

Windows API Sample Task

Paweł Aszklar

pawel.aszklar@pw.edu.pl

Warszawa, February 28, 2025

1 Introduction

The purpose of this workshop is to demonstrate a sample task similar to what you might encounter during in-class graded laboratories. In section 2 we will introduce the requirements. The following section 3 will guide you step-by-step through the solution, gradually introducing various parts of Windows API. Please note the guide, for the sake of brevity, omits most error-checking and might not always properly release all resources. In your own code, however, should always take time to add those parts back.

The lecture examples available [here](#) also include a solution to this task, but here we'll take a slightly different approach, putting majority of the logic into one application class instead of defining separate window classes. Please compare the two implementations later at home.

Topics we'd like to cover, that didn't fit into the described task are added in section 4 as extensions to our program. This also takes form of a step-by-step guide. Don't worry if you don't get to it during the class, but make sure to finish it and the next section at home.

Finally, section 5 will extended the set of requirements akin to a graded task done at home. You will mostly have to try to solve it on your own.

2 Laboratory Task

The task is to implement an program that can be a basis for a clone of the game 2048. General requirements are listed below. There is also and executable **2048.exe** provided with this task demonstrates the intended functionality. In cases unspecified by the description you must match the behaviour of that program.

General Requirements You must not use any GDI drawing functions in your implementation. You must also remember about proper resource management — freeing/destroying any acquired objects that are no longer needed.

Layout Requirements

- Two windows with the same content,
- Only one of the windows should show up on the taskbar,
- Size of the windows dependent on the board content and unchanging,
- Title set to "2048",
- The taskbar icon and the caption bar icon of at least one of the windows uses `2048_icon.ico` provided with the description,
- Window background set to (250, 247, 238),
- In each window a board of 4×4 tiles,
- Tile background set to (250, 192, 174), size of 60px and a margin of 10px in-between tiles and around the window border.

Application Behaviour

- Moving one window should cause the position of the other to change symmetrically with respect to the centre of the screen – the centre of a window should be considered as its position.
- When two windows overlap, one of them (you can choose which) should turn semitransparent.

Hints Consider the following functions, messages and window attributes when implementing your program:

- `CreateSolidBrush`
- `AdjustWindowRectEx`
- `WM_WINDOWPOSCHANGED`
- `WM_CTLCOLORSTATIC`
- `DwmGetWindowAttribute`
- `DWMWA_EXTENDED_FRAME_BOUNDS`
- `SetWindowLongPtrW`
- `GetSystemMetrics`
- `SetWindowPos`

3 Solution

3.1 Creating a Project

First we need to create a Visual Studio project that we will use to implement our solution.

1. Start Visual Studio and click *Create a new project*.
2. Select `Windows Desktop Wizard` template and name it something.
3. In the wizard window select `Desktop Application (.exe)` from the drop down under `Application type` and check `Empty project` in `Additional options`.
4. Once the project is created, add a new source file `main.cpp`. Initially you should only add an empty `wWinMain`:

```
#include <windows.h>

int WINAPI wWinMain(HINSTANCE instance,
                    HINSTANCE /*prevInstance*/,
                    LPWSTR /*command_line*/,
                    int show_command)
{ }
```

5. Lastly we need to adjust project configuration. In the project properties, under `Configuration Properties` `General` set the *C++ Language Standard* to at least *ISO C++20 Standard (/std:c++20)*.
6. Under `Configuration Properties` `C/C++` `Preprocessor` expand the drop-down for `Preprocessor Definitions`, select `<Edit...>` and add the following:

```
WIN32_LEAN_AND_MEAN
NOMINMAX
```

First one excludes rarely used parts in `<windows.h>` header. The second disables `min` and `max` macros which tend to mess with C++ standard library.

The project should now compile and run, but the program immediately exits.

3.2 Application Class, First Window and a Message Loop

Let us first create a custom class that will be responsible for running the message loop of our program. It will store all the necessary information, for now: application instance handle (`HINSTANCE`), main window handle (`HWND`), and the window class name.

For the window procedure we will use an approach similar to the one presented in the lecture, where a static window procedure forwards messages to the non-static variant on our custom class instance. The windows will not be fully independent, however, and they will need to cooperate when it comes to movement. Therefore we will not implement the window procedure in a separate *window* class. Instead we'll make the application class handle the messages as well, since it will have all the necessary information.

The class will also include implementation of window class registration and creation of windows.

1. Add **app_2048.h** header file and **app_2048.cpp** source file to the project.
2. In the header file add the definition of **app_2048** class that will represent our application

```
#pragma once
#include <windows.h>
#include <string>
class app_2048
{
private:
    bool register_class();
    static std::wstring const s_class_name;
    static LRESULT CALLBACK window_proc_static(
        HWND window, UINT message,
        WPARAM wparam, LPARAM lparam);
    LRESULT window_proc(
        HWND window, UINT message,
        WPARAM wparam, LPARAM lparam);
    HWND create_window();
    HINSTANCE m_instance;
    HWND m_main;
public:
    app_2048(HINSTANCE instance);
    int run(int show_command);
};
```

3. In the source file we need to initialize the window class name field:

```
#include "app_2048.h"
#include <stdexcept>

std::wstring const app_2048::s_class_name{ L"2048 Window" };
```

4. The lecture and the [documentation](#) tell us that before any window is created one must register its class, using e.g. **RegisterClassExW**,

passing an address of a structure containing the parameters. For now we will only set the absolute minimum, mainly: window procedure address, application instance handle, cursor, and the class name. The documentation page also mentions `GetClassInfoExW` which we can use to see if the window class has not been registered yet.

```
bool app_2048::register_class() {
    WNDCLASSEXW desc{};
    if (GetClassInfoExW(m_instance, s_class_name.c_str(),
                        &desc) != 0) return true;
    desc = { .cbSize = sizeof(WNDCLASSEXW),
            .lpfnWndProc = window_proc_static,
            .hInstance = m_instance,
            .hCursor = LoadCursorW(nullptr, L"IDC_ARROW"),
            .lpszClassName = s_class_name.c_str() };
    return RegisterClassExW(&desc) != 0;
}
```

