Direct3D 12 Programming Guide

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.

In this section

Topic	Description
What is Direct3D 12?	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.
Understanding Direct3D 12	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.
Work Submission in Direct3D 12	To improve the CPU efficiency of Direct3D apps, Direct3D 12 no longer supports an immediate context associated with a device. Instead, apps record and then submit *command lists*, which contain drawing and resource management calls. These command lists can be submitted from multiple threads to one or more command queues, which manage the execution of the commands. This fundamental change increases single-threaded efficiency by allowing apps to pre-compute rendering work for later re-use, and it takes advantage of multi-core systems by spreading rendering work across multiple threads.
Resource Binding in Direct3D 12	Binding is the process of linking resource objects to the graphics pipeline. The key to resource binding in DirectX 12 are the concepts of a *descriptor*, *descriptor tables*, *descriptor heaps*, and a *root signature*.
Memory Management in Direct3D 12	There are two major aspects that applications must manage for D3D12: synchronization and physical memory residency.
Multi-Engine and Multi-Adapter Synchronization	Provides an overview and lists APIs relevant to multi-engine (the 3D, compute, and copy engines), and multi-adapter.
Rendering	This section contains information about rendering features new to Direct3D 12 (and Direct3D 11.3).
Counters, Queries and Performance Measurement	The following sections describe features for use in performance testing and improvement, such as queries, counters, timing, and predication.
Direct3D 12 Interop	D3D12 can interop both with existing code written using other Windows graphics APIs, as

	well as with new code written using D3D12 which implements a portion of a graphics engine.
Working Samples	Working samples are available for download, showing the usage of a number of features of Direct3D 12.
D3D12 Code Walk-Throughs	This section provides code for sample scenarios. Many of the walk-throughs provide details on what coding is required to be added to a basic sample, to avoid repeating the basic component code for each scenario.

What is Direct3D 12?

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.

What makes Direct3D 12 better?

Direct3D 12 provides a lower level of hardware abstraction than ever before, which allows developers to significantly improve the multi-thread scaling and CPU utilization of their titles. With Direct3D 12, titles are responsible for their **memory management**. In addition, by using Direct3D 12, games and titles benefit from reduced GPU overhead via features such as **command queues and lists**, **descriptor tables**, and concise **pipeline state objects**.

Direct3D 12, and Direct3D 11.3, introduce a set of new features for the rendering pipeline: conservative rasterization to enable reliable hit detection, volume-tiled resources to enable streamed three dimension resources to be treated as if they were all in video memory, rasterizer ordered views to enable reliable transparency rendering, Adaptive Scalable Texture Compression (ASTC) to standardize texture compression, setting the stencil reference within a shader to enable special shadowing and other effects, and also improved texture mapping and typed Unordered Access View (UAV) loads.

Who is Direct3D 12 for?

Direct3D 12 provides four main benefits to graphics developers (compared with Direct3D 11): vastly reduced CPU overhead, significantly improved power consumption, up to around twenty percent improvement in GPU efficiency, and cross-platform development for a Windows 10 device (PC, tablet, console or phone).

Direct3D 12 is certainly for advanced graphics programmers, it requires a fine level of tuning and significant graphics expertise. Direct3D 12 is designed to make full use of multi-threading, careful CPU/GPU synchronization, and the transition and re-use of resources from one purpose to another. All techniques that require a considerable amount of memory level programming skill.

Another advantage that Direct3D 12 has is its small API footprint. There are around 200 methods, and about one third of these do all the heavy lifting. This means that a graphics developer should be able to educate themselves on - and master - the full API set without the weight of having to memorize a great volume of API calls.

Direct3D 12 does not replace Direct3D 11. The new rendering features of Direct3D 12 are available in Direct3D 11.3. Direct3D 11.3 is a low level graphics engine API, and Direct3D 12 goes even deeper.

There are at least two ways a development team can approach a Direct3D 12 title.

For a project that takes full advantage of the benefits of Direct3D 12, a highly customized Direct3D 12 game engine should be developed from the ground up.

One approach is that if graphics developers understand the use and re-use of resources within their titles, and can take advantage of this by minimizing uploading and copying, then a highly efficient engine can be developed and customized for these titles. The performance improvements could be very considerable, freeing up CPU time to increase the number of draw calls, and so adding more luster to graphics rendering.

The programming investment is significant, and debugging and instrumentation of the project should be considered from the very start: threading, synchronization and other timing bugs can be challenging.

A shorter term approach would be to address known bottlenecks in a Direct3D 11 title; these can be addressed by using the 11on12 or interop techniques enabling the two APIs to work together. This approach minimizes the changes necessary to an existing Direct3D 11 graphics engine, however the performance gains will be limited to the relief of the bottleneck that the Direct3D 12 code addresses.

Direct3D 12 is all about dramatic graphics engine performance: ease of development, high level constructs, and compiler support have been scaled back to enable this. Driver support and ease of debugging remain on a par with Direct3D 11.

Direct3D 12 is new territory, for the inquisitive expert to explore.

Understanding Direct3D 12

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.

Use the topics in this section to set up and learn about the environment in which you will program your apps and games with Direct3D 12. This content will also help you to port your Direct3D 11 apps and games to Direct3D 12, so you can take advantage of Direct3D 12 features and efficiencies.

To program with Direct3D 12, you need these components:

- A hardware platform with a Direct3D 12-compatible GPU
- Display drivers that support the Windows Display Driver Model (WDDM) 2.0

In this section

Topic	Description
Direct3D 12 Programming Environment Setup	The Direct3D 12 headers and libraries are part of the Windows 10 SDK. There is no separate download or installation required to use Direct3D 12.
Creating a Basic Direct3D 12 Component	This topic describes the call flow to create a basic Direct3D 12 component.
Important Changes from Direct3D 11 to Direct3D 12	Direct3D 12 represents a significant departure from the Direct3D 11 programming model. Direct3D 12 lets apps get closer to hardware than ever before.
Hardware Feature Levels	Describes the functionality of the 12_0 and 12_1 hardware feature levels.

Direct3D 12 Programming Environment Setup

The Direct3D 12 headers and libraries are part of the Windows 10 SDK. There is no separate download or installation required to use Direct3D 12.

Development environment

After you install the Windows 10 SDK software, and Visual Studio 2015, the setup of your Direct3D 12 programming environment is complete. Visual Studio 2015 is recommended, as it will include the D3D12 graphics debugging tools, but earlier versions of Visual Studio will work for program development.

To use the Direct3D 12 API, include D3d12.h and link to D3d12.lib, or query the entry points directly in D3d12.dll.

The following headers and libraries are available. The location of the static libraries depends on the version (32-bit or 64-bit) of Windows 10 that is running on your computer.

Header or library file name	Description	Install location
D3d12.h	Direct3D 12 API header	%DXSDK_DIR%\Include
D3d12.lib	Static Direct3D 12 API stub library	%DXSDK_DIR%\Lib
D3d12.dll	Dynamic Direct3D 12 API library	%WINDIR%\System32
D3d12SDKLayers.h	Direct3D 12 debug header	%DXSDK_DIR%\Include
D3d12SDKLayers.dll	Dynamic Direct3D 12 debug library	%WINDIR%\System32

Helper structures

There are a number of helper structures that, in particular, make it easy to initialize a number of the D3D12 structures. These structures, and some utility functions are in the header D3dx12.h. This header is open source, and can be modified by a developer as required. Refer to Helper Structures and Functions for D3D12.

Supported tools and libraries

The DirectXMath library can be used with D3D 12. Refer to DirectXMath documentation at MSDN.

Check on the status of the following libraries:

- DirectXTex : a texture processing library.
- DirectXTK : a collection of helper classes for writing DirectX 11.x code in C++.
- DirectXMesh : a geometry processing library.

Samples

For a list of working D3D12 samples and how to locate and run them, refer to Working Samples.

For walk-throughs on how to add code to enable particular features, refer to D3D12 Code Walk-Throughs.

Debug layer

The debug layer provides extensive additional parameter and consistency validation (such as validating shader linkage and resource binding, validating parameter consistency, and reporting error descriptions).

The header required to support the debugging layer, D3D12SDKLayers.h, is included by default from d3d12.h.

When the debug layer lists memory leaks, it outputs a list of object interface pointers along with their friendly names. The default friendly name is "<unnamed>". You can set the friendly name by using the **ID3D12Object::SetName** method. Typically, you should compile these calls out of your production version.

We recommend that you use the debug layer to debug your apps to ensure that they are clean of errors and warnings. The debug layer helps you write Direct3D 12 code. In addition, your productivity can increase when you use the debug layer because you can immediately see the causes of obscure rendering errors or even black screens at their source. The debug layer provides warnings for many issues. For example:

- Forgot to set a texture but read from it in your pixel shader.
- Output depth but have no depth-stencil state bound.
- Texture creation failed with INVALIDARG.

For details of all the debug interfaces and methods, refer to the Debug Layer Reference.

Educational videos

There are a number of Direct3D 12 related videos at Microsoft DirectX 12 Graphics Education.

Creating a basic Direct3D 12 component

This topic describes the call flow to create a basic Direct3D 12 component.

Code flow for a simple app

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

Tip Features new to Direct3D 12 are followed by a note.

- 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.
  - Create the command queue.
  - Create the swap chain.
  - Create a render target view (RTV) descriptor heap.
  - **Note** A descriptor heap can be thought of as an array of descriptors. Where each descriptor fully describes an object to the GPU.
  - Create frame resources (a render target view for each frame).
  - Create a command allocator.
  - **Note** A command allocator manages the underlying storage for command lists and bundles.
- Initialize the assets.
  - Create an empty root signature.
  - **Note** 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.
  - **Note** 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*).
  - Create the command list.
  - Close the command list.
  - Create and load the vertex buffers.
  - Create the vertex buffer views.
  - Create a fence.
  - **Note** A fence is used to synchronize the CPU and GPU.
  - Create an event handle.
  - Wait for the GPU to finish.
  - **Note** Check on the fence!

Refer to class D3D12HelloTriangle, OnInit, LoadPipeline and LoadAssets.

Update

Update everything that should change since the last frame.

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

Refer to **OnUpdate**.

Render

Draw the new world.

- Populate the command list.
  - Reset the command list allocator.
  - **Note** Re-use the memory that is associated with the command allocator.
  - Reset the command list.
  - Set the graphics root signature.
  - **Note** 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.
  - **Note** 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.
  - **Note** 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.
- **Note** Keep updating and checking the fence.

Refer to **OnRender**.

Destroy

Cleanly close down the app.

- Wait for the GPU to finish.
- **Note** Final check on the fence.
- Close the event handle.

Refer to **OnDestroy**.

Code example for a simple app

The following expands the code flow above to include the code required for a basic sample.

class D3D12HelloTriangle

First define the global variables in a header file, including the structures:

- D3D12_VIEWPORT
- D3D12_RECT (for the scissors rectangle)
- D3D12_VERTEX_BUFFER_VIEW

C++

```cpp
#include "DXSample.h"

using namespace DirectX;
using namespace Microsoft::WRL;

class D3D12HelloTriangle : public DXSample {
public:
    D3D12HelloTriangle(UINT width, UINT height, std::wstring name);

protected:
    virtual void OnInit();
    virtual void OnUpdate();
    virtual void OnRender();
    virtual void OnDestroy();
    virtual bool OnEvent(MSG msg);

private:
    static const UINT FrameCount = 2;

    struct Vertex {
        XMFLOAT3 position;
        XMFLOAT4 color;
    };

    // Pipeline objects.
    D3D12_VIEWPORT m_viewport;
    D3D12_RECT m_scissorRect;
    ComPtr<IDXGISwapChain3> m_swapChain;
    ComPtr<ID3D12Device> m_device;
    ComPtr<ID3D12Resource> m_renderTargets[FrameCount];
    ComPtr<ID3D12CommandAllocator> m_commandAllocator;
    ComPtr<ID3D12CommandQueue> m_commandQueue;
    ComPtr<ID3D12RootSignature> m_rootSignature;
    ComPtr<ID3D12DescriptorHeap> m_rtvHeap;
    ComPtr<ID3D12PipelineState> m_pipelineState;
    ComPtr<ID3D12GraphicsCommandList> m_commandList;
    UINT m_rtvDescriptorSize;

    // App resources.
    ComPtr<ID3D12Resource> m_vertexBuffer;
    D3D12_VERTEX_BUFFER_VIEW m_vertexBufferView;

    // Synchronization objects.
    UINT m_frameIndex;
    HANDLE m_fenceEvent;
    ComPtr<ID3D12Fence> m_fence;
    UINT64 m_fenceValue;

    void LoadPipeline();
    void LoadAssets();
    void PopulateCommandList();
    void WaitForPreviousFrame();
};
```

OnInit()

In the projects main source file, start initializing the objects:

C++

```cpp
#include "stdafx.h"
#include "D3D12HelloTriangle.h"

D3D12HelloTriangle::D3D12HelloTriangle(UINT width, UINT height, std::wstring name)
    : DXSample(width, height, name),
      m_frameIndex(0),
      m_viewport(),
      m_scissorRect(),
      m_rtvDescriptorSize(0)
{
    m_viewport.Width = static_cast<float>(width);
    m_viewport.Height = static_cast<float>(height);
    m_viewport.MaxDepth = 1.0f;
    m_scissorRect.right = static_cast<LONG>(width);
    m_scissorRect.bottom = static_cast<LONG>(height);
}
```

C++

```cpp
void D3D12HelloTriangle::OnInit() {
    LoadPipeline();
    LoadAssets();
}
```

LoadPipeline()

The following code tries to create a hardware device, and if that fails, falls back on a WARP (software rasterizer) device. The process of creating devices and swap chains is very similar to Direct3D 11.

The following code calls the creation methods in turn:

- Enables the debug layer, with calls to:
  - D3D12GetDebugInterface
  - ID3D12Debug::EnableDebugLayer
- Creates the device:
  - CreateDXGIFactory1
  - **IDXGIFactory4::EnumWarpAdapter**
  - D3D12CreateDevice
- Fill out a command queue description, then create the command queue:
  - D3D12_COMMAND_QUEUE_DESC
  - ID3D12Device::CreateCommandQueue
- Fill out a swapchain description, then create the swap chain:

    - DXGI_SWAP_CHAIN_DESC
    - IDXGIFactory::CreateSwapChain
    - IDXGISwapChain3::GetCurrentBackBufferIndex
- Fill out a heap description. then create a descriptor heap:
    - D3D12_DESCRIPTOR_HEAP_DESC
    - ID3D12Device::CreateDescriptorHeap
    - ID3D12Device::GetDescriptorHandleIncrementSize
- Create the render target view:
    - CD3DX12_CPU_DESCRIPTOR_HANDLE
    - GetCPUDescriptorHandleForHeapStart
    - IDXGISwapChain::GetBuffer
    - ID3D12Device::CreateRenderTargetView
- Create the command allocator: ID3D12Device::CreateCommandAllocator.

In later steps, command lists are obtained from the command allocator and submitted to the command queue.

Load the rendering pipeline dependencies.

C++

```cpp
void D3D12HelloTriangle::LoadPipeline() {
#ifdef _DEBUG
    // Enable the D3D12 debug layer.
    {
        ComPtr<ID3D12Debug> debugController;
        if (SUCCEEDED(D3D12GetDebugInterface(IID_PPV_ARGS(&debugController)))) {
            debugController->EnableDebugLayer();
        }
    }
#endif
    ComPtr<IDXGIFactory4> factory;
    ThrowIfFailed(CreateDXGIFactory1(IID_PPV_ARGS(&factory)));

    if (m_useWarpDevice) {
        ComPtr<IDXGIAdapter> warpAdapter;
        ThrowIfFailed(factory->EnumWarpAdapter(IID_PPV_ARGS(&warpAdapter)));
        ThrowIfFailed(D3D12CreateDevice(
            warpAdapter.Get(),
            D3D_FEATURE_LEVEL_11_0,
            IID_PPV_ARGS(&m_device)
            ));
    } else {
        ThrowIfFailed(D3D12CreateDevice(
            nullptr,
            D3D_FEATURE_LEVEL_11_0,
            IID_PPV_ARGS(&m_device)
            ));
    }

    // Describe and create the command queue.
    D3D12_COMMAND_QUEUE_DESC queueDesc = {};
    queueDesc.Flags = D3D12_COMMAND_QUEUE_FLAG_NONE;
    queueDesc.Type = D3D12_COMMAND_LIST_TYPE_DIRECT;

    ThrowIfFailed(m_device->CreateCommandQueue(&queueDesc, IID_PPV_ARGS(&m_commandQueue)));

    // Describe and create the swap chain.
    DXGI_SWAP_CHAIN_DESC swapChainDesc = {};
    swapChainDesc.BufferCount = FrameCount;
    swapChainDesc.BufferDesc.Width = m_width;
    swapChainDesc.BufferDesc.Height = m_height;
    swapChainDesc.BufferDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
    swapChainDesc.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;
    swapChainDesc.SwapEffect = DXGI_SWAP_EFFECT_FLIP_DISCARD;
    swapChainDesc.OutputWindow = m_hwnd;
    swapChainDesc.SampleDesc.Count = 1;
    swapChainDesc.Windowed = TRUE;

    ComPtr<IDXGISwapChain> swapChain;
    ThrowIfFailed(factory->CreateSwapChain(
        m_commandQueue.Get(),        // Swap chain needs the queue so that it can force a flush on it.
        &swapChainDesc,
        &swapChain
        ));

    ThrowIfFailed(swapChain.As(&m_swapChain));

    m_frameIndex = m_swapChain->GetCurrentBackBufferIndex();

    // Create descriptor heaps.
    {
        // Describe and create a render target view (RTV) descriptor heap.
        D3D12_DESCRIPTOR_HEAP_DESC rtvHeapDesc = {};
        rtvHeapDesc.NumDescriptors = FrameCount;
        rtvHeapDesc.Type = D3D12_DESCRIPTOR_HEAP_TYPE_RTV;
        rtvHeapDesc.Flags = D3D12_DESCRIPTOR_HEAP_FLAG_NONE;
        ThrowIfFailed(m_device->CreateDescriptorHeap(&rtvHeapDesc, IID_PPV_ARGS(&m_rtvHeap)));
        m_rtvDescriptorSize = m_device->GetDescriptorHandleIncrementSize(D3D12_DESCRIPTOR_HEAP_TYPE_RTV);
    }

    // Create frame resources.
    {
        CD3DX12_CPU_DESCRIPTOR_HANDLE rtvHandle(m_rtvHeap->GetCPUDescriptorHandleForHeapStart());

        // Create a RTV for each frame.
        for (UINT n = 0; n < FrameCount; n++) {
            ThrowIfFailed(m_swapChain->GetBuffer(n, IID_PPV_ARGS(&m_renderTargets[n])));
            m_device->CreateRenderTargetView(m_renderTargets[n].Get(), nullptr, rtvHandle);
            rtvHandle.Offset(1, m_rtvDescriptorSize);
        }
    }

    ThrowIfFailed(m_device->CreateCommandAllocator(D3D12_COMMAND_LIST_TYPE_DIRECT,
    IID_PPV_ARGS(&m_commandAllocator)));
}
```

LoadAssets()

Loading and preparing assets is a lengthy process. Many of these stages are similar to D3D 11, though some are new to D3D 12.

In Direct3D 12, required pipeline state is attached to a command list via a pipeline state object (PSO). This example shows how to create a PSO. You can store the PSO as a member variable and reuse it as many times as needed.

A descriptor heap defines the views and how to access resources (for example, a render target view).

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

The following APIs and processes are called in succession.

- Create an empty root signature, using the available helper structure:
  - CD3DX12_ROOT_SIGNATURE_DESC
  - D3D12SerializeRootSignature
  - ID3D12Device::CreateRootSignature
- Load and compile the shaders: D3DCompileFromFile.
- Create the vertex input layout: D3D12_INPUT_ELEMENT_DESC.
- Fill out a pipeline state description, using the helper structures available, then create the graphics pipeline state:
  - D3D12_GRAPHICS_PIPELINE_STATE_DESC
  - CD3DX12_RASTERIZER_DESC
  - CD3DX12_BLEND_DESC
  - ID3D12Device::CreateGraphicsPipelineState
- Create, then close, a command list:
  - ID3D12Device::CreateCommandList
  - ID3D12GraphicsCommandList::Close
- Create the vertex buffer: ID3D12Device::CreateCommittedResource.
- Copy the vertex data to the vertex buffer:
  - ID3D12Resource::Map
  - ID3D12Resource::Unmap
- Initialize the vertex buffer view: GetGPUVirtualAddress.
- Create and initialize the fence: ID3D12Device::CreateFence.
- Create an event handle for use with frame synchronization.
- Wait for the GPU to finish.

