

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

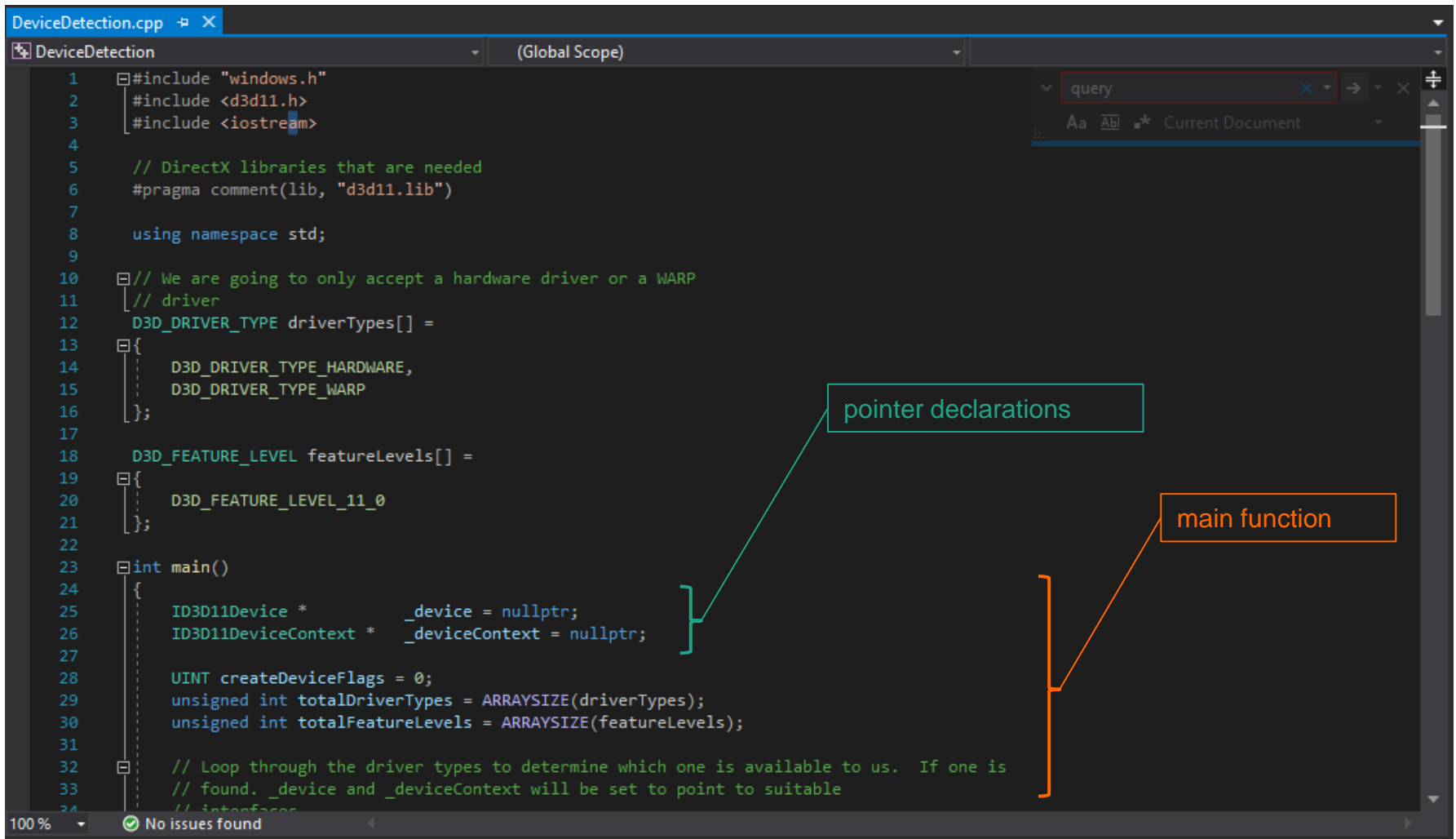
COM (Component Object Model)

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

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



```
1 #include "windows.h"
2 #include <d3d11.h>
3 #include <iostream>
4
5 // DirectX libraries that are needed
6 #pragma comment(lib, "d3d11.lib")
7
8 using namespace std;
9
10 // We are going to only accept a hardware driver or a WARP
11 // driver
12 D3D_DRIVER_TYPE driverTypes[] =
13 {
14     D3D_DRIVER_TYPE_HARDWARE,
15     D3D_DRIVER_TYPE_WARP
16 };
17
18 D3D_FEATURE_LEVEL featureLevels[] =
19 {
20     D3D_FEATURE_LEVEL_11_0
21 };
22
23 int main()
24 {
25     ID3D11Device * _device = nullptr;
26     ID3D11DeviceContext * _deviceContext = nullptr;
27
28     UINT createDeviceFlags = 0;
29     unsigned int totalDriverTypes = ARRAYSIZE(driverTypes);
30     unsigned int totalFeatureLevels = ARRAYSIZE(featureLevels);
31
32     // Loop through the driver types to determine which one is available to us. If one is
33     // found, _device and _deviceContext will be set to point to suitable
34     // interfaces
```

Annotations:

- pointer declarations (points to lines 25-26)
- main function (points to the main function block)

COM (Component Object Model)

- Once we have a 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  X
DeviceDetection      (Global Scope)

1  #include "windows.h"
2  #include <d3d11.h>
3  #include <wrl.h>
4  #include <iostream>
5
6  // DirectX libraries that are needed
7  #pragma comment(lib, "d3d11.lib")
8
9  using namespace std;
10
11  using Microsoft::WRL::ComPtr;
12
13  // We are going to only accept a hardware driver or a WARP
14  // driver
15  D3D_DRIVER_TYPE driverTypes[] =
16  {
17      D3D_DRIVER_TYPE_HARDWARE,
18      D3D_DRIVER_TYPE_WARP
19  };
20
21  D3D_FEATURE_LEVEL featureLevels[] =
22  {
23      D3D_FEATURE_LEVEL_11_0
24  };
25
26  int main()
27  {
28      ComPtr<ID3D11Device> _device;
29      ComPtr<ID3D11DeviceContext> _deviceContext;
30
31      UINT createDeviceFlags = 0;
32      unsigned int totalDriverTypes = ARRAYSIZE(driverTypes);
33      unsigned int totalFeatureLevels = ARRAYSIZE(featureLevels);
34

```

using ComPtr

ComPtr pointer declarations

main function

100 % No issues found

“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.

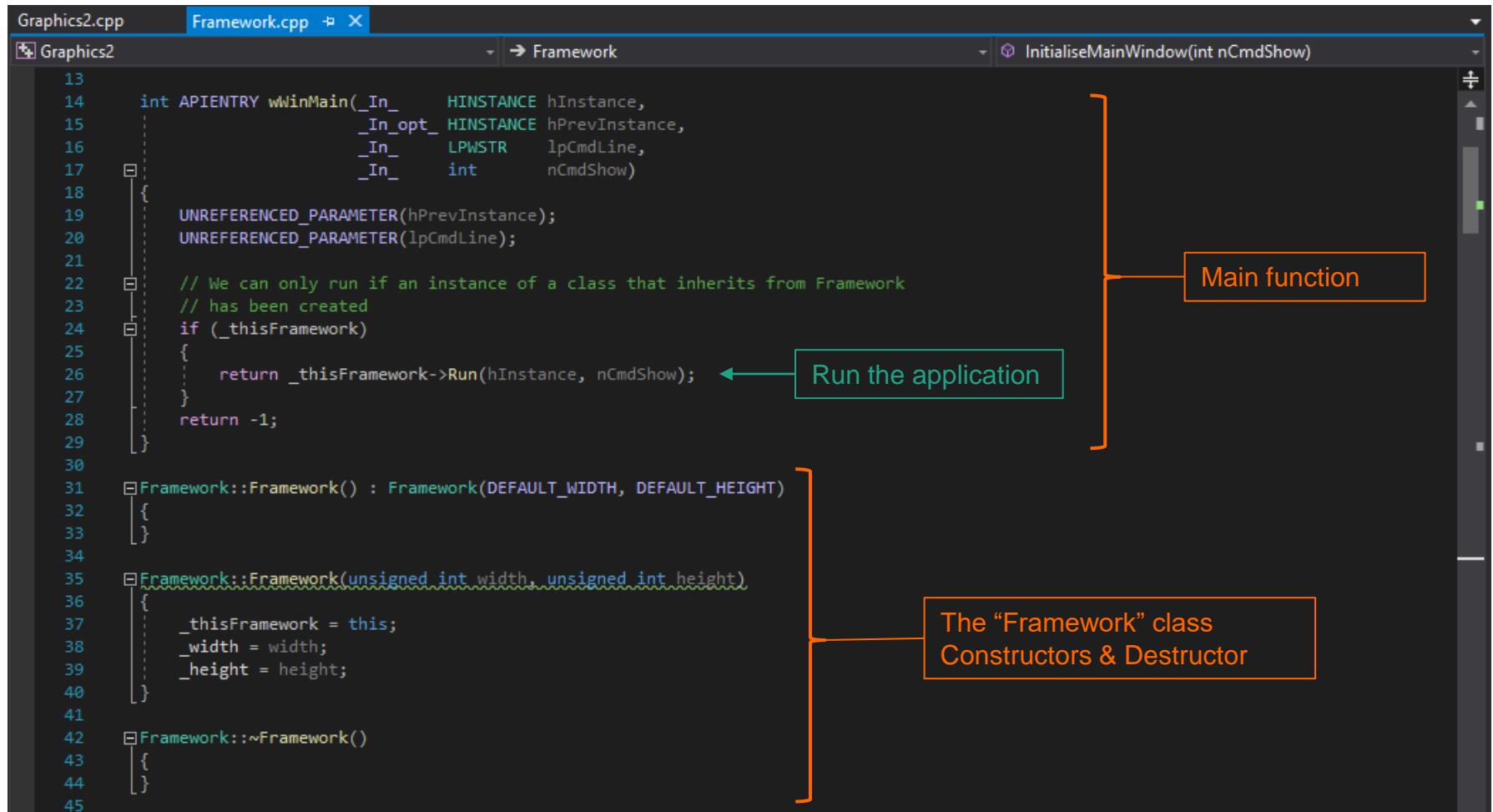
Main Driver Types

- **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



The screenshot shows a C++ IDE with two tabs: `Graphics2.cpp` and `Framework.cpp`. The `Framework.cpp` tab is active, showing the `Framework` class and its `wWinMain` function. The `wWinMain` function is annotated as the "Main function". The `Framework` class constructors and destructor are annotated as "The 'Framework' class Constructors & Destructor". A specific line in the `wWinMain` function, `return _thisFramework->Run(hInstance, nCmdShow);`, is highlighted with a green box and labeled "Run the application".

