

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 № 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)»

Выполнил:
студент гр. 153504
Подвальников А.С.

Проверил:
Гриценко Н.Ю.

СОДЕРЖАНИЕ

1 Цели работы	3
2 Краткие теоретические сведения	4
3 Полученные результаты.....	12
Выводы	13

1 Цели работы

1. Изучить основные элементы интерфейса Win 32 и GDI.
2. Изучить механизмы обработки различных сообщений в оконном интерфейсе, включая обработку сообщений мыши и клавиатуры.
3. Изучение механизма перехвата сообщений (winhook).
4. Реализовать графическое приложение для анимации движения объектов с возможностью настройки траектории и скорости.

2 Краткие теоретические сведения

Оконный интерфейс Win32 и графический драйвер интерфейса (GDI) являются основой для разработки приложений под операционные системы Windows. Win32 API - это набор функций и структур, которые обеспечивают доступ к ресурсам операционной системы и позволяют программе работать с графическим пользовательским интерфейсом.

GDI основан на модели устройства с точки зрения рисования, где каждый элемент, например, кнопка или окно, представлен набором объектов GDI. Эти объекты могут быть созданы и изменены с помощью соответствующих функций API. Взаимодействие с GDI может быть осуществлено как в библиотеке DLL, так и непосредственно из окна.

В Win32 API управление элементами интерфейса осуществляется с помощью сообщений. Сообщения - это события, которые происходят в приложении, например, клик по кнопке или перемещение мыши. Они могут быть обработаны с помощью функции оконной процедуры. Для создания элементов управления, таких как кнопки, текстовые поля или ползунки, используются структуры, определенные в библиотеках Win32 API и GDI.

Чтобы обеспечить более гибкое и мощное управление сообщениями, можно использовать механизм перехвата сообщений, такой как WinHook. Этот механизм позволяет отслеживать и перехватывать сообщения, отправляемые любому приложению, и обрабатывать их внутри программы. Это может быть полезно, например, для реализации горячих клавиш или фильтрации входящих сообщений.

В целом, Win32 API и GDI предоставляют разработчикам большое количество инструментов для создания не только простых, но и сложных приложений с графическим интерфейсом. Они позволяют создавать элементы управления, обрабатывать сообщения и создавать сложные изображения, а также использовать механизм перехвата сообщений для гибкого управления взаимодействием пользователей с приложением.

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;
Trajectory trajectory{0, 0};
bool animation_running = false;
int animation_speed = 80;
DWORD last_update = 0;

LRESULT CALLBACK KeyboardProc(int nCode, WPARAM wParam, LPARAM lParam)
{
    if (nCode >= 0 && wParam == WM_KEYDOWN)
    {
        KBDLLHOOKSTRUCT* hookStruct = (KBDLLHOOKSTRUCT*)lParam;
        if (hookStruct->vkCode == VK_F2)
        {
            MessageBox(NULL, "F2 key pressed.", "Info", MB_OK |
MB_ICONINFORMATION);
        }
    }
    return CallNextHookEx(NULL, nCode, wParam, lParam);
}

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,
        "My Win32 Application",
        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");
    AppendMenu(hMenu, MF_STRING, 4, "Trajectory LEFT");
    AppendMenu(hMenu, MF_STRING, 5, "Trajectory RIGHT");
    AppendMenu(hMenu, MF_STRING, 6, "Trajectory UP");
    AppendMenu(hMenu, MF_STRING, 7, "Trajectory DOWN");
```

```

AppendMenu(hMenu, MF_STRING, 8, "Toggle Animation");
AppendMenu(hMenu, MF_STRING, 9, "Reset Trajectory");
SetMenu(hwnd, hMenu);

HHOOK keyboardHook = SetWindowsHookEx(WH_KEYBOARD_LL, KeyboardProc,
hInstance, 0);
if (keyboardHook == NULL)
{
    MessageBox(NULL, "Failed to install keyboard hook.", "Error",
MB_OK | MB_ICONERROR);
    return 1;
}

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

MSG msg = {0};
while (1)
{
    if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
    {
        if (msg.message == WM_QUIT)
        {
            break;
        }

        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    else
    {
        DWORD current_tick = GetTickCount();
        if (animation_running && selected_shape != nullptr &&
current_tick - last_update >= animation_speed)
        {
            last_update = current_tick;
            InvalidateShape(hwnd, selected_shape);
            MoveShape(selected_shape, trajectory);
            InvalidateShape(hwnd, selected_shape);
            UpdateShapes(hwnd);
        }
        Sleep(1); // Sleep for a short duration to reduce CPU load
    }
}

