

DirectX12

How to get started with 3D in Direct3D

Hooman Salamat

Direct3D 12

Direct3D 12 provides an API and platform that allows apps to take advantage of the graphics and computing capabilities of PCs equipped with one or more Direct3D 12-compatible GPUs.

DirectX 12 introduces the next version of Direct3D, the 3D graphics API at the heart of DirectX. This version of Direct3D is faster and more efficient than any previous version. Direct3D 12 enables richer scenes, more objects, more complex effects, and full utilization of modern GPU hardware.

To write 3D games and apps for Windows 10 and Windows 10 Mobile, you must understand the basics of the Direct3D 12 technology, and how to prepare to use it in your games and apps.



Overview of the Windows Graphics Architecture

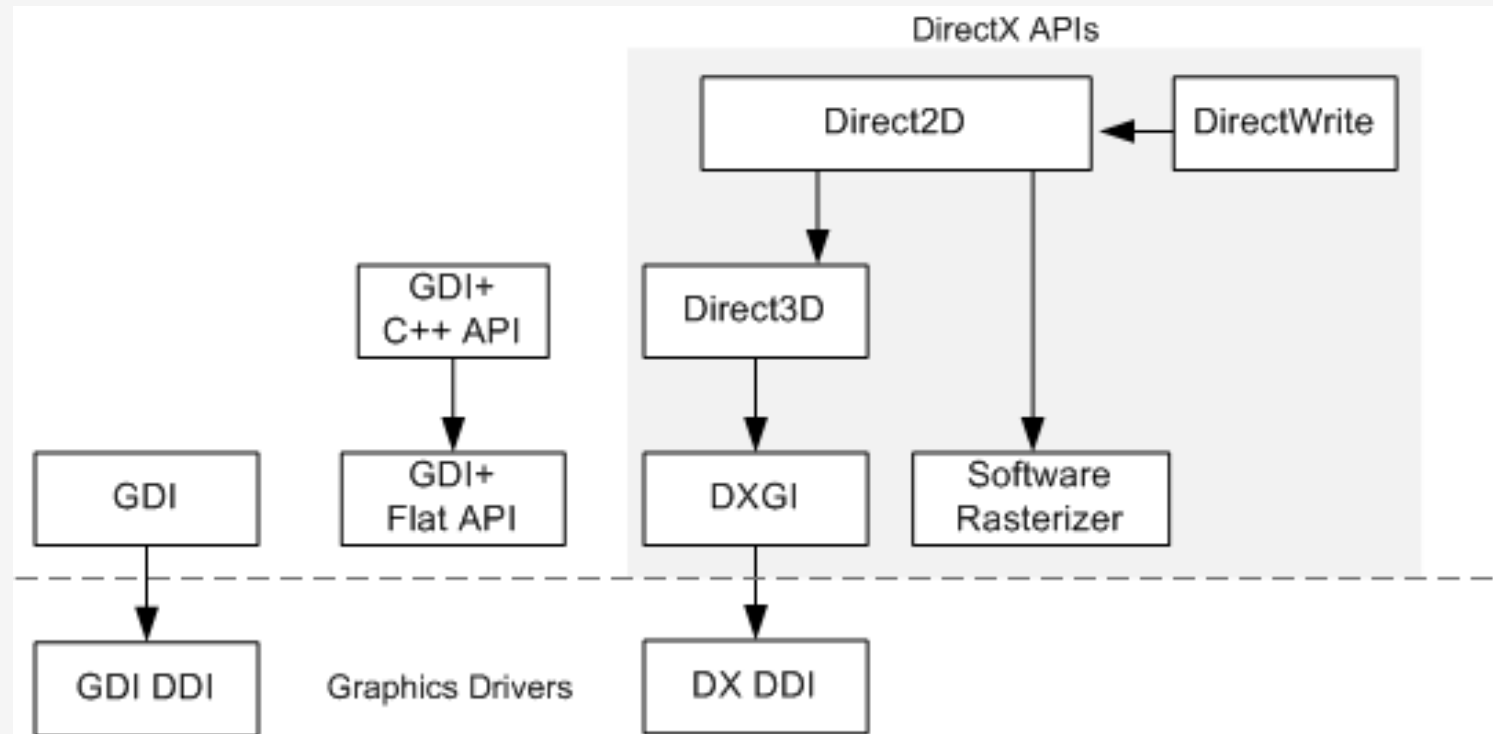
<https://docs.microsoft.com/en-us/windows/win32/learnwin32/overview-of-the-windows-graphics-architecture>

DXGI is factoring out some functionalities

from D3D into its own subsystem

that doesn't change as quickly:

1. Enumeration of hardware devices
2. Presenting rendered frames to an output
3. Controlling Gamma



DirectX APIs

Graphics Device Interface (GDI) is the original graphics interface for Windows. GDI was first written for 16-bit Windows and then updated for 32-bit and 64-bit Windows.

GDI+ was introduced in Windows XP as a successor to GDI. The GDI+ library is accessed through a set of C++ classes that wrap flat C functions. The .NET Framework also provides a managed version of GDI+ in the **System.Drawing** namespace.

Direct3D supports 3-D graphics.

Direct2D is a modern API for 2-D graphics, the successor to both GDI and GDI+.

DirectWrite is a text layout and rasterization engine. You can use either GDI or Direct2D to draw the rasterized text.

DirectX Graphics Infrastructure (DXGI) performs low-level tasks, such as presenting frames for output.

Most applications do not use DXGI directly. Rather, it serves as an intermediate layer between the graphics driver and Direct3D.

Though most graphics programming is done using Direct3D, you can use DXGI to present frames to a window, monitor, or other graphics component for eventual composition and display.

DirectX Graphics Infrastructure

Microsoft DirectX Graphics Infrastructure (DXGI) handles enumerating graphics adapters, enumerating display modes, selecting buffer formats, sharing resources between processes (such as, between applications and the Desktop Window Manager (DWM)), and presenting rendered frames to a window or monitor for display.

DXGI is used by Direct3D 10, Direct3D 11 and Direct3D 12.

Though most graphics programming is done using Direct3D, you can use DXGI to present frames to a window, monitor, or other graphics component for eventual composition and display. You can also use DXGI to read the contents on a monitor.

For example, a 2D rendering API would need swap chains and page flipping for smooth animation just as much as a 3D rendering API; thus the swap chain interface `IDXGISwapChain` is actually part of the DXGI API

Display adapters implement graphical functionality. Usually, the *display adapter* is a physical piece of hardware (e.g., graphics card); however, a system can also have a software display adapter that emulates hardware graphics functionality. A system can have several adapters (e.g., if it has several graphics cards). An adapter is represented by the `IDXGIAdapter` interface.

DXGI handles other common graphical functionality like full-screen mode transitions, enumerating graphical system information like display adapters, monitors, and supported display modes (resolution, refresh rate, and such); it also defines the various supported surface formats (`DXGI_FORMAT`).

One of the key DXGI interfaces is the `IDXGIFactory` interface, which is primarily used to create the `IDXGISwapChain` interface and enumerate display adapters.

Code flow for a simple app

The outermost loop of a D3D 12 program follows a very standard graphics process:

[Initialize](#)

Repeat

[Update](#)

[Render](#)

[Destroy](#)



Initialize

Initialization involves first setting up the global variables and classes, and an initialize function must prepare the pipeline and assets.

Initialize the pipeline.

Enable the debug layer.

Create the device or create a **Windows Advanced Rasterization Platform (WARP)**! For warp device, you will need to use DXGI factory to access WARP adapter. You will need this if you have an old graphics card.

[WARP: Enabling Rendering When Direct3D Hardware is Not Available](#)

Create the command queue.

Create the swap chain.

Create a render target view (RTV) descriptor heap. A [descriptor heap](#) can be thought of as an array of [descriptors](#). Where each descriptor fully describes an object to the GPU. A descriptor heap defines the views and how to access resources (for example, a render target view).

Create frame resources (a render target view for each frame).

Create a command allocator. A command allocator manages the underlying storage for [command lists and bundles](#).

Direct3D 12

Direct3D 12 introduces the concept of feature, which roughly correspond to various Direct3D versions from version 9 to 12:

```
enum D3D_FEATURE_LEVEL
{
    D3D_FEATURE_LEVEL_9_1 = 0x9100,
    D3D_FEATURE_LEVEL_9_2 = 0x9200,
    D3D_FEATURE_LEVEL_9_3 = 0x9300,
    D3D_FEATURE_LEVEL_10_0 = 0xa000,
    D3D_FEATURE_LEVEL_10_1 = 0xa100,
    D3D_FEATURE_LEVEL_11_0 = 0xb000,
    D3D_FEATURE_LEVEL_11_1 = 0xb100,
    D3D_FEATURE_LEVEL_12_0 = 0xc000,
    D3D_FEATURE_LEVEL_12_1 = 0xc100
}
```

}D3D_FEATURE_LEVEL;

Direct3D 12 requires graphics hardware conforming to feature levels 11_0 and 11_1 which support virtual memory address translations. There are two new feature levels, 12_0 and 12_1, which include some features that are optional on levels 11_0 and 11_1.

https://en.wikipedia.org/wiki/Feature_levels_in_Direct3D

When it creates a device, it creates the highest feature available in the system!

```
ThrowIfFailed(CreateDXGIFactory1(IID_PPV_ARGS(&mdxgiFactory)));
// Try to create hardware device.
HRESULT hardwareResult = D3D12CreateDevice(
    nullptr,           // default adapter
    D3D_FEATURE_LEVEL_12_0,
    IID_PPV_ARGS(&md3dDevice));

// Fallback to WARP device.
if(FAILED(hardwareResult))
{
    ComPtr<IDXGIAdapter> pWarpAdapter;
    ThrowIfFailed(mdxgiFactory->EnumWarpAdapter(IID_PPV_ARGS(&pWarpAdapter)));

    ThrowIfFailed(D3D12CreateDevice(
        pWarpAdapter.Get(),
        D3D_FEATURE_LEVEL_12_0,
        IID_PPV_ARGS(&md3dDevice)));
}
```


Initialize the assets

Create an empty root signature: A graphics [root signature](#) defines what resources are bound to the graphics pipeline.

Compile the shaders.

Create the vertex input layout.

Create a [pipeline state object](#) description, then create the object. A pipeline state object maintains the state of all currently set shaders as well as certain fixed function state objects (such as the *input assembler*, *tesselator*, *rasterizer* and *output merger*). In Direct3D 12, required pipeline state is attached to a command list via a [pipeline state object](#) (PSO).

Create the command list. With the command list allocator and PSO, you can create the actual command list, which will be executed at a later time.

Close the command list.

Create and load the vertex buffers.

Create the vertex buffer views.

Create a fence: A fence is used to [synchronize the CPU and GPU](#).

Create an event handle.

Wait for the GPU to finish.

Update

Update everything that should change since the last frame.

Modify the constant, vertex, index buffers, and everything else, as necessary.



Render: Draw the new world

Populate the command list.

- Reset the command list allocator: Re-use the memory that is associated with the command allocator.
- Reset the command list.
- Set the graphics root signature
- Sets the graphics root signature to use for the current command list.
- Set the viewport and scissor rectangles
- Set a resource barrier, indicating the back buffer is to be used as a render target. Resource barriers are used to manage resource transitions.
- Record commands into the command list.
- Indicate the back buffer will be used to present after the command list has executed. Another call to set a resource barrier.
- Close the command list to further recording

Execute the command list.

Present the frame.

Wait for the GPU to finish. Keep updating and checking the fence.

Destroy

Cleanly close down the app.

Wait for the GPU to finish

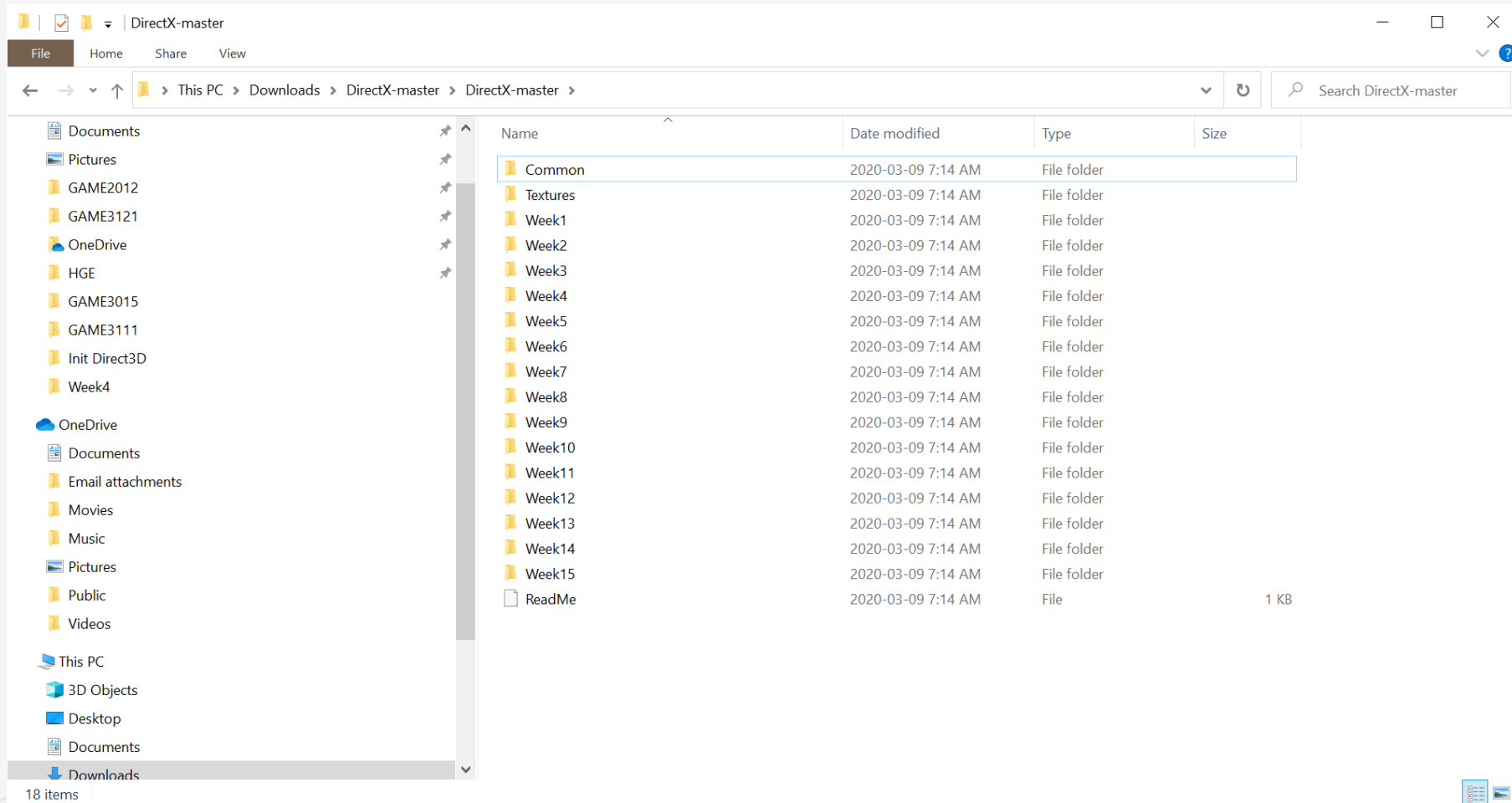
Final check on the fence.

Close the event handle.

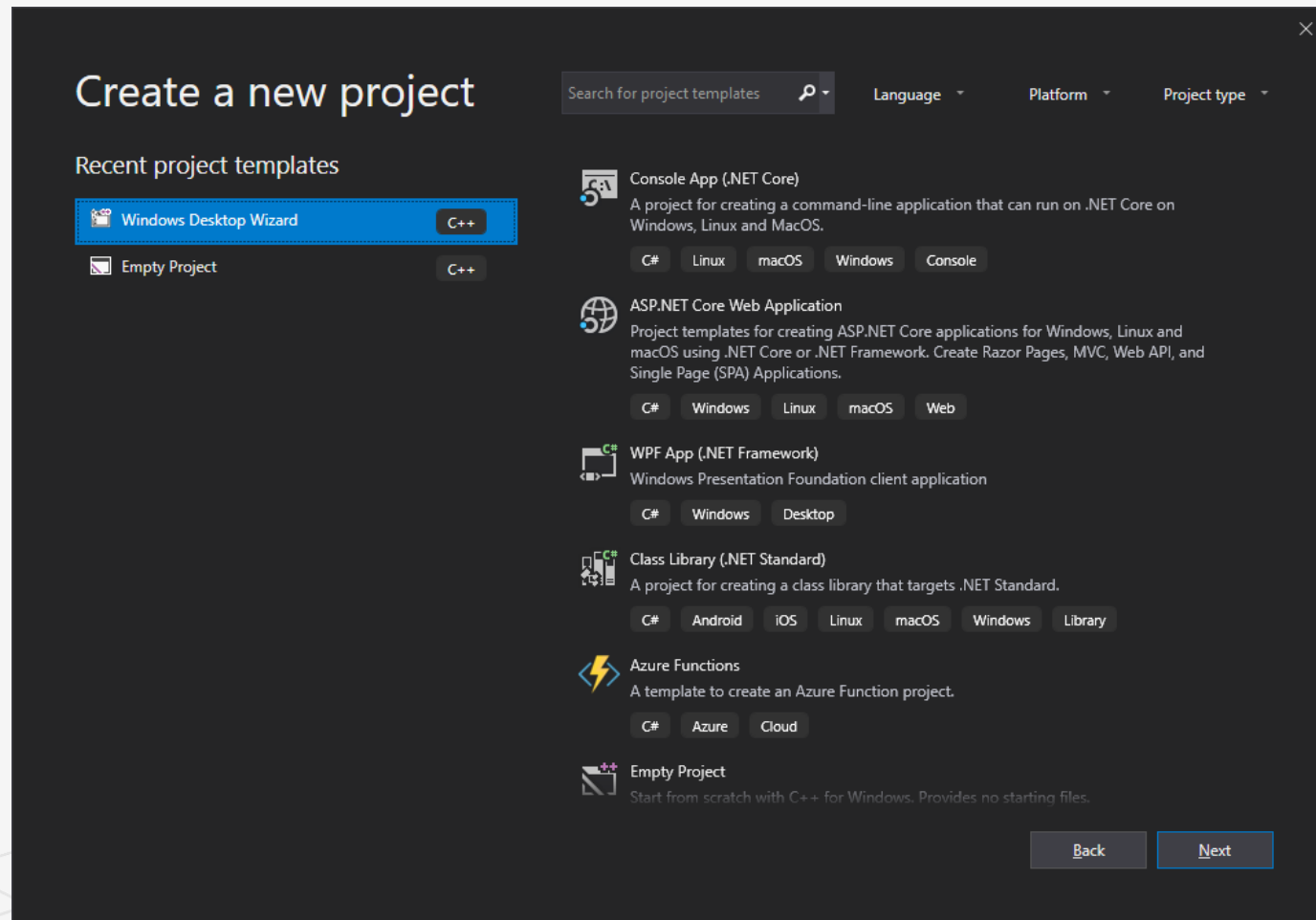
Demo

1. Download <https://github.com/hsalamat/DirectX> and unzip
2. Create an empty Window Desktop project under DirectX-master/Project1
3. Your project folder should be at the same level as "Common" and "Texture folder" under "DirectX-master"
4. Copy all the files and folder under "Week2/Triangle" folder under DirectXMaster and add them to your project1/Project1 folder (yes there are two of them)
5. Add the triangle0.cpp to your project from solution explorer.
6. Add all the files under "Common" folder to your project from solution explorer.
7. Create a filter called "Shaders" under your project from solution explorer
7. Add PS.hlsl or VS.hlsl shaders that exist under "Shaders" folder to your project from solution explorer under "Shaders filter"
8. Right click on each shader, go to HLSL compiler and change the shader type to Pixel/Vertex shader, entry point to "main" or the proper function name, and shader model to 5.1
9. if you are using visual studio 2019, you need to set the character mode to "Unicode" in the project properties under general section
9. if you are using visual studio 2019, you need to set the conformance mode to "none" in the project properties under C/C++ → Language
10. Run the application

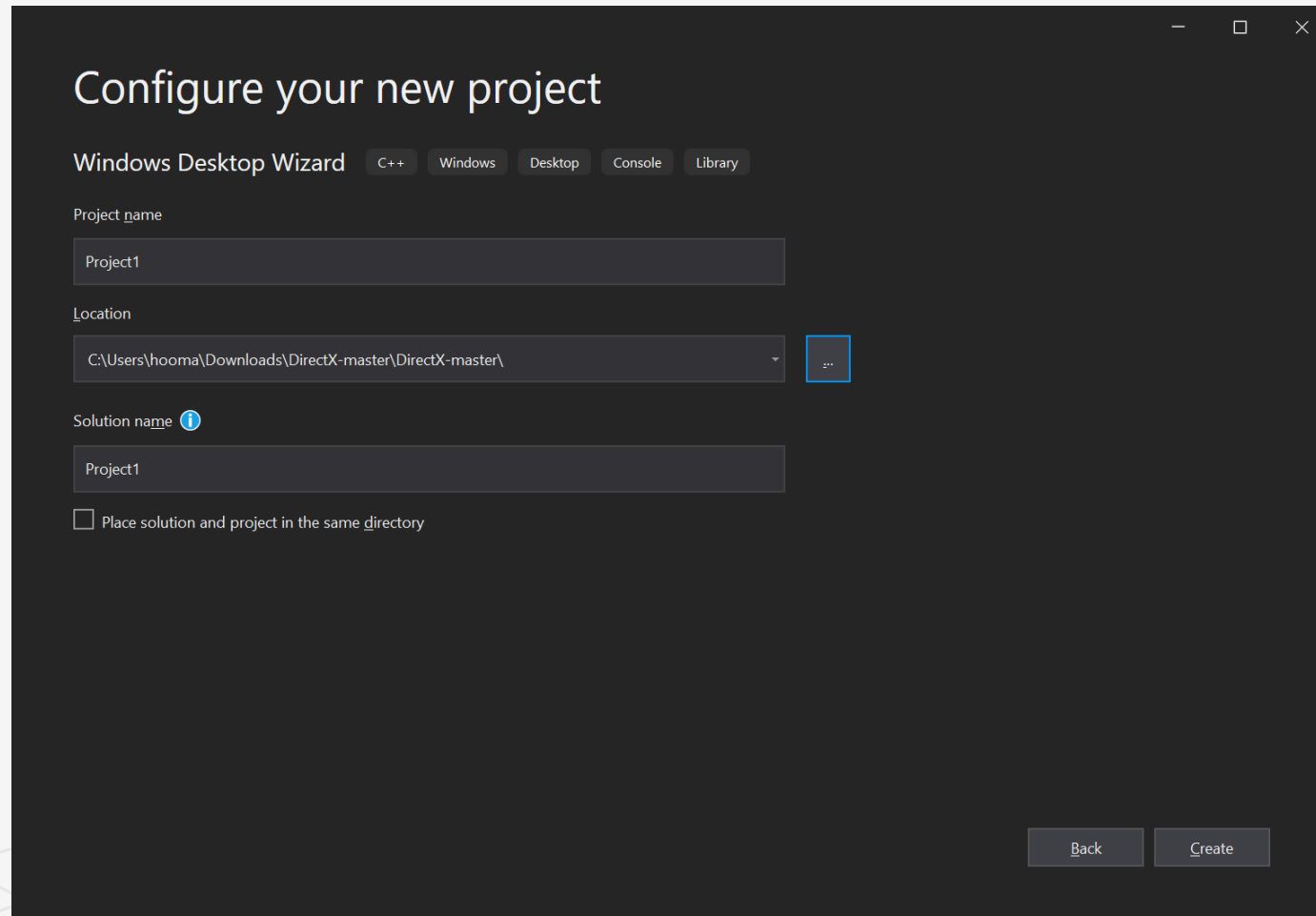
Down and Unload the files from the Github



Create a new project in Visual Studio 2019



Configure your new project



The image shows a 'Configure your new project' dialog box with a dark theme. At the top, it says 'Configure your new project' and 'Windows Desktop Wizard'. Below this are tabs for 'C++', 'Windows', 'Desktop', 'Console', and 'Library'. The 'Project name' field contains 'Project1'. The 'Location' field shows a file path 'C:\Users\hooma\Downloads\DirectX-master\DirectX-master\' with a blue box highlighting the '...' button to its right. The 'Solution name' field also contains 'Project1'. At the bottom, there is an unchecked checkbox labeled 'Place solution and project in the same directory'. The 'Back' and 'Create' buttons are at the bottom right.

