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

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Операционные системы и системное программирование

Отчёт
к лабораторной работе
на тему

Основы программирования в Win 32 API. Оконное приложение Win 32 с минимальной достаточной функциональностью. Обработка основных оконных сообщений.

Студент: гр.153502
Александрёнок И.А.

Проверил: Гриценко Н.Ю.

Минск 2023

СОДЕРЖАНИЕ

Цель работы 2
Теоретические сведения 3
Результат выполнения программы 3
Список использованных источников 4
Приложение А 5

1 ЦЕЛЬ РАБОТЫ

Возобновление, закрепление и развитие навыков программирования оконных приложений Windows: структура приложения, цикл обработки сообщений, организация взаимодействия посредством сообщений, создание и использование окон и оконных элементов управления, использование базовых средств графики Windows.

Реализовать оконное приложение, которое позволяет пользователю рисовать и редактировать графические фигуры (круги, прямоугольники) с помощью мыши и клавиш клавиатуры.

2 ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Win32 API (Application Programming Interface) – это набор функций и процедур, предоставляемых операционной системой Windows для разработки приложений на языке программирования C/C++. Оконное приложение Win32 – это приложение, которое состоит из одного или нескольких окон, в которых происходит взаимодействие с пользователем. Для создания окна необходимо зарегистрировать класс окна с помощью функции RegisterClassEx и создать окно с помощью функции CreateWindowEx. Окно может иметь различные свойства, такие как заголовок, размеры, стиль и обработчики сообщений. Важным аспектом программирования в Win32 API является обработка оконных сообщений. Оконные сообщения – это события, которые происходят в окне, например, нажатие кнопки мыши или клавиши, изменение размера окна и другие действия пользователя. Для обработки оконных сообщений необходимо определить функцию оконной процедуры (WndProc), которая будет вызываться системой при возникновении сообщения. В функции WndProc нужно обрабатывать различные типы сообщений с помощью условных операторов и выполнять соответствующие действия.

3 РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ ПРОГРАММЫ

Оконное приложение с простым пользовательским интерфейсом (рисунок 1).

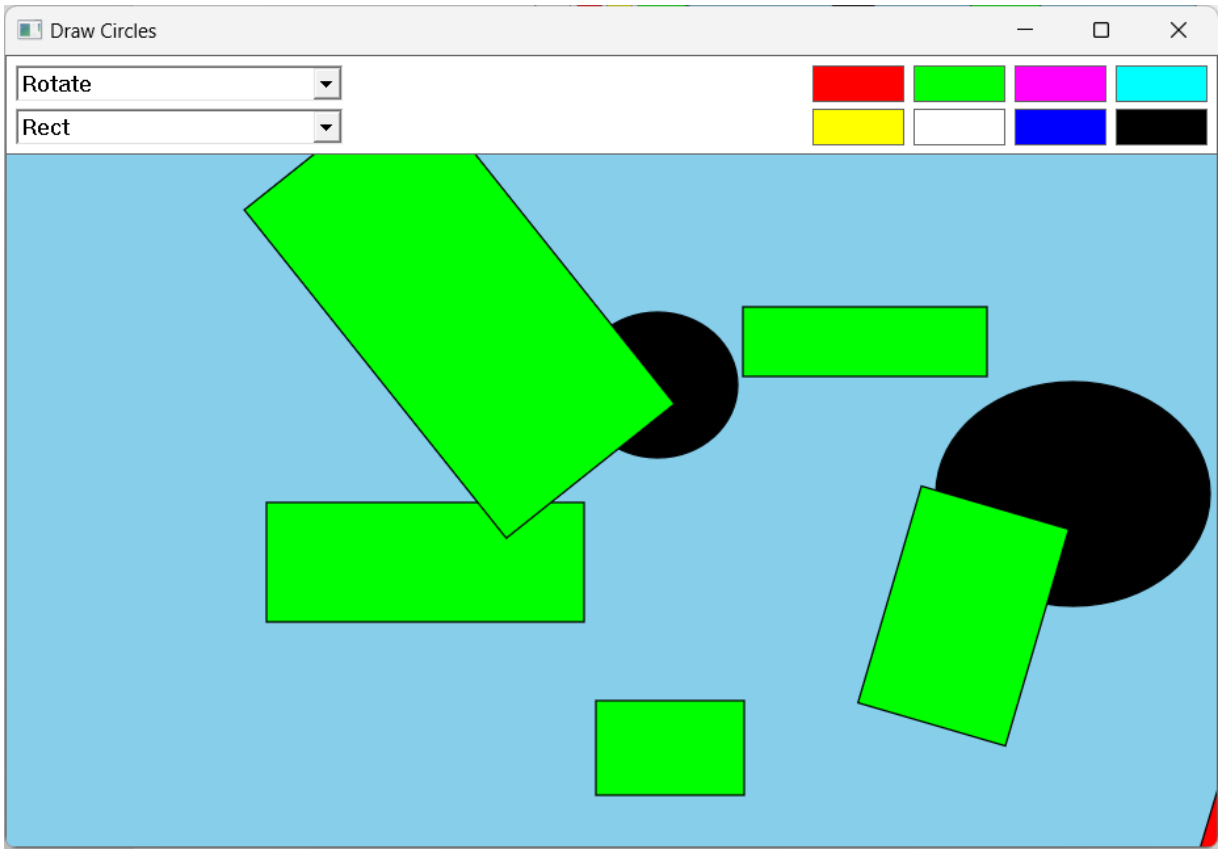

Рисунок 1 – Графический интерфейс программы

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 BaseFigure.h

```
#pragma once

#include <windows.h>
#include <Windowsx.h>
#include <d2d1.h>

class BaseFigure
{
private:
    static const D2D1_COLOR_F DEFAULT_BORDER_COLOR;

public:
    BaseFigure(D2D1_COLOR_F color, D2D1_COLOR_F borderColor = DEFAULT_BORDER_COLOR, D2D1::Matrix3x2F matrix = D2D1::Matrix3x2F::Identity());

    void Translate(D2D1_SIZE_F size);
    void Rotate(FLOAT angle, D2D1_POINT_2F center);
    void Scale(D2D1_SIZE_F size, D2D1_POINT_2F center);
    void RevertTransform();
    void SaveTransform();

    void SetColor(D2D1_COLOR_F color) { this->color = color; }
    void SetBorderColor(D2D1_COLOR_F borderColor) { this->borderColor = borderColor; }
    void SetMatrix(D2D1::Matrix3x2F matrix) { this->matrix = matrix; }

    D2D1_COLOR_F GetColor() { return color; }
    D2D1_COLOR_F GetBorderColor() { return borderColor; }
    D2D1::Matrix3x2F GetMatrix() { return matrix; }

    virtual void Draw(ID2D1RenderTarget* pRT, ID2D1SolidColorBrush* pBrush) = 0;
    virtual D2D1_POINT_2F GetCenter() = 0;
    virtual void PlaceIn(D2D1_RECT_F rect) = 0;
    virtual BOOL HitTest(D2D1_POINT_2F hitPoint) = 0;

protected:
    D2D1_COLOR_F    color;
    D2D1_COLOR_F    borderColor;
    D2D1::Matrix3x2F matrix;
    D2D1::Matrix3x2F lastMatrix;
};
```