UnhookWindowsHookEx(keyboardHook);

return msg.wParam;
}

LRESULT CALLBACK MainWindowProc(HWND hwnd, UINT msg, WPARAM wParam,
LPARAM lParam)
{
    static DWORD last_update = 0;
    DWORD current_tick = GetTickCount();

    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;
    case 4:
        trajectory.dx = -MOVE_DELTA;
        break;
    case 5:
        trajectory.dx = MOVE_DELTA;
        break;
    case 6:
        trajectory.dy = -MOVE_DELTA;
        break;
    case 7:
        trajectory.dy = MOVE_DELTA;
        break;
    case 8:
        animation_running = !animation_running;
        break;
    case 9:
        trajectory.dx = 0;
        trajectory.dy = 0;
        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);
}

if (animation_running && selected_shape != nullptr && current_tick -
last_update >= animation_speed)
{
    last_update = current_tick;
    InvalidateShape(hwnd, selected_shape);
    MoveShape(selected_shape, trajectory);
    InvalidateShape(hwnd, selected_shape);
    UpdateShapes(hwnd);
}

}

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_LBUTTONDOWN :
WM_RBUTTONDOWN))
        {
            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);
    }

void MoveShape(Shape *shape, const Trajectory &trajectory)
{
    shape->x += trajectory.dx;
    shape->y += trajectory.dy;
}
```

3 Полученные результаты

Результат работы программы показан на рисунках 3.1 и 3.2



Рисунок 3.1 – Результат работы программы(1)

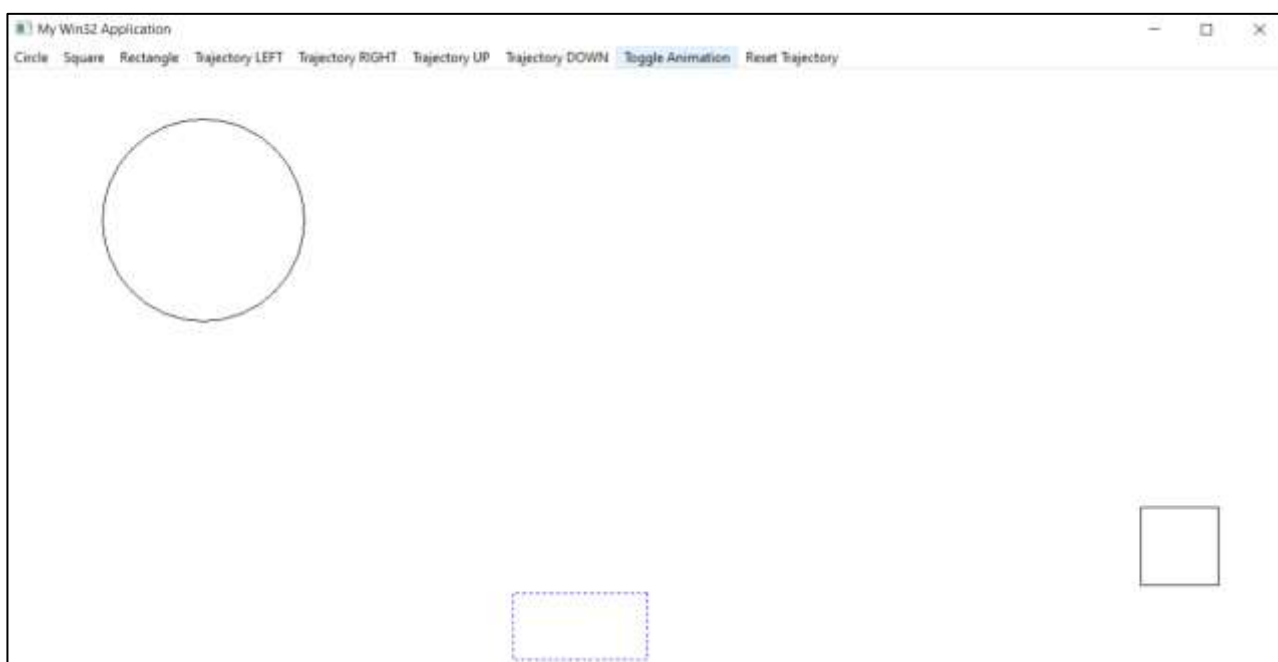


Рисунок 3.2 – Результат работы программы(2)

Выводы

В ходе выполнения данной лабораторной работы были изучены особенности расширенного использования оконного интерфейса Win32 и GDI. Были изучены принципы создания и использования элементов управления. Были также изучены механизмы обработки различных сообщений и механизм перехвата сообщений с использованием Winhook.