The DirectX Framework

Code Tutorial

Dr Panagiotis Perakis (MC)

“THE COM FRAMEWORK” - CODE ANALYSIS

COM (Component Object Model)

- DirectX is provided as a series of COM components
- Enables DirectX to be language-independent and provides for backwards compatibility
- Rather than using the C++ *new* keyword, we obtain pointers to COM interfaces via calls to specific functions

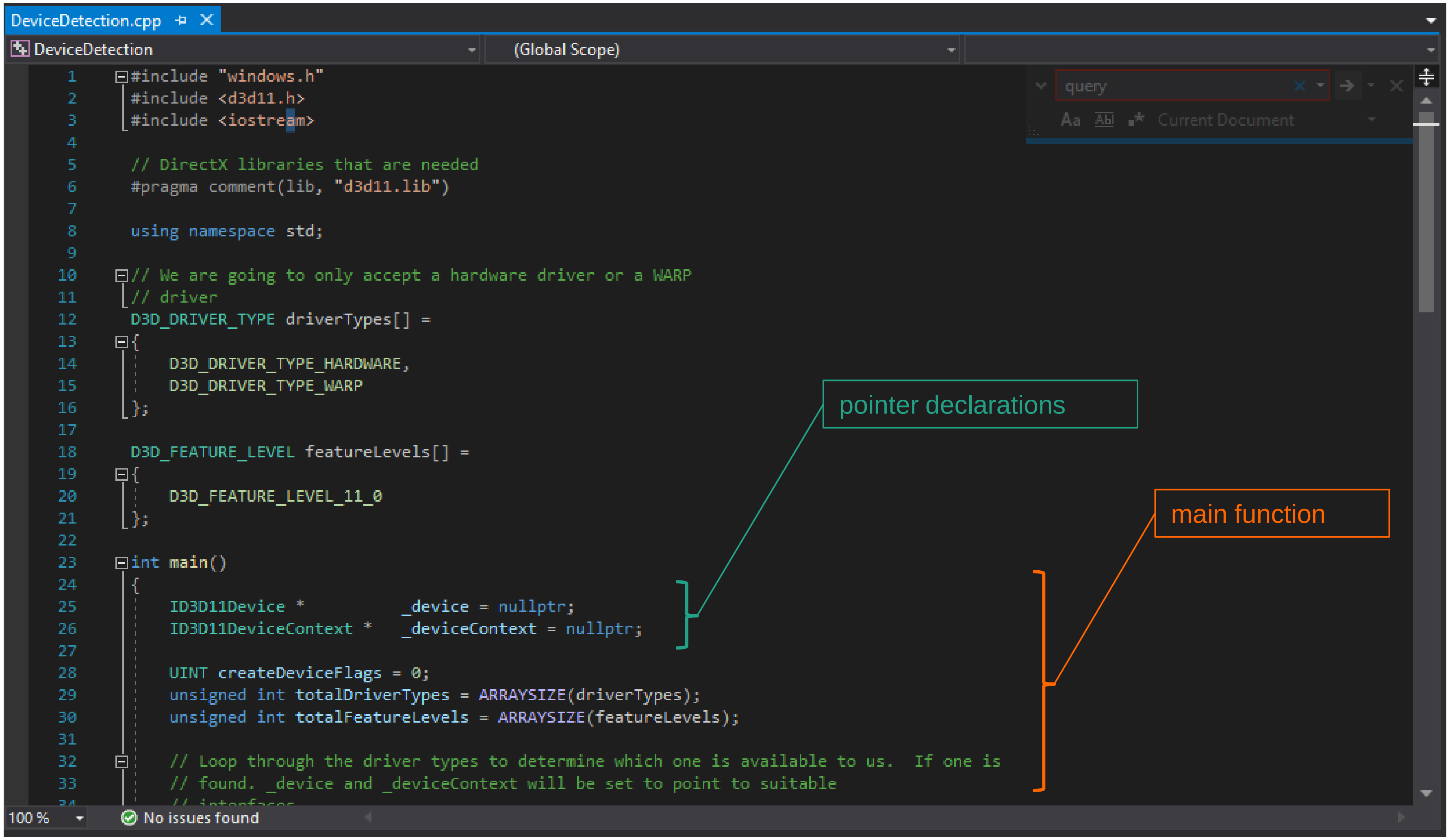
COM (Component Object Model)

- All COM components are accessed via interfaces that inherit from the IUnknown interface,
- The IUnknown interface has the following methods:

Method	Description
AddRef	Increments the reference count for an interface on an object.
QueryInterface	Retrieves pointers to the supported interfaces on an object.
Release	Decrements the reference count for an interface on an object.

- When using DirectX components, methods are provided that retrieve the interface for us and do an AddRef.
  - See the DeviceDetectionWithoutComptr example.

Example: DeviceDetectionWithoutComptr



```
#include "windows.h"
#include <d3d11.h>
#include <iostream>

// DirectX libraries that are needed
#pragma comment(lib, "d3d11.lib")

using namespace std;

// We are going to only accept a hardware driver or a WARP
// driver
D3D_DRIVER_TYPE driverTypes[] =
{
    D3D_DRIVER_TYPE_HARDWARE,
    D3D_DRIVER_TYPE_WARP
};

D3D_FEATURE_LEVEL featureLevels[] =
{
    D3D_FEATURE_LEVEL_11_0
};

int main()
{
    ID3D11Device *          _device = nullptr;
    ID3D11DeviceContext *   _deviceContext = nullptr;

    UINT createDeviceFlags = 0;
    unsigned int totalDriverTypes = ARRAYSIZE(driverTypes);
    unsigned int totalFeatureLevels = ARRAYSIZE(featureLevels);

    // Loop through the driver types to determine which one is available to us.  If one is
    // found. _device and _deviceContext will be set to point to suitable
```

COM (Component Object Model)

- Once we a have pointer to a COM interface, we can use that interface to call other functions/methods
- Once we are done with an interface, we **must** remember to call its Release method (inherited from IUnknown)
  - If you don't Release the object, you will find your application does not terminate!
- All COM components handle their own memory management
  - If an interface is not released, there will be significant memory leaks
- The problem is that it is really easy to forget to Release an object.
  - We are going to be using a lot of COM interfaces and remembering to release each one is a chore.

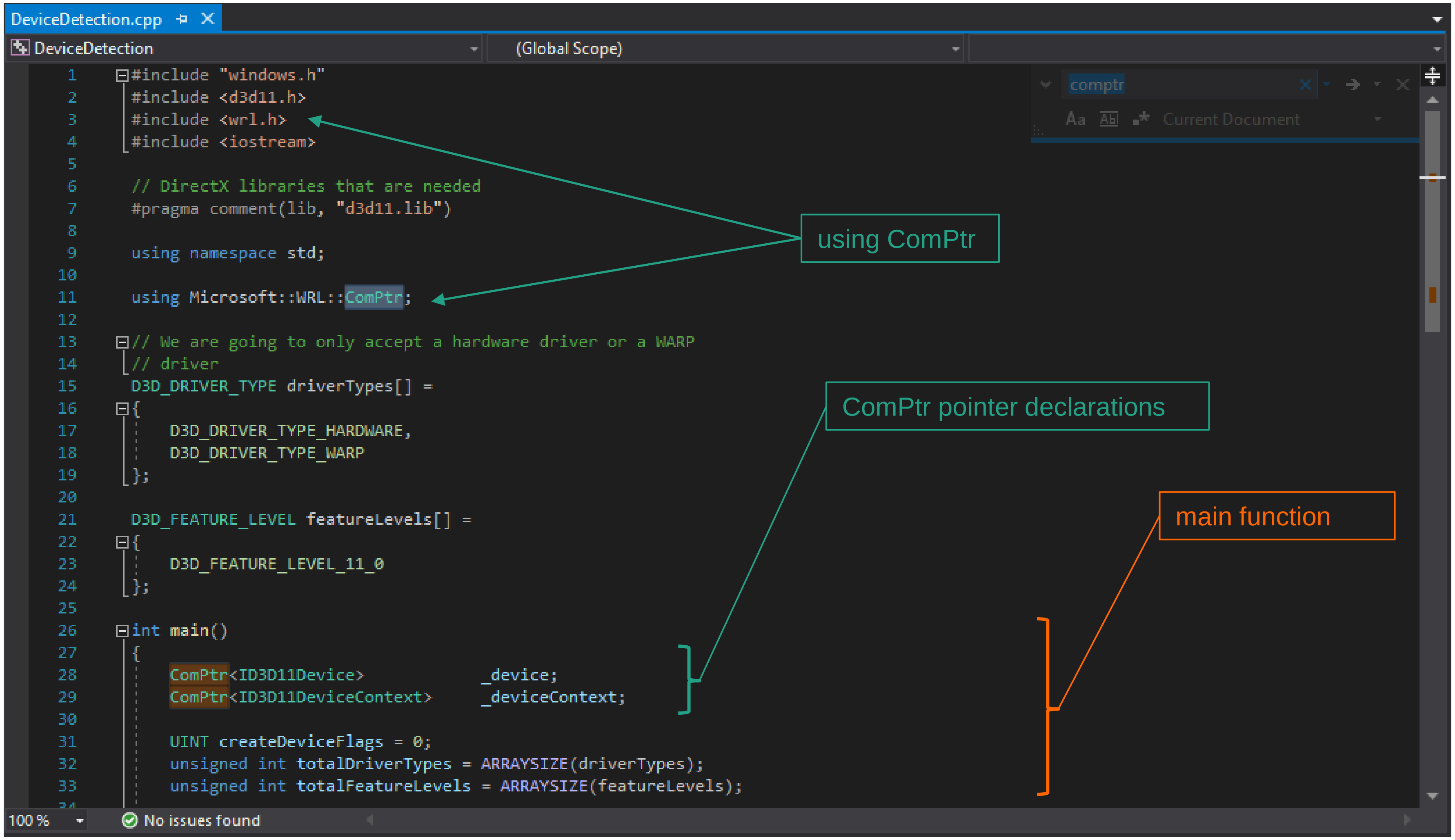
ComPtr

- The solution to this problem is to use the ComPtr template class
  - ComPtr is a *smart pointer* type that represents a specified interface
  - ComPtr automatically maintains a reference count for the underlying interface pointer and releases the interface when the variable goes out of scope.
- There are two key methods for a ComPtr:

Get()	Returns the pointer to the underlying interface
GetAddressOf()	Returns the address of the underlying pointer

- One of the reasons why many books do not reference ComPtr is that it did not appear in the Windows SDK until Windows 8. However, because it is implemented as a template class, you can use it on code that needs to run on Windows 7 or later.
  - See the DeviceDetectionWithComPtr example.