5. Referring again to the lecture and the [documentation](#), windows can be created using `CreateWindowExW`. We can leave most parameters with their usual defaults for now, but we have to pay attention to the styles. Reading carefully through the [list](#) of available options to match the requirements for our main window we need a top-level window (`WS_OVERLAPPED`) with a caption bar and a non-sizing border (`WS_CAPTION`), an icon for the system menu and a close button (`WS_SYSMENU`). We can include the minimize button (`WS_MINIMIZEBOX`), but we should avoid the sizing border (`WS_SIZEBOX`) and the maximize button (`WS_MAXIMIZEBOX`). Lastly similarly to the example in the lecture, we will use the last parameter to pass the address of our application class object.

```
HWND app_2048::create_window()
{
    return CreateWindowExW(
        0 /*empty extended styles*/,
        s_class_name.c_str(),
        L"2048",
        WS_OVERLAPPED | WS_SYSMENU | WS_CAPTION |
        WS_BORDER | WS_MINIMIZEBOX,
        CW_USEDEFAULT, 0, /*default position*/
        CW_USEDEFAULT, 0, /*default size*/
        nullptr,
        nullptr,
        m_instance,
        this);
}
```

6. Our static window procedure should try to extract the address of the application instance and forward the message to a non-static version. In the [messages](#) arriving during window creation, the address can be taken from [CREATESTRUCTW](#) which address is passed as [lparam](#) for [WM_NCCREATE](#) and [WM_CREATE](#). The address can be attached to the window handle via [SetWindowLongPtrW](#) as the hints for this task helpfully suggest.

Assuming the address was stored there during creation, it can be later retrieved for all other messages arriving for a given window.

If the application instance address cannot be located, messages can be safely passed to the default window procedure.

```
LRESULT app_2048::window_proc_static(
    HWND window,
    UINT message,
    WPARAM wparam,
    LPARAM lparam)
{
    app_2048 *app = nullptr;
    if (message == WM_NCCREATE)
    {
        auto p = reinterpret_cast<LPCREATESTRUCTW>(lparam);
        app = static_cast<app_2048 *>(p->lpCreateParams);
        SetWindowLongPtrW(window, GWLP_USERDATA,
            reinterpret_cast<LONG_PTR>(app));
    }
    else
    {
        app = reinterpret_cast<app_2048 *>(
            GetWindowLongPtrW(window, GWLP_USERDATA));
    }
    if (app != nullptr)
    {
        return app->window_proc(window, message,
                                wparam, lparam);
    }
    return DefWindowProcW(window, message, wparam, lparam);
}
```

7. The non-static window procedure is where we will put all message handling. For now we should ensure we can at least close our program. Checking the [documentation](#) on window destruction we can see that in response to a window being closed we need to destroy it manually. And for the [message loop](#) to end when main window is destroyed, we must call [PostQuitMessage](#).

```

LRESULT app_2048::window_proc(
    HWND window, UINT message,
    WPARAM wparam, LPARAM lparam)
{
    switch (message) {
        case WM_CLOSE:
            DestroyWindow(window);
            return 0;
        case WM_DESTROY:
            if (window == m_main)
                PostQuitMessage(EXIT_SUCCESS);
            return 0;
    }
    return DefWindowProcW(window, message, wparam, lparam);
}

```

8. In the constructor of the application class we should ensure our window class is registered and the main window created.

```

app_2048::app_2048(HINSTANCE instance)
    : m_instance{ instance }, m_main{}
{
    register_class()
    m_main = create_window();
}

```

9. The `run` method should make the window visible and implement the message loop.

```

int app_2048::run(int show_command)
{
    ShowWindow(m_main, show_command);
    MSG msg{};
    BOOL result = TRUE;
    while ((result = GetMessageW(&msg, nullptr, 0, 0)) != 0)
    {
        if (result == -1)
            return EXIT_FAILURE;
        TranslateMessage(&msg);
        DispatchMessageW(&msg);
    }
    return EXIT_SUCCESS;
}

```

10. Finally, back in `wWinMain` we can create our application instance and run it.

```
app_2048 app{ instance };  
return app.run(show_command);
```

We should now be able to run our program and see a single non-resizable window. While this section was rather long, with no way to test much along the way, it implements the well-known basic steps of any Windows API application. It will serve as a base-line for more incremental steps to add individual features required in the task, but can also be reused later for other projects.

3.3 Second Window

Next step will be to add another window to our program. We have to ensure that it is not placed on the taskbar. The documentation on [taskbar buttons](#) suggests that owned windows do not get placed on the taskbar by default. Running the sample executable also demonstrates that the second window generally stays on top of the main one. This also indicates an [owner-owned](#) relation between them.