C++

```cpp
void D3D12HelloTriangle::LoadAssets() {
    // Create an empty root signature.
    {
        CD3DX12_ROOT_SIGNATURE_DESC rootSignatureDesc;
        rootSignatureDesc.Init(0, nullptr, 0, nullptr,
                               D3D12_ROOT_SIGNATURE_FLAG_ALLOW_INPUT_ASSEMBLER_INPUT_LAYOUT);

        ComPtr<ID3DBlob> signature;
        ComPtr<ID3DBlob> error;
        ThrowIfFailed(D3D12SerializeRootSignature(&rootSignatureDesc, D3D_ROOT_SIGNATURE_VERSION_1,
                      &signature, &error));
        ThrowIfFailed(m_device->CreateRootSignature(0, signature->GetBufferPointer(),
                      signature->GetBufferSize(), IID_PPV_ARGS(&m_rootSignature)));
    }

    // Create the pipeline state, which includes compiling and loading shaders.
    {
        ComPtr<ID3DBlob> vertexShader;
        ComPtr<ID3DBlob> pixelShader;

#ifdef _DEBUG
        // Enable better shader debugging with the graphics debugging tools.
        UINT compileFlags = D3DCOMPILE_DEBUG | D3DCOMPILE_SKIP_OPTIMIZATION;
#else
        UINT compileFlags = 0;
#endif

        ThrowIfFailed(D3DCompileFromFile(GetAssetFullPath(L"shaders.hlsl").c_str(), nullptr, nullptr,
                      "VSMain", "vs_5_0", compileFlags, 0, &vertexShader, nullptr));
        ThrowIfFailed(D3DCompileFromFile(GetAssetFullPath(L"shaders.hlsl").c_str(), nullptr, nullptr,
                      "PSMain", "ps_5_0", compileFlags, 0, &pixelShader, nullptr));

        // Define the vertex input layout.
        D3D12_INPUT_ELEMENT_DESC inputElementDescs[] = {
            { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0, D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0 },
            { "COLOR", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0, 12, D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0 }
        };

        // Describe and create the graphics pipeline state object (PSO).
        D3D12_GRAPHICS_PIPELINE_STATE_DESC psoDesc = {};
        psoDesc.InputLayout = { inputElementDescs, _countof(inputElementDescs) };
        psoDesc.pRootSignature = m_rootSignature.Get();
        psoDesc.VS = { reinterpret_cast<UINT8*>(vertexShader->GetBufferPointer()),
                       vertexShader->GetBufferSize() };
        psoDesc.PS = { reinterpret_cast<UINT8*>(pixelShader->GetBufferPointer()),
                       pixelShader->GetBufferSize() };
        psoDesc.RasterizerState = CD3DX12_RASTERIZER_DESC(D3D12_DEFAULT);
        psoDesc.BlendState = CD3DX12_BLEND_DESC(D3D12_DEFAULT);
        psoDesc.DepthStencilState.DepthEnable = FALSE;
        psoDesc.DepthStencilState.StencilEnable = FALSE;
        psoDesc.SampleMask = UINT_MAX;
        psoDesc.PrimitiveTopologyType = D3D12_PRIMITIVE_TOPOLOGY_TYPE_TRIANGLE;
        psoDesc.NumRenderTargets = 1;
        psoDesc.RTVFormats[0] = DXGI_FORMAT_R8G8B8A8_UNORM;
        psoDesc.SampleDesc.Count = 1;
        ThrowIfFailed(m_device->CreateGraphicsPipelineState(&psoDesc, IID_PPV_ARGS(&m_pipelineState)));
    }

    // Create the command list.
    ThrowIfFailed(m_device->CreateCommandList(0, D3D12_COMMAND_LIST_TYPE_DIRECT, m_commandAllocator.Get(),
                  m_pipelineState.Get(), IID_PPV_ARGS(&m_commandList)));

    // Command lists are created in the recording state, but there is nothing
    // to record yet. The main loop expects it to be closed, so close it now.
    ThrowIfFailed(m_commandList->Close());

    // Create the vertex buffer.
    {
        // Define the geometry for a triangle.
        Vertex triangleVertices[] = {
            { { 0.0f, 0.25f * m_aspectRatio, 0.0f },{ 1.0f, 0.0f, 0.0f, 1.0f } },
            { { 0.25f, -0.25f * m_aspectRatio, 0.0f },{ 0.0f, 1.0f, 0.0f, 1.0f } },
            { { -0.25f, -0.25f * m_aspectRatio, 0.0f },{ 0.0f, 0.0f, 1.0f, 1.0f } }
        };

        const UINT vertexBufferSize = sizeof(triangleVertices);

        // Note: using upload heaps to transfer static data like vert buffers is not
        // recommended. Every time the GPU needs it, the upload heap will be marshalled
        // over. Please read up on Default Heap usage. An upload heap is used here for
        // code simplicity and because there are very few verts to actually transfer.
        ThrowIfFailed(m_device->CreateCommittedResource(
            &CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_UPLOAD),
            D3D12_HEAP_FLAG_NONE,
            &CD3DX12_RESOURCE_DESC::Buffer(vertexBufferSize),
            D3D12_RESOURCE_STATE_GENERIC_READ,
            nullptr,
            IID_PPV_ARGS(&m_vertexBuffer)));

        // Copy the triangle data to the vertex buffer.
        UINT8* pVertexDataBegin;
        ThrowIfFailed(m_vertexBuffer->Map(0, nullptr, reinterpret_cast<void**>(&pVertexDataBegin)));
        memcpy(pVertexDataBegin, triangleVertices, sizeof(triangleVertices));
        m_vertexBuffer->Unmap(0, nullptr);

        // Initialize the vertex buffer view.
        m_vertexBufferView.BufferLocation = m_vertexBuffer->GetGPUVirtualAddress();
        m_vertexBufferView.StrideInBytes = sizeof(Vertex);
        m_vertexBufferView.SizeInBytes = vertexBufferSize;
    }

    // Create synchronization objects and wait until assets have been uploaded to the GPU.
    {
        ThrowIfFailed(m_device->CreateFence(0, D3D12_FENCE_FLAG_NONE, IID_PPV_ARGS(&m_fence)));
        m_fenceValue = 1;

        // Create an event handle to use for frame synchronization.
        m_fenceEvent = CreateEventEx(nullptr, FALSE, FALSE, EVENT_ALL_ACCESS);
        if (m_fenceEvent == nullptr) {
            ThrowIfFailed(HRESULT_FROM_WIN32(GetLastError()));
        }

        // Wait for the command list to execute; we are reusing the same command
        // list in our main loop but for now, we just want to wait for setup to
        // complete before continuing.
        WaitForPreviousFrame();
    }
}
```

OnUpdate()

For a simple example, nothing is updated.

C++

```
void D3D12HelloTriangle::OnUpdate() {
}
```

OnRender()

During set up, the member variable **m_commandList** was used to record and execute all of the set up commands. You can now reuse that member in the main render loop.

Rendering involves a call to populate the command list, then the command list can be executed and the next buffer in the swap chain presented:

- Populate the command list.
- Execute the command list: ID3D12CommandQueue::ExecuteCommandLists.
- IDXGISwapChain::Present the frame.
- Wait on the GPU to finish.

C++

```cpp
void D3D12HelloTriangle::OnRender() {
    // Record all the commands we need to render the scene into the command list.
    PopulateCommandList();

    // Execute the command list.
    ID3D12CommandList* ppCommandLists[] = { m_commandList.Get() };
    m_commandQueue->ExecuteCommandLists(_countof(ppCommandLists), ppCommandLists);

    // Present the frame.
    ThrowIfFailed(m_swapChain->Present(1, 0));

    WaitForPreviousFrame();
}
```

PopulateCommandList()

You must reset the command list allocator and the command list itself before you can reuse them. In more advanced scenarios it might make sense to reset the allocator every several frames. Memory is associated with the allocator that can't be released immediately after executing a command list. This example shows how to reset the allocator after every frame.

Now, reuse the command list for the current frame. Reattach the viewport to the command list (which must be done whenever a command list is reset, and before the command list is executed), indicate that the resource will be in use as a render target, record commands, and then indicate that the render target will be used to present when the command list is done executing.

Populating command lists calls the following methods and processes in turn:

- Reset the command allocator, and command list:
  - ID3D12CommandAllocator::Reset
  - ID3D12GraphicsCommandList::Reset
- Set the root signature, viewport and scissors rectangles:
  - ID3D12GraphicsCommandList::SetGraphicsRootSignature
  - ID3D12GraphicsCommandList::RSSetViewports
  - ID3D12GraphicsCommandList::RSSetScissorRects
- Indicate that the back buffer is to be used as a render target:
  - ID3D12GraphicsCommandList::ResourceBarrier
  - ID3D12DescriptorHeap::GetCPUDescriptorHandleForHeapStart
  - ID3D12GraphicsCommandList::OMSetRenderTargets
- Record the commands:
  - ID3D12GraphicsCommandList::ClearRenderTargetView
  - ID3D12GraphicsCommandList::IASetPrimitiveTopology
  - ID3D12GraphicsCommandList::IASetVertexBuffers

    - ID3D12GraphicsCommandList::DrawInstanced

- Indicate the back buffer will now be used to present: ID3D12GraphicsCommandList::ResourceBarrier.
- Close the command list: ID3D12GraphicsCommandList::Close.

C++

```cpp
void D3D12HelloTriangle::PopulateCommandList() {
    // Command list allocators can only be reset when the associated
    // command lists have finished execution on the GPU; apps should use
    // fences to determine GPU execution progress.
    ThrowIfFailed(m_commandAllocator->Reset());

    // However, when ExecuteCommandList() is called on a particular command
    // list, that command list can then be reset at any time and must be before
    // re-recording.
    ThrowIfFailed(m_commandList->Reset(m_commandAllocator.Get(), m_pipelineState.Get()));

    // Set necessary state.
    m_commandList->SetGraphicsRootSignature(m_rootSignature.Get());
    m_commandList->RSSetViewports(1, &m_viewport);
    m_commandList->RSSetScissorRects(1, &m_scissorRect);

    // Indicate that the back buffer will be used as a render target.
    m_commandList->ResourceBarrier(
       1,
       &CD3DX12_RESOURCE_BARRIER::Transition(m_renderTargets[m_frameIndex].Get(),
                                              D3D12_RESOURCE_STATE_PRESENT,
                                              D3D12_RESOURCE_STATE_RENDER_TARGET));

    CD3DX12_CPU_DESCRIPTOR_HANDLE rtvHandle(m_rtvHeap->GetCPUDescriptorHandleForHeapStart(), m_frameIndex,
                                            m_rtvDescriptorSize);
    m_commandList->OMSetRenderTargets(1, &rtvHandle, FALSE, nullptr);

    // Record commands.
    const float clearColor[] = { 0.0f, 0.2f, 0.4f, 1.0f };
    m_commandList->ClearRenderTargetView(rtvHandle, clearColor, 0, nullptr);
    m_commandList->IASetPrimitiveTopology(D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST);
    m_commandList->IASetVertexBuffers(0, 1, &m_vertexBufferView);
    m_commandList->DrawInstanced(3, 1, 0, 0);

    // Indicate that the back buffer will now be used to present.
    m_commandList->ResourceBarrier(
       1,
       &CD3DX12_RESOURCE_BARRIER::Transition(m_renderTargets[m_frameIndex].Get(),
                                              D3D12_RESOURCE_STATE_RENDER_TARGET,
                                              D3D12_RESOURCE_STATE_PRESENT));

    ThrowIfFailed(m_commandList->Close());
}
```

WaitForPreviousFrame()

The following code shows an over-simplified use of fences.

Note Waiting for a frame to complete is too inefficient for most apps.

The following APIs and processes are called in order:

- ID3D12CommandQueue::Signal
- ID3D12Fence::GetCompletedValue
- ID3D12Fence::SetEventOnCompletion
- Wait for the event.
- Update the frame index: IDXGISwapChain3::GetCurrentBackBufferIndex.

C++

```
void D3D12HelloTriangle::WaitForPreviousFrame() {
    // WAITING FOR THE FRAME TO COMPLETE BEFORE CONTINUING IS NOT BEST PRACTICE.
    // This is code implemented as such for simplicity. More advanced samples
    // illustrate how to use fences for efficient resource usage.

    // Signal and increment the fence value.
    const UINT64 fence = m_fenceValue;
    ThrowIfFailed(m_commandQueue->Signal(m_fence.Get(), fence));
    m_fenceValue++;

    // Wait until the previous frame is finished.
    if (m_fence->GetCompletedValue() < fence) {
        ThrowIfFailed(m_fence->SetEventOnCompletion(fence, m_fenceEvent));
        WaitForSingleObject(m_fenceEvent, INFINITE);
    }

    m_frameIndex = m_swapChain->GetCurrentBackBufferIndex();
}
```

OnDestroy()

Cleanly close down the app.

- Wait for the GPU to finish.
- Close the event.

C++

```
void D3D12HelloTriangle::OnDestroy() {
    // Wait for the GPU to be done with all resources.
    WaitForPreviousFrame();
    CloseHandle(m_fenceEvent);
}
```

For working samples, refer to Working Samples.

Important Changes from Direct3D 11 to Direct3D 12

Direct3D 12 represents a significant departure from the Direct3D 11 programming model. Direct3D 12 lets apps get closer to hardware than ever before. By being closer to hardware, Direct3D 12 is faster and more efficient. But, the trade-off of your app having increased speed and efficiency with Direct3D 12 is that you are responsible for more tasks than you were with Direct3D 11.

Direct3D 12 is a return to low-level programming; it gives you more control over the graphical elements of your games and apps by introducing these new features: objects to represent the overall state of the pipeline, command lists and bundles for work submission, and descriptor heaps and tables for resource access.

Direct3D 12 versus Direct3D 11 trade-offs

Your app has increased speed and efficiency with Direct3D 12, but you are responsible for more tasks than you were with Direct3D 11.

- In Direct3D 12, CPU-GPU synchronization is now the explicit responsibility of the app and is no longer implicitly performed by the runtime, as it is in Direct3D 11. This fact also means that no automatic checking for pipeline hazards is performed by Direct3D 12, so again this is the apps responsibility.
- In Direct3D 12, apps are responsible for pipelining data updates. That is, the "Map/Lock-DISCARD" pattern in Direct3D 11 must be performed manually in Direct3D 12. In Direct3D 11, if the GPU is still using the buffer when you call ID3D11DeviceContext::Map with D3D11_MAP_WRITE_DISCARD, the

runtime returns a pointer to a new region of memory instead of the old buffer data. This allows the GPU to continue using the old data while the app places data in the new buffer. No additional memory management is required in the app; the old buffer is reused or destroyed automatically when the GPU is finished with it.

- In Direct3D 12, all dynamic updates (including constant buffers, dynamic vertex buffers, dynamic textures, and so on) are explicitly controlled by the app. These dynamic updates include any required GPU fences or buffering. The app is responsible for keeping the memory available until it is no longer needed.
- Direct3D 12 uses COM-style reference counting only for the lifetimes of interfaces (by using the weak reference model of Direct3D tied to the lifetime of the device). All resource and description memory lifetimes are the sole responsibly of the app to maintain for the proper duration, and are not reference counted. Direct3D 11 uses reference counting to manage the lifetimes of interface dependencies as well.

Pipeline state objects

Direct3D 11 allows pipeline state manipulation through a large set of independent objects. For example, input assembler state, pixel shader state, rasterizer state, and output merger state can all be independently modified. This design provides a convenient and relatively high-level representation of the graphics pipeline, but it doesn’t utilize the capabilities of modern hardware, primarily because the various states are often interdependent. For example, many GPUs combine pixel shader and output merger state into a single hardware representation. But because the Direct3D 11 API allows these pipeline stages to be set separately, the display driver can't resolve issues of pipeline state until the state is finalized, which isn’t until draw time. This scheme delays hardware state setup, which means extra overhead and fewer maximum draw calls per frame.

Direct3D 12 addresses this scheme by unifying much of the pipeline state into immutable pipeline state objects (PSOs), which are finalized upon creation. Hardware and drivers can then immediately convert the PSO into whatever hardware native instructions and state are required to execute GPU work. You can still dynamically change which PSO is in use, but to do so, the hardware only needs to copy the minimal amount of pre-computed state directly to the hardware registers, rather than computing the hardware state on the fly. By using PSOs, draw call overhead is reduced significantly, and many more draw calls can occur per frame. For more information about PSOs, see **Managing graphics pipeline state in Direct3D 12**.

Command lists and bundles

In Direct3D 11, all work submission is done via the **immediate context**, which represents a single stream of commands that go to the GPU. To achieve multithreaded scaling, games also have **deferred contexts** available to them. Deferred contexts in Direct3D 11 don't map perfectly to hardware, so relatively little work can be done in them.

Direct3D 12 introduces a new model for work submission based on command lists that contain the entirety of information needed to execute a particular workload on the GPU. Each new command list contains information such as which PSO to use, what texture and buffer resources are needed, and the arguments to all draw calls. Because each command list is self-contained and inherits no state, the driver can pre-compute all necessary GPU commands up-front and in a free-threaded manner. The only serial process necessary is the final submission of command lists to the GPU via the command queue.

In addition to command lists, Direct3D 12 also introduces a second level of work pre-computation: *bundles*. Unlike command lists, which are completely self-contained and are typically constructed, submitted once, and discarded, bundles provide a form of state inheritance that permits reuse. For example, if a game wants to draw two character models with different textures, one approach is to record a command list with two sets of identical draw calls. But another approach is to "record" one bundle that draws a single character model, then "play back" the bundle twice on the command list using different resources. In the latter case, the display driver only has to compute the appropriate instructions once, and creating the command list essentially amounts to two low-cost function calls.

For more information about command lists and bundles, see **Work Submission in Direct3D 12**.

Descriptor heaps and tables

Resource binding in Direct3D 11 is highly abstracted and convenient, but leaves many modern hardware capabilities underutilized. In Direct3D 11, games create *view* objects of resources, then bind those views to several *slots* at various shader stages in the pipeline. Shaders, in turn, read data from those explicit bind slots, which are fixed at draw time. This model means that whenever a game will draw using different resources, it must re-bind different views to different slots, and call draw again. This case also represents overhead that can be eliminated by fully utilizing modern hardware capabilities.

Direct3D 12 changes the binding model to match modern hardware and significantly improves performance. Instead of requiring standalone resource views and explicit mapping to slots, Direct3D 12 provides a descriptor heap into which games create their various resource views. This scheme provides a mechanism for the GPU to directly write the hardware-native resource description (descriptor) to memory up-front. To declare which resources are to be used by the pipeline for a particular draw call, games specify one or more descriptor tables that represent sub-ranges of the full descriptor heap. As the descriptor heap has already been populated with the appropriate hardware-specific descriptor data, changing descriptor tables is an extremely low-cost operation.

In addition to the improved performance offered by descriptor heaps and tables, Direct3D 12 also allows resources to be dynamically indexed in shaders, which provides unprecedented flexibility and unlocks new rendering techniques. As an example, modern deferred rendering engines typically encode a material or object identifier of some kind to the intermediate g-buffer. In Direct3D 11, these engines must be careful to avoid using too many materials, as including too many in one g-buffer can significantly slow down the final render pass. With dynamically indexable resources, a scene with a thousand materials can be finalized just as quickly as one with only ten.

For more information about descriptor heaps and tables, see **Resource Binding**.

Hardware Feature Levels

Describes the functionality of the 12_0 and 12_1 hardware feature levels.

- **Background**
- **Feature Level Support**
- **Hardware support for DXGI Formats**
- **Related topics**

Background

To handle the diversity of video cards in new and existing machines, Microsoft Direct3D 11 introduced the concept of feature levels. Each video card implements a certain level of Microsoft DirectX (DX) functionality depending on the graphics processing units (GPUs) installed. A feature level is a well defined set of GPU functionality. For instance, the 11_0 feature level implements the functionality that was implemented in Direct3D 11.

Now when you create a device, you can attempt to create a device for the feature level that you want to request. If the device creation works, that feature level exists, if not, the hardware does not support that feature level. You can either try to recreate a device at a lower feature level or you can choose to exit the application.

The basic properties of feature levels are:

- All Direct3D 12 drivers will be Feature Level 11.0 or better.
- A GPU that allows a device to be created meets or exceeds the functionality of that feature level.
- A feature level always includes the functionality of previous or lower feature levels.
- A feature level does not imply performance, only functionality. Performance is dependent on hardware implementation.
- A feature level is chosen when you call D3D12CreateDevice.
- For more detailed information on supported features (especially those marked *Optional* in the table below, which means that the hardware might support the feature but is not required to) call CheckFeatureSupport.

For information about limitations creating non-hardware type devices on certain feature levels, see Limitations Creating WARP and Reference Devices. For more information on the introduction of feature levels, refer to the Direct3D 11 documentation on Direct3D feature levels.

Note that hardware feature levels are *not* the same as API versions (there is a D3D11.3 API, but there is no 11_3 hardware feature level). Feature levels are defined in the D3D_FEATURE_LEVEL enum.

Feature Level Support

The following table lists the features available per feature level:

	12.1	12.0	11_1	11_0
Shader Model	5.1	5.1	5.1	5.1
Resource Binding Tier	Tier2+	Tier2+	Tier1+	Tier1+
Tiled Resources	Tier2+	Tier2+	Optional	Optional
Conservative Rasterization	Tier1+	Optional	Optional	No
Rasterizer Ordered Views	Yes	Optional	Optional	No
Min/Max Filters	Yes	Yes	Optional	No
Map Default Buffer	Optional	Optional	Optional	Optional
Shader Specified Stencil Reference Value	Optional	Optional	Optional	No
Typed Unordered Access View Loads	18 formats, more optional	18 formats, more optional	3 formats, more optional	3 formats, more optional
Volume Tiled Resources	Yes	Yes	No	No
Geometry Shader	Yes	Yes	Yes	Yes
Stream Out	Yes	Yes	Yes	Yes

DirectCompute / Compute Shader	Yes	Yes	Yes	Yes
Hull and Domain Shaders	Yes	Yes	Yes	Yes
Texture Resource Arrays	Yes	Yes	Yes	Yes
Cubemap Resource Arrays	Yes	Yes	Yes	Yes
BC1 to BC7 Compression	Yes	Yes	Yes	Yes
Alpha-to-coverage	Yes	Yes	Yes	Yes
Logic Operations (Output Merger)	Yes	Yes	Yes	Optional
Target-independent rasterization	Yes	Yes	Yes	No
Multiple render target(MRT) with ForcedSampleCount 1	Yes	Yes	Yes	Optional
Max forced sample count for UAV-only rendering	16	16	16	8
Max Texture Dimension	16384	16384	16384	16384
Max Cubemap Dimension	16384	16384	16384	16384
Max Volume Extent	2048	2048	2048	2048
Max Texture Repeat	16384	16384	16384	16384
Max Anisotropy	16	16	16	16
Max Primitive Count	2^32 – 1	2^32 – 1	2^32 – 1	2^32 – 1
Max Vertex Index	2^32 – 1	2^32 – 1	2^32 – 1	2^32 – 1
Max Input Slots	32	32	32	32
Simultaneous Render Targets	8	8	8	8
Occlusion Queries	Yes	Yes	Yes	Yes
Separate Alpha Blend	Yes	Yes	Yes	Yes
Mirror Once	Yes	Yes	Yes	Yes
Overlapping Vertex Elements	Yes	Yes	Yes	Yes
Independent Write Masks	Yes	Yes	Yes	Yes
Instancing	Yes	Yes	Yes	Yes

- Feature levels 12.0 and 12.1 require the Direct3D 11.3 or Direct3D 12 runtime.
- Feature level 11.1 requires the Direct3D 11.1 runtime.
- Feature level 11.0 requires the Direct3D 11.0 runtime.

Hardware support for DXGI Formats

To view tables of DXGI formats and hardware features, refer to:

- DXGI Format Support for Direct3D Feature Level 12.1 Hardware
- DXGI Format Support for Direct3D Feature Level 12.0 Hardware
- DXGI Format Support for Direct3D Feature Level 11.1 Hardware
- DXGI Format Support for Direct3D Feature Level 11.0 Hardware
- Hardware Support for Direct3D 10Level9 Formats
- Hardware Support for Direct3D 10.1 Formats
- Hardware Support for Direct3D 10 Formats

Work Submission in Direct3D 12

To improve the CPU efficiency of Direct3D apps, Direct3D 12 no longer supports an immediate context associated with a device. Instead, apps record and then submit *command lists*, which contain drawing and resource management calls. These command lists can be submitted from multiple threads to one or more command queues, which manage the execution of the commands. This fundamental change increases single-threaded efficiency by allowing apps to pre-compute rendering work for later re-use, and it takes advantage of multi-core systems by spreading rendering work across multiple threads.