Example: DeviceDetectionWithComPtr



DeviceDetection.cpp

DeviceDetection (Global Scope)

```
#include "windows.h"
#include <d3d11.h>
#include <wrl.h>
#include <iostream>

// DirectX libraries that are needed
#pragma comment(lib, "d3d11.lib")

using namespace std;

using Microsoft::WRL::ComPtr;

// We are going to only accept a hardware driver or a WARP
// driver
D3D_DRIVER_TYPE driverTypes[] =
{
    D3D_DRIVER_TYPE_HARDWARE,
    D3D_DRIVER_TYPE_WARP
};

D3D_FEATURE_LEVEL featureLevels[] =
{
    D3D_FEATURE_LEVEL_11_0
};

int main()
{
    ComPtr<ID3D11Device>            _device;
    ComPtr<ID3D11DeviceContext>     _deviceContext;

    UINT createDeviceFlags = 0;
    unsigned int totalDriverTypes = ARRAYSIZE(driverTypes);
    unsigned int totalFeatureLevels = ARRAYSIZE(featureLevels);
```

“THE DIRECTX BASE FRAMEWORK” - CODE ANALYSIS

Starting with DirectX Code
Example: Graphics2_DirectX11_Base

- The following slides should be read in conjunction with reading the sample code provided to you.
- The basic code that is going to be used for all DirectX applications in this module is provided in the solution Graphics2_DirectX11_Base.
- This builds on the basic framework introduced in Graphics 1 (with a few minor modifications) and includes the code needed to initialise Direct3D 11.
- Please do not treat this code as a ‘black box’ that you can use, but ignore how it works. You need to understand code we give you.
- At the moment, it just clears the window to a black background
- Next week, we will see how to extend this to render objects.

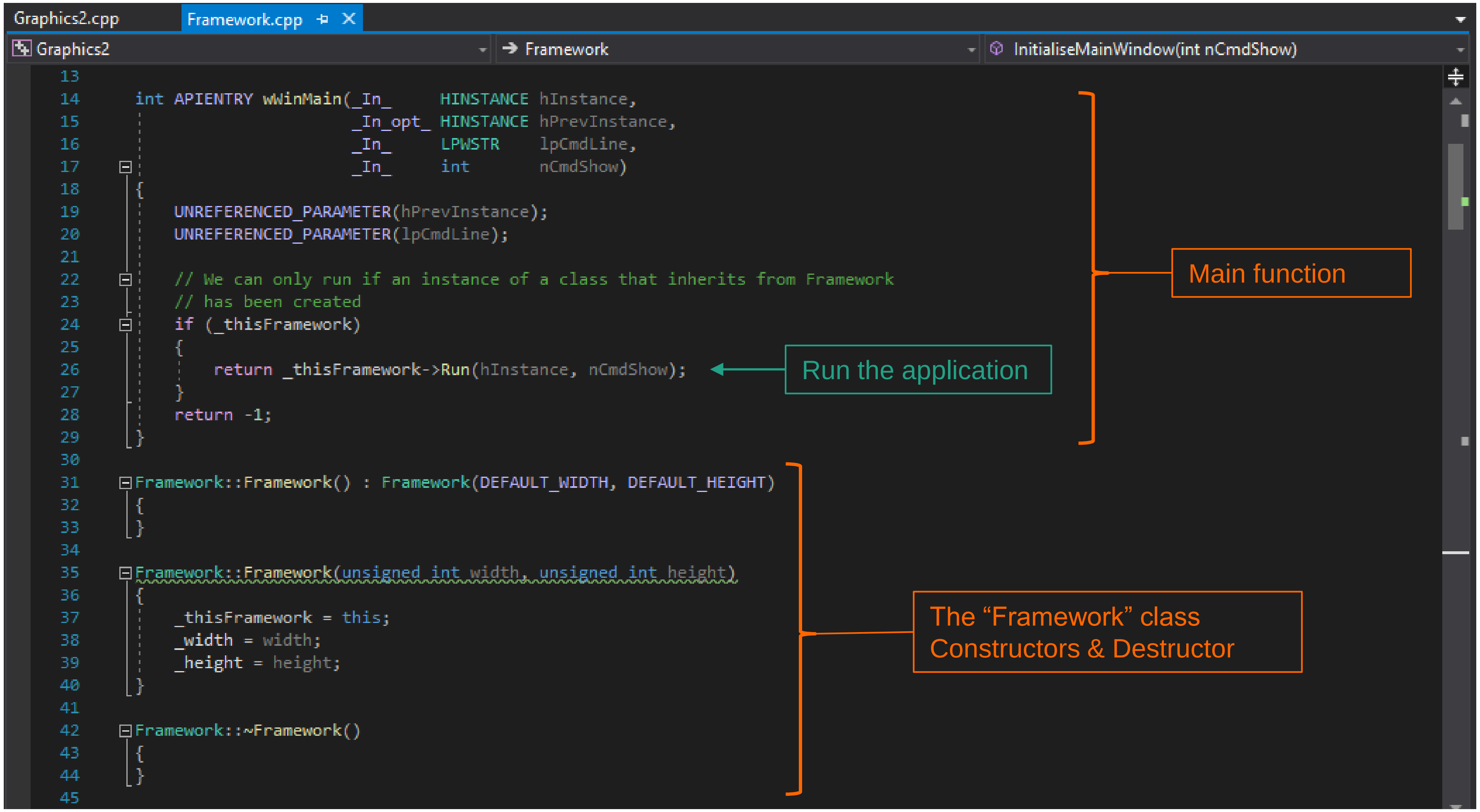
Main Driver Types

- D3D_DRIVER_TYPE_HARDWARE:
  - The GPU hardware supports the required feature levels.
- D3D_DRIVER_TYPE_WARP:
  - A highly optimised software driver that supports all Direct3D 11 features. It makes use of whatever hardware support is available.
- D3D_DRIVER_TYPE_SOFTWARE:
  - The driver is implemented completely in software. We don’t want this one since the performance is just too slow.

Feature Levels

- Indicates which levels of Direct3D are required by the application
  - We only specify D3D_FEATURE_LEVEL_11_0., because we only want Direct3D 11.0 features for now.

Application’s Main Function



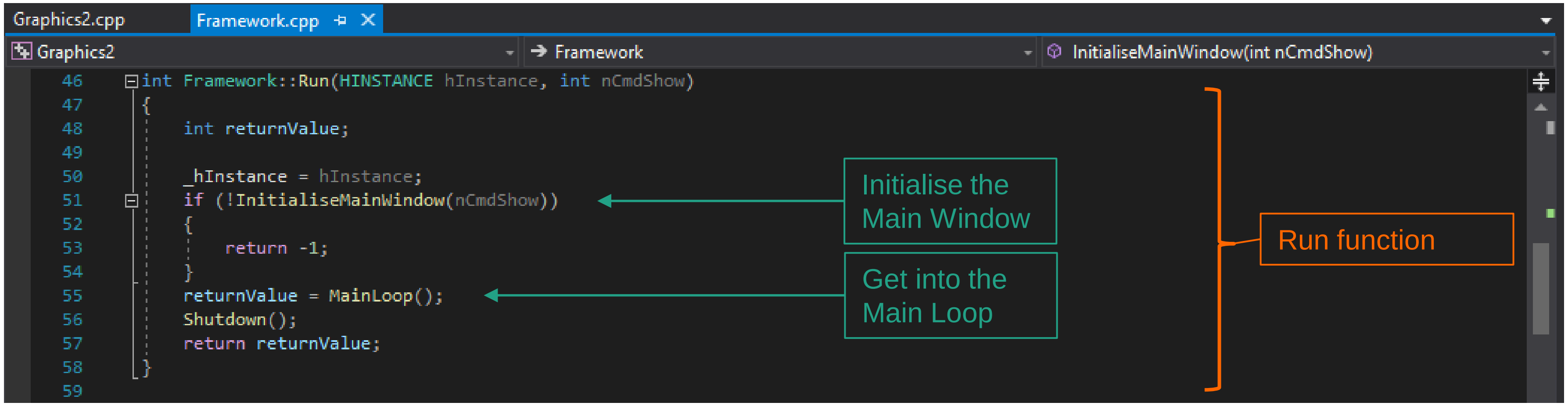
```cpp
int APIENTRY wWinMain(_In_     HINSTANCE hInstance,
                      _In_opt_ HINSTANCE hPrevInstance,
                      _In_     LPWSTR    lpCmdLine,
                      _In_     int       nCmdShow)
{
    UNREFERENCED_PARAMETER(hPrevInstance);
    UNREFERENCED_PARAMETER(lpCmdLine);

    // We can only run if an instance of a class that inherits from Framework
    // has been created
    if (_thisFramework)
    {
        return _thisFramework->Run(hInstance, nCmdShow);
    }
    return -1;
}

Framework::Framework() : Framework(DEFAULT_WIDTH, DEFAULT_HEIGHT)
{
}

Framework::Framework(unsigned int width, unsigned int height)
{
    _thisFramework = this;
    _width = width;
    _height = height;
}

Framework::~Framework()
{
}
```