1. First we should add another field to `app_2048` to store another window handle:

```
class app_2048  
{  
    private:  
        ...  
        HWND m_main, m_popup;  
        ...  
};
```

2. For the second (pop-up) window we need to tweak the styles a little and also must be able to specify the parent/owner of a window. But other than those few minor changes the window will largely have to look and behave like the main one. It should then use the same window class and be created in a similar manner. Let us then first modify the `create_window` method of to allow some additional parameters. First the declaration in the header file:

```
class app_2048  
{  
    private:  
        ...  
        HWND create_window(DWORD style, HWND parent = nullptr);  
        ...  
};
```

and then its implementation in the source file:

```
HWND app_2048::create_window(DWORD style, HWND parent)
{
    return CreateWindowExW(
        0,
        s_class_name.c_str(),
        L"2048",
        style, //change here
        CW_USEDEFAULT, 0,
        CW_USEDEFAULT, 0,
        parent, //and here
        nullptr,
        m_instance,
        this);
}
```

3. The constructor must now create both windows. The pop-up window in the example has no icon, no system menu, no close or minimize buttons. Referring back to [window styles](#) we should then omit `WS_SYSMENU` and `WS_MINIMIZEBOX` compared to the main window.

```
app_2048::app_2048(HINSTANCE instance)
: m_instance{ instance }, m_main{}, m_popup{}
{
    register_class();
    DWORD main_style = WS_OVERLAPPED | WS_SYSMENU |
        WS_CAPTION | WS_MINIMIZEBOX;
    DWORD popup_style = WS_OVERLAPPED | WS_CAPTION;
    m_main = create_window(main_style);
    m_popup = create_window(popup_style, m_main);
}
```

4. Lastly we need the window to appear in the `run` method. We should make sure the main window appears first. However, [by default](#) the last window shown will be active which does not match the behaviour of our example executable. Fortunately, if we browse the [flags](#) available for `ShowWindow` we will see that we can make the window appear without activating it.

```
int app_2048::run(int show_command)
{
    ShowWindow(m_main, show_command);
    ShowWindow(m_popup, SW_SHOWNA);
    ...
}
```

3.4 Board Layout and Window Sizes

Both windows of our program will have to visualise the state of the same game board at the same time. Instead of storing two copies tied to the windows and trying to keep them in sync, it would be advisable to separate that state from the rest of the application. This is not strictly necessary right now, as the board is static. In the homework part though that will not be the case, and having that separation be there from the start will take much less effort then trying to retroactively add it later.

The board state will contain all information about the layout: the size of the board, and positions and sizes of tiles, so that there will be no logic necessary to convert the state to some visual representation.

1. Add a header file **board.h** and a source file **board.cpp** to the project.
2. In the header file add a structure representing a tile. For now it will only hold the position and the size.

```
#pragma once
#include <array>
#include <windows.h>
struct field
{
    RECT position;
};
```

3. Below add the board class. Besides storing the array of fields, it will have constants for the numerical values taken or derived from the task description.

```
class board
{
public:
    static constexpr LONG columns = 4;
    static constexpr LONG rows = 4;
    static constexpr LONG margin = 10;
    static constexpr LONG field_count = rows * columns;
    static constexpr LONG field_size = 60;
    static constexpr LONG width =
        columns * (field_size + margin) + margin;
    static constexpr LONG height =
        rows * (field_size + margin) + margin;
    using field_array = std::array<field, field_count>;
    board();
    field_array const & fields() const { return m_fields; }
private:
    field_array m_fields;
};
```

4. The source file needs only contain the board constructor which initializes the tile fields

```
#include "board.h"
board::board() : m_fields{ }
{
    for (LONG row = 0; row < rows; ++row)
        for (LONG column = 0; column < columns; ++column)
        {
            auto &f = m_fields[row * columns + column];
            f.position.top =
                row * (field_size + margin) + margin;
            f.position.left =
                column * (field_size + margin) + margin;
            f.position.bottom = f.position.top + field_size;
            f.position.right = f.position.left + field_size;
        }
}
```

5. Modify the `app_2048` class to add a `board` member field

```
...
#include "board.h"
class app_2048
{
private:
    ...
    board m_board;
    ...
};
```

6. The application can now use some of that layout information to correctly size the windows. However, the board size should fill the client area, while the window size must also account for things like border, caption bar, etc. The size needs to be adjusted by `AdjustWindowRectEx`, as hinted in the description. We can do so just before window creation and use it when calling `CreateWindowExW`.

```
HWND app_2048::create_window(DWORD style, HWND parent)
{
    RECT size{ 0, 0, board::width, board::height };
    AdjustWindowRectEx(&size, style, false, 0);
    return CreateWindowExW(
        0, s_class_name.c_str(),
        L"2048", style,
        CW_USEDEFAULT, 0,
        size.right - size.left, size.bottom - size.top,
        parent, nullptr, m_instance, this);
}
```

3.5 Board Tiles and Colours

Since we cannot use any GDI drawing calls to show the representation of board tiles within each window, the only other option we have are [child windows](#) which are only visible in the client area of the parent. Reusing the window class of our top-level windows doesn't seem like a good idea since there is some logic in their window procedure that we don't want applied to the children.

We could create a new window class just for them, but the documentation hints at some pre-existing [system classes](#). Another clue is the mention of [WM_CTLCOLORSTATIC](#) in the description, which is a notification used by a [static control](#) - a control (child window) which purpose is to show static text or images.

1. Extend the window creation method in our application class to add a static control child window for each of the board tiles. Controls often define some additional [styles](#) that can be applied during creation.

As a side note, a static control will draw the text of its title within its bounds, but for now we will leave the title empty.

```
HWND app_2048::create_window(DWORD style, HWND parent)
{
    ...
    HWND window = CreateWindowExW(...);
    for (auto &f : m_board.fields())
        CreateWindowExW(
            0,
            L"STATIC",
            nullptr,
            WS_CHILD | WS_VISIBLE | SS_CENTER,
            f.position.left, f.position.top,
            f.position.right - f.position.left,
            f.position.bottom - f.position.top,
            window,
            nullptr,
            m_instance,
            nullptr);
    return window;
}
```

2. Running the program after that change should present the board layout in each of the two windows albeit with wrong colours. Fixing the background is the easiest, since it can be done during window class registration with the help of [CreateSolidBrush](#) function:

```

bool app_2048::register_class()
{
    WNDCLASSEXW desc{};
    if (GetClassInfoExW(m_instance, s_class_name.c_str(),
        &desc) != 0)
        return true;
    desc = {
        .cbSize = sizeof(WNDCLASSEXW),
        .lpfnWndProc = window_proc_static,
        .hInstance = m_instance,
        .hCursor = LoadCursorW(nullptr, L"IDC_ARROW"),
        .hbrBackground =
            CreateSolidBrush(RGB(250, 247, 238)),
        .lpszClassName = s_class_name.c_str()
    };
    return RegisterClassExW(&desc) != 0;
}

```

3. For tiles themselves, since we are not registering that class, we cannot set the background the same way. On the other hand controls often communicate with their parent for much of their functionality via notifications, which are just messages send to the parent's window procedure. An example of that is the aforementioned `WM_CTLCOLORSTATIC` message which the parent can `handle` to influence how the control is drawn.