```

13
14 int APIENTRY wWinMain(_In_ HINSTANCE hInstance,
15                        _In_opt_ HINSTANCE hPrevInstance,
16                        _In_ LPWSTR lpCmdLine,
17                        _In_ int nCmdShow)
18 {
19     UNREFERENCED_PARAMETER(hPrevInstance);
20     UNREFERENCED_PARAMETER(lpCmdLine);
21
22     // We can only run if an instance of a class that inherits from Framework
23     // has been created
24     if (_thisFramework)
25     {
26         return _thisFramework->Run(hInstance, nCmdShow);
27     }
28     return -1;
29 }
30
31 Framework::Framework() : Framework(DEFAULT_WIDTH, DEFAULT_HEIGHT)
32 {
33 }
34
35 Framework::Framework(unsigned int width, unsigned int height)
36 {
37     _thisFramework = this;
38     _width = width;
39     _height = height;
40 }
41
42 Framework::~Framework()
43 {
44 }
45

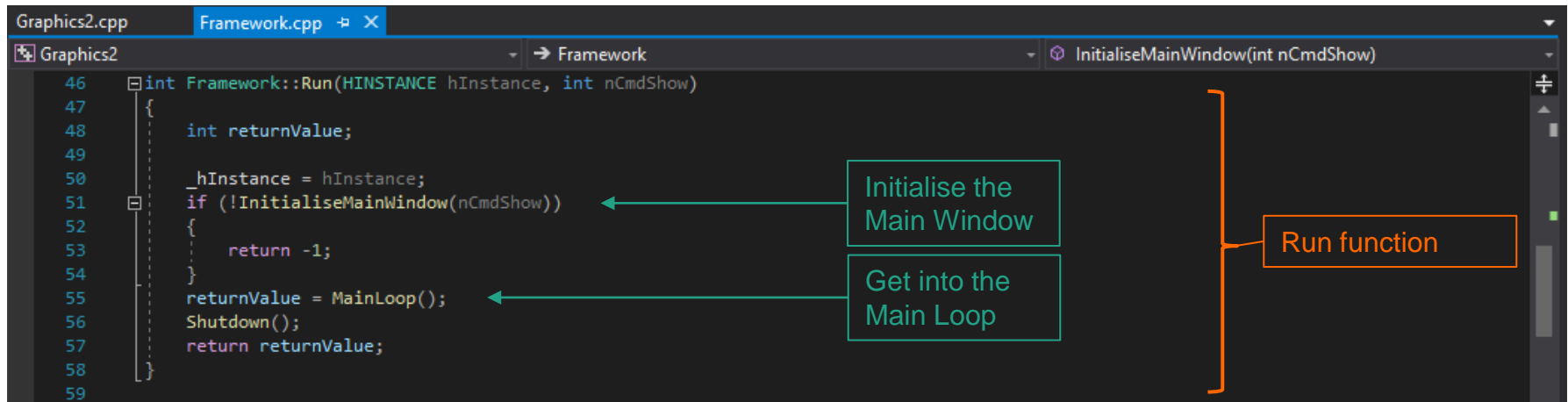
```

Main function

Run the application

The "Framework" class Constructors & Destructor

Running the Application

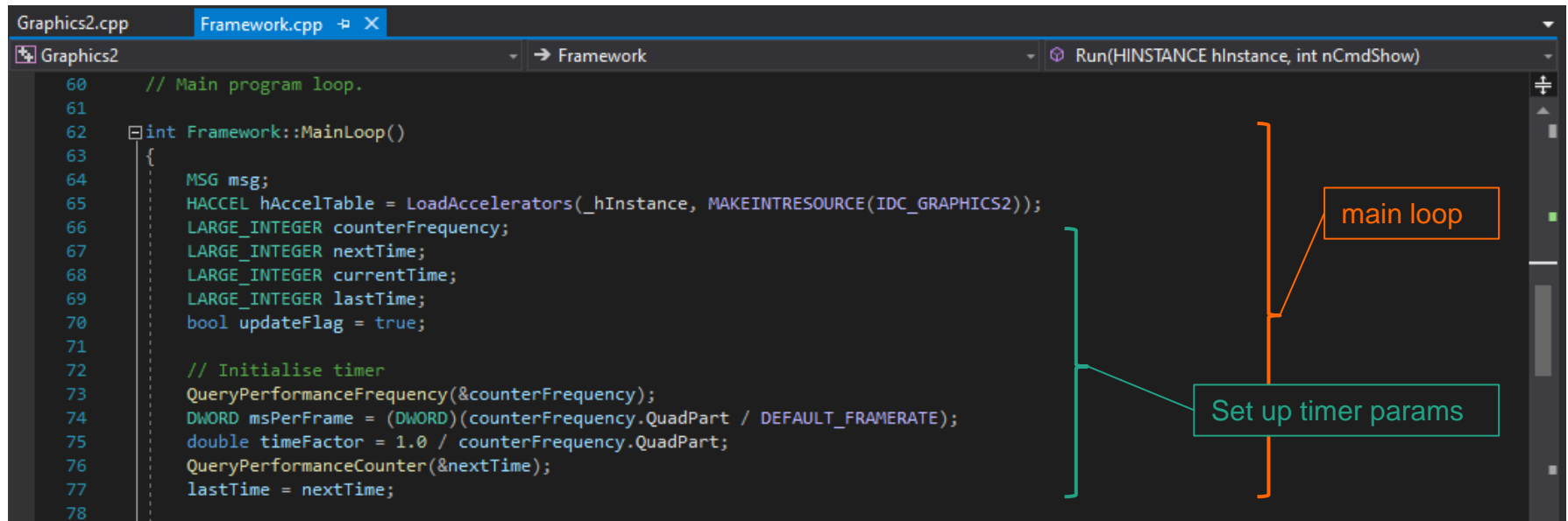