Configure your new project

Windows Desktop Wizard C++ Windows Desktop Console Library

Project name

Project1

Location

C:\Users\hooma\Downloads\DirectX-master\DirectX-master\

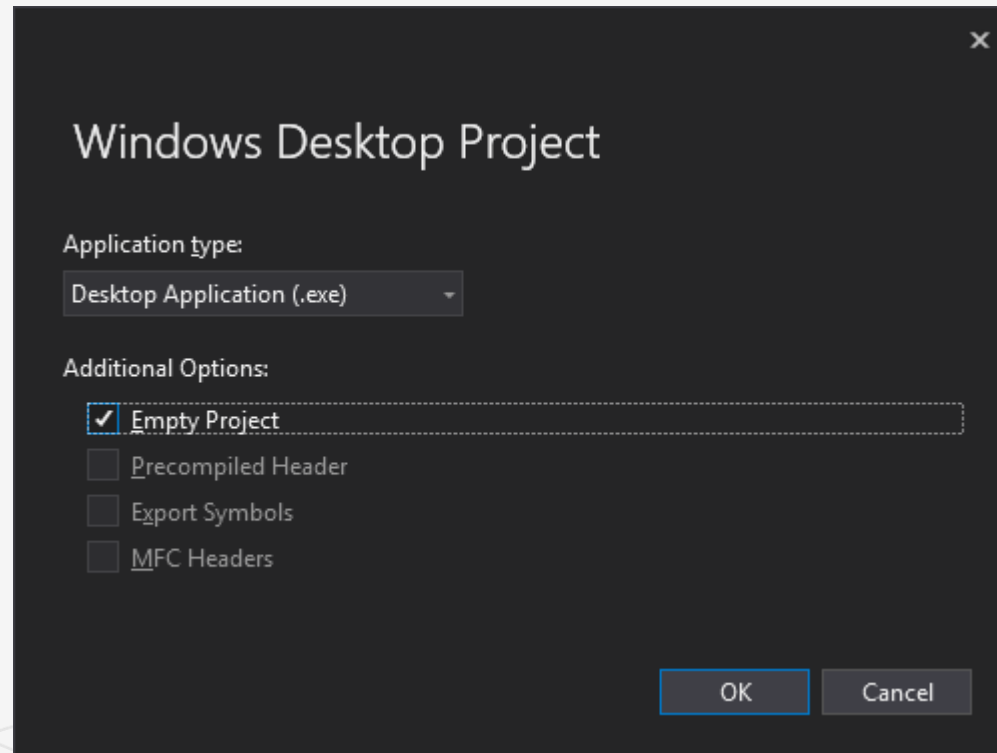
Solution name ⓘ

Project1

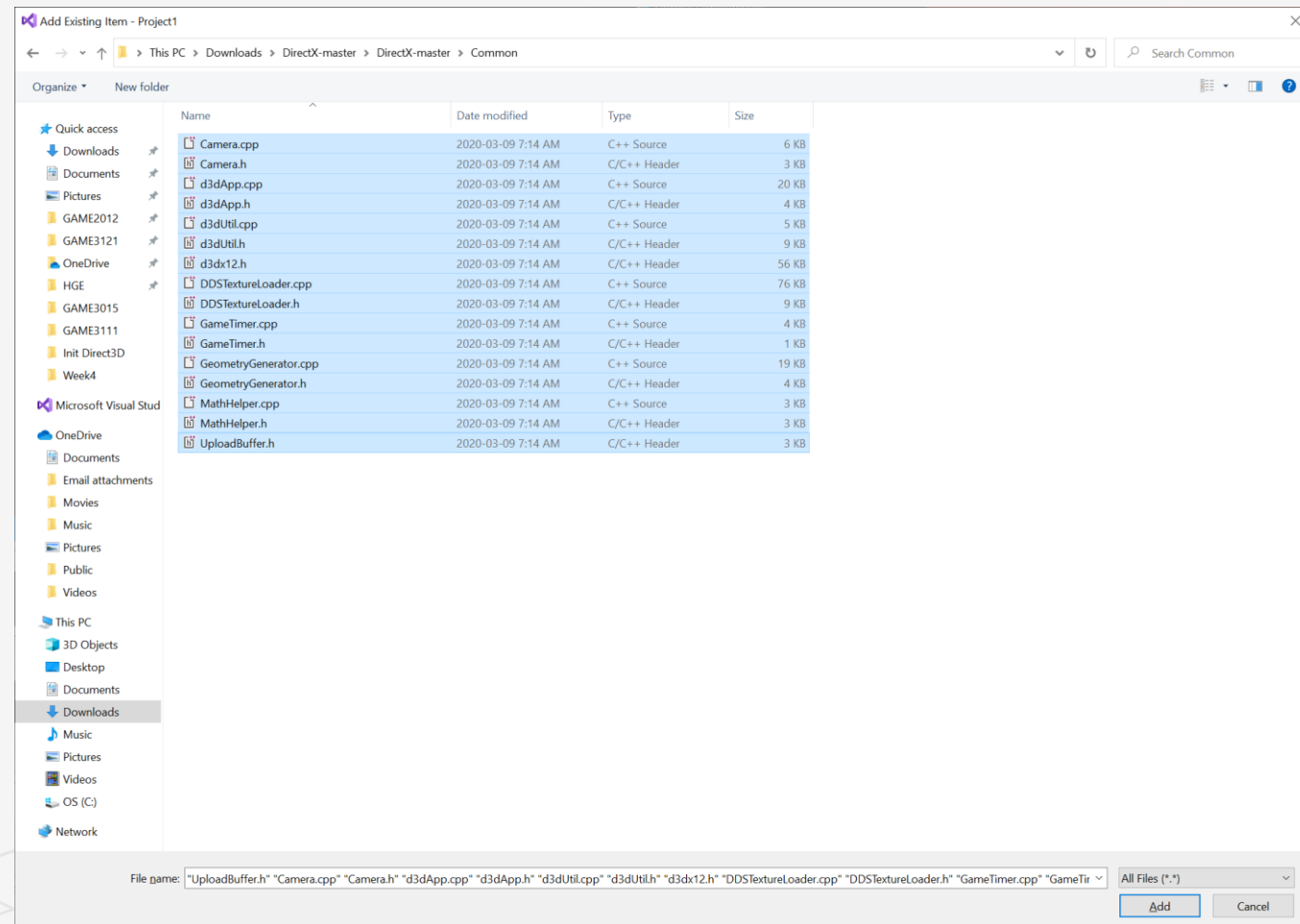
☐ Place solution and project in the same directory

Back Create

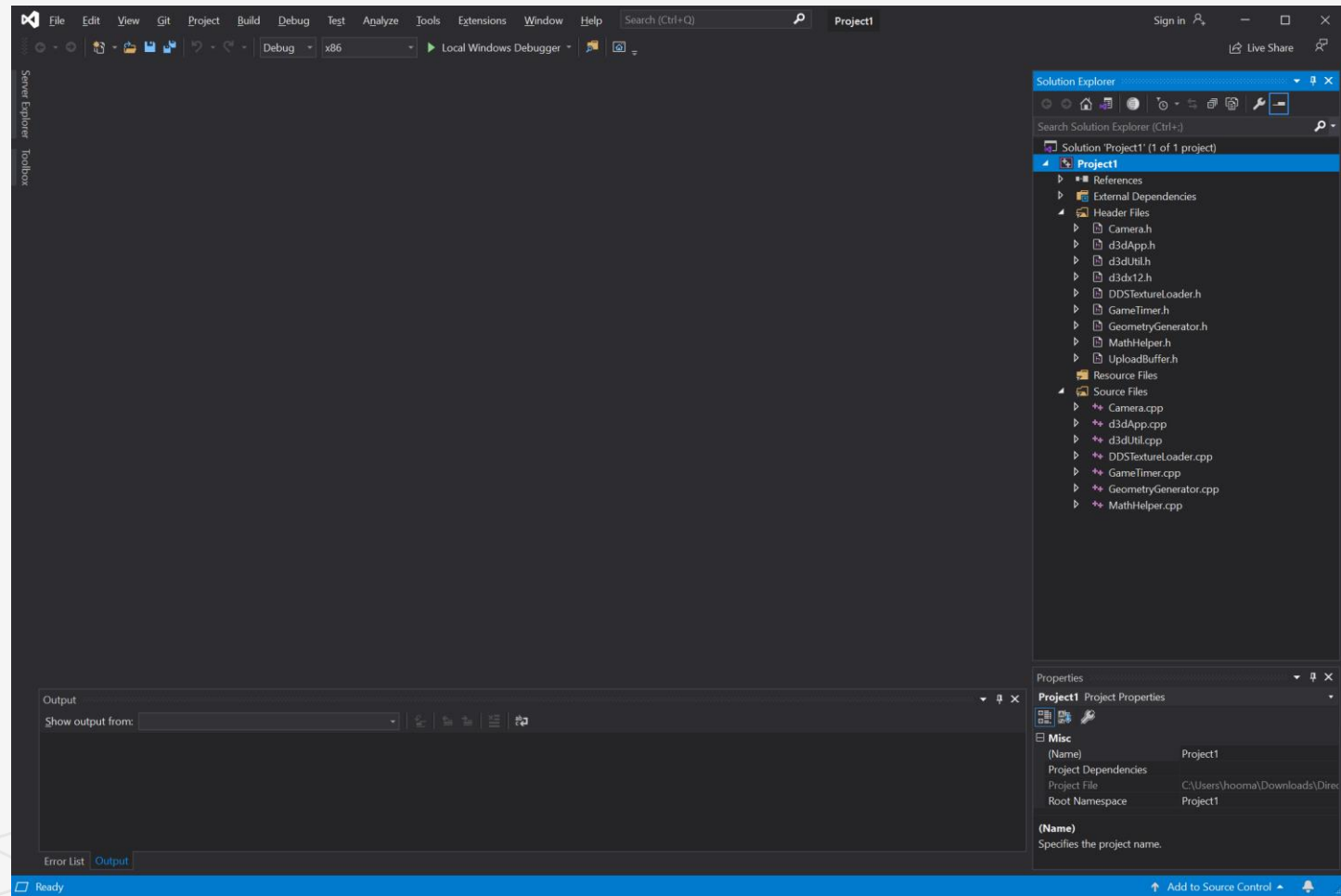
Create an Empty Windows Desktop Project



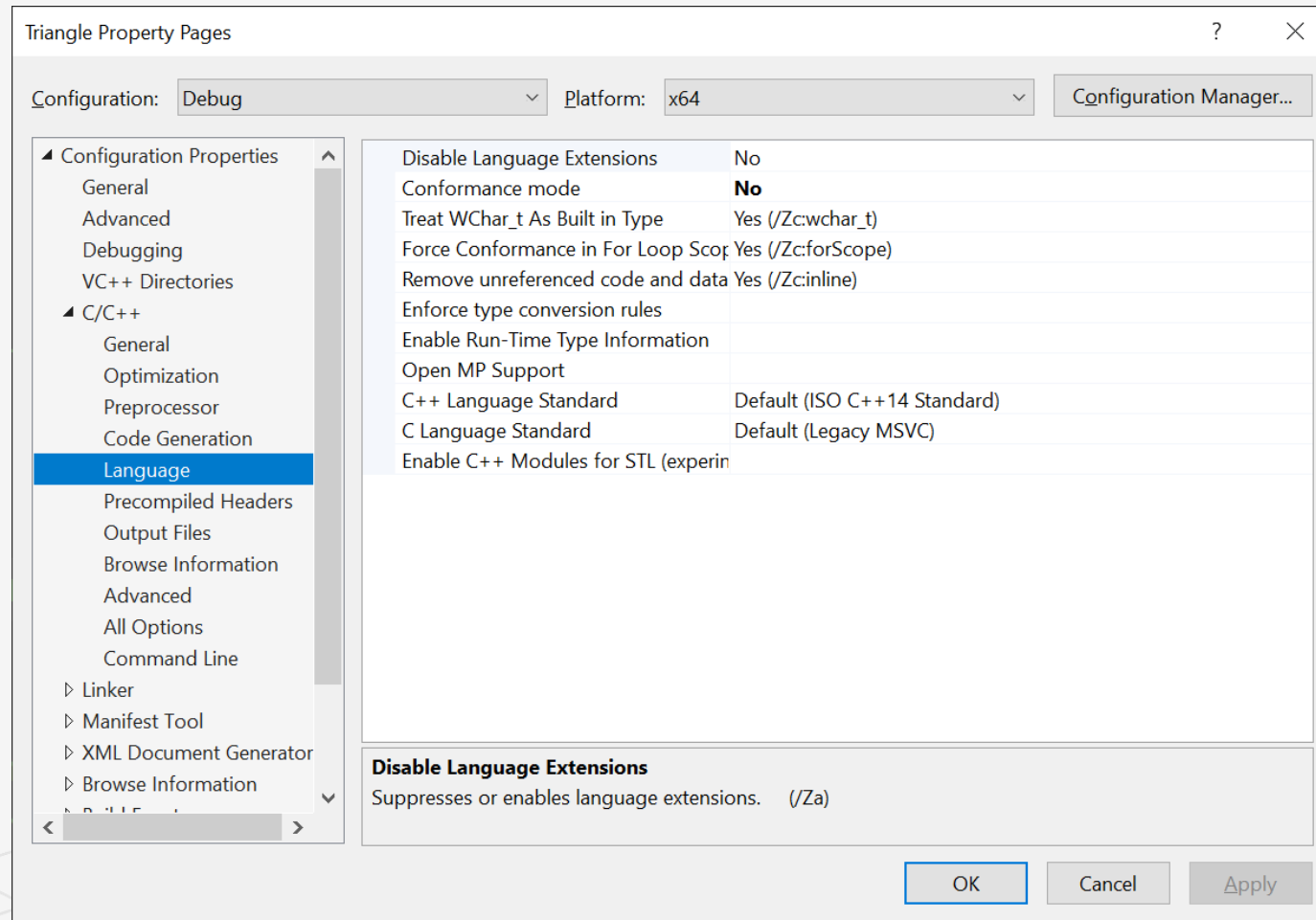
Add Existing Item



Project1

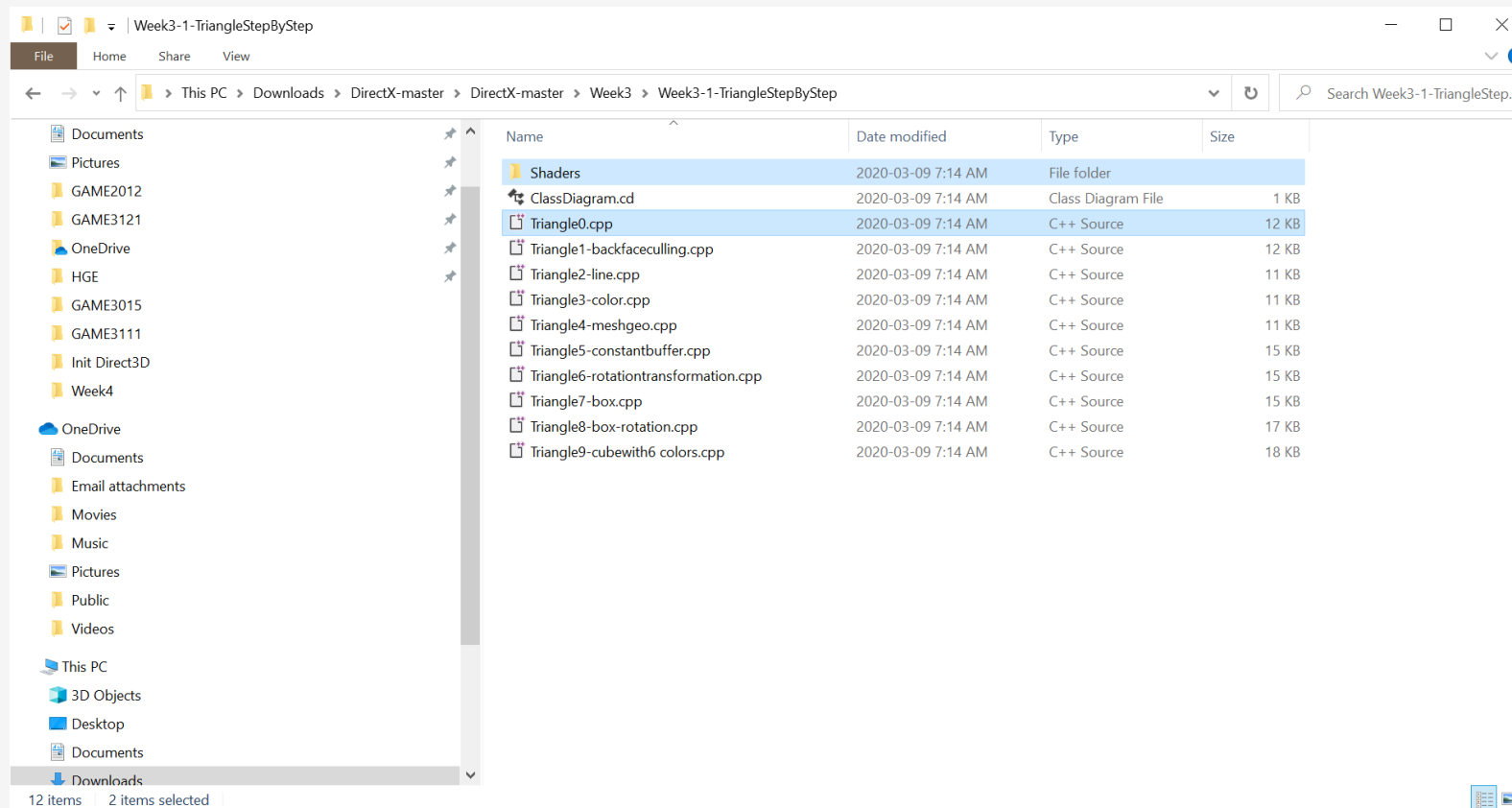


Change the conformance mode to "No"

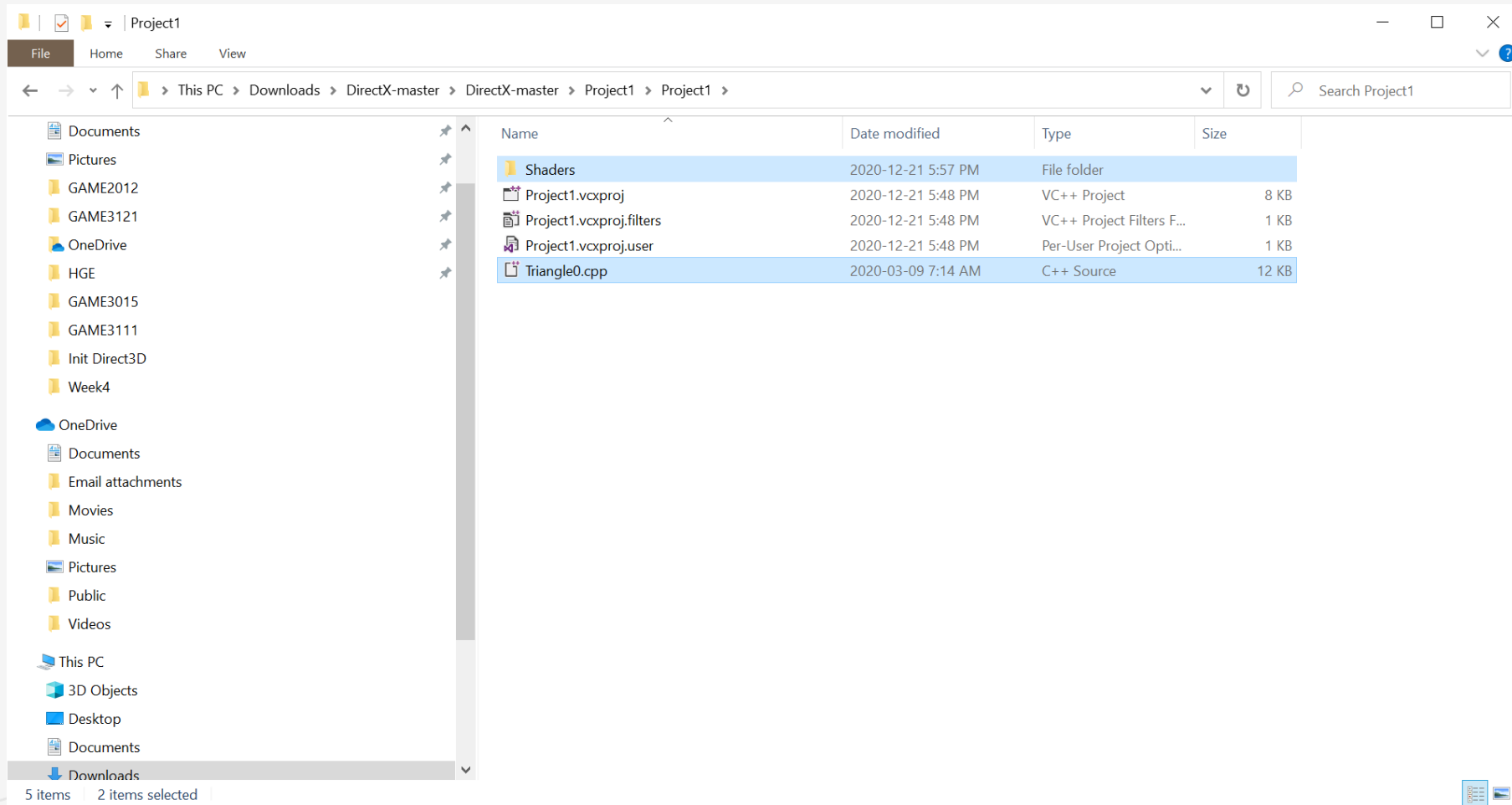


Copy the following files and place it under your project1

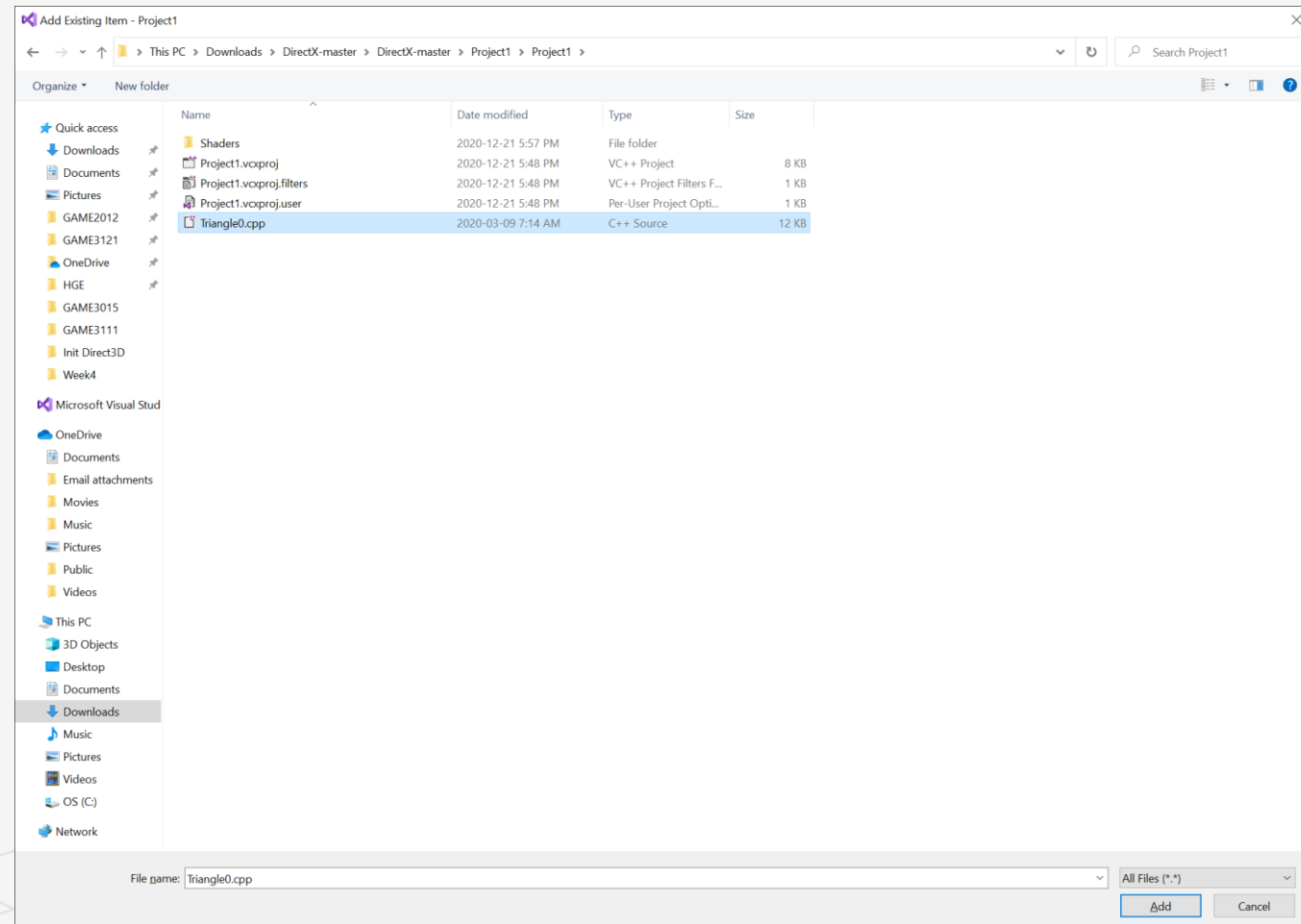
C:\Users\((USERNAME))\Downloads\DirectX-master\DirectX-master\Week3\Week3-1-TriangleStepByStep