But let us first create a brush that can be reused for all controls. We should add a field for it in our application class.

```

class app_2048
{
private:
    ..
    HBRUSH m_field_brush;
    ..
};

```

4. It should be initialised in the constructor

```

app_2048::app_2048(HINSTANCE instance)
    : m_instance{ instance }, m_main{}, m_popup{},
      m_field_brush{ CreateSolidBrush(RGB(204, 192, 174)) }
{
    ...
}

```

5. To set the background of a control, the parent's window procedure needs to just return the brush in response to the message we've just discussed.

```
LRESULT app_2048::window_proc(  
    HWND window, UINT message,  
    WPARAM wparam, LPARAM lparam)  
{  
    switch (message)  
    {  
        ...  
        case WM_CTLCOLORSTATIC:  
            return reinterpret_cast<INT_PTR>(m_field_brush);  
    }  
    return DefWindowProcW(window, message, wparam, lparam);  
}
```

3.6 Window Movement

The documentation provides plenty of information about setting the [initial size and position](#) of the window and how to [affect them and react to changes](#) later. For this task most of the functionality needed is covered by [SetWindowPos](#) function and [WM_WINDOWPOSCHANGED](#) message.

1. Since the positions of windows need to change with relation to the centre of the screen, we should first find its size. Add a field to our application class that will store it.

```
class app_2048  
{  
private:  
    ...  
    POINT m_screen_size;  
    ...  
};
```

2. Next, initialize it in the constructor. The [GetSystemMetrics](#) [reference page](#) can tell us how to read the resolution of the primary display.

```
app_2048::app_2048(HINSTANCE instance)  
    : m_instance{ instance }, m_main{}, m_popup{},  
      m_field_brush{},  
      m_screen_size{ GetSystemMetrics(SM_CXSCREEN),  
                     GetSystemMetrics(SM_CYSCREEN) }  
{  
    ...  
}
```

For now we will assume the display resolution and configuration does not change. However, an [article](#) in the documentation points us to `WM_DISPLAYCHANGE` message we can handle if we'd like to adjust that dynamically later.

Additionally, if you'd like to consider the desktop composed of multiple monitors, constants `SM_CXVIRTUALSCREEN` and `SM_CYVIRTUALSCREEN` can be used in the code above instead.

3. We don't need to modify the starting position as `WM_WINDOWPOSCHANGED` is received as soon as the window is shown, so we can treat it as any other window movement. Declare in the application class a method in which we will encapsulate the symmetric movement logic.

```
class app_2048
{
    private:
        ...
        void on_window_move(HWND window, LPWINDOWPOS params);
        ...
};
```

Then add the all to this function in the window procedure.

```
LRESULT app_2048::window_proc(
    HWND window,
    UINT message,
    WPARAM wparam,
    LPARAM lparam)
{
    switch (message)
    {
        ...
        case WM_WINDOWPOSCHANGED:
            on_window_move(window,
                reinterpret_cast<LPWINDOWPOS>(lparam));
            return 0;

    }
    return DefWindowProcW(window, message, wparam, lparam);
}
```

4. We already know the size of the screen \mathbf{S}_s (`m_screen_size` field). For the window that has just moved, `WINDOWPOS` structure will contain the position of its top-left corner \mathbf{P}_w (fields `x` and `y`) and thse size \mathbf{S}_w (fields `cx` and `cy`). Its centre is given by:

$$\mathbf{C}_w = \mathbf{P}_w + \frac{\mathbf{S}_w}{2}$$

Size of the other window \mathbf{S}_o (variable `other_size` below) can be obtained from `GetWindowRect`. The centre of that other window must be placed symmetrically to \mathbf{C}_w with regards the center of the screen ($\frac{\mathbf{S}_s}{2}$):

$$\mathbf{C}_o = \frac{\mathbf{S}_s}{2} - \left(\mathbf{C}_w - \frac{\mathbf{S}_s}{2} \right) = \mathbf{S}_s - \mathbf{C}_w$$

Finally the new position of the top-left corner of the other window \mathbf{P}_o (variable `new_pos` in the snippet below) which we need to pass to `SetWindowPos` needs to be calculated from the following equation:

$$\mathbf{C}_o = \mathbf{P}_o + \frac{\mathbf{S}_o}{2}$$

In the code below the calculations is left as an exercise.

```
void app_2048::on_window_move(
    HWND window,
    LPWINDOWPOS params)
{
    HWND other = (window == m_main) ? m_popup : m_main;
    RECT other_rc;
    GetWindowRect(other, &other_rc);
    SIZE other_size{
        .cx = other_rc.right - other_rc.left,
        .cy = other_rc.bottom - other_rc.top };
    POINT new_pos{
        /*calculate the new position of the other window*/
    };
    if (new_pos.x == other_rc.left &&
        new_pos.y == other_rc.top)
        return;
    SetWindowPos(other, nullptr, new_pos.x, new_pos.y,
        0, 0, SWP_NOSIZE | SWP_NOACTIVATE | SWP_NOZORDER);
}
```

5. You might have noticed that centre of the board is not exactly in the centre of the window. This is due to the fact that the window size includes the borders and the caption bar (recall how we adjusted the window size with `AdjustWindowRectEx`.) If you'd like the board centres to be symmetric with regards to the screen centre, the solution from previous step needs to be modified.