In this section

Topic	Description
Design Philosophy of Command Queues and Command Lists	The goals of enabling reuse of rendering work and of multi-threaded scaling, required fundamental changes to how Direct3D apps submit rendering work to the GPU.
Creating and Recording Command Lists and Bundles	This topic describes recording command lists and bundles in Direct3D 12 apps. Command lists and bundles both allow apps to record drawing or state-changing calls for later execution on the graphics processing unit (GPU).
Executing and Synchronizing Command Lists	In Microsoft Direct3D 12, the immediate mode of previous versions is no longer present. Instead, apps create command lists and bundles and then record sets of GPU commands. Command queues are used to submit command lists to be executed. This model allows developers to have more control over the efficient usage of both GPU and CPU.
Managing Graphics Pipeline State in Direct3D 12	This topic describes how graphics pipeline state is set in Direct3D 12.
Using Resource Barriers to Synchronize Resource States in Direct3D 12	To reduce overall CPU usage and enable driver multi-threading and pre-processing, Direct3D 12 moves the responsibility of per-resource state management from the graphics driver to the application.
Pipelines and Shaders with Direct3D 12	The Direct3D 12 programmable pipeline significantly increases rendering performance compared to previous generation graphics programming interfaces.

Design Philosophy of Command Queues and Command Lists

The goals of enabling reuse of rendering work and of multi-threaded scaling, required fundamental changes to how Direct3D apps submit rendering work to the GPU. In Direct3D 12, the process of submitting rendering work differs from earlier versions in three important ways:

1. Elimination of the immediate context. This enables multi-threading.

2. Apps now own how rendering calls are grouped into graphics processing unit (GPU) work items. This enables re-use.

3. Apps now explicitly control when work is submitted to the GPU . This enables items 1 and 2.

Removal of the immediate context

The largest change from Microsoft Direct3D 11 to Microsoft Direct3D 12 is that there is no longer a single, immediate context associated with a device. Instead, to render, apps create command lists in which traditional rendering APIs can be called. A command list looks similar to the render method of a Direct3D 11 app that used the immediate context in that it contains calls that draw primitives or change the rendering state. Like immediate contexts, each command list is not free-threaded; however, multiple command lists can be recorded concurrently, which takes advantage of modern, multi-core processors.

Command lists are typically executed once. However, a command list can be executed multiple times if the application ensures that the previous executions are complete before submitting new executions. For more information about command list synchronization, see **Executing and synchronizing command lists**.

Grouping of GPU work items

In addition to command lists, Direct3D 12 takes advantage of functionality present in all hardware today by adding a second level of command lists, which are called *bundles*. To help to distinguish these two types, the first-level command lists can be referred to as *direct command lists*. The purpose of bundles is to allow apps to group a small number of API commands together for later, repeated execution from within direct command lists. At the time of creating a bundle, the driver will perform as much pre-processing as possible to make later execution efficient. Bundles can then be executed from within multiple command lists and multiple times within the same command list.

The re-use of bundles is a large driver of improved efficiency with single CPU threads. Because bundles are pre-processed and can be submitted multiple times, there are certain restrictions on what operations can be performed within a bundle. For more information, see **Creating and recording command lists and bundles**.

GPU work submission

To execute work on the GPU, an app must explicitly submit a command list to a command queue associated with the Direct3D device. A direct command list can be submitted for execution multiple times, but the app is responsible for ensuring that the direct command list has finished executing on the GPU before submitting it again. Bundles have no concurrent-use restrictions and can be executed multiple times in multiple command lists, but bundles cannot be directly submitted to a command queue for execution.

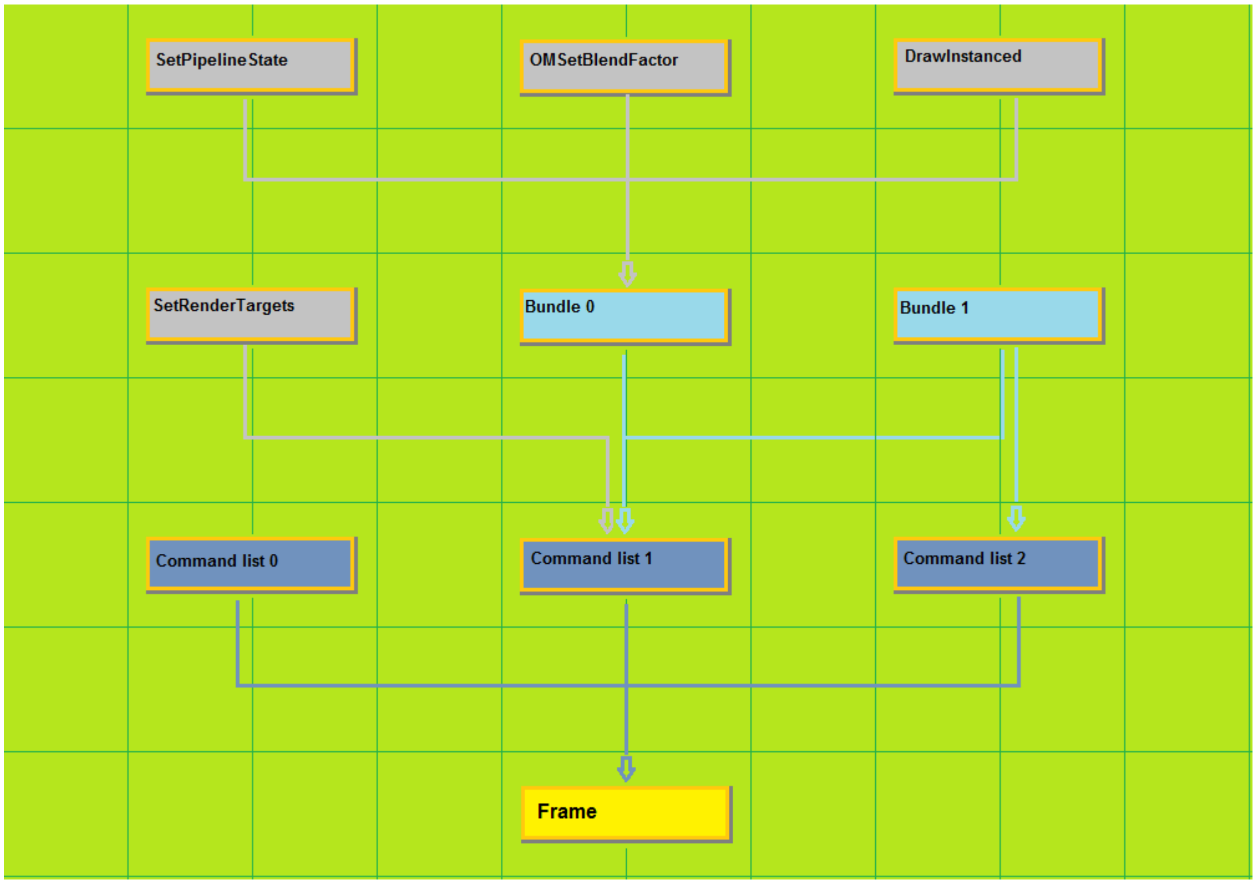
Any thread may submit a command list to any command queue at any time, and the runtime will automatically serialize submission of the command list in the command queue while preserving the submission order.

Creating and Recording Command Lists and Bundles

This topic describes recording command lists and bundles in Direct3D 12 apps. Command lists and bundles both allow apps to record drawing or state-changing calls for later execution on the graphics processing unit (GPU).

Beyond command lists, the API exploits functionality present in GPU hardware by adding a second level of command lists, which are referred to as *bundles*. The purpose of bundles is to allow apps to group a small number of API commands together for later execution. At bundle creation time, the driver will perform as much pre-processing as is possible to make these cheap to execute later. Bundles are designed to be used and re-used any number of times. Command lists, on the other hand, are typically executed only a single time. However, a command list *can* be executed multiple times (as long as the application ensures that the previous executions have completed before submitting new executions).

Typically though, the build up of API calls into bundles, and API calls and bundles into command lists, and command lists into a single frame, is shown in the following diagram, noting the reuse of **Bundle 1** in**Command list 1** and **Command list 2**, and that the API method names in the diagram are just as examples, many different API calls can be used.



There are different restrictions for creating and executing bundles and direct command lists, and these differences are noted throughout this topic.

Creating command lists

Direct command lists and bundles are created by calling **ID3D12Device::CreateCommandList**. This method takes the following parameters as input:

D3D12_COMMAND_LIST_TYPE

The **D3D12_COMMAND_LIST_TYPE** enumeration indicates the type of command list that is being created. It can be a direct command list, a bundle, a compute command list, or a copy command list.

ID3D12CommandAllocator

A command allocator allows the app to manage the memory that is allocated for command lists. The command allocator is created by calling**CreateCommandAllocator**. When creating a command list, the command list type of the allocator, specified by **D3D12_COMMAND_LIST_TYPE**, must match the type of command list being created, and a given allocator can be associated no more than one currently recording command list at a time.

To reclaim the memory allocated by a command allocator, an app calls **ID3D12CommandAllocator::Reset**. But before doing so, the app must make sure that the GPU is no longer executing any command lists which are associated with the allocator; otherwise, the call will fail. Also, note that this API is not free-threaded and therefore can't be called on the same allocator at the same time from multiple threads.

ID3D12PipelineState

The initial pipeline state for the command list. In Microsoft Direct3D 12, most graphics pipeline state is set within a command list using the**ID3D12PipelineState** object. An app will create a large number of these, typically during app initialization, and then the state is updated by changing the currently bound state object using **ID3D12GraphicsCommandList::SetPipelineState**. For more information about pipeline state objects, see **Managing graphics pipeline state in Direct3D 12**.

Note that bundles don't inherit the pipeline state set by previous calls in direct command lists that are their parents.

If this parameter is NULL, a default state is used.

ID3D12DescriptorHeap

The **ID3D12DescriptorHeap** allows command lists to bind resources to the graphics pipeline. Direct command lists must specify an initial descriptor heap, but may change the currently-bound descriptor heap inside the command list by calling **ID3D12GraphicsCommandList::SetDescriptorHeap**.

Specifying a descriptor heap at creation time is optional for bundles. If a descriptor heap is not specified, however, the application is not allowed to set any descriptor tables within that bundle. Either way, bundles are not permitted to change the descriptor heap within a bundle. If a heap is specified for a bundle, it must match the currently bound heap in the direct command list that is the calling parent.

For more information, refer to **Descriptor Heaps**.

Recording command lists

Immediately after being created, command lists are in the recording state. You can also re-use an existing command list by calling I**D3D12GraphicsCommandList::Reset**, which also leaves the command list in the recording state. Unlike **ID3D12CommandAllocator::Reset**, you can call**Reset** while the command list is still being executed. A typical pattern is to submit a command list and then immediately reset it to reuse the allocated memory for another command list. Note that only one command list associated with each command allocator may be in a recording state at one time.

Once a command list is in the recording state, you simply call methods of the **ID3D12GraphicsCommandList** interface to add commands to the list. Many of these methods enable common Direct3D functionality that will be familiar to Microsoft Direct3D 11 developers; other APIs are new for Direct3D 12.

After adding commands to the command list, you transition the command list out of the recording state by calling **Close**.

Example

The following code snippets illustrate the creation and recording of a command list. Note that this example includes the following Direct3D 12 features:

- Pipeline state objects - These are used to set most of the state parameters of the render pipeline from within a command list. For more information, see**Managing graphics pipeline state in Direct3D 12**.
- Descriptor heap - Apps use descriptor heaps to manage pipeline binding to memory resources.
- Resource barrier - This is used to manage the transition of resources from one state to another, such as from a render target view to a shader resource view. For more information, see **Using resource barriers to synchronize resource states**.

For example,

C++

```cpp
void D3D12HelloTriangle::LoadAssets() {
    // Create an empty root signature.
    {
        CD3DX12_ROOT_SIGNATURE_DESC rootSignatureDesc;
        rootSignatureDesc.Init(0, nullptr, 0, nullptr,
                              D3D12_ROOT_SIGNATURE_FLAG_ALLOW_INPUT_ASSEMBLER_INPUT_LAYOUT);

        ComPtr<ID3DBlob> signature;
        ComPtr<ID3DBlob> error;
        ThrowIfFailed(D3D12SerializeRootSignature(&rootSignatureDesc, D3D_ROOT_SIGNATURE_VERSION_1,
                                                  &signature, &error));
        ThrowIfFailed(m_device->CreateRootSignature(0, signature->GetBufferPointer(),
                      signature->GetBufferSize(), IID_PPV_ARGS(&m_rootSignature)));
    }

    // Create the pipeline state, which includes compiling and loading shaders.
    {
        ComPtr<ID3DBlob> vertexShader;
        ComPtr<ID3DBlob> pixelShader;

#ifdef _DEBUG
        // Enable better shader debugging with the graphics debugging tools.
        UINT compileFlags = D3DCOMPILE_DEBUG | D3DCOMPILE_SKIP_OPTIMIZATION;
#else
        UINT compileFlags = 0;
#endif

        ThrowIfFailed(D3DCompileFromFile(GetAssetFullPath(L"shaders.hlsl").c_str(), nullptr, nullptr,
                                         "VSMain", "vs_5_0", compileFlags, 0, &vertexShader, nullptr));
        ThrowIfFailed(D3DCompileFromFile(GetAssetFullPath(L"shaders.hlsl").c_str(), nullptr, nullptr,
                                         "PSMain", "ps_5_0", compileFlags, 0, &pixelShader, nullptr));

        // Define the vertex input layout.
        D3D12_INPUT_ELEMENT_DESC inputElementDescs[] = {
        { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0, D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0 },
        { "COLOR", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0, 12, D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0 }
        };

        // Describe and create the graphics pipeline state object (PSO).
        D3D12_GRAPHICS_PIPELINE_STATE_DESC psoDesc = {};
        psoDesc.InputLayout = { inputElementDescs, _countof(inputElementDescs) };
        psoDesc.pRootSignature = m_rootSignature.Get();
        psoDesc.VS = { reinterpret_cast<UINT8*>(vertexShader->GetBufferPointer()),
                       vertexShader->GetBufferSize() };
        psoDesc.PS = { reinterpret_cast<UINT8*>(pixelShader->GetBufferPointer()),
                       pixelShader->GetBufferSize() };
        psoDesc.RasterizerState = CD3DX12_RASTERIZER_DESC(D3D12_DEFAULT);
        psoDesc.BlendState = CD3DX12_BLEND_DESC(D3D12_DEFAULT);
        psoDesc.DepthStencilState.DepthEnable = FALSE;
        psoDesc.DepthStencilState.StencilEnable = FALSE;
        psoDesc.SampleMask = UINT_MAX;
        psoDesc.PrimitiveTopologyType = D3D12_PRIMITIVE_TOPOLOGY_TYPE_TRIANGLE;
        psoDesc.NumRenderTargets = 1;
        psoDesc.RTVFormats[0] = DXGI_FORMAT_R8G8B8A8_UNORM;
        psoDesc.SampleDesc.Count = 1;
        ThrowIfFailed(m_device->CreateGraphicsPipelineState(&psoDesc, IID_PPV_ARGS(&m_pipelineState)));
    }

    // Create the command list.
    ThrowIfFailed(m_device->CreateCommandList(0, D3D12_COMMAND_LIST_TYPE_DIRECT, m_commandAllocator.Get(),
                  m_pipelineState.Get(), IID_PPV_ARGS(&m_commandList)));

    // Command lists are created in the recording state, but there is nothing
    // to record yet. The main loop expects it to be closed, so close it now.
    ThrowIfFailed(m_commandList->Close());

    // Create the vertex buffer.
    {
        // Define the geometry for a triangle.
        Vertex triangleVertices[] = {
            { { 0.0f, 0.25f * m_aspectRatio, 0.0f }, { 1.0f, 0.0f, 0.0f, 1.0f } },
            { { 0.25f, -0.25f * m_aspectRatio, 0.0f }, { 0.0f, 1.0f, 0.0f, 1.0f } },
            { { -0.25f, -0.25f * m_aspectRatio, 0.0f }, { 0.0f, 0.0f, 1.0f, 1.0f } }
        };

        const UINT vertexBufferSize = sizeof(triangleVertices);

        // Note: using upload heaps to transfer static data like vert buffers is not
        // recommended. Every time the GPU needs it, the upload heap will be marshalled
        // over. Please read up on Default Heap usage. An upload heap is used here for
        // code simplicity and because there are very few verts to actually transfer.
        ThrowIfFailed(m_device->CreateCommittedResource(
            &CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_UPLOAD),
            D3D12_HEAP_FLAG_NONE,
            &CD3DX12_RESOURCE_DESC::Buffer(vertexBufferSize),
            D3D12_RESOURCE_STATE_GENERIC_READ,
            nullptr,
            IID_PPV_ARGS(&m_vertexBuffer)));

        // Copy the triangle data to the vertex buffer.
        UINT8* pVertexDataBegin;
        ThrowIfFailed(m_vertexBuffer->Map(0, nullptr, reinterpret_cast<void**>(&pVertexDataBegin)));
        memcpy(pVertexDataBegin, triangleVertices, sizeof(triangleVertices));
        m_vertexBuffer->Unmap(0, nullptr);

        // Initialize the vertex buffer view.
        m_vertexBufferView.BufferLocation = m_vertexBuffer->GetGPUVirtualAddress();
        m_vertexBufferView.StrideInBytes = sizeof(Vertex);
        m_vertexBufferView.SizeInBytes = vertexBufferSize;
    }

    // Create synchronization objects and wait until assets have been uploaded to the GPU.
    {
        ThrowIfFailed(m_device->CreateFence(0, D3D12_FENCE_FLAG_NONE, IID_PPV_ARGS(&m_fence)));
        m_fenceValue = 1;

        // Create an event handle to use for frame synchronization.
        m_fenceEvent = CreateEventEx(nullptr, FALSE, FALSE, EVENT_ALL_ACCESS);
        if (m_fenceEvent == nullptr) {
            ThrowIfFailed(HRESULT_FROM_WIN32(GetLastError()));
        }

        // Wait for the command list to execute; we are reusing the same command
        // list in our main loop but for now, we just want to wait for setup to
        // complete before continuing.
        WaitForPreviousFrame();
    }
}
```

After a command list has been created and recorded, it can be executed using a command queue. For more information, see Executing and synchronizing command lists.

Reference counting

Most D3D12 APIs continue to use reference counting following COM conventions. A notable exception to this is the D3D12 graphics command list APIs. All APIs on ID3D12GraphicsCommandList do not hold references to the objects passed into those APIs. This means applications are responsible for ensuring that a command list is never submitted for execution that references a destroyed resource.

Command list errors

Most APIs on ID3D12GraphicsCommandList do not return errors. Errors encountered during command list creation are deferred untilID3D12GraphicsCommandList::Close. The one exception is DXGI_ERROR_DEVICE_REMOVED, which is deferred even further. Note that this is different from D3D11, where many parameter validation errors are silently dropped and never returned to the caller.

Applications can expect to see DXGI_DEVICE_REMOVED errors in the following API calls:

- Any resource creation method
- ID3D12Resource::Map
- Present
- GetDeviceRemovedReason

Bundle restrictions

Restrictions enable Direct3D 12 drivers to do most of the work associated with bundles at record time, thus enabling the ExecuteBundle API to be run with low overhead. All pipeline state objects referenced by a bundle must have the same render target formats, depth buffer format, and sample descriptions.

The following command list API calls are not allowed on command lists created with type: D3D12_COMMAND_LIST_TYPE_BUNDLE:

- Any Clear method
- Any Copy method
- DiscardResource
- ExecuteBundle
- ResourceBarrier
- ResolveSubresource
- SetPredication
- BeginQuery
- EndQuery
- SOSetTargets
- OMSetRenderTargets
- RSSetViewports
- RSSetScissorRects

SetDescriptorHeaps can be called on a bundle, but the bundle descriptor heaps must match the calling command list descriptor heap.

If any of these APIs are called on a bundle, the runtime will drop the call. The debug layer will issue an error whenever this occurs.

Executing and Synchronizing Command Lists

In Microsoft Direct3D 12, the immediate mode of previous versions is no longer present. Instead, apps create command lists and bundles and then record sets of GPU commands. Command queues are used to submit command lists to be executed. This model allows developers to have more control over the efficient usage of both graphics processing unit (GPU) and CPU.

Command queue overview

Direct3D 12 command queues replace hidden runtime and driver synchronization of immediate mode work submission with APIs for explicitly managing concurrency, parallelism and synchronization. Command queues provide the following improvements for developers:

- Allows developers to avoid accidental inefficiencies caused by unexpected synchronization.
- Allows developers to introduce synchronization at a higher level where the required synchronization can be determined more efficiently and accurately. This means the runtime and the graphics driver will spend less time reactively engineering parallelism.
- Makes expensive operations more explicit.

These improvements enable or enhance the following scenarios:

- Increased parallelism - Applications can use deeper queues for background workloads, such as video decoding, when they have separate queues for foreground work.
- Asynchronous and low-priority GPU work - The command queue model enables concurrent execution of low-priority GPU work and atomic operations that enable one GPU thread to consume the results of another unsynchronized thread without blocking.
- High-priority compute work - This design enables scenarios that require interrupting 3D rendering to do a small amount of high-priority compute work so that the result can be obtained early for additional processing on the CPU.

Initializing a command queue

When the Direct3D graphics device is initialized, a default command queue is created. Most applications should use the default queue for rendering because it allows for shared usage by other components.

For applications that need to support additional concurrency scenarios, additional queues can be created by calling **ID3D12Device::CreateCommandQueue**. This method takes a **D3D12_COMMAND_LIST_TYPE** indicating what type of queue should be created, and therefore, what type of commands can be submitted to that queue. Remember that bundles can only be called from direct command lists and can't be submitted directly to a queue. The supported queue types are:

- D3D12_COMMAND_LIST_TYPE_DIRECT
- D3D12_COMMAND_LIST_TYPE_COMPUTE
- D3D12_COMMAND_LIST_TYPE_COPY

In general, DIRECT queues and command lists accept any command, COMPUTE queues and command lists accept compute and copy related commands, and COPY queues and command lists accept only copy commands.

Executing command Lists

After you have recorded a command list and either retrieved the default command queue or created a new one, you execute command lists by calling**ID3D12CommandQueue::ExecuteCommandLists**.

Applications can submit command lists to any command queue from multiple threads. The runtime will perform the work of serializing these requests in the order of submission.

The runtime will validate the submitted command list and will drop the call to **ExecuteCommandLists** if any of the restrictions are violated. Calls will be dropped for the following reasons:

- The supplied command list is a bundle, not a direct command list.
- ID3D12GraphicsCommandList::Close has not been called on the supplied command list to complete recording.
- ID3D12CommandAllocator::Reset has been called on the command allocator associated with the command list since it was recorded. For more information about command allocators, see Creating and recording command lists and bundles.
- The command queue fence indicates that a previous execution of the command list has not yet completed. Command queue fences are discussed in detail below.
- The before and after states of queries, set with calls to ID3D12GraphicsCommandList::BeginQuery and ID3D12GraphicsCommandList::EndQuery, are not matched properly.
- The before and after states of resource transition barriers are not matched properly. For more information, see Using resource barriers to synchronize resource states.

Accessing resources from multiple command queues

There are a few rules imposed by the runtime that restrict the access of resources from multiple command queues. These rules are as follows:

1. A resource can be accessed for read and write from multiple command queues simultaneously, including across processes, only if it is in the stateD3D12_RESOURCE_STATE_UNORDERED_ACCESS.