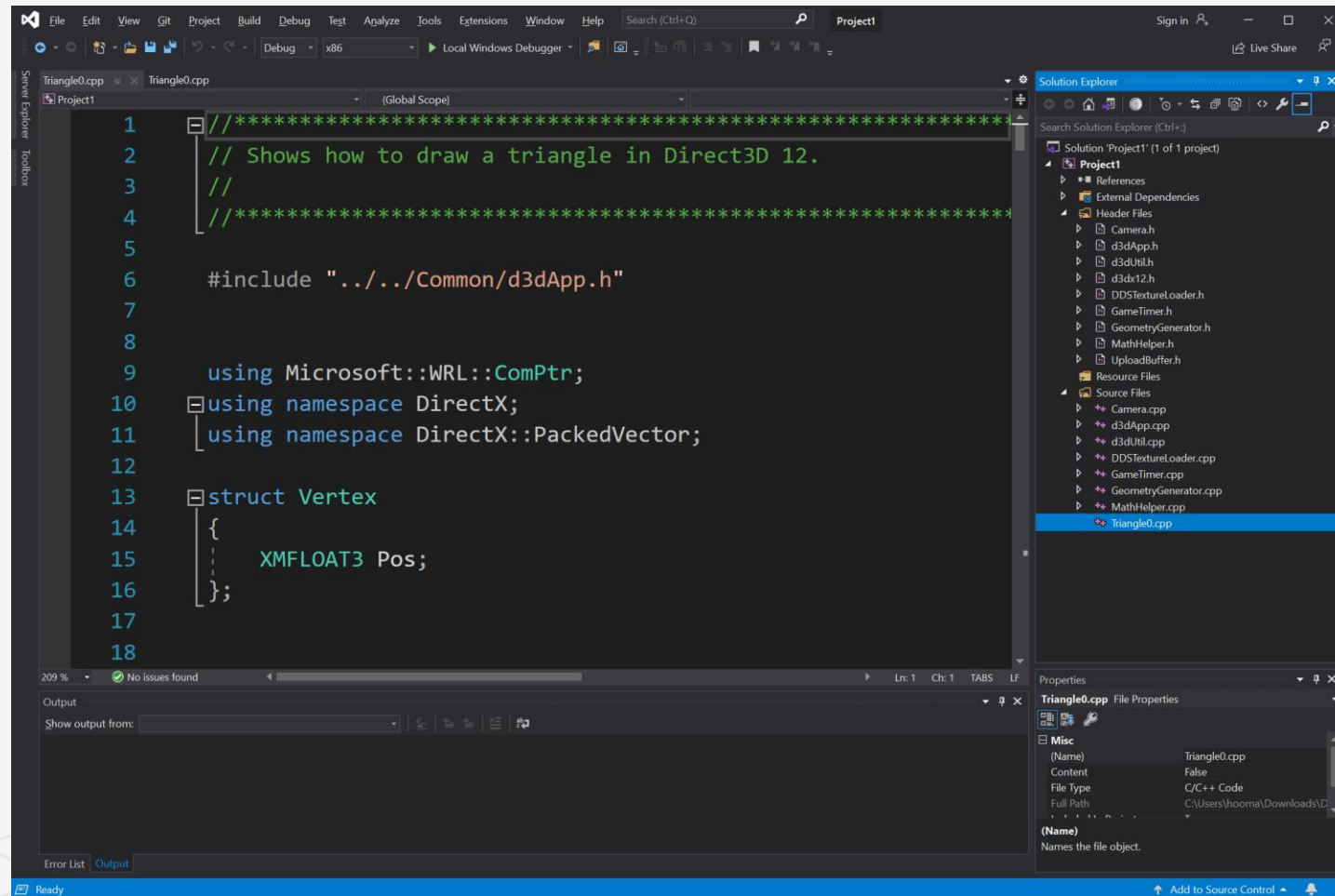
Paste it here



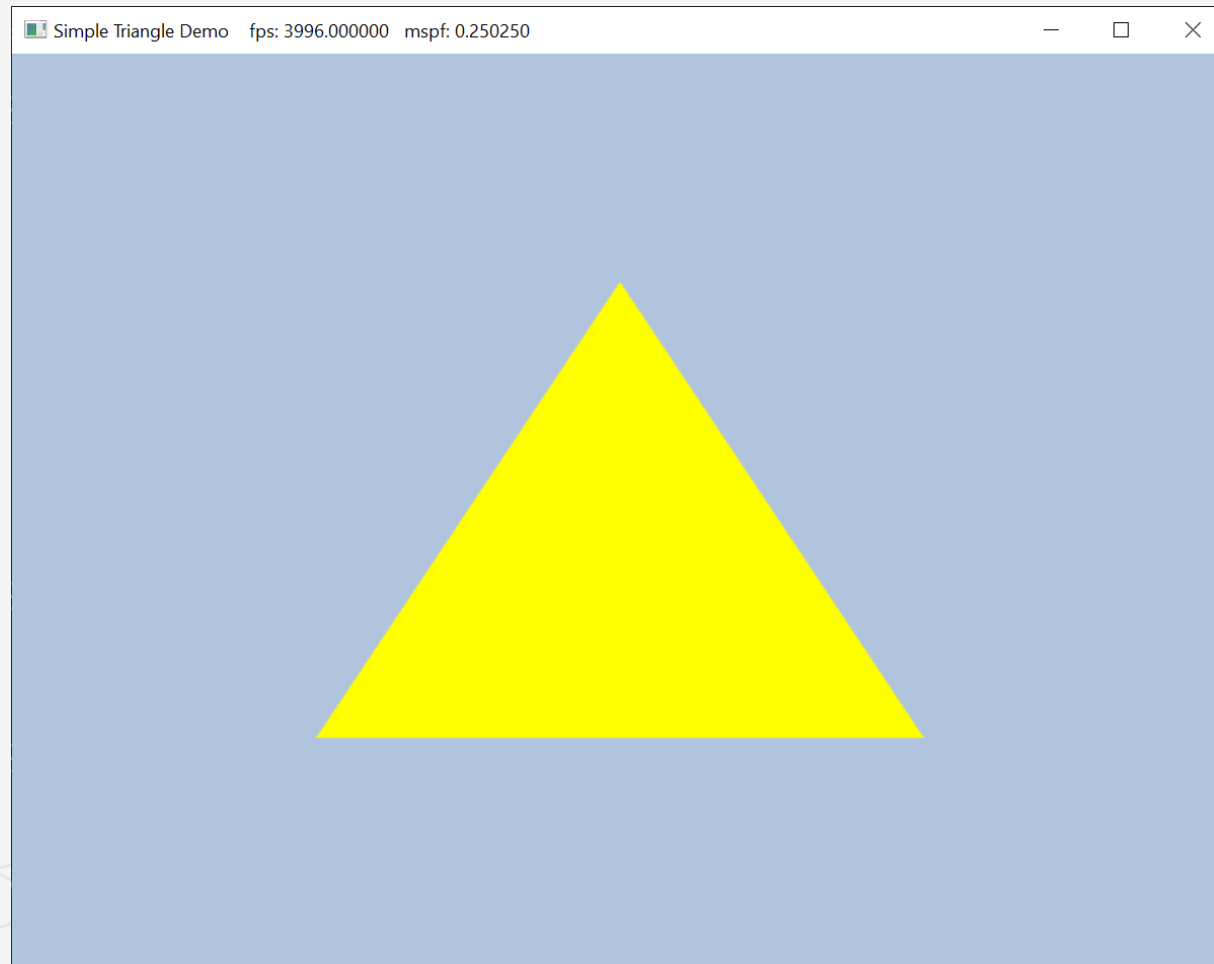
Add Triangle0.cpp to your project1



Change to 64 bit and Compile the code



Simple Triangle



Graphics diagnostics tools

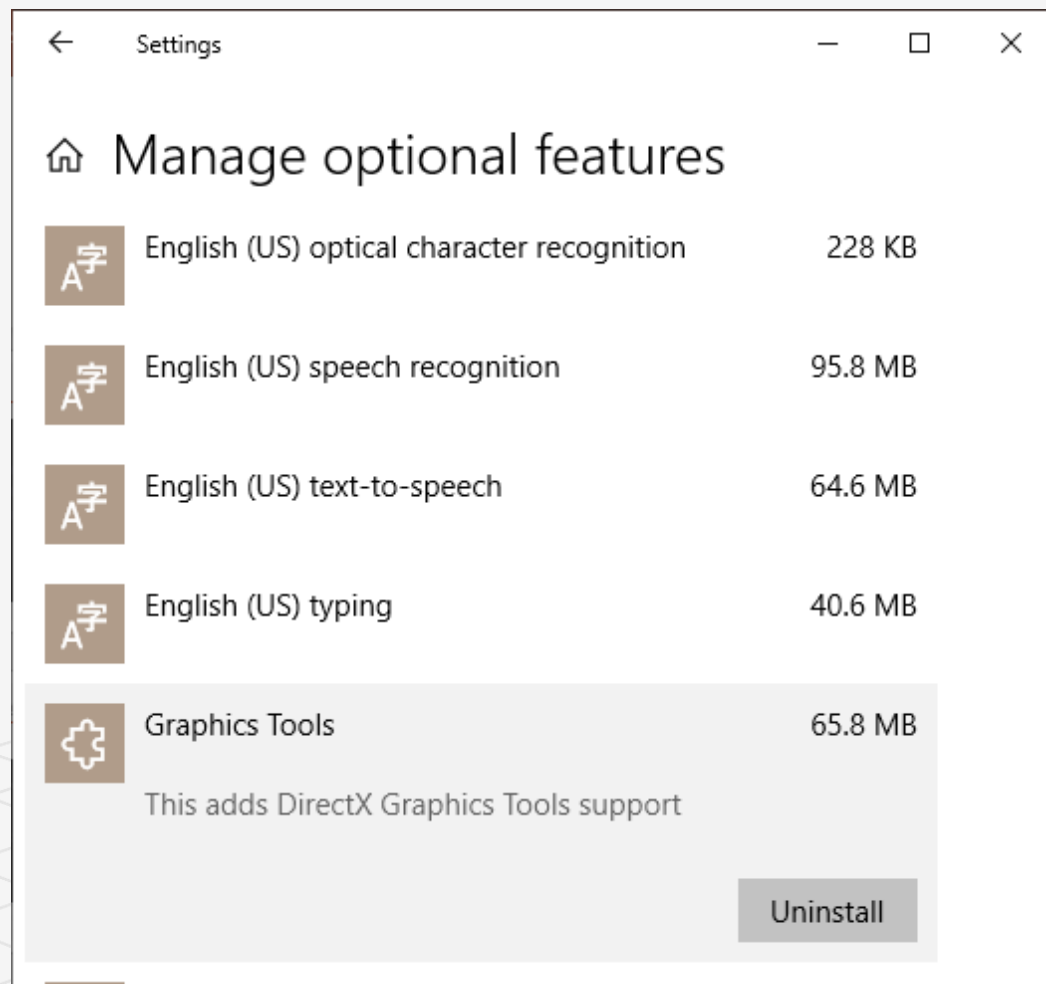
With Windows 10, the graphics diagnostic tools are now available from within Windows as an optional feature. To use the graphics diagnostic features provided in the runtime and Visual Studio to develop DirectX apps or games, install the optional Graphics Tools feature:

Go to **Settings**, select **Apps**, and then click **Manage optional features**.

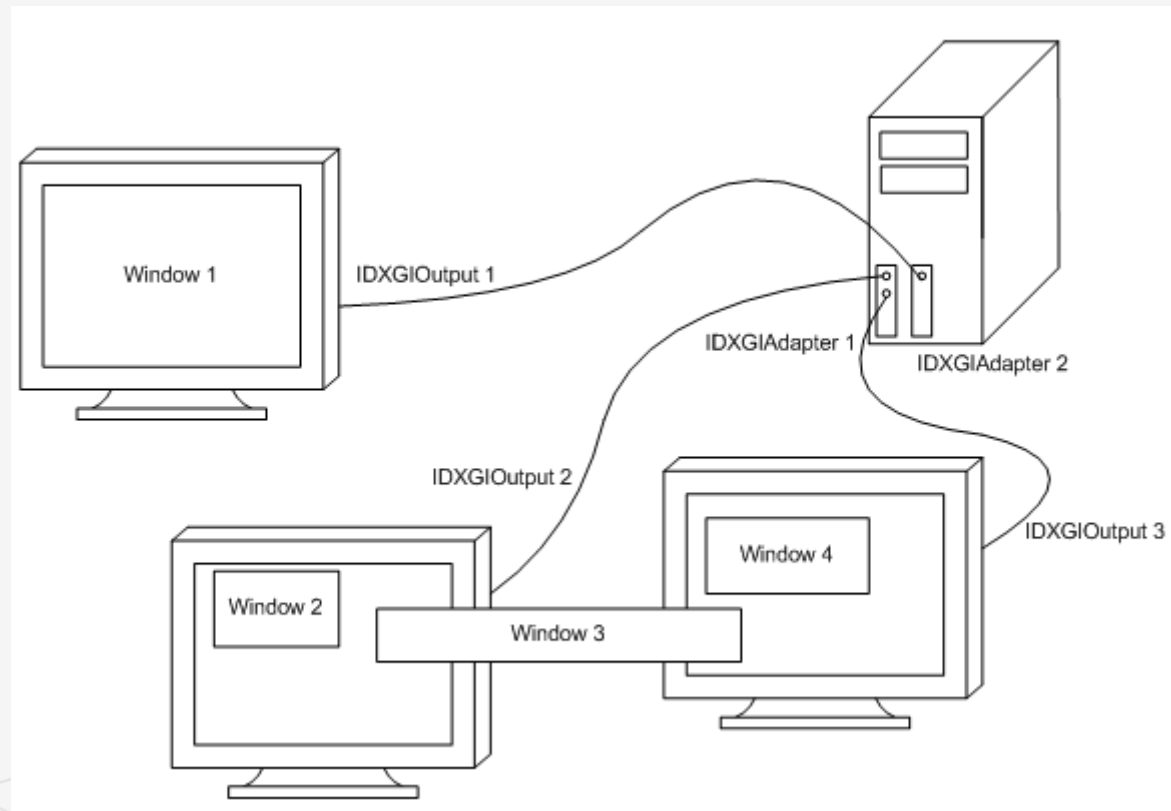
Click **Add a feature**

In the **Optional features** list, select **Graphics Tools** and then click **Install**.

Graphics diagnostics features include the ability to create Direct3D debug devices (via Direct3D SDK Layers) in the DirectX runtime, plus Graphics Debugging, Frame Analysis, and GPU Usage.

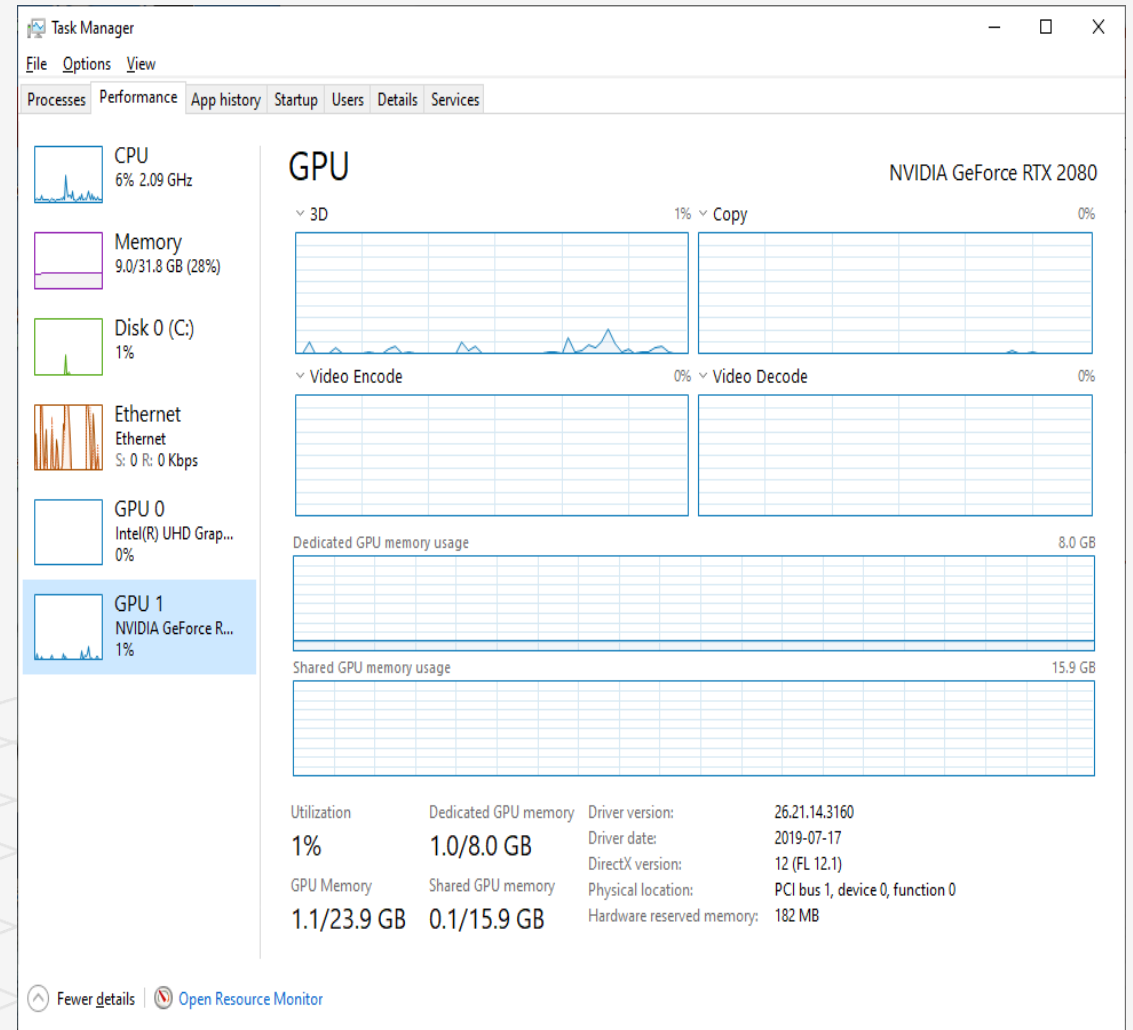
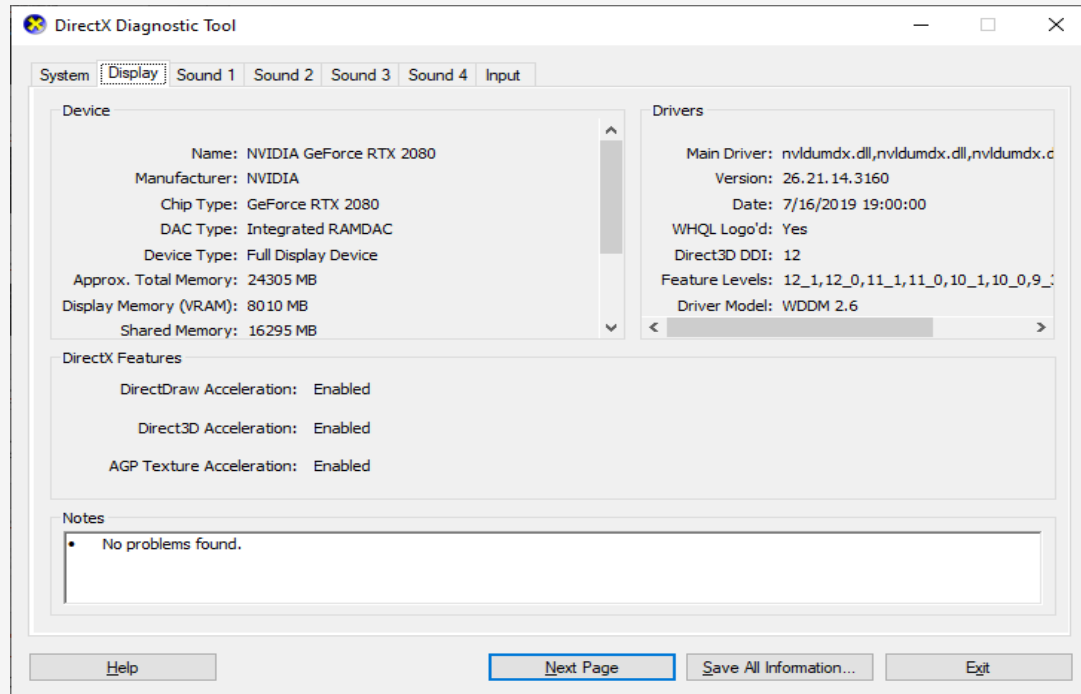


Enumerating the adapters



Check your Graphics card and GPU

1. On the Start menu, click Run.
2. In the Open box, type "dxdiag" and then click OK.
3. The DirectX Diagnostic Tool opens. Click the **Display** tab.



