Contents

Create your first Windows app using DirectX

12/13/2021 • 2 minutes to read • Edit Online

If you know DirectX, you can develop a DirectX app using native C++ and HLSL to take full advantage of graphics hardware.

Use this basic tutorial to get started with DirectX app development, then use the roadmap to continue exploring DirectX.

Windows desktop app with C++ and DirectX

A Windows desktop app with DirectX is an app developed using native C++ and DirectX APIs. This model is more complex than an app written in a managed framework, but it provides greater flexibility and greater access to system resources especially graphics devices. So, it is a good model for the experienced developer.

Why develop a Windows app with DirectX?

The answer is simple: you want to make a game that is graphics- or multimedia-intensive, and can use the features that many graphics devices support. This won't be easy if you are new to game development or to Windows development and C/C++, but there's some good news: DirectX 11 is the simplest and most cohesive version of Microsoft DirectX yet. It's also the most powerful and feature-rich. If your goal is to master game development and learn the most advanced rendering techniques, then DirectX can provide the opportunity for you to do that.

That said, planning your game (or interactive, real-time app) is essential. If you are new to game development, and your game doesn't have demanding graphics requirements, consider developing it with the .NET framework instead. Also, many "middleware" graphics and game development packages are available for Windows platforms, and some do not require significant programming skills.

If you are confident, or simply have a dream of making a game with high-fidelity graphics (or an app with complex graphics content), then read on!

In this section

TOPIC	DESCRIPTION
Prerequisites for developing with DirectX	When you start to develop a Windows app using DirectX, keep the prerequisites on this page in mind. This includes the technologies you need to know before you dive in.
Get started with DirectX for Windows	Creating a DirectX game for Windows is a challenge for a new developer. Here we quickly review the concepts involved and the steps you must take to begin developing a game using DirectX and C++.
Roadmap for Desktop DirectX apps	Here are key resources to help you get started with using DirectX and C++ to develop graphics-intensive Desktop apps, like games.

Prerequisites for developing with DirectX

12/13/2021 • 2 minutes to read • Edit Online

When you start to develop a Windows app using DirectX, keep the prerequisites on this page in mind. This includes the technologies you need to know before you dive in.

What should I know to develop a Windows game using DirectX?

Before you start to develop a Windows Store app using DirectX, you need to know how to program in Windows with C++. Windows apps using DirectX are developed at a low level of programming, which means that you will be exposed to many features of the operating system. These include memory and resource management, and the interface for the graphics device itself. If you are new to game or graphics app development, you may find this challenging. But you will also find it rewarding, because learning game development at this level creates far, far greater possibilities for game and graphics app design and development.

You'll also need to understand the basics of 2D and 3D graphics programming and mathematics, because many of the APIs you'll use were developed with these principles in mind. It'll be easier for you to understand their parameters and results if you are familiar with the operations behind them.

At a minimum, you should have a grasp of the following:

- Windows C/C++ programming. This means that you understand pointers and references, events and callbacks, and perhaps a few of the common libraries like ATL.
- Win32 programming. You understand how windows are created, and how user interface events are processed. You also understand a little bit about COM and essential Win32 APIs.
- Linear algebra and trigonometry. While not essential, you'll have an easier time if you are familiar with concepts from these two math disciplines, because they are the foundation of much of 3D graphics programming.
- Basic graphics terminology and concepts, such as bitmaps, textures, vertices, meshes, and viewports.

What does DirectX provide me?

DirectX is the primary set of graphics APIs you'll use to develop Windows games. Here are the categories of features that you must become familiar with when you decide how to develop your game.

LIBRARY	DESCRIPTION
Direct3D	A powerful, performance-oriented, hardware-accelerated set of libraries for rendering 3D graphics.
Direct2D	A set of 2D graphics libraries for hardware-accelerated bitmap and vector 2D drawing.
DirectXMath	A library of common, optimized math operations used in 2D and 3D graphics, such as vector and matrix operations.
DirectWrite	A library of 2D text rendering and layout APIs. It supports both hardware acceleration and software rasterization.

LIBRARY	DESCRIPTION
XAudio2	A low-level, cross-platform audio API for Microsoft Windows that provides a signal processing and audio mixing foundation for game development.
XInput	A library that supports various traditional gaming controls, with an emphasis on the Xbox 360 controller model.

What tools do I need to develop a Windows game with DirectX?

To get started with this tutorial, you need:

- Windows 8.1 or greater
- Microsoft Visual Studio 2013 or greater

Get started with DirectX for Windows

12/13/2021 • 2 minutes to read • Edit Online

Creating a Microsoft DirectX game for Windows is a challenge for a new developer. Here we quickly review the concepts involved and the steps you must take to begin developing a game using DirectX and C++.

Let's get started.

What skills do you need?

To develop a game in DirectX for Windows, you must have a few basic skills. Specifically, you must be able to:

- Read and write modern C++ code (C++11 helps the most), and be familiar with basic C++ design principles and patterns like templates and the factory model. You must also be familiar with common C++ libraries like the Standard Template Library, and specifically with the casting operators, pointer types, and the standard template library data structures (such as std::vector).
- Understand basic geometry, trigonometry, and linear algebra. Much of the code you will find in the examples assumes you understand these forms of mathematics and their common rules.
- Have familiarity with COM—especially `Microsoft::WRL::ComPtr` (smart pointer).
- Understand the foundations of graphics and graphics technology, particularly 3D graphics. While DirectX itself has its own terminology, it still builds upon a well-established understanding of general 3D graphics principles.
- Understand the concept of a message loop, because you'll be implementing a loop that listens to the Windows operating system.

And we're off!

Ready to start? Let's review before we head on. You have:

- An updated and working installation of Windows 8.1.
- An installation of Microsoft Visual Studio.
- An intrepid spirit and a desire to learn more about DirectX game development!

Next steps

TOPIC	DESCRIPTION
Work with DirectX device resources	Learn how to use DXGI to create a virtualized graphics device, and create and configure a swap chain.
Understand the Direct3D 11 rendering pipeline	Learn how to hook into the DirectX device resources class, and draw using the Direct3D graphics pipeline.
Work with shaders and shader resources	Learn how to write HLSL shader programs for Direct3D graphics pipeline stages.

Work with DirectX device resources

12/13/2021 • 12 minutes to read • Edit Online

Understand the role of the Microsoft DirectX Graphics Infrastructure (DXGI) in your Windows Store DirectX game. DXGI is a set of APIs used to configure and manage low-level graphics and graphics adapter resources. Without it, you'd have no way to draw your game's graphics to a window.

Think of DXGI this way: to directly access the GPU and manage its resources, you must have a way to describe it to your app. The most important piece of info you need about the GPU is the place to draw pixels so it can send those pixels to the screen. Typically this is called the "back buffer"—a location in GPU memory where you can draw the pixels and then have it "flipped" or "swapped" and sent to the screen on a refresh signal. DXGI lets you acquire that location and the means to use that buffer (called a *swap chain* because it is a chain of swappable buffers, allowing for multiple buffering strategies).

To do this, you need access to write to the swap chain, and a handle to the window that will display the current back buffer for the swap chain. You also need to connect the two to ensure that the operating system will refresh the window with the contents of the back buffer when you request it to do so.

The overall process for drawing to the screen is as follows:

- Get a **CoreWindow** for your app.
- Get an interface for the Direct3D device and context.
- Create the swap chain to display your rendered image in the **CoreWindow**.
- Create a render target for drawing and populate it with pixels.
- Present the swap chain!

Create a window for your app

The first thing we need to do is create a window. First, create a window class by populating an instance of **WNDCLASS**, then register it using **RegisterClass**. The window class contains essential properties of the window, including the icon it uses, the static message processing function (more on this later), and a unique name for the window class.

```
if(m_hInstance == NULL)
    m_hInstance = (HINSTANCE)GetModuleHandle(NULL);

HICON hIcon = NULL;
WCHAR szExePath[MAX_PATH];
    GetModuleFileName(NULL, szExePath, MAX_PATH);

// If the icon is NULL, then use the first one found in the exe
if(hIcon == NULL)
    hIcon = ExtractIcon(m_hInstance, szExePath, 0);

// Register the windows class
WNDCLASS wndClass;
wndClass.style = CS_DBLCLKS;
wndClass.lpfnWndProc = MainClass::StaticWindowProc;
wndClass.cbClsExtra = 0;
wndClass.cbWndExtra = 0;
wndClass.hInstance = m_hInstance;
wndClass.hIcon = hIcon;
wndClass.hCursor = LoadCursor(NULL, IDC_ARROW);
wndClass.hbrBackground = (HBRUSH)GetStockObject(BLACK_BRUSH);
wndClass.lpszMenuName = NULL;
wndClass.lpszClassName = m_windowClassName.c_str();

if(!RegisterClass(&wndClass))
{
    DWORD dwError = GetLastError();
    if(dwError != ERROR_CLASS_ALREADY_EXISTS)
        return HRESULT_FROM_WIN32(dwError);
}
```

Next, you create the window. We also need to provide size information for the window and the name of the window class we just created. When you call **CreateWindow**, you get back an opaque pointer to the window called an HWND; you'll need to keep the HWND pointer and use it any time you need to reference the window, including destroying or recreating it, and (especially important) when creating the DXGI swap chain you use to draw in the window.

```
m_rc;
int x = CW_USEDEFAULT;
int y = CW_USEDEFAULT;

// No menu in this example.
m_hMenu = NULL;

// This example uses a non-resizable 640 by 480 viewport for simplicity.
int nDefaultWidth = 640;
int nDefaultHeight = 480;
SetRect(&m_rc, 0, 0, nDefaultWidth, nDefaultHeight);
AdjustWindowRect(
    &m_rc,
    WS_OVERLAPPEDWINDOW,
    (m_hMenu != NULL) ? true : false
    );

// Create the window for our viewport.
m_hWnd = CreateWindow(
    m_windowClassName.c_str(),
    L"Cube11",
    WS_OVERLAPPEDWINDOW,
    x, y,
    (m_rc.right-m_rc.left), (m_rc.bottom-m_rc.top),
    0,
    m_hMenu,
    m_hInstance,
    0
    );

if(m_hWnd == NULL)
{
    DWORD dwError = GetLastError();
    return HRESULT_FROM_WIN32(dwError);
}
```

The Windows desktop app model includes a hook into the Windows message loop. You'll need to base your main program loop off of this hook by writing a "StaticWindowProc" function to process windowing events. This must be a static function because Windows will call it outside of the context of any class instance. Here's a very simple example of a static message processing function.

```
LRESULT CALLBACK MainClass::StaticWindowProc(
    HWND hWnd,
    UINT uMsg,
    WPARAM wParam,
    LPARAM lParam
    )
{
    switch(uMsg)
    {
        case WM_CLOSE:
        {
            HMENU hMenu;
            hMenu = GetMenu(hWnd);
            if (hMenu != NULL)
            {
                DestroyMenu(hMenu);
            }
            DestroyWindow(hWnd);
            UnregisterClass(
                m_windowClassName.c_str(),
                m_hInstance
                );
            return 0;
        }

        case WM_DESTROY:
            PostQuitMessage(0);
            break;
    }

    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
```

This simple example only checks for program exit conditions: **WM_CLOSE**, sent when the window is requested to be closed, and **WM_DESTROY**, which is sent after the window is actually removed from the screen. A full, production app needs to handle other windowing events too—for the complete list of windowing events, see Window Notifications.

The main program loop itself needs to acknowledge Windows messages by allowing Windows the opportunity to run the static message proc. Help the program run efficiently by forking the behavior: each iteration should choose to process new Windows messages if they are available, and if no messages are in the queue it should render a new frame. Here's a very simple example:

```
bool bGotMsg;
MSG  msg;
msg.message = WM_NULL;
PeekMessage(&msg, NULL, 0U, 0U, PM_NOREMOVE);

while (WM_QUIT != msg.message)
{
    // Process window events.
    // Use PeekMessage() so we can use idle time to render the scene.
    bGotMsg = (PeekMessage(&msg, NULL, 0U, 0U, PM_REMOVE) != 0);

    if (bGotMsg)
    {
        // Translate and dispatch the message
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    else
    {
        // Update the scene.
        renderer->Update();

        // Render frames during idle time (when no messages are waiting).
        renderer->Render();

        // Present the frame to the screen.
        deviceResources->Present();
    }
}
```

Get an interface for the Direct3D device and context

The first step to using Direct3D is to acquire an interface for the Direct3D hardware (the GPU), represented as instances of **ID3D11Device** and **ID3D11DeviceContext**. The former is a virtual representation of the GPU resources, and the latter is a device-agnostic abstraction of the rendering pipeline and process. Here's an easy way to think of it: **ID3D11Device** contains the graphics methods you call infrequently, usually before any rendering occurs, to acquire and configure the set of resources you need to start drawing pixels. **ID3D11DeviceContext**, on the other hand, contains the methods you call every frame: loading in buffers and views and other resources, changing output-merger and rasterizer state, managing shaders, and drawing the results of passing those resources through the states and shaders.

There's one very important part of this process: setting the feature level. The feature level tells DirectX the minimum level of hardware your app supports, with D3D_FEATURE_LEVEL_9_1 as the lowest feature set and D3D_FEATURE_LEVEL_11_1 as the current highest. You should support 9_1 as the minimum if you want to reach the widest possible audience. Take some time to read up on Direct3D feature levels and assess for yourself the minimum and maximum feature levels you want your game to support and to understand the implications of your choice.