```
46 int Framework::Run(HINSTANCE hInstance, int nCmdShow)
47 {
48     int returnValue;
49
50     _hInstance = hInstance;
51     if (!InitialiseMainWindow(nCmdShow))
52     {
53         return -1;
54     }
55     returnValue = MainLoop();
56     Shutdown();
57     return returnValue;
58 }
59
```

Annotations:

- Initialise the Main Window**: Points to the `InitialiseMainWindow(nCmdShow)` call on line 51.
- Get into the Main Loop**: Points to the `MainLoop()` call on line 55.
- Run function**: A bracket on the right side of the code block, spanning from line 46 to line 58, indicating the entire `Run` function.

The Application's Main-Loop



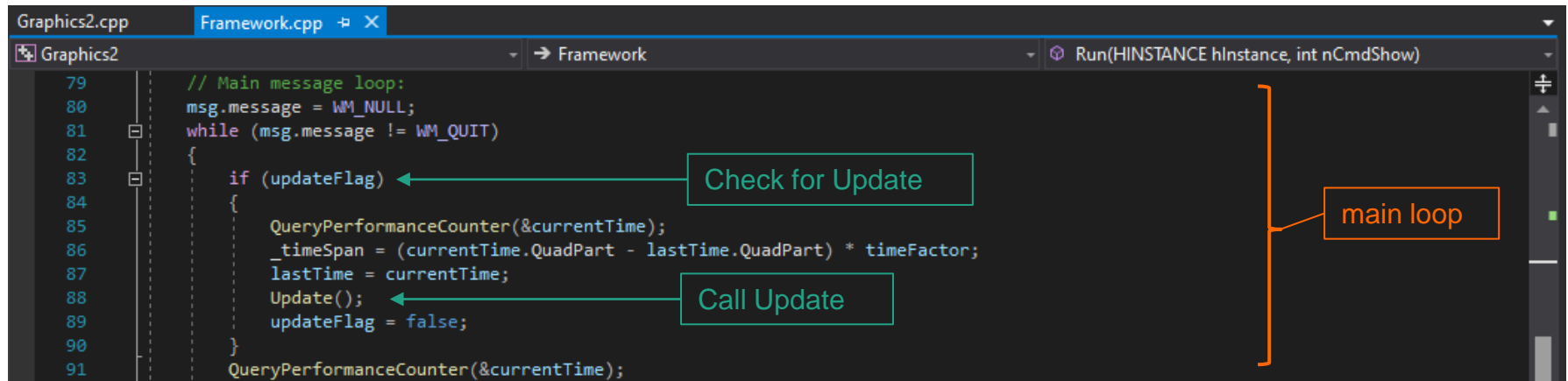
```
Graphics2.cpp | Framework.cpp [X]
Graphics2 | Framework | Run(HINSTANCE hInstance, int nCmdShow)

60 // Main program loop.
61
62 int Framework::MainLoop()
63 {
64     MSG msg;
65     HACCEL hAccelTable = LoadAccelerators(_hInstance, MAKEINTRESOURCE(IDC_GRAPHICS2));
66     LARGE_INTEGER counterFrequency;
67     LARGE_INTEGER nextTime;
68     LARGE_INTEGER currentTime;
69     LARGE_INTEGER lastTime;
70     bool updateFlag = true;
71
72     // Initialise timer
73     QueryPerformanceFrequency(&counterFrequency);
74     DWORD msPerFrame = (DWORD)(counterFrequency.QuadPart / DEFAULT_FRAMERATE);
75     double timeFactor = 1.0 / counterFrequency.QuadPart;
76     QueryPerformanceCounter(&nextTime);
77     lastTime = nextTime;
78 }
```

main loop

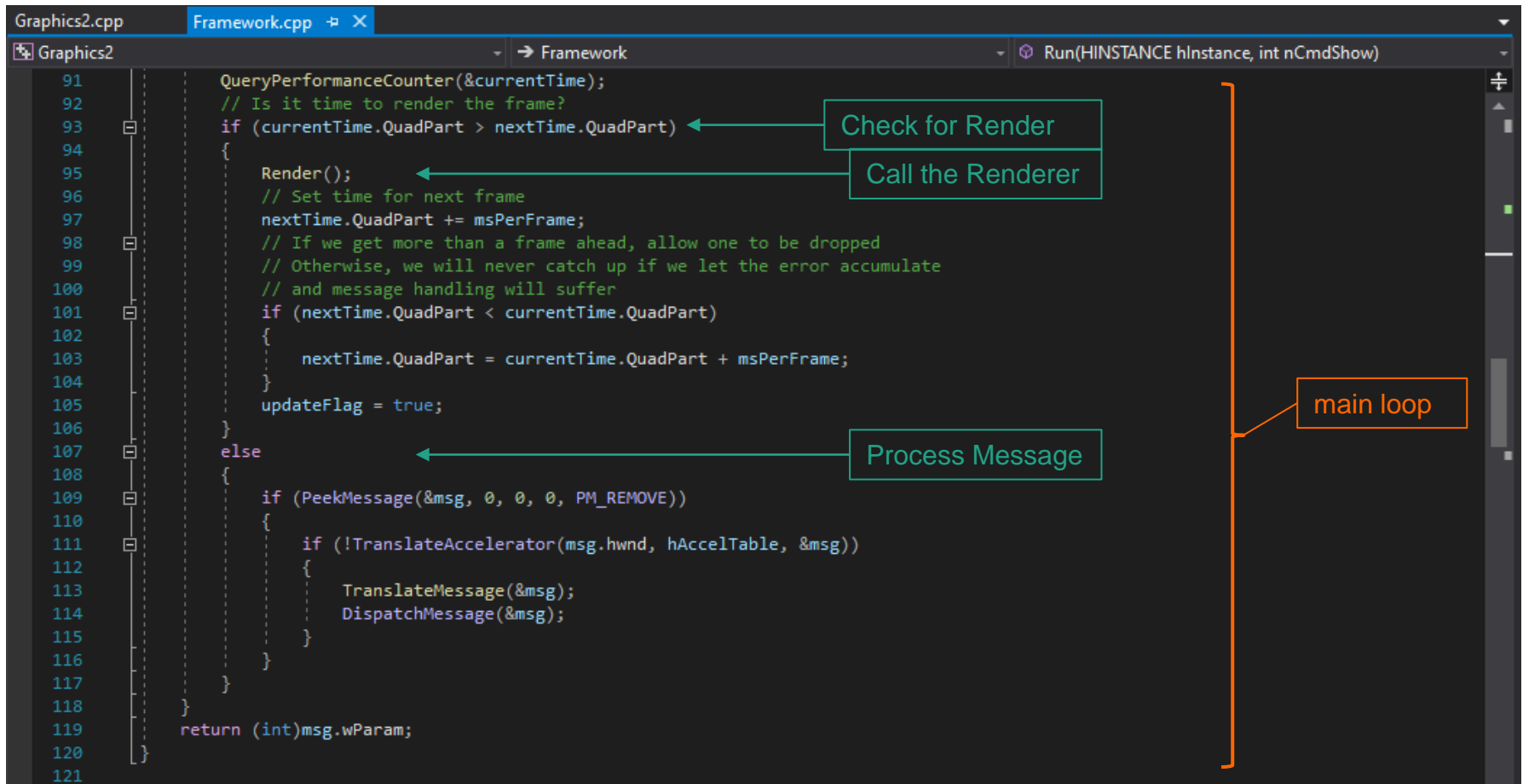
Set up timer params

The Application's Main-Loop



```
79 // Main message loop:
80 msg.message = WM_NULL;
81 while (msg.message != WM_QUIT)
82 {
83     if (updateFlag) ← Check for Update
84     {
85         QueryPerformanceCounter(&currentTime);
86         _timeSpan = (currentTime.QuadPart - lastTime.QuadPart) * timeFactor;
87         lastTime = currentTime;
88         Update(); ← Call Update
89         updateFlag = false;
90     }
91     QueryPerformanceCounter(&currentTime);
```

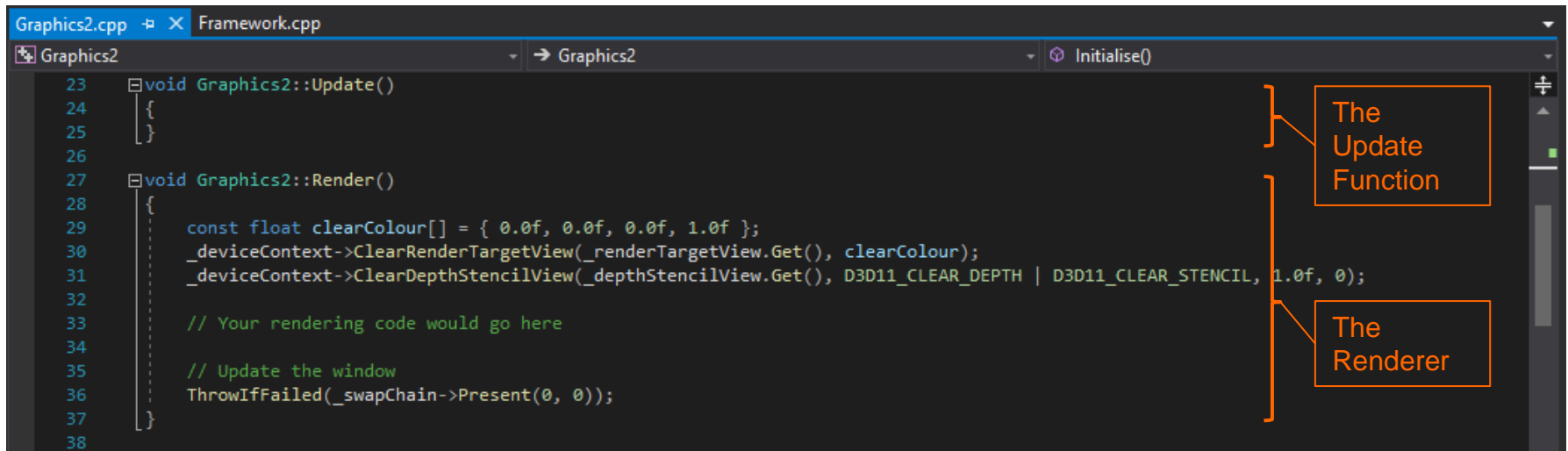

The Application's Main-Loop