COM

Component Object Model (COM) is the technology that allows DirectX to be programming-language independent.

Most of the details of COM are hidden to us when programming DirectX with C++.

We obtain pointers to COM interfaces through special functions or by the methods of another COM interface—we do not create a COM interface with the C++ `new` keyword.

COM objects are reference counted; when we are done with an interface we call its `Release` method.

To help manage the lifetime of COM objects, the Windows Runtime Library (WRL) provides the `Microsoft::WRL::ComPtr` class (`#include <wrl.h>`), which can be thought of as a smart pointer for COM objects.

The three main `ComPtr` methods we use:

1. `Get`: Returns a pointer to the underlying COM interface.
2. `GetAddressOf`: Returns the address of the pointer to the underlying COM interface.
3. `Reset`: Sets the `ComPtr` instance to `nullptr` and decrements the reference count of the underlying COM interface.

Understanding COM Components

```
#include <Windows.h>
#include <Wininet.h>
#include <ShlObj.h>
#include <iostream>
int main()
{
    CoInitialize(nullptr);
    IActiveDesktop* pDesktop = nullptr;
    WCHAR wszWallpaper[MAX_PATH];
    //factory function to create a COM object
    CoCreateInstance(
        CLSID_ActiveDesktop, //underlying COM object that we want to create
        nullptr,
        CLSCTX_INPROC_SERVER, //context of the object --> process context
        __uuidof(IActiveDesktop),
        reinterpret_cast<void**>(&pDesktop) //this pointer will be filled
    );
    pDesktop->GetWallpaper(wszWallpaper, MAX_PATH, 0);
    pDesktop->Release();
    std::wcout << wszWallpaper;
    CoUninitialize();
    std::cin.get();
    return 0;
}
```

```
#pragma region Add this section below CoUninitialize
//the following section create a short cut to the wall paper
IShellLink* pLink = nullptr;
CoCreateInstance(
    CLSID_ShellLink, //underlying COM object that we want to create
    nullptr,
    CLSCTX_INPROC_SERVER, //context of the object --> process context
    __uuidof(IShellLink),
    reinterpret_cast<void**>(&pLink) //this pointer will be filled
);
pLink->SetPath(wszWallpaper);
IPersistFile* pPersist = nullptr;
pLink->QueryInterface(__uuidof(IPersistFile), reinterpret_cast<void**> (&pPersist));
pPersist->Save(L"c:\\hooman\\wallpaper.lnk", FALSE);
pPersist->Release();
pLink->Release();
#pragma endregion
```

Get Started with Win32 and C++

<https://docs.microsoft.com/en-us/windows/win32/learnwin32/learn-to-program-for-windows>

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE prevInstance,
    PSTR cmdLine, int showCmd)
{
    try
    {
        InitDirect3DApp theApp(hInstance);
        if(!theApp.Initialize())
            return 0;

        return theApp.Run();
    }
    catch(DxException& e)
    {
        MessageBox(nullptr, e.ToString().c_str(), L"HR Failed", MB_OK);
        return 0;
    }
}
```

WinMain --> every win32 application needs a WinMain() function, like console application needs main()

HINSTANCE, PSTR.. are part of <Windows.h>

Windows is about Windows (window class that you register and instance of the class) and Messages
CmdLine receives the parameter that somebody passes.

D3DApp::LogAdapters

```
void D3DApp::LogAdapters()
{
    UINT i = 0;
    IDXGIAdapter* adapter = nullptr;
    std::vector<IDXGIAdapter*> adapterList;
    while(mdxgiFactory->EnumAdapters(i, &adapter) != DXGI_ERROR_NOT_FOUND)
    {
        DXGI_ADAPTER_DESC desc;
        adapter->GetDesc(&desc);

        std::wstring text = L"***Adapter: ";
        text += desc.Description;
        text += L"\n";

        OutputDebugString(text.c_str());
        adapterList.push_back(adapter);
        ++i;
    }
    for(size_t i = 0; i < adapterList.size(); ++i)
    {
        LogAdapterOutputs(adapterList[i]);
        ReleaseCom(adapterList[i]);
    }
}
```

***Adapter: NVIDIA GeForce RTX 2080 (NVIDIA Turing Architecture – Realtime Ray Tracing, AI: <https://www.nvidia.com/en-us/geforce/turing/>)

***Adapter: Intel(R) UHD Graphics 630 (Integrated graphics processor introduced by Intel in 2017 for the performance desktop Coffee Lake microprocessors)

***Adapter: Microsoft Basic Render Driver (is a software adapter included with Windows 10. the "Microsoft Basic Render Driver" has no display outputs!)

***Output: \\.\DISPLAY1

D3DApp::LogOutputDisplayModes

In order to get optimal full-screen performance, the specified display mode (including refresh rate), must match exactly a display mode the monitor supports. We can get a list of all supported display modes an output supports in that format with the following code:

```
void D3DApp::LogOutputDisplayModes(IDXGIOutput* output, DXGI_FORMAT format)
{
    UINT count = 0;
    UINT flags = 0;

    // Call with nullptr to get list count.
    output->GetDisplayModeList(format, flags, &count, nullptr);

    std::vector<DXGI_MODE_DESC> modeList(count);
    output->GetDisplayModeList(format, flags, &count, &modeList[0]);

    for(auto& x : modeList)
    {
        UINT n = x.RefreshRate.Numerator;
        UINT d = x.RefreshRate.Denominator;
        std::wstring text =
            L"Width = " + std::to_wstring(x.Width) + L" " +
            L"Height = " + std::to_wstring(x.Height) + L" " +
            L"Refresh = " + std::to_wstring(n) + L"/" + std::to_wstring(d) +
            L"\n";

        ::OutputDebugString(text.c_str());
    }
}
```

***Output: [\\DISPLAY1](#) (A system can have several monitors. A monitor is an example of a *display output*. An output is represented by the IDXGIOutput interface)

Width = 640 Height = 480 Refresh = 25174825/420000

Width = 720 Height = 480 Refresh = 27000000/450450

...

DXGI_MODE_DESC

Each monitor has a set of display modes it supports. A display mode refers to the following data in DXGI_MODE_DESC:

```
typedef struct DXGI_MODE_DESC
{
    UINT Width;

    UINT Height;

    DXGI_RATIONAL RefreshRate;

    DXGI_FORMAT Format;

    DXGI_MODE_SCANLINE_ORDER ScanlineOrdering;

    DXGI_MODE_SCALING Scaling;
} DXGI_MODE_DESC;
```

DXGI_FORMAT Examples

DXGI_FORMAT_UNKNOWN	The format is not known.
DXGI_FORMAT_R32G32B32A32_TYPELESS	A four-component, 128-bit typeless format that supports 32 bits per channel including alpha. ¹
DXGI_FORMAT_R32G32B32A32_FLOAT	A four-component, 128-bit floating-point format that supports 32 bits per channel including alpha. ^{1,5,8}
DXGI_FORMAT_R32G32B32A32_UINT	A four-component, 128-bit unsigned-integer format that supports 32 bits per channel including alpha. ¹
DXGI_FORMAT_R32G32B32A32_SINT	A four-component, 128-bit signed-integer format that supports 32 bits per channel including alpha.

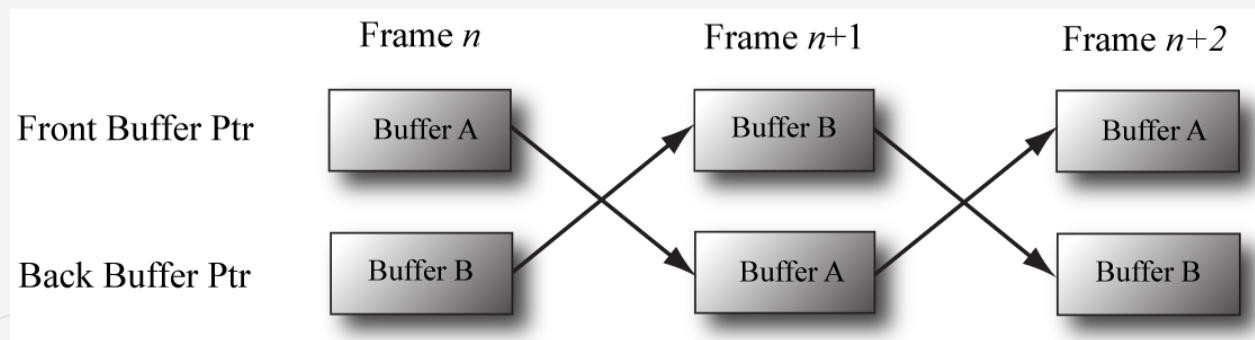
The Swap Chain and Page Flipping

To avoid flickering in animation, it is best to draw an entire frame of animation into an off-screen texture called the back buffer. Once the entire scene has been drawn to the back buffer for the given frame of animation, it is presented to the screen as one complete frame;

To implement this, two texture buffers are maintained by the hardware, one called the *front buffer* and a second called the *back buffer*. The front buffer stores the image data currently being displayed on the monitor, while the next frame of animation is being drawn to the back buffer. Using two buffers (front and back) is called *double buffering*.

After the frame has been drawn to the back buffer, the roles of the back buffer and front buffer are reversed: the back buffer becomes the front buffer and the front buffer becomes the back buffer for the next frame of animation. Swapping the roles of the back and front buffers is called *presenting*.

The front and back buffer form a *swap chain*. In Direct3D, a swap chain is represented by the IDXGISwapChain interface. This interface stores the front and back buffer textures, as well as provides methods for resizing the buffers (`IDXGISwapChain::ResizeBuffers`) and presenting (`IDXGISwapChain::Present`).

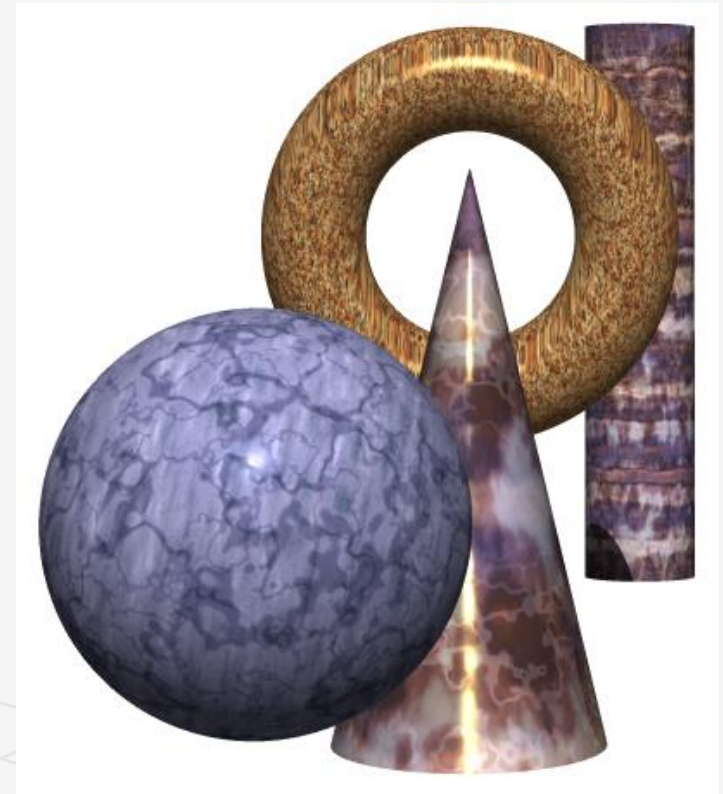


Depth Buffering

The *depth buffer* is an example of a texture that does not contain image data, but rather depth information about a particular pixel. The possible depth values range from 0.0 to 1.0, where 0.0 denotes the closest an object in the view frustum can be to the viewer and 1.0 denotes the farthest an object in the view frustum can be from the viewer.

There is a one-to-one correspondence between each element in the depth buffer and each pixel in the back buffer (i.e., the i th element in the back buffer corresponds to the i th element in the depth buffer). So if the back buffer had a resolution of 1280×1024 , there would be 1280×1024 depth entries.

In order for Direct3D to determine which pixels of an object are in front of another, it uses a technique called *depth buffering* or *z-buffering*. With depth buffering, the order in which we draw the objects does not matter.

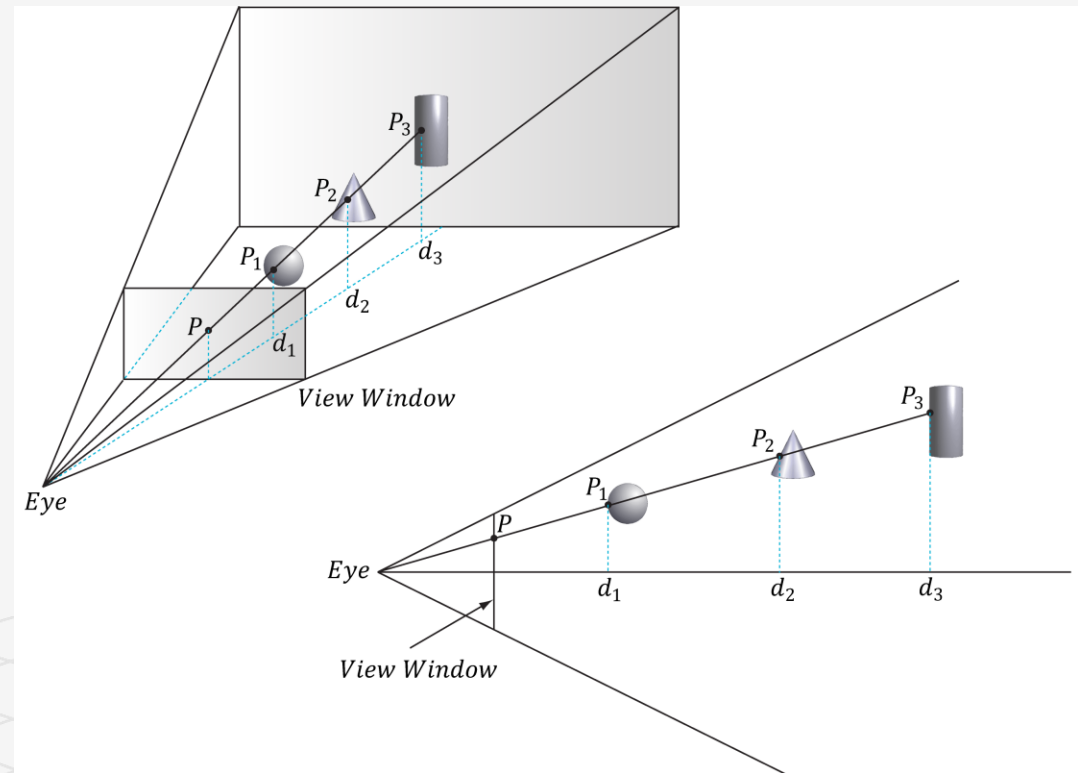


Depth Buffer

The depth buffer is a texture, so it must be created with certain data formats. The formats used for depth buffering are as follows:

1. `DXGI_FORMAT_D32_FLOAT_S8X24_UINT`: Specifies a 32-bit floating-point depth buffer, with 8-bits (unsigned integer) reserved for the stencil buffer mapped to the $[0, 255]$ range and 24-bits not used for padding.
2. `DXGI_FORMAT_D32_FLOAT`: Specifies a 32-bit floating-point depth buffer.
3. `DXGI_FORMAT_D24_UNORM_S8_UINT`: Specifies an unsigned 24-bit depth buffer mapped to the $[0, 1]$ range with 8-bits (unsigned integer) reserved for the stencil buffer mapped to the $[0, 255]$ range.
4. `DXGI_FORMAT_D16_UNORM`: Specifies an unsigned 16-bit depth buffer mapped to the $[0, 1]$ range.

An application is not required to have a stencil buffer, but if it does, the stencil buffer is always attached to the depth buffer. For example, the 32-bit format `DXGI_FORMAT_D24_UNORM_S8_UINT` uses 24-bits for the depth buffer and 8-bits for the stencil buffer. For this reason, the depth buffer is better called the depth/stencil buffer.



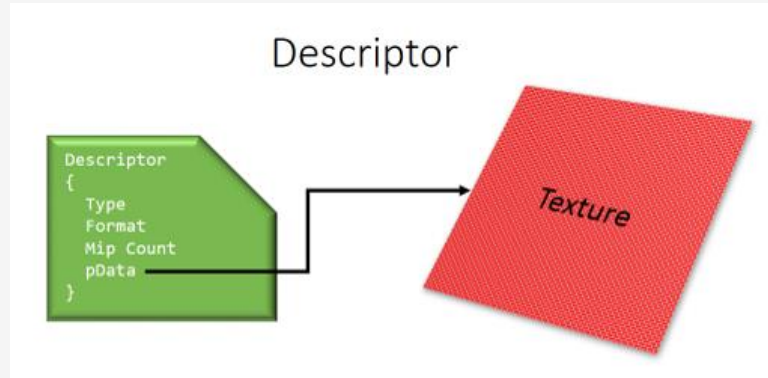
Resources and Descriptors

During the rendering process, the GPU will write to resources (e.g., the back buffer, the depth/stencil buffer), and read from resources (e.g., textures that describe the appearance of surfaces, buffers that store the 3D positions of geometry in the scene).

Before we issue a draw command, we need to *bind* (or link) the resources to the rendering pipeline that are going to be referenced in that draw call. Some of the resources may change per draw call, so we need to update the bindings per draw call if necessary.

However, GPU resources are not bound directly. Instead, a resource is referenced through a *descriptor* object, which can be thought of as lightweight structure that describes the resource to the GPU. Given a resource descriptor, the GPU can get the actual resource data and know the necessary information about it.

Why go to this extra level of indirection with descriptors? The reason is that GPU resources are essentially generic chunks of memory. Resources are kept generic so they can be used at different stages of the rendering pipeline; a common example is to use a texture as a render target (i.e., Direct3D draws into the texture) and later as a shader resource (i.e., the texture will be sampled and serve as input data for a shader).

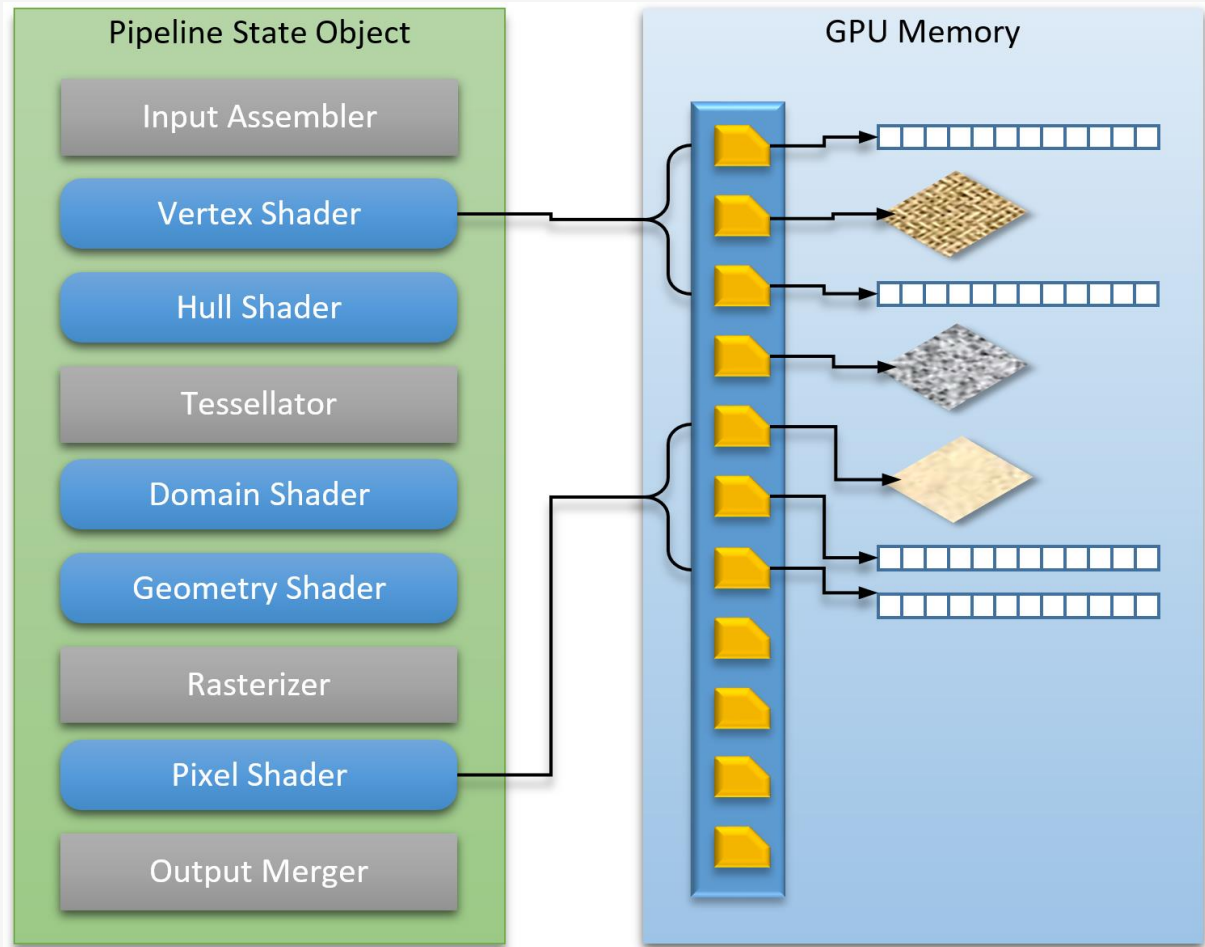


Descriptors have a type, and the type implies how the resource will be used. The types of descriptors we use are:

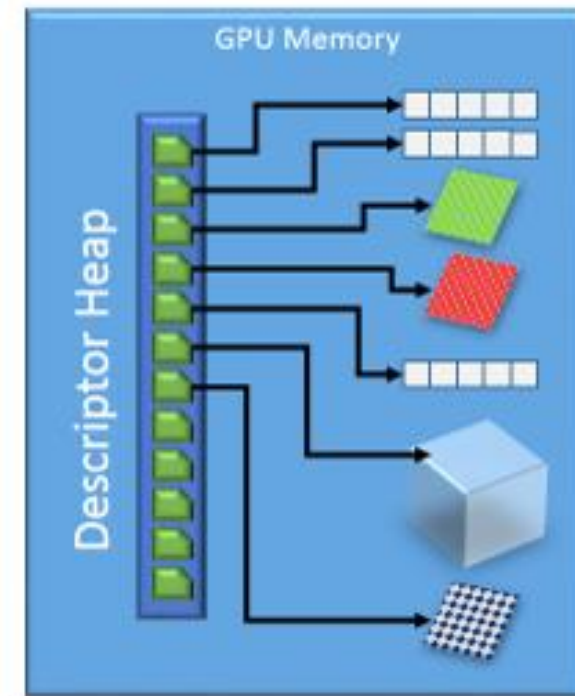
1. **CBV/SRV/UAV** descriptors describe constant buffers, shader resources and unordered access view resources.
2. **Sampler** descriptors describe sampler resources (used in texturing).
3. **RTV** descriptors describe render target resources.
4. **DSV** descriptors describe depth/stencil resources.

A **descriptor heap** is an array of descriptors; it is the memory backing for all the descriptors of a particular type your application uses. You will need a separate descriptor heap for each type of descriptor. You can also create multiple heaps of the same descriptor type.

Descriptor Heap



Descriptor Heaps



Multisampling Theory

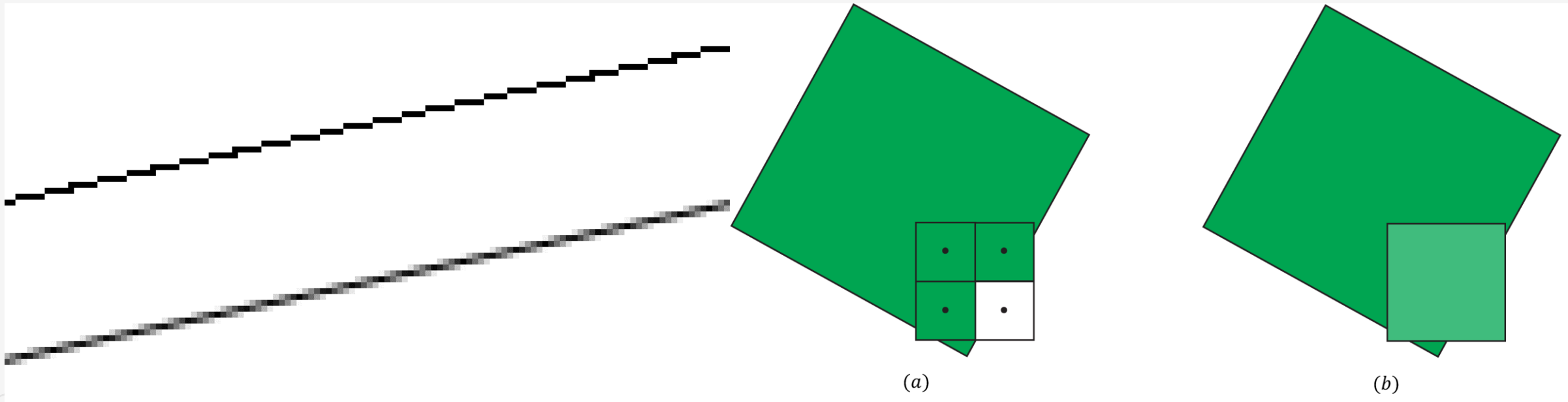
Because the pixels on a monitor are not infinitely small, an arbitrary line cannot be represented perfectly on the computer monitor. When increasing the monitor resolution is not possible or not enough, we can apply *antialiasing* techniques.

Supersampling: **First make the back buffer and depth buffer 4X bigger than the screen resolution.** The 3D scene is then rendered to the back buffer at this larger resolution. Then, when it comes time to present the back buffer to the screen, the back buffer is **resolved** (or **downsampled**) such that 4 pixel block colors are averaged together to get an averaged pixel color.

Supersampling is expensive because it increases the amount of pixel processing and memory by fourfold.

Direct3D supports a compromising antialiasing technique called **multisampling**, which shares some computational information across subpixels making it less expensive than supersampling. Assuming we are using 4X multisampling (4 subpixels per pixel), multisampling also uses a back buffer and depth buffer 4X bigger than the screen resolution; however, **instead of computing the image color for each subpixel, it computes it only once per pixel, at the pixel center**, and then shares that color information with its subpixels based on visibility.

a) one pixel that crosses the edge of a polygon b) apply multisampling technique by computing the color per pixel (not subpixels!)



D3D12_FEATURE Enumeration

Specifies a Direct3D 12 feature or feature set to query about. When you want to query for the level to which an adapter supports a feature, pass one of these values to [ID3D12Device::CheckFeatureSupport](#).

