i].y : minY;
         maxX = (dataPoints[i].x > maxX) ? dataPoints[i].x : maxX;
         maxY = (dataPoints[i].y > maxY) ? dataPoints[i].y : maxY;
      }

      // Выравнивание максимальных значений до ближайшего кратного
      10
      int minXAligned = ((minX - 9) / 10) * 10;
      int minYAligned = ((minY - 9) / 10) * 10;
      int maxXAligned = ((maxX + 9) / 10) * 10;
      int maxYAligned = ((maxY + 9) / 10) * 10;

      int numDivisionsX = 21;
      int numDivisionsY = 21;

      MoveToEx(hdc, 2 * margin, centerY, NULL);
      LineTo(hdc, width, centerY);
      MoveToEx(hdc, centerX, margin, NULL);
      LineTo(hdc, centerX, height);

      double stepX = width / (numDivisionsX + 2);
```

```
double stepY = height / (numDivisionsY + 2);

int div = maxXAligned > abs(minXAligned) ? -maxXAligned : minXAligned;
int step = div / 10;
for (int i = 0; i < numDivisionsX; i++)
{
   double x = 2 * margin + i * stepX;
   MoveToEx(hdc, x, centerY - 5, NULL);
   LineTo(hdc, x, centerY + 5);

   std::wstring labelText = std::to_wstring(div);

   HFONT hFont = CreateFont(12, 0, 0, 0, FW_NORMAL, FALSE, FALSE, FALSE, DEFAULT_CHARSET, OUT_OUTLINE_PRECIS, CLIP_DEFAULT_PRECIS, CLEARTYPE_QUALITY, DEFAULT_PITCH | FF_DONTCARE, L"Arial");
   HFONT hPrevFont = (HFONT)SelectObject(hdc, hFont);
   TextOut(hdc, x - 10, centerY + 10, labelText.c_str(), labelText.size());
   SelectObject(hdc, hPrevFont);
   DeleteObject(hFont);

   div += abs(step);
}

div = maxYAligned > abs(minYAligned) ? -maxYAligned : minYAligned;
step = div / 10;
for (int i = 0; i < numDivisionsY; i++)
{
   double y = height - i * stepY;
   MoveToEx(hdc, centerX - 5, y, NULL);
   LineTo(hdc, centerX + 5, y);

   if (div != 0)
   {
      std::wstring labelText = std::to_wstring(div);

      HFONT hFont = CreateFont(12, 0, 0, 0, FW_NORMAL, FALSE, FALSE, FALSE, DEFAULT_CHARSET, OUT_OUTLINE_PRECIS, CLIP_DEFAULT_PRECIS, CLEARTYPE_QUALITY, DEFAULT_PITCH | FF_DONTCARE, L"Arial");
      HFONT hPrevFont = (HFONT)SelectObject(hdc, hFont);
```

```
            TextOut(hdc, centerX - 20, y - 5, labelText.c_str(), labelText.size());
            SelectObject(hdc, hPrevFont);
            DeleteObject(hFont);
        }

        div += abs(step);
    }

    TextOut(hdc, width + margin, centerY, L"X", 1);
    TextOut(hdc, centerX - 20, margin - 30, L"Y", 1);

    // Рисование линии, соединяющей точки данных
    if (!dataPoints.empty())
    {
        double max_X = abs(maxXAligned) > abs(minXAligned) ? abs(maxXAligned) : abs(minXAligned);
        double div_X = (width / 2) / max_X;
        double max_Y = abs(maxYAligned) > abs(minYAligned) ? abs(maxYAligned) : abs(minYAligned);
        double div_Y = (height / 2) / max_Y;

        MoveToEx(hdc, 400.0 + dataPoints[0].x * div_X, 400.0 - dataPoints[0].y * div_Y, NULL);
        for (int i = 1; i < dataPoints.size(); i++)
        {
            double x = 400.0 + dataPoints[i].x * div_X;
            double y = 400.0 - dataPoints[i].y * div_Y;
            LineTo(hdc, x, y);
        }
    }
}
else
{
    int width = 600;
    int height = 400;
    int margin = 50;
    int centerX = width / 2;
    int centerY = height / 2;
    int radius = (width < height ? width : height) / 2 - margin;

    // Расчет суммы значений для вычисления процентов
    double totalValue = 0.0;
    for (const auto& data : pieChartData) {
        totalValue += data.value;
```

```
}

COLORREF pieColors[] = { RGB(255, 0, 0), RGB(0, 255, 0), RGB(0, 0,
255), RGB(255, 255, 0), RGB(255, 0, 255) };
int numColors = sizeof(pieColors) / sizeof(pieColors[0]);
int colorIndex = 0; // Индекс текущего цвета

double startAngle = 0.0;
for (const auto& data : pieChartData)
{
   double sweepAngle = (data.value / totalValue) * 360.0;

   // Рисование сегмента диаграммы с текущим цветом
   HBRUSH brush = CreateSolidBrush(pieColors[colorIndex]);
   SelectObject(hdc, brush);