To obtain the center of the board for the window that that has just moved \mathbf{C}_w , you can take the position of the board relative to the board's top-left corner (`{ board::width / 2, board::height / 2 }`) and convert it to a screen position with `ClientToScreen`. Please consult the [documentation](#) to see how to use it.

The position of the board centre for the other window C_o can be calculated just like in the previous step. However, the new position of the other window P_0 no longer can be obtained by subtracting $\frac{S_o}{2}$ from C_o . The solution to that problem as well as the implementation details are again left as an exercise.

3.7 Semitransparent Window

Windows that can turn transparent are called [layered windows](#) and the feature can be enabled by an extended window style. In the example program the popup window shows as transparent so we'll try to match that behaviour.

1. We need to extend `create_window` method of our application once again to accept the extended style. First the declaration.

```
class app_2048
{
private:
    ...
    HWND create_window(DWORD style, HWND parent = nullptr,
                      DWORD ex_style = 0);
    ...
};
```

Next the definition.

```
HWND app_2048::create_window(DWORD style, HWND parent,
                             DWORD ex_style)
{
    ...
    HWND window = CreateWindowExW(ex_style,
    s_class_name.c_str(), L"2048", style,
    CW_USEDEFAULT, 0,
    size.right - size.left, size.bottom - size.top,
    parent, nullptr, m_instance, this);
    ...
}
```

2. The popup window now should get `WS_EX_LAYERED` style (the main window remains unchanged). The window will be completely transparent until we call `SetLayeredWindowAttributes`. For now we will make it completely opaque.

```
app_2048::app_2048(HINSTANCE instance)
: ...
{
    ...
    m_main = create_window(main_style);
```

```

        m_popup = create_window(
            popup_style, m_main, WS_EX_LAYERED);
        SetLayeredWindowAttributes(m_popup, 0, 255, LWA_ALPHA);
        ...
    }

```

3. Add a method to the application which will handle updating the popup window's transparency.

```

class app_2048
{
private:
    ...
    void update_transparency();
    ...
};

```

4. In it we can select the transparency level based on the intersection of the two window rectangles.

```

void app_2048::update_transparency()
{
    RECT main_rc, popup_rc, inter;
    GetWindowRect(m_main, &main_rc);
    GetWindowRect(m_popup, &popup_rc);
    IntersectRect(&inter, &main_rc, &popup_rc);
    BYTE a = IsRectEmpty(&inter) ? 255 : 255 * 50 / 100;
    SetLayeredWindowAttributes(m_popup, 0, a, LWA_ALPHA);
}

```

5. This function should be called after each change to windows' positions:

```

void app_2048::on_window_move(
    HWND window,
    LPWINDOWPOS params)
{
    ...
    update_transparency();
}

```

6. If you try the program now, it should look almost right. However, the popup window becomes semitransparent even if the two windows do not visually touch each other quite yet. This is because the invisible part of the border that serves as the drop shadow is included in the rectangle obtained from `GetWindowRect`.

The [reference](#) for the function (and the two remaining hints from the description) suggest we should try `DwmGetWindowAttribute` instead. In

turn, [reference](#) page for that function lists it as available through the `<dwmmapi.h>` header which we need to include in `app_2048.cpp`.

There it also says the library `dwmmapi.lib` needs to be added in the project settings. Navigate to `Configuration Properties >> Linker >> Input`, expand the drop-down for `Additional Dependencies`, select `<Edit...>` and input the library name there.

7. Replace the calls to `GetWindowRect` and compare the result

```
void app_2048::update_transparency()
{
    RECT main_rc, popup_rc, inter;
    DwmGetWindowAttribute( m_main,
        DWMWA_EXTENDED_FRAME_BOUNDS,
        &main_rc, sizeof(RECT));
    DwmGetWindowAttribute( m_popup,
        DWMWA_EXTENDED_FRAME_BOUNDS,
        &popup_rc, sizeof(RECT));
    IntersectRect(&inter, &main_rc, &popup_rc);
    BYTE a = IsRectEmpty(&inter) ? 255 : 255 * 50 / 100;
    SetLayeredWindowAttributes(m_popup, 0, a, LWA_ALPHA);
}
```

3.8 Program Icon

Setting an icon for the window is as simple as assigning an icon handle (`HICON`) to `hIcon` field of `WNDCLASSEX` structure when registering the window class. However, the icon object must first be created. We could [load](#) the icon from a loose file, but in this example we will store it as resource. Resources are pieces of data that are embedded in the executable file itself. Icon in stored in such a way can also show as the icon for the executable when it is shown in file explorer.

1. First let's copy the `2048_icon.ico` file into your project directory (the folder, where all your source files for the project are located).
2. To be able to add resources we need to create a *Resource File* which describes all data associated with the executable. From the main menu of Visual Studio select `Project >> Add New Item...`, then under `Visual C++ >> Resource` select the `Resource File (.rc)` template and add it to your project with the name `Resource.rc`.
3. Find the file in the Solution Explorer under *Resource Files* and double-click it. This should open the *Resource View* window.
4. In there right-click on the resource file and select `Add Resource...`. In the dialog window click `Import...`.