```
HRESULT ID3D12Device::CheckFeatureSupport(  
    D3D12_FEATURE Feature,  
    void *pFeatureSupportData,  
    UINT FeatureSupportDataSize);
```

Feature: A member of the D3D12_FEATURE enumerated type identifying the type of features we want to check the support:

1. D3D12_FEATURE_D3D12_OPTIONS: Checks support for various Direct3D12 features.
2. D3D12_FEATURE_ARCHITECTURE1: Checks support for hardware architecture features like Unified Memory Architecture (UMA).
3. D3D12_FEATURE_FEATURE_LEVELS: Checks feature level support.
4. D3D12_FEATURE_FORMAT_SUPPORT: Check feature support for a given texture format (e.g., can the format be used as a render target, can the format be used with blending).
5. D3D12_FEATURE_MULTISAMPLE_QUALITY_LEVELS: Check multisampling feature support.

pFeatureSupportData: Pointer to a data structure to retrieve the feature support information.

FeatureSupportDataSize: The size of the data structure passed into pFeatureSupportData the parameter.

Resource Residency

A complex game will use a lot of resources such as textures and 3D meshes, but many of these resources will not be needed by the GPU all the time.

For example, if we imagine a game with an outdoor forest that has a large cave in it, the cave resources will not be needed until the player enters the cave, and when the player enters the cave, the forest resources will no longer be needed.

In Direct3D 12, applications manage resource residency (essentially, whether a resource is in GPU memory) by evicting resources from GPU memory and then making them resident on the GPU again as needed. The basic idea is to minimize how much GPU memory the application is using because there might not be enough to store every resource for the entire game, or the user has other applications running that require GPU memory.

By default, when a resource is created it is made resident and it is evicted when it is destroyed. However, an application can manually control residency with the following methods:

```
HRESULT ID3D12Device::MakeResident(
```

```
    UINT NumObjects,
```

```
    ID3D12Pageable* const* ppObjects);
```

```
HRESULT ID3D12Device::Evict(
```

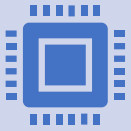
```
    UINT NumObjects,
```

```
    ID3D12Pageable* const* ppObjects);
```

For both methods, the second parameter is an array of ID3D12Pageable resources, and the first parameter is the number of resources in the array.

******For simplicity and due to our demos being small compared to a game, we do not manage residency

CPU/GPU INTERACTION



The CPU and GPU work in parallel and sometimes need to be synchronized.



The goal is to keep both busy for as long as possible and minimize synchronizations.



Synchronizations are undesirable because it means one processing unit is idle while waiting on the other to finish some work (it ruins the parallelism)

The Command Queue and Command Lists

The GPU has a command queue. The CPU submits commands to the queue through the Direct3D API using command lists.

The commands submitted to the command queue are not immediately executed by the GPU. They sit in the queue until the GPU is ready to process them, as the GPU is likely busy processing previously inserted commands.

If the command queue gets empty, the GPU will idle because it does not have any work to do; on the other hand, if the command queue gets too full, the CPU will at some point have to idle while the GPU catches up.

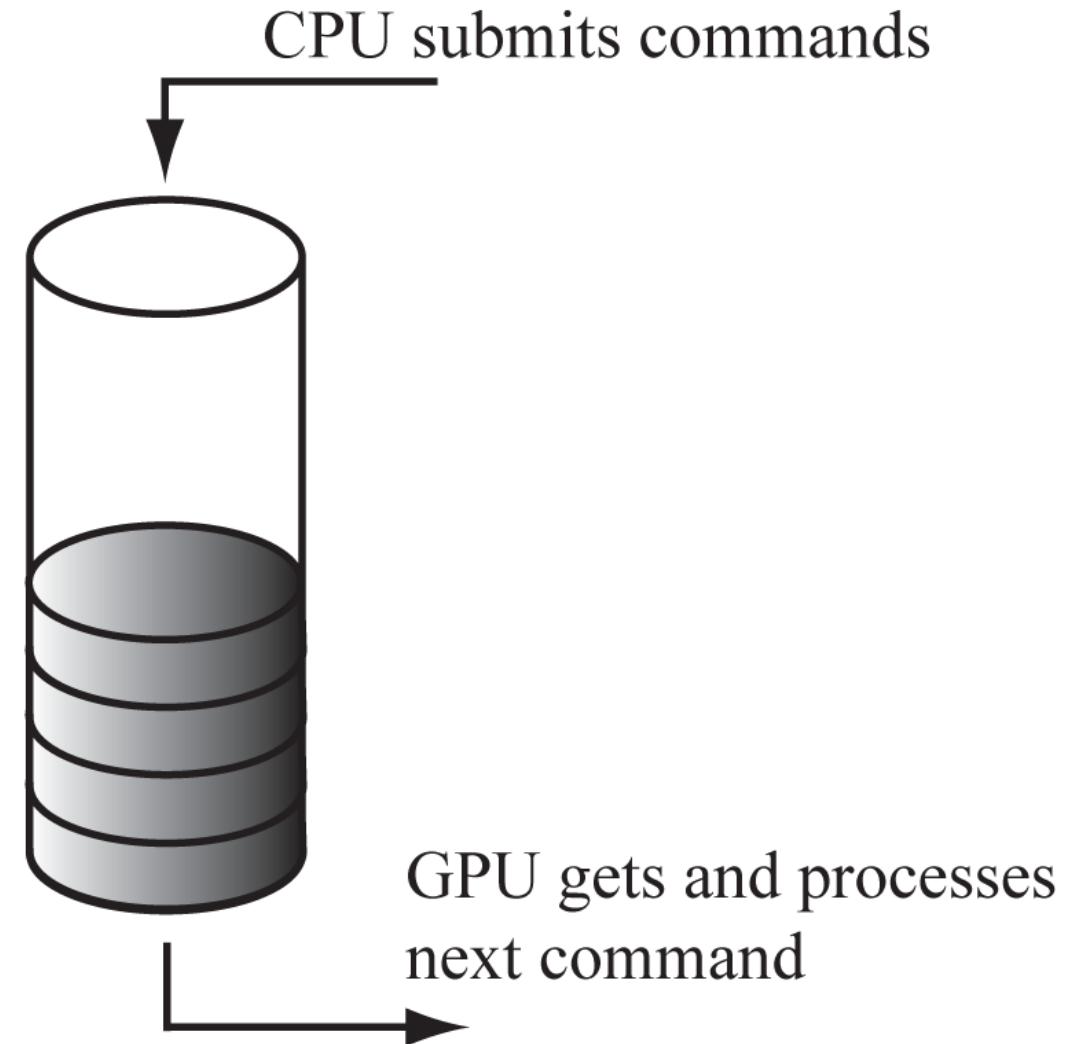
For high performance applications like games, the goal is to keep both CPU and GPU busy to take full advantage of the hardware resources available.

In Direct3D 12, the command queue is represented by the `ID3D12CommandQueue` interface.

It is created by filling out a `D3D12_COMMAND_QUEUE_DESC` structure describing the queue and then calling `ID3D12Device::CreateCommandQueue`.

```
Microsoft::WRL::ComPtr<ID3D12CommandQueue> mCommandQueue;  
D3D12_COMMAND_QUEUE_DESC queueDesc = {};  
queueDesc.Type = D3D12_COMMAND_LIST_TYPE_DIRECT; //Direct, Bundle, Video, Copy  
queueDesc.Flags = D3D12_COMMAND_QUEUE_FLAG_NONE;  
ThrowIfFailed(md3dDevice->CreateCommandQueue(&queueDesc, IID_PPV_ARGS(&mCommandQueue)));
```

We call this function from `D3DApp::CreateCommandObjects()`



ExecuteCommandLists

One of the primary methods of this interface is the ExecuteCommandLists method which adds the commands in the command lists to the queue:

```
void ID3D12CommandQueue::ExecuteCommandLists(  
    // Number of commands lists in the array  
    UINT Count,  
    // Pointer to the first element in an array of command lists  
    ID3D12CommandList* const* ppCommandLists);
```

The ID3D12GraphicsCommandList interface has numerous methods for adding commands to the command list. For example, the following code adds commands that set the viewport, clear the render target view, and issue a draw call:

```
// mCommandList pointer to ID3D12CommandList  
mCommandList->RSSetViewports(1, &mScreenViewport);  
mCommandList->ClearRenderTargetView(mBackBufferView, Colors::LightSteelBlue, 0, nullptr);  
mCommandList->DrawIndexedInstanced(36, 1, 0, 0, 0);
```

```
// Done recording commands.  
ThrowIfFailed(mCommandList->Close());
```

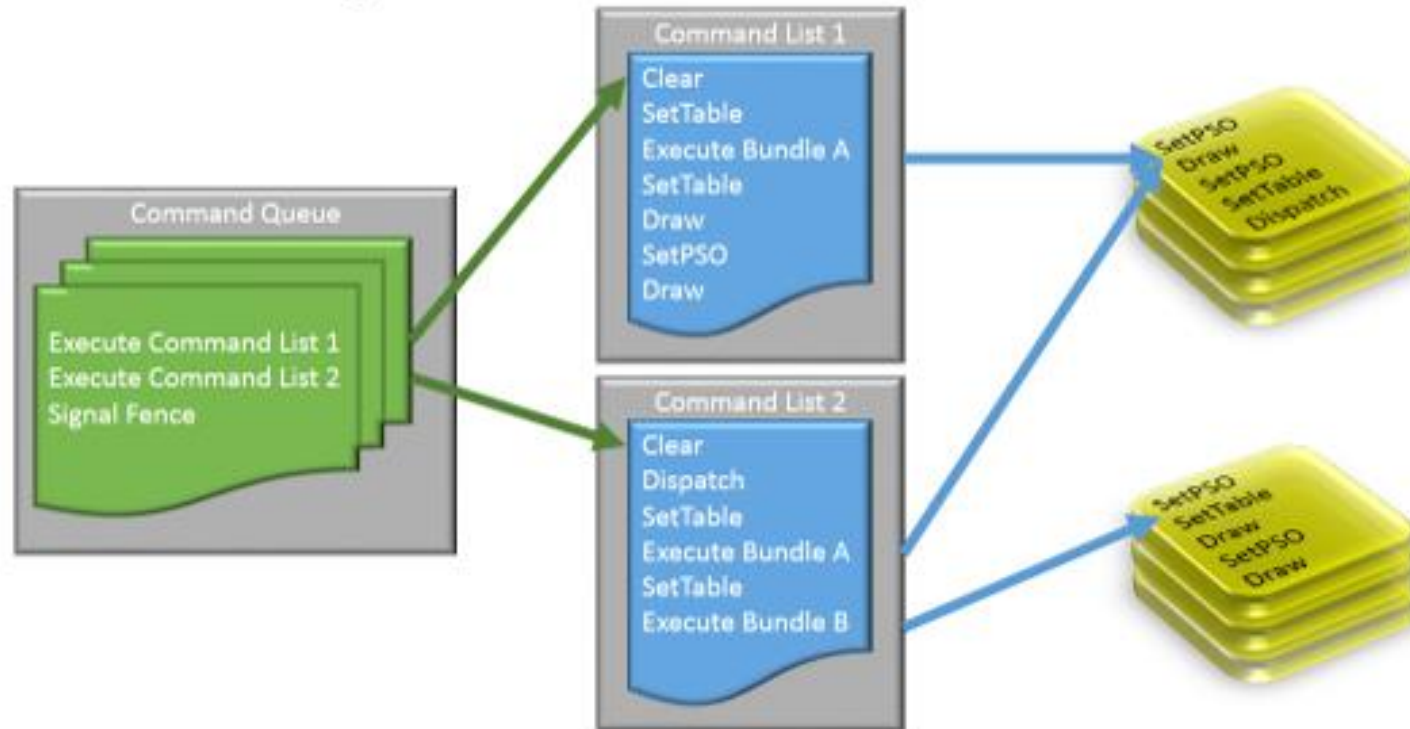
```
// Add the command list to the queue for execution.
```

```
ID3D12CommandList* cmdsLists[] = { mCommandList.Get() };
```

```
mCommandQueue->ExecuteCommandLists(_countof(cmdsLists), cmdsLists);
```

Commands

Command Queue



ID3D12CommandAllocator

As commands are recorded to the command list, they will actually be stored in the associated command allocator. When a command list is executed via `ID3D12CommandQueue::ExecuteCommandLists`, the command queue will **reference** the commands in the allocator. A command allocator is created from the `ID3D12Device`:


```
HRESULT ID3D12Device::CreateCommandAllocator(D3D12_COMMAND_LIST_TYPE type, REFIID riid, void** ppCommandAllocator);
```

`type`: The type of command lists that can be associated with this allocator.

1. `D3D12_COMMAND_LIST_TYPE_DIRECT`: Stores a list of commands to directly be executed by the GPU (the type of command list we have been describing so far).
2. `D3D12_COMMAND_LIST_TYPE_BUNDLE`: Specifies the command list represents a *bundle*.

`riid`: The COM ID of the `ID3D12CommandAllocator` interface we want to create.

`ppCommandAllocator`: Outputs a pointer to the created command allocator.



ID3D12Device::CreateCommandList

Command lists are also created from the ID3D12Device. **ID3D12Device** represents a virtual adapter; it is used to create command allocators, command lists, command queues, fences, resources, pipeline state objects, heaps, root signatures, samplers, and many resource views.

HRESULT ID3D12Device::CreateCommandList(

UINT nodeMask, → Set to 0 for single GPU system

D3D12_COMMAND_LIST_TYPE type, → D3D12_COMMAND_LIST_TYPE_DIRECT: a command buffer that the GPU can execute (no GPU State).

ID3D12CommandAllocator* pCommandAllocator, → The device creates command lists from the command allocator.

ID3D12PipelineState* pInitialState, → Pointer to Pipeline State Object (will discuss later!)

REFIID riid, → The globally unique identifier (**GUID**) for the command list interface

void** ppCommandList); → A pointer to a memory block that receives a pointer to the [ID3D12CommandList](#)

- You can create multiple command lists associated with the same allocator, but you cannot record at the same time.
- All command lists must be closed except the one whose commands we are going to record.
- All commands from a given command list will be added to the allocator contiguously.

ID3D12CommandList::Reset

After we have called `ID3D12CommandQueue::ExecuteCommandList(C)`, it is safe to reuse the internal memory of `C` to record a new set of commands by calling the `ID3D12CommandList::Reset` method. The parameters of this method are the same as the matching parameters in `ID3D12Device::CreateCommandList`.

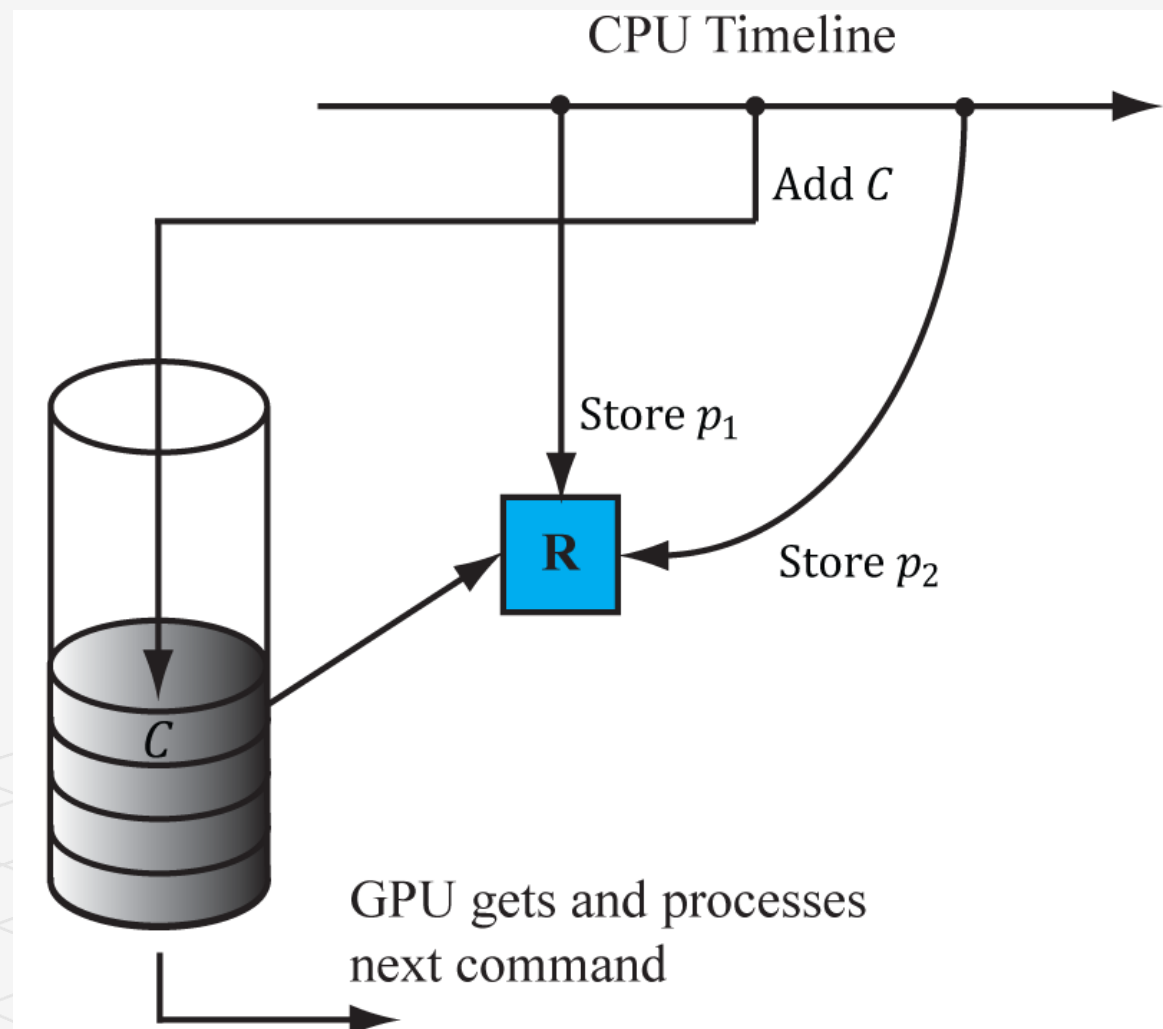
```
HRESULT ID3D12CommandList::Reset(  
  
ID3D12CommandAllocator* pAllocator,  
  
ID3D12PipelineState* pInitialState);
```

After we have submitted the rendering commands for a complete frame to the GPU, we would like to reuse the memory in the command allocator for the next frame. The `ID3D12CommandAllocator::Reset` method may be used for this:

```
void InitDirect3DApp::Draw(const GameTimer& gt)  
  
{// Reuse the memory associated with command recording. We can only reset when the associated command lists have finished  
execution on the GPU.  
  
ThrowIfFailed(mDirectCmdListAlloc->Reset());  
  
// A command list can be reset after it has been added to the command queue via ExecuteCommandList. Reusing the command list  
//reuses memory. Note that resetting the command list does not affect the commands in the command queue because the associated command allocator  
//still has the commands in memory that the command queue references.  
  
ThrowIfFailed(mCommandList->Reset(mDirectCmdListAlloc.Get(), nullptr));
```

CPU/GPU Synchronization

- Suppose we have some resource R that stores the position of some geometry we wish to draw.
- Suppose the CPU updates the data of R to store position p_1 and then adds a drawing command C that references R to the command queue with the intent of drawing the geometry at position p_1 .
- Adding commands to the command queue does not block the CPU, so the CPU continues on.
- It would be an error for the CPU to continue on and overwrite the data of R to store a new position p_2 before the GPU executed the draw command C .
- Because the command queue may be referencing data in an allocator, **a command allocator must not be reset until we are sure the GPU has finished executing all the commands in the allocator.**
- How to implement this?



Flushing the command queue

One solution is to force the CPU to wait until the GPU has finished processing all the commands in the queue up to a specified fence point.

We call this *flushing the command queue*.

We can do this using a *fence*. A fence is represented by the ID3D12Fence interface and is used to synchronize the GPU and CPU. A fence object can be created with the following method:

`HRESULT ID3D12Device::CreateFence(`

`UINT64 InitialValue,` → The initial value for the fence

`D3D12_FENCE_FLAGS Flags,` → like D3D12_FENCE_FLAG_NONE. Or a combination of [D3D12_FENCE_FLAGS](#)-typed, if it's shared by another GPU

`REFIID riid,` → The globally unique identifier (**GUID**) for the fence interface

`void** ppFence);` → A pointer to a memory block that receives a pointer to the [ID3D12Fence](#) interface

Example:

```
ThrowIfFailed(md3dDevice->CreateFence(0,D3D12_FENCE_FLAG_NONE, IID_PPV_ARGS(&mFence)));
```

D3DApp::FlushCommandQueue

A fence object maintains a UINT64 value, which is just an integer to identify a fence point in time.

We start at value zero and every time we need to mark a new fence point, we just increment the integer. Now, the following code/comments show how we can use a fence to flush the command queue.