Файл BaseFigure.cpp

```
#include "BaseFigure.h"

const D2D1_COLOR_F BaseFigure::DEFAULT_BORDER_COLOR = D2D1::ColorF(D2D1::ColorF::Black);

BaseFigure::BaseFigure(D2D1_COLOR_F color, D2D1_COLOR_F borderColor, D2D1::Matrix3x2F matrix) :
    color(color), borderColor(borderColor), matrix(matrix)
{
```

```
    SaveTransform();
}

void BaseFigure::Translate(D2D1_SIZE_F size)
{
    lastMatrix = matrix;
    matrix = lastMatrix * D2D1::Matrix3x2F::Translation(size);
}

void BaseFigure::Rotate(FLOAT angle, D2D1_POINT_2F center)
{
    matrix = lastMatrix = matrix;
    matrix = lastMatrix * D2D1::Matrix3x2F::Rotation(angle, center);
}

void BaseFigure::Scale(D2D1_SIZE_F size, D2D1_POINT_2F center)
{
    lastMatrix = matrix;
    matrix = lastMatrix * D2D1::Matrix3x2F::Scale(size, center);
}

void BaseFigure::RevertTransform()
{
    matrix = lastMatrix;
}

void BaseFigure::SaveTransform()
{
    lastMatrix = matrix;
}
```

Файл BaseWindow.h

```
#pragma once

template <class DERIVED_TYPE>
class BaseWindow
{
public:
    const PCWSTR CLASS_NAME;

    static LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam,
LPARAM lParam)
    {
        DERIVED_TYPE *pThis = NULL;

        if (uMsg == WM_NCCREATE)
        {
            CREATESTRUCT* pCreate = (CREATESTRUCT*)lParam;
            pThis = (DERIVED_TYPE*)pCreate->lpCreateParams;
            SetWindowLongPtr(hwnd, GWLP_USERDATA, (LONG_PTR)pThis);

            pThis->m_hwnd = hwnd;
        }
        else
        {
            pThis = (DERIVED_TYPE*)GetWindowLongPtr(hwnd, GWLP_USERDATA);
        }
```

```
        if (pThis)
        {
            return pThis->HandleMessage(uMsg, wParam, lParam);
        }
        else
        {
            return DefWindowProc(hwnd, uMsg, wParam, lParam);
        }
    }

    BaseWindow(PCWSTR CLASS_NAME) : CLASS_NAME(CLASS_NAME), m_hwnd(NULL) { }

    BOOL Create(
        PCWSTR lpWindowName,
        DWORD dwStyle,
        HWND hWndParent = 0,
        DWORD dwExStyle = 0,
        int x = CW_USEDEFAULT,
        int y = CW_USEDEFAULT,
        int nWidth = CW_USEDEFAULT,
        int nHeight = CW_USEDEFAULT,
        HMENU hMenu = 0
        )
    {
        WNDCLASS wc = {};

        wc.lpfnWndProc   = DERIVED_TYPE::WindowProc;
        wc.hInstance     = GetModuleHandle(NULL);
        wc.lpszClassName = CLASS_NAME;

        RegisterClass(&wc);

        m_hwnd = CreateWindowEx(
            dwExStyle, CLASS_NAME, lpWindowName, dwStyle, x, y,
            nWidth, nHeight, hWndParent, hMenu, GetModuleHandle(NULL), this
            );

        return (m_hwnd ? TRUE : FALSE);
    }

    HWND Window() const { return m_hwnd; }

protected:
    virtual LRESULT HandleMessage(UINT uMsg, WPARAM wParam, LPARAM lParam) = 0;

    HWND m_hwnd;
};
```

Файл DPIScale.h

```
#pragma once

#include <d2d1.h>

class DPIScale
{
    static float scaleX;
    static float scaleY;
```

```
public:
    static void Initialize()
    {
        FLOAT dpi = GetDpiForSystem();
        scaleX = dpi / 96.0f;
        scaleY = dpi / 96.0f;
    }

    template <typename T>
    static float PixelsToDipsX(T x)
    {
        return static_cast<float>(x) / scaleX;
    }

    template <typename T>
    static float PixelsToDipsY(T y)
    {
        return static_cast<float>(y) / scaleY;
    }

    template <typename T>
    static T DipXToPixels(float x)
    {
        return static_cast<T>(x) * scaleX;
    }

    template <typename T>
    static T DipYToPixels(float y)
    {
        return static_cast<T>(y) * scaleY;
    }
};
```

Файл EllipseFigure.h

```
#pragma once

#include "BaseFigure.h"

class EllipseFigure : public BaseFigure
{
private:
    static const D2D1_COLOR_F DEFAULT_BORDER_COLOR;

public:
    EllipseFigure(D2D1_ELLIPSE ellipse, D2D1_COLOR_F color, D2D1_COLOR_F borderColor = DEFAULT_BORDER_COLOR, D2D1::Matrix3x2F matrix = D2D1::Matrix3x2F::Identity());

    void SetEllipse(D2D1_ELLIPSE ellipse) { this->ellipse = ellipse; }

    D2D1_ELLIPSE GetEllipse() { return ellipse; }

    virtual void Draw(ID2D1RenderTarget* pRT, ID2D1SolidColorBrush* pBrush) override;
    virtual D2D1_POINT_2F GetCenter() override { return matrix.TransformPoint(ellipse.point); }
    virtual void PlaceIn(D2D1_RECT_F rect) override;
    virtual BOOL HitTest(D2D1_POINT_2F hitPoint) override;
```

```
protected:
    D2D1_ELLIPSE ellipse;
};
```