   // Рисование сегмента диаграммы
    Pie(hdc, centerX - radius, centerY - radius, centerX + radius, centerY
    + radius,
    centerX + static_cast<int>(radius * cos(startAngle *
    3.14159265358979323846 / 180.0)),
    centerY - static_cast<int>(radius * sin(startAngle *
    3.14159265358979323846 / 180.0)),
    centerX + static_cast<int>(radius * cos((startAngle + sweepAngle) *
    3.14159265358979323846 / 180.0)),
    centerY - static_cast<int>(radius * sin((startAngle + sweepAngle) *
    3.14159265358979323846 / 180.0)));

   // Увеличение индекса цвета для следующего сегмента
   colorIndex = (colorIndex + 1) % numColors;

   // Обновление угла для следующего сегмента
   startAngle += sweepAngle;
}

// Отрисовка легенды
int legendX = width - 100;
int legendY = 50;
for (const auto& data : pieChartData)
{
    std::string legendText = data.label + " (" +
    std::to_string(static_cast<int>((data.value / totalValue) * 100.0)) +
    "%)";
   std::wstring widelegendText;
```

```
                widelegendText.assign(legendText.begin(), legendText.end());

                LPCWSTR wideCStrlegendText = widelegendText.c_str();
                TextOut(hdc, legendX, legendY, wideCStrlegendText,
                legendText.size());
                legendY += 20;
            }
        }
    }

    EndPaint(hwnd, &ps);
  }
  break;
  case WM_COMMAND:
    switch (LOWORD(wParam)) {
    case ID_GRAPH:
      isGraphSelected = true;
      shouldDraw = true;
      break;
    case ID_DIAGRAM:
      isGraphSelected = false;
      shouldDraw = true;
      break;
    default:
      break;
    }
    InvalidateRect(hwnd, NULL, TRUE);
    break;
  default:
    return DefWindowProc(hwnd, uMsg, wParam, lParam);
  }
  return 0;
}

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
LPSTR lpCmdLine, int nCmdShow)
{
  // Загрузка данных из текстового файла
  std::ifstream file("graph_data.txt");
  if (!file.is_open()) {
    MessageBox(NULL, L"Не удалось открыть файл graph_data.txt",
L"Ошибка", MB_OK | MB_ICONERROR);
    return 1;
  }
```

```
    std::string line;
    while (std::getline(file, line))
    {
        if (!line.empty())
        {
            std::istringstream iss(line);
            DataPoint point;
            if (iss >> point.x >> point.y)
            {
                char c;
                if (!(iss >> c))
                {
                    dataPoints.push_back(point);
                }
                else
                {
                    MessageBox(NULL, L"Файл graph_data.txt содержит некорректные
                    данные.", L"Ошибка", MB_OK | MB_ICONERROR);
                    return 1;
                }
            }
            else
            {

                MessageBox(NULL, L"Файл graph_data.txt содержит некорректные
                данные.", L"Ошибка", MB_OK | MB_ICONERROR);
                return 1;
            }
        }
    }

    std::ifstream pieChartFile("pie_chart_data.txt");
    if (!pieChartFile.is_open())
    {
        MessageBox(NULL, L"Не удалось открыть файл pie_chart_data.txt",
L"Ошибка", MB_OK | MB_ICONERROR);
        return 1;
    }

    while (std::getline(pieChartFile, line))
    {
        std::istringstream iss(line);
        PieChartData data;
```

```
        std::string label;
        double value;

        if (iss >> label >> value)
        {
            // Проверка, что после считывания строки и числа не остались другие данные
            char c;
            if (!(iss >> c))
            {
                data.label = label;
                data.value = value;
                pieChartData.push_back(data);
            }
            else
            {
                // Если после строки и числа есть другие символы, это не соответствует ожидаемому формату
                MessageBox(NULL, L"Файл pie_chart_data.txt содержит некорректные данные.", L"Ошибка", MB_OK | MB_ICONERROR);
                return 1;
            }
        }
        else
        {
            // Если не удается считать строку и число, это не соответствует ожидаемому формату
            MessageBox(NULL, L"Файл pie_chart_data.txt содержит некорректные данные.", L"Ошибка", MB_OK | MB_ICONERROR);
            return 1;
        }
    }

    WNDCLASS wc = { 0 };
    wc.lpfnWndProc = WindowProc;
    wc.hInstance = hInstance;
    wc.lpszClassName = L"MyDataVisualizationApp";
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);

    RegisterClass(&wc);

    // Создание окна
    HWND hwnd = CreateWindow(L"MyDataVisualizationApp", L"Data Visualization App",
```

```
        WS_OVERLAPPEDWINDOW, 100, 100, 800, 800, NULL, NULL, hInstance,
NULL);

    CreateWindow(L"BUTTON", L"График", WS_VISIBLE | WS_CHILD, 50, 10,
100, 30, hwnd, (HMENU)ID_GRAPH, hInstance, NULL);
    CreateWindow(L"BUTTON",  L"Диаграмма",  WS_VISIBLE  |  WS_CHILD,
200, 10, 100, 30, hwnd, (HMENU)ID_DIAGRAM, hInstance, NULL);

    ShowWindow(hwnd, nCmdShow);

    MSG msg = { 0 };
    while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

    return 0;
}
```