Main function

Run the application

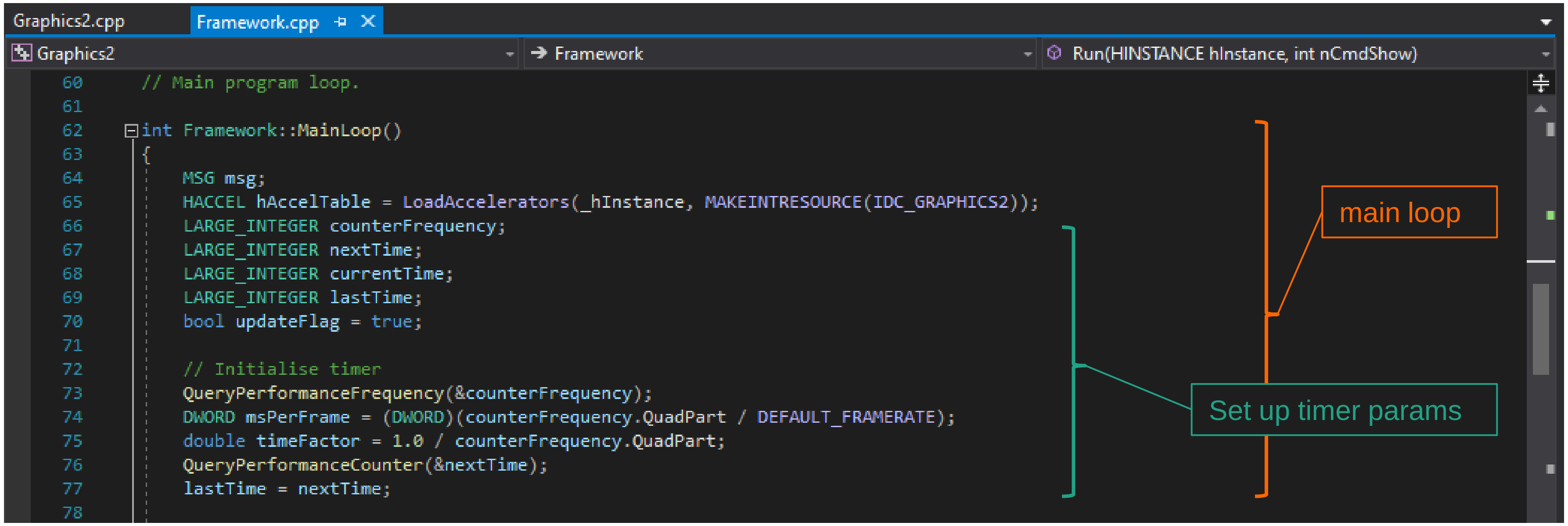
The “Framework” class
Constructors & Destructor

Running the Application



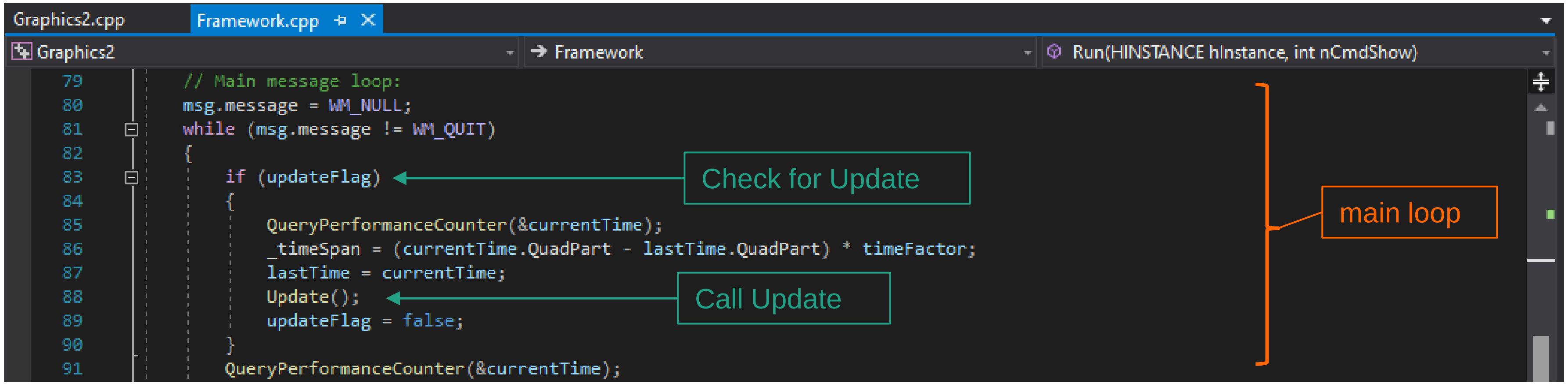
```
int Framework::Run(HINSTANCE hInstance, int nCmdShow)
{
    int returnValue;

    _hInstance = hInstance;
    if (!InitialiseMainWindow(nCmdShow))
    {
        return -1;
    }
    returnValue = MainLoop();
    Shutdown();
    return returnValue;
}
```

The Application's Main-Loop



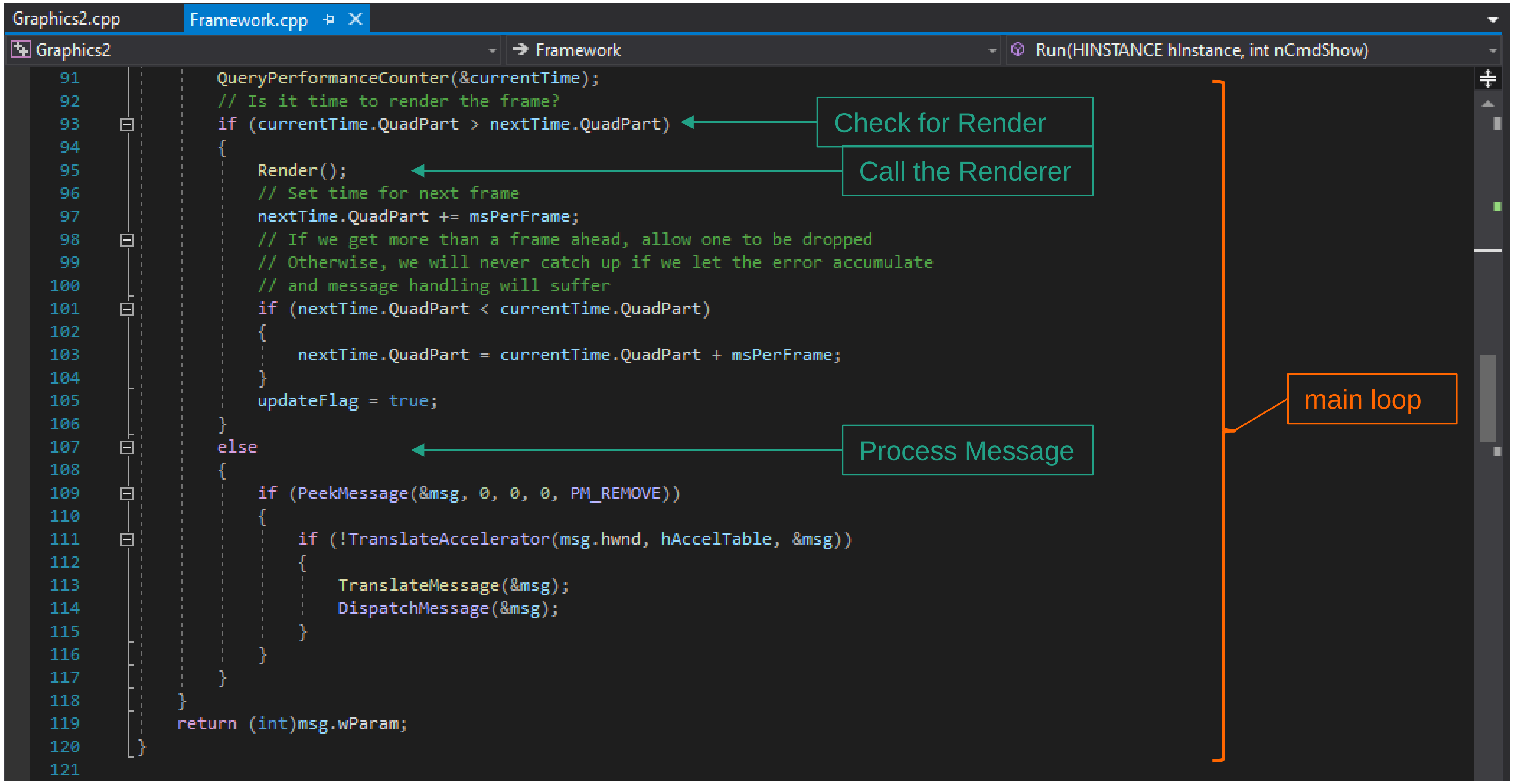
```
// Main program loop.

int Framework::MainLoop()
{
    MSG msg;
    HACCEL hAccelTable = LoadAccelerators(_hInstance, MAKEINTRESOURCE(IDC_GRAPHICS2));
    LARGE_INTEGER counterFrequency;
    LARGE_INTEGER nextTime;
    LARGE_INTEGER currentTime;
    LARGE_INTEGER lastTime;
    bool updateFlag = true;

    // Initialise timer
    QueryPerformanceFrequency(&counterFrequency);
    DWORD msPerFrame = (DWORD)(counterFrequency.QuadPart / DEFAULT_FRAMERATE);
    double timeFactor = 1.0 / counterFrequency.QuadPart;
    QueryPerformanceCounter(&nextTime);
    lastTime = nextTime;