2. A resource can be read from multiple command queues simultaneously, including across processes, only if it is in the stateD3D12_RESOURCE_STATE_GENERIC_READ.

3. All write operations, except for the unordered access case described in case 1 above, must be done exclusively by a single command queue at a time. When a resource has transitioned to a writeable state on a queue, it is considered exclusively owned by that queue and it must transition toD3D12_RESOURCE_STATE_GENERIC_READ before it can be accessed by another queue.

For more information about resource access restrictions and using resource barriers to synchronize access to resources, see Using resource barriers to synchronize resource states.

Synchronizing command list execution using command queue fences

The support for multiple parallel command queues in Direct3D 12 gives you more flexibility and control over the prioritization of asynchronous work on the GPU. This design also means that apps need to explicitly manage the synchronization of work, especially when the command lists in one queue depend on resources that are being operated on by another command queue. Some examples of this include waiting for an operation on a compute queue to complete so that the result can be used for a rendering operation on the 3D queue, and waiting for a 3D operation to complete so that an operation on the compute queue can access a resource for writing. To enable the synchronization of work between queues, Direct3D 12 uses the concept of fences, which are represented in the API by the ID3D12Fence interface.

A fence is an integer that represents the current unit of work being processed. When the app advances the fence, by calling ID3D12CommandQueue::Signal, the integer is updated. Apps can check the value of a fence and determine if a unit of work has been completed in order to decide whether a subsequent operation can be started.

Synchronizing resources accessed by command queues

In Direct3D 12, synchronizing the state of some resources is implemented with resource barriers. At each resource barrier, an app declares the before and after states of a resource. A common example is for a resource to transition between a shader resource view to a render target view. For the most part, these resource barriers are managed within command lists. When the debug layers are enabled, the system enforces that the before and after states of all resources match up, guaranteeing that the resource is in the correct state for a particular operation at a barrier transition.

For more information about synchronizing resource state, see **Using resource barriers to synchronize resource states**.

Command queue support for tiled resources

Methods for managing tiled resources, which were exposed through the **ID3D11DeviceContext2** interface in Direct3D 11, are provided by the**ID3D12CommandQueue** interface in Direct3D 12. These methods include:

Method	Description
CopyTileMappings	Copies mappings from a source tiled resource to a destination tiled resource.
UpdateTileMappings	Updates mappings of tile locations in tiled resources to memory locations in a resource heap.

Managing Graphics Pipeline State in Direct3D 12

This topic describes how graphics pipeline state is set in Direct3D 12.

- **Pipeline state overview**
- **Graphics pipeline states set with pipeline state objects**
- **Graphics pipeline states set outside of the pipeline state object**
- **Graphics pipeline state inheritance**
- **Related topics**

Pipeline state overview

When geometry is submitted to the graphics processing unit (GPU) to be drawn, there are a wide range of hardware settings that determine how the input data is interpreted and rendered. Collectively, these settings are called the graphics pipeline state and include common settings such as the rasterizer state, blend state, and depth stencil state, as well as the primitive topology type of the submitted geometry and the shaders that will be used for rendering. In Microsoft Direct3D 12, most graphics pipeline state is set by using pipeline state objects (PSO). An app can create an unlimited number of these objects, represented by the **ID3D12PipelineState** interface, typically at initialization time. Then, at render time, command lists can quickly switch multiple settings of the pipeline state by calling **ID3D12GraphicsCommandList::SetPipelineState** in a direct command list or bundle to set the active PSO.

In Direct3D 11, graphics pipeline state was bundled into large, coarse-grained state objects like **ID3D11BlendState** that could be created and set at render time in the immediate context with methods like **ID3D11DeviceContext::OMSetBlendState**. The idea behind this was that the GPU could gain efficiencies by setting related settings, like the blend state settings, all at once. However, with today's graphics hardware, there are dependencies between the different hardware units. For example, the hardware blend state might have dependencies on the render state as well as the blend state. PSOs in Direct3D 12 were designed to allow the GPU to pre-process all of the dependent settings in each pipeline state, typically during initialization, to make switching between states at render time as efficient as possible.

Note that while most of the pipeline state settings are set using PSOs, there are some state settings which are set separately using APIs provided byID3D12GraphicsCommandList. These settings and the associated APIs are discussed in detail below. Also, there are differences in the way that graphics pipeline state is inherited by and persisted from direct command lists and bundles. This topic provides details on both of these below.

Graphics pipeline states set with pipeline state objects

The easiest way to see all of the different pipeline states that can be set using a pipeline state object is to look at the reference topic for theD3D12_GRAPHICS_PIPELINE_STATE_DESC which you pass to ID3D12Device::CreateGraphicsPipelineState when you initialize the object. A quick summary of the states that can be set includes:

- The bytecode for all shaders including, vertex, pixel, domain, hull, and geometry shaders.
- The input vertex format.
- The primitive topology type. Note that the input-assembler primitive topology type (point, line, triangle, patch) is set within the PSO using theD3D12_PRIMITIVE_TOPOLOGY_TYPE enumeration. The primitive adjacency and ordering (line list, line strip, line strip with adjacency data, etc.) is set from within a command list using the ID3D12GraphicsCommandList::IASetPrimitiveTopology method.
- The blend state, rasterizer state, depth stencil state.
- The depth stencil and render target formats, as well as the render target count.
- Multi-sampling parameters.
- A streaming output buffer.
- The root signature. For more information, see Root Signatures.

Graphics pipeline states set outside of the pipeline state object

Most graphics pipeline states are set using PSOs. However, there are a set of pipeline state parameters that are set by calling methods of theID3D12GraphicsCommandList interface from within a command list. The following table shows the states that are set this way and the corresponding methods.

State	Method
Resource bindings	IASetIndexBuffer IASetVertexBuffers SOSetTargets OMSetRenderTargets SetDescriptorHeaps All **SetGraphicsRoot...** methods All **SetComputeRoot...** methods
Viewports	RSSetViewports
Scissor rects	RSSetScissorRects
Blend factor	OMSetBlendFactor
The depth stencil reference value	OMSetStencilRef
The input-assembler primitive topology order and adjacency type.	IASetPrimitiveTopology

Graphics pipeline state inheritance

Because direct command lists are generally intended for one use at a time and bundles are intended to be used multiple times concurrently, there are different rules about how they inherit graphics pipeline state that was set by previous command lists or bundles.

For the graphics pipeline states that are set using PSOs, none of these states are inherited by either direct command lists or bundles. The initial graphics pipeline state for both direct command lists and bundles is set at creation time with the ID3D12PipelineState parameter toID3D12Device::CreateCommandList. If no PSO is specified in the call, a default initial state is used. You can change the current PSO within a command list by calling ID3D12GraphicsCommandList::SetPipelineState.

Direct command lists also do not inherit state that is set with command list methods like RSSetViewports. For details about the default initial values for non-PSO states, see ID3D12GraphicsCommandList::ClearState.

Bundles inherit all graphics pipeline state that is not set with PSOs except for the primitive topology type. The primitive topology is always set toD3D12_PRIMITIVE_TOPOLOGY_TYPE_UNDEFINED when a bundle begins executing. Any non-PSO state that is set within a bundle affects the state of its parent direct command list. For example, if a RSSetViewports is called from within a bundle, the specified viewports will continue to be set in the parent direct command list for calls subsequent to the ExecuteBundle call that set the viewports.

Resource bindings that are set within a command list or bundle do persist. So resource bindings modified in a direct command list will still be set within subsequent child bundle execution. And resource bindings modified from within a bundle will still be set for subsequent calls within the parent direct command list.

For more information on bindings, refer to the **Bundle Semantics** section of Using a Root Signature.

Using Resource Barriers to Synchronize Resource States in Direct3D 12

To reduce overall CPU usage and enable driver multi-threading and pre-processing, Direct3D 12 moves the responsibility of per-resource state management from the graphics driver to the application. An example of per-resource state is whether a texture resource is currently being accessed as through a Shader Resource View or as a Render Target View. In Direct3D 11, drivers were required to track this state in the background. This is expensive from a CPU perspective and significantly complicates any sort of multi-threaded design. In Microsoft Direct3D 12, most per-resource state is managed by the application with a single API, ID3D12GraphicsCommandList::ResourceBarrier.

Using the ResourceBarrier API to manage per-resource state

ResourceBarrier notifies the graphics driver of situations in which the driver may need to synchronize multiple accesses to the memory in which a resource is stored. The method is called with one or more resource barrier description structures indicating the type of resource barrier being declared.

There are three types of resource barriers:

- Transition barrier - A transition barrier indicates that a set of subresources transition between different usages. AD3D12_RESOURCE_TRANSITION_BARRIER structure is used to specify the subresource that is transitioning as well as the before and after states of the subresource.

- The system verifies that subresource transitions in a command list are consistent with previous transitions in the same command list. The debug layer further tracks subresource state to find other errors however this validation is conservative and not exhaustive.
- Note that you can use the D3D12_RESOURCE_BARRIER_ALL_SUBRESOURCES flag to specify that all subresources within a resource are being transitioned.
- Aliasing barrier - An aliasing barrier indicates a transition between usages of two different resources which have mappings into the same tile pool. AD3D12_RESOURCE_ALIASING_BARRIER structure is used to specify the before resource and the after resource.
- Note that one or both resources can be NULL, which indicates that any tiled resource could cause aliasing. For more information about using tiled resources, see Tiled resources and Volume Tiled Resources.
- Unordered access view (UAV) barrier - A UAV barrier indicates that all UAV accesses, both read or write, to a particular resource must complete between any future UAV accesses, both read or write. It's not necessary for an app to put a UAV barrier between two draw or dispatch calls that only read from a UAV. Also, it's not necessary to put a UAV barrier between two draw or dispatch calls that write to the same UAV if the application knows that it is safe to execute the UAV access in any order. A D3D12_RESOURCE_UAV_BARRIER structure is used to specify the UAV resource to which the barrier applies. The application can specify NULL for the barrier's UAV, which indicates that any UAV access could require the barrier.

When ResourceBarrier is called with an array of resource barrier descriptions, the API behaves as if it was called once for each element, in the order in which they were supplied.

At any given time, a subresource is in exactly one state, determined by the set of D3D12_RESOURCE_STATES flags supplied to ResourceBarrier. The application must ensure that the before and after states of consecutive calls to **ResourceBarrier** agree.

Tip

Applications should batch multiple transitions into one API call wherever possible.

Resource states

For the complete list of resource states that a resource can transition between, see the reference topic for the D3D12_RESOURCE_STATES enumeration.

For split resource barriers, also refer to D3D12_RESOURCE_BARRIER_FLAGS.

Initial states for resources

Resources can have a user-specified initial state. All resources have no resource usage states set by the system when they are initialized except for the following exception:

- Dynamic heaps start out in the state D3D12_RESOURCE_STATE_GENERIC_READ which is a bitwise OR combination of:
  - D3D12_RESOURCE_STATE_VERTEX_AND_CONSTANT_BUFFER
  - D3D12_RESOURCE_STATE_INDEX_BUFFER
  - D3D12_RESOURCE_STATE_COPY_SOURCE
  - D3D12_RESOURCE_STATE_NON_PIXEL_SHADER_RESOURCE

- D3D12_RESOURCE_STATE_PIXEL_SHADER_RESOURCE
- D3D12_RESOURCE_STATE_INDIRECT_ARGUMENT

Before a heap can be the target of a GPU copy operation, normally the heap must first be transitioned to the D3D12_RESOURCE_STATE_COPY_DEST state. However, resources created on UPLOAD heaps must start in and cannot change from the GENERIC_READ state. Committed resources created in READBACK heaps must start in and cannot change from the COPY_DEST state.

Read/write resource state restrictions

The resource state usage bits that are used to describe a resource state are divided into read-only and read/write states. The reference topic for the**D3D12_RESOURCE_STATES** indicates the read/write access level for each bit in the enumeration.

At most, only one read/write bit can be set for any resource. If a write bit is set, then no read-only bit may be set for that resource. If no write bit is set, then any number of read bits may be set.

Resource states for presenting back buffers

Before a back buffer is presented, it must be in the D3D12_RESOURCE_STATE_COMMON state. If one of the **IDXGISwapChain** **Present** methods is called on a resource that is not currently in this state, the runtime will transition the resource to D3D12_RESOURCE_STATE_COMMON on behalf of the application. A debug layer warning is emitted in this case.

The resource state D3D12_RESOURCE_STATE_PRESENT is a synonym for D3D12_RESOURCE_STATE_COMMON, and they both have a value of 0.

Discarding resources

A subresource can be in any state when the following method is called:

ID3D12GraphicsCommandList::DiscardResource

Resource barrier example scenario

The following snippets show the use of the **ResourceBarrier** method in a multi-threading sample.

Creating the depth stencil view, transitioning it to a writeable state.

C++

```cpp
// Create the depth stencil.
{
    CD3DX12_RESOURCE_DESC shadowTextureDesc(
        D3D12_RESOURCE_DIMENSION_TEXTURE2D,
        0,
        static_cast<UINT>(m_viewport.Width),
        static_cast<UINT>(m_viewport.Height),
        1,
        1,
        DXGI_FORMAT_D32_FLOAT,
        1,
        0,
        D3D12_TEXTURE_LAYOUT_UNKNOWN,
        D3D12_RESOURCE_FLAG_ALLOW_DEPTH_STENCIL | D3D12_RESOURCE_FLAG_DENY_SHADER_RESOURCE);

    D3D12_CLEAR_VALUE clearValue; // Performance tip: Tell the runtime at resource creation the desired clear
value.
    clearValue.Format = DXGI_FORMAT_D32_FLOAT;
    clearValue.DepthStencil.Depth = 1.0f;
    clearValue.DepthStencil.Stencil = 0;
    ThrowIfFailed(m_device->CreateCommittedResource(
        &CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_DEFAULT),
        D3D12_HEAP_FLAG_NONE,
        &shadowTextureDesc,
        D3D12_RESOURCE_STATE_DEPTH_WRITE,
        &clearValue,
        IID_PPV_ARGS(&m_depthStencil)));

    // Create the depth stencil view.
    m_device->CreateDepthStencilView(m_depthStencil.Get(), nullptr,
                                     m_dsvHeap->GetCPUDescriptorHandleForHeapStart());
}
```

Creating the vertex buffer view, first changing it from a common state to a destination, then from a destination to a generic readable state.

C++

```
// Create the vertex buffer.
{
    ThrowIfFailed(m_device->CreateCommittedResource(
        &CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_DEFAULT),
        D3D12_HEAP_FLAG_NONE,
        &CD3DX12_RESOURCE_DESC::Buffer(SampleAssets::VertexDataSize),
        D3D12_RESOURCE_STATE_COPY_DEST,
        nullptr,
        IID_PPV_ARGS(&m_vertexBuffer)));

    {
        ThrowIfFailed(m_device->CreateCommittedResource(
            &CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_UPLOAD),
            D3D12_HEAP_FLAG_NONE,
            &CD3DX12_RESOURCE_DESC::Buffer(SampleAssets::VertexDataSize),
            D3D12_RESOURCE_STATE_GENERIC_READ,
            nullptr,
            IID_PPV_ARGS(&m_vertexBufferUpload)));

        // Copy data to the upload heap and then schedule a copy
        // from the upload heap to the vertex buffer.
        D3D12_SUBRESOURCE_DATA vertexData = {};
        vertexData.pData = pAssetData + SampleAssets::VertexDataOffset;
        vertexData.RowPitch = SampleAssets::VertexDataSize;
        vertexData.SlicePitch = vertexData.RowPitch;

        PIXBeginEvent(commandList.Get(), 0, L"Copy vertex buffer data to default resource...");

        UpdateSubresources<1>(commandList.Get(), m_vertexBuffer.Get(), m_vertexBufferUpload.Get(), 0, 0, 1,
                              &vertexData);
        commandList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(m_vertexBuffer.Get(),
                                     D3D12_RESOURCE_STATE_COPY_DEST,
                                     D3D12_RESOURCE_STATE_VERTEX_AND_CONSTANT_BUFFER));

        PIXEndEvent(commandList.Get());
    }
```

Creating the index buffer view, first changing it from a common state to a destination, then from a destination to a generic readable state.

C++

```
// Create the index buffer.
{
    ThrowIfFailed(m_device->CreateCommittedResource(
        &CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_DEFAULT),
        D3D12_HEAP_FLAG_NONE,
        &CD3DX12_RESOURCE_DESC::Buffer(SampleAssets::IndexDataSize),
        D3D12_RESOURCE_STATE_COPY_DEST,
        nullptr,
        IID_PPV_ARGS(&m_indexBuffer)));

    {
        ThrowIfFailed(m_device->CreateCommittedResource(
            &CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_UPLOAD),
            D3D12_HEAP_FLAG_NONE,
            &CD3DX12_RESOURCE_DESC::Buffer(SampleAssets::IndexDataSize),
            D3D12_RESOURCE_STATE_GENERIC_READ,
            nullptr,
            IID_PPV_ARGS(&m_indexBufferUpload)));

        // Copy data to the upload heap and then schedule a copy
        // from the upload heap to the index buffer.
        D3D12_SUBRESOURCE_DATA indexData = {};
        indexData.pData = pAssetData + SampleAssets::IndexDataOffset;
        indexData.RowPitch = SampleAssets::IndexDataSize;
        indexData.SlicePitch = indexData.RowPitch;

        PIXBeginEvent(commandList.Get(), 0, L"Copy index buffer data to default resource...");

        UpdateSubresources<1>(commandList.Get(), m_indexBuffer.Get(), m_indexBufferUpload.Get(), 0, 0, 1,
                              &indexData);
        commandList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(m_indexBuffer.Get(),
                                     D3D12_RESOURCE_STATE_COPY_DEST, D3D12_RESOURCE_STATE_INDEX_BUFFER));

        PIXEndEvent(commandList.Get());
    }

    // Initialize the index buffer view.
    m_indexBufferView.BufferLocation = m_indexBuffer->GetGPUVirtualAddress();
    m_indexBufferView.SizeInBytes = SampleAssets::IndexDataSize;
    m_indexBufferView.Format = SampleAssets::StandardIndexFormat;
}
```

Creating textures and shader resource views. The texture is changed from a common state to a destination, and then from a destination to a pixel shader resource.

C++

```
    // Create each texture and SRV descriptor.
    const UINT srvCount = _countof(SampleAssets::Textures);
    PIXBeginEvent(commandList.Get(), 0, L"Copy diffuse and normal texture data to default resources...");
    for (int i = 0; i < srvCount; i++) {
        // Describe and create a Texture2D.
        const SampleAssets::TextureResource &tex = SampleAssets::Textures[i];
        CD3DX12_RESOURCE_DESC texDesc(
            D3D12_RESOURCE_DIMENSION_TEXTURE2D,
            0,
            tex.Width,
            tex.Height,
            1,
            static_cast<UINT16>(tex.MipLevels),
            tex.Format,
            1,
            0,
            D3D12_TEXTURE_LAYOUT_UNKNOWN,
            D3D12_RESOURCE_FLAG_NONE);

        ThrowIfFailed(m_device->CreateCommittedResource(
            &CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_DEFAULT),
            D3D12_HEAP_FLAG_NONE,
            &texDesc,
            D3D12_RESOURCE_STATE_COPY_DEST,
            nullptr,
            IID_PPV_ARGS(&m_textures[i])));

        {
            const UINT subresourceCount = texDesc.DepthOrArraySize * texDesc.MipLevels;
            UINT64 uploadBufferSize = GetRequiredIntermediateSize(m_textures[i].Get(), 0, subresourceCount);
            ThrowIfFailed(m_device->CreateCommittedResource(
                &CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_UPLOAD),
                D3D12_HEAP_FLAG_NONE,
                &CD3DX12_RESOURCE_DESC::Buffer(uploadBufferSize),
                D3D12_RESOURCE_STATE_GENERIC_READ,
                nullptr,
                IID_PPV_ARGS(&m_textureUploads[i])));

            // Copy data to the intermediate upload heap and then schedule a copy
            // from the upload heap to the Texture2D.
            D3D12_SUBRESOURCE_DATA textureData = {};
            textureData.pData = pAssetData + tex.Data->Offset;
            textureData.RowPitch = tex.Data->Pitch;
            textureData.SlicePitch = tex.Data->Size;

            UpdateSubresources(commandList.Get(), m_textures[i].Get(), m_textureUploads[i].Get(), 0, 0,
                               subresourceCount, &textureData);
            commandList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(m_textures[i].Get(),
                                             D3D12_RESOURCE_STATE_COPY_DEST,
                                             D3D12_RESOURCE_STATE_PIXEL_SHADER_RESOURCE));
        }

        // Describe and create an SRV.
        D3D12_SHADER_RESOURCE_VIEW_DESC srvDesc = {};
        srvDesc.ViewDimension = D3D12_SRV_DIMENSION_TEXTURE2D;
        srvDesc.Shader4ComponentMapping = D3D12_DEFAULT_SHADER_4_COMPONENT_MAPPING;
        srvDesc.Format = tex.Format;
        srvDesc.Texture2D.MipLevels = tex.MipLevels;
        srvDesc.Texture2D.MostDetailedMip = 0;
        srvDesc.Texture2D.ResourceMinLODClamp = 0.0f;
        m_device->CreateShaderResourceView(m_textures[i].Get(), &srvDesc, cbvSrvHandle);

        // Move to the next descriptor slot.
        cbvSrvHandle.Offset(cbvSrvDescriptorSize);
    }
    PIXEndEvent(commandList.Get());
}
```

Beginning a frame; this not only uses ResourceBarrier to indicate that the backbuffer is to be used as a render target, but also initializes the frame resource (which calls **ResourceBarrier** on the depth stencil buffer).

C++

```
// Assemble the CommandListPre command list.
void D3D12Multithreading::BeginFrame() {
    m_pCurrentFrameResource->Init();

    // Indicate that the back buffer will be used as a render target.
    m_pCurrentFrameResource->m_commandLists[CommandListPre]->ResourceBarrier(
         1,
         &CD3DX12_RESOURCE_BARRIER::Transition(m_renderTargets[m_frameIndex].Get(),
                                              D3D12_RESOURCE_STATE_PRESENT,
                                              D3D12_RESOURCE_STATE_RENDER_TARGET));

    // Clear the render target and depth stencil.
    const float clearColor[] = { 0.0f, 0.0f, 0.0f, 1.0f };
    CD3DX12_CPU_DESCRIPTOR_HANDLE rtvHandle(m_rtvHeap->GetCPUDescriptorHandleForHeapStart(),
                                           m_frameIndex,
                                           m_rtvDescriptorSize);
    m_pCurrentFrameResource->m_commandLists[CommandListPre]->ClearRenderTargetView(
         rtvHandle,
         clearColor,
         0,
         nullptr);
    m_pCurrentFrameResource->m_commandLists[CommandListPre]->ClearDepthStencilView(
         m_dsvHeap->GetCPUDescriptorHandleForHeapStart(),
         D3D12_CLEAR_FLAG_DEPTH,
         1.0f,
         0,
         0,
         nullptr);

    ThrowIfFailed(m_pCurrentFrameResource->m_commandLists[CommandListPre]->Close());
}
```

Ending a frame, indicating the back buffer is now used to present.