Файл EllipseFigure.cpp

```
#include "EllipseFigure.h"

const D2D1_COLOR_F EllipseFigure::DEFAULT_BORDER_COLOR = D2D1::ColorF(D2D1::ColorF::Black);

EllipseFigure::EllipseFigure(D2D1_ELLIPSE ellipse, D2D1_COLOR_F color, D2D1_COLOR_F borderColor, D2D1::Matrix3x2F matrix) :
    BaseFigure(color, borderColor, matrix), ellipse(ellipse)
{
}

void EllipseFigure::Draw(ID2D1RenderTarget* pRT, ID2D1SolidColorBrush* pBrush)
{
    pRT->SetTransform(matrix);
    pBrush->SetColor(color);
    pRT->FillEllipse(ellipse, pBrush);
    pBrush->SetColor(borderColor);
    pRT->DrawEllipse(ellipse, pBrush, 1.0f);
    pRT->SetTransform(D2D1::Matrix3x2F::Identity());
}

void EllipseFigure::PlaceIn(D2D1_RECT_F rect)
{
    matrix = lastMatrix = D2D1::Matrix3x2F::Identity();
    D2D1_POINT_2F center = D2D1::Point2F((rect.right + rect.left) / 2, (rect.bottom + rect.top) / 2);
    FLOAT radiusX = (rect.right - rect.left) / 2;
    FLOAT radiusY = (rect.bottom - rect.top) / 2;
    ellipse = D2D1::Ellipse(center, radiusX, radiusY);
}

BOOL EllipseFigure::HitTest(D2D1_POINT_2F hitPoint)
{
    D2D1::Matrix3x2F invertedMatrix = matrix;
    invertedMatrix.Invert();
    hitPoint = invertedMatrix.TransformPoint(hitPoint);

    const float a = ellipse.radiusX;
    const float b = ellipse.radiusY;
    const float x1 = hitPoint.x - ellipse.point.x;
    const float y1 = hitPoint.y - ellipse.point.y;
    const float d = ((x1 * x1) / (a * a)) + ((y1 * y1) / (b * b));
    return d <= 1.0f;
}
```

Файл GraphicsScene.h

```
#pragma once

#include <windows.h>
#include <Windowsx.h>
```

```
#include <d2d1.h>

#include <memory>
#include <list>

#include "settings.h"
#include "BaseFigure.h"
#include "EllipseFigure.h"
#include "RectFigure.h"
#include "BaseWindow.h"
#include "DPIScale.h"
#include "resource.h"

class GraphicsScene : public BaseWindow<GraphicsScene>
{
private:
    static const PCWSTR DEFAULT_CLASS_NAME;
    static const D2D1_COLOR_F DEFAULT_BORDER_COLOR;
    static const D2D1_COLOR_F DEFAULT_SELECTION_COLOR;
    static const float DEFAULT_FIGURE_SIZE;

public:
    GraphicsScene(Mode* mode = NULL, Figure* figure = NULL, D2D1_COLOR_F* color =
NULL, ID2D1Factory* pFactory = NULL,
        PCWSTR CLASS_NAME = DEFAULT_CLASS_NAME, D2D1_COLOR_F borderColor =
DEFAULT_BORDER_COLOR,                                    D2D1_COLOR_F
selectionColor=DEFAULT_SELECTION_COLOR);

    std::shared_ptr<BaseFigure> Selection();
    void ClearSelection();
    BOOL Select(D2D1_POINT_2F hitPoint);

    HRESULT CreateGraphicsResources();
    void DiscardGraphicsResources();

    void InsertFigure(float dipX, float dipY);
    void ColorChanged();

    void Resize();
    void OnPaint();
    void OnLButtonDown(int pixelX, int pixelY, DWORD flags);
    void OnLButtonUp();
    void OnMouseMove(int pixelX, int pixelY, DWORD flags);
    void OnKeyDown(UINT vkey);

    virtual LRESULT HandleMessage(UINT uMsg, WPARAM wParam, LPARAM lParam)
override;

protected:
    D2D1_COLOR_F borderColor;
    D2D1_COLOR_F selectionColor;

    ID2D1Factory* pFactory;
    Mode* mode;
    Figure* figure;
    D2D1_COLOR_F* color;
```

```
    ID2D1HwndRenderTarget* pRenderTarget;
    ID2D1SolidColorBrush* pBrush;
    D2D1_POINT_2F ptMouse;

    std::list<std::shared_ptr<BaseFigure>> figures;
    std::list<std::shared_ptr<BaseFigure>>::iterator selection;

    bool tracking;
    TRACKMOUSEEVENT trackingStruct;
};
```