```

The Application's Main-Loop



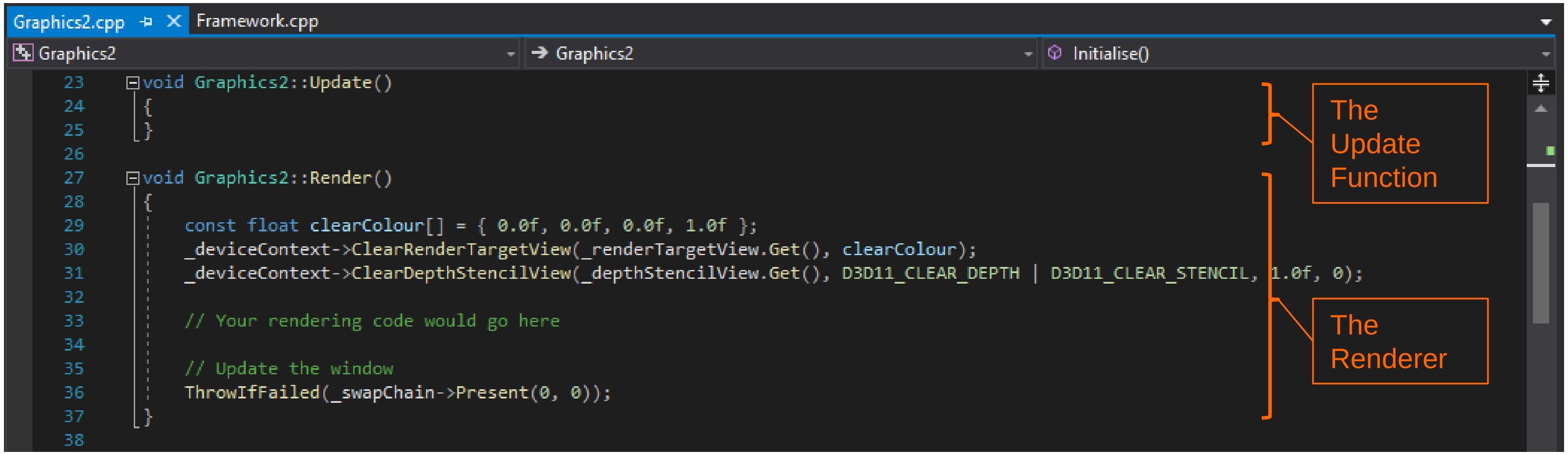
```
// Main message loop:
msg.message = WM_NULL;
while (msg.message != WM_QUIT)
{
    if (updateFlag)
    {
        QueryPerformanceCounter(&currentTime);
        _timeSpan = (currentTime.QuadPart - lastTime.QuadPart) * timeFactor;
        lastTime = currentTime;
        Update();
        updateFlag = false;
    }
    QueryPerformanceCounter(&currentTime);
```

The Application's Main-Loop



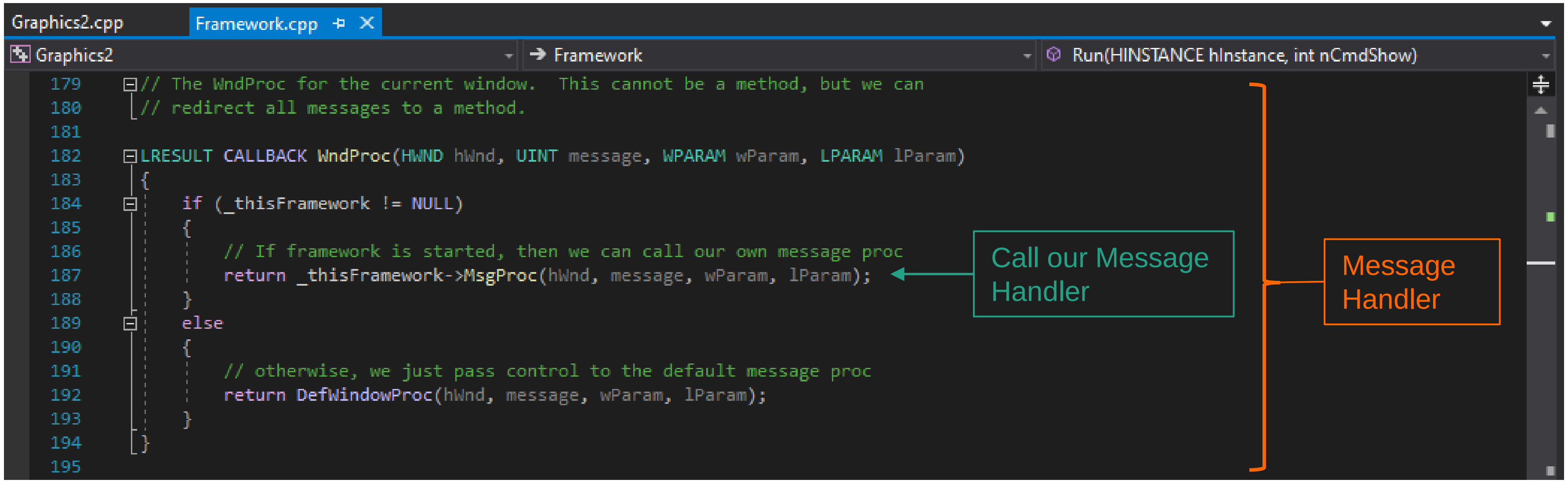
```
        QueryPerformanceCounter(&currentTime);
        // Is it time to render the frame?
        if (currentTime.QuadPart > nextTime.QuadPart)
        {
            Render();
            // Set time for next frame
            nextTime.QuadPart += msPerFrame;
            // If we get more than a frame ahead, allow one to be dropped
            // Otherwise, we will never catch up if we let the error accumulate
            // and message handling will suffer
            if (nextTime.QuadPart < currentTime.QuadPart)
            {
                nextTime.QuadPart = currentTime.QuadPart + msPerFrame;
            }
            updateFlag = true;
        }
        else
        {
            if (PeekMessage(&msg, 0, 0, 0, PM_REMOVE))
            {
                if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
                {
                    TranslateMessage(&msg);
                    DispatchMessage(&msg);
                }
            }
        }
    }
    return (int)msg.wParam;
}
```

The Update and Render Functions



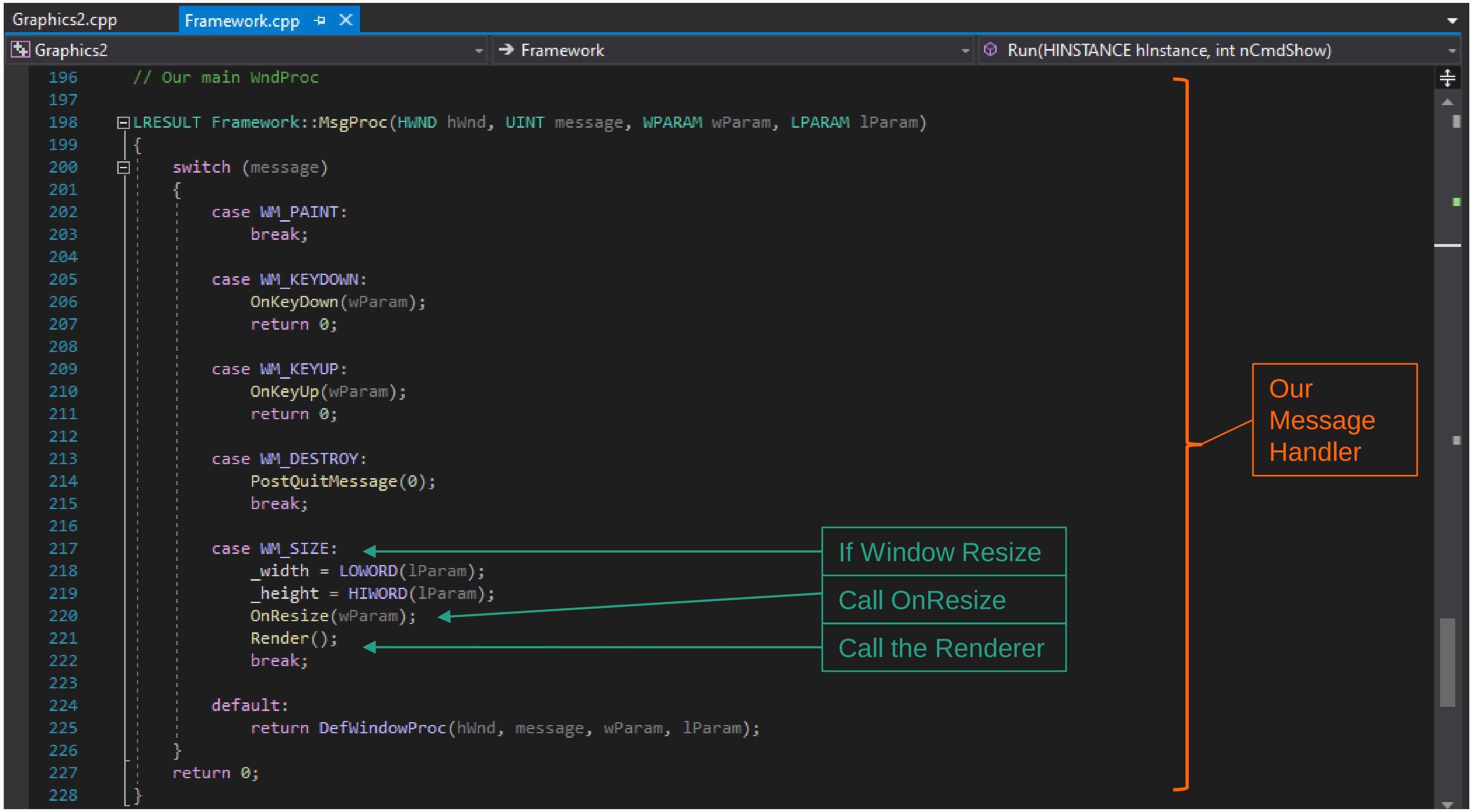
```
void Graphics2::Update()
{
}

void Graphics2::Render()
{
    const float clearColour[] = { 0.0f, 0.0f, 0.0f, 1.0f };
    _deviceContext->ClearRenderTargetView(_renderTargetView.Get(), clearColour);
    _deviceContext->ClearDepthStencilView(_depthStencilView.Get(), D3D11_CLEAR_DEPTH | D3D11_CLEAR_STENCIL, 1.0f, 0);

    // Your rendering code would go here

    // Update the window
    ThrowIfFailed(_swapChain->Present(0, 0));
}
```