```
Graphics2.cpp Framework.cpp → X
Graphics2 → Framework Run(HINSTANCE hInstance, int nCmdShow)

91 QueryPerformanceCounter(&currentTime);
92 // Is it time to render the frame?
93 if (currentTime.QuadPart > nextTime.QuadPart) ← Check for Render
94 {
95     Render(); ← Call the Renderer
96     // Set time for next frame
97     nextTime.QuadPart += msPerFrame;
98     // If we get more than a frame ahead, allow one to be dropped
99     // Otherwise, we will never catch up if we let the error accumulate
100    // and message handling will suffer
101    if (nextTime.QuadPart < currentTime.QuadPart)
102    {
103        nextTime.QuadPart = currentTime.QuadPart + msPerFrame;
104    }
105    updateFlag = true;
106 }
107 else ← Process Message
108 {
109     if (PeekMessage(&msg, 0, 0, 0, PM_REMOVE))
110     {
111         if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
112         {
113             TranslateMessage(&msg);
114             DispatchMessage(&msg);
115         }
116     }
117 }
118 }
119 return (int)msg.wParam;
120 }
121 }
```

The Update and Render Functions



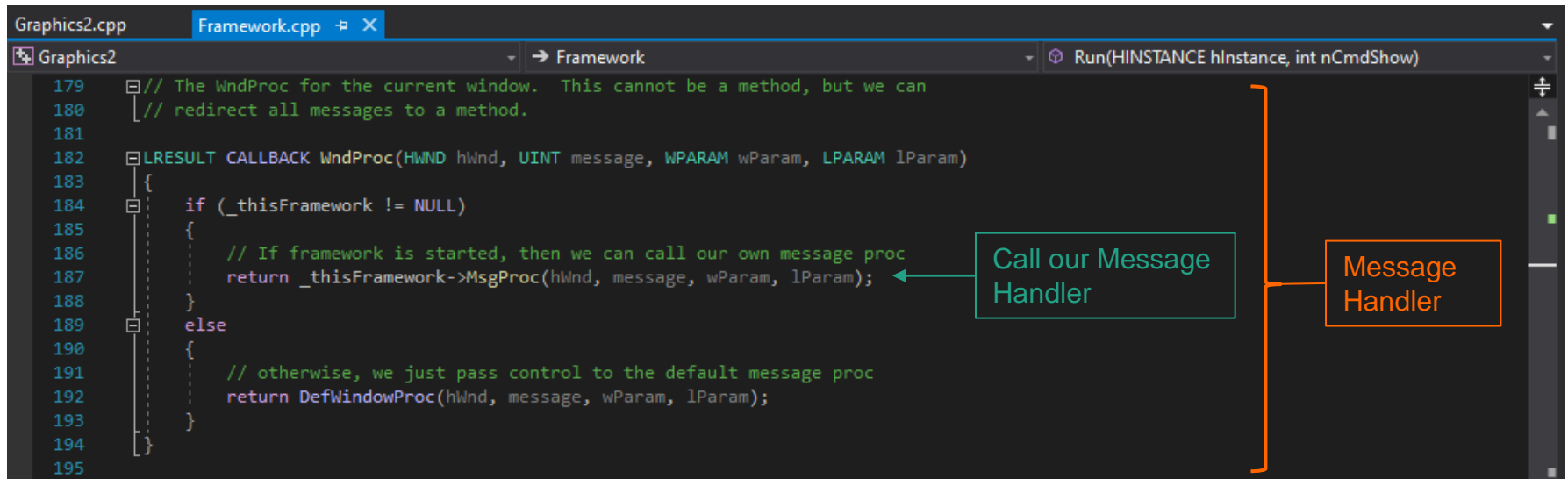
```
Graphics2.cpp Framework.cpp
Graphics2 → Graphics2 Initialise()

23 void Graphics2::Update()
24 {
25 }
26
27 void Graphics2::Render()
28 {
29     const float clearColour[] = { 0.0f, 0.0f, 0.0f, 1.0f };
30     _deviceContext->ClearRenderTargetView(_renderTargetView.Get(), clearColour);
31     _deviceContext->ClearDepthStencilView(_depthStencilView.Get(), D3D11_CLEAR_DEPTH | D3D11_CLEAR_STENCIL, 1.0f, 0);
32
33     // Your rendering code would go here
34
35     // Update the window
36     ThrowIfFailed(_swapChain->Present(0, 0));
37 }
38
```

The Update Function

The Renderer

The Application's Message Handler

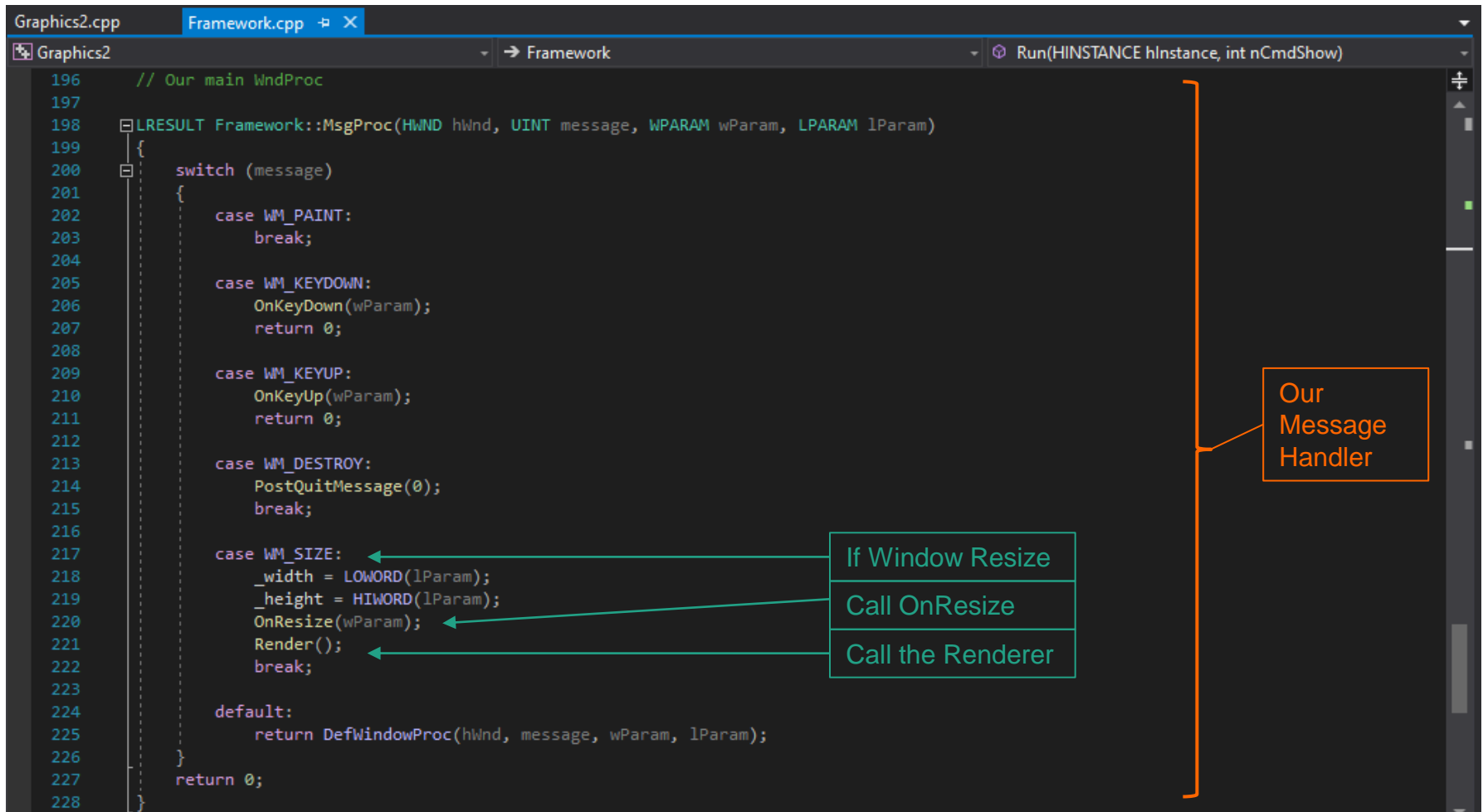