Файл GraphicsScene.cpp

```
#include "GraphicsScene.h"
#include "helper_functions.h"

const PCWSTR GraphicsScene::DEFAULT_CLASS_NAME = L"Graphics";
const D2D1_COLOR_F GraphicsScene::DEFAULT_BORDER_COLOR = D2D1::ColorF(D2D1::ColorF::Black);
const D2D1_COLOR_F GraphicsScene::DEFAULT_SELECTION_COLOR = D2D1::ColorF(D2D1::ColorF::Red);
const float GraphicsScene::DEFAULT_FIGURE_SIZE = 2.0F;

GraphicsScene::GraphicsScene(Mode* mode, Figure* figure, D2D1_COLOR_F* color,
    ID2D1Factory* pFactory, PCWSTR CLASS_NAME, D2D1_COLOR_F borderColor, D2D1_COLOR_F selectionColor) :
    BaseWindow<GraphicsScene>(CLASS_NAME), pFactory(pFactory), mode(mode), figure(figure), color(color),
    borderColor(borderColor), selectionColor(selectionColor), pRenderTarget(NULL), pBrush(NULL), ptMouse(D2D1::Point2F()), selection(figures.end()),
    tracking(false), trackingStruct{ sizeof(trackingStruct), NULL, NULL, NULL }
{
}

std::shared_ptr<BaseFigure> GraphicsScene::Selection()
{
    if (selection == figures.end())
    {
        return nullptr;
    }
    else
    {
        return (*selection);
    }
}

void GraphicsScene::ClearSelection()
{
    if (Selection())
    {
        Selection()->SetBorderColor(DEFAULT_BORDER_COLOR);
    }
    selection = figures.end();
}

BOOL GraphicsScene::Select(D2D1_POINT_2F hitPoint)
{
    ClearSelection();
```

```
    for (auto i = figures.rbegin(); i != figures.rend(); ++i)
    {
        if ((*i)->HitTest(hitPoint))
        {
            selection = (++i).base();
            Selection()->SetBorderColor(DEFAULT_SELECTION_COLOR);
            return TRUE;
        }
    }
    return FALSE;
}

HRESULT GraphicsScene::CreateGraphicsResources()
{
    HRESULT hr = S_OK;
    if (pRenderTarget == NULL)
    {
        RECT rc;
        GetClientRect(m_hwnd, &rc);

        D2D1_SIZE_U size = D2D1::SizeU(rc.right, rc.bottom);

        hr = pFactory->CreateHwndRenderTarget(
            D2D1::RenderTargetProperties(),
            D2D1::HwndRenderTargetProperties(m_hwnd, size),
            &pRenderTarget);

        if (SUCCEEDED(hr))
        {
            const D2D1_COLOR_F color = D2D1::ColorF(1.0f, 1.0f, 0);
            hr = pRenderTarget->CreateSolidColorBrush(color, &pBrush);
        }
    }
    return hr;
}

void GraphicsScene::DiscardGraphicsResources()
{
    SafeRelease(&pRenderTarget);
    SafeRelease(&pBrush);
}

void GraphicsScene::InsertFigure(float dipX, float dipY)
{
    ClearSelection();
    switch (*figure)
    {
    case Figure::Ellipse:
    {
        ptMouse = D2D1::Point2F(dipX, dipY);
        D2D1_ELLIPSE ellipse = D2D1::Ellipse(ptMouse, DEFAULT_FIGURE_SIZE,
DEFAULT_FIGURE_SIZE);
        selection = figures.insert(
            figures.end(),
            std::shared_ptr<BaseFigure>(new EllipseFigure(ellipse, *color,
DEFAULT_SELECTION_COLOR)));
        break;
```

```
        }
        case Figure::Rect:
        {
            ptMouse = D2D1::Point2F(dipX, dipY);
            D2D1_RECT_F rect = D2D1::Rect(ptMouse.x - DEFAULT_FIGURE_SIZE, ptMouse.y - DEFAULT_FIGURE_SIZE, ptMouse.x + DEFAULT_FIGURE_SIZE, ptMouse.y + DEFAULT_FIGURE_SIZE);
            selection = figures.insert(
                figures.end(),
                std::shared_ptr<BaseFigure>(new RectFigure(rect, *color, DEFAULT_SELECTION_COLOR)));
        }
        }
    }

    void GraphicsScene::ColorChanged()
    {
        if ((*mode == Mode::SelectMode) && Selection())
            Selection()->SetColor(*color);
        InvalidateRect(m_hwnd, NULL, FALSE);
    }

    void GraphicsScene::Resize()
    {
        if (pRenderTarget != NULL)
        {
            RECT rc;
            GetClientRect(m_hwnd, &rc);

            D2D1_SIZE_U size = D2D1::SizeU(rc.right, rc.bottom);

            pRenderTarget->Resize(size);

            InvalidateRect(m_hwnd, NULL, FALSE);
        }
    }

    void GraphicsScene::OnPaint()
    {
        HRESULT hr = CreateGraphicsResources();
        if (SUCCEEDED(hr))
        {
            PAINTSTRUCT ps;
            BeginPaint(m_hwnd, &ps);

            pRenderTarget->BeginDraw();

            pRenderTarget->Clear(D2D1::ColorF(D2D1::ColorF::SkyBlue));

            for (auto i = figures.begin(); i != figures.end(); ++i)
            {
                (*i)->Draw(pRenderTarget, pBrush);
            }

            hr = pRenderTarget->EndDraw();
            if (FAILED(hr) || hr == D2DERR_RECREATE_TARGET)
            {
```

```
            DiscardGraphicsResources();
        }
        EndPaint(m_hwnd, &ps);
    }
}

void GraphicsScene::OnLButtonDown(int pixelX, int pixelY, DWORD flags)
{
    const float dipX = DPIScale::PixelsToDipsX(pixelX);
    const float dipY = DPIScale::PixelsToDipsY(pixelY);

    POINT pt = { pixelX, pixelY };
    ptMouse = { dipX, dipY };

    if (DragDetect(m_hwnd, pt))
    {
        SetCapture(m_hwnd);
        switch (*mode)
        {
        case Mode::DrawMode:
            InsertFigure(dipX, dipY);
            break;
        }
    }
    else
    {
        if (*mode == Mode::SelectMode)
        {
            Select(ptMouse);
        }
    }
    InvalidateRect(m_hwnd, NULL, FALSE);
}

void GraphicsScene::OnLButtonUp()
{
    if (Selection())
    {
        Selection()->SaveTransform();
    }
    ReleaseCapture();
}


void GraphicsScene::OnMouseMove(int pixelX, int pixelY, DWORD flags)
{
    const float dipX = DPIScale::PixelsToDipsX(pixelX);
    const float dipY = DPIScale::PixelsToDipsY(pixelY);

    if ((flags & MK_LBUTTON) && Selection())
    {
        switch (*mode)
        {
        case Mode::DrawMode:
        {
            float left;
            float right;
```

```
            float top;
            float bottom;

            if (ptMouse.x > dipX)
            {
                left = dipX;
                right = ptMouse.x;
            }
            else
            {
                left = ptMouse.x;
                right = dipX;
            }

            if (ptMouse.y > dipY)
            {
                top = dipY;
                bottom = ptMouse.y;
            }
            else
            {
                top = ptMouse.y;
                bottom = dipY;
            }

            Selection()->PlaceIn(D2D1::Rect(left, top, right, bottom));
            break;
        }
        case Mode::DragMode:
        {
            Selection()->Translate({ dipX - ptMouse.x, dipY - ptMouse.y });
            ptMouse = { dipX, dipY };
            break;
        }
        {
        case Mode::ScaleMode:
            Selection()->RevertTransform();
            D2D1_POINT_2F center = Selection()->GetCenter();
            D2D1_SIZE_F size = { abs((dipX - center.x) / (ptMouse.x - center.x)), abs((dipY - center.y)
/ (ptMouse.y - center.y)) };
            Selection()->Scale(size, Selection()->GetCenter());
            break;
        }
        case Mode::RotateMode:
        {
            D2D1_POINT_2F center = Selection()->GetCenter();
            FLOAT ax = ptMouse.x - center.x;
            FLOAT ay = ptMouse.y - center.y;
            FLOAT bx = dipX - center.x;
            FLOAT by = dipY - center.y;
            FLOAT aLengthSquare = ax * ax + ay * ay;
            FLOAT bLengthSquare = bx * bx + by * by;
            FLOAT angle = ToDegrees((ax * by - ay * bx) / sqrtf(aLengthSquare * bLengthSquare));
            Selection()->Rotate(angle, Selection()->GetCenter());
            ptMouse = { dipX, dipY };
            break;
        }
```

```
        }
        InvalidateRect(m_hwnd, NULL, FALSE);
    }
}

void GraphicsScene::OnKeyDown(UINT vkey)
{
    switch (vkey)
    {
    case VK_DELETE:
        if ((*mode == Mode::SelectMode) && Selection())
        {
            figures.erase(selection);
            selection = figures.end();
            ClearSelection();
            InvalidateRect(m_hwnd, NULL, FALSE);
        }
        break;
    }
}

LRESULT GraphicsScene::HandleMessage(UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    switch (uMsg)
    {
    case WM_CREATE:
        if (!GetParent(m_hwnd))
        {
            if (FAILED(D2D1CreateFactory(
                D2D1_FACTORY_TYPE_SINGLE_THREADED, &pFactory)))
            {
                return -1;  // Fail CreateWindowEx.
            }
            DPIScale::Initialize();
        }
        trackingStruct.hwndTrack = m_hwnd;
        return 0;

    case WM_DESTROY:
        if (!GetParent(m_hwnd))
        {
            SafeRelease(&pFactory);
            PostQuitMessage(0);
        }
        DiscardGraphicsResources();
        return 0;

    case WM_PAINT:
        OnPaint();
        return 0;

    case WM_SIZE:
        Resize();
        return 0;

    case WM_LBUTTONDOWN:
        OnLButtonDown(GET_X_LPARAM(lParam),                GET_Y_LPARAM(lParam),
```

```
(DWORD)wParam);
            return 0;

        case WM_LBUTTONUP:
            OnLButtonUp();
            return 0;

        case WM_MOUSEMOVE:
            if (tracking)
            {
                OnMouseMove(GET_X_LPARAM(lParam),          GET_Y_LPARAM(lParam),
(DWORD)wParam);
            }
            else
            {
                trackingStruct.dwFlags = TME_HOVER | TME_LEAVE;
                TrackMouseEvent(&trackingStruct);
                tracking = true;
            }
            return 0;

        case WM_MOUSEHOVER:
            SetFocus(m_hwnd);
            trackingStruct.dwFlags = TME_LEAVE;
            TrackMouseEvent(&trackingStruct);
            return 0;

        case WM_MOUSELEAVE:
            tracking = false;
            return 0;

        case WM_KEYDOWN:
            OnKeyDown((UINT)wParam);
            return 0;

        case WM_COMMAND:
            if (~GetKeyState(VK_LBUTTON) & 0x8000)
            {
                HWND parentWND = GetParent(m_hwnd);
                if (!parentWND)
                {
                    parentWND = m_hwnd;
                }

                switch (LOWORD(wParam))
                {
                case ID_DRAW_MODE:
                    *mode = Mode::DrawMode;
                    PostMessage(parentWND, WM_MODE_CHANGED, NULL, NULL);
                    return 0;

                case ID_SELECT_MODE:
                    *mode = Mode::SelectMode;
                    PostMessage(parentWND, WM_MODE_CHANGED, NULL, NULL);
                    return 0;

                case ID_DRAG_MODE:
```

```
            *mode = Mode::DragMode;
            PostMessage(parentWND, WM_MODE_CHANGED, NULL, NULL);
            return 0;

        case ID_SCALE_MODE:
            *mode = Mode::ScaleMode;
            PostMessage(parentWND, WM_MODE_CHANGED, NULL, NULL);
            return 0;

        case ID_ROTATE_MODE:
            *mode = Mode::RotateMode;
            PostMessage(parentWND, WM_MODE_CHANGED, NULL, NULL);
            return 0;


        case ID_ELLIPSE:
            if (*mode == Mode::DrawMode)
            {
                *figure = Figure::Ellipse;
                PostMessage(parentWND, WM_FIGURE_CHANGED, NULL, NULL);
            }
            return 0;

        case ID_RECT:
            if (*mode == Mode::DrawMode)
            {
                *figure = Figure::Rect;
                PostMessage(parentWND, WM_FIGURE_CHANGED, NULL, NULL);
            }
            return 0;
        }
    }
    break;

  case WM_MODE_CHANGED:
    return 0;

  case WM_FIGURE_CHANGED:
    return 0;

  case WM_COLOR_CHANGED:
    ColorChanged();
    return 0;
  }
  return DefWindowProc(m_hwnd, uMsg, wParam, lParam);
}
```