5. Change the file type filter to *Icon Files (*.ico)*, then find and open the icon file you have copied in the first step.
6. The icon will be added with an auto-generated identifier, in all caps with an **ID-** prefix. You can change it by clicking on the identifier, going to the Properties Window and editing the *ID* field. If you don't have that window open, from main menu select **View** **Other Windows** **Properties Window**. Change the identifier now to **ID_APPICON**.
7. Each resource is accessible via a symbolic constant with the same name as the resource identifier (the one we have just changed). The symbolic constant is introduced in an auto-generated header file, usually called **resource.h**, which we should include in our program. Add the following line at the top of **app_2048.cpp**.

```
#include "resource.h"
```

8. The same **function** for loading icons (or images in general) from loose files can also help us load one from a resource. The **MAKEINTRESOURCE** macro will allow us to use the resource identifier in place of a string with the resource name or a file path. Modify the registration of the window class as follows:

```
bool app_2048::register_class()
{
    ...
    desc = {
        .cbSize = sizeof(WNDCLASSEXW),
        .lpfnWndProc = window_proc_static,
        .hInstance = m_instance,
        .hIcon = static_cast<HICON>(LoadImageW(
            m_instance,
            MAKEINTRESOURCEW(ID_APPICON),
            IMAGE_ICON,
            0, 0,
            LR_SHARED | LR_DEFAULTSIZE)),
        /*you can also use:
        LoadIconW(m_instance,
            MAKEINTRESOURCEW(ID_APPICON)) */
        .hCursor = LoadCursorW(nullptr, L"IDC_ARROW"),
        .hbrBackground =
            CreateSolidBrush(RGB(250, 247, 238)),
        .pszClassName = s_class_name.c_str()
    };
    return RegisterClassExW(&desc) != 0;
}
```

The `LR_SHARED` flag will ensure the icon handle is reused even if we load it multiple times. It also makes it so the system is responsible for destroying the handle. It's worth noting that instead of specifying the width and height of the icon, we pass 0 for both and pass the `LR_DEFAULTSIZE` flag so the actual resolution of the icon is used.

You should now see it as the icon for the executable in file explorer, on the caption bar of the application window, on the task bar, and also when switching between windows using  +  or  + .

4 Extended Functionality

While most of the solution for task described in section 5 is left as an exercise, there are few things left to demonstrate, which should get you started with that task.

4.1 Timers

So far all interaction in our program are driven by user input (namely dragging one of the windows or closing them). There are plenty of cases, however, where we need to update the state of our program without having to wait for the user to do something, e.g. showing animations, etc.

As a demonstration we'll try to show on the title bar of our windows how long a given game has been running. There are many ways we could achieve that effect (please refer to the lecture for the discussion of options). For this the precision and accuracy are not particularly important, which makes `timers` a good choice.

1. First we'll specify the ID for our timer. We'll also need to remember when has the game started. In the definition of `app_2048` class add the appropriate field:

```
#include <chrono>

class app_2048
{
private:
    ...
    static constexpr UINT_PTR s_timer = 1;
    ...
    std::chrono::time_point<std::chrono::system_clock>
        m_startTime;
    ...
};
```

2. Additionally, declare the method we'll use to update the window titles

```

class app_2048
{
private:
    ...
    void on_timer();
    ...
}

```

and implement it in `app_2048.cpp`.

```

void app_2048::on_timer()
{
    using namespace std::chrono;
    auto title = std::format(L"2048 {:%M:%S}",
        duration_cast<duration<int>>(
            system_clock::now() - m_startTime));
    SetWindowTextW(m_main, title.c_str());
    SetWindowTextW(m_popup, title.c_str());
}

```

3. After the windows are shown in the `run` method start the timer and store the start time. We'll aim the timer at the main window, but both windows' titles will update in response.
-

```

int app_2048::run(int show_command)
{
    ShowWindow(m_main, show_command);
    ShowWindow(m_popup, SW_SHOWNA);
    SetTimer(m_main, s_timer, 1000, nullptr);
    m_startTime = std::chrono::system_clock::now();
    ...
}

```

4. When timer elapses, `WM_TIMER` notifies the target window. This repeats until the timer is destroyed. Handle the message to change the titles:
-

```

LRESULT app_2048::window_proc(HWND window, UINT message,
                               WPARAM wparam, LPARAM lparam)
{
    switch (message)
    {
    case WM_TIMER:
        on_timer();
        return 0;
    ...
    }
    ...
}

```

The timer ID we specified in `SetTimer` will be the value of `wparam`, but since we only have one timer, we will ignore it.

4.2 Main Menu

Main menu is a rather commonly used element of a window that shows just below the caption bar and allows the user to execute some command defined by the program. Menus can be created and edited programmatically (more on that in the lecture and in [documentation](#)), but if the content is not going to change, by far the easiest way to create them is to add them to resources. This allows us to edit them using a visual WYSIWYG editor.

1. In *Solution Explorer* under *Resource Files* double-click *Resource.rc*. Like before it should open it in *Resource View* window.
2. Right-click on the resource file name and select `Add Resource...`.
3. In the dialog box select *Menu* from the list and click `New`.
4. Just like before the menu will be added with a some auto-generated identifier. You can change it to `ID_MAINMENU` by clicking on the identifier and going to the Properties Window, then editing the *ID* field. If you don't have that window open, go to the main menu and select `View >> Other Windows >> Properties Window`.
5. Double-clicking the identifier in *Resource View* window should open a menu editor which shows the visualisation of the menu bar and allows you to edit it. Click on *Type Here* and enter **Game**
6. Similarly In the popup menu below the added item enter **New Game**.
7. The subitems are also added with generated identifiers, which we'd like to change. Click on the **New Game** menu item you've just added, go to *Properties* window and change to ID field to `ID_NEWGAME`
8. To aid with keyboard navigation of the menu we can indicate which menu item should be selected if a letter is pressed on the keyboard while holding `[Alt]` key. We do so by putting an ampersand **&** before one of the letters in of the menu item. Select the **Game** item and in the *Properties* window under *Caption* place **&** before letter **G** in **Game**. Do the same for **New Game** item placing it before letter **N**.
9. Menu items can also indicate which global keyboard shortcuts (a.k.a. keyboard accelerators) are associated with the same command. This doesn't automatically register the shortcut, but only serves as visual guide. We'll discuss keyboard accelerators in the next step, but we'll add the hint now. The shortcut for *New Game* will be `[Ctrl] + [N]`. Hint

is added in the same *Caption* field separated by `\t` from the main text. The caption for **New Game** should now be `&New Game\tCtrl+N`.