```
Graphics2.cpp Framework.cpp X
Graphics2 Framework Run(HINSTANCE hInstance, int nCmdShow)
179 // The WndProc for the current window. This cannot be a method, but we can
180 // redirect all messages to a method.
181
182 LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
183 {
184     if (_thisFramework != NULL)
185     {
186         // If framework is started, then we can call our own message proc
187         return _thisFramework->MsgProc(hWnd, message, wParam, lParam);
188     }
189     else
190     {
191         // otherwise, we just pass control to the default message proc
192         return DefWindowProc(hWnd, message, wParam, lParam);
193     }
194 }
195
```

Call our Message Handler

Message Handler

Our Message Handler



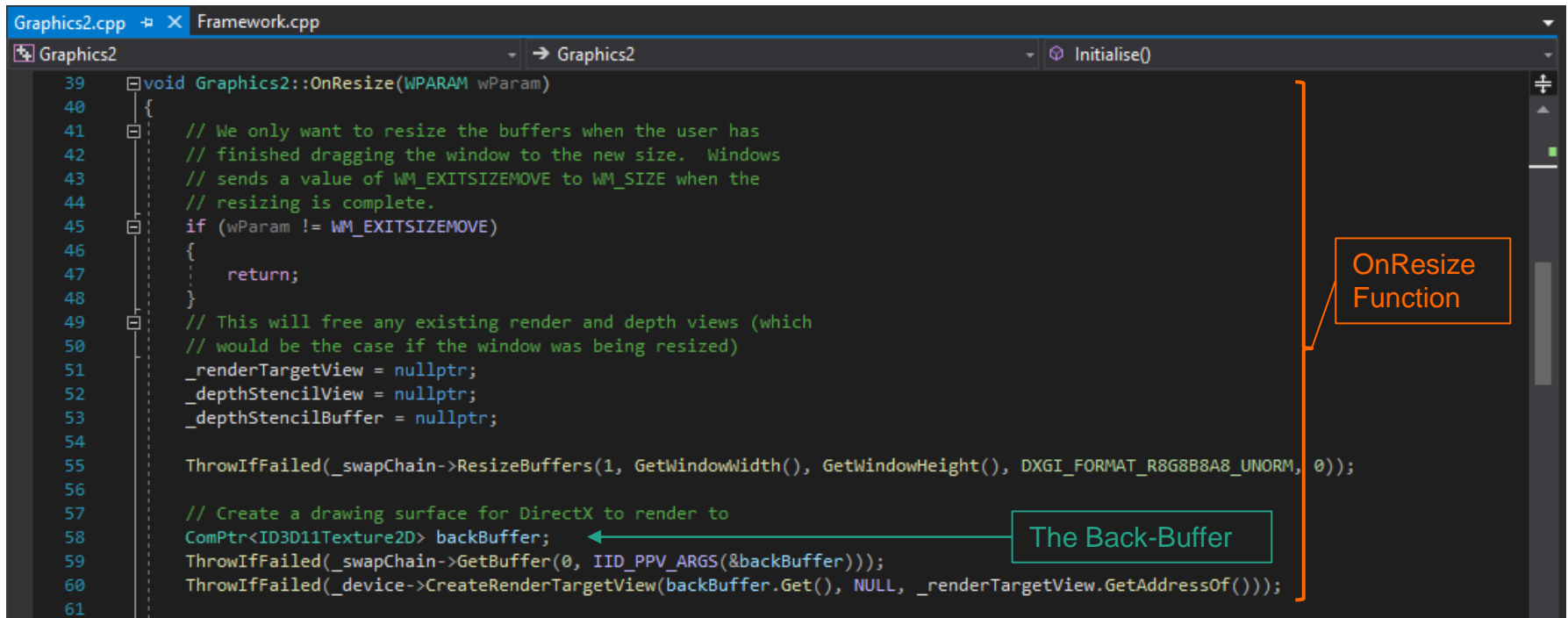
The screenshot shows a C++ IDE with a file named `Framework.cpp` open. The code defines a message handler function `Framework::MsgProc` which uses a `switch` statement to process different Windows messages. Annotations highlight the `WM_SIZE` case and the overall function.

```
196 // Our main WndProc
197
198 LRESULT Framework::MsgProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
199 {
200     switch (message)
201     {
202         case WM_PAINT:
203             break;
204
205         case WM_KEYDOWN:
206             OnKeyDown(wParam);
207             return 0;
208
209         case WM_KEYUP:
210             OnKeyUp(wParam);
211             return 0;
212
213         case WM_DESTROY:
214             PostQuitMessage(0);
215             break;
216
217         case WM_SIZE:
218             _width = LOWORD(lParam);
219             _height = HIWORD(lParam);
220             OnResize(wParam);
221             Render();
222             break;
223
224         default:
225             return DefWindowProc(hWnd, message, wParam, lParam);
226     }
227     return 0;
228 }
```

Our Message Handler

If Window Resize
Call OnResize
Call the Renderer

Setting up the Back-Buffer



```
39 void Graphics2::OnResize(WPARAM wParam)
40 {
41     // We only want to resize the buffers when the user has
42     // finished dragging the window to the new size. Windows
43     // sends a value of WM_EXITSIZEMOVE to WM_SIZE when the
44     // resizing is complete.
45     if (wParam != WM_EXITSIZEMOVE)
46     {
47         return;
48     }
49     // This will free any existing render and depth views (which
50     // would be the case if the window was being resized)
51     _renderTargetView = nullptr;
52     _depthStencilView = nullptr;
53     _depthStencilBuffer = nullptr;
54
55     ThrowIfFailed(_swapChain->ResizeBuffers(1, GetWindowWidth(), GetWindowHeight(), DXGI_FORMAT_R8G8B8A8_UNORM, 0));
56
57     // Create a drawing surface for DirectX to render to
58     ComPtr<ID3D11Texture2D> backBuffer;
59     ThrowIfFailed(_swapChain->GetBuffer(0, IID_PPV_ARGS(&backBuffer)));
60     ThrowIfFailed(_device->CreateRenderTargetView(backBuffer.Get(), NULL, _renderTargetView.GetAddressOf()));
61 }
```

OnResize Function

The Back-Buffer

Setting up the Depth-Buffer

```
Graphics2.cpp Framework.cpp
Graphics2 → Graphics2 Initialise()