Файл helper_functions.h

```
#pragma once

const double PI = 3.14;

template <class T> void SafeRelease(T** ppT)
{
  if (*ppT)
  {
    (*ppT)->Release();
```

```
        *ppT = NULL;
    }
}

double ToDegrees(double rad);
```

Файл helper_functions.cpp

```
#pragma once

const double PI = 3.14;

template <class T> void SafeRelease(T** ppT)
{
    if (*ppT)
    {
        (*ppT)->Release();
        *ppT = NULL;
    }
}

double ToDegrees(double rad);
```

Файл input.rc

```
//<Snippetinput_rc>
#include "resource.h"

IDR_ACCEL1 ACCELERATORS
{
    0x70,   ID_DRAW_MODE, VIRTKEY            // F1
    0x71,   ID_SELECT_MODE, VIRTKEY          // F2
    0x72,   ID_SCALE_MODE, VIRTKEY
    0X73,   ID_ROTATE_MODE, VIRTKEY
    0x74,   ID_DRAG_MODE, VIRTKEY
}

IDR_ACCEL2 ACCELERATORS
{
    0x75,   ID_ELLIPSE, VIRTKEY
    0x76,   ID_RECT, VIRTKEY
}
//</Snippetinput_rc>
```

Файл main.cpp

```
#include <windows.h>
#include <Windowsx.h>
#include <d2d1.h>

#include <list>
#include <memory>

using namespace std;

#pragma comment(lib, "d2d1")

#include "BaseWindow.h"
#include "EllipseFigure.h"
```

```
#include "resource.h"
#include "GraphicsScene.h"
#include "SceneControl.h"
#include "MainWindow.h"

int WINAPI wWinMain(HINSTANCE hInstance, HINSTANCE, PWSTR, int nCmdShow)
{
    MainWindow win = MainWindow();
    //Mode mode = Mode::SelectMode;
    //Figure figure = Figure::Ellipse;
    //D2D1_COLOR_F color = D2D1::ColorF(D2D1::ColorF::Black);
    //SceneControl win = SceneControl(&mode, &figure, &color);
    //GraphicsScene win = GraphicsScene(&mode, &figure, &color);

    if (!win.Create(L"Draw Circles", WS_OVERLAPPEDWINDOW))
    {
        return 0;
    }

    HACCEL hAccel1 = LoadAccelerators(hInstance, MAKEINTRESOURCE(IDR_ACCEL1));
    HACCEL hAccel2 = LoadAccelerators(hInstance, MAKEINTRESOURCE(IDR_ACCEL2));

    ShowWindow(win.Window(), nCmdShow);

    MSG msg;
    while (GetMessage(&msg, NULL, 0, 0))
    {
        if (!TranslateAccelerator(GetFocus(), hAccel1, &msg) &&
!TranslateAccelerator(GetFocus(), hAccel2, &msg))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    }
    return 0;
}
```