```
UINT64 mCurrentFence = 0;
void D3DApp::FlushCommandQueue()
{
    // Advance the fence value to mark commands up to this fence point.
    mCurrentFence++;

    // Add an instruction to the command queue to set a new fence point. Because we are on the GPU timeline, the new fencepoint
    // won't be set until the GPU finishes processing all the commands prior to this Signal().

    ThrowIfFailed(mCommandQueue->Signal(mFence.Get(), mCurrentFence));

    // Wait until the GPU has completed commands up to this fence point.
    if (mFence->GetCompletedValue() < mCurrentFence)
    {
        HANDLE eventHandle = CreateEventEx(nullptr, false, false, EVENT_ALL_ACCESS);

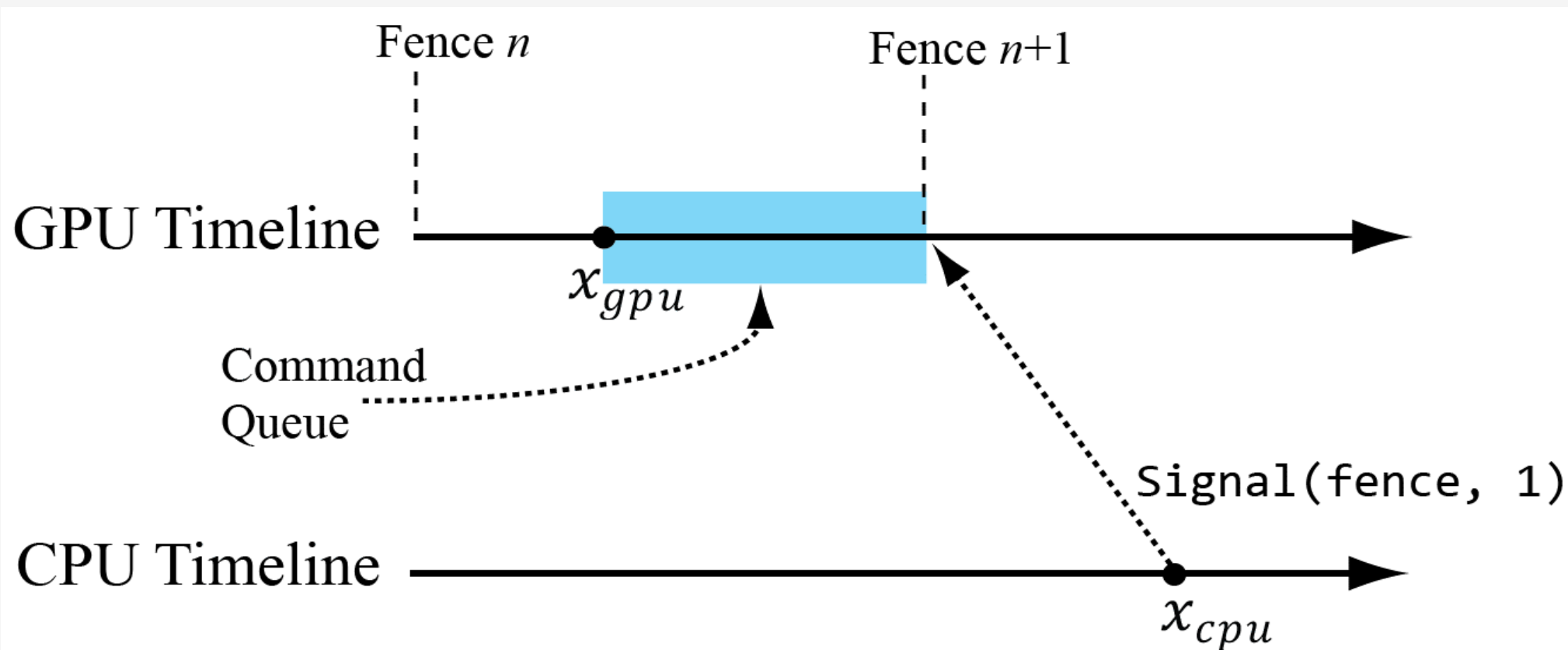
        // Fire event when GPU hits current fence.
        ThrowIfFailed(mFence -> SetEventOnCompletion(mCurrentFence, eventHandle));

        // Wait until the GPU hits current fence event is fired.
        WaitForSingleObject(eventHandle, INFINITE);

        CloseHandle(eventHandle);
    }
}
```

Any thread of the calling process can specify the event-object handle in a call to one of the [wait functions](#). The single-object wait functions return when the state of the specified object is signaled.

Sync CPU/GPU using Fence



The GPU has processed commands up to x_{gpu} and the CPU has just called the `ID3D12CommandQueue::Signal(fence, $n+1$)` method. This essentially adds an instruction to the end of the queue to change the fence value to $n+1$. However, `mFence->GetCompletedValue()` will continue to return n until the GPU processes all the commands in the queue that were added prior to the `Signal(fence, $n+1$)` instruction.

Resource Transition

To implement common rendering effects, it is common for the GPU to write to a resource *R* in one step, and then, in a later step, read from the resource *R*.

However, it would be a *resource hazard* to read from a resource if the GPU has not finished writing to it or not started writing at all.

To solve this problem, **Direct3D associates a state to resources.**

A resource transition is specified by setting an array of *transition resource barriers* on the command list.

It is an array in case you want to transition multiple resources with one API call.

In code, a resource barrier is represented by the D3D12_RESOURCE_BARRIER structure.

```
struct CD3DX12_RESOURCE_BARRIER : public D3D12_RESOURCE_BARRIER
{
    CD3DX12_RESOURCE_BARRIER()
    {}
    explicit CD3DX12_RESOURCE_BARRIER(const D3D12_RESOURCE_BARRIER &o) :
        D3D12_RESOURCE_BARRIER(o)
    {}
    static inline CD3DX12_RESOURCE_BARRIER Transition(
        _In_ ID3D12Resource* pResource,
        D3D12_RESOURCE_STATES stateBefore,
        D3D12_RESOURCE_STATES stateAfter,
        UINT subresource = D3D12_RESOURCE_BARRIER_ALL_SUBRESOURCES,
        D3D12_RESOURCE_BARRIER_FLAGS flags = D3D12_RESOURCE_BARRIER_FLAG_NONE)
    {
        CD3DX12_RESOURCE_BARRIER result;
        ZeroMemory(&result, sizeof(result));
        D3D12_RESOURCE_BARRIER &barrier = result;
        result.Type = D3D12_RESOURCE_BARRIER_TYPE_TRANSITION;
        result.Flags = flags;
        barrier.Transition.pResource = pResource;
        barrier.Transition.StateBefore = stateBefore;
        barrier.Transition.StateAfter = stateAfter;
        barrier.Transition.Subresource = subresource;
        return result;
    }
}
```

mCommandList->ResourceBarrier

Resources are in a default state when they are created, and it is up to the application to tell Direct3D any state transitions.

ResourceBarrier method notifies the driver that it needs to synchronize multiple accesses to resources.

For example: If we are writing to a resource, say a texture, we will set the texture state to a render target state;

When we need to read the texture, we will change its state to a shader resource state.

This code transitions a texture representing the image we are displaying on screen from a presentation state to a render target state.

Observe that the resource barrier is added to the command list. You can think of the resource barrier transition as a command itself instructing the GPU that the state of a resource is being transitioned, so that it can take the necessary steps to prevent a resource hazard when executing subsequent commands.

```
void ResourceBarrier( UINT NumBarriers, const D3D12_RESOURCE_BARRIER *pBarriers );
```

```
// Indicate a state transition on the resource usage.
```

```
mCommandList->ResourceBarrier(1,  
&CD3DX12_RESOURCE_BARRIER::Transition(CurrentBackBuffer(),  
  
D3D12_RESOURCE_STATE_PRESENT, D3D12_RESOURCE_STATE_RENDER_TARGET));
```

```
.....
```

```
mCommandList->DrawIndexedInstanced( mBoxGeo->DrawArgs["box"].IndexCount, 1, 0, 0,  
0);
```

```
// Indicate a state transition on the resource usage.
```

```
mCommandList->ResourceBarrier(1,  
&CD3DX12_RESOURCE_BARRIER::Transition(CurrentBackBuffer(),  
  
D3D12_RESOURCE_STATE_RENDER_TARGET, D3D12_RESOURCE_STATE_PRESENT));
```


Multithreading with Commands

Direct3D 12 was designed for efficient multithreading.

The command list design is one way Direct3D takes advantage of multithreading.

For large scenes with lots of objects, building the command list to draw the entire scene can take CPU time.

The idea is to build command lists in parallel; for example: spawn four threads, each responsible for building a command list to draw 25% of the scene objects.

A few things to note about command list multithreading:

1. Command lists are not free-threaded; that is, multiple threads may not share the same command list and call its methods concurrently. So generally, each thread will get its own command list.
2. Command allocators are not free-threaded; that is, multiple threads may not share the same command allocator and call its methods concurrently. So generally, each thread will get its own command allocator.
3. The command queue is free-threaded, so multiple threads can access the command queue and call its methods concurrently. In particular, each thread can submit their generated command list to the thread queue concurrently.

For simplicity, we will not use multithreading in our demos.

INITIALIZING DIRECT3D

1. Create the ID3D12Device using the D3D12CreateDevice function.
2. Create an ID3D12Fence object and query descriptor sizes.
3. Check 4X MSAA quality level support.
4. Create the command queue, command list allocator, and main command list.
5. Create the swap chain.
6. Create the descriptor heaps the application requires.
7. Resize the back buffer and create a render target view to the back buffer.
8. Create the depth/stencil buffer and its associated depth/stencil view.
9. Set the viewport and scissor rectangles.

Microsoft::WRL::ComPtr



WRL(Windows Runtime Library) defines the fundamental types that make up the Windows Runtime C++ Template Library.



ComPtr class creates a smart pointer type that represents the interface specified by the template parameter. ComPtr automatically maintains a reference count for the underlying interface pointer and releases the interface when the reference count goes to zero.



```
template <typename T> class ComPtr;
```



For example: IDXGIFactory4 enables creating Microsoft DirectX Graphics Infrastructure (DXGI) objects.



```
Microsoft::WRL::ComPtr<IDXGIFactory4> mdxgiFactory;
```

D3DApp

```
class D3DApp
{
protected:
    D3DApp(HINSTANCE hInstance);
...
public:
    static D3DApp* GetApp();
...
void FlushCommandQueue();
protected:
    static D3DApp* mApp
    GameTimer mTimer;
    Microsoft::WRL::ComPtr<IDXGIFactory4> mdxgiFactory;
    Microsoft::WRL::ComPtr<IDXGISwapChain> mSwapChain;
    Microsoft::WRL::ComPtr<ID3D12Device> md3dDevice;
    Microsoft::WRL::ComPtr<ID3D12Fence> mFence;
    UINT64 mCurrentFence = 0;
    Microsoft::WRL::ComPtr<ID3D12CommandQueue> mCommandQueue;
    Microsoft::WRL::ComPtr<ID3D12CommandAllocator> mDirectCmdListAlloc;
    Microsoft::WRL::ComPtr<ID3D12GraphicsCommandList> mCommandList;
static const int SwapChainBufferCount = 2;
int mCurrBackBuffer = 0;
    Microsoft::WRL::ComPtr<ID3D12Resource> mSwapChainBuffer[SwapChainBufferCount];
    Microsoft::WRL::ComPtr<ID3D12Resource> mDepthStencilBuffer;
    Microsoft::WRL::ComPtr<ID3D12DescriptorHeap> mRtvHeap;
    Microsoft::WRL::ComPtr<ID3D12DescriptorHeap> mDsvHeap;
    D3D12_VIEWPORT mScreenViewport;
    D3D12_RECT mScissorRect;
```

.....

Create the device

Initializing Direct3D begins by creating the Direct3D 12 device (ID3D12Device).

The device represents a display adapter.

Usually, the display adapter is a physical piece of 3D hardware (e.g., graphics card);

However, a system can also have a software display adapter that emulates 3D hardware functionality (e.g., the WARP adapter – WARP stands for Windows Advanced Rasterization Platform- which is a high-performance software rasterizer if you have old driver!).

The Direct3D12 device is used to check feature support, and create all other Direct3D interface objects like resources, views, and command lists. The device can be created with the following function (d3d12.h):

```
HRESULT WINAPI D3D12CreateDevice(
    IUnknown* pAdapter, // you can choose your adapter if your system has multiple
                        //graphics card - system chooses the default one
    D3D_FEATURE_LEVEL MinimumFeatureLevel, // although you are working D3D12, you can
                        //target direct3D 9 as your minimum feature!
    REFIID riid, // The COM ID of the ID3D12Device that we want to create
    void** ppDevice); // Returns the created device

bool D3DApp::InitDirect3D()
{
    ThrowIfFailed(CreateDXGIFactory1(IID_PPV_ARGS(&mdxgiFactory)));

    // Try to create hardware device.
    HRESULT hardwareResult = D3D12CreateDevice(
        nullptr, // default adapter
        D3D_FEATURE_LEVEL_12_0, IID_PPV_ARGS(&md3dDevice));

    // Fallback to WARP device.
    if(FAILED(hardwareResult))
    {
        ComPtr<IDXGIAdapter> pWarpAdapter;
        ThrowIfFailed(mdxgiFactory->EnumWarpAdapter(IID_PPV_ARGS(&pWarpAdapter)));

        ThrowIfFailed(D3D12CreateDevice( pWarpAdapter.Get(), D3D_FEATURE_LEVEL_12_0,
            IID_PPV_ARGS(&md3dDevice)));
    }
}
```

IID_PPV_ARGS

Every interface is derived from IUnknown.

IUnknown::QueryInterface method: A helper function template that infers an interface identifier, and calls [QueryInterface\(REFIID,void\)](#).

You must use an interface identifier (IID) when making calls to the **QueryInterface** method. An IID is a globally unique identifier (GUID) value.

IID_PPV_ARGS is used to retrieve an interface pointer, supplying the IID value of the requested interface automatically based on the type of the interface pointer used.

This macro computes the IID based on the type of interface pointer, which prevents coding errors in which the IID and interface pointer type do not match.

```
void IID_PPV_ARGS( ppType );
```

A common syntax in methods that retrieve an interface pointer (most notably [QueryInterface](#) and [CoCreateInstance](#)) includes two parameters:

An [in] parameter, normally of type **REFIID**, to specify the IID of the interface to retrieve.

An [out] parameter, normally of type **void****, to receive the interface pointer.

The following example shows the use of **IID_PPV_ARGS** to create the memory property store object using [IPropertyStore](#).

```
IPropertyStore *pPropertyStore; CoCreateInstance(CLSID_PropertyStore, NULL, CLSCTX_INPROC_SERVER, IID_PPV_ARGS(&pPropertyStore));
```

Descriptor size

Descriptor sizes can vary across GPUs so we need to query this information. We cache the descriptor sizes so that it is available when we need it for various descriptor types:

```
mRtvDescriptorSize = md3dDevice->GetDescriptorHandleIncrementSize(D3D12_DESCRIPTOR_HEAP_TYPE_RTV);
```

```
mDsvDescriptorSize = md3dDevice->GetDescriptorHandleIncrementSize(D3D12_DESCRIPTOR_HEAP_TYPE_DSV);
```

```
mCbvSrvUavDescriptorSize = md3dDevice->GetDescriptorHandleIncrementSize(D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV);
```

ID3D12Device::GetDescriptorHandleIncrementSize method

A descriptor is a relatively small block of data that fully describes an object to the GPU, in a GPU specific opaque format.

Descriptors are of varying size, typically 32 to 64 bytes for an SRV, UAV or CBV (depending on the GPU hardware)

A descriptor handle is the unique address of the descriptor. It is similar to a pointer, but is opaque as its implementation is hardware specific.

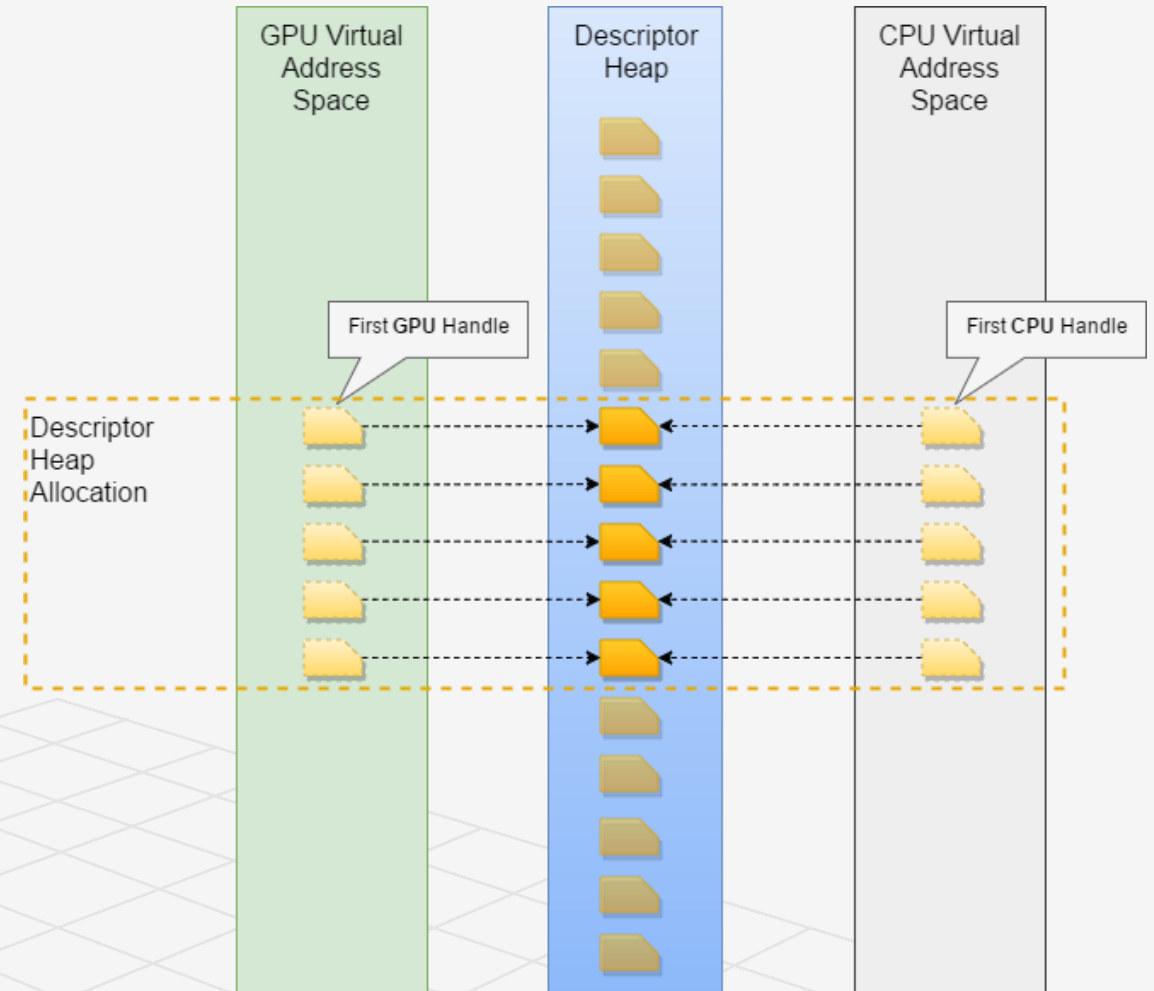
GetDescriptorHandleIncrementSize: Gets the size of the handle increment for the given type of descriptor heap. This value is typically used to increment a handle into a descriptor array by the correct amount.

CPU handles are for immediate use, such as copying where both the source and destination need to be identified.

GPU handles are not for immediate use, they identify locations from a command list, for use at GPU execution time.

```
UINT GetDescriptorHandleIncrementSize( D3D12_DESCRIPTOR_HEAP_TYPE  
DescriptorHeapType );
```

It is safe to offset a starting location with a number of increments, to copy handles, and to pass handles into API calls.



Where do we use the RTV descriptor size?

After we create the heaps, we need to be able to access the descriptors they store.

Our application references descriptors through handles.

```
D3D12_CPU_DESCRIPTOR_HANDLE
D3DApp::CurrentBackBufferView()const
{
    return CD3DX12_CPU_DESCRIPTOR_HANDLE(
        mRtvHeap->GetCPUDescriptorHandleForHeapStart(),
        mCurrBackBuffer,
        mRtvDescriptorSize);
}
```

```
CD3DX12_CPU_DESCRIPTOR_HANDLE rtvHeapHandle(mRtvHeap->GetCPUDescriptorHandleForHeapStart());
```

```
for (UINT i = 0; i < SwapChainBufferCount; i++)
```

```
{
```

```
    ThrowIfFailed(mSwapChain->GetBuffer(i,
        IID_PPV_ARGS(&mSwapChainBuffer[i])));
```

```
    md3dDevice->CreateRenderTargetView(
        mSwapChainBuffer[i].Get(), nullptr, rtvHeapHandle);
```

```
    rtvHeapHandle.Offset(1, mRtvDescriptorSize);
```

```
}
```

Check 4X MSAA Quality Support

“dxgicommon.h”

```
typedef struct DXGI_SAMPLE_DESC
```

```
{  
    UINT Count;  
    UINT Quality;  
} DXGI_SAMPLE_DESC;
```

The Count member specifies the number of samples to take per pixel,

The Quality member is used to specify the desired quality level (what “quality level” means can vary across hardware manufacturers). Higher sample counts or higher quality is more expensive to render, so a tradeoff between quality and speed must be made.

The range of quality levels depends on the texture format and the number of samples to take per pixel.

We can query the number of quality levels for a given texture format and sample count using the `ID3D12Device::CheckFeatureSupport`

```
typedef struct D3D12_FEATURE_DATA_MULTISAMPLE_QUALITY_LEVELS  
{  
    _In_   DXGI_FORMAT Format;  
    _In_   UINT SampleCount;  
    _In_   D3D12_MULTISAMPLE_QUALITY_LEVEL_FLAGS Flags;  
    _Out_  UINT NumQualityLevels;  
} D3D12_FEATURE_DATA_MULTISAMPLE_QUALITY_LEVELS;
```

```
D3D12_FEATURE_DATA_MULTISAMPLE_QUALITY_LEVELS msQualityLevels;  
msQualityLevels.Format = mBackBufferFormat;  
msQualityLevels.SampleCount = 4;  
msQualityLevels.Flags =  
D3D12_MULTISAMPLE_QUALITY_LEVELS_FLAG_NONE;  
msQualityLevels.NumQualityLevels = 0;  
ThrowIfFailed(md3dDevice->CheckFeatureSupport(  
D3D12_FEATURE_MULTISAMPLE_QUALITY_LEVELS,  
&msQualityLevels,  
sizeof(msQualityLevels)));
```

Create Command Queue and Command List

```
Microsoft::WRL::ComPtr<ID3D12CommandQueue> mCommandQueue;
Microsoft::WRL::ComPtr<ID3D12CommandAllocator> mDirectCmdListAlloc;
Microsoft::WRL::ComPtr<ID3D12GraphicsCommandList> mCommandList;

void D3DApp::CreateCommandObjects()
{
    D3D12_COMMAND_QUEUE_DESC queueDesc = {};
    queueDesc.Type = D3D12_COMMAND_LIST_TYPE_DIRECT;
    queueDesc.Flags = D3D12_COMMAND_QUEUE_FLAG_NONE;
    ThrowIfFailed(md3dDevice->CreateCommandQueue(&queueDesc, IID_PPV_ARGS(&mCommandQueue)));

    ThrowIfFailed(md3dDevice->CreateCommandAllocator(
        D3D12_COMMAND_LIST_TYPE_DIRECT,
        IID_PPV_ARGS(mDirectCmdListAlloc.GetAddressOf())));

    ThrowIfFailed(md3dDevice->CreateCommandList(
        0,
        D3D12_COMMAND_LIST_TYPE_DIRECT,
        mDirectCmdListAlloc.Get(), // Associated command allocator
        nullptr, // Initial PipelineStateObject
        IID_PPV_ARGS(mCommandList.GetAddressOf())));

    // Start off in a closed state. This is because the first time we refer
    // to the command list we will Reset it, and it needs to be closed before
    // calling Reset.
    mCommandList->Close();
}
```

Describe the Swap Chain

DXGI_SWAP_CHAIN_DESC structure describes the characteristics of the swap chain we are going to create.