C++

```
// Assemble the CommandListPost command list.
void D3D12Multithreading::EndFrame() {
    m_pCurrentFrameResource->Finish();

    // Indicate that the back buffer will now be used to present.
    m_pCurrentFrameResource->m_commandLists[CommandListPost]->ResourceBarrier(
         1,
         &CD3DX12_RESOURCE_BARRIER::Transition(m_renderTargets[m_frameIndex].Get(),
                                              D3D12_RESOURCE_STATE_RENDER_TARGET,
                                              D3D12_RESOURCE_STATE_PRESENT));

    ThrowIfFailed(m_pCurrentFrameResource->m_commandLists[CommandListPost]->Close());
}
```

Initializing a frame resource, called when beginning a frame, transitions the depth stencil buffer to writeable.

C++

```cpp
void FrameResource::Init() {
    // Reset the command allocators and lists for the main thread.
    for (int i = 0; i < CommandListCount; i++) {
        ThrowIfFailed(m_commandAllocators[i]->Reset());
        ThrowIfFailed(m_commandLists[i]->Reset(m_commandAllocators[i].Get(), m_pipelineState.Get()));
    }

    // Clear the depth stencil buffer in preparation for rendering the shadow map.
    m_commandLists[CommandListPre]->ClearDepthStencilView(
         m_shadowDepthView,
         D3D12_CLEAR_FLAG_DEPTH,
         1.0f,
         0,
         0,
         nullptr);

    // Reset the worker command allocators and lists.
    for (int i = 0; i < NumContexts; i++) {
        ThrowIfFailed(m_shadowCommandAllocators[i]->Reset());
        ThrowIfFailed(m_shadowCommandLists[i]->Reset(
            m_shadowCommandAllocators[i].Get(),
            m_pipelineStateShadowMap.Get()));

        ThrowIfFailed(m_sceneCommandAllocators[i]->Reset());
        ThrowIfFailed(m_sceneCommandLists[i]->Reset(
            m_sceneCommandAllocators[i].Get(),
            m_pipelineState.Get()));
    }
}
```

Barriers are swapped mid-frame, transitioning the shadow map from writeable to readable.

C++

```cpp
void FrameResource::SwapBarriers() {
    // Transition the shadow map from writeable to readable.
    m_commandLists[CommandListMid]->ResourceBarrier(
         1,
         &CD3DX12_RESOURCE_BARRIER::Transition(m_shadowTexture.Get(), D3D12_RESOURCE_STATE_DEPTH_WRITE,
                                              D3D12_RESOURCE_STATE_PIXEL_SHADER_RESOURCE));
}
```

Finish is called when a frame is ended, transitioning the shadow map to a common state.

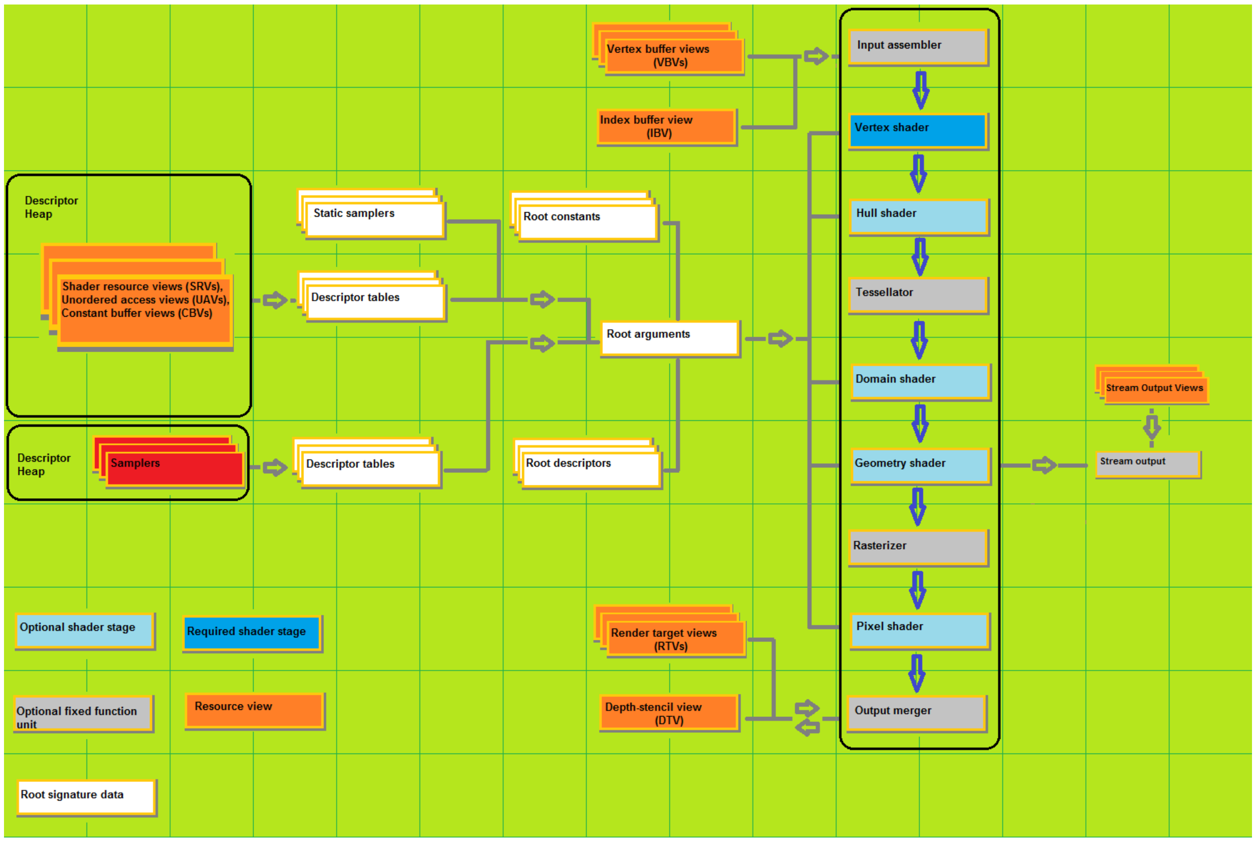
C++

```cpp
void FrameResource::Finish() {
    m_commandLists[CommandListPost]->ResourceBarrier(
         1,
         &CD3DX12_RESOURCE_BARRIER::Transition(m_shadowTexture.Get(),
                                              D3D12_RESOURCE_STATE_PIXEL_SHADER_RESOURCE,
                                              D3D12_RESOURCE_STATE_DEPTH_WRITE));
}
```

Pipelines and Shaders with Direct3D 12

The Direct3D 12 programmable pipeline significantly increases rendering performance compared to previous generation graphics programming interfaces.

The following diagram illustrates the Direct3D 12 graphics pipeline and state.



A graphics pipeline is the sequential flow of data inputs and outputs as the GPU renders frames. Given the pipeline state and inputs, the GPU performs a series of operations to create the resulting images. A graphics pipeline contains shaders, which perform programmable rendering effects and calculations, and fixed function operations.

Note the following when referring to the pipeline state diagram:

- The descriptor tables and heaps can be arbitrarily arranged: SRVs, CBVs and UAVs can be referenced and allocated in any order.
- Some operations of the pipeline are configurable. For example, the output merger typically operates on a read-modify-write basis with the render target and depth stencil views. However the pipeline can be configured so that either of these views are read-only or write-only.

Direct3D 12 introduces the pipeline state object (PSO). Rather than storing and representing pipeline state across a large number of high-level objects, the states of pipeline components like the input assembler, rasterizer, pixel shader, and output merger are stored in a PSO. A PSO is a unified pipeline state object that is immutable after creation. The currently selected PSO can be changed quickly and dynamically, and the hardware and drivers can directly convert a PSO into native hardware instructions and state, readying the GPU for graphics processing. To apply a PSO, the hardware copies a minimal amount of pre-computed state directly to the hardware registers. This removes the overhead caused by the graphics driver continually recomputing hardware state based on all currently applicable rendering and pipeline settings. The result is significantly reduced draw call overhead, increased performance, and more draw calls per frame.

The currently applied PSO defines and connects all of the shaders being used in the rendering pipeline. **Microsoft High Level Shader Language (HLSL)** is pre-compiled into shader objects, which are then used at run time as input for pipeline state objects. For more information about how the PSO functions within the graphics pipeline, see **Managing graphics pipeline state in Direct3D 12**.

Resource Binding

Binding is the process of linking resource objects to the graphics pipeline. The key to resource binding in DirectX 12 are the concepts of a *descriptor*,*descriptor tables*, *descriptor heaps*, and a *root signature*.

- **Overview of Resource Binding**
- **Resource Binding Flow of Control**
- **Suballocation**
- **Freeing Resources**
- **Related topics**

Overview of Resource Binding

Shader resources (such as textures, constant tables, images, buffers and so on) are not bound directly to the shader pipeline; instead, they are referenced through a *descriptor*. A descriptor is a small object that contains information about one resource.

Descriptors are grouped together to form *descriptor tables*. Each descriptor table stores information about one range of types of resource. There are many different types of resources. The most common resources are:

- Constant buffer views (CBVs)
- Unordered access views (UAVs)
- Shader resource views (SRVs)
- Samplers

SRV, UAV, and CBVs descriptors can be combined into the same descriptor table.

The graphics and compute pipelines gain access to resources by referencing into descriptor tables by index.

Descriptor tables are stored in a *descriptor heap*. Descriptor heaps will ideally contain all the descriptors (in descriptor tables) for one or more frames to be rendered. All the resources will be stored in user mode heaps.

Another concept is that of a *root signature*. The root signature is a binding convention, defined by the application, that is used by shaders to locate the resources that they need access to. The root signature can store:

- Indexes to descriptor tables in a descriptor heap, where the layout of the descriptor table has been pre-defined.
- Constants, so apps can bind user-defined constants (known as *root constants*) directly to shaders without having to go through descriptors and descriptor tables.
- A very small number of descriptors directly inside the root signature, such as a constant buffer view (CBV) that changes per draw, thereby saving the application from needing to put those descriptors in a descriptor heap.

In other words, the root signature provides performance optimizations suitable for small amounts of data that change per draw.

The DirectX 12 design for binding separates it from other tasks, such as memory management, object lifetime management, state tracking, and memory synchronization. This is discussed in the Binding Model section.

Resource Binding Flow of Control

Focusing just on root signatures, root descriptors, root constants, descriptor tables, and descriptor heaps, the flow of rendering logic for an app should be similar to the following:

- Create one or more root signature objects – one for every different binding configuration an application needs.
- Create shaders and pipeline state with the root signature objects they will be used with.
- Create one (or, if necessary, more) descriptor heaps that will contain all the SRV, UAV, and CBV descriptors for each frame of rendering.
- Initialize the descriptor heap(s) with descriptors where possible for sets of descriptors that will be reused across many frames.
- For each frame to be rendered:
    - For each command list:
        - Set the current root signature to use (and change if needed during rendering – which is rarely required).
        - Update some root signature's constants and/or root signature descriptors for the new view (such as world/view projections).
        - For each item to draw:
            - Define any new descriptors in descriptor heaps as needed for per-object rendering. For shader-visible descriptor heaps, the app must make sure to use descriptor heap space that isn't already being referenced by rendering that could be in flight – for example, linearly allocating space through the descriptor heap during rendering.
            - Update the root signature with pointers to the required regions of the descriptor heaps. For example, one descriptor table might point to some static (unchanging) descriptors initialized earlier, while another descriptor table might point to some dynamic descriptors configured for the current rendering.
            - Update some root signature's constants and/or root signature descriptors for per-item rendering.
            - Set the pipeline state for the item to draw (only if change needed), compatible with the currently bound root signature.
            - Draw
        - Repeat (next item)
    - Repeat (next command list)
    - Strictly when the GPU has finished with any memory that will no longer be used, it can be released. Descriptors' references to it do not need to be deleted if additional rendering that uses those descriptors is not submitted. So, subsequent rendering can point to other areas in descriptor heaps, or stale descriptors can be overwritten with valid descriptors to reuse the descriptor heap space.
- Repeat (next frame)

Note that other descriptor types, render target views (RTVs), depth stencil views (DSV), index buffer views (IBVs), vertex buffer views (VBVs), and shader object views (SOV), are managed differently. The driver handles the versioning of the set of descriptors bound for each draw during recording of the command list (similar to how the root signature bindings are versioned by the hardware/driver). This is different from the contents of shader-visible descriptor heaps, for which the application must manually allocate through the heap as it references different descriptors between draws. Versioning of heap content that is shader-visible is left to the application because it allows applications to do things like reuse descriptors that don't change, or use large static sets of descriptors and use shader indexing (such as by material ID) to select descriptors to use from the descriptor heap, or use combinations of techniques for different sets of descriptors. The hardware isn't equipped to handle this type of flexibility for the other descriptor types (RTV, DSV, IBV, VBV, SOV).

Suballocation

In Direct3D 12, the app has low-level control over memory management. In earlier versions of Direct3D, including Direct3D 11, there would be one allocation per resource. In Direct3D 12, the app can use the API to allocate a large block of memory, larger than any single object would need. After this is done, the app can create descriptors to point to sections of that large memory block. This process of deciding what to put where (smaller blocks inside the large block) is known as *suballocation*. Enabling the app to do this can yield gains in efficient use of computation and memory. For example, resource renaming is rendered obsolete. In place of this, apps can use fences to determine when a particular resource is being used and when it's not by fencing on command list executions where the command list requires the use of that particular resource.

Freeing Resources

Before any memory that has been bound to the pipeline can be freed, the GPU must be finished with it.

Waiting for frame rendering is probably the coarsest way to be certain that the GPU has finished. At a finer grain, you can again use fences—when a command is recorded into a command list that you want to track the completion of, insert a fence immediately after it. Then, you can do various synchronization operations with the fence. You submit new work (command lists) that waits until a specified fence has passed on the GPU, which indicates that everything before it is complete, or you can request that a CPU event be raised when the fence has passed (which the app can be waiting on with a sleeping thread). In Direct3D 11, this was `EnqueueSetEvent()`.

In this section

Topic	Description
Differences in the Binding Model from Direct3D 11	One of the main design decisions behind DirectX12 binding is to separate it from other management tasks. This places some requirements on the app to manage certain potential hazards.
Descriptors	Descriptors are the primary unit of binding for a single resource in D3D12.
Descriptor Heaps	A descriptor heap is a collection of contiguous allocations of descriptors, one allocation for every descriptor. Descriptor heaps contain many object types that are not part of a Pipeline State Object (PSO), such as Shader Resource Views (SRVs), Unordered Access Views (UAVs), Constant Buffer Views (CBVs), and Samplers.
Descriptor Tables	A descriptor table is logically an array of descriptors. Each descriptor table stores descriptors of one or more types - SRVs, UAVe, CBVs, and Samplers. A descriptor table is not an allocation of memory; it is simply an offset and length into a descriptor heap.

Root Signatures	The root signature defines what resources are bound to the graphics pipeline. A root signature is configured by the app and links command lists to the resources the shaders require. Currently, there is one graphics and one compute root signature per app.
Capability Querying	Applications can discover the level of support for resource binding, and many other features, via theID3D12Device::CheckFeatureSupport call.
Resource Binding in HLSL	This section describes some specific features of using High Level Shading Language (HLSL) Shader Model 5.1 with D3D12.
Default Texture Mapping and Standard Swizzle	The use of default texture mapping reduces copying and memory usage while sharing image data between the GPU and the CPU. However, it should only be used in specific situations. The standard swizzle layout avoids copying or swizzling data in multiple layouts.
Typed Unordered Access View Loads	Unordered Access View (UAV) Typed Load is the ability for a shader to read from a UAV with a specific DXGI_FORMAT.
Volume Tiled Resources	Volume (3D) textures can be used as tiled resources, noting that tile resolution is three-dimensional.
Subresources	Describes how a resource is divided into subresources, and how to reference a single, multiple or slice of subresources.

Differences in the Binding Model from Direct3D 11

One of the main design decisions behind DirectX12 binding is to separate it from other management tasks. This places some requirements on the app to manage certain potential hazards.

The main advantage of the D3D12 Binding Model is that it enables apps to change texture bindings frequently, without a huge CPU performance cost. Other benefits are that shaders have access to a very large number of resources, shaders need not know in advance how many resources will be bound, and that a unified resource binding model can be used regardless of hardware or the apps content flow.

To improve performance, the binding model does not require the system to keep track of what bindings an app has requested the GPU to use, and there is a clean integration between binding and multi-threaded command lists.

The following sections list some of the changes to the resource binding model since D3D11.

- Memory Residency Management Separated From Binding
- Object Lifetime Management Separated From Binding
- Driver Resource State Tracking Separated From Binding
- CPU GPU Mapped Memory Synchronization Separated From Binding
- Related topics

Memory Residency Management Separated From Binding

Applications have explicit control over which surfaces they need to be available for the GPU to use directly (called being "resident"). Conversely, they can apply other states on resources such as explicitly making them not resident, or letting the OS choose for certain classes of applications that require a minimal memory footprint. The important point here is that the application's management of what is resident is completely decoupled from how it gives access to resources to shaders.

The decoupling of residency management from the mechanism for giving shaders access to resources reduces the system/hardware cost for rendering since the OS doesn't have to constantly inspect the local binding state to know what to make resident. Furthermore, shaders no longer have to know which exact surfaces they may need to reference, as long as the entire set of possibly accessible resources has been made resident ahead of time.

Object Lifetime Management Separated From Binding

Unlike previous APIs, the system no longer tracks bindings of resources to the pipeline. This used to enable the system to keep alive resources that the application has released because they are still referenced by outstanding GPU work.

Before freeing any resource, such as a texture, applications now must make sure the GPU has completed referencing it. This means before an application can safely free a resource the GPU must have completed execution of the command list referencing the resource.

Driver Resource State Tracking Separated From Binding

The system no longer inspects resource bindings to understand when resource transitions have occurred which require additional driver or GPU work. A common example for many GPUs and drivers is having to know when a surface transitions from being used as a Render Target View (RTV) to Shader Resource View (SRV). Applications themselves must now identify when any resource transitions that the system might care about are happening via dedicated APIs.

CPU GPU Mapped Memory Synchronization Separated From Binding

The system no longer inspects resource bindings to understand if rendering needs to be delayed because it depends on a resource that has been mapped for CPU access but has not been unmapped yet. Applications now have the responsibility to synchronize CPU and GPU memory accesses. To help with this, the system provides mechanisms for the application to request the sleeping of a CPU thread until work completes. Polling could also be done, but can be less efficient.

Descriptors

Descriptors are the primary unit of binding for a single resource in D3D12.

- Null descriptors
- Related topics

A descriptor is a relatively small block of data that fully describes an object to the GPU, in a GPU specific opaque format. There are many different types of descriptors: Shader Resource Views (SRVs), Unordered Access Views (UAVs), Constant Buffer Views (CBVs) and Samplers are a few examples.

Object descriptors do not need to be freed or released. Drivers do not attach any allocations to the creation of a descriptor. A descriptor may, however, encode references to other allocations for which the application owns the lifetime. For instance, a descriptor for an SRV must contain the virtual address of the D3D resource (e.g. a texture) that the SRV refers to. It is the application's responsibility to make sure that it does not use an SRV descriptor when the underlying D3D resource it depends on has been destroyed or is being modified (such as being declared as nonresident).

The primary way to use descriptors is to place them in descriptor heaps, which are backing memory for descriptors.

Null descriptors

When creating descriptors using API calls, applications pass NULL for the resource pointer in the descriptor definition to achieve the effect of an unbound resource.

The rest of the descriptor must be populated as much as possible. For example, in the case of Shader Resource Views (SRVs), the descriptor can be used to distinguish the type of view it is (Texture1D, Texture2D, and so on). Numerical parameters in the view descriptor, such as the number of mipmaps, must all be set to values that are valid for a resource.

In many cases, there is a defined behavior for accessing an unbound resource, such as SRVs which return default values. Those will be honored when accessing a NULL descriptor as long as the type of shader access is compatible with the descriptor type. For example, if a shader expects a Texture2D SRV and accesses a NULL SRV defined as a Texture1D, the behavior is undefined and could result in device reset.

In this section

Topic	Description
Creating Descriptors	Describes and shows examples for creating index, vertex, and constant buffer views; shader resource, render target, unordered access, stream output and depth-stencil views; and samplers. All methods for creating descriptors are free threaded.
Copying Descriptors	The ID3D12Device::CopyDescriptors method on the device interface uses the CPU to immediately copy descriptors. This can be called free threaded as long as multiple threads on the CPU or GPU do not perform any potentially conflicting writes.

Creating Descriptors

Describes and shows examples for creating index, vertex, and constant buffer views; shader resource, render target, unordered access, stream output and depth-stencil views; and samplers. All methods for creating descriptors are free threaded.

- Index Buffer View
- Vertex Buffer View
- Shader Resource View
- Constant Buffer View
- Sampler
- Unordered Access View
- Stream Output View

Index Buffer View

To create an index buffer view, fill out a D3D12_INDEX_BUFFER_VIEW structure:

```
typedef struct D3D12_INDEX_BUFFER_VIEW {
    D3D12_GPU_VIRTUAL_ADDRESS BufferLocation;
    UINT SizeInBytes;
    DXGI_FORMAT Format;
} D3D12_INDEX_BUFFER_VIEW;
```

Set the location (call GetGPUVirtualAddress) and size of the buffer, noting that D3D12_GPU_VIRTUAL_ADDRESS is defined as:

```
typedef UINT64 D3D12_GPU_VIRTUAL_ADDRESS;
```

Refer to the DXGI_FORMAT enum. Typically for an index buffer the following define might be used:

```
const DXGI_FORMAT StandardIndexFormat = DXGI_FORMAT_R32_UINT;
```

Finally call ID3D12GraphicsCommandList::IASetIndexBuffer.

For example,

C++

```
// Initialize the index buffer view.
m_indexBufferView.BufferLocation = m_indexBuffer->GetGPUVirtualAddress();
m_indexBufferView.SizeInBytes = SampleAssets::IndexDataSize;
m_indexBufferView.Format = SampleAssets::StandardIndexFormat;
```

Vertex Buffer View

To create a vertex buffer view, fill out a D3D12_VERTEX_BUFFER_VIEW structure:

```
typedef struct D3D12_VERTEX_BUFFER_VIEW {
    D3D12_GPU_VIRTUAL_ADDRESS BufferLocation;
    UINT SizeInBytes;
    UINT StrideInBytes;
} D3D12_VERTEX_BUFFER_VIEW;
```

Set the location (call GetGPUVirtualAddress) and size of the buffer, noting that D3D12_GPU_VIRTUAL_ADDRESS is defined as:

```
typedef UINT64 D3D12_GPU_VIRTUAL_ADDRESS;
```

The stride is typically the size of a single vertex data structure, such as `sizeof(Vertex);`, then call ID3D12GraphicsCommandList::IASetVertexBuffers.