The Application's Message Handler



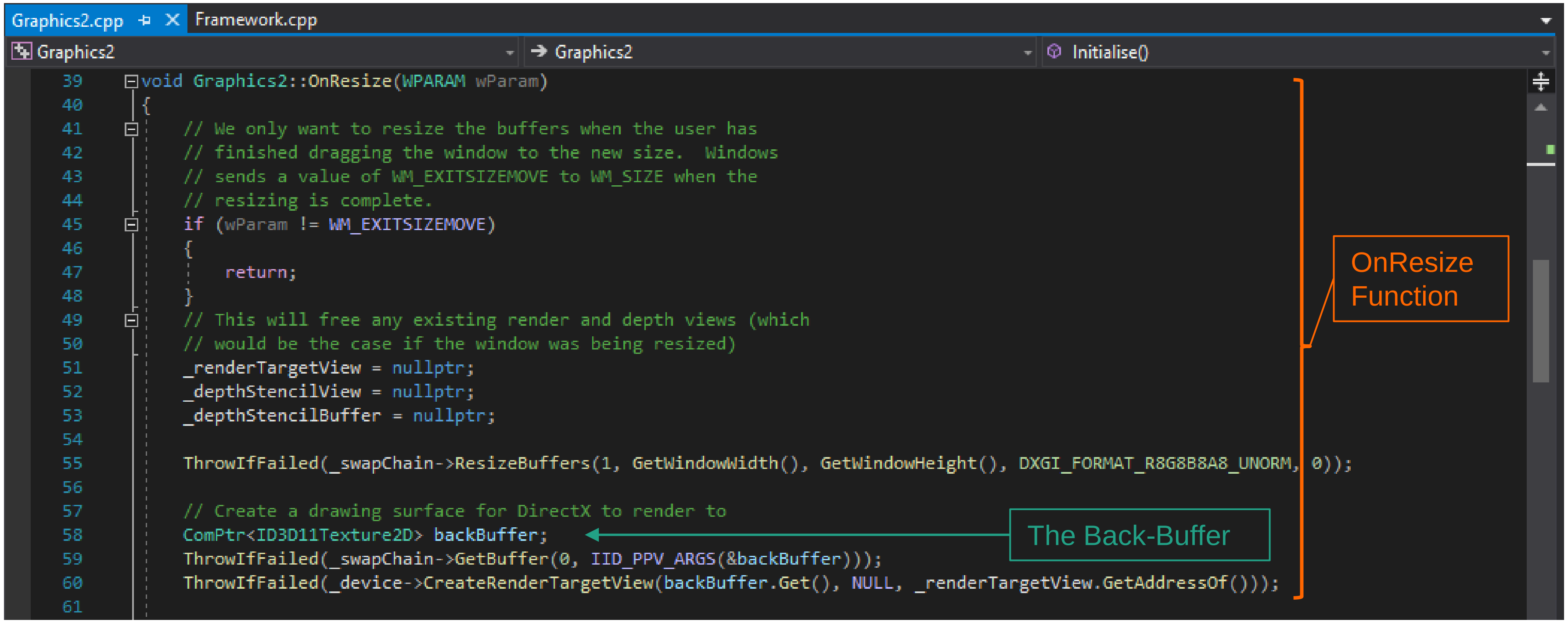
```
// The WndProc for the current window.  This cannot be a method, but we can
// redirect all messages to a method.

LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    if (_thisFramework != NULL)
    {
        // If framework is started, then we can call our own message proc
        return _thisFramework->MsgProc(hWnd, message, wParam, lParam);
    }
    else
    {
        // otherwise, we just pass control to the default message proc
        return DefWindowProc(hWnd, message, wParam, lParam);
    }
}
```

Our Message Handler



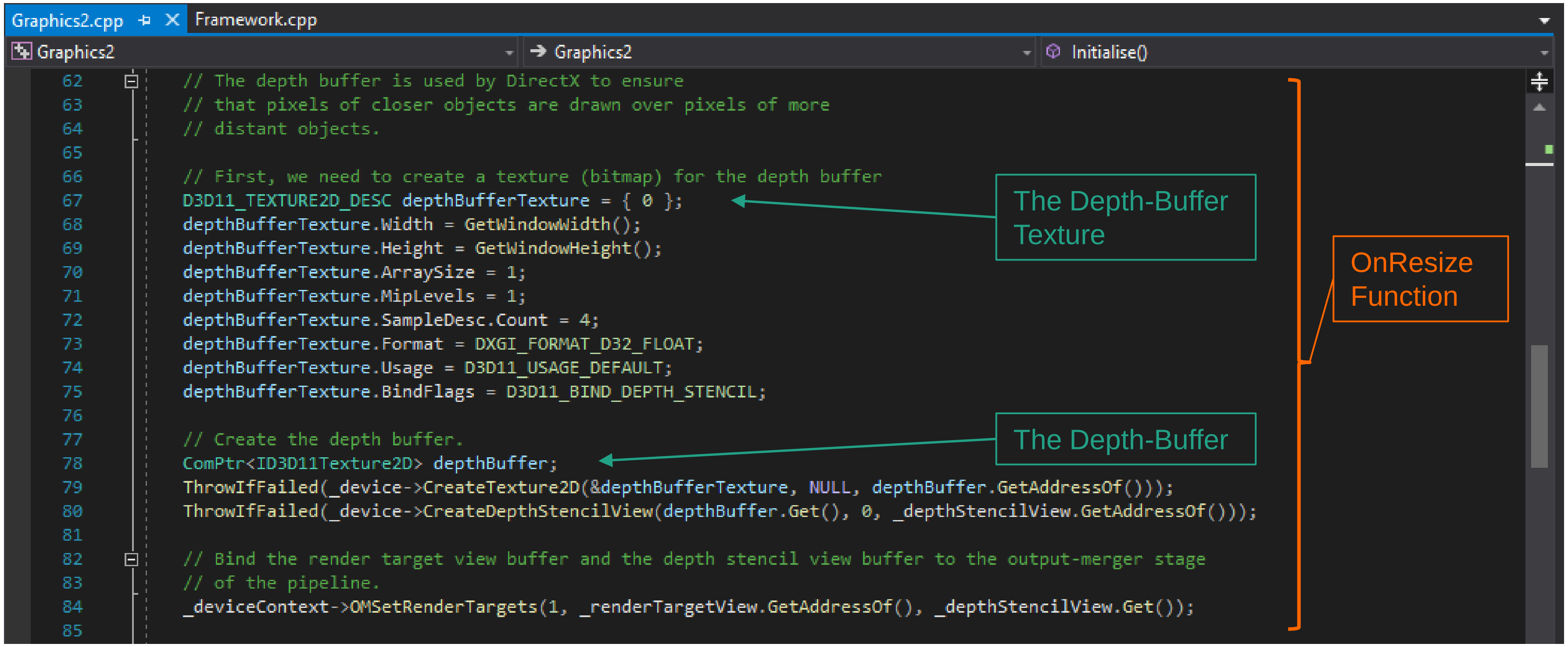
```
// Our main WndProc

LRESULT Framework::MsgProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
        case WM_PAINT:
            break;

        case WM_KEYDOWN:
            OnKeyDown(wParam);
            return 0;

        case WM_KEYUP:
            OnKeyUp(wParam);
            return 0;

        case WM_DESTROY:
            PostQuitMessage(0);
            break;

        case WM_SIZE:
            _width = LOWORD(lParam);
            _height = HIWORD(lParam);
            OnResize(wParam);
            Render();
            break;

        default:
            return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
}
```