```
typedef struct DXGI_SWAP_CHAIN_DESC
```

```
{
```

`DXGI_MODE_DESC` BufferDesc; → The main properties are the width and height, and pixel format;

`DXGI_SAMPLE_DESC` SampleDesc; → The number of multisamples

`DXGI_USAGE` BufferUsage; → we are going to be rendering to the back buffer?

`UINT` BufferCount; → The number of buffers to use in the swap chain: 2 for double buffering

`HWND` OutputWindow; → A handle to the window we are rendering into

`BOOL` Windowed; → Specify true to run in windowed mode or false for full-screen mode.

`DXGI_SWAP_EFFECT` SwapEffect; → Use this flag to specify the flip presentation model and/or discard the back buffer

`UINT` Flags; → Optional flags – for instance choose a display mode that best matches the current application window dimensions

```
} DXGI_SWAP_CHAIN_DESC;
```

Create the Swap Chain

After we have described our swap chain, we can create it with the `IDXGIFactory::CreateSwapChain` method:

```
void D3DApp::CreateSwapChain()
{
    // Release the previous swapchain we will be recreating.
    mSwapChain.Reset();

    DXGI_SWAP_CHAIN_DESC sd;
    sd.BufferDesc.Width = mClientWidth;
    sd.BufferDesc.Height = mClientHeight;
    sd.BufferDesc.RefreshRate.Numerator = 60;
    sd.BufferDesc.RefreshRate.Denominator = 1;
    sd.BufferDesc.Format = mBackBufferFormat;
    sd.BufferDesc.ScanlineOrdering = DXGI_MODE_SCANLINE_ORDER_UNSPECIFIED;
    sd.BufferDesc.Scoring = DXGI_MODE_SCALING_UNSPECIFIED;
    sd.SampleDesc.Count = m4xMsaaState ? 4 : 1;
    sd.SampleDesc.Quality = m4xMsaaState ? (m4xMsaaQuality - 1) : 0;
    sd.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;
    sd.BufferCount = SwapChainBufferCount;
    sd.OutputWindow = mhMainWnd;
    sd.Windowed = true;
    sd.SwapEffect = DXGI_SWAP_EFFECT_FLIP_DISCARD;
    sd.Flags = DXGI_SWAP_CHAIN_FLAG_ALLOW_MODE_SWITCH;
    ThrowIfFailed(mdxgiFactory->CreateSwapChain(
        mCommandQueue.Get(),
        &sd,
        mSwapChain.GetAddressOf()));
}
```

Create the Descriptor Heaps

We need to create the descriptor heaps to store the descriptors/views.

A heap is created with the `ID3D12Device::CreateDescriptorHeap` method.

We need `SwapChainBufferCount` many render target views (RTVs) to describe the buffer resources in the swap chain we will render into, and one depth/stencil view (DSV) to describe the depth/stencil buffer resource for depth testing.

```
void D3DApp::CreateRtvAndDsvDescriptorHeaps()
{
    D3D12_DESCRIPTOR_HEAP_DESC rtvHeapDesc;
    rtvHeapDesc.NumDescriptors = SwapChainBufferCount;
    rtvHeapDesc.Type = D3D12_DESCRIPTOR_HEAP_TYPE_RTV;
    rtvHeapDesc.Flags = D3D12_DESCRIPTOR_HEAP_FLAG_NONE;
    rtvHeapDesc.NodeMask = 0;
    ThrowIfFailed(md3dDevice->CreateDescriptorHeap(&rtvHeapDesc,
        IID_PPV_ARGS(mRtvHeap.GetAddressOf())));

    D3D12_DESCRIPTOR_HEAP_DESC dsvHeapDesc;
    dsvHeapDesc.NumDescriptors = 1;
    dsvHeapDesc.Type = D3D12_DESCRIPTOR_HEAP_TYPE_DSV;
    dsvHeapDesc.Flags = D3D12_DESCRIPTOR_HEAP_FLAG_NONE;
    dsvHeapDesc.NodeMask = 0;
    ThrowIfFailed(md3dDevice->CreateDescriptorHeap(&dsvHeapDesc,
        IID_PPV_ARGS(mDsvHeap.GetAddressOf())));
}
```

ID3D12DescriptorHeap::GetCPUDescriptorHandleForHeapStart

After we create the heaps, we need to be able to access the descriptors they store.

Our application references descriptors through handles.

A handle to the first descriptor in a heap is obtained with the

`ID3D12DescriptorHeap::GetCPUDescriptorHandleForHeapStart` method.

We now see an example of where the descriptor size is needed. In order to offset to the current back buffer RTV descriptor, we need to know the RTV descriptor byte size.

The following functions get the current back buffer RTV and DSV, respectively:

```
ID3D12Resource* D3DApp::CurrentBackBuffer()const
{
    return mSwapChainBuffer[mCurrBackBuffer].Get();
}
```

```
D3D12_CPU_DESCRIPTOR_HANDLE D3DApp::CurrentBackBufferView()const
{
    // CD3DX12 constructor to offset to the RTV of the current back buffer.

    return CD3DX12_CPU_DESCRIPTOR_HANDLE(
        mRtvHeap->GetCPUDescriptorHandleForHeapStart(),
        mCurrBackBuffer, // index to offset
        mRtvDescriptorSize); // byte size of descriptor
}
```

```
D3D12_CPU_DESCRIPTOR_HANDLE D3DApp::DepthStencilView()const
{
    return mDsvHeap->GetCPUDescriptorHandleForHeapStart();
}
```

Create the Render Target View

We do not bind a resource to a pipeline stage directly; instead, we must create a resource view (descriptor) to the resource and bind the view to the pipeline stage.

In order to bind the back buffer to the output merger stage of the pipeline (so Direct3D can render onto it), we need to create a render target view to the back buffer.

The first step is to get the buffer resources which are stored in the swap chain:

```
HRESULT IDXGISwapChain::GetBuffer( UINT Buffer, REFIID riid, void** ppSurface);
```

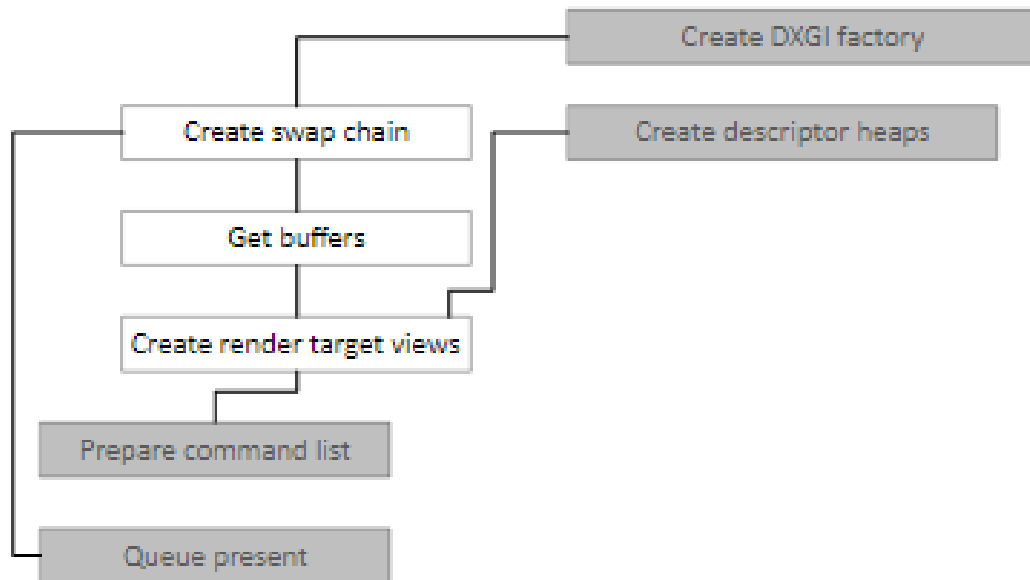
1. Buffer: An index identifying the particular back buffer we want to get (in case there is more than one).
2. riid: The COM ID of the ID3D12Resource interface we want to obtain a pointer to.
3. ppSurface: Returns a pointer to an ID3D12Resource that represents the back buffer.

To create the render target view, we use the `ID3D12Device::CreateRenderTargetView` method:

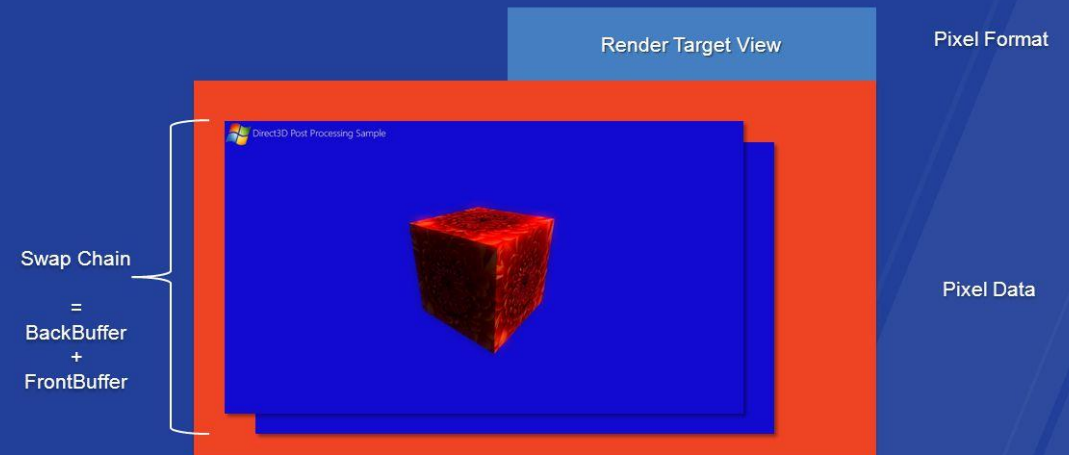
```
void ID3D12Device::CreateRenderTargetView(ID3D12Resource* pResource, const D3D12_RENDER_TARGET_VIEW_DESC* pDesc, D3D12_CPU_DESCRIPTOR_HANDLE DestDescriptor);
```

```
CD3DX12_CPU_DESCRIPTOR_HANDLE rtvHeapHandle(mRtvHeap->GetCPUDescriptorHandleForHeapStart());
for (UINT i = 0; i < SwapChainBufferCount; i++)
{
    ThrowIfFailed(mSwapChain->GetBuffer(i, IID_PPV_ARGS(&mSwapChainBuffer[i])));
    md3dDevice->CreateRenderTargetView(mSwapChainBuffer[i].Get(), nullptr, rtvHeapHandle);
    rtvHeapHandle.Offset(1, mRtvDescriptorSize);
}
```


Render Target View



RenderTarget object



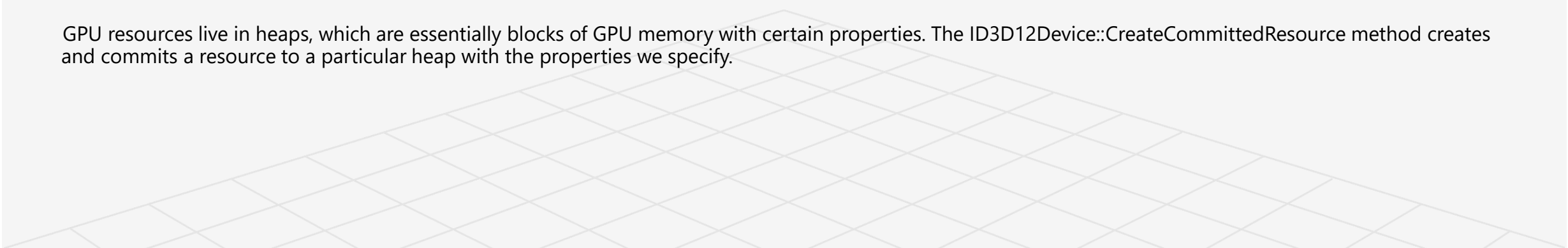
Create the Depth/Stencil Buffer

The depth buffer is just a 2D texture that stores the depth information of the nearest visible objects (and stencil information if using stenciling).

A texture is a kind of GPU resource, so we create one by filling out a `D3D12_RESOURCE_DESC` structure describing the texture resource, and then calling the `ID3D12Device::CreateCommittedResource` method.

```
typedef struct D3D12_RESOURCE_DESC
{
    D3D12_RESOURCE_DIMENSION Dimension;
    UINT64 Alignment;
    UINT64 Width;
    UINT Height;
    UINT16 DepthOrArraySize;
    UINT16 MipLevels;
    DXGI_FORMAT Format;
    DXGI_SAMPLE_DESC SampleDesc;
    D3D12_TEXTURE_LAYOUT Layout;
    D3D12_RESOURCE_FLAGS Flags;
} D3D12_RESOURCE_DESC;
```

GPU resources live in heaps, which are essentially blocks of GPU memory with certain properties. The `ID3D12Device::CreateCommittedResource` method creates and commits a resource to a particular heap with the properties we specify.

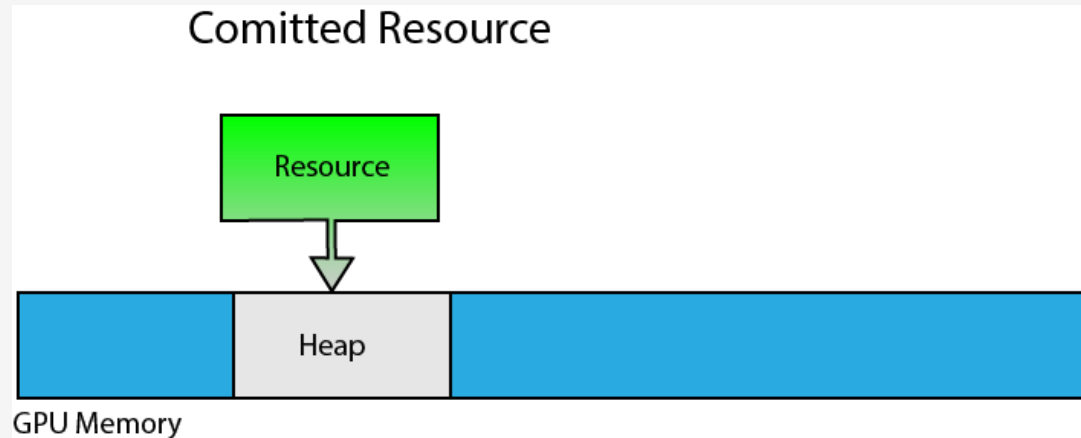


ID3D12Device::CreateCommittedResource

```
HRESULT ID3D12Device::CreateCommittedResource(  
    const D3D12_HEAP_PROPERTIES* pHeapProperties,  
    D3D12_HEAP_MISC_FLAG HeapMiscFlags,  
    const D3D12_RESOURCE_DESC* pResourceDesc,  
    D3D12_RESOURCE_USAGE InitialResourceState,  
    const D3D12_CLEAR_VALUE* pOptimizedClearValue,  
    REFIID riidResource,  
    void** ppvResource);
```

```
typedef struct D3D12_HEAP_PROPERTIES {  
    D3D12_HEAP_TYPE Type;  
    D3D12_CPU_PAGE_PROPERTIES CPUPageProperties;  
    D3D12_MEMORY_POOL MemoryPoolPreference;  
    UINT CreationNodeMask;  
    UINT VisibleNodeMask;  
} D3D12_HEAP_PROPERTIES;
```

```
struct D3D12_CLEAR_VALUE  
{  
    DXGI_FORMAT Format;  
    union  
    {  
        FLOAT Color[4];  
        D3D12_DEPTH_STENCIL_VALUE DepthStencil;  
    };  
} D3D12_CLEAR_VALUE;
```



Resources should be placed in the default heap for optimal performance. Only use upload or read back heaps if you need those features. This method creates both the resource and an implicit heap that is large enough to hold the resource. The resource is also mapped to the heap. Committed resources are easy to manage because the graphics programmer doesn't need to be concerned with placing the resource within the heap.

Create the Depth/Stencil View

In addition, before using the depth/stencil buffer, we must create an associated depth/stencil view to be bound to the pipeline. This is done similarly to creating the render target view. The following code example shows how we create the depth/stencil texture and its corresponding depth/stencil view:

```
D3D12_RESOURCE_DESC depthStencilDesc;
depthStencilDesc.Dimension = D3D12_RESOURCE_DIMENSION_TEXTURE2D;
depthStencilDesc.Alignment = 0;
depthStencilDesc.Width = mClientWidth;
depthStencilDesc.Height = mClientHeight;
depthStencilDesc.DepthOrArraySize = 1;
depthStencilDesc.MipLevels = 1;
depthStencilDesc.Format = DXGI_FORMAT_R24G8_TYPELESS;

depthStencilDesc.SampleDesc.Count = m4xMsaaState ? 4 : 1;
depthStencilDesc.SampleDesc.Quality = m4xMsaaQuality - 1 : 0;
depthStencilDesc.Layout = D3D12_TEXTURE_LAYOUT_UNKNOWN;
depthStencilDesc.Flags = D3D12_RESOURCE_FLAG_ALLOW_DEPTH_STENCIL;

D3D12_CLEAR_VALUE optClear;
optClear.Format = mDepthStencilFormat;
optClear.DepthStencil.Depth = 1.0f;
optClear.DepthStencil.Stencil = 0;
ThrowIfFailed(md3dDevice->CreateCommittedResource(&CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_DEFAULT), D3D12_HEAP_FLAG_NONE, &depthStencilDesc,
D3D12_RESOURCE_STATE_COMMON,
&optClear, IID_PPV_ARGS(mDepthStencilBuffer.GetAddressOf())));

// Create descriptor to mip level 0 of entire resource using the format of the resource.

D3D12_DEPTH_STENCIL_VIEW_DESC dsvDesc;
dsvDesc.Flags = D3D12_DSV_FLAG_NONE;
dsvDesc.ViewDimension = D3D12_DSV_DIMENSION_TEXTURE2D;
dsvDesc.Format = mDepthStencilFormat;
dsvDesc.Texture2D.MipSlice = 0;
md3dDevice->CreateDepthStencilView(mDepthStencilBuffer.Get(), &dsvDesc, DepthStencilView());

// Transition the resource from its initial state to be used as a depth buffer.

mCommandList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(mDepthStencilBuffer.Get(),
D3D12_RESOURCE_STATE_COMMON, D3D12_RESOURCE_STATE_DEPTH_WRITE));
```

Set the Viewport

Usually we like to draw the 3D scene to the entire back buffer, where the back buffer size corresponds to the entire screen (full-screen mode) or the entire client area of a window.

However, sometimes we only want to draw the 3D scene into a subrectangle of the back buffer. The subrectangle of the back buffer we draw into is called the viewport and it is described by the following structure:

```
typedef struct D3D12_VIEWPORT
{
    FLOAT TopLeftX;

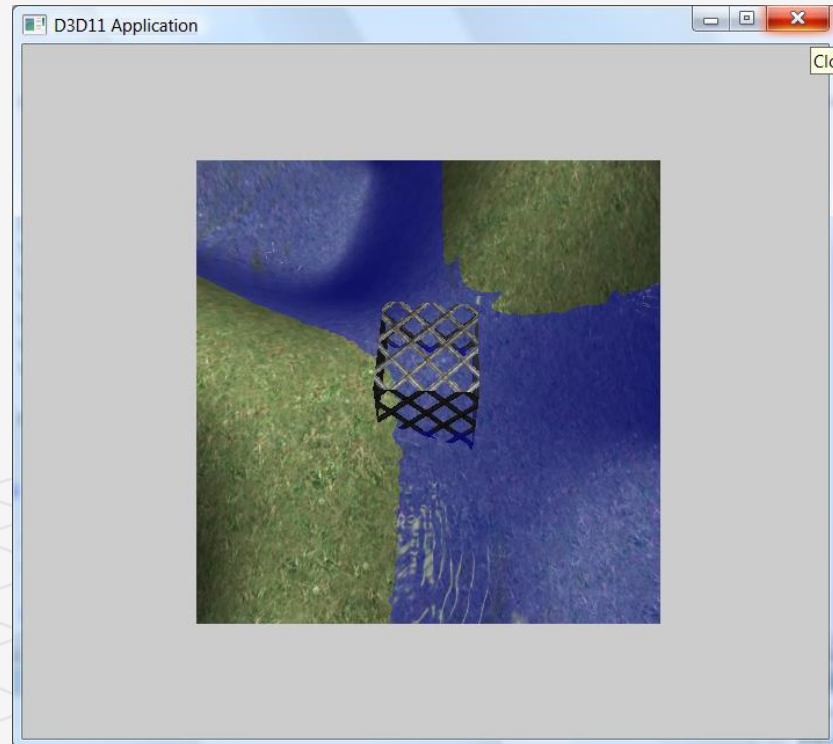
    FLOAT TopLeftY;

    FLOAT Width;

    FLOAT Height;

    FLOAT MinDepth;

    FLOAT MaxDepth;
} D3D12_VIEWPORT;
```



ID3D12CommandList::RSSetViewports

Once we have filled out the D3D12_VIEWPORT structure, we set the viewport with Direct3D with the ID3D12CommandList::RSSetViewports method.

```
D3D12_VIEWPORT vp;  
  
vp.TopLeftX = 0.0f;  
  
vp.TopLeftY = 0.0f;  
  
vp.Width = static_cast<float>(mClientWidth);  
  
vp.Height = static_cast<float>(mClientHeight);  
  
vp.MinDepth = 0.0f;  
  
vp.MaxDepth = 1.0f;  
  
mCommandList->RSSetViewports(1, &vp);
```

The first parameter is the number of viewports to bind (using more than one is for advanced effects), and the second parameter is a pointer to an array of viewports.

Set the Scissor Rectangles

We can define a *scissor rectangle* relative to the back buffer such that pixels outside this rectangle are culled (i.e., not rasterized to the back buffer). This can be used for optimizations. A scissor rectangle is defined by a D3D12_RECT structure which is typedefed to the following structure:

```
typedef struct tagRECT
{
    LONG left;
    LONG top;
    LONG right;
    LONG bottom;
} RECT;
```

We set the scissor rectangle with Direct3D with the ID3D12CommandList::RSSetScissorRects method. The following example creates and sets a scissor rectangle that covers the upper-left quadrant of the back buffer:

```
D3D12_RECT mScissorRect;

mScissorRect = { 0, 0, mClientWidth / 2, mClientHeight / 2};

mCommandList->RSSetScissorRects(1, &mScissorRect);
```

Query Performance Counter in Windows

Counters are used to provide information as to how well the operating system or an application, service, or driver is performing.

QPC is used for measure time intervals or high resolution time stamps.

QPC measures the total computation of response time of code execution.


QPC is independent and isn't synchronized to any external time reference.

If we want to use synchronize time stamps then, we must use `GetSystemTimePreciseAsFileTime`.

QPC has two types of API's:

Application: `QueryPerformanceCounter(QPC)`

Device Driver: `KeQueryPerformanceCounter`



TIMING AND ANIMATION

For accurate time measurements, we use the performance timer (or performance counter).

To use the Win32 functions for querying the performance timer, we must #include <windows.h>

The performance timer measures time in units called counts. We obtain the current time value, measured in counts, of the performance timer with the QueryPerformanceCounter function.

It retrieves the current value of the performance counter, which is a high resolution (<1us) time stamp that can be used for time-interval measurements.