Obtain references (pointers) to both the Direct3D device and device context and store them as class-level variables on the **DeviceResources** instance (as **ComPtr** smart pointers). Use these references whenever you need to access the Direct3D device or device context.

```
D3D_FEATURE_LEVEL levels[] = {
    D3D_FEATURE_LEVEL_9_1,
    D3D_FEATURE_LEVEL_9_2,
    D3D_FEATURE_LEVEL_9_3,
    D3D_FEATURE_LEVEL_10_0,
    D3D_FEATURE_LEVEL_10_1,
    D3D_FEATURE_LEVEL_11_0,
    D3D_FEATURE_LEVEL_11_1
};

// This flag adds support for surfaces with a color-channel ordering different
// from the API default. It is required for compatibility with Direct2D.
UINT deviceFlags = D3D11_CREATE_DEVICE_BGRA_SUPPORT;

#if defined(DEBUG) || defined(_DEBUG)
deviceFlags |= D3D11_CREATE_DEVICE_DEBUG;
#endif

// Create the Direct3D 11 API device object and a corresponding context.
Microsoft::WRL::ComPtr<ID3D11Device>        device;
Microsoft::WRL::ComPtr<ID3D11DeviceContext> context;

hr = D3D11CreateDevice(
    nullptr,                    // Specify nullptr to use the default adapter.
    D3D_DRIVER_TYPE_HARDWARE,   // Create a device using the hardware graphics driver.
    0,                          // Should be 0 unless the driver is D3D_DRIVER_TYPE_SOFTWARE.
    deviceFlags,                // Set debug and Direct2D compatibility flags.
    levels,                     // List of feature levels this app can support.
    ARRAYSIZE(levels),          // Size of the list above.
    D3D11_SDK_VERSION,          // Always set this to D3D11_SDK_VERSION for Windows Store apps.
    &device,                    // Returns the Direct3D device created.
    &m_featureLevel,            // Returns feature level of device created.
    &context                    // Returns the device immediate context.
    );

if (FAILED(hr))
{
    // Handle device interface creation failure if it occurs.
    // For example, reduce the feature level requirement, or fail over
    // to WARP rendering.
}

// Store pointers to the Direct3D 11.1 API device and immediate context.
device.As(&m_pd3dDevice);
context.As(&m_pd3dDeviceContext);
```

Create the swap chain

Okay: You have a window to draw in, and you have an interface to send data and give commands to the GPU. Now let's see how to bring them together.

First, you tell DXGI what values to use for the properties of the swap chain. Do this using a DXGI_SWAP_CHAIN_DESC structure. Six fields are particularly important for desktop apps:

- **Windowed**: Indicates whether the swap chain is full-screen or clipped to the window. Set this to TRUE to put the swap chain in the window you created earlier.
- **BufferUsage**: Set this to DXGI_USAGE_RENDER_TARGET_OUTPUT. This indicates that the swap chain will be a drawing surface, allowing you to use it as a Direct3D render-target.
- **SwapEffect**: Set this to DXGI_SWAP_EFFECT_FLIP_SEQUENTIAL.
- **Format**: The DXGI_FORMAT_B8G8R8A8_UNORM format specifies 32-bit color: 8 bits for each of the three RGB color channels, and 8 bits for the alpha channel.
- **BufferCount**: Set this to 2 for a traditional double-buffered behavior to avoid tearing. Set the buffer count to

3 if your graphics content takes more than one monitor refresh cycle to render a single frame (at 60 Hz for example, the threshold is more than 16 ms).

- **SampleDesc**: This field controls multisampling. Set **Count** to 1 and **Quality** to 0 for flip-model swap chains. (To use multisampling with flip-model swap chains, draw on a separate multisampled render target and then resolve that target to the swap chain just before presenting it. Example code is provided in Multisampling in Windows Store apps.)

After you have specified a configuration for the swap chain, you must use the same DXGI factory that created the Direct3D device (and device context) in order to create the swap chain.

**Short form: **

Get the **ID3D11Device** reference you created previously. Upcast it to **IDXGIDevice3** (if you haven't already) and then call **IDXGIDevice::GetAdapter** to acquire the DXGI adapter. Get the parent factory for that adapter by calling **IDXGIFactory2::GetParent** (**IDXGIFactory2** inherits from **IDXGIObject**)—now you can use that factory to create the swap chain by calling **CreateSwapChainForHwnd**, as seen in the following code sample.

```
DXGI_SWAP_CHAIN_DESC desc;
ZeroMemory(&desc, sizeof(DXGI_SWAP_CHAIN_DESC));
desc.Windowed = TRUE; // Sets the initial state of full-screen mode.
desc.BufferCount = 2;
desc.BufferDesc.Format = DXGI_FORMAT_B8G8R8A8_UNORM;
desc.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;
desc.SampleDesc.Count = 1;      //multisampling setting
desc.SampleDesc.Quality = 0;    //vendor-specific flag
desc.SwapEffect = DXGI_SWAP_EFFECT_FLIP_SEQUENTIAL;
desc.OutputWindow = hWnd;

// Create the DXGI device object to use in other factories, such as Direct2D.
Microsoft::WRL::ComPtr<IDXGIDevice3> dxgiDevice;
m_pd3dDevice.As(&dxgiDevice);

// Create swap chain.
Microsoft::WRL::ComPtr<IDXGIAdapter> adapter;
Microsoft::WRL::ComPtr<IDXGIFactory> factory;

hr = dxgiDevice->GetAdapter(&adapter);

if (SUCCEEDED(hr))
{
    adapter->GetParent(IID_PPV_ARGS(&factory));

    hr = factory->CreateSwapChain(
        m_pd3dDevice.Get(),
        &desc,
        &m_pDXGISwapChain
        );
}
```

If you're just starting out, it's probably best to use the configuration shown here. Now at this point, if you are already familiar with previous versions of DirectX you might be asking: "Why didn't we create the device and swap chain at the same time, instead of walking back through all of those classes?" The answer is efficiency: swap chains are Direct3D device resources, and device resources are tied to the particular Direct3D device that created them. If you create a new device with a new swap chain, you have to recreate all your device resources using the new Direct3D device. So by creating the swap chain with the same factory (as shown above), you are able to recreate the swap chain and continue using the Direct3D device resources you already have loaded!

Now you've got a window from the operating system, a way to access the GPU and its resources, and a swap chain to display the rendering results. All that's left is to wire the whole thing together!

Create a render target for drawing

The shader pipeline needs a resource to draw pixels into. The simplest way to create this resource is to define a ID3D11Texture2D resource as a back buffer for the pixel shader to draw into, and then read that texture into the swap chain.

To do this, you create a render-target *view*. In Direct3D, a view is a way to access a specific resource. In this case, the view enables the pixel shader to write into the texture as it completes its per-pixel operations.

Let's look at the code for this. When you set DXGI_USAGE_RENDER_TARGET_OUTPUT on the swap chain, that enabled the underlying Direct3D resource to be used as a drawing surface. So to get our render target view, we just need to get the back buffer from the swap chain and create a render target view bound to the back buffer resource.

```
hr = m_pDXGISwapChain->GetBuffer(
    0,
    __uuidof(ID3D11Texture2D),
    (void**) &m_pBackBuffer);

hr = m_pd3dDevice->CreateRenderTargetView(
    m_pBackBuffer.Get(),
    nullptr,
    m_pRenderTarget.GetAddressOf()
    );

m_pBackBuffer->GetDesc(&m_bbDesc);
```

Also create a *depth-stencil buffer*. A depth-stencil buffer is just a particular form of ID3D11Texture2D resource, which is typically used to determine which pixels have draw priority during rasterization based on the distance of the objects in the scene from the camera. A depth-stencil buffer can also be used for stencil effects, where specific pixels are discarded or ignored during rasterization. This buffer must be the same size as the render target. Note that you cannot read from or render to the frame buffer depth-stencil texture because it is used exclusively by the shader pipeline before and during final rasterization.

Also create a view for the depth-stencil buffer as an ID3D11DepthStencilView. The view tells the shader pipeline how to interpret the underlying ID3D11Texture2D resource - so if you don't supply this view no per-pixel depth testing is performed, and the objects in your scene may seem a bit inside-out at the very least!

```
CD3D11_TEXTURE2D_DESC depthStencilDesc(
    DXGI_FORMAT_D24_UNORM_S8_UINT,
    static_cast<UINT> (m_bbDesc.Width),
    static_cast<UINT> (m_bbDesc.Height),
    1, // This depth stencil view has only one texture.
    1, // Use a single mipmap level.
    D3D11_BIND_DEPTH_STENCIL
    );

m_pd3dDevice->CreateTexture2D(
    &depthStencilDesc,
    nullptr,
    &m_pDepthStencil
    );

CD3D11_DEPTH_STENCIL_VIEW_DESC depthStencilViewDesc(D3D11_DSV_DIMENSION_TEXTURE2D);

m_pd3dDevice->CreateDepthStencilView(
    m_pDepthStencil.Get(),
    &depthStencilViewDesc,
    &m_pDepthStencilView
    );
```

The last step is to create a viewport. This defines the visible rectangle of the back buffer displayed on the screen; you can change the part of the buffer that is displayed on the screen by changing the parameters of the viewport. This code targets the whole window size—or the screen resolution, in the case of full-screen swap chains. For fun, change the supplied coordinate values and observe the results.

```
ZeroMemory(&m_viewport, sizeof(D3D11_VIEWPORT));
m_viewport.Height = (float) m_bbDesc.Height;
m_viewport.Width = (float) m_bbDesc.Width;
m_viewport.MinDepth = 0;
m_viewport.MaxDepth = 1;

m_pd3dDeviceContext->RSSetViewports(
    1,
    &m_viewport
    );
```

And that's how you go from nothing to drawing pixels in a window! As you start out, it's a good idea to become familiar with how DirectX, through DXGI, manages the core resources you need to start drawing pixels.

Next you'll look at the structure of the graphics pipeline; see Understand the DirectX app template's rendering pipeline.

Related topics

Up next

Work with shaders and shader resources

Understand the Direct3D 11 rendering pipeline

12/13/2021 • 11 minutes to read • Edit Online

Previously, you looked at how to create a window you can use for drawing in Work with DirectX device resources. Now, you learn how to build the graphics pipeline, and where you can hook into it.

You'll recall that there are two Direct3D interfaces that define the graphics pipeline: **ID3D11Device**, which provides a virtual representation of the GPU and its resources; and **ID3D11DeviceContext**, which represents the graphics processing for the pipeline. Typically, you use an instance of **ID3D11Device** to configure and obtain the GPU resources you need to start processing the graphics in a scene, and you use **ID3D11DeviceContext** to process those resources at each appropriate shader stage in the graphics pipeline. You usually call **ID3D11Device** methods infrequently—that is, only when you set up a scene or when the device changes. On the other hand, you'll call **ID3D11DeviceContext** every time you process a frame for display.

This example creates and configures a minimal graphics pipeline suitable for displaying a simple spinning, vertex-shaded cube. It demonstrates approximately the smallest set of resources necessary for display. As you read the info here, note the limitations of the given example where you may have to extend it to support the scene you want to render.

This example covers two C++ classes for graphics: a device resource manager class, and 3D scene renderer class. This topic focuses specifically on the 3D scene renderer.

What does the cube renderer do?

The graphics pipeline is defined by the 3D scene renderer class. The scene renderer is able to:

- Define constant buffers to store your uniform data.
- Define vertex buffers to hold your object vertex data, and corresponding index buffers to enable the vertex shader to walk the triangles correctly.
- Create texture resources and resource views.
- Load your shader objects.
- Update the graphics data to display each frame.
- Render (draw) the graphics to the swap chain.

The first four processes typically use the **ID3D11Device** interface methods for initializing and managing graphics resources, and the last two use the **ID3D11DeviceContext** interface methods to manage and execute the graphics pipeline.

An instance of the **Renderer** class is created and managed as a member variable on the main project class. The **DeviceResources** instance is managed as a shared pointer across several classes, including the main project class, the **App** view-provider class, and **Renderer**. If you replace **Renderer** with your own class, consider declaring and assigning the **DeviceResources** instance as a shared pointer member as well:

```
std::shared_ptr<DX::DeviceResources> m_deviceResources;
```

Just pass the pointer into the class constructor (or other initialization method) after the **DeviceResources** instance is created in the **Initialize** method of the **App** class. You can also pass a **weak_ptr** reference if, instead, you want your main class to completely own the **DeviceResources** instance.

Create the cube renderer

In this example, we organize the scene renderer class with the following methods:

- **CreateDeviceDependentResources**: Called whenever the scene must be initialized or restarted. This method loads your initial vertex data, textures, shaders, and other resources, and constructs the initial constant and vertex buffers. Typically, most of the work here is done with **ID3D11Device** methods, not **ID3D11DeviceContext** methods.
- **CreateWindowSizeDependentResources**: Called whenever the window state changes, such as when resizing occurs or when orientation changes. This method rebuilds transform matrices, such as those for your camera.
- **Update**: Typically called from the part of the program that manages immediate game state; in this example, we just call it from the **Main** class. Have this method read from any game-state information that affects rendering, such as updates to object position or animation frames, plus any global game data like light levels or changes to game physics. These inputs are used to update the per-frame constant buffers and object data.
- **Render**: Typically called from the part of the program that manages the game loop; in this case, it's called from the **Main** class. This method constructs the graphics pipeline: it binds shaders, binds buffers and resources to shader stages, and invokes drawing for the current frame.