For example,

C++

```
// Initialize the vertex buffer view.
m_vertexBufferView.BufferLocation = m_vertexBuffer->GetGPUVirtualAddress();
m_vertexBufferView.SizeInBytes = SampleAssets::VertexDataSize;
m_vertexBufferView.StrideInBytes = SampleAssets::StandardVertexStride;
```

Shader Resource View

To create a shader resource view, fill out a D3D12_SHADER_RESOURCE_VIEW_DESC structure:

```
typedef struct D3D12_SHADER_RESOURCE_VIEW_DESC {
    DXGI_FORMAT Format;
    D3D12_SRV_DIMENSION ViewDimension;
    UINT Shader4ComponentMapping;
    union {
        D3D12_BUFFER_SRV Buffer;
        D3D12_TEX1D_SRV Texture1D;
        D3D12_TEX1D_ARRAY_SRV Texture1DArray;
        D3D12_TEX2D_SRV Texture2D;
        D3D12_TEX2D_ARRAY_SRV Texture2DArray;
        D3D12_TEX2DMS_SRV Texture2DMS;
        D3D12_TEX2DMS_ARRAY_SRV Texture2DMSArray;
        D3D12_TEX3D_SRV Texture3D;
        D3D12_TEXCUBE_SRV TextureCube;
        D3D12_TEXCUBE_ARRAY_SRV TextureCubeArray;
    };
} D3D12_SHADER_RESOURCE_VIEW_DESC;
```

The `ViewDimension` field is set to zero, or one value of the D3D12_BUFFER_SRV_FLAGS enum.

The enums and structures referenced by D3D12_SHADER_RESOURCE_VIEW_DESC are:

- DXGI_FORMAT
- D3D12_BUFFER_SRV
- D3D12_TEX1D_SRV
- D3D12_TEX1D_ARRAY_SRV
- D3D12_TEX2D_SRV
- D3D12_TEX2D_ARRAY_SRV
- D3D12_TEX2DMS_SRV
- D3D12_TEX2DMS_ARRAY_SRV
- D3D12_TEX3D_SRV
- D3D12_TEXCUBE_SRV
- D3D12_TEXCUBE_ARRAY_SRV

Note below that float `ResourceMinLODClamp` has been added to SRVs for Tex1D/2D/3D/Cube. In D3D11, it was a property of a resource, but this did not match how it was implemented in hardware. `StructureByteStride` has been added to Buffer SRVs, where in D3D11 it was a property of the resource. If the stride is nonzero, that indicates a structured buffer view, and the format must be set to DXGI_FORMAT_UNKNOWN.

Finally, to create the shader resource view, call ID3D12Device::CreateShaderResourceView.

For example,

C++

```cpp
// Describe and create an SRV.
D3D12_SHADER_RESOURCE_VIEW_DESC srvDesc = {};
srvDesc.ViewDimension = D3D12_SRV_DIMENSION_TEXTURE2D;
srvDesc.Shader4ComponentMapping = D3D12_DEFAULT_SHADER_4_COMPONENT_MAPPING;
srvDesc.Format = tex.Format;
srvDesc.Texture2D.MipLevels = tex.MipLevels;
srvDesc.Texture2D.MostDetailedMip = 0;
srvDesc.Texture2D.ResourceMinLODClamp = 0.0f;
m_device->CreateShaderResourceView(m_textures[i].Get(), &srvDesc, cbvSrvHandle);
```

Constant Buffer View

To create a constant buffer view, fill out a D3D12_CONSTANT_BUFFER_VIEW_DESC structure:

```cpp
typedef struct D3D12_CONSTANT_BUFFER_VIEW_DESC {
    UINT64 OffsetInBytes;
    UINT SizeInBytes;
} D3D12_CONSTANT_BUFFER_VIEW_DESC;
```

Then call ID3D12Device::CreateConstantBufferView.

For example,

C++

```cpp
// Describe and create a constant buffer view.
D3D12_CONSTANT_BUFFER_VIEW_DESC cbvDesc = {};
cbvDesc.BufferLocation = m_constantBuffer->GetGPUVirtualAddress();
cbvDesc.SizeInBytes = (sizeof(ConstantBuffer) + 255) & ~255;    // CB size is required to be 256-byte
aligned.
m_device->CreateConstantBufferView(&cbvDesc, m_cbvHeap->GetCPUDescriptorHandleForHeapStart());
```

Sampler

To create a sample, fill out a D3D12_SAMPLER_DESC structure:

```cpp
typedef struct D3D12_SAMPLER_DESC {
    D3D12_FILTER Filter;
    D3D12_TEXTURE_ADDRESS_MODE AddressU;
    D3D12_TEXTURE_ADDRESS_MODE AddressV;
    D3D12_TEXTURE_ADDRESS_MODE AddressW;
    FLOAT MipLODBias;
    UINT MaxAnisotropy;
    D3D12_COMPARISON_FUNC ComparisonFunc;
    FLOAT BorderColor[4]; // RGBA
    FLOAT MinLOD;
    FLOAT MaxLOD;
} D3D12_SAMPLER_DESC;
```

To fill out this structure, refer to the following enums:

- D3D12_FILTER
- D3D12_FILTER_TYPE
- D3D12_FILTER_REDUCTION_TYPE
- D3D12_TEXTURE_ADDRESS_MODE
- D3D12_COMPARISON_FUNC

Finally, call ID3D12Device::CreateSampler.

For example,

C++

```cpp
// Describe and create a sampler.
D3D12_SAMPLER_DESC samplerDesc = {};
samplerDesc.Filter = D3D12_FILTER_MIN_MAG_MIP_LINEAR;
samplerDesc.AddressU = D3D12_TEXTURE_ADDRESS_MODE_WRAP;
samplerDesc.AddressV = D3D12_TEXTURE_ADDRESS_MODE_WRAP;
samplerDesc.AddressW = D3D12_TEXTURE_ADDRESS_MODE_WRAP;
samplerDesc.MinLOD = 0;
samplerDesc.MaxLOD = D3D12_FLOAT32_MAX;
samplerDesc.MipLODBias = 0.0f;
samplerDesc.MaxAnisotropy = 1;
samplerDesc.ComparisonFunc = D3D12_COMPARISON_FUNC_ALWAYS;
m_device->CreateSampler(&samplerDesc, m_samplerHeap->GetCPUDescriptorHandleForHeapStart());
```

Unordered Access View

To create an unordered access view, fill out a D3D12_UNORDERED_ACCESS_VIEW_DESC structure:

```cpp
typedef struct D3D12_UNORDERED_ACCESS_VIEW_DESC {
    DXGI_FORMAT Format;
    D3D12_UAV_DIMENSION ViewDimension;
    union {
        D3D12_BUFFER_UAV Buffer;
        D3D12_TEX1D_UAV Texture1D;
        D3D12_TEX1D_ARRAY_UAV Texture1DArray;
        D3D12_TEX2D_UAV Texture2D;
        D3D12_TEX2D_ARRAY_UAV Texture2DArray;
        D3D12_TEX3D_UAV Texture3D;
    };
} D3D12_UNORDERED_ACCESS_VIEW_DESC;
```

The `ViewDimension` field is set to zero, or one value of the D3D12_BUFFER_UAV_FLAGS enum.

Refer to the following enums and structures:

- DXGI_FORMAT
- D3D12_BUFFER_UAV
- D3D12_TEX1D_UAV
- D3D12_TEX1D_ARRAY_UAV
- D3D12_TEX2D_UAV
- D3D12_TEX2D_ARRAY_UAV
- D3D12_TEX3D_UAV

`StructureByteStride` has been added to Buffer UAVs, where in D3D11 it was a property of the resource. If the stride is nonzero, that indicates a structured buffer view, and the format must be set to DXGI_FORMAT_UNKNOWN.

Finally call ID3D12Device::CreateUnorderedAccessView.

For example,

C++

```
D3D12_UNORDERED_ACCESS_VIEW_DESC uavDesc = {};
uavDesc.Format = DXGI_FORMAT_UNKNOWN;
uavDesc.ViewDimension = D3D12_UAV_DIMENSION_BUFFER;
uavDesc.Buffer.FirstElement = 0;
uavDesc.Buffer.NumElements = TriangleCount;
uavDesc.Buffer.StructureByteStride = sizeof(IndirectCommand);
uavDesc.Buffer.CounterOffsetInBytes = 0;
uavDesc.Buffer.Flags = D3D12_BUFFER_UAV_FLAG_NONE;

m_device->CreateUnorderedAccessView(
    m_processedCommandBuffers[frame].Get(),
    m_processedCommandBufferCounters[frame].Get(),
    &uavDesc,
    processedCommandsHandle);
```

Stream Output View

To create a stream output view, fill out a D3D12_STREAM_OUTPUT_DESC structure.

```
typedef struct D3D12_STREAM_OUTPUT_DESC {
    _Field_size_full_(NumEntries) const D3D12_SO_DECLARATION_ENTRY *pSODeclaration;
    UINT NumEntries;
    _Field_size_full_(NumStrides) const UINT *pBufferStrides;
    UINT NumStrides;
    UINT RasterizedStream;
} D3D12_STREAM_OUTPUT_DESC;
```

Then call ID3D12GraphicsCommandList::SOSetTargets.

Render Target View

To create a render target view, fill out a D3D12_RENDER_TARGET_VIEW_DESC structure.

```
typedef struct D3D12_RENDER_TARGET_VIEW_DESC {
    DXGI_FORMAT Format;
    D3D12_RTV_DIMENSION ViewDimension;
    union {
        D3D12_BUFFER_RTV Buffer;
        D3D12_TEX1D_RTV Texture1D;
        D3D12_TEX1D_ARRAY_RTV Texture1DArray;
        D3D12_TEX2D_RTV Texture2D;
        D3D12_TEX2D_ARRAY_RTV Texture2DArray;
        D3D12_TEX2DMS_RTV Texture2DMS;
        D3D12_TEX2DMS_ARRAY_RTV Texture2DMSArray;
        D3D12_TEX3D_RTV Texture3D;
    };
} D3D12_RENDER_TARGET_VIEW_DESC;
```

The `ViewDimension` field is set to zero, or one value of the D3D12_RTV_DIMENSION enum.

Refer to the following enums and structures:

- DXGI_FORMAT
- D3D12_BUFFER_RTV
- D3D12_TEX1D_RTV
- D3D12_TEX1D_ARRAY_RTV
- D3D12_TEX2D_RTV

- D3D12_TEX2DMS_RTV
- D3D12_TEX2D_ARRAY_RTV
- D3D12_TEX2DMS_ARRAY_RTV
- D3D12_TEX3D_RTV

Finally, call ID3D12Device::CreateRenderTargetView.

Note that an alternative way to create a render target view is to create a render target descriptor heap.

For example,

C++

```
// Create descriptor heaps.
{
    // Describe and create a render target view (RTV) descriptor heap.
    D3D12_DESCRIPTOR_HEAP_DESC rtvHeapDesc = {};
    rtvHeapDesc.NumDescriptors = FrameCount;
    rtvHeapDesc.Type = D3D12_DESCRIPTOR_HEAP_TYPE_RTV;
    rtvHeapDesc.Flags = D3D12_DESCRIPTOR_HEAP_FLAG_NONE;
    ThrowIfFailed(m_device->CreateDescriptorHeap(&rtvHeapDesc, IID_PPV_ARGS(&m_rtvHeap)));
    m_rtvDescriptorSize = m_device->GetDescriptorHandleIncrementSize(D3D12_DESCRIPTOR_HEAP_TYPE_RTV);
}

// Create frame resources.
{
    CD3DX12_CPU_DESCRIPTOR_HANDLE rtvHandle(m_rtvHeap->GetCPUDescriptorHandleForHeapStart());

    // Create a RTV for each frame.
    for (UINT n = 0; n < FrameCount; n++) {
        ThrowIfFailed(m_swapChain->GetBuffer(n, IID_PPV_ARGS(&m_renderTargets[n])));
        m_device->CreateRenderTargetView(m_renderTargets[n].Get(), nullptr, rtvHandle);
        rtvHandle.Offset(1, m_rtvDescriptorSize);
    }
}
```

Depth Stencil View

To create a depth stencil view, fill out a D3D12_DEPTH_STENCIL_VIEW_DESC structure:

```
typedef struct D3D12_DEPTH_STENCIL_VIEW_DESC {
    DXGI_FORMAT Format;
    D3D12_DSV_DIMENSION ViewDimension;
    D3D12_DSV_FLAGS Flags;
    union {
        D3D12_TEX1D_DSV Texture1D;
        D3D12_TEX1D_ARRAY_DSV Texture1DArray;
        D3D12_TEX2D_DSV Texture2D;
        D3D12_TEX2D_ARRAY_DSV Texture2DArray;
        D3D12_TEX2DMS_DSV Texture2DMS;
        D3D12_TEX2DMS_ARRAY_DSV Texture2DMSArray;
    }     ;
} D3D12_DEPTH_STENCIL_VIEW_DESC;
```

The `ViewDimension` field is set to zero, or one value of the D3D12_DSV_DIMENSION enum. Refer to the D3D12_DSV_FLAGS enum for the flag settings.

Refer to the following enums and structures:

- DXGI_FORMAT
- D3D12_TEX1D_DSV
- D3D12_TEX1D_ARRAY_DSV

- D3D12_TEX2D_DSV
- D3D12_TEX2D_ARRAY_DSV
- D3D12_TEX2DMS_DSV
- D3D12_TEX2DMS_ARRAY_DSV

Finally, call ID3D12Device::CreateDepthStencilView.

For example,

C++

```
// Create the depth stencil view.
{
    D3D12_DEPTH_STENCIL_VIEW_DESC depthStencilDesc = {};
    depthStencilDesc.Format = DXGI_FORMAT_D32_FLOAT;
    depthStencilDesc.ViewDimension = D3D12_DSV_DIMENSION_TEXTURE2D;
    depthStencilDesc.Flags = D3D12_DSV_FLAG_NONE;

    D3D12_CLEAR_VALUE depthOptimizedClearValue = {};
    depthOptimizedClearValue.Format = DXGI_FORMAT_D32_FLOAT;
    depthOptimizedClearValue.DepthStencil.Depth = 1.0f;
    depthOptimizedClearValue.DepthStencil.Stencil = 0;

    ThrowIfFailed(m_device->CreateCommittedResource(
        &CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_DEFAULT),
        D3D12_HEAP_FLAG_NONE,
        &CD3DX12_RESOURCE_DESC::Tex2D(DXGI_FORMAT_D32_FLOAT, m_width, m_height, 1, 0, 1, 0,
            D3D12_RESOURCE_FLAG_ALLOW_DEPTH_STENCIL),
         D3D12_RESOURCE_STATE_DEPTH_WRITE,
         &depthOptimizedClearValue,
         IID_PPV_ARGS(&m_depthStencil)
        ));

    m_device->CreateDepthStencilView(
         m_depthStencil.Get(),
         &depthStencilDesc,
         m_dsvHeap->GetCPUDescriptorHandleForHeapStart());
}
```

Copying Descriptors

The ID3D12Device::CopyDescriptors method on the device interface uses the CPU to immediately copy descriptors. This can be called free threaded as long as multiple threads on the CPU or GPU do not perform any potentially conflicting writes.

Copying Descriptors Immediately (CPU Timeline)

The number of source descriptors (to copy from), specified as a set of descriptor ranges, must equal the number of destination descriptors (to copy to), specified as a separate set of descriptor ranges. The source and destination ranges do not otherwise have to line up. For example, a sparse set of descriptors could be copied to a contiguous destination, vice versa, or in some combination.

Multiple descriptor heaps can be involved in the copy operation, both as source and destination. The use of descriptor handles as parameters means the copy methods don’t care about which heap(s) any given descriptor lies in – they are all just memory.

The descriptor heap types being copied from and to must match, so the methods take a single descriptor heap type as input. The driver needs to know the heap type of all the descriptors in the given copy operation, so it knows what size of data is involved in the copy operation. The driver might also need to do custom copying work if a given descriptor heap type warrants it – an implementation detail. Note that descriptor handles themselves do not otherwise identify what type they are pointing to; therefore, an additional parameter is required for the copy operation.

An alternative API to **CopyDescriptors** is provided for the simple case of copying a single range of descriptors from one location to another –**CopyDescriptorsSimple**.

For these device based (CPU timeline) descriptor copy methods, source descriptors must come from a non-shader visible descriptor heap. The destination descriptors can be in any descriptor heap that is CPU visible (shader visible or not).

Descriptor Heaps

A descriptor heap is a collection of contiguous allocations of descriptors, one allocation for every descriptor. Descriptor heaps contain many object types that are not part of a Pipeline State Object (PSO), such as Shader Resource Views (SRVs), Unordered Access Views (UAVs), Constant Buffer Views (CBVs), and Samplers.

The primary purpose of a descriptor heap is to encompass the bulk of memory allocation required for storing the descriptor specifications of object types that shaders reference for as large of a window of rendering as possible (ideally an entire frame of rendering or more). If an application is switching which textures the pipeline sees rapidly from the API, there has to be space in the descriptor heap to define descriptor tables on the fly for every set of state needed. The application can choose to reuse definitions if the resources are used again in another object, for example, or just assign the heap space sequentially as it switches various object types.

Descriptor heaps also allow individual software components to manage descriptor storage separately from each other.

All heaps are visible to the CPU. The application can also request which CPU access properties a descriptor heap should have (if any) – write combined, write back, and so on. Apps can create as many descriptor heaps as desired with whatever properties are desired. Apps always have the option to create descriptor heaps that are purely for staging purposes that are unconstrained in size, and copying to descriptor heaps that are used for rendering as necessary.

There are some restrictions in what can go in the same descriptor heap. CBV, UAV and SRV entries can be in the same descriptor heap. However, Samplers entries cannot share a heap with CBV, UAV or SRV entries. Typically, there are two sets of descriptor heaps, one for the common resources and the second for Samplers.

Switching Descriptor Heaps

It is acceptable for an application to switch heaps within the same command list or in different ones using the **SetDescriptorHeaps** and **Reset** APIs. On some hardware, this can be an expensive operation, requiring a GPU stall to flush all work that depends on the currently bound descriptor heap. As a result, if descriptor heaps must be changed, applications should try to do so when the GPU workload is relatively light.

In this section

Topic	Description
Hardware Tiers	The levels of hardware from Tier 1 to Tier 3 have increasing resources available to the pipeline.
Shader Visible Descriptor Heaps	Descriptor heaps that can be referenced by shaders through descriptor tables.
Non Shader Visible Descriptor Heaps	Descriptor heaps that cannot be referenced by shaders through descriptor tables.
Creating Descriptor Heaps	To create and configure a descriptor heap, you must select a descriptor heap type, determine how many descriptors it contains, and set flags that indicate whether it is CPU visible and/or shader visible.
Setting and Populating Descriptor Heaps	The descriptor heap types that can be set on a command list are those that contain descriptors for which descriptor tables can be used (at most one of each at a time).
Descriptor Heap Configurability Summary	The following table summarizes information about Shader and non-Shader visible heap support.

Hardware Tiers

The levels of hardware from Tier 1 to Tier 3 have increasing resources available to the pipeline.

- Limits dependant on hardware
- Invariable limits
- Related topics

Limits dependant on hardware

Resources Available to the Pipeline	Tier 1	Tier 2	Tier 3
Feature levels	11.0+	11.0+	11.1+
Maximum number of descriptors in a Constant Buffer View (CBV), Shader Resource View (SRV), or Unordered Access View(UAV) heap used for rendering	1,000,000	1,000,000	1,000,000+
Maximum number of Constant Buffer Views in all descriptor tables per shader stage	14	14	**full heap**
Maximum number of Shader Resource Views in all descriptor tables per shader stage	128	**full heap**	full heap
Maximum number of Unordered Access	64 for feature levels 11.1+. 8 for lower feature levels.	64	**full heap**

Views in all descriptor tables across all stages			
Maximum number of Samplers in all descriptor tables per shader stage	16	**full heap**	full heap
Maximum number of simultaneous descriptor tables containing Shader Resource Views (same separate limit for sampler tables)	5	5	**no limit**

Bold entries highlight significant improvements over the previous tier.

Invariable limits

The maximum number of samplers in a shader visible descriptor heap is 2048.

The maximum number of unique static samplers across live root signatures is 2032 (which leaves 16 for drivers that need their own samplers).

Shader Visible Descriptor Heaps

Descriptor heaps that can be referenced by shaders through descriptor tables.

Descriptor heaps that can be referenced by shaders through descriptor tables come in a couple of flavors: One heap type, D3D12_SRV_UAV_CBV_DESCRIPTOR_HEAP, can hold Shader Resource Views, Unordered Access Views, and Constant Buffer Views, all intermixed. Any given location in the heap can be any one of the listed types of descriptors. Another heap type, D3D12_DESCRIPTOR_HEAP_TYPE_SAMPLER, only stores samplers, reflecting the fact that for the majority of hardware, samplers are managed separately from SRVs, UAVs, CBVs.

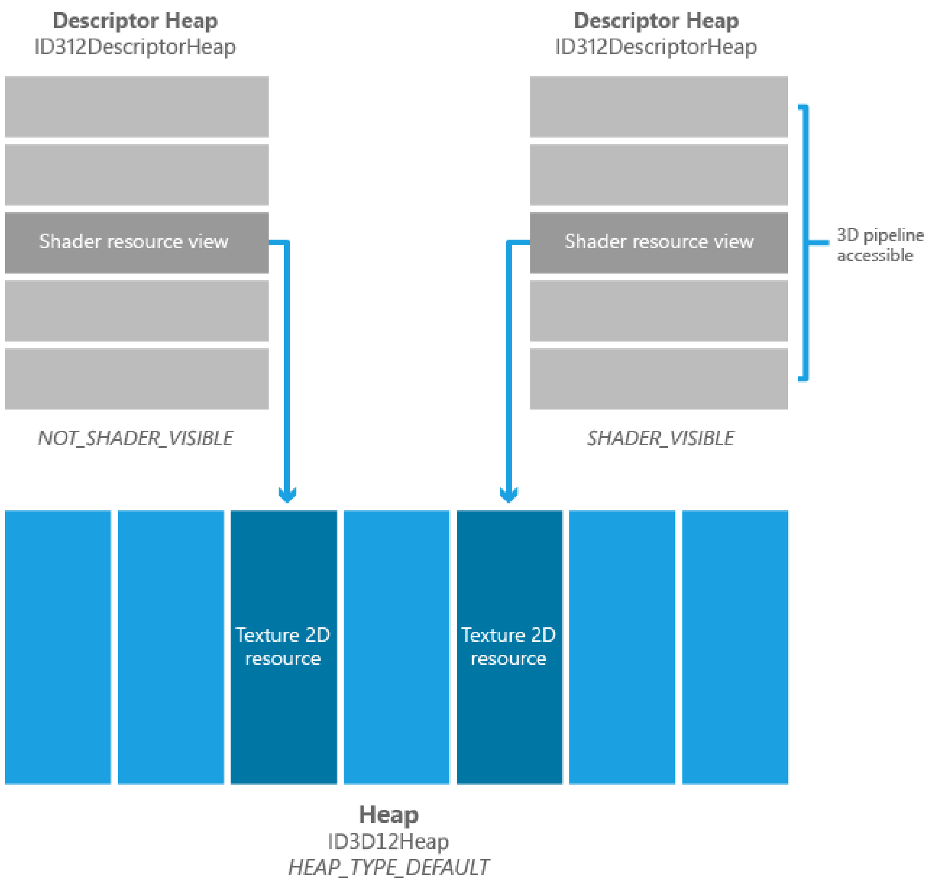
Descriptor heaps of these types may be requested to be shader visible or not when the heap is created (the latter – non shader visible - can be useful for staging descriptors on the CPU). When requested to be shader visible, each of the above heap types may have a hardware size limit for any individual descriptor heap allocation.