10. The simplest way to associate a main menu with a window is to set it when registering the window class. Modify the `register_class` method of `app_2048`:

```
bool app_2048::register_class()
{
    ...
    desc = {
        ...
        .hbrBackground =
            CreateSolidBrush(RGB(250, 247, 238)),
        .lpszMenuName =
            MAKEINTRESOURCEW(ID_MAINMENU), //add this
        .lpszClassName = s_class_name.c_str()
    };
    ...
}
```

11. If you run the program now, you'll notice that tiles no longer all fit in the window. We also need to update the window size, to account for the added menu bar, so that the client area size remains unchanged. Recall the function `AdjustWindowRectEx` we called to calculate the desired size. Its third parameter is a boolean value indicating if a window will have a main menu. Change its value to `true` in `create_window` method.
12. Whenever a menu item is selected, the window receives `WM_COMMAND` message. The lower word (i.e. the lowest 16 bits) of `wparam` will contain the identifier of the menu item, e.g. `ID_NEWGAME` for our **New Game**. Lets declare a method in `app_2048` called `on_command`:

```
class app_2048
{
private:
    ...
    void on_command(WORD cmdID);
    ...
};
```

then implement it in `app_2048.cpp` so that it resets the game time:

```
void app_2048::on_command(WORD cmdID)
{
    switch (cmdID)
    {
        case ID_NEWGAME:
```

```

        m_startTime = std::chrono::system_clock::now();
        on_timer();
        break;
    }
}

```

and use it to handle `WM_COMMAND`:

```

LRESULT app_2048::window_proc(
    HWND window, UINT message,
    WPARAM wparam, LPARAM lparam)
{
    switch (message)
    {
        case WM_COMMAND:
            on_command(LOWORD(wparam));
            return 0;
        ...
    }
    ...
}

```

4.3 Keyboard Accelerators

Another, hinted above, method of allowing the user to execute predefined commands is via shortcuts, known in Windows API as keyboard accelerators. A group of shortcuts is represented as an accelerator table, and again it is easiest to manage from resources.

Accelerator table is loaded using `LoadAcceleratorsW`. It is safe to call the function for the same table multiple times, it will only be loaded once, with subsequent calls returning the same handle. The table also doesn't need to be freed.

To actually produce accelerator commands in response to keyboard input, the message loop needs to be modified as well to include the call to `TranslateAcceleratorW` before the message is dispatched.

For more details consult the lecture and the [documentation](#).

1. In *Solution Explorer* under *Resource Files* double-click *Resource.rc*. Like before it should open it in *Resource View* window.
2. Right-click on the resource file name and select `Add Resource...`.
3. In the dialog box select *Accelerator* from the list and click `New`.
4. Just like before the accelerator table will be added with a some auto-generated identifier. You can change it to `ID_SHORTCUTS` by clicking on the identifier and going to the Properties Window, then editing the

ID field. If you don't have that window open, go to the main menu and select **View** » **Other Windows** » **Properties Window**.

5. Double-clicking the identifier in *Resource View* window should open a accelerator table editor which allows us to edit the shortcuts. There should be one accelerator already present, with auto-generated ID. In the *ID* column expand the drop-down, and select **ID_NEWGAME** which is the same ID we used for the **New Game** menu item.
6. Ensure the value in *Type* column is **VIRTKEY**.
7. In the *Modifiers* column select **Ctrl** from the drop-down.
8. Click in the *Key* column and type in **N**.
9. Modify the main loop in **run** method of **app_2048** class to load the accelerators and process them:

```
int app_2048::run(int show_command)
{
    ...
    HACCEL shortcuts = LoadAcceleratorsW(m_instance,
        MAKEINTRESOURCEW(IDR_SHORTCUTS));
    while ((result = GetMessageW(&msg, nullptr, 0, 0)) != 0)
    {
        if (result == -1)
            return EXIT_FAILURE;
        if (!TranslateAcceleratorW(
            msg.hwnd, shortcuts, &msg))
        {
            TranslateMessage(&msg);
            DispatchMessageW(&msg);
        }
    }
    return EXIT_SUCCESS;
}
```

Note that if message was processed by **TranslateAcceleratorW**, it should not be processed further.

10. No additional message handling is necessary. Since we've chosen an existing command ID for our shortcut, our **on_command** method will respond to it as well.

4.4 GDI Drawing

It is generally possible to draw on a window from any part of our program, but the recommended approach is to put it in a handler of **WM_PAINT** message. Besides just being the most convenient, it also helps to keep the

drawing logic together. Then, whenever we need our window to repaint itself we can call `InvalidateRect`.

To execute any drawing operation we need a *Device Context* handle. In `WM_PAINT` the context is obtained by calling `BeginPaint`. A corresponding call to `EndPaint` is also required after the drawing is done.

The parameters of most drawing functions in GDI only describe the type and geometry of the shape we want to draw. Parameters that affect how the drawn shape will look like are kept by the state of the *Device Context*. So if we want to change the style or color of the shape outline and interior, font used for drawing text, or the myriad of other options, we need to modify the context's state first, and call the drawing function second.

As an example we'll change the board fields visual representation from static child controls to rounded rectangles drawn with GDI. As always, refer to the lecture and the [reference](#) for more information.