These methods comprise the body of behaviors for rendering a scene with Direct3D using your assets. If you extend this example with a new rendering class, declare it on the main project class. So this:

```
std::unique_ptr<Sample3DSceneRenderer> m_sceneRenderer;
```

becomes this:

```
std::unique_ptr<MyAwesomeNewSceneRenderer> m_sceneRenderer;
```

Again, note that this example assumes that the methods have the same signatures in your implementation. If the signatures have changed, review the **Main** loop and make the changes accordingly.

Let's take a look at scene-rendering methods in more detail.

Create device dependent resources

CreateDeviceDependentResources consolidates all the operations for initializing the scene and its resources using **ID3D11Device** calls. This method assumes that the Direct3D device has just been initialized (or has been recreated) for a scene. It recreates or reloads all scene-specific graphics resources, such as the vertex and pixel shaders, the vertex and index buffers for objects, and any other resources (for example, as textures and their corresponding views).

Here's example code for **CreateDeviceDependentResources**:

```
void Renderer::CreateDeviceDependentResources()
{
    // Compile shaders using the Effects library.
    auto CreateShadersTask = Concurrency::create_task(
            [this]( )
            {
                CreateShaders();
            }
        );

    // Load the geometry for the spinning cube.
    auto CreateCubeTask = CreateShadersTask.then(
            [this]()
            {
                CreateCube();
            }
        );
}

void Renderer::CreateWindowSizeDependentResources()
{
    // Create the view matrix and the perspective matrix.
    CreateViewAndPerspective();
}
```

Any time you load resources from disk—resources like compiled shader object (CSO, or .cso) files or textures—do so asynchronously. This allows you to keep other work going at the same time (like other setup tasks), and because the main loop isn't blocked you can keep displaying something visually interesting to the user (like a loading animation for your game). This example uses the Concurrency::Tasks API that is available starting in Windows 8; note the lambda syntax used to encapsulate asynchronous loading tasks. These lambdas represent the functions called off-thread, so a pointer to the current class object (**this**) is explicitly captured.

Here's an example of how you can load shader bytecode:

```
HRESULT hr = S_OK;

// Use the Direct3D device to load resources into graphics memory.
ID3D11Device* device = m_deviceResources->GetDevice();

// You'll need to use a file loader to load the shader bytecode. In this
// example, we just use the standard library.
FILE* vShader, * pShader;
BYTE* bytes;

size_t destSize = 4096;
size_t bytesRead = 0;
bytes = new BYTE[destSize];

fopen_s(&vShader, "CubeVertexShader.cso", "rb");
bytesRead = fread_s(bytes, destSize, 1, 4096, vShader);
hr = device->CreateVertexShader(
    bytes,
    bytesRead,
    nullptr,
    &m_pVertexShader
    );

D3D11_INPUT_ELEMENT_DESC iaDesc [] =
{
    { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT,
    0, 0, D3D11_INPUT_PER_VERTEX_DATA, 0 },

    { "COLOR", 0, DXGI_FORMAT_R32G32B32_FLOAT,
    0, 12, D3D11_INPUT_PER_VERTEX_DATA, 0 },
```

```
};

hr = device->CreateInputLayout(
    iaDesc,
    ARRAYSIZE(iaDesc),
    bytes,
    bytesRead,
    &m_pInputLayout
    );

delete bytes;


bytes = new BYTE[destSize];
bytesRead = 0;
fopen_s(&pShader, "CubePixelShader.cso", "rb");
bytesRead = fread_s(bytes, destSize, 1, 4096, pShader);
hr = device->CreatePixelShader(
    bytes,
    bytesRead,
    nullptr,
    m_pPixelShader.GetAddressOf()
    );

delete bytes;

CD3D11_BUFFER_DESC cbDesc(
    sizeof(ConstantBufferStruct),
    D3D11_BIND_CONSTANT_BUFFER
    );

hr = device->CreateBuffer(
    &cbDesc,
    nullptr,
    m_pConstantBuffer.GetAddressOf()
    );

fclose(vShader);
fclose(pShader);
```

Here's an example of how to create vertex and index buffers:

```
HRESULT Renderer::CreateCube()
{
    HRESULT hr = S_OK;

    // Use the Direct3D device to load resources into graphics memory.
    ID3D11Device* device = m_deviceResources->GetDevice();

    // Create cube geometry.
    VertexPositionColor CubeVertices[] =
    {
        {DirectX::XMFLOAT3(-0.5f,-0.5f,-0.5f), DirectX::XMFLOAT3(  0,   0,   0),},
        {DirectX::XMFLOAT3(-0.5f,-0.5f, 0.5f), DirectX::XMFLOAT3(  0,   0,   1),},
        {DirectX::XMFLOAT3(-0.5f, 0.5f,-0.5f), DirectX::XMFLOAT3(  0,   1,   0),},
        {DirectX::XMFLOAT3(-0.5f, 0.5f, 0.5f), DirectX::XMFLOAT3(  0,   1,   1),},

        {DirectX::XMFLOAT3( 0.5f,-0.5f,-0.5f), DirectX::XMFLOAT3(  1,   0,   0),},
        {DirectX::XMFLOAT3( 0.5f,-0.5f, 0.5f), DirectX::XMFLOAT3(  1,   0,   1),},
        {DirectX::XMFLOAT3( 0.5f, 0.5f,-0.5f), DirectX::XMFLOAT3(  1,   1,   0),},
        {DirectX::XMFLOAT3( 0.5f, 0.5f, 0.5f), DirectX::XMFLOAT3(  1,   1,   1),},
    };

    // Create vertex buffer:

    CD3D11_BUFFER_DESC vDesc(
        sizeof(CubeVertices),
        D3D11_BIND_VERTEX_BUFFER
```

```
        D3D11_BIND_VERTEX_BUFFER
        );

    D3D11_SUBRESOURCE_DATA vData;
    ZeroMemory(&vData, sizeof(D3D11_SUBRESOURCE_DATA));
    vData.pSysMem = CubeVertices;
    vData.SysMemPitch = 0;
    vData.SysMemSlicePitch = 0;

    hr = device->CreateBuffer(
        &vDesc,
        &vData,
        &m_pVertexBuffer
        );

    // Create index buffer:
    unsigned short CubeIndices [] =
    {
        0,2,1, // -x
        1,2,3,

        4,5,6, // +x
        5,7,6,

        0,1,5, // -y
        0,5,4,

        2,6,7, // +y
        2,7,3,

        0,4,6, // -z
        0,6,2,

        1,3,7, // +z
        1,7,5,
    };

    m_indexCount = ARRAYSIZE(CubeIndices);

    CD3D11_BUFFER_DESC iDesc(
        sizeof(CubeIndices),
        D3D11_BIND_INDEX_BUFFER
        );

    D3D11_SUBRESOURCE_DATA iData;
    ZeroMemory(&iData, sizeof(D3D11_SUBRESOURCE_DATA));
    iData.pSysMem = CubeIndices;
    iData.SysMemPitch = 0;
    iData.SysMemSlicePitch = 0;

    hr = device->CreateBuffer(
        &iDesc,
        &iData,
        &m_pIndexBuffer
        );

    return hr;
}
```

This example does not load any meshes or textures. You must create the methods for loading the mesh and texture types that are specific to your game, and call them asynchronously.

Populate initial values for your per-scene constant buffers here, too. Examples of per-scene constant buffer include fixed lights, or other static scene elements and data.

Implement the CreateWindowSizeDependentResources method

CreateWindowSizeDependentResources methods are called every time the window size, orientation, or resolution changes.

Window size resources are updated like so: The static message proc gets one of several possible events indicating a change in window state. Your main loop is then informed about the event and calls **CreateWindowSizeDependentResources** on the main class instance, which then calls the **CreateWindowSizeDependentResources** implementation on the scene renderer class.

The primary job of this method is to make sure the visuals don't become confused or invalid because of a change in window properties. In this example, we update the project matrices with a new field of view (FOV) for the resized or reoriented window.

We already saw the code for creating window resources in **DeviceResources** - that was the swap chain (with back buffer) and render target view. Here's how the renderer creates aspect ratio-dependent transforms:

```
void Renderer::CreateViewAndPerspective()
{
    // Use DirectXMath to create view and perspective matrices.

    DirectX::XMVECTOR eye = DirectX::XMVectorSet(0.0f, 0.7f, 1.5f, 0.f);
    DirectX::XMVECTOR at  = DirectX::XMVectorSet(0.0f,-0.1f, 0.0f, 0.f);
    DirectX::XMVECTOR up  = DirectX::XMVectorSet(0.0f, 1.0f, 0.0f, 0.f);

    DirectX::XMStoreFloat4x4(
        &m_constantBufferData.view,
        DirectX::XMMatrixTranspose(
            DirectX::XMMatrixLookAtRH(
                eye,
                at,
                up
                )
            )
        );

    float aspectRatioX = m_deviceResources->GetAspectRatio();
    float aspectRatioY = aspectRatioX < (16.0f / 9.0f) ? aspectRatioX / (16.0f / 9.0f) : 1.0f;

    DirectX::XMStoreFloat4x4(
        &m_constantBufferData.projection,
        DirectX::XMMatrixTranspose(
            DirectX::XMMatrixPerspectiveFovRH(
                2.0f * std::atan(std::tan(DirectX::XMConvertToRadians(70) * 0.5f) / aspectRatioY),
                aspectRatioX,
                0.01f,
                100.0f
                )
            )
        );
}
```

If your scene has a specific layout of components that depends on the aspect ratio, this is the place to rearrange them to match that aspect ratio. You may want to change the configuration of post-processing behavior here also.

Implement the Update method

The **Update** method is called once per game loop - in this example, it is called by the main class's method of the same name. It has a simple purpose: update scene geometry and game state based on the amount of elapsed time (or elapsed time steps) since the previous frame. In this example, we simply rotate the cube once per frame. In a real game scene, this method contains a lot more code for checking game state, updating per-frame (or other dynamic) constant buffers, geometry buffers, and other in-memory assets accordingly. Since

communication between the CPU and GPU incurs overhead, make sure you only update buffers that have actually changed since the last frame - your constant buffers can be grouped, or split up, as needed to make this more efficient.

```
void Renderer::Update()
{
    // Rotate the cube 1 degree per frame.
    DirectX::XMStoreFloat4x4(
        &m_constantBufferData.world,
        DirectX::XMMatrixTranspose(
            DirectX::XMMatrixRotationY(
                DirectX::XMConvertToRadians(
                    (float) m_frameCount++
                    )
                )
            )
        );

    if (m_frameCount == MAXUINT)  m_frameCount = 0;
}
```

In this case, **Rotate** updates the constant buffer with a new transformation matrix for the cube. The matrix will be multiplied per-vertex during the vertex shader stage. Since this method is called with every frame, this is a good place to aggregate any methods that update your dynamic constant and vertex buffers, or to perform any other operations that prepare the objects in the scene for transformation by the graphics pipeline.

Implement the Render method

This method is called once per game loop after calling **Update**. Like **Update**, the **Render** method is also called from the main class. This is the method where the graphics pipeline is constructed and processed for the frame using methods on the ID3D11DeviceContext instance. This culminates in a final call to ID3D11DeviceContext::DrawIndexed. It's important to understand that this call (or other similar **Draw*** calls defined on **ID3D11DeviceContext**) actually executes the pipeline. Specifically, this is when Direct3D communicates with the GPU to set drawing state, runs each pipeline stage, and writes the pixel results into the render-target buffer resource for display by the swap chain. Since communication between the CPU and GPU incurs overhead, combine multiple draw calls into a single one if you can, especially if your scene has a lot of rendered objects.

```
void Renderer::Render()
{
    // Use the Direct3D device context to draw.
    ID3D11DeviceContext* context = m_deviceResources->GetDeviceContext();

    ID3D11RenderTargetView* renderTarget = m_deviceResources->GetRenderTarget();
    ID3D11DepthStencilView* depthStencil = m_deviceResources->GetDepthStencil();

    context->UpdateSubresource(
        m_pConstantBuffer.Get(),
        0,
        nullptr,
        &m_constantBufferData,
        0,
        0
        );

    // Clear the render target and the z-buffer.
    const float teal [] = { 0.098f, 0.439f, 0.439f, 1.000f };
    context->ClearRenderTargetView(
        renderTarget,
        teal
        );
```

```
    context->ClearDepthStencilView(
        depthStencil,
        D3D11_CLEAR_DEPTH | D3D11_CLEAR_STENCIL,
        1.0f,
        0);

    // Set the render target.
    context->OMSetRenderTargets(
        1,
        &renderTarget,
        depthStencil
        );

    // Set up the IA stage by setting the input topology and layout.
    UINT stride = sizeof(VertexPositionColor);
    UINT offset = 0;

    context->IASetVertexBuffers(
        0,
        1,
        m_pVertexBuffer.GetAddressOf(),
        &stride,
        &offset
        );

    context->IASetIndexBuffer(
        m_pIndexBuffer.Get(),
        DXGI_FORMAT_R16_UINT,
        0
        );

    context->IASetPrimitiveTopology(
        D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST
        );

    context->IASetInputLayout(m_pInputLayout.Get());

    // Set up the vertex shader stage.
    context->VSSetShader(
        m_pVertexShader.Get(),
        nullptr,
        0
        );

    context->VSSetConstantBuffers(
        0,
        1,
        m_pConstantBuffer.GetAddressOf()
        );

    // Set up the pixel shader stage.
    context->PSSetShader(
        m_pPixelShader.Get(),
        nullptr,
        0
        );

    // Calling Draw tells Direct3D to start sending commands to the graphics device.
    context->DrawIndexed(
        m_indexCount,
        0,
        0
        );
}
```

It's good practice to set the various graphics pipeline stages on the context in order. Typically, the order is:

- Refresh constant buffer resources with new data as needed (using data from **Update**).
- Input assembly (IA): This is where we attach the vertex and index buffers that define the scene geometry. You need to attach each vertex and index buffer for each object in the scene. Because this example has just the cube, it's pretty simple.
- Vertex shader (VS): Attach any vertex shaders that will transform the data in the vertex buffers, and attach constant buffers for the vertex shader.
- Pixel shader (PS): Attach any pixel shaders that will perform per-pixel operations in the rasterized scene, and attach device resources for the pixel shader (constant buffers, textures, and so on).
- Output merger (OM): This is the stage where pixels are blended, after the shaders are finished. This is an exception to the rule, because you attach your depth stencils and render targets before setting any of the other stages. You may have multiple stencils and targets if you have additional vertex and pixel shaders that generate textures such as shadow maps, height maps, or other sampling techniques - in this case, each drawing pass will need the appropriate target(s) set before you call a draw function.