Setting up the Back-Buffer



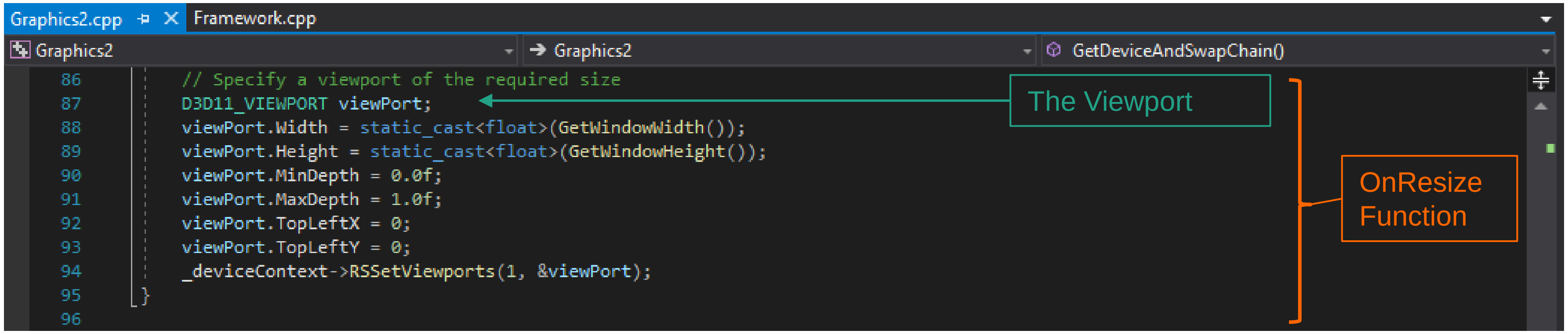
```
void Graphics2::OnResize(WPARAM wParam)
{
    // We only want to resize the buffers when the user has
    // finished dragging the window to the new size.  Windows
    // sends a value of WM_EXITSIZEMOVE to WM_SIZE when the
    // resizing is complete.
    if (wParam != WM_EXITSIZEMOVE)
    {
        return;
    }
    // This will free any existing render and depth views (which
    // would be the case if the window was being resized)
    _renderTargetView = nullptr;
    _depthStencilView = nullptr;
    _depthStencilBuffer = nullptr;

    ThrowIfFailed(_swapChain->ResizeBuffers(1, GetWindowWidth(), GetWindowHeight(), DXGI_FORMAT_R8G8B8A8_UNORM, 0));

    // Create a drawing surface for DirectX to render to
    ComPtr<ID3D11Texture2D> backBuffer;
    ThrowIfFailed(_swapChain->GetBuffer(0, IID_PPV_ARGS(&backBuffer)));
    ThrowIfFailed(_device->CreateRenderTargetView(backBuffer.Get(), NULL, _renderTargetView.GetAddressOf()));

```

Setting up the Depth-Buffer



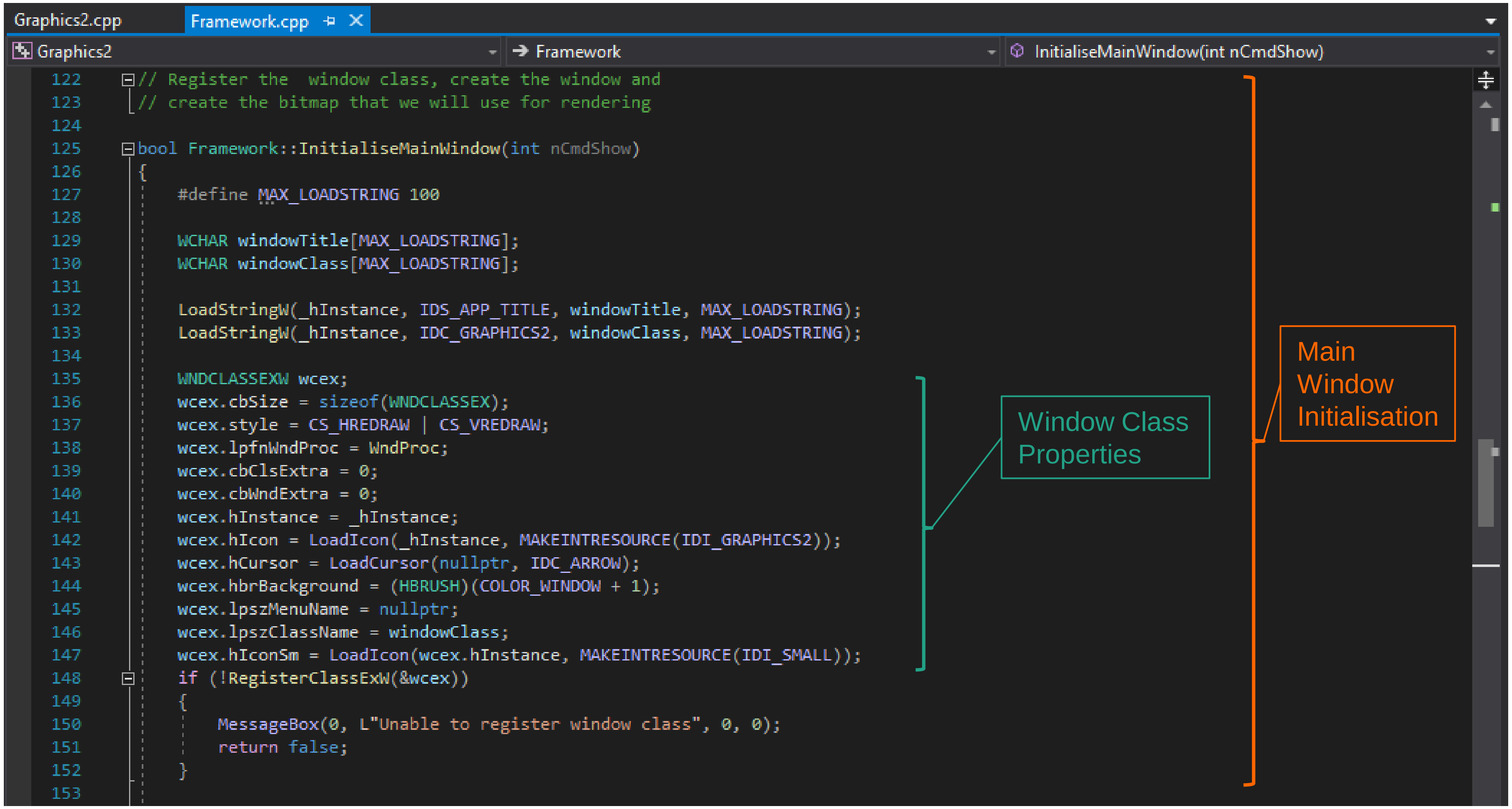
```
// The depth buffer is used by DirectX to ensure
// that pixels of closer objects are drawn over pixels of more
// distant objects.

// First, we need to create a texture (bitmap) for the depth buffer
D3D11_TEXTURE2D_DESC depthBufferTexture = { 0 };
depthBufferTexture.Width = GetWindowWidth();
depthBufferTexture.Height = GetWindowHeight();
depthBufferTexture.ArraySize = 1;
depthBufferTexture.MipLevels = 1;
depthBufferTexture.SampleDesc.Count = 4;
depthBufferTexture.Format = DXGI_FORMAT_D32_FLOAT;
depthBufferTexture.Usage = D3D11_USAGE_DEFAULT;
depthBufferTexture.BindFlags = D3D11_BIND_DEPTH_STENCIL;

// Create the depth buffer.
ComPtr<ID3D11Texture2D> depthBuffer;
ThrowIfFailed(_device->CreateTexture2D(&depthBufferTexture, NULL, depthBuffer.GetAddressOf()));
ThrowIfFailed(_device->CreateDepthStencilView(depthBuffer.Get(), 0, _depthStencilView.GetAddressOf()));

// Bind the render target view buffer and the depth stencil view buffer to the output-merger stage
// of the pipeline.
_deviceContext->OMSetRenderTargets(1, _renderTargetView.GetAddressOf(), _depthStencilView.Get());
```

Setting up the Viewport



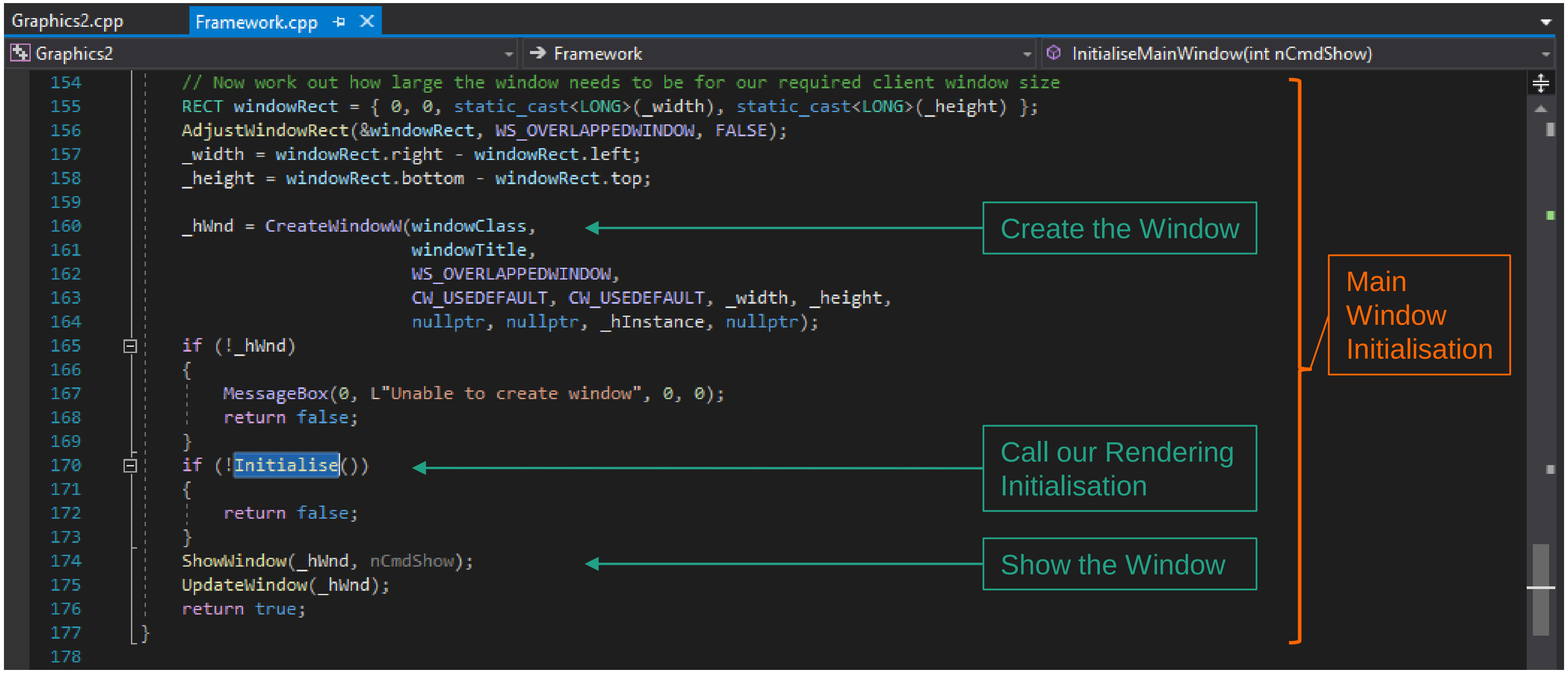
```
	// Specify a viewport of the required size
	D3D11_VIEWPORT viewPort;
	viewPort.Width = static_cast<float>(GetWindowWidth());
	viewPort.Height = static_cast<float>(GetWindowHeight());
	viewPort.MinDepth = 0.0f;
	viewPort.MaxDepth = 1.0f;
	viewPort.TopLeftX = 0;
	viewPort.TopLeftY = 0;
	_deviceContext->RSSetViewports(1, &viewPort);
}
```