1. First we need to get rid of the child controls. Comment out the loop that creates them in `create_window` method of `app_2048` class. Also `WM_CTLCOLORSTATIC` handling in `window_proc` is no longer needed.
2. Let's declare `on_paint` method in `app_2048`:

```
class app_2048
{
private:
    ...
    void on_paint(HWND window);
    ...
};
```

and use it to handle `WM_PAINT` message:

```
LRESULT app_2048::window_proc(HWND window, UINT message,
                               WPARAM wparam, LPARAM lparam)
{
    switch (message)
    {
        case WM_PAINT:
            on_paint(window);
            return 0;
        ...
    }
    ...
}
```

3. Place it's implementation in `app_2048.cpp`. First we need to call `BeginPaint` to obtain the *Device Context* handle. We'll also need a `PAINTSTRUCT` variable in which `BeginPaint` will store some additional

information, such as the area of the window that needs to be repainted. We could use that to only redraw fields that overlap that area, but our painting logic is too simple to warrant such an optimization.

```
void app_2048::on_paint(HWND window)
{
    PAINTSTRUCT ps;
    HDC dc = BeginPaint(window, &ps);
}
```

4. To change the colour and style of shape outline we need a pen object handle `HPEN`. Colour and style the shape is filled with are controlled by a brush object `HBRUSH`. Those are just two examples of GDI object handles and many more such types exist in GDI.

GDI objects are bound to the context using `SelectObject` function. It also returns the handle to the previously selected object. We need to keep the old ones around, because we must restore them after the painting is done.

For filling we'll use the `m_field_brush` we've created before, and since we don't want an outline, we'll use a `NULL_PEN`:

```
void app_2048::on_paint(HWND window)
{
    ...
    auto oldBrush = SelectObject(dc, m_field_brush);
    auto oldPen = SelectObject(dc,
                               GetStockObject(NULL_PEN));
}
```

5. Once we've set up how we want our fields to be drawn, we can go over them and draw a rounded rectangle for each:

```
void app_2048::on_paint(HWND window)
{
    ...
    for (auto &f : m_board.fields())
        RoundRect(dc, f.position.left, f.position.top,
                  f.position.right, f.position.bottom, 11, 11);
}
```

6. What's left is the clean-up. Restore the old object handles to the context and call `EndPaint`:

```
void app_2048::on_paint(HWND window)
{
    ...
    SelectObject(dc, oldBrush);
    SelectObject(dc, oldPen);
    EndPaint(window, &ps);
}
```

We don't call `DeleteObject` on our brush at the end, because we want to keep it around. Stock object, like `NULL_PEN`, don't need to be deleted either.

However, since it's not a good idea to keep too many GDI objects alive outside `WM_PAINT`, you'll often need to create temporary ones in the message handler. Make sure to delete them when you no longer use them, e.g.:

```
DeleteObject(SelectObject(dc, oldPen));
```

Be very careful not to leak any GDI objects! There is a limited global pool for them in the system. If you fill it up, it might not only mess up the drawing of your own window, but also impede other programs' ability to repaint their windows as well.

5 Homework Task

With the laboratory part completed you can attempt to extend the application according to the following requirements to finish the implementation of the game. The requirements are listed below and once again you are provided with the sample executable `2048_home.exe` which demonstrates the intended functionality. Just like before, in any unspecified cases your solution should handle in the same manner as the example.

General Requirements

- For drawing/creating shapes GDI library can be used instead of child windows (but not other libraries like GDI+).
- Proper resource management is required. Any and all acquired objects must be properly freed/destroyed.

Layout Requirements

- Two windows with the same board as content,
- The board contains a score counter
- Each tile can contain a block with a number shown. Colour of the block should depend on that value, for example:

2 – (238, 228, 198)
4 – (239, 225, 218)
8 – (243, 179, 124)
16 – (246, 153, 100)
32 – (246, 125, 98)
64 – (247, 93, 60)
128 – (237, 206, 116)
256 – (239, 204, 98)
512 – (243, 201, 85)
1024 – (238, 200, 72)
2048 – (239, 192, 47)

- Main menu that allows to:
 - Start a new game,
 - Select the goal number for the block value that will result in winning the game (8, 16, 64, 2048)
 - See the current goal number indicated in the menu

Game Rules

- The game is played by moving and combining the blocks to achieve a block with the goal number (8, 16, 64, 2048).
- All block move in the direction indicated by the user.
- When a pair of adjacent blocks with the same number move in the direction along which they neighbour each other, they are combined into one block containing the sum of their values.
- A block resulting from joining cannot be combined with another in the same move.
- Each block moves in the direction until it is blocked by another block or the board edge.
- After each move a new block with a value of 2 appears on one of the empty tiles.
- The player wins when a pair of blocks combines to achieve the goal number (8, 16, 64, 2048).
- Game ends with a loss if there are no empty tiles and there is no pair of neighbouring block that can be combined in the next move.
- Each time a pair of blocks is combined the value of the resulting block is added to the score.

Board View Logic

- If the goal value of a block is reached, the board should be covered with a green, semi-transparent surface with a text announcing the player has won. No further moves should be accepted.
- If there is no more moves to be made, a similar red surface should cover the board announcing the player has lost. Again, no further moves should be accepted.
- When a new block is added to an empty tile the program should animate its size to grow until reaching the size of the tile.
- When block is created by combining two others, its size should be animated to first grow a bit, then return to the normal size.
- When the program is started it should restore the game from the moment it was last closed. That should include position and values of blocks on tiles and the current state of the game (won, lost, in progress).

Recommended Controls The move direction should be selected with W, S, A and D keys.

Hints Consider the following functions and messages when implementing your program:

- `CreateSolidBrush`
- `WM_PAINT`
- `WM_TIMER`
- `WM_KEYDOWN`
- `WritePrivateProfileStringW`
- `GetPrivateProfileStringW`
- `FillRect`
- `AlphaBlend`
- `DrawText`
- `CheckMenuItem`
- `RoundRect`