Next, in the final section (Work with shaders and shader resources), we'll look at the shaders and discuss how Direct3D executes them.

Related topics

Up next

Work with shaders and shader resources

Work with shaders and shader resources

12/13/2021 • 13 minutes to read • Edit Online

It's time to learn how to work with shaders and shader resources in developing your Microsoft DirectX game for Windows 8. We've seen how to set up the graphics device and resources, and perhaps you've even started modifying its pipeline. So now let's look at pixel and vertex shaders.

If you aren't familiar with shader languages, a quick discussion is in order. Shaders are small, low-level programs that are compiled and run at specific stages in the graphics pipeline. Their specialty is very fast floating-point mathematical operations. The most common shader programs are:

- **Vertex shader**—Executed for each vertex in a scene. This shader operates on vertex buffer elements provided to it by the calling app, and minimally results in a 4-component position vector that will be rasterized into a pixel position.
- **Pixel shader**—Executed for each pixel in a render target. This shader receives rasterized coordinates from previous shader stages (in the simplest pipelines, this would be the vertex shader) and returns a color (or other 4-component value) for that pixel position, which is then written into a render target.

This example includes very basic vertex and pixel shaders that only draw geometry, and more complex shaders that add basic lighting calculations.

Shader programs are written in Microsoft High Level Shader Language (HLSL). HLSL syntax looks a lot like C, but without the pointers. Shader programs must be very compact and efficient. If your shader compiles to too many instructions, it cannot be run and an error is returned. (Note that the exact number of instructions allowed is part of the Direct3D feature level.)

In Direct3D, shaders are not compiled at run time; they are compiled when the rest of the program is compiled. When you compile your app with Microsoft Visual Studio 2013, the HLSL files are compiled to CSO (.cso) files that your app must load and place in GPU memory prior to drawing. Make sure you include these CSO files with your app when you package it; they are assets just like meshes and textures.

Understand HLSL semantics

It's important to take a moment to discuss HLSL semantics before we continue, because they are often a point of confusion for new Direct3D developers. HLSL semantics are strings that identify a value passed between the app and a shader program. Although they can be any of a variety of possible strings, the best practice is to use a string like `POSITION` or `COLOR` that indicates the usage. You assign these semantics when you are constructing a constant buffer or input layout. You can also append a number between 0 and 7 to the semantic so that you use separate registers for similar values. For example: COLOR0, COLOR1, COLOR2...

Semantics that are prefixed with "SV_" are *system value* semantics that are written to by your shader program; your game itself (running on the CPU) cannot modify them. Typically, these semantics contain values that are inputs or outputs from another shader stage in the graphics pipeline, or that are generated entirely by the GPU.

Additionally, `SV_` semantics have different behaviors when they are used to specify input to or output from a shader stage. For example, `SV_POSITION` (output) contains the vertex data transformed during the vertex shader stage, and `SV_POSITION` (*input*) contains the pixel position values that were interpolated by the GPU during the rasterization stage.

Here are a few common HLSL semantics:

- `POSITION`(*n*) for vertex buffer data. `SV_POSITION` provides a pixel position to the pixel shader and cannot be

written by your game.

- `NORMAL` (*n*) for normal data provided by the vertex buffer.
- `TEXCOORD` (*n*) for texture UV coordinate data supplied to a shader.
- `COLOR` (n) for RGBA color data supplied to a shader. Note that it is treated identically to coordinate data, including interpolating the value during rasterization; the semantic simply helps you identify that it is color data.
- `SV_Target` [n] for writing from a pixel shader to a target texture or other pixel buffer.

We'll see some examples of HLSL semantics as we review the example.

Read from the constant buffers

Any shader can read from a constant buffer if that buffer is attached to its stage as a resource. In this example, only the vertex shader is assigned a constant buffer.

The constant buffer is declared in two places: in the C++ code, and in the corresponding HLSL files that will access it.

Here's how the constant buffer struct is declared in the C++ code.

```
typedef struct _constantBufferStruct {
    DirectX::XMFLOAT4X4 world;
    DirectX::XMFLOAT4X4 view;
    DirectX::XMFLOAT4X4 projection;
} ConstantBufferStruct;
```

When declaring the structure for the constant buffer in your C++ code, ensure that all of the data is correctly aligned along 16-byte boundaries. The easiest way to do this is to use DirectXMath types, like **XMFLOAT4** or **XMFLOAT4X4**, as seen in the example code. You can also guard against misaligned buffers by declaring a static assert:

```
// Assert that the constant buffer remains 16-byte aligned.
static_assert((sizeof(ConstantBufferStruct) % 16) == 0, "Constant Buffer size must be 16-byte aligned");
```

This line of code will cause an error at compile time if **ConstantBufferStruct** is not 16-byte aligned. For more information about constant buffer alignment and packing, see Packing Rules for Constant Variables.

Now, here's how the constant buffer is declared in the vertex shader HLSL.

```
cbuffer ModelViewProjectionConstantBuffer : register(b0)
{
    matrix mWorld;      // world matrix for object
    matrix View;        // view matrix
    matrix Projection;  // projection matrix
};
```

All buffers—constant, texture, sampler, or other—must have a register defined so the GPU can access them. Each shader stage allows up to 15 constant buffers, and each buffer can hold up to 4,096 constant variables. The register-usage declaration syntax is as follows:

- **b***#*: A register for a constant buffer (**cbuffer**).
- **t***#*: A register for a texture buffer (**tbuffer**).
- **s***#*: A register for a sampler. (A sampler defines the lookup behavior for texels in the texture resource.)

For example, the HLSL for a pixel shader might take a texture and a sampler as input with a declaration like this.

```
Texture2D simpleTexture : register(t0);
SamplerState simpleSampler : register(s0);
```

It's up to you to assign constant buffers to registers—when you set up the pipeline, you attach a constant buffer to the same slot you assigned it to in the HLSL file. For example, in the previous topic the call to VSSetConstantBuffers indicates '0' for the first parameter. That tells Direct3D to attach the constant buffer resource to register 0, which matches the buffer's assignment to **register(b0)** in the HLSL file.

Read from the vertex buffers

The vertex buffer supplies the triangle data for the scene objects to the vertex shader(s). As with the constant buffer, the vertex buffer struct is declared in the C++ code, using similar packing rules.

```
typedef struct _vertexPositionColor
{
    DirectX::XMFLOAT3 pos;
    DirectX::XMFLOAT3 color;
} VertexPositionColor;
```

There is no standard format for vertex data in Direct3D 11. Instead, we define our own vertex data layout using a descriptor; the data fields are defined using an array of D3D11_INPUT_ELEMENT_DESC structures. Here, we show a simple input layout that describes the same vertex format as the preceding struct:

```
D3D11_INPUT_ELEMENT_DESC iaDesc [] =
{
    { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT,
    0, 0, D3D11_INPUT_PER_VERTEX_DATA, 0 },

    { "COLOR", 0, DXGI_FORMAT_R32G32B32_FLOAT,
    0, 12, D3D11_INPUT_PER_VERTEX_DATA, 0 },
};

hr = device->CreateInputLayout(
    iaDesc,
    ARRAYSIZE(iaDesc),
    bytes,
    bytesRead,
    &m_pInputLayout
    );
```

If you add data to the vertex format when modifying the example code, be sure to update the input layout as well, or the shader will not be able to interpret it. You might modify the vertex layout like this:

```
typedef struct _vertexPositionColorTangent
{
    DirectX::XMFLOAT3 pos;
    DirectX::XMFLOAT3 normal;
    DirectX::XMFLOAT3 tangent;
} VertexPositionColorTangent;
```

In that case, you'd modify the input-layout definition as follows.

```
D3D11_INPUT_ELEMENT_DESC iaDescExtended[] =
{
    { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT,
    0, 0, D3D11_INPUT_PER_VERTEX_DATA, 0 },

    { "NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT,
    0, 12, D3D11_INPUT_PER_VERTEX_DATA, 0 },

    { "TANGENT", 0, DXGI_FORMAT_R32G32B32_FLOAT,
    0, 12, D3D11_INPUT_PER_VERTEX_DATA, 0 },
};

hr = device->CreateInputLayout(
    iaDesc,
    ARRAYSIZE(iaDesc),
    bytes,
    bytesRead,
    &m_pInputLayoutExtended
    );
```

Each of the input-layout element definitions is prefixed with a string, like "POSITION" or "NORMAL"—that is the semantic we discussed earlier in this topic. It's like a handle that helps the GPU identify that element when processing the vertex. Choose common, meaningful names for your vertex elements.

Just as with the constant buffer, the vertex shader has a corresponding buffer definition for incoming vertex elements. (That's why we provided a reference to the vertex shader resource when creating the input layout - Direct3D validates the per-vertex data layout with the shader's input struct.) Note how the semantics match between the input layout definition and this HLSL buffer declaration. However, `COLOR` has a "0" appended to it. It isn't necessary to add the 0 if you have only one `COLOR` element declared in the layout, but it's a good practice to append it in case you you choose to add more color elements in the future.

```
struct VS_INPUT
{
    float3 vPos   : POSITION;
    float3 vColor : COLOR0;
};
```

Pass data between shaders

Shaders take input types and return output types from their main functions upon execution. For the vertex shader defined in the previous section, the input type was the VS_INPUT structure, and we defined a matching input layout and C++ struct. An array of this struct is used to create a vertex buffer in the **CreateCube** method.

The vertex shader returns a PS_INPUT structure, which must minimally contain the 4-component (float4) final vertex position. This position value must have the system value semantic, `SV_POSITION`, declared for it so the GPU has the data it needs to perform the next drawing step. Notice that there is not a 1:1 correspondence between vertex shader output and pixel shader input; the vertex shader returns one structure for each vertex it is given, but the pixel shader runs once for each pixel. That's because the per-vertex data first passes through the rasterization stage. This stage decides which pixels "cover" the geometry you're drawing, computes interpolated per-vertex data for each pixel, and then calls the pixel shader once for each of those pixels. Interpolation is the default behavior when rasterizing output values, and is essential in particular for the correct processing of output vector data (light vectors, per-vertex normals and tangents, and others).

```
struct PS_INPUT
{
    float4 Position : SV_POSITION;  // interpolated vertex position (system value)
    float4 Color    : COLOR0;       // interpolated diffuse color
};
```

Review the vertex shader

The example vertex shader is very simple: take in a vertex (position and color), transform the position from model coordinates into perspective projected coordinates, and return it (along with the color) to the rasterizer. Notice that the color value is interpolated right along with the position data, providing a different value for each pixel even though the vertex shader didn't perform any calculations on the color value.

```
VS_OUTPUT main(VS_INPUT input) // main is the default function name
{
    VS_OUTPUT Output;

    float4 pos = float4(input.vPos, 1.0f);

    // Transform the position from object space to homogeneous projection space
    pos = mul(pos, mWorld);
    pos = mul(pos, View);
    pos = mul(pos, Projection);
    Output.Position = pos;

    // Just pass through the color data
    Output.Color = float4(input.vColor, 1.0f);

    return Output;
}
```

A more complex vertex shader, such as one that sets up an object's vertices for Phong shading, might look more like this. In this case, we're taking advantage of the fact that the vectors and normals are interpolated to approximate a smooth-looking surface.

```
// A constant buffer that stores the three basic column-major matrices for composing geometry.
cbuffer ModelViewProjectionConstantBuffer : register(b0)
{
    matrix model;
    matrix view;
    matrix projection;
};

cbuffer LightConstantBuffer : register(b1)
{
    float4 lightPos;
};

struct VertexShaderInput
{
    float3 pos : POSITION;
    float3 normal : NORMAL;
};

// Per-pixel color data passed through the pixel shader.

struct PixelShaderInput
{
    float4 position : SV_POSITION;
    float3 outVec : POSITION0;
    float3 outNormal : NORMAL0;
    float3 outLightVec : POSITION1;
};

PixelShaderInput main(VertexShaderInput input)
{
    // Inefficient -- doing this only for instruction. Normally, you would
 // premultiply them on the CPU and place them in the cbuffer.
    matrix mvMatrix = mul(model, view);
    matrix mvpMatrix = mul(mvMatrix, projection);

    PixelShaderInput output;

    float4 pos = float4(input.pos, 1.0f);
    float4 normal = float4(input.normal, 1.0f);
    float4 light = float4(lightPos.xyz, 1.0f);

    //
    float4 eye = float4(0.0f, 0.0f, -2.0f, 1.0f);

    // Transform the vertex position into projected space.
    output.gl_Position = mul(pos, mvpMatrix);
    output.outNormal = mul(normal, mvMatrix).xyz;
    output.outVec = -(eye - mul(pos, mvMatrix)).xyz;
    output.outLightVec = mul(light, mvMatrix).xyz;

    return output;
}
```

Review the pixel shader

This pixel shader in this example is quite possibly the absolute minimum amount of code you can have in a pixel shader. It takes the interpolated pixel color data generated during rasterization and returns it as output, where it will be written to a render target. How boring!

```
PS_OUTPUT main(PS_INPUT In)
{
    PS_OUTPUT Output;

    Output.RGBColor = In.Color;

    return Output;
}
```

The important part is the `SV_TARGET` system-value semantic on the return value. It indicates that the output is to be written to the primary render target, which is the texture buffer supplied to the swap chain for display. This is required for pixel shaders - without the color data from the pixel shader, Direct3D wouldn't have anything to display!