Файл MainWindow.h

```
#pragma once

#include <windows.h>
#include <Windowsx.h>
#include <d2d1.h>

#include "settings.h"
#include "BaseWindow.h"
#include "DPIScale.h"
#include "resource.h"
#include "GraphicsScene.h"
#include "SceneControl.h"

class MainWindow : public BaseWindow<MainWindow>
{
private:
    static const PCWSTR DEFAULT_CLASS_NAME;
    static const Mode DEFAULT_MODE;
    static const Figure DEFAULT_FIGURE;
```

```
    static const D2D1_COLOR_F DEFAULT_COLOR;

public:
    MainWindow(Mode mode = DEFAULT_MODE, Figure figure = DEFAULT_FIGURE, D2D1_COLOR_F color = DEFAULT_COLOR, PCWSTR CLASS_NAME = DEFAULT_CLASS_NAME);
    ~MainWindow();

    Mode* GetMode() { return &mode; }
    Figure* GetFigure() { return &figure; }
    D2D1_COLOR_F* GetColor() { return &color; }
    ID2D1Factory* GetFactory() { return pFactory; }
    HWND GetScene() { return graphicsScene->Window(); }

    virtual LRESULT HandleMessage(UINT uMsg, WPARAM wParam, LPARAM lParam) override;

protected:
    void CreateLayout();
    void SetLayout();

    Mode mode;
    Figure figure;
    D2D1_COLOR_F color;
    ID2D1Factory* pFactory;

    SceneControl* sceneControl;
    GraphicsScene* graphicsScene;
};
```

Файл MainWindow.cpp

```
#include "MainWindow.h"
#include "helper_functions.h"

const PCWSTR MainWindow::DEFAULT_CLASS_NAME = L"Graphics";
const Mode MainWindow::DEFAULT_MODE = Mode::SelectMode;
const Figure MainWindow::DEFAULT_FIGURE = Figure::Ellipse;
const D2D1_COLOR_F MainWindow::DEFAULT_COLOR = D2D1::ColorF(D2D1::ColorF::Black);

MainWindow::MainWindow(Mode mode, Figure figure, D2D1_COLOR_F color, PCWSTR CLASS_NAME) :
    BaseWindow<MainWindow>(CLASS_NAME), mode(mode), figure(figure), color(color), sceneControl(NULL), graphicsScene(NULL)
{
}

MainWindow::~MainWindow()
{
    if (sceneControl)
        delete sceneControl;
    if (graphicsScene)
        delete graphicsScene;
}

LRESULT MainWindow::HandleMessage(UINT uMsg, WPARAM wParam, LPARAM lParam)
{
```

```
        switch (uMsg)
        {
        case WM_CREATE:
            if (FAILED(D2D1CreateFactory(
                D2D1_FACTORY_TYPE_SINGLE_THREADED, &pFactory)))
            {
                return -1;  // Fail CreateWindowEx.
            }
            DPIScale::Initialize();
            CreateLayout();
            return 0;

        case WM_SIZE:
            SetLayout();
            return 0;

        case WM_DESTROY:
            SafeRelease(&pFactory);
            PostQuitMessage(0);
            return 0;

        case WM_MODE_CHANGED:
            PostMessage(sceneControl->Window(), uMsg, wParam, lParam);
            PostMessage(graphicsScene->Window(), uMsg, wParam, lParam);
            return 0;

        case WM_FIGURE_CHANGED:
            PostMessage(sceneControl->Window(), uMsg, wParam, lParam);
            PostMessage(graphicsScene->Window(), uMsg, wParam, lParam);
            return 0;

        case WM_COLOR_CHANGED:
            PostMessage(sceneControl->Window(), uMsg, wParam, lParam);
            PostMessage(graphicsScene->Window(), uMsg, wParam, lParam);
            return 0;
        }
        return DefWindowProc(m_hwnd, uMsg, wParam, lParam);
    }

    void MainWindow::CreateLayout()
    {
        sceneControl = new SceneControl(&mode, &figure, &color);
        sceneControl->Create(L"Scene control", WS_CHILD | WS_BORDER | WS_VISIBLE,
m_hwnd);

        graphicsScene = new GraphicsScene(&mode, &figure, &color, pFactory);
        graphicsScene->Create(L"Scene", WS_CHILD | WS_BORDER | WS_VISIBLE, m_hwnd);
    }

    void MainWindow::SetLayout()
    {
        RECT rcClient;
        GetClientRect(m_hwnd, &rcClient);

        MoveWindow(sceneControl->Window(),
            rcClient.left,
            rcClient.top,
```

```
            rcClient.right,
            sceneControl->GetRealWindowHeight(),
            FALSE);

        RECT rcControl;
        GetClientRect(sceneControl->Window(), &rcControl);

        MoveWindow(graphicsScene->Window(),
            rcClient.left,
            rcControl.bottom,
            rcClient.right,
            rcClient.bottom - rcControl.bottom,
            FALSE);

        InvalidateRect(m_hwnd, NULL, FALSE);
    }
```

Файл RectFigure.h

```
#pragma once

#include "BaseFigure.h"

class RectFigure : public BaseFigure
{
private:
    static const D2D1_COLOR_F DEFAULT_BORDER_COLOR;

public:
    RectFigure(D2D1_RECT_F rect, D2D1_COLOR_F color, D2D1_COLOR_F borderColor =
DEFAULT_BORDER_COLOR, D2D1::Matrix3x2F matrix = D2D1::Matrix3x2F::Identity());

    void SetEllipse(D2D1_RECT_F rect) { this->rect = rect; }

    D2D1_RECT_F GetRect() { return rect; }

    virtual void Draw(ID2D1RenderTarget* pRT, ID2D1SolidColorBrush* pBrush) override;
    virtual D2D1_POINT_2F GetCenter() override { return
matrix.TransformPoint(D2D1::Point2F((rect.right + rect.left) / 2, (rect.top + rect.bottom) / 2)); }
    virtual void PlaceIn(D2D1_RECT_F rect) override;
    virtual BOOL HitTest(D2D1_POINT_2F hitPoint) override;

protected:
    D2D1_RECT_F rect;
};
```