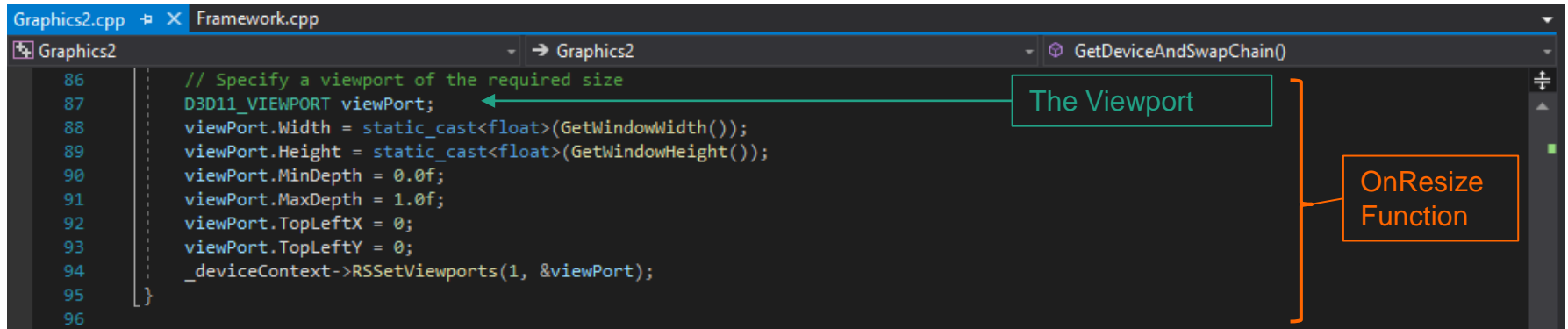
62 // The depth buffer is used by DirectX to ensure
63 // that pixels of closer objects are drawn over pixels of more
64 // distant objects.
65
66 // First, we need to create a texture (bitmap) for the depth buffer
67 D3D11_TEXTURE2D_DESC depthBufferTexture = { 0 };
68 depthBufferTexture.Width = GetWindowWidth();
69 depthBufferTexture.Height = GetWindowHeight();
70 depthBufferTexture.ArraySize = 1;
71 depthBufferTexture.MipLevels = 1;
72 depthBufferTexture.SampleDesc.Count = 4;
73 depthBufferTexture.Format = DXGI_FORMAT_D32_FLOAT;
74 depthBufferTexture.Usage = D3D11_USAGE_DEFAULT;
75 depthBufferTexture.BindFlags = D3D11_BIND_DEPTH_STENCIL;
76
77 // Create the depth buffer.
78 ComPtr<ID3D11Texture2D> depthBuffer;
79 ThrowIfFailed(_device->CreateTexture2D(&depthBufferTexture, NULL, depthBuffer.GetAddressOf()));
80 ThrowIfFailed(_device->CreateDepthStencilView(depthBuffer.Get(), 0, _depthStencilView.GetAddressOf()));
81
82 // Bind the render target view buffer and the depth stencil view buffer to the output-merger stage
83 // of the pipeline.
84 _deviceContext->OMSetRenderTargets(1, _renderTargetView.GetAddressOf(), _depthStencilView.Get());
85
```

The Depth-Buffer Texture

The Depth-Buffer

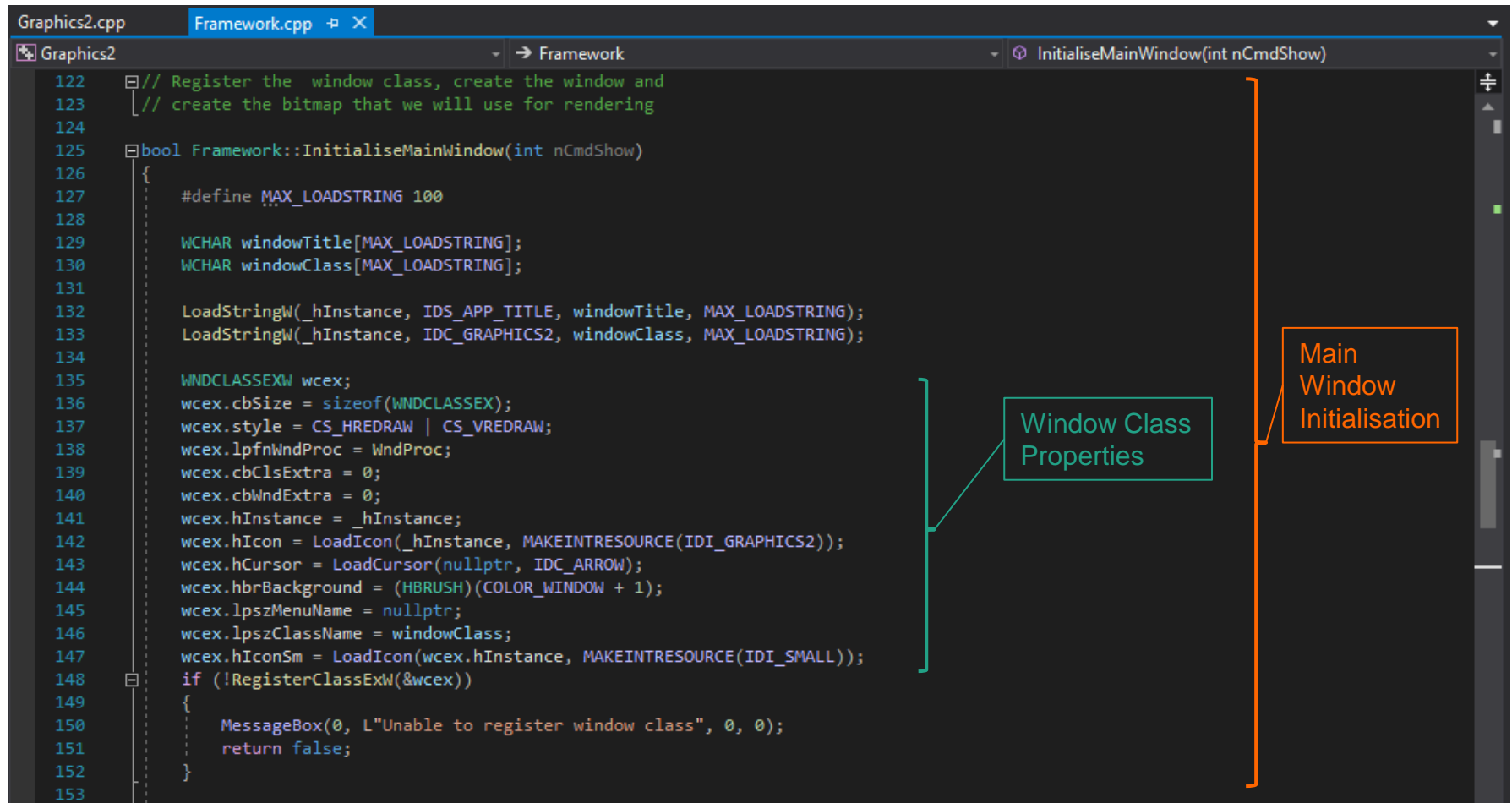
OnResize Function

Setting up the Viewport



```
86 // Specify a viewport of the required size
87 D3D11_VIEWPORT viewport;
88 viewport.Width = static_cast<float>(GetWindowWidth());
89 viewport.Height = static_cast<float>(GetWindowHeight());
90 viewport.MinDepth = 0.0f;
91 viewport.MaxDepth = 1.0f;
92 viewport.TopLeftX = 0;
93 viewport.TopLeftY = 0;
94 _deviceContext->RSSetViewports(1, &viewport);
95 }
```

Initialising the Application's Main Window

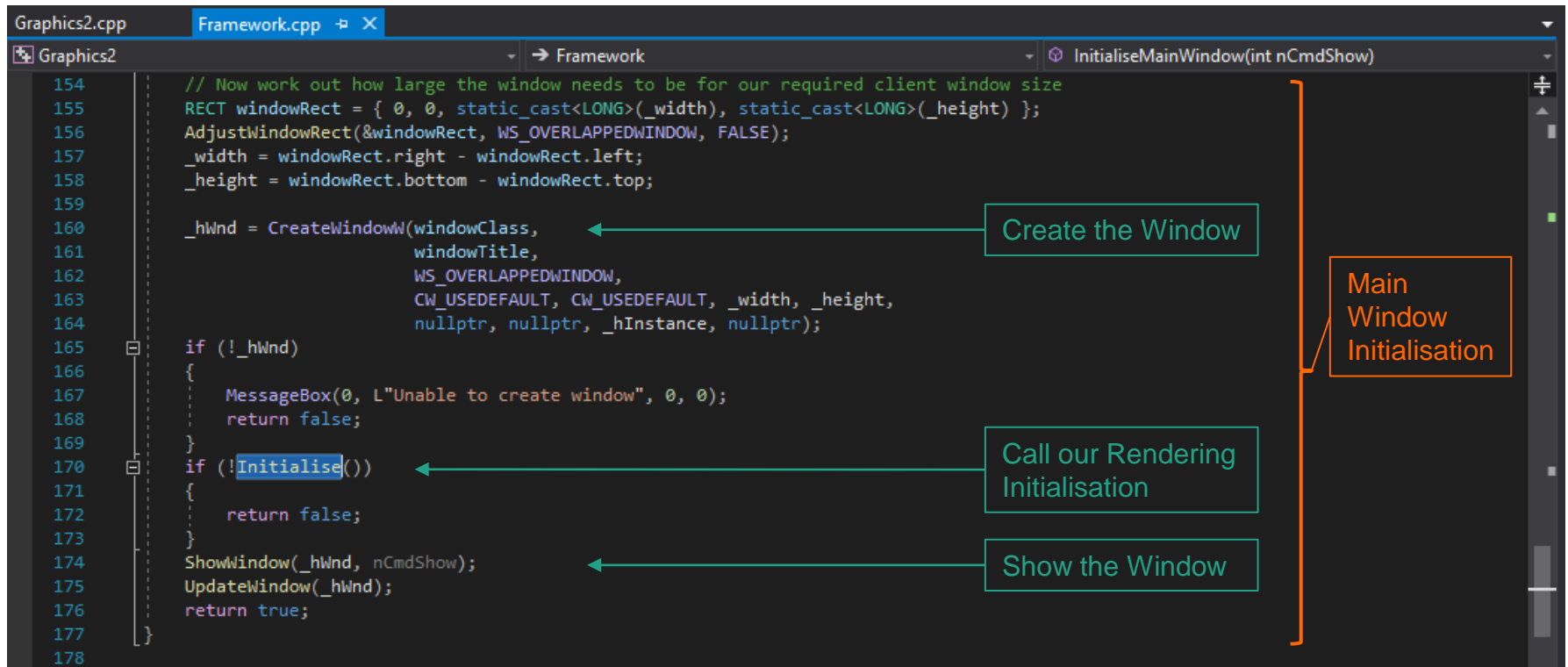