An example of a more complex pixel shader to perform Phong shading might look like this. Since the vectors and normals were interpolated, we don't have to compute them on a per-pixel basis. However, we do have to re-normalize them because of how interpolation works; conceptually, we need to gradually "spin" the vector from direction at vertex A to direction at vertex B, maintaining its length—wheras interpolation instead cuts across a straight line between the two vector endpoints.

```
cbuffer MaterialConstantBuffer : register(b2)
{
    float4 lightColor;
    float4 Ka;
    float4 Kd;
    float4 Ks;
    float4 shininess;
};

struct PixelShaderInput
{
    float4 position : SV_POSITION;
    float3 outVec : POSITION0;
    float3 normal : NORMAL0;
    float3 light : POSITION1;
};

float4 main(PixelShaderInput input) : SV_TARGET
{
    float3 L = normalize(input.light);
    float3 V = normalize(input.outVec);
    float3 R = normalize(reflect(L, input.normal));

    float4 diffuse = Ka + (lightColor * Kd * max(dot(input.normal, L), 0.0f));
    diffuse = saturate(diffuse);

    float4 specular = Ks * pow(max(dot(R, V), 0.0f), shininess.x - 50.0f);
    specular = saturate(specular);

    float4 finalColor = diffuse + specular;

    return finalColor;
}
```

In another example, the pixel shader takes its own constant buffers that contain light and material information. The input layout in the vertex shader would be expanded to include normal data, and the output from that vertex shader is expected to include transformed vectors for the vertex, the light, and the vertex normal in the view coordinate system.

If you have texture buffers and samplers with assigned registers (**t** and **s**, respectively), you can access them in the pixel shader also.

```
Texture2D simpleTexture : register(t0);
SamplerState simpleSampler : register(s0);

struct PixelShaderInput
{
    float4 pos : SV_POSITION;
    float3 norm : NORMAL;
    float2 tex : TEXCOORD0;
};

float4 SimplePixelShader(PixelShaderInput input) : SV_TARGET
{
    float3 lightDirection = normalize(float3(1, -1, 0));
    float4 texelColor = simpleTexture.Sample(simpleSampler, input.tex);
    float lightMagnitude = 0.8f * saturate(dot(input.norm, -lightDirection)) + 0.2f;
    return texelColor * lightMagnitude;
}
```

Shaders are very powerful tools that can be used to generate procedural resources like shadow maps or noise textures. In fact, advanced techniques require that you think of textures more abstractly, not as visual elements but as buffers. They hold data like height information, or other data that can be sampled in the final pixel shader pass or in that particular frame as part of a multi-stage effects pass. Multi-sampling is a powerful tool and the backbone of many modern visual effects.

Next steps

Hopefully, you're comfortable with DirectX 11at this point and are ready to start working on your project. Here are some links to help answer other questions you may have about development with DirectX and C++:

- Developing games
- Use Visual Studio tools for DirectX game programming
- DirectX game development and sample walkthroughs
- Additional game programming resources

Related topics

Work with DirectX device resources

Understand the Direct3D 11 rendering pipeline

Complete code for a DirectX framework

12/13/2021 • 17 minutes to read • Edit Online

This topic provides the complete code sample used in the tutorial Get started with DirectX for Windows.

This code assumes that you are using Microsoft Visual Studio 2013, and that you have created an empty Win32 project.

This topic contains these sections:

Download location

This sample is not available for download.

Technologies

REQUIREMENT	VALUE
Programming languages	C++
Programming models	Windows/C++

Requirements

REQUIREMENT	VALUE
Minimum supported client	Windows 8.1
Minimum supported server	Windows Server 2012 R2
Minimum required SDK	Visual Studio 2013

View the code (C++)

Cube.cpp

```
//// THIS CODE AND INFORMATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF
//// ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO
//// THE IMPLIED WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A
//// PARTICULAR PURPOSE.
////
//// Copyright (c) Microsoft Corporation. All rights reserved
```

```
//--------------------------------------------------------------------------------
// File: Cube.cpp
//
// Desktop app that renders a spinning, colorful cube.
//
//--------------------------------------------------------------------------------


//--------------------------------------------------------------------------------
// Includes
//--------------------------------------------------------------------------------
#include "dxstdafx.h"
#include "resource.h"

#include <string>
#include <memory>

#include "DeviceResources.h"
#include "Renderer.h"
#include "MainClass.h"

//--------------------------------------------------------------------------------
// Main function: Creates window, calls initialization functions, and hosts
// the render loop.
//--------------------------------------------------------------------------------
INT WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
    HRESULT hr = S_OK;

    // Enable run-time memory check for debug builds.
#if defined(DEBUG) | defined(_DEBUG)
    _CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF);
#endif

    // Begin initialization.

    // Instantiate the window manager class.
    std::shared_ptr<MainClass> winMain = std::shared_ptr<MainClass>(new MainClass());
    // Create a window.
    hr = winMain->CreateDesktopWindow();

    if (SUCCEEDED(hr))
    {
        // Instantiate the device manager class.
        std::shared_ptr<DeviceResources> deviceResources = std::shared_ptr<DeviceResources>(new
DeviceResources());
        // Create device resources.
        deviceResources->CreateDeviceResources();

        // Instantiate the renderer.
        std::shared_ptr<Renderer> renderer = std::shared_ptr<Renderer>(new Renderer(deviceResources));
        renderer->CreateDeviceDependentResources();

        // We have a window, so initialize window size-dependent resources.
        deviceResources->CreateWindowResources(winMain->GetWindowHandle());
        renderer->CreateWindowSizeDependentResources();

        // Go full-screen.
        deviceResources->GoFullScreen();

        // Whoops! We resized the "window" when we went full-screen. Better
        // tell the renderer.
        renderer->CreateWindowSizeDependentResources();

        // Run the program.
        hr = winMain->Run(deviceResources, renderer);
    }

    // Cleanup is handled in destructors.
```

```
    return hr;
}
```

MainClass.h

```
//// THIS CODE AND INFORMATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF
//// ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO
//// THE IMPLIED WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A
//// PARTICULAR PURPOSE.
////
//// Copyright (c) Microsoft Corporation. All rights reserved
#pragma once


//--------------------------------------------------------------------------------
// File: Cube.cpp
//
// Desktop app that renders a spinning, colorful cube.
//
//--------------------------------------------------------------------------------

//--------------------------------------------------------------------------------
// Includes
//--------------------------------------------------------------------------------
#include "DeviceResources.h"
#include "Renderer.h"

//--------------------------------------------------------------------------------
// Class declarations
//--------------------------------------------------------------------------------

class MainClass
{
public:
    MainClass();
    ~MainClass();

    HRESULT CreateDesktopWindow();

    HWND GetWindowHandle() { return m_hWnd; };

    static LRESULT CALLBACK StaticWindowProc(
        HWND hWnd,
        UINT uMsg,
        WPARAM wParam,
        LPARAM lParam
        );

    HRESULT Run(
        std::shared_ptr<DeviceResources> deviceResources,
        std::shared_ptr<Renderer> renderer
        );

private:
    //--------------------------------------------------------------------------------
    // Desktop window resources
    //--------------------------------------------------------------------------------
    HMENU     m_hMenu;
    RECT      m_rc;
    HWND      m_hWnd;
};

// These are STATIC because this sample only creates one window.
// If your app can have multiple windows, MAKE SURE this is dealt with
// differently.
static HINSTANCE m_hInstance;
static std::wstring m_windowClassName;
```

MainClass.cpp

```
//// THIS CODE AND INFORMATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF
//// ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO
//// THE IMPLIED WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A
//// PARTICULAR PURPOSE.
////
//// Copyright (c) Microsoft Corporation. All rights reserved


//--------------------------------------------------------------------------------
// File: MainClass.cpp
//
// Creates and owns a desktop window resource.
//
//--------------------------------------------------------------------------------


//--------------------------------------------------------------------------------
// Includes
//--------------------------------------------------------------------------------
#include "dxstdafx.h"
#include "resource.h"

#include <string>
#include <memory>

#include "MainClass.h"




//--------------------------------------------------------------------------------
// Constructor
//--------------------------------------------------------------------------------
MainClass::MainClass()
{
    m_windowClassName = L"Direct3DWindowClass";
    m_hInstance = NULL;
}

//--------------------------------------------------------------------------------
// Create a window for our Direct3D viewport.
//--------------------------------------------------------------------------------
HRESULT MainClass::CreateDesktopWindow()
{
    // Window resources are dealt with here.

    if(m_hInstance == NULL)
        m_hInstance = (HINSTANCE)GetModuleHandle(NULL);

    HICON hIcon = NULL;
    WCHAR szExePath[MAX_PATH];
        GetModuleFileName(NULL, szExePath, MAX_PATH);

    // If the icon is NULL, then use the first one found in the exe
    if(hIcon == NULL)
        hIcon = ExtractIcon(m_hInstance, szExePath, 0);

    // Register the windows class
    WNDCLASS wndClass;
    wndClass.style = CS_DBLCLKS;
    wndClass.lpfnWndProc = MainClass::StaticWindowProc;
    wndClass.cbClsExtra = 0;
    wndClass.cbWndExtra = 0;
    wndClass.hInstance = m_hInstance;
    wndClass.hIcon = hIcon;
    wndClass.hCursor = LoadCursor(NULL, IDC_ARROW);
    wndClass.hbrBackground = (HBRUSH)GetStockObject(BLACK_BRUSH);
    wndClass.lpszMenuName = NULL;
    wndClass.lpszClassName = m_windowClassName.c_str();
```

```
    if(!RegisterClass(&wndClass))
    {
        DWORD dwError = GetLastError();
        if(dwError != ERROR_CLASS_ALREADY_EXISTS)
            return HRESULT_FROM_WIN32(dwError);
    }

    m_rc;
    int x = CW_USEDEFAULT;
    int y = CW_USEDEFAULT;

    // No menu in this example.
    m_hMenu = NULL;

    // This example uses a non-resizable 640 by 480 viewport for simplicity.
    int nDefaultWidth = 640;
    int nDefaultHeight = 480;
    SetRect(&m_rc, 0, 0, nDefaultWidth, nDefaultHeight);
    AdjustWindowRect(
        &m_rc,
        WS_OVERLAPPEDWINDOW,
        (m_hMenu != NULL) ? true : false
        );

    // Create the window for our viewport.
    m_hWnd = CreateWindow(
        m_windowClassName.c_str(),
        L"Cube11",
        WS_OVERLAPPEDWINDOW,
        x, y,
        (m_rc.right-m_rc.left), (m_rc.bottom-m_rc.top),
        0,
        m_hMenu,
        m_hInstance,
        0
        );

    if(m_hWnd == NULL)
    {
        DWORD dwError = GetLastError();
        return HRESULT_FROM_WIN32(dwError);
    }

    return S_OK;
}

HRESULT MainClass::Run(
    std::shared_ptr<DeviceResources> deviceResources,
    std::shared_ptr<Renderer> renderer
    )
{
    HRESULT hr = S_OK;

    if (!IsWindowVisible(m_hWnd))
        ShowWindow(m_hWnd, SW_SHOW);

    // The render loop is controlled here.
    bool bGotMsg;
    MSG  msg;
    msg.message = WM_NULL;
    PeekMessage(&msg, NULL, 0U, 0U, PM_NOREMOVE);

    while (WM_QUIT != msg.message)
    {
        // Process window events.
        // Use PeekMessage() so we can use idle time to render the scene.
        bGotMsg = (PeekMessage(&msg, NULL, 0U, 0U, PM_REMOVE) != 0);
```

```
        if (bGotMsg)
        {
            // Translate and dispatch the message
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
        else
        {
            // Update the scene.
            renderer->Update();

            // Render frames during idle time (when no messages are waiting).
            renderer->Render();

            // Present the frame to the screen.
            deviceResources->Present();
        }
    }

    return hr;
}


//--------------------------------------------------------------------------------
// Destructor.
//--------------------------------------------------------------------------------
MainClass::~MainClass()
{

}

//--------------------------------------------------------------------------------
// Process windows messages. This looks for window close events, letting us
// exit out of the sample.
//--------------------------------------------------------------------------------
LRESULT CALLBACK MainClass::StaticWindowProc(
    HWND hWnd,
    UINT uMsg,
    WPARAM wParam,
    LPARAM lParam
    )
{
    switch(uMsg)
    {
        case WM_CLOSE:
        {
            HMENU hMenu;
            hMenu = GetMenu(hWnd);
            if (hMenu != NULL)
            {
                DestroyMenu(hMenu);
            }
            DestroyWindow(hWnd);
            UnregisterClass(
                m_windowClassName.c_str(),
                m_hInstance
                );
            return 0;
        }

        case WM_DESTROY:
            PostQuitMessage(0);
            break;
    }

    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
```

