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

ОТЧЕТ

к лабораторной работе № 1
на тему «Основы программирования в Win 32 API. Оконное приложение Win 32 с минимальной достаточной функциональностью. Обработка основных оконных сообщений»

Выполнил:
студент гр. 153504
Подвальников А.С.

Проверил:
Гриценко Н.Ю.

Минск 2023

СОДЕРЖАНИЕ

1 Цели работы

1. Изучить основы программирования в Win 32 API.
2. Создать оконное приложение Win 32 с минимальной достаточной функциональностью.
3. Реализовать обработку основных оконных сообщений.
4. Разработать оконное приложение, которое позволяет пользователю рисовать и редактировать графические фигуры (круги, прямоугольники) с помощью мыши и клавиш клавиатуры.

2 Краткие теоретические сведения

Оконное приложение Win32 API - это приложение Windows, которое использует библиотеки Win32 API для создания и управления окнами и элементами пользовательского интерфейса. Минимально достаточное оконное приложение на Win32 API обычно состоит из окна, которое может быть открыто, закрыто и перерисовано.

Для создания окна в Win32 API, программист должен зарегистрировать класс окна и создать экземпляр этого класса. Зарегистрированный класс содержит информацию о том, как окно должно выглядеть и какие обработчики событий должны быть вызваны для обработки сообщений, отправленных в окно.

Обработка основных оконных сообщений включает в себя обработку сообщений, таких как WM_CREATE, WM_PAINT, WM_COMMAND, WM_RBUTTONDOWN, WM_LBUTTONDOWN, WM_KEYDOWN WM_CLOSE и WM_DESTROY. Сообщение WM_CREATE отправляется системой в окно при создании окна, сообщение WM_PAINT отправляется при необходимости перерисовки окна, сообщение WM_COMMAND отправляется при действиях с элементами управления(кнопки, меню), сообщение WM_RBUTTONDOWN(WM_LBUTTONDOWN) отправляется при нажатии правой(левой) кнопки мыши, сообщение WM_KEYDOWN отправляется при нажатии клавиши на клавиатуре, сообщение WM_CLOSE отправляется, когда пользователь закрывает окно, а сообщение WM_DESTROY отправляется, когда окно должно быть уничтожено.

Реализация обработки этих сообщений в приложении Win32 API обычно осуществляется через обработчики сообщений оконной процедуры, которые определены программистом. Оконная процедура приложения может быть определена как статическая функция в коде приложения, которая будет вызвана при каждом получении сообщения окном.

Также возможно использование других функций Win32 API, таких как CreateWindow, ShowWindow и UpdateWindow, чтобы создавать, отображать и обновлять окна приложения.

Результатом использования Win32 API являются интуитивно понятные и функциональные приложения, которые позволяют пользователям взаимодействовать с компьютером посредством элементов управления, таких как текст, графика, кнопки и поля ввода. Тем не менее, следует учитывать, что в различных контекстах некоторые термины в документации Windows могут иметь разные значения, например, слово "служба" может относиться к вызываемой подпрограмме, драйверу устройства или к обслуживающему процессу