```
Graphics2.cpp Framework.cpp
Graphics2 Framework InitialiseMainWindow(int nCmdShow)
122 // Register the window class, create the window and
123 // create the bitmap that we will use for rendering
124
125 bool Framework::InitialiseMainWindow(int nCmdShow)
126 {
127     #define MAX_LOADSTRING 100
128
129     WCHAR windowTitle[MAX_LOADSTRING];
130     WCHAR windowClass[MAX_LOADSTRING];
131
132     LoadStringW(_hInstance, IDS_APP_TITLE, windowTitle, MAX_LOADSTRING);
133     LoadStringW(_hInstance, IDC_GRAPHICS2, windowClass, MAX_LOADSTRING);
134
135     WNDCLASSEXW wcex;
136     wcex.cbSize = sizeof(WNDCLASSEX);
137     wcex.style = CS_HREDRAW | CS_VREDRAW;
138     wcex.lpfnWndProc = WndProc;
139     wcex.cbClsExtra = 0;
140     wcex.cbWndExtra = 0;
141     wcex.hInstance = _hInstance;
142     wcex.hIcon = LoadIcon(_hInstance, MAKEINTRESOURCE(IDI_GRAPHICS2));
143     wcex.hCursor = LoadCursor(nullptr, IDC_ARROW);
144     wcex.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
145     wcex.lpszMenuName = nullptr;
146     wcex.lpszClassName = windowClass;
147     wcex.hIconSm = LoadIcon(wcex.hInstance, MAKEINTRESOURCE(IDI_SMALL));
148     if (!RegisterClassExW(&wcex))
149     {
150         MessageBox(0, L"Unable to register window class", 0, 0);
151         return false;
152     }
153 }
```

Window Class Properties

Main Window Initialisation

Initialising the Application's Main Window

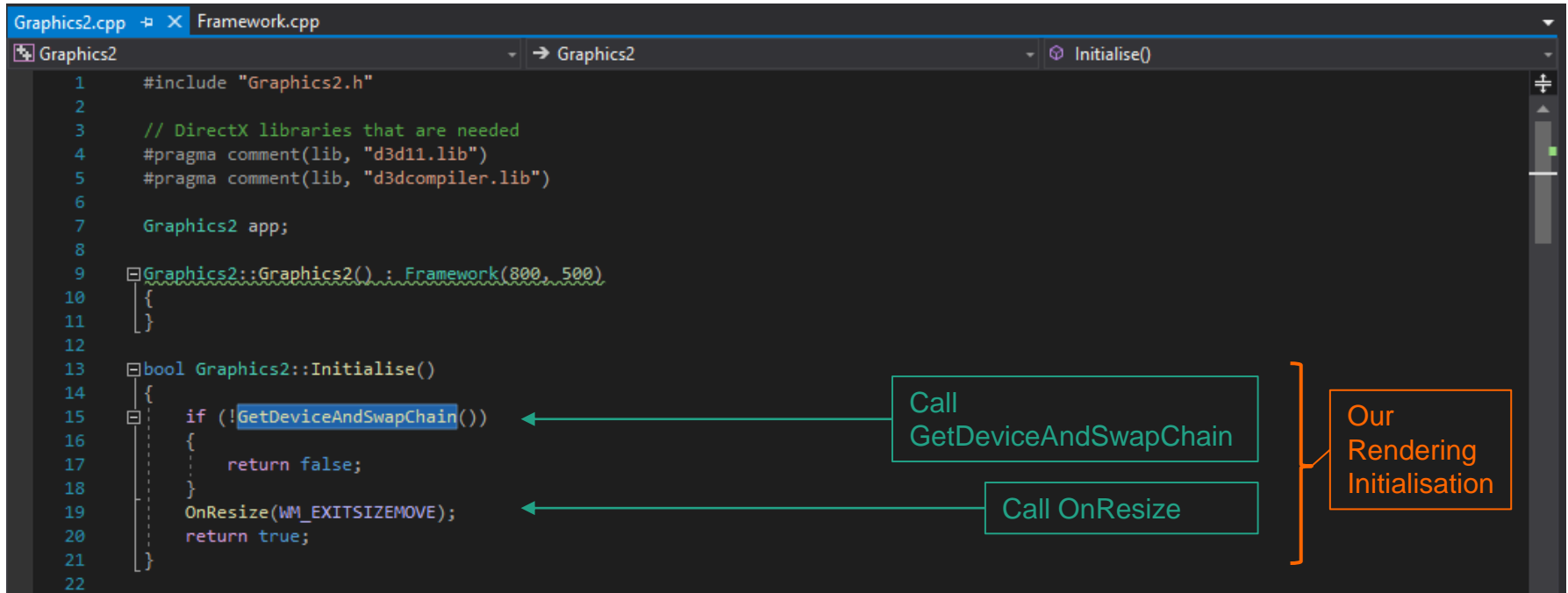


The screenshot shows a C++ IDE with the 'InitialiseMainWindow' function in 'Framework.cpp'. The function is annotated with three callouts: 'Create the Window' pointing to the 'CreateWindowW' call, 'Call our Rendering Initialisation' pointing to the 'Initialise' call, and 'Show the Window' pointing to the 'ShowWindow' and 'UpdateWindow' calls. A large orange bracket on the right groups these three steps under the label 'Main Window Initialisation'.