DeviceResources.h

```
//// THIS CODE AND INFORMATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF
//// ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO
//// THE IMPLIED WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A
//// PARTICULAR PURPOSE.
////
//// Copyright (c) Microsoft Corporation. All rights reserved
#pragma once


//--------------------------------------------------------------------------------
// File: Cube.cpp
//
// Desktop app that renders a spinning, colorful cube.
//
//--------------------------------------------------------------------------------


//--------------------------------------------------------------------------------
// Class declarations
//--------------------------------------------------------------------------------

class DeviceResources
{
public:
    DeviceResources();
    ~DeviceResources();

    HRESULT CreateDeviceResources( HWND hWnd );
    HRESULT CreateDeviceResources( );
    HRESULT CreateWindowResources( HWND hWnd );

    HRESULT ConfigureBackBuffer();
    HRESULT ReleaseBackBuffer();
    HRESULT GoFullScreen();
    HRESULT GoWindowed();

    float GetAspectRatio();

    ID3D11Device*           GetDevice() { return m_pd3dDevice.Get(); };
    ID3D11DeviceContext*    GetDeviceContext() { return m_pd3dDeviceContext.Get(); };
    ID3D11RenderTargetView* GetRenderTarget() { return m_pRenderTarget.Get(); }
    ID3D11DepthStencilView* GetDepthStencil() { return m_pDepthStencilView.Get(); }

    void Present();

private:

    //--------------------------------------------------------------------------------
    // Direct3D device
    //--------------------------------------------------------------------------------
    Microsoft::WRL::ComPtr<ID3D11Device>        m_pd3dDevice;
    Microsoft::WRL::ComPtr<ID3D11DeviceContext> m_pd3dDeviceContext; // immediate context
    Microsoft::WRL::ComPtr<IDXGISwapChain>      m_pDXGISwapChain;


    //--------------------------------------------------------------------------------
    // DXGI swap chain device resources
    //--------------------------------------------------------------------------------
    Microsoft::WRL::ComPtr < ID3D11Texture2D>        m_pBackBuffer;
    Microsoft::WRL::ComPtr < ID3D11RenderTargetView> m_pRenderTarget;


    //--------------------------------------------------------------------------------
    // Direct3D device resources for the depth stencil
    //--------------------------------------------------------------------------------
```

```
    Microsoft::WRL::ComPtr<ID3D11Texture2D>         m_pDepthStencil;
    Microsoft::WRL::ComPtr<ID3D11DepthStencilView>  m_pDepthStencilView;


    //---------------------------------------------------------------------------
    // Direct3D device metadata and device resource metadata
    //---------------------------------------------------------------------------
    D3D_FEATURE_LEVEL       m_featureLevel;
    D3D11_TEXTURE2D_DESC    m_bbDesc;
    D3D11_VIEWPORT          m_viewport;
};
```

DeviceResources.cpp

```
//// THIS CODE AND INFORMATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF
//// ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO
//// THE IMPLIED WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A
//// PARTICULAR PURPOSE.
////
//// Copyright (c) Microsoft Corporation. All rights reserved


//---------------------------------------------------------------------------
// File: Cube.cpp
//
// Creates and owns a DirectX virtual device, along with accompanying DXGI swap
// chain and Direct3D device resources.
//
//---------------------------------------------------------------------------


//---------------------------------------------------------------------------
// Includes
//---------------------------------------------------------------------------
#include "dxstdafx.h"
#include "resource.h"

#include <string>
#include <memory>

#include "DeviceResources.h"


//---------------------------------------------------------------------------
// Constructor
//---------------------------------------------------------------------------
DeviceResources::DeviceResources()
{

};


//---------------------------------------------------------------------------
//
// Method 1: Create device and swap chain at the same time.
//
// Benefit:  It's easy.
// Drawback: You have to create a new device, and therefore
//           reload all DirectX device resources, every time
//           you recreate the swap chain.
//
//---------------------------------------------------------------------------
HRESULT DeviceResources::CreateDeviceResources(HWND hWnd)
{
    HRESULT hr = S_OK;

    D3D_FEATURE_LEVEL levels[] = {
```

```
    D3D_FEATURE_LEVEL levels [] = {
        D3D_FEATURE_LEVEL_9_1,
        D3D_FEATURE_LEVEL_9_2,
        D3D_FEATURE_LEVEL_9_3,
        D3D_FEATURE_LEVEL_10_0,
        D3D_FEATURE_LEVEL_10_1,
        D3D_FEATURE_LEVEL_11_0,
        D3D_FEATURE_LEVEL_11_1
    };

    // This flag adds support for surfaces with a color-channel ordering different
    // from the API default. It is required for compatibility with Direct2D.
    UINT deviceFlags = D3D11_CREATE_DEVICE_BGRA_SUPPORT;

#if defined(DEBUG) || defined(_DEBUG)
    deviceFlags |= D3D11_CREATE_DEVICE_DEBUG;
#endif

    DXGI_SWAP_CHAIN_DESC desc;
    ZeroMemory(&desc, sizeof(DXGI_SWAP_CHAIN_DESC));
    desc.Windowed = TRUE;
    desc.BufferCount = 2;
    desc.BufferDesc.Format = DXGI_FORMAT_B8G8R8A8_UNORM;
    desc.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;
    desc.SampleDesc.Count = 1;      //multisampling setting
    desc.SampleDesc.Quality = 0;    //vendor-specific flag
    desc.SwapEffect = DXGI_SWAP_EFFECT_FLIP_SEQUENTIAL;
    desc.OutputWindow = hWnd;

    Microsoft::WRL::ComPtr<ID3D11Device> device;
    Microsoft::WRL::ComPtr<ID3D11DeviceContext> context;
    Microsoft::WRL::ComPtr<IDXGISwapChain> swapChain;

    hr = D3D11CreateDeviceAndSwapChain(
        nullptr,
        D3D_DRIVER_TYPE::D3D_DRIVER_TYPE_HARDWARE,
        nullptr,
        deviceFlags,
        levels,
        ARRAYSIZE(levels),
        D3D11_SDK_VERSION,
        &desc,
        swapChain.GetAddressOf(),
        device.GetAddressOf(),
        &m_featureLevel,
        context.GetAddressOf()
        );

    device.As(&m_pd3dDevice);
    context.As(&m_pd3dDeviceContext);
    swapChain.As(&m_pDXGISwapChain);

    // Configure the back buffer and viewport.
    hr = m_pDXGISwapChain->GetBuffer(
        0,
        __uuidof(ID3D11Texture2D),
        (void**) &m_pBackBuffer);

    m_pBackBuffer->GetDesc(&m_bbDesc);

    ZeroMemory(&m_viewport, sizeof(D3D11_VIEWPORT));
    m_viewport.Height = (float) m_bbDesc.Height;
    m_viewport.Width = (float) m_bbDesc.Width;
    m_viewport.MinDepth = 0;
    m_viewport.MaxDepth = 1;


    m_pd3dDeviceContext->RSSetViewports(
        1,
```

```
        &m_viewport
        );

    hr = m_pd3dDevice->CreateRenderTargetView(
        m_pBackBuffer.Get(),
        nullptr,
        m_pRenderTarget.GetAddressOf()
        );

    return hr;
}

//----------------------------------------------------------------------------
//
// Method 2: Create the device and swap chain separately.
//
// Benefit:  You can recreate the swap chain on-the-fly.
// Drawback: Slight increase in your initial investment.
//
//----------------------------------------------------------------------------
HRESULT DeviceResources::CreateDeviceResources()
{
    HRESULT hr = S_OK;

    D3D_FEATURE_LEVEL levels[] = {
        D3D_FEATURE_LEVEL_9_1,
        D3D_FEATURE_LEVEL_9_2,
        D3D_FEATURE_LEVEL_9_3,
        D3D_FEATURE_LEVEL_10_0,
        D3D_FEATURE_LEVEL_10_1,
        D3D_FEATURE_LEVEL_11_0,
        D3D_FEATURE_LEVEL_11_1
    };

    // This flag adds support for surfaces with a color-channel ordering different
    // from the API default. It is required for compatibility with Direct2D.
    UINT deviceFlags = D3D11_CREATE_DEVICE_BGRA_SUPPORT;

#if defined(DEBUG) || defined(_DEBUG)
    deviceFlags |= D3D11_CREATE_DEVICE_DEBUG;
#endif

    // Create the Direct3D 11 API device object and a corresponding context.
    Microsoft::WRL::ComPtr<ID3D11Device>        device;
    Microsoft::WRL::ComPtr<ID3D11DeviceContext> context;

    hr = D3D11CreateDevice(
        nullptr,                    // Specify nullptr to use the default adapter.
        D3D_DRIVER_TYPE_HARDWARE,   // Create a device using the hardware graphics driver.
        0,                          // Should be 0 unless the driver is D3D_DRIVER_TYPE_SOFTWARE.
        deviceFlags,                // Set debug and Direct2D compatibility flags.
        levels,                     // List of feature levels this app can support.
        ARRAYSIZE(levels),          // Size of the list above.
        D3D11_SDK_VERSION,          // Always set this to D3D11_SDK_VERSION for Windows Store apps.
        &device,                    // Returns the Direct3D device created.
        &m_featureLevel,            // Returns feature level of device created.
        &context                    // Returns the device immediate context.
        );

    if (FAILED(hr))
    {
        // Handle device interface creation failure if it occurs.
        // For example, reduce the feature level requirement, or fail over
        // to WARP rendering.
    }

    // Store pointers to the Direct3D 11.1 API device and immediate context.
    device.As(&m_pd3dDevice);
    context.As(&m_pd3dDeviceContext);
```

```
    return hr;
}

//----------------------------------------------------------------------------
// Method 2, continued. Creates the swap chain.
//----------------------------------------------------------------------------
HRESULT DeviceResources::CreateWindowResources(HWND hWnd)
{
    HRESULT hr = S_OK;


    DXGI_SWAP_CHAIN_DESC desc;
    ZeroMemory(&desc, sizeof(DXGI_SWAP_CHAIN_DESC));
    desc.Windowed = TRUE; // Sets the initial state of full-screen mode.
    desc.BufferCount = 2;
    desc.BufferDesc.Format = DXGI_FORMAT_B8G8R8A8_UNORM;
    desc.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;
    desc.SampleDesc.Count = 1;      //multisampling setting
    desc.SampleDesc.Quality = 0;    //vendor-specific flag
    desc.SwapEffect = DXGI_SWAP_EFFECT_FLIP_SEQUENTIAL;
    desc.OutputWindow = hWnd;

    // Create the DXGI device object to use in other factories, such as Direct2D.
    Microsoft::WRL::ComPtr<IDXGIDevice3> dxgiDevice;
    m_pd3dDevice.As(&dxgiDevice);

    // Create swap chain.
    Microsoft::WRL::ComPtr<IDXGIAdapter> adapter;
    Microsoft::WRL::ComPtr<IDXGIFactory> factory;

    hr = dxgiDevice->GetAdapter(&adapter);

    if (SUCCEEDED(hr))
    {
        adapter->GetParent(IID_PPV_ARGS(&factory));

        hr = factory->CreateSwapChain(
            m_pd3dDevice.Get(),
            &desc,
            &m_pDXGISwapChain
            );
    }

    // Configure the back buffer, stencil buffer, and viewport.
    hr = ConfigureBackBuffer();

    return hr;
}

HRESULT DeviceResources::ConfigureBackBuffer()
{
    HRESULT hr = S_OK;

    hr = m_pDXGISwapChain->GetBuffer(
        0,
        __uuidof(ID3D11Texture2D),
        (void**) &m_pBackBuffer);

    hr = m_pd3dDevice->CreateRenderTargetView(
        m_pBackBuffer.Get(),
        nullptr,
        m_pRenderTarget.GetAddressOf()
        );

    m_pBackBuffer->GetDesc(&m_bbDesc);

    // Create a depth-stencil view for use with 3D rendering if needed.
    CD3D11_TEXTURE2D_DESC depthStencilDesc(
```

```
        DXGI_FORMAT_D24_UNORM_S8_UINT,
        static_cast<UINT> (m_bbDesc.Width),
        static_cast<UINT> (m_bbDesc.Height),
        1, // This depth stencil view has only one texture.
        1, // Use a single mipmap level.
        D3D11_BIND_DEPTH_STENCIL
        );

    m_pd3dDevice->CreateTexture2D(
        &depthStencilDesc,
        nullptr,
        &m_pDepthStencil
        );

    CD3D11_DEPTH_STENCIL_VIEW_DESC depthStencilViewDesc(D3D11_DSV_DIMENSION_TEXTURE2D);

    m_pd3dDevice->CreateDepthStencilView(
        m_pDepthStencil.Get(),
        &depthStencilViewDesc,
        &m_pDepthStencilView
        );


    ZeroMemory(&m_viewport, sizeof(D3D11_VIEWPORT));
    m_viewport.Height = (float) m_bbDesc.Height;
    m_viewport.Width = (float) m_bbDesc.Width;
    m_viewport.MinDepth = 0;
    m_viewport.MaxDepth = 1;

    m_pd3dDeviceContext->RSSetViewports(
        1,
        &m_viewport
        );

    return hr;
}

HRESULT DeviceResources::ReleaseBackBuffer()
{
    HRESULT hr = S_OK;

    // Release the render target view based on the back buffer:
    m_pRenderTarget.Reset();

    // Release the back buffer itself:
    m_pBackBuffer.Reset();

    // The depth stencil will need to be resized, so release it (and view):
    m_pDepthStencilView.Reset();
    m_pDepthStencil.Reset();

    // After releasing references to these resources, we need to call Flush() to
    // ensure that Direct3D also releases any references it might still have to
    // the same resources - such as pipeline bindings.
    m_pd3dDeviceContext->Flush();

    return hr;
}

HRESULT DeviceResources::GoFullScreen()
{
    HRESULT hr = S_OK;

    hr = m_pDXGISwapChain->SetFullscreenState(TRUE, NULL);

    // Swap chains created with the DXGI_SWAP_EFFECT_FLIP_SEQUENTIAL flag need to
    // call ResizeBuffers in order to realize a full-screen mode switch. Otherwise,
    // your next call to Present will fail.
```

```
    // Before calling ResizeBuffers, you have to release all references to the back
    // buffer device resource.
    ReleaseBackBuffer();

    // Now we can call ResizeBuffers.
    hr = m_pDXGISwapChain->ResizeBuffers(
        0,                  // Number of buffers. Set this to 0 to preserve the existing setting.
        0, 0,               // Width and height of the swap chain. Set to 0 to match the screen resolution.
        DXGI_FORMAT_UNKNOWN, // This tells DXGI to retain the current back buffer format.
        0
        );

    // Then we can recreate the back buffer, depth buffer, and so on.
    hr = ConfigureBackBuffer();

    return hr;
}

HRESULT DeviceResources::GoWindowed()
{
    HRESULT hr = S_OK;

    hr = m_pDXGISwapChain->SetFullscreenState(FALSE, NULL);

    // Swap chains created with the DXGI_SWAP_EFFECT_FLIP_SEQUENTIAL flag need to
    // call ResizeBuffers in order to realize a change to windowed mode. Otherwise,
    // your next call to Present will fail.

    // Before calling ResizeBuffers, you have to release all references to the back
    // buffer device resource.
    ReleaseBackBuffer();

    // Now we can call ResizeBuffers.
    hr = m_pDXGISwapChain->ResizeBuffers(
        0,                  // Number of buffers. Set this to 0 to preserve the existing setting.
        0, 0,               // Width and height of the swap chain. MUST be set to a non-zero value. For
example, match the window size.
        DXGI_FORMAT_UNKNOWN, // This tells DXGI to retain the current back buffer format.
        0
        );

    // Then we can recreate the back buffer, depth buffer, and so on.
    hr = ConfigureBackBuffer();

    return hr;
}

//--------------------------------------------------------------------------------
// Returns the aspect ratio of the back buffer.
//--------------------------------------------------------------------------------
float DeviceResources::GetAspectRatio()
{
    return static_cast<float>(m_bbDesc.Width) / static_cast<float>(m_bbDesc.Height);
}

//--------------------------------------------------------------------------------
// Present frame:
// Show the frame on the primary surface.
//--------------------------------------------------------------------------------
void DeviceResources::Present()
{
    m_pDXGISwapChain->Present(1, 0);
}


//--------------------------------------------------------------------------------
// Destructor.
//--------------------------------------------------------------------------------
DeviceResources::~DeviceResources()
```

```
{

}
```