Initialising the Application's Main Window



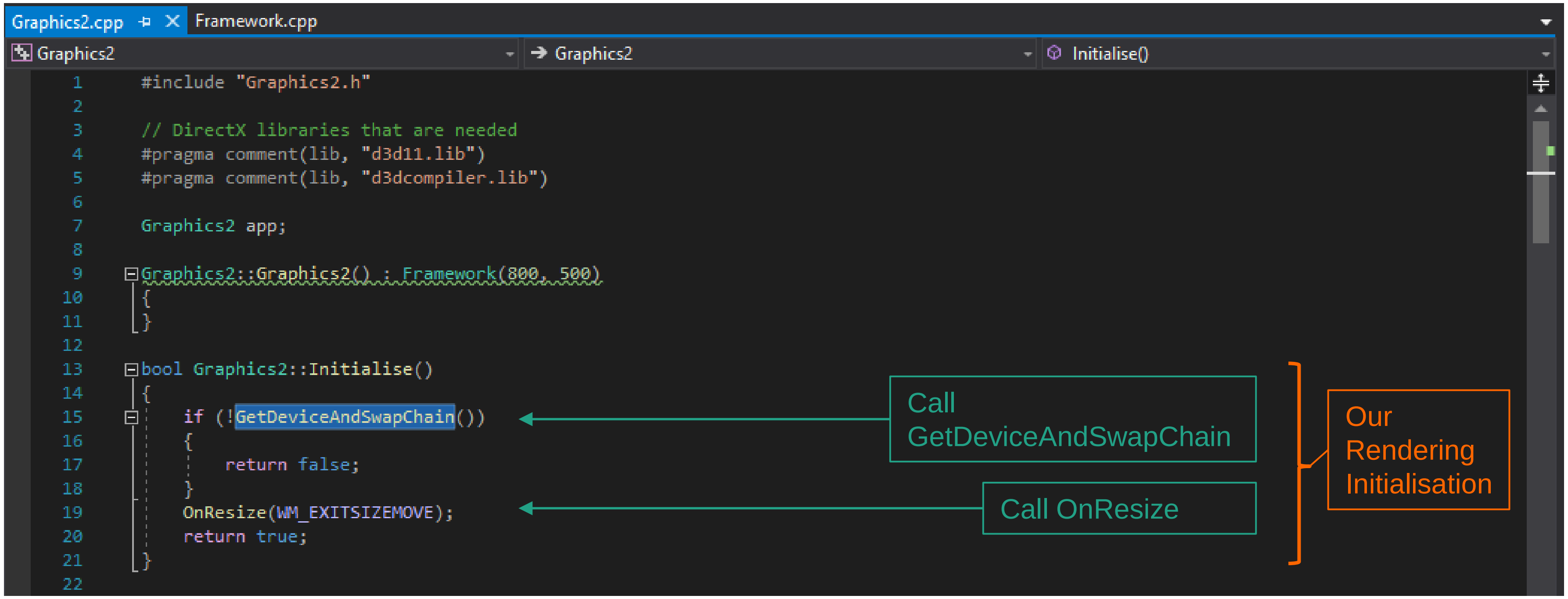
```
// Register the  window class, create the window and
// create the bitmap that we will use for rendering

bool Framework::InitialiseMainWindow(int nCmdShow)
{
    #define MAX_LOADSTRING 100

    WCHAR windowTitle[MAX_LOADSTRING];
    WCHAR windowClass[MAX_LOADSTRING];

    LoadStringW(_hInstance, IDS_APP_TITLE, windowTitle, MAX_LOADSTRING);
    LoadStringW(_hInstance, IDC_GRAPHICS2, windowClass, MAX_LOADSTRING);

    WNDCLASSEXW wcex;
    wcex.cbSize = sizeof(WNDCLASSEX);
    wcex.style = CS_HREDRAW | CS_VREDRAW;
    wcex.lpfnWndProc = WndProc;
    wcex.cbClsExtra = 0;
    wcex.cbWndExtra = 0;
    wcex.hInstance = _hInstance;
    wcex.hIcon = LoadIcon(_hInstance, MAKEINTRESOURCE(IDI_GRAPHICS2));
    wcex.hCursor = LoadCursor(nullptr, IDC_ARROW);
    wcex.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
    wcex.lpszMenuName = nullptr;
    wcex.lpszClassName = windowClass;
    wcex.hIconSm = LoadIcon(wcex.hInstance, MAKEINTRESOURCE(IDI_SMALL));
    if (!RegisterClassExW(&wcex))
    {
        MessageBox(0, L"Unable to register window class", 0, 0);
        return false;
    }

```

Initialising the Application's Main Window


Graphics2.cpp
Framework.cpp
Graphics2
Framework
InitialiseMainWindow(int nCmdShow)
Create the Window
Call our Rendering Initialisation
Show the Window
Main Window Initialisation

```
    // Now work out how large the window needs to be for our required client window size
    RECT windowRect = { 0, 0, static_cast<LONG>(_width), static_cast<LONG>(_height) };
    AdjustWindowRect(&windowRect, WS_OVERLAPPEDWINDOW, FALSE);
    _width = windowRect.right - windowRect.left;
    _height = windowRect.bottom - windowRect.top;

    _hWnd = CreateWindowW(windowClass,
                          windowTitle,
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT, _width, _height,
                          nullptr, nullptr, _hInstance, nullptr);
    if (!_hWnd)
    {
        MessageBox(0, L"Unable to create window", 0, 0);
        return false;
    }
    if (!Initialise())
    {
        return false;
    }
    ShowWindow(_hWnd, nCmdShow);
    UpdateWindow(_hWnd);
    return true;
}
```

Initialising the Rendering Application



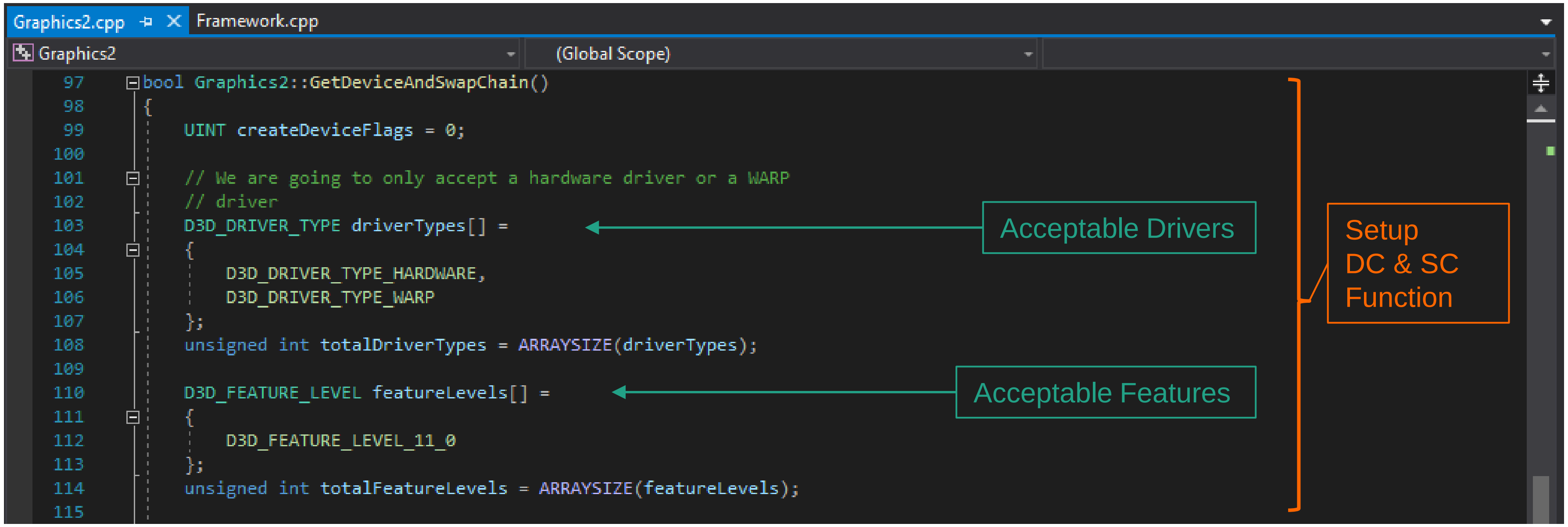
```cpp
#include "Graphics2.h"

// DirectX libraries that are needed
#pragma comment(lib, "d3d11.lib")
#pragma comment(lib, "d3dcompiler.lib")

Graphics2 app;

Graphics2::Graphics2() : Framework(800, 500)
{
}

bool Graphics2::Initialise()
{
    if (!GetDeviceAndSwapChain())
    {
        return false;
    }
    OnResize(WM_EXITSIZEMOVE);
    return true;
}
```