Листинг 1 – Код исходной программы:

```
#include <cmath>
#include <algorithm>
#include <windows.h>
#include "utility.h"

const char *MAIN_WINDOW_CLASS_NAME = "Main Window Class";
constexpr int MOVE_DELTA = 10;
std::vector<std::unique_ptr<Shape>> shapes;
Shape *selected_shape = nullptr;
ShapeType drawing_shape_type = ShapeType::Circle;

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR
lpCmdLine, int nCmdShow)
{
    WNDCLASS wc = {0};
    wc.lpfnWndProc = MainWindowProc;
    wc.hInstance = hInstance;
    wc.lpszClassName = MAIN_WINDOW_CLASS_NAME;
    wc.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
    RegisterClass(&wc);

    HWND hwnd = CreateWindow(
                    MAIN_WINDOW_CLASS_NAME,
                    "Win32 App",
                    WS_OVERLAPPEDWINDOW,
                    CW_USEDEFAULT, CW_USEDEFAULT,
                    CW_USEDEFAULT, CW_USEDEFAULT,
                    NULL, NULL, hInstance, NULL);

    if (hwnd == NULL)
    {
        MessageBox(NULL, "Failed to create main window", "Error", MB_OK |
MB_ICONERROR);
        return 1;
    }
    HMENU hMenu = CreateMenu();
    AppendMenu(hMenu, MF_STRING, 1, "Circle");
    AppendMenu(hMenu, MF_STRING, 2, "Square");
    AppendMenu(hMenu, MF_STRING, 3, "Rectangle");
    SetMenu(hwnd, hMenu);

    ShowWindow(hwnd, nCmdShow);
    UpdateWindow(hwnd);


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

    return msg.wParam;
}

LRESULT CALLBACK MainWindowProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM
lParam)
{
    switch (msg)
    {
    case WM_COMMAND:
        switch (wParam)
```

```
    {
    case 1:
        drawing_shape_type = ShapeType::Circle;
        break;
    case 2:
        drawing_shape_type = ShapeType::Square;
        break;
    case 3:
        drawing_shape_type = ShapeType::Rectangle;
        break;
    }
    return 0;
case WM_PAINT:
{
    PAINTSTRUCT ps;
    HDC hdc = BeginPaint(hwnd, &ps);

    SendMessage(hwnd, WM_ERASEBKGND, (WPARAM)hdc, 0);

    for (const auto &shape : shapes)
    {
        shape->Draw(hdc);
    }
    if (selected_shape != nullptr)
    {
        HPEN hPen = CreatePen(PS_DOT, 1, RGB(0, 0, 255));
        HPEN hOldPen = static_cast<HPEN>(SelectObject(hdc, hPen));
        COLORREF old_bg = SetBkColor(hdc, RGB(255, 255, 255));

        selected_shape->Draw(hdc);

        SelectObject(hdc, hOldPen);
        DeleteObject(hPen);
        SetBkColor(hdc, old_bg);
    }
    EndPaint(hwnd, &ps);
}
return 0;

case WM_RBUTTONDOWN:
case WM_LBUTTONDOWN:
{
    int x = LOWORD(lParam);
    int y = HIWORD(lParam);

    if (wParam == MK_LBUTTON && selected_shape != nullptr)
    {
        selected_shape = SelectShapeAt(x, y);
        if (selected_shape == nullptr)
        {
            UpdateShapes(hwnd);
        }
    }

    OnMouseButtonDown(hwnd, x, y, wParam);
    return 0;
}
case WM_CLOSE:
    DestroyWindow(hwnd);
    return 0;

case WM_DESTROY:
    PostQuitMessage(0);
    return 0;
```

```
    case WM_KEYDOWN:
        OnKeyDown(hwnd, wParam);
        return 0;
    default:
        return DefWindowProc(hwnd, msg, wParam, lParam);
    }
}

void UpdateShapes(HWND hwnd)
{
    InvalidateRect(hwnd, nullptr, TRUE);
    UpdateWindow(hwnd);
}

void InvalidateShape(HWND hwnd, Shape *shape)
{
    if (auto *circle = dynamic_cast<Circle *>(shape))
    {
        RECT rect{circle->x - circle->radius - 1,
                  circle->y - circle->radius - 1,
                  circle->x + circle->radius + 1,
                  circle->y + circle->radius + 1};

        InvalidateRect(hwnd, &rect, TRUE);
    }
    else if (auto *square = dynamic_cast<Square *>(shape))
    {
        RECT rect{square->x - square->side_length / 2 - 1,
                  square->y - square->side_length / 2 - 1,
                  square->x + square->side_length / 2 + 1,
                  square->y + square->side_length / 2 + 1};

        InvalidateRect(hwnd, &rect, TRUE);
    }
}

void OnKeyDown(HWND hwnd, WPARAM wParam)
{
    if (selected_shape == nullptr)
    {
        return;
    }

    InvalidateShape(hwnd, selected_shape);

    switch (wParam)
    {
    case VK_UP:
        selected_shape->y -= MOVE_DELTA;
        break;
    case VK_DOWN:
        selected_shape->y += MOVE_DELTA;
        break;
    case VK_LEFT:
        selected_shape->x -= MOVE_DELTA;
        break;
    case VK_RIGHT:
        selected_shape->x += MOVE_DELTA;
        break;
    case VK_SPACE:
        selected_shape = nullptr;
        break;
    case VK_OEM_PLUS:
```

```
    case VK_ADD:
        if (auto *circle = dynamic_cast<Circle *>(selected_shape))
        {
            circle->radius += MOVE_DELTA;
        }
        else if (auto *square = dynamic_cast<Square *>(selected_shape))
        {
            square->side_length += MOVE_DELTA;
        }
        else if (auto *rectangle = dynamic_cast<RectangleShape
*>(selected_shape))
        {
            rectangle->width += MOVE_DELTA;
            rectangle->height += MOVE_DELTA;
        }
        break;
    case VK_OEM_MINUS:
    case VK_SUBTRACT:
        if (auto *circle = dynamic_cast<Circle *>(selected_shape))
        {
            circle->radius = std::max(circle->radius - MOVE_DELTA, 1);
        }
        else if (auto *square = dynamic_cast<Square *>(selected_shape))
        {
            square->side_length = std::max(square->side_length - MOVE_DELTA,
1);
        }
        else if (auto *rectangle = dynamic_cast<RectangleShape
*>(selected_shape))
        {
            rectangle->width = std::max(rectangle->width - MOVE_DELTA, 1);
            rectangle->height = std::max(rectangle->height - MOVE_DELTA, 1);
        }
        break;
    default:
        return;
    }

    InvalidateShape(hwnd, selected_shape);
    UpdateShapes(hwnd);
}

void OnMouseButtonDown(HWND hWnd, int x, int y, WPARAM wParam)
{
    HDC hdc = GetDC(hWnd);

    if (wParam == MK_LBUTTON)
    {
        selected_shape = SelectShapeAt(x, y);
        if (selected_shape != nullptr)
        {
            UpdateShapes(hWnd);
        }
    }

    int old_value = 0, new_value, startX = x, startY = y;
    MSG msg;
    WPARAM button = wParam;

    while (GetMessage(&msg, nullptr, 0, 0))
    {
        if (msg.message == (button == MK_LBUTTON ? WM_LBUTTONUP :
WM_RBUTTONUP))
        {
```

```
            ReleaseCapture();
            break;
        }
        else if (msg.message == WM_MOUSEMOVE && (msg.wParam == wParam))
        {
            new_value = CalculateSize(LOWORD(msg.lParam), HIWORD(msg.lParam),
startX, startY);
            if (new_value != old_value) old_value = new_value;
        }
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

    std::unique_ptr<Shape> shape;
    switch (drawing_shape_type)
    {
    case ShapeType::Circle:
        shape = std::make_unique<Circle>(startX, startY, old_value);
        break;
    case ShapeType::Square:
        shape = std::make_unique<Square>(startX, startY, old_value);
        break;
    case ShapeType::Rectangle:
        shape = std::make_unique<RectangleShape>(startX, startY, old_value,
old_value);
        break;
    }

    if (shape)
    {
        shapes.push_back(std::move(shape));
    }

    UpdateShapes(hWnd);
    ReleaseDC(hWnd, hdc);
}

Shape* SelectShapeAt(int x, int y)
{
    std::vector<Shape*> sorted_shapes;

    for (const auto &shape : shapes)
    {
        sorted_shapes.push_back(shape.get());
    }

    std::sort(sorted_shapes.begin(), sorted_shapes.end(), [](const Shape *a,
const Shape *b) -> bool
    {
        double area_a = 0.0, area_b = 0.0;
        if (auto *circle = dynamic_cast<const Circle *>(a))
        {
            area_a = M_PI * circle->radius * circle->radius;
        }
        if (auto *circle = dynamic_cast<const Circle *>(b))
        {
            area_b = M_PI * circle->radius * circle->radius;
        }
        return area_a < area_b;
    });

    for (const auto &shape : sorted_shapes)
    {
        if (auto *circle = dynamic_cast<Circle *>(shape))
```

```
        {
            if (std::sqrt((circle->x - x) * (circle->x - x) + (circle->y - y) *
(circle->y - y)) <= circle->radius)
            {
                return shape;
            }
        }
        else if (auto *square = dynamic_cast<Square *>(shape))
        {
            if (std::abs(square->x - x) <= square->side_length / 2 &&
std::abs(square->y - y) <= square->side_length / 2)
            {
                return shape;
            }
        }
        else if (auto *rectangle = dynamic_cast<RectangleShape *>(shape))
        {
            if (std::abs(rectangle->x - x) <= rectangle->width / 2 &&
std::abs(rectangle->y - y) <= rectangle->height / 2)
            {
                return shape;
            }
        }
    }

    return nullptr;
}

int CalculateSize(int x, int y, int mouseX, int mouseY)
{
    int centerX = mouseX;
    int centerY = mouseY;

    int distance = static_cast<int>(std::sqrt((x - centerX) * (x - centerX) +
(y - centerY) * (y - centerY)));

    constexpr int delta = 10;
    return std::max((distance / delta) * delta, 10);
}
```