Renderer.h

```
//// THIS CODE AND INFORMATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF
//// ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO
//// THE IMPLIED WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A
//// PARTICULAR PURPOSE.
////
//// Copyright (c) Microsoft Corporation. All rights reserved
#pragma once


//--------------------------------------------------------------------------------
// Includes
//--------------------------------------------------------------------------------
#include "DeviceResources.h"

//--------------------------------------------------------------------------------
// Class declarations
//--------------------------------------------------------------------------------

class Renderer
{
public:
    Renderer(std::shared_ptr<DeviceResources> deviceResources);
    ~Renderer();

    void CreateDeviceDependentResources();
    void CreateWindowSizeDependentResources();
    void Update();
    void Render();

private:
    HRESULT CreateShaders();
    HRESULT CreateCube();
    void    CreateViewAndPerspective();

    //--------------------------------------------------------------------------------
    // Pointer to device resource manager
    //--------------------------------------------------------------------------------
    std::shared_ptr<DeviceResources> m_deviceResources;

    //--------------------------------------------------------------------------------
    // Variables for rendering the cube
    //--------------------------------------------------------------------------------
    typedef struct _constantBufferStruct {
        DirectX::XMFLOAT4X4 world;
        DirectX::XMFLOAT4X4 view;
        DirectX::XMFLOAT4X4 projection;
    } ConstantBufferStruct;

    // Assert that the constant buffer remains 16-byte aligned.
    static_assert((sizeof(ConstantBufferStruct) % 16) == 0, "Constant Buffer size must be 16-byte aligned");

    //--------------------------------------------------------------------------------
    // Per-vertex data
    //--------------------------------------------------------------------------------
    typedef struct _vertexPositionColor
    {
        DirectX::XMFLOAT3 pos;
        DirectX::XMFLOAT3 color;
    } VertexPositionColor;
```

```
    //--------------------------------------------------------------------------------
    // Per-vertex data (extended)
    //--------------------------------------------------------------------------------
    typedef struct _vertexPositionColorTangent
    {
        DirectX::XMFLOAT3 pos;
        DirectX::XMFLOAT3 normal;
        DirectX::XMFLOAT3 tangent;
    } VertexPositionColorTangent;

    ConstantBufferStruct m_constantBufferData;
    unsigned int  m_indexCount;
    unsigned int  m_frameCount;

    //--------------------------------------------------------------------------------
    // Direct3D device resources
    //--------------------------------------------------------------------------------
    //ID3DXEffect* m_pEffect;
    Microsoft::WRL::ComPtr<ID3D11Buffer>            m_pVertexBuffer;
    Microsoft::WRL::ComPtr<ID3D11Buffer>            m_pIndexBuffer;
    Microsoft::WRL::ComPtr<ID3D11VertexShader>      m_pVertexShader;
    Microsoft::WRL::ComPtr<ID3D11InputLayout>       m_pInputLayout;
    Microsoft::WRL::ComPtr<ID3D11InputLayout>       m_pInputLayoutExtended;
    Microsoft::WRL::ComPtr<ID3D11PixelShader>       m_pPixelShader;
    Microsoft::WRL::ComPtr<ID3D11Buffer>            m_pConstantBuffer;
};
```

Renderer.cpp

```
//// THIS CODE AND INFORMATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF
//// ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO
//// THE IMPLIED WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A
//// PARTICULAR PURPOSE.
////
//// Copyright (c) Microsoft Corporation. All rights reserved


//--------------------------------------------------------------------------------
// File: Cube.cpp
//
// Renders a spinning, colorful cube.
//
//--------------------------------------------------------------------------------


//--------------------------------------------------------------------------------
// Includes
//--------------------------------------------------------------------------------
#include "dxstdafx.h"
#include "resource.h"

#include <string>
#include <memory>
#include <ppltasks.h>

#include "Renderer.h"


//--------------------------------------------------------------------------------
// Constructor
//--------------------------------------------------------------------------------
Renderer::Renderer(std::shared_ptr<DeviceResources> deviceResources)
    :
    m_frameCount(0),
    m_deviceResources(deviceResources)
{
```

```
    m_frameCount = 0; // init frame count
}

//--------------------------------------------------------------------------
// Create Direct3D shader resources by loading the .cso files.
//--------------------------------------------------------------------------
HRESULT Renderer::CreateShaders()
{
    HRESULT hr = S_OK;

    // Use the Direct3D device to load resources into graphics memory.
    ID3D11Device* device = m_deviceResources->GetDevice();

    // You'll need to use a file loader to load the shader bytecode. In this
    // example, we just use the standard library.
    FILE* vShader, * pShader;
    BYTE* bytes;

    size_t destSize = 4096;
    size_t bytesRead = 0;
    bytes = new BYTE[destSize];

    fopen_s(&vShader, "CubeVertexShader.cso", "rb");
    bytesRead = fread_s(bytes, destSize, 1, 4096, vShader);
    hr = device->CreateVertexShader(
        bytes,
        bytesRead,
        nullptr,
        &m_pVertexShader
        );

    D3D11_INPUT_ELEMENT_DESC iaDesc [] =
    {
        { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT,
        0, 0, D3D11_INPUT_PER_VERTEX_DATA, 0 },

        { "COLOR", 0, DXGI_FORMAT_R32G32B32_FLOAT,
        0, 12, D3D11_INPUT_PER_VERTEX_DATA, 0 },
    };

    hr = device->CreateInputLayout(
        iaDesc,
        ARRAYSIZE(iaDesc),
        bytes,
        bytesRead,
        &m_pInputLayout
        );

    delete bytes;


    bytes = new BYTE[destSize];
    bytesRead = 0;
    fopen_s(&pShader, "CubePixelShader.cso", "rb");
    bytesRead = fread_s(bytes, destSize, 1, 4096, pShader);
    hr = device->CreatePixelShader(
        bytes,
        bytesRead,
        nullptr,
        m_pPixelShader.GetAddressOf()
        );

    delete bytes;

    CD3D11_BUFFER_DESC cbDesc(
        sizeof(ConstantBufferStruct),
        D3D11_BIND_CONSTANT_BUFFER
        );
```

```
hr = device->CreateBuffer(
    &cbDesc,
    nullptr,
    m_pConstantBuffer.GetAddressOf()
    );

fclose(vShader);
fclose(pShader);


// Load the extended shaders with lighting calculations.
/*
bytes = new BYTE[destSize];
bytesRead = 0;
fopen_s(&vShader, "CubeVertexShaderLighting.cso", "rb");
bytesRead = fread_s(bytes, destSize, 1, 4096, vShader);
hr = device->CreateVertexShader(
    bytes,
    bytesRead,
    nullptr,
    &m_pVertexShader
    );

D3D11_INPUT_ELEMENT_DESC iaDescExtended[] =
{
    { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT,
    0, 0, D3D11_INPUT_PER_VERTEX_DATA, 0 },

    { "NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT,
    0, 12, D3D11_INPUT_PER_VERTEX_DATA, 0 },

    { "TANGENT", 0, DXGI_FORMAT_R32G32B32_FLOAT,
    0, 12, D3D11_INPUT_PER_VERTEX_DATA, 0 },
};

hr = device->CreateInputLayout(
    iaDesc,
    ARRAYSIZE(iaDesc),
    bytes,
    bytesRead,
    &m_pInputLayoutExtended
    );

delete bytes;


bytes = new BYTE[destSize];
bytesRead = 0;
fopen_s(&pShader, "CubePixelShaderPhongLighting.cso", "rb");
bytesRead = fread_s(bytes, destSize, 1, 4096, pShader);
hr = device->CreatePixelShader(
    bytes,
    bytesRead,
    nullptr,
    m_pPixelShader.GetAddressOf()
    );

delete bytes;

fclose(vShader);
fclose(pShader);


bytes = new BYTE[destSize];
bytesRead = 0;
fopen_s(&pShader, "CubePixelShaderTexelLighting.cso", "rb");
bytesRead = fread_s(bytes, destSize, 1, 4096, pShader);
hr = device->CreatePixelShader(
bytes
```

```
    bytes,
    bytesRead,
    nullptr,
    m_pPixelShader.GetAddressOf()
    );

    delete bytes;

    fclose(pShader);
    */

    return hr;
}

//--------------------------------------------------------------------------------
// Create the cube:
// Creates the vertex buffer and index buffer.
//--------------------------------------------------------------------------------
HRESULT Renderer::CreateCube()
{
    HRESULT hr = S_OK;

    // Use the Direct3D device to load resources into graphics memory.
    ID3D11Device* device = m_deviceResources->GetDevice();

    // Create cube geometry.
    VertexPositionColor CubeVertices[] =
    {
        {DirectX::XMFLOAT3(-0.5f,-0.5f,-0.5f), DirectX::XMFLOAT3(  0,   0,   0),},
        {DirectX::XMFLOAT3(-0.5f,-0.5f, 0.5f), DirectX::XMFLOAT3(  0,   0,   1),},
        {DirectX::XMFLOAT3(-0.5f, 0.5f,-0.5f), DirectX::XMFLOAT3(  0,   1,   0),},
        {DirectX::XMFLOAT3(-0.5f, 0.5f, 0.5f), DirectX::XMFLOAT3(  0,   1,   1),},

        {DirectX::XMFLOAT3( 0.5f,-0.5f,-0.5f), DirectX::XMFLOAT3(  1,   0,   0),},
        {DirectX::XMFLOAT3( 0.5f,-0.5f, 0.5f), DirectX::XMFLOAT3(  1,   0,   1),},
        {DirectX::XMFLOAT3( 0.5f, 0.5f,-0.5f), DirectX::XMFLOAT3(  1,   1,   0),},
        {DirectX::XMFLOAT3( 0.5f, 0.5f, 0.5f), DirectX::XMFLOAT3(  1,   1,   1),},
    };

    // Create vertex buffer:

    CD3D11_BUFFER_DESC vDesc(
        sizeof(CubeVertices),
        D3D11_BIND_VERTEX_BUFFER
        );

    D3D11_SUBRESOURCE_DATA vData;
    ZeroMemory(&vData, sizeof(D3D11_SUBRESOURCE_DATA));
    vData.pSysMem = CubeVertices;
    vData.SysMemPitch = 0;
    vData.SysMemSlicePitch = 0;

    hr = device->CreateBuffer(
        &vDesc,
        &vData,
        &m_pVertexBuffer
        );

    // Create index buffer:
    unsigned short CubeIndices [] =
    {
        0,2,1, // -x
        1,2,3,

        4,5,6, // +x
        5,7,6,

        0,1,5, // -y
        0,5,4,
```

```
        2,6,7, // +y
        2,7,3,

        0,4,6, // -z
        0,6,2,

        1,3,7, // +z
        1,7,5,
    };

    m_indexCount = ARRAYSIZE(CubeIndices);

    CD3D11_BUFFER_DESC iDesc(
        sizeof(CubeIndices),
        D3D11_BIND_INDEX_BUFFER
        );

    D3D11_SUBRESOURCE_DATA iData;
    ZeroMemory(&iData, sizeof(D3D11_SUBRESOURCE_DATA));
    iData.pSysMem = CubeIndices;
    iData.SysMemPitch = 0;
    iData.SysMemSlicePitch = 0;

    hr = device->CreateBuffer(
        &iDesc,
        &iData,
        &m_pIndexBuffer
        );

    return hr;
}

//--------------------------------------------------------------------------------
// Create the view matrix and create the perspective matrix.
//--------------------------------------------------------------------------------
void Renderer::CreateViewAndPerspective()
{
    // Use DirectXMath to create view and perspective matrices.

    DirectX::XMVECTOR eye = DirectX::XMVectorSet(0.0f, 0.7f, 1.5f, 0.f);
    DirectX::XMVECTOR at  = DirectX::XMVectorSet(0.0f,-0.1f, 0.0f, 0.f);
    DirectX::XMVECTOR up  = DirectX::XMVectorSet(0.0f, 1.0f, 0.0f, 0.f);

    DirectX::XMStoreFloat4x4(
        &m_constantBufferData.view,
        DirectX::XMMatrixTranspose(
            DirectX::XMMatrixLookAtRH(
                eye,
                at,
                up
                )
            )
        );

    float aspectRatioX = m_deviceResources->GetAspectRatio();
    float aspectRatioY = aspectRatioX < (16.0f / 9.0f) ? aspectRatioX / (16.0f / 9.0f) : 1.0f;

    DirectX::XMStoreFloat4x4(
        &m_constantBufferData.projection,
        DirectX::XMMatrixTranspose(
            DirectX::XMMatrixPerspectiveFovRH(
                2.0f * std::atan(std::tan(DirectX::XMConvertToRadians(70) * 0.5f) / aspectRatioY),
                aspectRatioX,
                0.01f,
                100.0f
                )
            )
        );
```

```
}

//--------------------------------------------------------------------------------
// Create device-dependent resources for rendering.
//--------------------------------------------------------------------------------
void Renderer::CreateDeviceDependentResources()
{
    // Compile shaders using the Effects library.
    auto CreateShadersTask = Concurrency::create_task(
            [this]( )
            {
                CreateShaders();
            }
        );

    // Load the geometry for the spinning cube.
    auto CreateCubeTask = CreateShadersTask.then(
            [this]()
            {
                CreateCube();
            }
        );
}

void Renderer::CreateWindowSizeDependentResources()
{
    // Create the view matrix and the perspective matrix.
    CreateViewAndPerspective();
}


//--------------------------------------------------------------------------------
// Update the scene.
//--------------------------------------------------------------------------------
void Renderer::Update()
{
    // Rotate the cube 1 degree per frame.
    DirectX::XMStoreFloat4x4(
        &m_constantBufferData.world,
        DirectX::XMMatrixTranspose(
            DirectX::XMMatrixRotationY(
                DirectX::XMConvertToRadians(
                    (float) m_frameCount++
                    )
                )
            )
        );

    if (m_frameCount == MAXUINT)  m_frameCount = 0;
}

//--------------------------------------------------------------------------------
// Render the cube.
//--------------------------------------------------------------------------------
void Renderer::Render()
{
    // Use the Direct3D device context to draw.
    ID3D11DeviceContext* context = m_deviceResources->GetDeviceContext();

    ID3D11RenderTargetView* renderTarget = m_deviceResources->GetRenderTarget();
    ID3D11DepthStencilView* depthStencil = m_deviceResources->GetDepthStencil();

    context->UpdateSubresource(
        m_pConstantBuffer.Get(),
        0,
        nullptr,
        &m_constantBufferData,
        0,
        0
```

```
    );

// Clear the render target and the z-buffer.
const float teal [] = { 0.098f, 0.439f, 0.439f, 1.000f };
context->ClearRenderTargetView(
    renderTarget,
    teal
    );
context->ClearDepthStencilView(
    depthStencil,
    D3D11_CLEAR_DEPTH | D3D11_CLEAR_STENCIL,
    1.0f,
    0);

// Set the render target.
context->OMSetRenderTargets(
    1,
    &renderTarget,
    depthStencil
    );

// Set up the IA stage by setting the input topology and layout.
UINT stride = sizeof(VertexPositionColor);
UINT offset = 0;

context->IASetVertexBuffers(
    0,
    1,
    m_pVertexBuffer.GetAddressOf(),
    &stride,
    &offset
    );

context->IASetIndexBuffer(
    m_pIndexBuffer.Get(),
    DXGI_FORMAT_R16_UINT,
    0
    );

context->IASetPrimitiveTopology(
    D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST
    );

context->IASetInputLayout(m_pInputLayout.Get());

// Set up the vertex shader stage.
context->VSSetShader(
    m_pVertexShader.Get(),
    nullptr,
    0
    );

context->VSSetConstantBuffers(
    0,
    1,
    m_pConstantBuffer.GetAddressOf()
    );

// Set up the pixel shader stage.
context->PSSetShader(
    m_pPixelShader.Get(),
    nullptr,
    0
    );

// Calling Draw tells Direct3D to start sending commands to the graphics device.
context->DrawIndexed(
    m_indexCount,
    0,
```

```
        0
        );
}

//--------------------------------------------------------------------------
// Clean up cube resources when the Direct3D device is lost or destroyed.
//--------------------------------------------------------------------------
Renderer::~Renderer()
{
    // ComPtr will clean up references for us. But be careful to release
    // references to anything you don't need whenever you call Flush or Trim.
    // As always, clean up your system (CPU) memory resources before exit.
}
```