```
154 // Now work out how large the window needs to be for our required client window size
155 RECT windowRect = { 0, 0, static_cast<LONG>(_width), static_cast<LONG>(_height) };
156 AdjustWindowRect(&windowRect, WS_OVERLAPPEDWINDOW, FALSE);
157 _width = windowRect.right - windowRect.left;
158 _height = windowRect.bottom - windowRect.top;
159
160 _hWnd = CreateWindowW(windowClass,           ← Create the Window
161                      windowTitle,
162                      WS_OVERLAPPEDWINDOW,
163                      CW_USEDEFAULT, CW_USEDEFAULT, _width, _height,
164                      nullptr, nullptr, _hInstance, nullptr);
165
166 if (!_hWnd)
167 {
168     MessageBox(0, L"Unable to create window", 0, 0);
169     return false;
170 }
171 if (!Initialise()) ← Call our Rendering Initialisation
172 {
173     return false;
174 }
175 ShowWindow(_hWnd, nCmdShow); ← Show the Window
176 UpdateWindow(_hWnd);
177 return true;
178 }
```

Main Window Initialisation

Initialising the Rendering Application



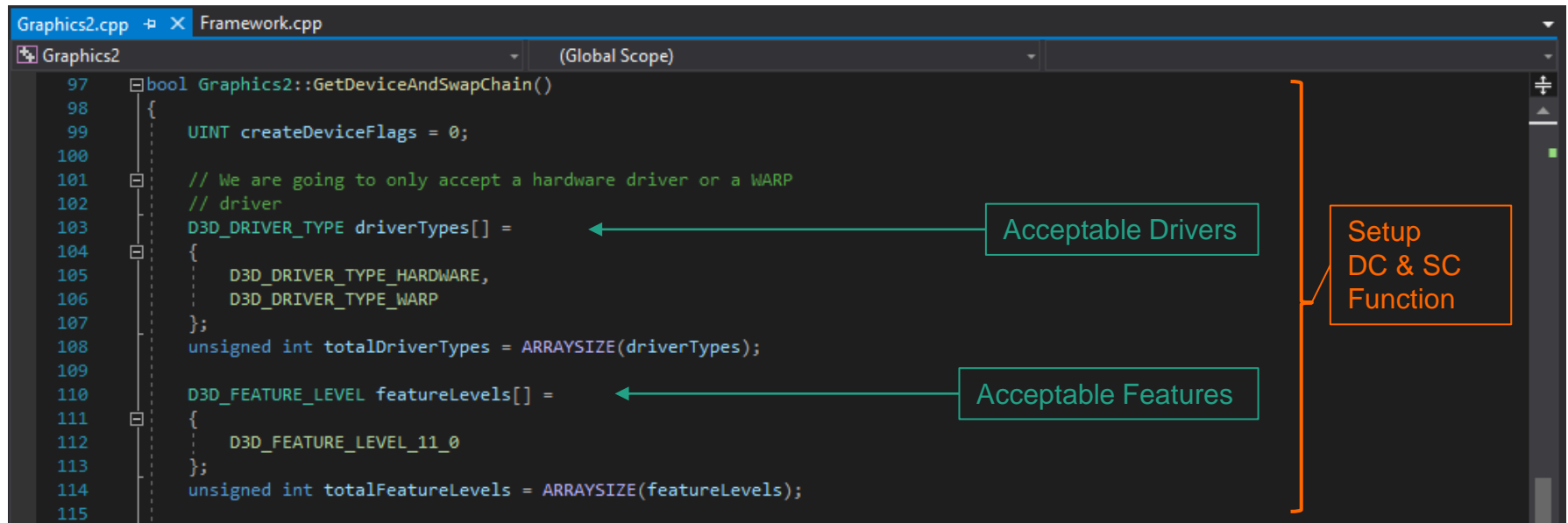
```
Graphics2.cpp  Framework.cpp
Graphics2  → Graphics2  Initialise()
1  #include "Graphics2.h"
2
3  // DirectX libraries that are needed
4  #pragma comment(lib, "d3d11.lib")
5  #pragma comment(lib, "d3dcompiler.lib")
6
7  Graphics2 app;
8
9  Graphics2::Graphics2() : Framework(800, 500)
10 {
11 }
12
13 bool Graphics2::Initialise()
14 {
15     if (!GetDeviceAndSwapChain())
16     {
17         return false;
18     }
19     OnResize(WM_EXITSIZEMOVE);
20     return true;
21 }
22
```

Call
GetDeviceAndSwapChain

Call OnResize

Our
Rendering
Initialisation

Setting up the Device-Context and Swap-Chain



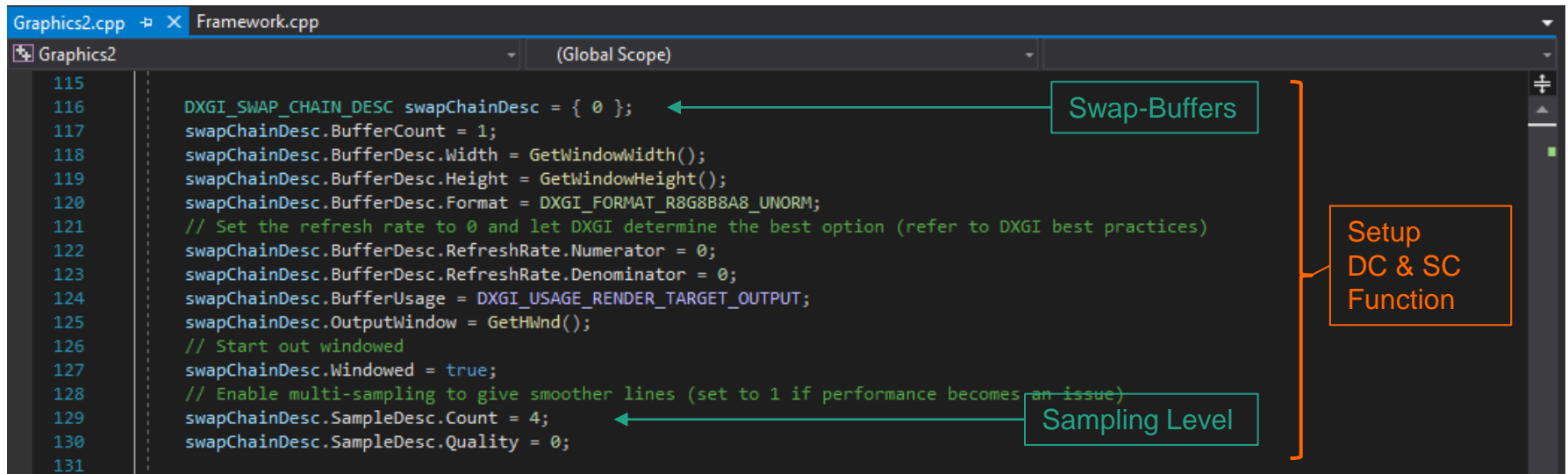
```
97 bool Graphics2::GetDeviceAndSwapChain()
98 {
99     UINT createDeviceFlags = 0;
100
101     // We are going to only accept a hardware driver or a WARP
102     // driver
103     D3D_DRIVER_TYPE driverTypes[] =
104     {
105         D3D_DRIVER_TYPE_HARDWARE,
106         D3D_DRIVER_TYPE_WARP
107     };
108     unsigned int totalDriverTypes = ARRAYSIZE(driverTypes);
109
110     D3D_FEATURE_LEVEL featureLevels[] =
111     {
112         D3D_FEATURE_LEVEL_11_0
113     };
114     unsigned int totalFeatureLevels = ARRAYSIZE(featureLevels);
115 }
```

Acceptable Drivers

Acceptable Features

Setup DC & SC Function

Setting up the Swap-Buffer and Multi-Sampling level

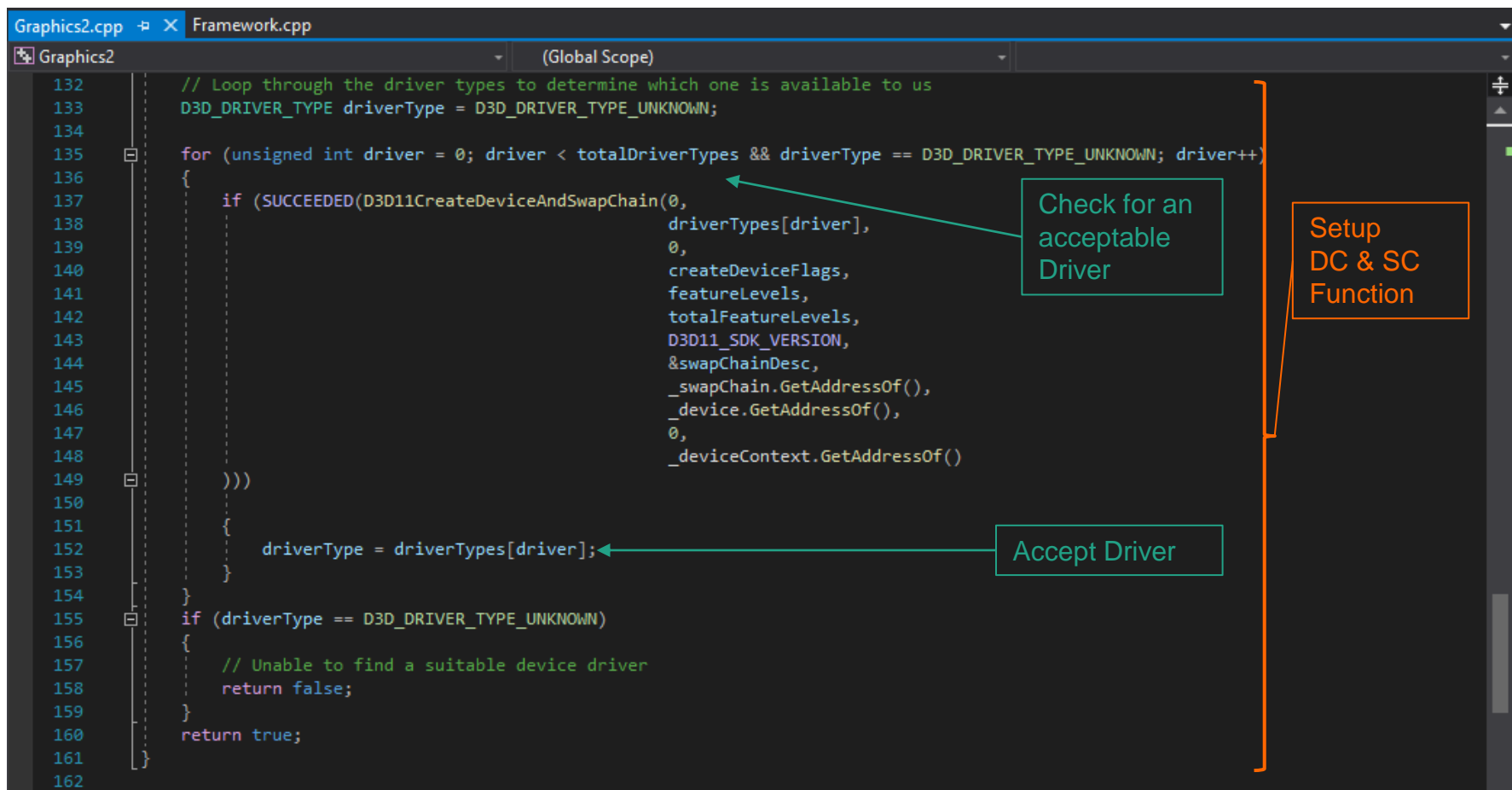


```
115
116     DXGI_SWAP_CHAIN_DESC swapChainDesc = { 0 };
117     swapChainDesc.BufferCount = 1;
118     swapChainDesc.BufferDesc.Width = GetWindowWidth();
119     swapChainDesc.BufferDesc.Height = GetWindowHeight();
120     swapChainDesc.BufferDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
121     // Set the refresh rate to 0 and let DXGI determine the best option (refer to DXGI best practices)
122     swapChainDesc.BufferDesc.RefreshRate.Numerator = 0;
123     swapChainDesc.BufferDesc.RefreshRate.Denominator = 0;
124     swapChainDesc.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;
125     swapChainDesc.OutputWindow = GetHwnd();
126     // Start out windowed
127     swapChainDesc.Windowed = true;
128     // Enable multi-sampling to give smoother lines (set to 1 if performance becomes an issue)
129     swapChainDesc.SampleDesc.Count = 4;
130     swapChainDesc.SampleDesc.Quality = 0;
131
```

Annotations:

- Swap-Buffers (points to line 116)
- Sampling Level (points to line 129)
- Setup DC & SC Function (bracketed around lines 116-130)

Finding an acceptable Driver



```
132 // Loop through the driver types to determine which one is available to us
133 D3D_DRIVER_TYPE driverType = D3D_DRIVER_TYPE_UNKNOWN;
134
135 for (unsigned int driver = 0; driver < totalDriverTypes && driverType == D3D_DRIVER_TYPE_UNKNOWN; driver++)
136 {
137     if (SUCCEEDED(D3D11CreateDeviceAndSwapChain(0,
138         driverTypes[driver],
139         0,
140         createDeviceFlags,
141         featureLevels,
142         totalFeatureLevels,
143         D3D11_SDK_VERSION,
144         &swapChainDesc,
145         _swapChain.GetAddressOf(),
146         _device.GetAddressOf(),
147         0,
148         _deviceContext.GetAddressOf()
149     )))
150     {
151         driverType = driverTypes[driver];
152     }
153 }
154 if (driverType == D3D_DRIVER_TYPE_UNKNOWN)
155 {
156     // Unable to find a suitable device driver
157     return false;
158 }
159 return true;
160
161
162
```

Check for an acceptable Driver

Accept Driver

Setup DC & SC Function