Файл RectFigure.cpp

```
#include "RectFigure.h"

const D2D1_COLOR_F RectFigure::DEFAULT_BORDER_COLOR =
D2D1::ColorF(D2D1::ColorF::Black);

RectFigure::RectFigure(D2D1_RECT_F rect, D2D1_COLOR_F color, D2D1_COLOR_F
borderColor, D2D1::Matrix3x2F matrix) :
    BaseFigure(color, borderColor, matrix), rect(rect)
{
}
```

```
void RectFigure::Draw(ID2D1RenderTarget* pRT, ID2D1SolidColorBrush* pBrush)
{
   pRT->SetTransform(matrix);
   pBrush->SetColor(color);
   pRT->FillRectangle(rect, pBrush);
   pBrush->SetColor(borderColor);
   pRT->DrawRectangle(rect, pBrush, 1.0f);
   pRT->SetTransform(D2D1::Matrix3x2F::Identity());
}

void RectFigure::PlaceIn(D2D1_RECT_F rect)
{
   matrix = lastMatrix = D2D1::Matrix3x2F::Identity();
   this->rect = rect;
}

BOOL RectFigure::HitTest(D2D1_POINT_2F hitPoint)
{
   D2D1::Matrix3x2F invertedMatrix = matrix;
   invertedMatrix.Invert();
   hitPoint = invertedMatrix.TransformPoint(hitPoint);

   if (hitPoint.x > rect.left && hitPoint.x < rect.right && hitPoint.y > rect.top && hitPoint.y <
rect.bottom)
      return true;
   return false;
}
```

Файл resource.h

```
//<SnippetResource_H>
#define IDR_ACCEL1              101
#define ID_DRAW_MODE            40002
#define ID_SELECT_MODE          40003
#define ID_ROTATE_MODE          40004
#define ID_SCALE_MODE           40005
#define ID_DRAG_MODE            40006

#define WM_MODE_CHANGED         WM_USER + 1


#define IDR_ACCEL2              102
#define ID_ELLIPSE              40102
#define ID_RECT                 40103

#define WM_FIGURE_CHANGED       WM_USER + 2

#define WM_COLOR_CHANGED        WM_USER + 3
//</SnippetResource_H>
```

Файл SceneControl.h

```
#pragma once

#include <windows.h>
#include <Windowsx.h>
#include <d2d1.h>
```

```
#include <CommCtrl.h>

#include <unordered_map>

#include "settings.h"
#include "BaseWindow.h"
#include "DPIScale.h"
#include "resource.h"

class SceneControl : public BaseWindow<SceneControl>
{
private:
    static const PCWSTR DEFAULT_CLASS_NAME;
    static const float MARGIN_X;
    static const float MARGIN_Y;
    static const float DEFAULT_PICKER_WIDTH;
    static const float DEFAULT_WINDOW_HEIGHT;
    static const float DEFAULT_BUTTON_WIDTH;
    static const wchar_t* const MODE_NAMES[];
    static const wchar_t* const FIGURE_NAMES[];
    static const COLORREF BUTTON_COLORS[];

public:
    SceneControl(Mode* mode = NULL, Figure* figure = NULL, D2D1_COLOR_F* color =
NULL, PCWSTR CLASS_NAME = DEFAULT_CLASS_NAME, float windowHeight =
DEFAULT_WINDOW_HEIGHT, float buttonWidth = DEFAULT_BUTTON_WIDTH, float pickerWidth
= DEFAULT_PICKER_WIDTH);
    ~SceneControl() { DeleteObject(brush); }

    HRESULT CreateGraphicsResources();
    void DiscardGraphicsResources();

    float GetWindowHeight() { return windowHeight; }
    int GetRealWindowHeight();

    virtual LRESULT HandleMessage(UINT uMsg, WPARAM wParam, LPARAM lParam)
override;

protected:
    void CreateLayout();
    void SetLayout();

    Mode* mode;
    Figure* figure;
    D2D1_COLOR_F* color;

    float windowHeight;
    float buttonWidth;
    float pickerWidth;

    std::unordered_map<HWND, COLORREF> buttons;

    HBRUSH brush;

    HWND modePicker;
    HWND figurePicker;
```

```
    bool tracking;
    TRACKMOUSEEVENT trackingStruct;
};
```