Call GetDeviceAndSwapChain

Call OnResize

Our Rendering Initialisation

Setting up the Device-Context and Swap-Chain

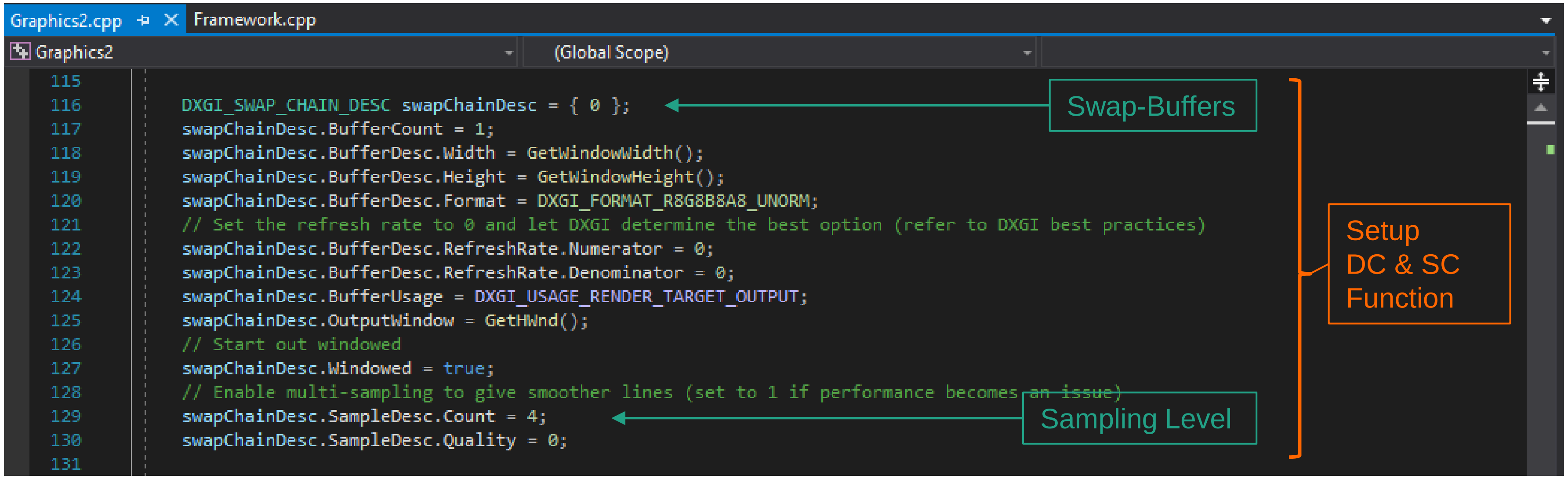


Graphics2.cpp   Framework.cpp

Graphics2   (Global Scope)

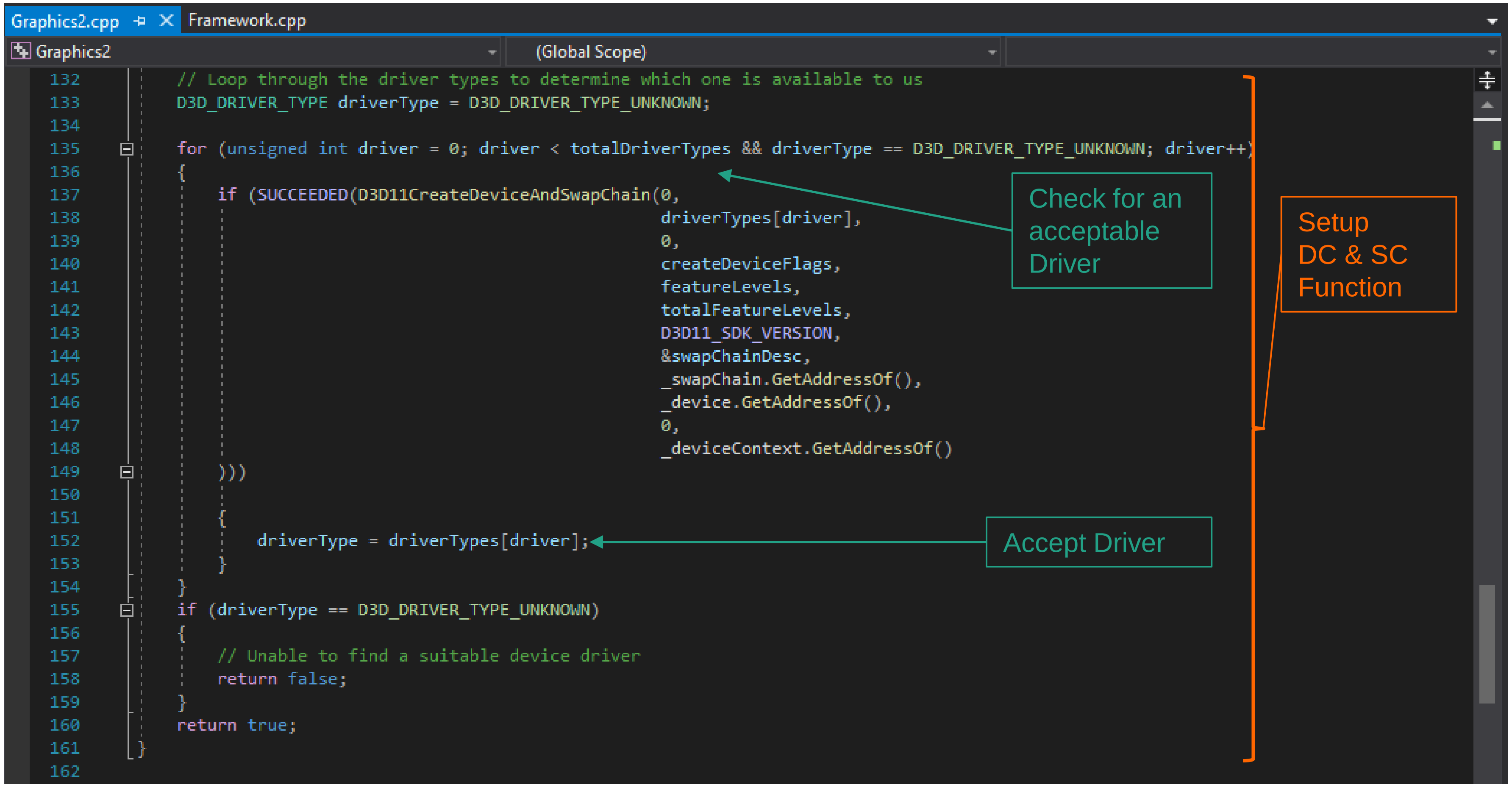
```cpp
bool Graphics2::GetDeviceAndSwapChain()
{
    UINT createDeviceFlags = 0;

    // We are going to only accept a hardware driver or a WARP
    // driver
    D3D_DRIVER_TYPE driverTypes[] =
    {
        D3D_DRIVER_TYPE_HARDWARE,
        D3D_DRIVER_TYPE_WARP
    };
    unsigned int totalDriverTypes = ARRAYSIZE(driverTypes);

    D3D_FEATURE_LEVEL featureLevels[] =
    {
        D3D_FEATURE_LEVEL_11_0
    };
    unsigned int totalFeatureLevels = ARRAYSIZE(featureLevels);

```

Setting up the Swap-Buffer and Multi-Sampling level



```
    DXGI_SWAP_CHAIN_DESC swapChainDesc = { 0 };
    swapChainDesc.BufferCount = 1;
    swapChainDesc.BufferDesc.Width = GetWindowWidth();
    swapChainDesc.BufferDesc.Height = GetWindowHeight();
    swapChainDesc.BufferDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
    // Set the refresh rate to 0 and let DXGI determine the best option (refer to DXGI best practices)
    swapChainDesc.BufferDesc.RefreshRate.Numerator = 0;
    swapChainDesc.BufferDesc.RefreshRate.Denominator = 0;
    swapChainDesc.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;
    swapChainDesc.OutputWindow = GetHWnd();
    // Start out windowed
    swapChainDesc.Windowed = true;
    // Enable multi-sampling to give smoother lines (set to 1 if performance becomes an issue)
    swapChainDesc.SampleDesc.Count = 4;
    swapChainDesc.SampleDesc.Quality = 0;
```

Finding an acceptable Driver



```
    // Loop through the driver types to determine which one is available to us
    D3D_DRIVER_TYPE driverType = D3D_DRIVER_TYPE_UNKNOWN;

    for (unsigned int driver = 0; driver < totalDriverTypes && driverType == D3D_DRIVER_TYPE_UNKNOWN; driver++)
    {
        if (SUCCEEDED(D3D11CreateDeviceAndSwapChain(0,
                                                    driverTypes[driver],
                                                    0,
                                                    createDeviceFlags,
                                                    featureLevels,
                                                    totalFeatureLevels,
                                                    D3D11_SDK_VERSION,
                                                    &swapChainDesc,
                                                    _swapChain.GetAddressOf(),
                                                    _device.GetAddressOf(),
                                                    0,
                                                    _deviceContext.GetAddressOf()
        )))

        {
            driverType = driverTypes[driver];
        }
    }
    if (driverType == D3D_DRIVER_TYPE_UNKNOWN)
    {
        // Unable to find a suitable device driver
        return false;
    }
    return true;
}
```