```
void GameTimer::Start()
{
    __int64 startTime;
    QueryPerformanceCounter((LARGE_INTEGER*)&startTime);

    // Accumulate the time elapsed between stop and start pairs.
    //
    //      |<-----d----->|
    //  ----*-----*-----*-----> time
    //  mBaseTime      mStopTime      startTime

    if( mStopped )
    {
        mPausedTime += (startTime - mStopTime);

        mPrevTime = startTime;
        mStopTime = 0;
        mStopped = false;
    }
}
```

LARGE_INTEGER

LARGE_INTEGER is a portable 64-bit integer. (i.e., you want it to function the same way on multiple platforms)

If the system doesn't support 64-bit integer, then it's defined as two 32-bit integers, High Part and a Low Part.

If the system does support 64-bit integer then it's a union between the two 32-bit integer and a 64-bit integer called the QuadPart.

```
typedef union _LARGE_INTEGER {  
  
    struct {  
  
        DWORD LowPart;  
  
        LONG HighPart;  
  
    } DUMMYSTRUCTNAME;  
  
    struct {  
  
        DWORD LowPart;  
  
        LONG HighPart;  
  
    } u;  
  
    LONGLONG QuadPart;  
  
} LARGE_INTEGER;
```

QueryPerformanceFrequency

To get the frequency (counts per second) of the performance timer, we use the QueryPerformanceFrequency function:

```
LARGE_INTEGER StartingTime, EndingTime, ElapsedMicroseconds;
LARGE_INTEGER Frequency; //Default frequency supported by your hardware

QueryPerformanceFrequency(&Frequency);
QueryPerformanceCounter(&StartingTime);

// Activity to be timed

QueryPerformanceCounter(&EndingTime);
ElapsedMicroseconds.QuadPart = EndingTime.QuadPart - StartingTime.QuadPart;

//
// We now have the elapsed number of ticks, along with the
// number of ticks-per-second. We use these values
// to convert to the number of elapsed microseconds.
// To guard against loss-of-precision, we convert
// to microseconds *before* dividing by ticks-per-second.
//

ElapsedMicroseconds.QuadPart *= 1000000;
ElapsedMicroseconds.QuadPart /= Frequency.QuadPart; //in microsecond
```

Game Timer Class

```
class GameTimer
{
public:
    GameTimer();

    float TotalTime()const; // in seconds
    float DeltaTime()const; // in seconds

    void Reset(); // Call before message loop.
    void Start(); // Call when unpaused.
    void Stop(); // Call when paused.
    void Tick(); // Call every frame.

private:
    double mSecondsPerCount;
    double mDeltaTime;

    __int64 mBaseTime;
    __int64 mPausedTime;
    __int64 mStopTime;
    __int64 mPrevTime;
    __int64 mCurrTime;

    bool mStopped;
};
```

GameTimer::GameTimer()

The constructor, in particular, queries the frequency of the performance counter.

```
GameTimer::GameTimer()

: mSecondsPerCount(0.0), mDeltaTime(-1.0), mBaseTime(0),

  mPausedTime(0), mPrevTime(0), mCurrTime(0), mStopped(false)

{

    __int64 countsPerSec;

    QueryPerformanceFrequency((__LARGE_INTEGER*)&countsPerSec);

    mSecondsPerCount = 1.0 / (double)countsPerSec;

}
```

Time Elapsed Between Frames

When we render our frames of animation, we will need to know how much time has elapsed between frames so that we can update our game objects based on how much time has passed.

```
void GameTimer::Tick()
{
    if( mStopped )
    {
        mDeltaTime = 0.0;
        return;
    }

    __int64 currTime;
    QueryPerformanceCounter((LARGE_INTEGER*)&currTime);
    mCurrTime = currTime;

    // Time difference between this frame and the previous.
    mDeltaTime = (mCurrTime - mPrevTime)*mSecondsPerCount;

    // Prepare for next frame.
    mPrevTime = mCurrTime;

    // Force nonnegative. The DXSDK's CDXUTimer mentions that if the
    // processor goes into a power save mode or we get shuffled to another
    // processor, then mDeltaTime can be negative.
    if(mDeltaTime < 0.0)
    {
        mDeltaTime = 0.0;
    }
}
```

The Application Message Loop

The function Tick is called in the application message loop as follows. In this way, Δt is computed every frame and fed into the UpdateScene method so that the scene can be updated based on how much time has passed since the previous frame of animation.

```
int D3DApp::Run()
{
    MSG msg = {0};
    mTimer.Reset();
    while(msg.message != WM_QUIT)
    {
        // If there are Window messages then process them.
        if(PeekMessage( &msg, 0, 0, 0, PM_REMOVE ))
        {
            TranslateMessage( &msg );
            DispatchMessage( &msg );
        }
        // Otherwise, do animation/game stuff.
        else
        {
            mTimer.Tick();
            if( !mAppPaused )
            {
                CalculateFrameStats();
                Update(mTimer);
                Draw(mTimer);
            }
            else
            {
                Sleep(100);
            }
        }
    }
    return (int)msg.wParam;
}
```

Total Time

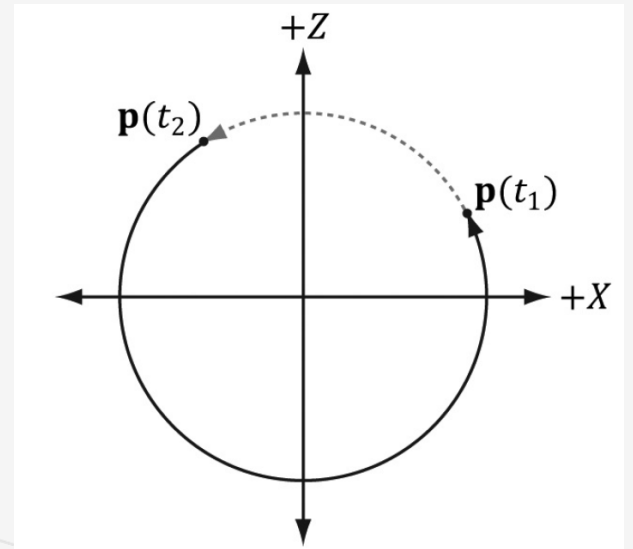
Another time measurement that can be useful is the amount of time that has elapsed since the application start, not counting paused time; we will call this *total time*. The following situation shows how this could be useful. Suppose the player has 300 seconds to complete a level. When the level starts, we can get the time t_{start} which is the time elapsed since the application started.

Another application of total time is when we want to animate a quantity as a function of time. For instance, suppose we wish to have a light orbit the scene as a function of time. Its position can be described by the parametric equations. Here t represents time, and as t (time) increases, the coordinates of the light are updated so that the light moves in a circle with radius 10 in the $y = 20$ plane. For this kind of animation, we also do not want to count paused time

$$X = \cos(t)$$

$$Y = 20$$

$$Z = 10 \sin(t)$$



GameTimer::TotalTime()

```
float GameTimer::TotalTime()const
{
    // If we are stopped, do not count the time that has passed since we stopped.
    // Moreover, if we previously already had a pause, the distance
    // mStopTime - mBaseTime includes paused time, which we do not want to count.
    // To correct this, we can subtract the paused time from mStopTime:
    //
    //      |<--paused time-->|
    //  ----*-----*-----*-----*-----> time
    //  mBaseTime      mStopTime      startTime      mStopTime      mCurrTime

    if( mStopped )
    {
        return (float)((((mStopTime - mPausedTime)-mBaseTime)*mSecondsPerCount));
    }

    // The distance mCurrTime - mBaseTime includes paused time,
    // which we do not want to count. To correct this, we can subtract
    // the paused time from mCurrTime:
    //
    //      (mCurrTime - mPausedTime) - mBaseTime
    //
    //      |<--paused time-->|
    //  ----*-----*-----*-----*-----> time
    //  mBaseTime      mStopTime      startTime      mCurrTime

    else
    {
        return (float)((((mCurrTime-mPausedTime)-mBaseTime)*mSecondsPerCount));
    }
}
```

GameTimer::Stop

Two important methods of the GameTimer class are Stop and Start. They should be called when the application is paused and unpaused, respectively, so that theGameTimer can keep track of paused time. The code comments explain the details of the stop method.

```
void GameTimer::Stop()

{

    if( !mStopped )

    {

        __int64 currTime;

        QueryPerformanceCounter((LARGE_INTEGER*)&currTime);

        mStopTime = currTime;

        mStopped = true;

    }

}
```

THE DEMO APPLICATION FRAMEWORK

The *d3dUtil.h* and *d3dUtil.cpp* files contain useful utility code, and the *d3dApp.h* and *d3dApp.cpp* files contain the core Direct3D application class code that is used to encapsulate a Direct3D sample application.

The D3DApp class is the base Direct3D application class, which provides functions for creating the main application window, running the application message loop, handling window messages, and initializing Direct3D. Moreover, the class defines the framework functions for the demo applications. Clients are to derive from D3DApp, override the virtual framework functions, and instantiate only a single instance of the derived D3DApp class.

For each sample application, we consistently override six virtual functions of D3DApp. These six functions are used to implement the code specific to the particular sample. The benefit of this setup is that the initialization code, message handling, etc., is implemented in the D3DApp class, so that the derived class needs to only focus on the specific code of the demo application.

1. **Initialize**: Use this method to put initialization code for the application such as allocating resources, initializing objects, and setting up the 3D scene.
2. **MsgProc**: This method implements the window procedure function for the main application window. Generally, you only need to override this method if there is a message you need to handle that D3DApp::MsgProc does not handle.
3. **CreateRtvAndDsvDescriptorHeaps**: Virtual function where you create the RTV and DSV descriptor heaps your application needs. The default implementation creates an RTV heap with SwapChainBufferCount many descriptors (for the buffer in the swap chain) and a DSV heap with one descriptor (for the depth/stencil buffer).
4. **OnResize**: This method is called by D3DApp::MsgProc when a WM_SIZE message is received. When the window is resized, some Direct3D properties need to be changed, as they depend on the client area dimensions.
5. **Update**: This abstract method is called every frame and should be used to update the 3D application over time (e.g., perform animations, move the camera, do collision detection, check for user input, and etc.).
6. **Draw**: This abstract method is invoked every frame and is where we issue rendering commands to actually draw our current frame to the back buffer. When we are done drawing our frame, we call the the IDXGISwapChain::Present method to present the back buffer to the screen.

Mouse Events

if you want to handle mouse messages, you can override these methods instead of overriding the MsgProc method. The first parameter is the same as the WPARAM parameter for the various mouse messages, which stores the mouse button states (i.e., which mouse buttons were pressed when the event was raised). The second and third parameters are the client area (x, y) coordinates of the mouse cursor.

```
virtual void OnMouseDown(WPARAM btnState, int x, int y) { }
```

```
virtual void OnMouseUp(WPARAM btnState, int x, int y) { }
```

```
virtual void OnMouseMove(WPARAM btnState, int x, int y) { }
```

Non-Framework methods

1. D3DApp: The constructor simply initializes the data members to default values.
2. ~D3DApp: The destructor releases the COM interfaces the D3DApp acquires, and flushes the command queue. The reason we need to flush the command queue in the destructor is that we need to wait until the GPU is done processing the commands in the queue before we destroy any resources the GPU is still referencing. Otherwise, the GPU might crash when the application exits.
3. AppInst: Trivial access function returns a copy of the application instance handle.
4. MainWnd: Trivial access function returns a copy of the main window handle.
5. AspectRatio: The aspect ratio is defined as the ratio of the back buffer width to its height.
6. Get4xMsaaState: Returns true if 4X MSAA is enabled and false otherwise.
7. Set4xMsaaState: Enables/disables 4X MSAA.
8. Run: This method wraps the application message loop. It uses the Win32 PeekMessage function so that it can process our game logic when no messages are present.
9. InitMainWindow: Initializes the main application window; we assume the reader is familiar with basic Win32 window initialization.
10. InitDirect3D: Initializes Direct3D.
11. CreateSwapChain: Creates the swap chain.
12. CreateCommandObjects: Creates the command queue, a command list allocator, and a command list.
13. FlushCommandQueue: Forces the CPU to wait until the GPU has finished processing all the commands in the queue.
14. CurrentBackBuffer: Returns an ID3D12Resource to the current back buffer in the swap chain.
15. CurrentBackBufferView: Returns the RTV (render target view) to the current back buffer.
16. DepthStencilView: Returns the DSV (depth/stencil view) to the main depth/stencil buffer.
17. CalculateFrameStats: Calculates the average frames per second and the average milliseconds per frame.
18. LogAdapters: Enumerates all the adapters on a system.
19. LogAdapterOutputs: Enumerates all the outputs associated with an adapter.
20. LogOutputDisplayModes: Enumerates all the display modes an output supports for a given format.

Frame Statistics

we simply count the number of frames processed (and store it in a variable n) over some specified time period t . Then, the average FPS over the time period t is $fps_{avg} = n/t$.

```
void D3DApp::CalculateFrameStats()
{
    // Code computes the average frames per second, and also the
    // average time it takes to render one frame. These stats
    // are appended to the window caption bar.

    static int frameCnt = 0;
    static float timeElapsed = 0.0f;

    frameCnt++;

    // Compute averages over one second period.
    if( (mTimer.TotalTime() - timeElapsed) >= 1.0f )
    {
        float fps = (float)frameCnt; // fps = frameCnt / 1
        float mspf = 1000.0f / fps;

        wstring fpsStr = to_wstring(fps);
        wstring mspfStr = to_wstring(mspf);

        wstring windowText = mMainWndCaption +
            L"    fps: " + fpsStr +
            L"    mspf: " + mspfStr;

        SetWindowText(mhMainWnd, windowText.c_str());

        // Reset for next average.
        frameCnt = 0;
        timeElapsed += 1.0f;
    }
}
```

The Message Handler

The first message we handle is the WM_ACTIVATE message.

```
LRESULT D3DApp::MsgProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch( msg )
    {
        // WM_ACTIVATE is sent when the window is activated or deactivated.
        // We pause the game when the window is deactivated and unpause it
        // when it becomes active.
        case WM_ACTIVATE:
            if( LOWORD(wParam) == WA_INACTIVE )
            {
                mAppPaused = true;
                mTimer.Stop();
            }
            else
            {
                mAppPaused = false;
                mTimer.Start();
            }
            return 0;
    }
}
```

when our application becomes deactivated, we set the data member mAppPaused to true, and when our application becomes active, we set the data member mAppPaused to false. In addition, when the application is paused, we stop the timer, and then resume the timer once the application becomes active again. If we look back at the implementation to D3DApp::Run, we find that if our application is paused, then we do not update our application code, but instead free some CPU cycles back to the OS; in this way, our application does not hog CPU cycles when it is inactive.

WM_SIZE message

This message is called when the window is resized. The main reason for handling this message is that we want the back buffer and depth/stencil dimensions to match the dimensions of the client area rectangle (so no stretching occurs). Thus, every time the window is resized, we want to resize the buffer dimensions.

```
case WM_SIZE:
    // Save the new client area dimensions.
    mClientWidth = LOWORD(lParam);
    mClientHeight = HIWORD(lParam);
    if( md3dDevice )
    {
        if( wParam == SIZE_MINIMIZED )
        {
            mAppPaused = true;
            mMinimized = true;
            mMaximized = false;
        }
        else if( wParam == SIZE_MAXIMIZED )
        {
            mAppPaused = false;
            mMinimized = false;
            mMaximized = true;
            OnResize();
        }
        else if( wParam == SIZE_RESTORED )
        {
            if( mMinimized ) // Restoring from minimized state?
            {
                mAppPaused = false;
                mMinimized = false;
                OnResize();
            }
            else if( mMaximized ) // Restoring from maximized state?
            {
                mAppPaused = false;
                mMaximized = false;
                OnResize();
            }
            else if( mResizing )
            {
            }
            else // API call such as SetWindowPos or mSwapChain->SetFullscreenState.
            {
                OnResize();
            }
        }
    }
    return 0;
```


WM_ENTERSIZEMOVE

```
// WM_EXITSIZEMOVE is sent when the user grabs the resize bars.
case WM_ENTERSIZEMOVE:
    mAppPaused = true;
    mResizing = true;
    mTimer.Stop();
    return 0;

// WM_EXITSIZEMOVE is sent when the user releases the resize bars.
// Here we reset everything based on the new window dimensions.
case WM_EXITSIZEMOVE:
    mAppPaused = false;
    mResizing = false;
    mTimer.Start();
    OnResize();
    return 0;

// WM_DESTROY is sent when the window is being destroyed.
case WM_DESTROY:
    PostQuitMessage(0);
    return 0;

// The WM_MENUCHAR message is sent when a menu is active and the user presses
// a key that does not correspond to any mnemonic or accelerator key.
case WM_MENUCHAR:
    // Don't beep when we alt-enter.
    return MAKELRESULT(0, MNC_CLOSE);

// Catch this message so to prevent the window from becoming too small.
case WM_GETMINMAXINFO:
    ((MINMAXINFO*)lParam)->ptMinTrackSize.x = 200;
    ((MINMAXINFO*)lParam)->ptMinTrackSize.y = 200;
    return 0;
```

mouse input virtual functions

```
case WM_LBUTTONDOWN:
case WM_MBUTTONDOWN:
case WM_RBUTTONDOWN:
OnMouseDown(wParam, GET_X_LPARAM(lParam), GET_Y_LPARAM(lParam));
return 0;
case WM_LBUTTONUP:
case WM_MBUTTONUP:
case WM_RBUTTONUP:
OnMouseUp(wParam, GET_X_LPARAM(lParam), GET_Y_LPARAM(lParam));
return 0;
case WM_MOUSEMOVE:
OnMouseMove(wParam, GET_X_LPARAM(lParam), GET_Y_LPARAM(lParam));
return 0;
case WM_KEYUP:
    if(wParam == VK_ESCAPE)
    {
        PostQuitMessage(0);
    }
    else if((int)wParam == VK_F2)
        Set4xMsaaState(!m4xMsaaState);

return 0;
```

ID3D12GraphicsCommandList::ClearRenderTargetView

We always clear the back buffer render target and depth/stencil buffer every frame before drawing to start the image fresh.

```
void ID3D12GraphicsCommandList::ClearRenderTargetView(  
  
D3D12_CPU_DESCRIPTOR_HANDLE RenderTargetView,  
  
const FLOAT ColorRGBA[4],  
  
UINT NumRects,  
  
const D3D12_RECT* pRects);
```

1. RenderTargetView: RTV to the resource we want to clear.
2. ColorRGBA: Defines the color to clear the render target to.
3. NumRects: The number of elements in the pRects array. This can be 0.
4. pRects: An array of D3D12_RECTs that identify rectangle regions on the render target to clear. This can be a nullptr to indicate to clear the entire render target.

ID3D12GraphicsCommandList::ClearDepthStencilView

```
void ID3D12GraphicsCommandList::ClearDepthStencilView(  
D3D12_CPU_DESCRIPTOR_HANDLE DepthStencilView,  
D3D12_CLEAR_FLAGS ClearFlags,  
FLOAT Depth,  
UINT8 Stencil,  
UINT NumRects,  
const D3D12_RECT* pRects);
```

1. DepthStencilView: DSV to the depth/stencil buffer to clear.
2. ClearFlags: Flags indicating which part of the depth/stencil buffer to clear. This can be either D3D12_CLEAR_FLAG_DEPTH, D3D12_CLEAR_FLAG_STENCIL, or both bitwised ORed together.
3. Depth: Defines the value to clear the depth values to.
4. Stencil: Defines the value to clear the stencil values to.
5. NumRects: The number of elements in the pRects array. This can be 0.
6. pRects: An array of D3D12_RECTs that identify rectangle regions on the render target to clear. This can be a nullptr to indicate to clear the entire render target.

ID3D12GraphicsCommandList::OMSetRenderTargets

This method sets the render target and depth/stencil buffer we want to use to the pipeline.

```
void ID3D12GraphicsCommandList::OMSetRenderTargets(  
    UINT NumRenderTargetDescriptors,  
    const D3D12_CPU_DESCRIPTOR_HANDLE * pRenderTargetDescriptors,  
    BOOL RTsSingleHandleToDescriptorRange,  
    const D3D12_CPU_DESCRIPTOR_HANDLE * pDepthStencilDescriptor);
```

1. NumRenderTargetDescriptors: Specifies the number of RTVs we are going to bind. Using multiple render targets simultaneously is used for some advanced techniques. For now, we always use one RTV.
2. pRenderTargetDescriptors: Pointer to an array of RTVs that specify the render targets we want to bind to the pipeline.
3. RTsSingleHandleToDescriptorRange: Specify true if all the RTVs in the previous array are contiguous in the descriptor heap. Otherwise, specify false.
4. pDepthStencilDescriptor: Pointer to a DSV that specifies the depth/stencil buffer we want to bind to the pipeline.

IDXGISwapChain::Present

When we Present the swap chain to swap the front and back buffers, we have to update the index to the current back buffer as well so that we render to the new back buffer on the subsequent frame:

```
// Done recording commands.

ThrowIfFailed(mCommandList->Close());

// Add the command list to the queue for execution.

ID3D12CommandList* cmdLists[] = { mCommandList.Get() };

mCommandQueue->ExecuteCommandLists(_countof(cmdLists), cmdLists);

// swap the back and front buffers

ThrowIfFailed(mSwapChain->Present(0, 0));

mCurrBackBuffer = (mCurrBackBuffer + 1) % SwapChainBufferCount;

FlushCommandQueue();
```

DEBUGGING DIRECT3D APPLICATIONS

Many Direct3D functions return HRESULT error codes, we check a returned HRESULT, and if it failed, we throw an exception that stores the error code, function name, filename, and line number of the offending call. This is done with the following code in *d3dUtil.h*. If an HRESULT fails, an exception is thrown, we output information about it via the MessageBox function, and then exit the application.

```
class DxException {
public:
    DxException() = default;
    DxException(HRESULT hr, const std::wstring& functionName, const std::wstring& filename, int lineNumber);
    std::wstring ToString()const;
    HRESULT ErrorCode = S_OK;
    std::wstring FunctionName;
    std::wstring Filename;
    int LineNumber = -1;
};

#ifdef ThrowIfFailed
#define ThrowIfFailed(x) \
{ \
    HRESULT hr__ = (x); \
    std::wstring wfn = AnsiToWString(__FILE__); \
    if(FAILED(hr__)) { throw DxException(hr__, L#x, wfn, __LINE__); } \
}
#endif
```

The L#x turns the ThrowIfFailed macro's argument token into a Unicode string. In this way, we can output the function call that caused the error to the message box.