Applications can create any number of descriptor heaps , and non shader visible descriptor heaps are not constrained in size. If a shader visible descriptor heap that is created by the application is smaller than the hardware size limit, the driver may choose to sub-allocate the descriptor heap out of a larger underlying descriptor heap so that multiple API descriptor heaps fit within one hardware descriptor heap. The reason this may happen is that for some hardware, switching between hardware descriptor heaps during execution requires a GPU wait for idle (to ensure that GPU references to the previously descriptor heap are finished). If all of the descriptor heaps that an application creates fit into the applicable hardware heap's maximum capacities, then no such waits will occur when switching API descriptor heaps during rendering. Applications must allow for the possibility, however, that switching the current descriptor heap may incur a wait for idle.

To avoid being impacted by this possible wait for idle on the descriptor heap switch, applications can take advantage of breaks in rendering that would cause the GPU to idle for other reasons as the time to do descriptor heap switches, since a wait for idle is happening anyway.

The mechanism and semantics for identifying descriptor heaps to shaders during command list / bundle recording are described in the API reference.

The image below shows two descriptor heaps referencing two separate 2D textures being stored in two slots of a large default heap. The descriptor heap that is shader visible can be accessed by the graphics pipeline (including the shaders), and hence the 2D texture is available to the pipeline.



Non Shader Visible Descriptor Heaps

Descriptor heaps that cannot be referenced by shaders through descriptor tables.

All descriptor heaps, including the shader accessible descriptor heaps described previously, can be manipulated by the CPU and/or command lists depending on the memory pool and CPU access properties the application selects for a descriptor heap.

For descriptor heap types that are shader visible and described above, the obvious reason for non-shader access to the descriptor heap is to populate its contents before shader execution. Other descriptor heap types store descriptors that the driver just snapshots a copy of during command list recording that the GPU will later reference during execution. For example, Render Target Views (RTVs), Depth Stencil Views (DSVs), Index Buffer Views (IBVs), and other descriptors get bound to the pipeline by having their contents recorded into the command list directly rather than via descriptor tables which reference the descriptor heap at command list execution. Even descriptor tables have options where an app can allow the implementation to choose to record the table contents at command list recording (rather than dereference the table pointer at execution).

Creating Descriptor Heaps

To create and configure a descriptor heap, you must select a descriptor heap type, determine how many descriptors it contains, and set flags that indicate whether it is CPU visible and/or shader visible.

- Descriptor Heap types
- Descriptor Heap Properties
- Descriptor Handles
- Descriptor Heap Methods
- Minimal descriptor heap wrapper
- Related topics

Descriptor Heap types

The type of heap is determined by one member of the D3D12_DESCRIPTOR_HEAP_TYPE enum:

```
typedef enum D3D12_DESCRIPTOR_HEAP_TYPE {
    D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV,    // Constant buffer/Shader resource/Unordered access views
    D3D12_DESCRIPTOR_HEAP_TYPE_SAMPLER,        // Samplers
    D3D12_DESCRIPTOR_HEAP_TYPE_RTV,            // Render target view
    D3D12_DESCRIPTOR_HEAP_TYPE_DSV,            // Depth stencil view
    D3D12_DESCRIPTOR_HEAP_TYPE_NUM_TYPES       // Simply the number of descriptor heap types
} D3D12_DESCRIPTOR_HEAP_TYPE;
```

Descriptor Heap Properties

Heap properties are set on the D3D12_DESCRIPTOR_HEAP_DESC structure, which references both theD3D12_DESCRIPTOR_HEAP_TYPE andD3D12_DESCRIPTOR_HEAP_FLAGS enums.

For CBV, SRV, and UAV descriptor heaps, and sampler descriptor heaps, the flag D3D12_DESCRIPTOR_HEAP_FLAG_SHADER_VISIBLE must be set.

The flag D3D12_DESCRIPTOR_HEAP_FLAG_SHADER_VISIBLE indicates that the heap is intended to be bound on a command list for reference by shaders. This flag does not apply to other descriptor heap types since shaders do not directly reference the other types.

For example, describe and create a sampler descriptor heap.

C++

```cpp
// Describe and create a sampler descriptor heap.
D3D12_DESCRIPTOR_HEAP_DESC samplerHeapDesc = {};
samplerHeapDesc.NumDescriptors = 1;
samplerHeapDesc.Type = D3D12_DESCRIPTOR_HEAP_TYPE_SAMPLER;
samplerHeapDesc.Flags = D3D12_DESCRIPTOR_HEAP_FLAG_SHADER_VISIBLE;
ThrowIfFailed(m_device->CreateDescriptorHeap(&samplerHeapDesc, IID_PPV_ARGS(&m_samplerHeap)));
```

Describe and create a constant buffer view (CBV), Shader resource view (SRV), and unordered access view (UAV) descriptor heap.

C++

```cpp
// Describe and create a shader resource view (SRV) and unordered
// access view (UAV) descriptor heap.
D3D12_DESCRIPTOR_HEAP_DESC srvUavHeapDesc = {};
srvUavHeapDesc.NumDescriptors = DescriptorCount;
srvUavHeapDesc.Type = D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV;
srvUavHeapDesc.Flags = D3D12_DESCRIPTOR_HEAP_FLAG_SHADER_VISIBLE;
ThrowIfFailed(m_device->CreateDescriptorHeap(&srvUavHeapDesc, IID_PPV_ARGS(&m_srvUavHeap)));

m_rtvDescriptorSize = m_device->GetDescriptorHandleIncrementSize(D3D12_DESCRIPTOR_HEAP_TYPE_RTV);
m_srvUavDescriptorSize = m_device->GetDescriptorHandleIncrementSize(D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV);
```

Descriptor Handles

The D3D12_GPU_DESCRIPTOR_HANDLE and D3D12_CPU_DESCRIPTOR_HANDLE structures identify specific descriptors in a descriptor heap. A handle is a bit like a pointer, but the application must not dereference it manually; otherwise, the behavior is undefined. The use of the handles must go through the API. A handle itself can be copied freely or passed into APIs that operate on/use descriptors. There is no ref counting, so the application must ensure that it does not use a handle after the underlying descriptor heap has been deleted.

Applications can find out the increment size of the descriptors for a given descriptor heap type, so that they can generate handles to any location in a descriptor heap manually starting from the handle to the base. Applications must never hardcode descriptor handle increment sizes, and should always query them for a given device instance; otherwise, the behavior is undefined. Applications must also not use the increment sizes and handles to do their own examination or manipulation of descriptor heap data, as the results from doing so are undefined. The handles may not actually be used as pointers, but rather as proxies for pointers so as to avoid accidental dereferencing.

Note

There is a helper structure, CD3DX12_GPU_DESCRIPTOR_HANDLE, defined in the header d3dx12.h, which inherits the D3D12_GPU_DESCRIPTOR_HANDLEstructure and provides initialization and other useful operations. Similarly the CD3DX12_CPU_DESCRIPTOR_HANDLE helper structure is defined for theD3D12_CPU_DESCRIPTOR_HANDLE structure.

Both of these helper structures are used when populating command lists.

C++

```cpp
// Fill the command list with all the render commands and dependent state.
void D3D12nBodyGravity::PopulateCommandList() {
    // Command list allocators can only be reset when the associated
    // command lists have finished execution on the GPU; apps should use
    // fences to determine GPU execution progress.
    ThrowIfFailed(m_commandAllocators[m_frameIndex]->Reset());

    // However, when ExecuteCommandList() is called on a particular command
    // list, that command list can then be reset at any time and must be before
    // re-recording.
    ThrowIfFailed(m_commandList->Reset(m_commandAllocators[m_frameIndex].Get(), m_pipelineState.Get()));

    // Set necessary state.
    m_commandList->SetPipelineState(m_pipelineState.Get());
    m_commandList->SetGraphicsRootSignature(m_rootSignature.Get());

    m_commandList->SetGraphicsRootConstantBufferView(
         RootParameterCB,
         m_constantBufferGS->GetGPUVirtualAddress() + m_frameIndex * sizeof(ConstantBufferGS));

    ID3D12DescriptorHeap* ppHeaps[] = { m_srvUavHeap.Get() };
    m_commandList->SetDescriptorHeaps(_countof(ppHeaps), ppHeaps);

    m_commandList->IASetVertexBuffers(0, 1, &m_vertexBufferView);
    m_commandList->IASetPrimitiveTopology(D3D11_PRIMITIVE_TOPOLOGY_POINTLIST);
    m_commandList->RSSetScissorRects(1, &m_scissorRect);

    // Indicate that the back buffer will be used as a render target.
    m_commandList->ResourceBarrier(
         1,
         &CD3DX12_RESOURCE_BARRIER::Transition(m_renderTargets[m_frameIndex].Get(),
                                              D3D12_RESOURCE_STATE_PRESENT,
                                              D3D12_RESOURCE_STATE_RENDER_TARGET));

    CD3DX12_CPU_DESCRIPTOR_HANDLE rtvHandle(
         m_rtvHeap->GetCPUDescriptorHandleForHeapStart(),
         m_frameIndex,
         m_rtvDescriptorSize);
    m_commandList->OMSetRenderTargets(1, &rtvHandle, FALSE, nullptr);

    // Record commands.
    const float clearColor[] = { 0.0f, 0.0f, 0.1f, 0.0f };
    m_commandList->ClearRenderTargetView(rtvHandle, clearColor, 0, nullptr);

    // Render the particles.
    float viewportHeight = static_cast<float>(static_cast<UINT>(m_viewport.Height) / m_heightInstances);
    float viewportWidth = static_cast<float>(static_cast<UINT>(m_viewport.Width) / m_widthInstances);
    for (UINT n = 0; n < ThreadCount; n++) {
        const UINT srvIndex = n + (m_srvIndex[n] == 0 ? SrvParticlePosVelo0 : SrvParticlePosVelo1);

        D3D12_VIEWPORT viewport;
        viewport.TopLeftX = (n % m_widthInstances) * viewportWidth;
        viewport.TopLeftY = (n / m_widthInstances) * viewportHeight;
        viewport.Width = viewportWidth;
        viewport.Height = viewportHeight;
        viewport.MinDepth = D3D12_MIN_DEPTH;
        viewport.MaxDepth = D3D12_MAX_DEPTH;
        m_commandList->RSSetViewports(1, &viewport);

        CD3DX12_GPU_DESCRIPTOR_HANDLE srvHandle(
                   m_srvUavHeap->GetGPUDescriptorHandleForHeapStart(),
                   srvIndex,
                   m_srvUavDescriptorSize);
        m_commandList->SetGraphicsRootDescriptorTable(RootParameterSRV, srvHandle);
        m_commandList->DrawInstanced(ParticleCount, 1, 0, 0);
    }

    m_commandList->RSSetViewports(1, &m_viewport);

    // Indicate that the back buffer will now be used to present.
    m_commandList->ResourceBarrier(
         1,
         &CD3DX12_RESOURCE_BARRIER::Transition(m_renderTargets[m_frameIndex].Get(),
                                              D3D12_RESOURCE_STATE_RENDER_TARGET,
                                              D3D12_RESOURCE_STATE_PRESENT));

    ThrowIfFailed(m_commandList->Close());
}
```

Descriptor Heap Methods

Descriptor heaps (ID3D12DescriptorHeap) inherit from ID3D12Pageable. This imposes the responsibility for the residency management of descriptor heaps on applications, just like resource heaps. The residency management methods only apply to shader visible heaps since the non shader visible heaps are not visible to the GPU directly.

The ID3D12Device::GetDescriptorHandleIncrementSize method allows applications to manually offset handles into a heap (producing handles into anywhere in a descriptor heap). The heap start location's handle comes fromID3D12DescriptorHeap::GetCPUDescriptorHandleForHeapStart/ID3D12DescriptorHeap::GetGPUDescriptorHandleForHeapStart. Offsetting is done by adding the increment size * the number of descriptors to offset to the descriptor heap start . Note that the increment size cannot be thought of as a byte size since applications must not dereference handles as if they are memory – the memory pointed to has a non-standardized layout and can vary even for a given device.

GetCPUDescriptorHandleForHeapStart returns a CPU handle for CPU visible descriptor heaps. It returns a NULL handle (and the debug layer will report an error) if the descriptor heap is not CPU visible.

GetGPUDescriptorHandleForHeapStart returns a GPU handle for shader visible descriptor heaps. It returns a NULL handle (and the debug layer will report an error) if the descriptor heap is not shader visible.

For example, creating render target views to display D2D text using an 11on12 device.

C++

```
// Create descriptor heaps.
{
    // Describe and create a render target view (RTV) descriptor heap.
    D3D12_DESCRIPTOR_HEAP_DESC rtvHeapDesc = {};
    rtvHeapDesc.NumDescriptors = FrameCount;
    rtvHeapDesc.Type = D3D12_DESCRIPTOR_HEAP_TYPE_RTV;
    rtvHeapDesc.Flags = D3D12_DESCRIPTOR_HEAP_FLAG_NONE;
    ThrowIfFailed(m_d3d12Device->CreateDescriptorHeap(&rtvHeapDesc, IID_PPV_ARGS(&m_rtvHeap)));
    m_rtvDescriptorSize = m_d3d12Device->GetDescriptorHandleIncrementSize(D3D12_DESCRIPTOR_HEAP_TYPE_RTV);
}

// Create frame resources.
{
    CD3DX12_CPU_DESCRIPTOR_HANDLE rtvHandle(m_rtvHeap->GetCPUDescriptorHandleForHeapStart());

    // Create a RTV, D2D render target, and a command allocator for each frame.
    for (UINT n = 0; n < FrameCount; n++) {
        ThrowIfFailed(m_swapChain->GetBuffer(n, IID_PPV_ARGS(&m_renderTargets[n])));
        m_d3d12Device->CreateRenderTargetView(m_renderTargets[n].Get(), nullptr, rtvHandle);

        // Create a wrapped 11On12 resource of this back buffer. Since we are
        // rendering all D3D12 content first and then all D2D content, we specify
        // the In resource state as RENDER_TARGET - because D3D12 will have last
        // used it in this state - and the Out resource state as PRESENT. When
        // ReleaseWrappedResources() is called on the 11On12 device, the resource
        // will be transitioned to the PRESENT state.
        D3D11_RESOURCE_FLAGS d3d11Flags = { D3D11_BIND_RENDER_TARGET };
        ThrowIfFailed(m_d3d11On12Device->CreateWrappedResource(
            m_renderTargets[n].Get(),
            &d3d11Flags,
            D3D12_RESOURCE_STATE_RENDER_TARGET,
            D3D12_RESOURCE_STATE_PRESENT,
            IID_PPV_ARGS(&m_wrappedBackBuffers[n])
            ));

        // Create a render target for D2D to draw directly to this back buffer.
        ComPtr<IDXGISurface> surface;
        ThrowIfFailed(m_wrappedBackBuffers[n].As(&surface));
        ThrowIfFailed(m_d2dDeviceContext->CreateBitmapFromDxgiSurface(
            surface.Get(),
            &bitmapProperties,
            &m_d2dRenderTargets[n]
            ));

        rtvHandle.Offset(1, m_rtvDescriptorSize);

        ThrowIfFailed(m_d3d12Device->CreateCommandAllocator(
           D3D12_COMMAND_LIST_TYPE_DIRECT,
           IID_PPV_ARGS(&m_commandAllocators[n])));
    }
}
```

Minimal descriptor heap wrapper

Application developers will likely want to build their own helper code for managing descriptor handles and heaps. A basic example is shown below. More sophisticated wrappers could, for example, keep track of what types of descriptors are where in a heap and store the descriptor creation arguments.

```
class CdescriptorHeapWrapper {
public:
    CDescriptorHeapWrapper() { memset(this, 0, sizeof(*this)); }

    HRESULT Create(
        ID3D12Device* pDevice,
        D3D12_DESCRIPTOR_HEAP_TYPE Type,
        UINT NumDescriptors,
        bool bShaderVisible = false)
    {
        D3D12_DESCRIPTOR_HEAP_DESC Desc;
        Desc.Type = Type;
        Desc.NumDescriptors = NumDescriptors;
        Desc.Flags = (bShaderVisible ? D3D12_DESCRIPTOR_HEAP_FLAG_SHADER_VISIBLE : 0);

        HRESULT hr = pDevice->CreateDescriptorHeap(
                                 &Desc,
                                 __uuidof(ID3D12DescriptorHeap),
                                 (void**)&pDH);
        if (FAILED(hr)) return hr;

        hCPUHeapStart = pDH->GetCPUDescriptorHandleForHeapStart();
        hGPUHeapStart = pDH->GetGPUDescriptorHandleForHeapStart();

        HandleIncrementSize = pDevice->GetDescriptorHandleIncrementSize(Desc.Type);
        return hr;
    }

    operator ID3D12DescriptorHeap*() { return pDH; }

    D3D12_CPU_DESCRIPTOR_HANDLE hCPU(UINT index) {
        return hCPUHeapStart.MakeOffsetted(index,HandleIncrementSize);
    }

    D3D12_GPU_DESCRIPTOR_HANDLE hGPU(UINT index) {
        assert(Desc.Flags&D3D12_DESCRIPTOR_HEAP_FLAG_SHADER_VISIBLE);
        return hGPUHeapStart.MakeOffsetted(index,HandleIncrementSize);
    }

    D3D12_DESCRIPTOR_HEAP_DESC Desc;
    CComPtr<ID3D12DescriptorHeap> pDH;
    D3D12_CPU_DESCRIPTOR_HANDLE hCPUHeapStart;
    D3D12_GPU_DESCRIPTOR_HANDLE hGPUHeapStart;
    UINT HandleIncrementSize;
};
```

Setting and Populating Descriptor Heaps

The descriptor heap types that can be set on a command list are those that contain descriptors for which descriptor tables can be used (at most one of each at a time).

- Setting descriptor heaps
- Populating descriptor heaps
- Related topics

Setting descriptor heaps

The types of descriptor heap that can be set on a command list are:

```
D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV
D3D12_DESCRIPTOR_HEAP_TYPE_SAMPLER
```

The heaps being set on the command list must also have been created as shader visible. There are three types of command list: DIRECT, BUNDLE, and COMPUTE.

After a descriptor heap is set on a command list, subsequent calls that define descriptor tables refer to the current descriptor heap. Descriptor table state is undefined at the beginning of a command list and after descriptor heaps are changed on a command list. Redundantly setting the same descriptor heap does not cause descriptor table settings to be undefined.

In a bundle, by contrast, the descriptor heaps can only be set once (redundant calls setting the same heap twice do not produce an error); otherwise, the behavior is undefined. The descriptor heaps that are set must match the state when any command list calls the bundle; otherwise, the behavior is undefined. This allows bundles to inherit and edit the command list's descriptor table settings. Bundles that don't change descriptor tables (only inherit them) don't need to set a descriptor heap at all and will just inherit from the calling command list.

When descriptor heaps are set (using ID3D12GraphicsCommandList::SetDescriptorHeaps), all the heaps being used are set in a single call (and all previously set heaps are unset by the call). At most one heap of each type listed above can be set in the call.

Populating descriptor heaps

After an application has created a descriptor heap, it can then use methods on the heap or on a command list to either generate descriptors directly into the heap or copy descriptors from one place to another.

The initial contents of descriptor heap memory is undefined, so asking the GPU or driver to reference uninitialized memory for rendering can cause undefined results such as a device reset.

If the application configures a descriptor heap to be CPU visible, then the CPU can call methods to create descriptors into the heap and copy from place to place (including across heaps) in an immediate, free threaded manner. If the heap has been configured with a write combine property, reading by the CPU is not permitted.

Descriptors can also record a descriptor copy call on a command list in the event that the application does not want the copy to occur immediately, but rather when the GPU is executing the command list. This can also be useful if an application chooses to put a descriptor heap in a non CPU-visible memory pool and therefore needs to do a GPU operation to manipulate its contents, or if an application simply wants the update to occur on the GPU time-line. Command list descriptor copies require the source of the copy to be in a non shader visible descriptor heap (from which the command list snapshots a copy the source descriptors into the command list at record time), and the destination must be a shader visible descriptor heap (which at command list execution gets written to by the GPU). The API reference for copying descriptors goes into more detail.

For example, setting descriptor heaps when populating command lists.

C++

```cpp
void D3D12Bundles::PopulateCommandList(FrameResource* pFrameResource) {
    // Command list allocators can only be reset when the associated
    // command lists have finished execution on the GPU; apps should use
    // fences to determine GPU execution progress.
    ThrowIfFailed(m_pCurrentFrameResource->m_commandAllocator->Reset());

    // However, when ExecuteCommandList() is called on a particular command
    // list, that command list can then be reset at any time and must be before
    // re-recording.
    ThrowIfFailed(m_commandList->Reset(
         m_pCurrentFrameResource->m_commandAllocator.Get(),
         m_pipelineState1.Get()));

    // Set necessary state.
    m_commandList->SetGraphicsRootSignature(m_rootSignature.Get());

    ID3D12DescriptorHeap* ppHeaps[] = { m_cbvSrvHeap.Get(), m_samplerHeap.Get() };
    m_commandList->SetDescriptorHeaps(_countof(ppHeaps), ppHeaps);

    m_commandList->RSSetViewports(1, &m_viewport);
    m_commandList->RSSetScissorRects(1, &m_scissorRect);

    // Indicate that the back buffer will be used as a render target.
    m_commandList->ResourceBarrier(
         1,
         &CD3DX12_RESOURCE_BARRIER::Transition(m_renderTargets[m_frameIndex].Get(),
                                               D3D12_RESOURCE_STATE_PRESENT,
                                               D3D12_RESOURCE_STATE_RENDER_TARGET));

    CD3DX12_CPU_DESCRIPTOR_HANDLE rtvHandle(
         m_rtvHeap->GetCPUDescriptorHandleForHeapStart(),
         m_frameIndex,
         m_rtvDescriptorSize);
    CD3DX12_CPU_DESCRIPTOR_HANDLE dsvHandle(m_dsvHeap->GetCPUDescriptorHandleForHeapStart());
    m_commandList->OMSetRenderTargets(1, &rtvHandle, FALSE, &dsvHandle);

    // Record commands.
    const float clearColor[] = { 0.0f, 0.2f, 0.4f, 1.0f };
    m_commandList->ClearRenderTargetView(rtvHandle, clearColor, 0, nullptr);
    m_commandList->ClearDepthStencilView(
         m_dsvHeap->GetCPUDescriptorHandleForHeapStart(),
         D3D12_CLEAR_FLAG_DEPTH,
         1.0f,
         0,
         0,
         nullptr);

    if (UseBundles) {
        // Execute the prebuilt bundle.
        m_commandList->ExecuteBundle(pFrameResource->m_bundle.Get());
    } else {
        // Populate a new command list.
        pFrameResource->PopulateCommandList(
            m_commandList.Get(),
            m_pipelineState1.Get(),
            m_pipelineState2.Get(),
            m_currentFrameResourceIndex,
            m_numIndices,
            &m_indexBufferView,
           &m_vertexBufferView,
            m_cbvSrvHeap.Get(),
            m_cbvSrvDescriptorSize,
            m_samplerHeap.Get(),
            m_rootSignature.Get());
    }

    // Indicate that the back buffer will now be used to present.
    m_commandList->ResourceBarrier(
         1,
         &CD3DX12_RESOURCE_BARRIER::Transition(m_renderTargets[m_frameIndex].Get(),
                                               D3D12_RESOURCE_STATE_RENDER_TARGET,
                                               D3D12_RESOURCE_STATE_PRESENT));

    ThrowIfFailed(m_commandList->Close());
}
```

Descriptor Heap Configurability Summary

The following table summarizes information about Shader and non-Shader visible heap support.