3 Полученные результаты

Результат работы программы представлен на рисунке 3.1.

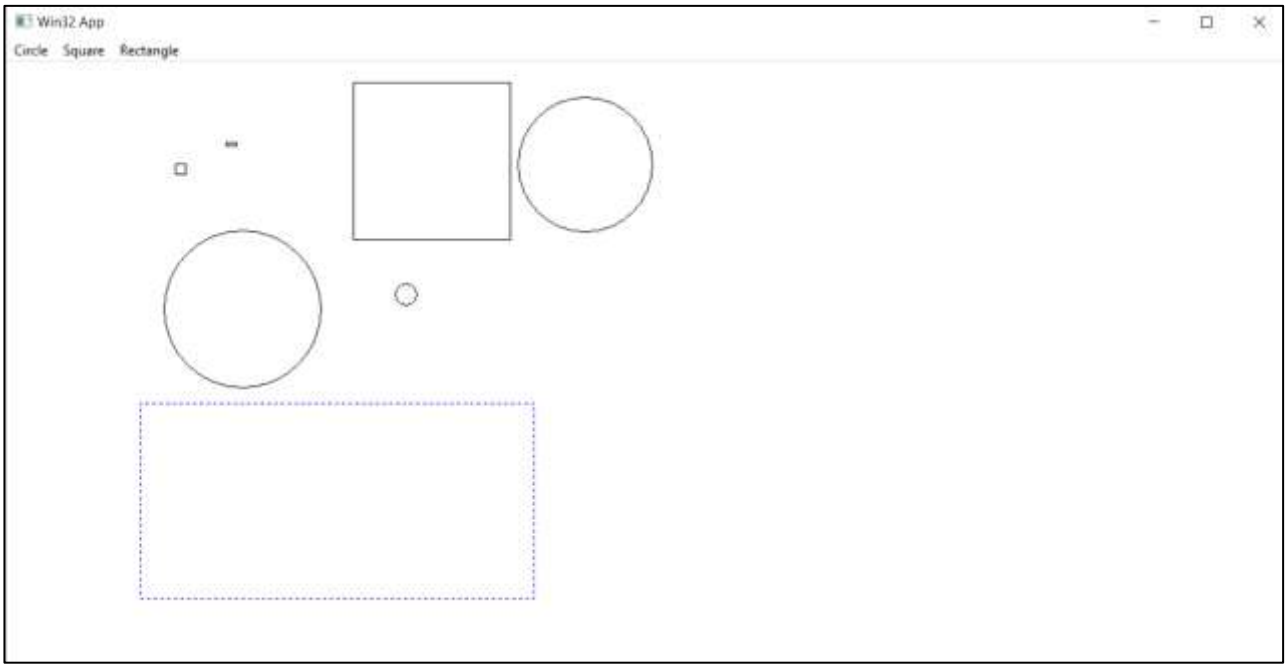

Рисунок 3.1 – Результат работы программы

Выводы

В ходе выполнения лабораторной работы были изучены основы программирования с использованием Win32 API, что позволило создать оконное приложение с необходимой функциональностью. Были применены функции, классы и методы Win32 API, такие как RegisterClass, CreateWindow, GetMessage, DefWindowProc, которые обеспечивают взаимодействие с операционной системой Windows и позволяют эффективно управлять окном и обрабатывать события. В результате было разработано приложение, которое успешно обрабатывает основные оконные сообщения и позволяет пользователю работать с графическими фигурами, такими как круги и прямоугольники, с помощью устройств ввода мыши и клавиатуры.