Файл SceneControl.cpp

```
#include "SceneControl.h"
#include "helper_functions.h"

const PCWSTR SceneControl::DEFAULT_CLASS_NAME = L"Graphics";
const float SceneControl::MARGIN_X = 6.0F;
const float SceneControl::MARGIN_Y = 6.0F;
const float SceneControl::DEFAULT_PICKER_WIDTH = 200.0F;
const float SceneControl::DEFAULT_WINDOW_HEIGHT = 60.0F;
const float SceneControl::DEFAULT_BUTTON_WIDTH = 70.0F;

const wchar_t* const SceneControl::MODE_NAMES[] = { // must follow the same order as enum
    L"Draw",
    L"Select",
    L"Drag",
    L"Scale",
    L"Rotate"
};
const wchar_t* const SceneControl::FIGURE_NAMES[] = { // must follow the same order as enum
    L"Ellipse",
    L"Rect"
};

const COLORREF SceneControl::BUTTON_COLORS[] = {
    0x00000000,
    0x00FF0000,
    0x0000FF00,
    0x000000FF,
    0x00FFFF00,
    0x0000FFFF,
    0x00FF00FF,
    0x00FFFFFF
};

SceneControl::SceneControl(Mode* mode, Figure* figure, D2D1_COLOR_F* color, PCWSTR
CLASS_NAME, float windowHeight, float buttonWidth, float pickerWidth) :
    BaseWindow<SceneControl>(CLASS_NAME), mode(mode), figure(figure), color(color),
modePicker(NULL), figurePicker(NULL), windowHeight(windowHeight), buttonWidth(buttonWidth),
pickerWidth(pickerWidth), buttons(), brush(NULL), tracking(false), trackingStruct{
sizeof(trackingStruct), NULL, NULL, NULL }
{
}

LRESULT SceneControl::HandleMessage(UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    switch (uMsg)
    {
    case WM_CREATE:
    {
        if (!GetParent(m_hwnd))
        {
            DPIScale::Initialize();
        }
```

```
            CreateLayout();

            RECT rcWindow;
            GetWindowRect(m_hwnd, &rcWindow);
            MoveWindow(m_hwnd,
                rcWindow.left,
                rcWindow.top,
                rcWindow.right - rcWindow.left,
                GetRealWindowHeight(),
                TRUE);

            trackingStruct.hwndTrack = m_hwnd;

            return 0;
        }

        case WM_SIZE:
            SetLayout();
            return 0;

        case WM_SIZING:
        {
            RECT* dragRc = (RECT*)lParam;

            if (wParam == WMSZ_BOTTOM || wParam == WMSZ_BOTTOMLEFT || wParam ==
WMSZ_BOTTOMRIGHT)
            {
                dragRc->bottom = dragRc->top + GetRealWindowHeight();
            }
            else if (wParam == WMSZ_TOP || wParam == WMSZ_TOPLEFT || wParam ==
WMSZ_TOPRIGHT)
            {
                dragRc->top = dragRc->bottom - GetRealWindowHeight();
            }
            return TRUE;
        }

        case WM_PAINT:
        {
            PAINTSTRUCT ps;
            HDC hdc = BeginPaint(m_hwnd, &ps);
            FillRect(hdc, &ps.rcPaint, (HBRUSH)(COLOR_WINDOW + 1));
            EndPaint(m_hwnd, &ps);
            return 0;
        }

        case WM_MOUSEMOVE:
            if (!tracking)
            {
                trackingStruct.dwFlags = TME_HOVER | TME_LEAVE;
                TrackMouseEvent(&trackingStruct);
                tracking = true;
            }
            return 0;

        case WM_MOUSEHOVER:
            SetFocus(m_hwnd);
```

```
            trackingStruct.dwFlags = TME_LEAVE;
            TrackMouseEvent(&trackingStruct);
            return 0;

        case WM_MOUSELEAVE:
            tracking = false;
            return 0;

        case WM_COMMAND:
            if (HIWORD(wParam) == CBN_SELCHANGE)
            {
                HWND parentWND = GetParent(m_hwnd);
                if (!parentWND)
                {
                    parentWND = m_hwnd;
                }

                int ItemIndex = SendMessage((HWND)lParam, (UINT)CB_GETCURSEL, (WPARAM)0,
(LPARAM)0);

                if ((HWND)lParam == modePicker)
                {
                    *mode = (Mode)ItemIndex;
                    PostMessage(parentWND, WM_MODE_CHANGED, NULL, NULL);
                    return 0;
                }
                else if ((HWND)lParam == figurePicker)
                {
                    *figure = (Figure)ItemIndex;
                    PostMessage(parentWND, WM_FIGURE_CHANGED, NULL, NULL);
                    return 0;
                }
            }
            else if (HIWORD(wParam) == BN_CLICKED)
            {
                HWND parentWND = GetParent(m_hwnd);
                if (!parentWND)
                {
                    parentWND = m_hwnd;
                }
                COLORREF colorRef = buttons[(HWND)lParam];
                *color = D2D1::ColorF(GetRValue(colorRef), GetGValue(colorRef),
GetBValue(colorRef));
                PostMessage(parentWND, WM_COLOR_CHANGED, NULL, NULL);

                return 0;
            }
            break;

        case WM_CTLCOLORBTN:
        {
            DeleteObject(brush);
            brush = CreateSolidBrush(buttons[(HWND)lParam]);
            return (LRESULT)brush;
        }

        case WM_MODE_CHANGED:
```

```
            SendMessage(modePicker, CB_SETCURSEL, (WPARAM)*mode, 0);
            return 0;

        case WM_FIGURE_CHANGED:
            SendMessage(figurePicker, CB_SETCURSEL, (WPARAM)*figure, 0);
            return 0;

        case WM_COLOR_CHANGED:
            return 0;
        }

        return DefWindowProc(m_hwnd, uMsg, wParam, lParam);
    }

    int SceneControl::GetRealWindowHeight()
    {
        RECT adjustedWindowHeightRect = { 0, 0, 0, DPIScale::DipYToPixels<int>(windowHeight) };
        AdjustWindowRect(&adjustedWindowHeightRect, GetWindowStyle(m_hwnd),
(BOOL)GetMenu(m_hwnd));
        return adjustedWindowHeightRect.bottom - adjustedWindowHeightRect.top;
    }

    void SceneControl::CreateLayout()
    {
        modePicker = CreateWindowEx(0,
            WC_COMBOBOX,
            NULL,
            CBS_DROPDOWNLIST | CBS_HASSTRINGS | WS_CHILD | WS_OVERLAPPED |
WS_VISIBLE,
            0, 0, 0, 0,
            m_hwnd,
            NULL,
            GetModuleHandle(NULL),
            NULL);

        figurePicker = CreateWindowEx(0,
            WC_COMBOBOX,
            NULL,
            CBS_DROPDOWNLIST | CBS_HASSTRINGS | WS_CHILD | WS_OVERLAPPED |
WS_VISIBLE,
            0, 0, 0, 0,
            m_hwnd,
            NULL,
            GetModuleHandle(NULL),
            NULL);

        for (auto& i : MODE_NAMES)
        {
            SendMessage(modePicker, CB_ADDSTRING, 0, (LPARAM)i);
        }
        SendMessage(modePicker, CB_SETCURSEL, (WPARAM)*mode, 0);

        for (auto& i : FIGURE_NAMES)
        {
            SendMessage(figurePicker, CB_ADDSTRING, 0, (LPARAM)i);
        }
        SendMessage(figurePicker, CB_SETCURSEL, (WPARAM)*figure, 0);
```

```
        for (auto& i : BUTTON_COLORS)
        {
            buttons[CreateWindow(L"BUTTON",
                NULL,
                WS_TABSTOP | WS_VISIBLE | WS_CHILD | BS_DEFPUSHBUTTON |
BS_OWNERDRAW | WS_BORDER,
                0,
                0,
                0,
                0,
                m_hwnd,
                NULL,
                GetModuleHandle(NULL),
                NULL)] = i;
        }
    }

    void SceneControl::SetLayout()
    {
        RECT rcClient;
        GetClientRect(m_hwnd, &rcClient);

        int MARGIN_XPix = DPIScale::DipXToPixels<int>(MARGIN_X);
        int MARGIN_YPix = DPIScale::DipYToPixels<int>(MARGIN_Y);
        int pickerWidthPix = DPIScale::DipXToPixels<int>(pickerWidth);

        MoveWindow(modePicker,
            MARGIN_XPix,
            MARGIN_YPix,
            pickerWidthPix,
            1000,
            FALSE);

        RECT rcMode;
        GetClientRect(modePicker, &rcMode);

        MoveWindow(figurePicker,
            MARGIN_XPix,
            rcClient.bottom - MARGIN_YPix - rcMode.bottom,
            pickerWidthPix,
            1000,
            FALSE);

        int i = 0;
        for (auto& it : buttons)
        {
            MoveWindow(it.first,
                rcClient.right - (MARGIN_XPix + buttonWidth) * (1 + i / 2),
                MARGIN_YPix * (1 - i % 2) + (rcClient.bottom - MARGIN_YPix - rcMode.bottom) * (i %
2),
                buttonWidth,
                rcMode.bottom,
                FALSE);
            ++i;
        }
```

```
    InvalidateRect(m_hwnd, NULL, FALSE);
}
```

Файл settings.h

```
#pragma once

enum class Mode
{
    DrawMode,
    SelectMode,
    DragMode,
    ScaleMode,
    RotateMode
};

enum class Figure
{
    Ellipse,
    Rect
};
```