Roadmap for Desktop DirectX apps

12/13/2021 • 6 minutes to read • Edit Online

Here are key resources to help you get started with using DirectX and C++ to develop graphics-intensive Desktop apps, such as games. This is not a comprehensive list of all of the features or available resources.

Get started

Here are some key topics. Setting up your DirectX project, acclimating yourself to Windows, and sample applications.

TOPIC	DESCRIPTION
Create your first Windows app using DirectX	Use this basic tutorial to get started with DirectX app development, then use the roadmap to continue exploring DirectX.
Get started with DirectX for Windows	Review the steps you must take to begin developing a game using DirectX and C++.
Complete code for a DirectX framework	Get the code for a basic DirectX rendering framework.
How to Use Direct3D 11	This section demonstrates how to use the Microsoft Direct3D 11 API to accomplish several common tasks.
Programming Guide for Direct3D 11	The programming guide contains information about how to use the Microsoft Direct3D 11 programmable pipeline to create realtime 3D graphics for desktop applications.
Tools for DirectX Graphics	Documentation for tools used to support DirectX development.
What's new in Direct3D 11	A breakdown of all the features added in the most recent versions of DirectX and Direct3D (currently 11.2).
Download Visual Studio 2013	You must have Visual Studio Express 2013 for Windows Desktop to create Windows Store games. For a tour of Visual Studio, see Develop Windows Store apps using Visual Studio 2012. For info about new features in Visual Studio, see Product Highlights for Visual Studio 2013.
Where is the DirectX SDK?	Contains guidance for devs who want to bring their DirectX projects into Microsoft Visual Studio.

Sample applications

TOPIC	DESCRIPTION
Direct3D Tutorial Win32 sample	Basic desktop Direct3D tutorial sample.

TOPIC	DESCRIPTION
DirectX video rendering sample	A sample that demonstrates custom video rendering with Direct3D.

Review key Direct3D 11 concepts

TOPIC	DESCRIPTION
Graphics Pipeline	Covers the basic Direct3D 11 graphics pipeline.
Rendering	Covers the Direct3D rendering models, components, shaders and API call flow.
Resources	Covers Direct3D "resources", such as buffers and other GPU resource types.
Effects	Covers Direct3D multi-shader instancing and effects.
How To: Create a Swap Chain	How to create the swap chain used to draw pixels to a region of the screen.
How To: Create a Device and Immediate Context	How to create a Direct3D device abstraction and an immediate context for drawing.
How to: Create a Vertex Buffer	How to create a simple list of mesh vertices for processing by the GPU.
How to: Create an Index Buffer	How to create an index buffer enabling the vertex shader to walk the order of vertices in a mesh.
How to: Create a Constant Buffer	How to pass constant (uniform) data between the CPU and the GPU during rendering.
How to: Create a Texture	How to create a texture or other buffer resource that can be sampled by the GPU.
How to: Initialize a Texture From a File	How to load a texture from a file and process it for use by the shader pipeline.
How To: Compile a Shader	How to compile a shader for use in your graphics application.

Graphics APIs

TOPIC	DESCRIPTION
Direct3D 11	Documentation of the core APIs for the virtualization of the GPU and its resources, and for drawing graphics using a unified shader model.

TOPIC	DESCRIPTION
Direct3D HLSL	Reference documentation for High-Level Shader Language, the syntax and rules used to define shader programs executed as part of the graphics pipeline in a unified shader model.
DirectX Graphics Interface (DXGI)	Documentation of the low-level APIs used to acquire the GPU interface and system resources.
Direct2D	Documentation for the Direct2D APIs, which support the drawing of 2D primitives. Typically, Direct2D is used for custom user interfaces, image processing and batching, and simple games.
DirectWrite	Documentation for the DirectWrite APIs, which support custom font rendering and scaling.
Windows Imaging Component (WIC)	Documentation for the WIC APIs, which are used for reading and managing different bitmap image formats.
DirectDraw Surfaces (DDS) for textures	Documentation for the DDS APIs, which are used for 2D texture compression and decompression in conjunction with the WIC APIs.
DirectXMath	Documentation for the DirectXMath APIs, which support Direct3D with a set of types and functions suited for 3D real-time graphics development. (Formerly XNAMath.)
DirectCompute	Documentation for the DirectCompute APIs, used for compute or general-use shader functionality.

Audio, media, and input APIs

TOPIC	DESCRIPTION
XAudio2 Programming Guide	Top-level node for the XAudio2 audio API conceptual documentation.
XAudio2 Programming Reference	Top-level node for the XAudio2 audio API reference documentation.
XInput Programming Guide	Top-level node for the XInput controller API conceptual documentation.
XInput Programming Reference	Top-level node for the XInput controller API reference documentation.
Media Foundation	Top-level node for the Media Foundation (MF) media (audio/video) playback API documentation. Typically, MF is used in games for soundtrack playback, whereas XAudio2 is used for dynamic audio.

Port to DirectX 11

TOPIC	DESCRIPTION
Migrating to Direct3D 11	Basic guidance for moving your DirectX 9 codebase to DirectX 11.
Dual-use Coding Techniques for Games	Detailed blog post on developing for both DirectX 9_* and DirectX 11_* feature levels in a single application.
Implementing shadow buffers for Direct3D feature level 9	Guidance for implementing shadow maps under DirectX feature level 9_*.

Work with C++

If you're an old hand with C++ on Windows platforms, things may look a little different. Here are some pointers to topics that can help you get a handle on the difference.

> **NOTE**
> Some of these topics exist to help you maintain your C++/CX application. But we recommend that you use C++/WinRT for new applications. C++/WinRT is an entirely standard modern C++17 language projection for Windows Runtime (WinRT) APIs, implemented as a header-file-based library, and designed to provide you with first-class access to the modern Windows API.

TOPIC	DESCRIPTION
Visual C++ language reference (C++/CX)	High-level page that links to content related to C++.
Quick Reference (Windows Runtime and Visual C++)	Table that provides quick info about Visual C++ component extensions (C++/CX) operators and keywords.
Type system (C++/CX)	Reference content for the types that are supported by C++/CX.
Namespaces (C++/CX)	Reference content for the namespaces that contain C++-specific types that can be used in Windows Store apps.

TOPIC	DESCRIPTION
Asynchronous programming (DirectX and C++)	Learn about asynchronous and multithreaded programming for DirectX apps and games.
Asynchronous programming in C++	Describes the basic ways to use the task class to consume Windows Runtime asynchronous methods.
task Class (Concurrency Runtime)	Reference documentation for the task class.
Task Parallelism (Concurrency Runtime)	In-depth discussion about the task class and how to use it.

Additional useful libraries for Windows C++ programming

TOPIC	DESCRIPTION
C++ Standard Template Library	Windows Runtime types play well with Standard Template Library types. Most C++ Windows Store apps use Standard Template Library collections and algorithms, except at the ABI boundary.
Parallel Patterns Library	PPL provides algorithms and types that simplify task parallelism and data parallelism on the CPU.
C++ Accelerated Massive Parallelism (C++ AMP)	C++ AMP provides access to the GPU for general-purpose data parallelism on video cards that support DirectX 11.

Blogs and other resources

TOPIC	DESCRIPTION
Games for Windows and DirectX blog	Regularly-updated blog from a key DirectX dev contributor, and an all-around great resource for DirectX devs.
Windows DirectX developer blog	The official Windows DirectX blog.
Shawn Hargreave's DirectX blog)	Dev blog from another well-respected Windows DirectX dev contributor.
DirectX Tool Kit	CodePlex site for the DirectX Tool Kit (DxTK), which contains a number of helpful support APIs for loading meshes, playing sounds, and other common behaviors.
DirectXTex texture processing library	Code plex site for the DirectX Texture Processing Library (DxTEX), which contains support APIs for texture processing and compression/decompression.