	Shader Visible Descriptor Heap	**Non Shader Visible Descriptor Heap**
Heap Types Supported	CBV_SRV_UAV, Sampler	All
CPU Page Properties Supported	NOT_AVAILABLE, WRITE_COMBINE	WRITE_BACK
Residency Management By App	Yes, app responsible	Not applicable (not GPU visible).
Descriptor Edit Support	Copy destination only, via command list update and/or CPU copy if CPU visible.	CPU read and write only. No direct GPU access. Can be used for immediate CPU copying (as a source and destination). Can be used as an update source on a command list – this will copy the descriptors into command list storage during command list record. On execution, the stored copy will get copied to the destination, which must be a shader visible heap.

Descriptor Tables

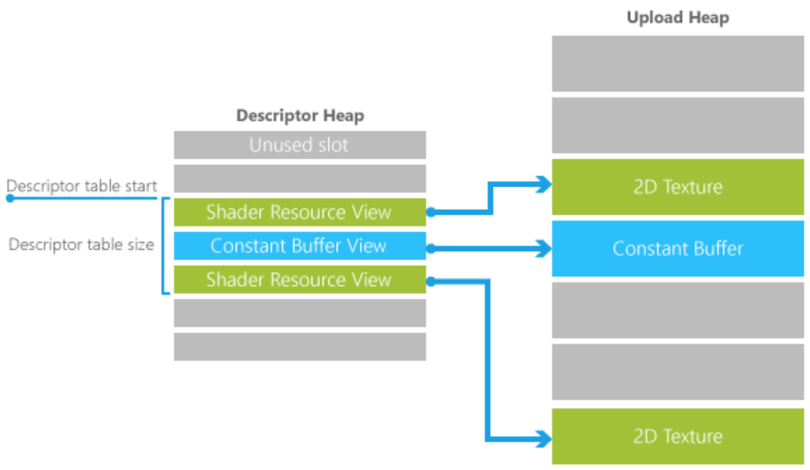
A descriptor table is logically an array of descriptors. Each descriptor table stores descriptors of one or more types - SRVs, UAVe, CBVs, and Samplers. A descriptor table is not an allocation of memory; it is simply an offset and length into a descriptor heap.

The graphics pipeline, through the root signature, gain access to resources by referencing into descriptor tables by index.

A descriptor table is actually just a sub-range of a descriptor heap. Descriptor heaps represent the underlying memory allocation for a collection of descriptors. Since memory allocation is a property of a creating a descriptor heap, defining a descriptor table out of one is guaranteed to be as cheap as identifying a region in the heap to the hardware. Descriptor tables don't need to be created or destroyed at the API level– they are merely identified to drivers as an offset and size out of a heap whenever referenced.

It is certainly possible for an app to define very large descriptor tables when its shaders want the freedom to select from a vast set of available descriptors (often referencing textures) on the fly (perhaps driven by material data).

The Root Signature references the descriptor table entry with a reference to the heap, the start location of the table (an offset from the start of the heap), and the length (in entries) of the table. The image below shows these concepts: the descriptor table pointers from the Root Signature and the descriptors within the descriptor heap referencing the full texture or buffer data in an upload heap.



In this section

Topic	Description
[Using Descriptor Tables](#)	Descriptor tables, each identifying a range in a descriptor heap, are bound at slots defined by the current root signature on a command list.
[Advanced Use of Descriptor Tables](#)	The following sections provide information about the advanced use of descriptor tables.

Using Descriptor Tables

Descriptor tables, each identifying a range in a descriptor heap, are bound at slots defined by the current root signature on a command list.

- Indexing Descriptor Tables
- Related topics

Shaders can locate resources referenced by the descriptors that make up the descriptor table. Other resource bindings - Index Buffers, Vertex Buffer, Stream Output Buffers, Render Targets, and Depth Stencil are done directly on a command list rather than via descriptors. To summarize:

The following resource references can share the same descriptor table and heap:

- Shader resource views
- Unordered access views
- Constant buffer views

The following resource references must be in their own descriptor heap:

- Samplers

The following resources are not placed in descriptor tables or heaps, but are bound directly using command lists:

- Index buffers
- Vertex buffers
- Stream output buffers
- Render targets
- Depth stencil views

Indexing Descriptor Tables

Shaders cannot dynamically index across descriptor table boundaries from a given call-site in the shader. However, the selection of a descriptor within a descriptor table is allowed to be dynamically indexed in shader code within ranges of the same descriptor type (such as indexing across a contiguous region of SRVs).

Advanced use of Descriptor Tables

The following sections provide information about the advanced use of descriptor tables.

Changing Descriptor Table Entries between Rendering Calls

After command lists that set descriptor tables have been submitted to a queue for execution, the application must not edit from the CPU the portions of descriptor heaps that the GPU might reference until the application knows that the GPU has finished using the references.

Work completion can be determined at a tight bound using API fences for tracking GPU progress, or more coarse mechanisms like waiting to see that rendering has been sent to display - whatever suits the application. If an application knows that only a subset of the region a descriptor table points to will be accessed (say due to flow control in the shader), the other unreferenced descriptors are still free to be changed. If an application needs to switch between different descriptor tables between rendering calls, there are a few approaches the application can choose from:

- Descriptor Table Versioning: Create (or reuse) a separate descriptor table for every unique collection of descriptors that is to be referenced by a command list. When editing and reusing previously populated areas on descriptor heaps, applications must first ensure that the GPU has finished using any portion of a descriptor heap that will be recycled.
- Dynamic Indexing: Applications can arrange objects that vary across draw/dispatch (or even vary within a draw) in a range of a descriptor heap, define a descriptor table that spans all of them, and from the shader, use dynamic indexing of the table during shader execution to select which object to use.
- Putting descriptors in the root signature directly. Only a very small number of descriptors can be managed this way because root signature space is limited.

The implication of using descriptor table versioning is that descriptor memory out of a descriptor heap must be burned through for every unique set of descriptors referenced by the graphics pipeline for every command list that could be either executing, queued for execution, or being recorded at any given time.

D3D12 leaves the responsibility of managing versioning to the application for the object types managed via descriptor heaps and descriptor tables. One benefit of this is that applications can choose to reuse descriptor table contents as much as possible rather than always defining a new descriptor table version for every command list submission. The root signature is a space that the D3D12 driver automatically versions.

The ability to bind multiple descriptor tables to the root signature (and thus to the pipeline) at a time allows applications to group and switch sets of descriptor references at different frequencies if desired. For example, an application could use a small number (perhaps just one) of large static descriptor tables that rarely change, or in which regions in the underlying descriptor heap memory are being populated as needed, with the use of dynamic indexing from the shader to select textures. At the same time, the application could maintain another class of resources where the set referenced by each draw call is switched from the CPU using the descriptor table versioning technique.

Out of Bounds Indexing

Out of bounds indexing of any descriptor table from the shader results in a largely undefined memory access, including the possibility of reading arbitrary in-process memory as if it is a hardware state descriptor and living with the consequence of what the hardware does with that. This could produce a device reset, but will not crash Windows.

Shader Derivatives and Divergent Indexing

If pixel shader invocations that are executing in a 2x2 stamp (to support derivative calculations) choose different texture indices to sample from out of a descriptor table, and if the selected sampler configuration and texture for any given pixel requires an LOD calculation from texture coordinate derivatives, then the LOD calculation and texture sampling process is done by the hardware independently for each texture lookup in the 2x2 stamp, which will impact performance.

Root Signatures

The root signature defines what resources are bound to the graphics pipeline. A root signature is configured by the app and links command lists to the resources the shaders require. Currently, there is one graphics and one compute root signature per app.

The root signature can reference root constants, root descriptors, and descriptor tables (a range in the descriptor heap). The root signature can also have space where applications can bind user defined constants (root constants) directly to shaders without having to go through descriptors and descriptor tables. Thirdly, the root signature can hold a very small amount of descriptors directly inside it (such as a CBV that is changing per draw), saving the application from having to put those descriptors in a descriptor heap.

The layout of the root signature is quite flexible, with some constraints imposed on less capable hardware. Regardless of the level of hardware, applications should always try to make the root signature as small as needed for maximum efficiency. Applications can trade off having more descriptor tables in the root signature but less room for root constants, or vice versa.

The contents of the root signature (the descriptor tables, root constants and root descriptors) that the application has bound automatically get versioned by the D3D12 driver whenever any part of the contents change between draw/dispatch calls. So each draw/dispatch gets a unique full set of root signature state.

The root constants are inline 32-bit values that show up in the shader as a constant buffer.

Ideally, there are groups of Pipeline State Objects (PSOs) that share the same root signature. After a root signature is set on the pipeline, all the bindings that it defines (descriptor tables, descriptors, constants) can each be individually set or changed, including inheritance into bundles.

An app can make its own tradeoff between how many descriptor tables it wants verses inline descriptors (which take more space but remove an indirection) verses inline constants (which have no indirection) they want in the root signature. Applications should use the root signature as sparingly as possible, relying on application controlled memory such as heaps and descriptor heaps pointing into them to represent bulk data. Exact size limits for the root signature are detailed later.

In this section

Topic	Description
Using a Root Signature	The root signature is the definition of an arbitrarily arranged collection of descriptor tables (including their layout), root constants and root descriptors. Each entry has a cost towards a maximum limit, so the application can trade off the balance between how many of each type of entry the root signature will contain.
Creating a Root Signature	Root signatures are a complex data structure containing nested structures. These can be defined programmatically using the data structure definition below (which includes methods to help initialize members). Alternatively, they can be authored in High Level Shading Language (HLSL) – giving the advantage that the compiler will validate early that the layout is compatible with the shader.
Root Signature Limits	The maximum size of a root signature is 64 DWORDs.
Using Constants Directly in the Root Signature	Applications can define root constants in the root signature, each as a set of 32-bit values. They appear in High Level Shading Language (HLSL) as a constant buffer. Note that constant buffers for historical reasons are viewed as sets of 4x32-bit values.
Using Descriptors Directly in the Root Signature	Applications can put descriptors directly in the root signature to avoid having to go through a descriptor heap. These descriptors take a lot of space in the root signature (see the root signature limits section), so applications have to use them sparingly.
Example Root Signatures	The following section shows root signatures varying in complexity from empty to completely full.

Specifying Root Signatures in HLSL	Specifying root signatures in HLSL Shader Model 5.1 is an alternative to specifying them in C++ code.

Using a Root Signature

The root signature is the definition of an arbitrarily arranged collection of descriptor tables (including their layout), root constants and root descriptors. Each entry has a cost towards a maximum limit, so the application can trade off the balance between how many of each type of entry the root signature will contain.

The root signature is an object that can be created by manual specification at the API. All shaders in a PSO must be compatible with the root layout specified with the PSO, or else the individual shaders must include embedded root layouts that match each other; otherwise, PSO creation will fail. One property of the root signature is that shaders don't have to know about it when authored, although root signatures can also be authored directly in shaders if desired. Existing shader assets do not require any changes to be compatible with root signatures. Shader Model 5.1 is introduced to provide some extra flexibility (dynamic indexing of descriptors from within shaders), and can be incrementally adopted starting from existing shader assets as desired.

Command List Semantic

At the beginning of a command list, the root signature is undefined. Graphics shaders have a separate root signature from the compute shader, each independently assigned on a command list. The root signature set on a command list or bundle must also match the currently set PSO at Draw/Dispatch; otherwise, the behavior is undefined. Transient root signature mismatches before Draw/Dispatch are fine - such as setting an incompatible PSO before switching to a compatible root signature (as long as these are compatible by the time Draw/Dispatch is called). Setting a PSO does not change the root signature. The application must call a dedicated API for setting the root signature.

Once a root signature has been set on a command list, the layout defines the set of bindings that the application is expected to provide, and which PSOs can be used (those compiled with the same layout) for the next draw/dispatch calls. For example, a root signature could be defined by the application to have the following entries. Each entry is referred to as a "slot".

- [0] A CBV descriptor inline (root descriptors)
- [1] A descriptor table containing 2 SRVs, 1 CBVs, and 1 UAV
- [2] A descriptor table containing 1 sampler
- [3] A 4x32-bit collection of root constants
- [4] A descriptor table containing an unspecified number of SRVs

In this case, before being able to issue a Draw/Dispatch, the application is expected to set the appropriate binding to each of the slots [0..4] that the application defined with its current root signature. For instance, at slot [1], a descriptor table must be bound, which is a contiguous region in a descriptor heap that contains (or will contain at execution) 2 SRVs, 1 CBVs, and 1 UAV. Similarly, descriptor tables must be set at slots [2] and [4].

The application can change part of the root signature bindings at a time (the rest remain unchanged). For example, if the only thing that needs to change between draws is one of the constants at slot [2], that is all the application needs to rebind. As discussed previously, the driver/hardware versions all root signature bind state as it is modified automatically. If a root signature is changed on a command list, all previous root signature bindings become stale and all newly expected bindings must be set before Draw/Dispatch; otherwise, the behavior is undefined. If the root signature is redundantly set to the same one currently set, existing root signature bindings do not become stale.

Bundle Semantics

Bundles inherit the command list's root signature bindings (what is bound to the various slots in the example above). If a bundle needs to change some of the inherited root signature bindings, it must first set the root signature to be the same as the calling command list (the inherited bindings do not become stale). If the bundle sets the root signature to be different than the calling command list, that has the same effect as changing the root signature on the command list described above: all previous root signature bindings are stale and newly expected bindings must be set before Draw/Dispatch; otherwise, the behavior is undefined. If a bundle does not need to change any root signature bindings, it does not need to set the root signature.

Coming out of a bundle, any root layout changes and/or binding changes a bundle makes are inherited back to the calling command list when a bundle finishes executing. The API syntax for authoring/referencing root signatures is described later.

For more information on inheritance, refer to the **Graphics pipeline state inheritance** section of Managing Graphics Pipeline State in Direct3D 12.

Creating a Root Signature

Root signatures are a complex data structure containing nested structures. These can be defined programmatically using the data structure definition below (which includes methods to help initialize members). Alternatively, they can be authored in High Level Shading Language (HLSL) – giving the advantage that the compiler will validate early that the layout is compatible with the shader.

The API for creating a root signature takes in a serialized (self contained, pointer free) version of the layout description described below. A method is provided for generating this serialized version from the C++ data structure, but another way to obtain a serialized root signature definition is to retrieve it from a shader that has been compiled with a root signature.

- Descriptor Table Bind Types
- Descriptor Range
- Descriptor Table Layout
- Root Constants
- Root Descriptor
- Shader Visibility
- Root Signature Definition
- Root Signature Data Structure Serialization / Deserialization
- Root Signature Creation API
- Root Signature in Pipeline state objects
- Related topics

Descriptor Table Bind Types

The enum D3D12_DESCRIPTOR_RANGE_TYPE defines the types of descriptors that can be referenced as part of a descriptor table layout definition.

It is a range so that, for example if part of a descriptor table a descriptor table has 100 SRVs, that range can be declared in one entry rather than 100. So a descriptor table definition is a collection of ranges.

```
typedef enum D3D12_DESCRIPTOR_RANGE_TYPE {
    D3D12_DESCRIPTOR_RANGE_TYPE_SRV,
    D3D12_DESCRIPTOR_RANGE_TYPE_UAV,
    D3D12_DESCRIPTOR_RANGE_TYPE_CBV,
    D3D12_DESCRIPTOR_RANGE_TYPE_SAMPLER
} D3D12_DESCRIPTOR_RANGE_TYPE;
```

Descriptor Range

The D3D12_DESCRIPTOR_RANGE structure defines a range of descriptors of a given type (such as SRVs) within a descriptor table.

The `D3D12_DESCRIPTOR_RANGE_OFFSET_APPEND` #define can typically be used for the `OffsetInDescriptorsFromTableStart` parameter of D3D12_DESCRIPTOR_RANGE. This means append the descriptor range being defined after the previous one in the descriptor table. If the application wants to alias descriptors or for some reason wants to skip slots, it can set `OffsetInDescriptorsFromTableStart` to whatever offset is desired. Defining overlapping ranges of different types is invalid.

The set of shader registers specified by the combination of `RangeType`, `NumDescriptors`, `BaseShaderRegister`, and `RegisterSpace` cannot conflict or overlap across any declarations in a root signature that have common D3D12_SHADER_VISIBILITY (refer to the shader visibility section below).

Descriptor Table Layout

The D3D12_ROOT_DESCRIPTOR_TABLE structure declares the layout of a descriptor table as a collection of descriptor ranges that appear one after the other in a descriptor heap. Samplers are not allowed in the same descriptor table as CBV/UAV/SRVs.

This struct is used when the root signature slot type is set to `D3D12_ROOT_PARAMETER_TYPE_DESCRIPTOR_TABLE`.

To set a graphics (CBV, SRV, UAV, Sampler) descriptor table, use ID3D12GraphicsCommandList::SetGraphicsRootDescriptorTable.

To set a compute descriptor table, use ID3D12GraphicsCommandList::SetComputeRootDescriptorTable.

Root Constants

The D3D12_ROOT_CONSTANTS structure declares constants inline in the root signature that appear in shaders as one constant buffer.

This struct is used when the root signature slot type is set to `D3D12_ROOT_PARAMETER_TYPE_32BIT_CONSTANTS`.

Root Descriptor

The D3D12_ROOT_DESCRIPTOR structure declares descriptors (that appear in shaders) inline in the root signature.

This struct is used when the root signature slot type is set to `D3D12_ROOT_PARAMETER_TYPE_CBV`, `D3D12_ROOT_PARAMETER_TYPE_SRV` or `D3D12_ROOT_PARAMETER_TYPE_UAV`.

Shader Visibility

The member of D3D12_SHADER_VISIBILITY enum set into the shader visibility parameter of D3D12_ROOT_PARAMETER determines which shaders see the contents of a given root signature slot. Compute always uses _ALL (since there is only one active stage). Graphics can choose, but if it uses _ALL, all shader stages see whatever is bound at the root signature slot.

One use of shader visibility is to help with shaders that are authored expecting different bindings per shader stage using an overlapping namespace. For example, a vertex shader may declare: Texture2D foo : register(t0);" and the pixel shader may also declare: Texture2D bar : register(t0);

If the application makes a root signature binding to t0 VISIBILITY_ALL, both shaders see the same texture. If the shader defines actually wants each shader to see different textures, it can define 2 root signature slots with VISIBILITY_VERTEX and _PIXEL. No matter what the visibility is on a root signature slot, it always has the same cost (cost only depending on what the SlotType is) towards one fixed maximum root signature size.

On low end D3D11 hardware, SHADER_VISIBILITY is also taken into account used when validating the sizes of descriptor tables in a root layout, since some D3D11 hardware can only support a maximum amount of bindings per-stage. These restrictions are only imposed when running on low tier hardware and do not limit more modern hardware at all.

If a root signature has multiple descriptor tables defined that overlap each other in namespace (the register bindings to the shader) and any one of them specifies _ALL for visibility, the layout is invalid (creation will fail).

Root Signature Definition

The D3D12_ROOT_SIGNATURE_DESC structure can contain descriptor tables and inline constants, each slot type defined by the enumD3D12_ROOT_PARAMETER_TYPE.

To initiate a root signature slot, refer to the methods of D3D12_ROOT_PARAMETER.

A number of flags limit the access of certain shaders to the root signature, refer to D3D12_ROOT_SIGNATURE_FLAGS.

Root Signature Data Structure Serialization / Deserialization

The methods described in this section are exported by D3D12Core.dll and provide methods for serializing and deserializing a root signature data structure.

The serialized form is what is passed into the API when creating a root signature. If a shader has been authored with a root signature in it (when that capability is added), then the compiled shader will contain a serialized root signature in it already.

If an application procedurally generates a D3D12_ROOT_SIGNATURE_DESC data structure, it must make the serialized form usingD3D12SerializeRootSignature. The output of that can be passed into ID3D12Device::CreateRootSignature.

If an application has a serialized root signature already, or has a compiled shader that contains a root signature and wishes to programmatically discover the layout definition (known as "reflection"), D3D12CreateRootSignatureDeserializer can be called. This generates an ID3D12RootSignatureDeserializerinterface, which contains a method to return the deserialized D3D12_ROOT_SIGNATURE_DESC data structure. The interface owns the lifetime of the deserialized data structure.

Root Signature Creation API

The **ID3D12Device::CreateRootSignature** API takes in a serialized version of a root signature.

Root Signature in Pipeline state objects

The methods to create pipeline state (**ID3D12Device::CreateGraphicsPipelineState** and **ID3D12Device::CreateComputePipelineState**) take an optional**ID3D12RootSignature** interface as an input parameter (stored in a **D3D12_GRAPHICS_PIPELINE_STATE_DESC** structure).

If a root signature is passed into one of the create pipeline state methods, this root signature is validated against all the shaders in the PSO for compatibility and given to the driver to use with all the shaders. If any of the shaders has a different root signature in it, it gets replaced by the root signature passed in at the API. If a root signature is not passed in, all shaders passed in must have a root signature and they must match – this will be given to the driver. Setting a PSO on a command list or bundle does not change the root signature. That is accomplished by the methods **SetGraphicsRootSignature** and**SetComputeRootSignature**. By the time Draw*()/Dispatch*() is invoked, the application must ensure that the current PSO matches the current root signature; otherwise, the behavior is undefined.

Root Signature Limits

The maximum size of a root signature is 64 DWORDs.

This maximum size is chosen to prevent abuse of the root signature as a way of storing bulk data. Each entry in the root signature has a cost towards this 64 DWORD limit:

- Descriptor tables cost 1 DWORD each.
- Root constants cost 1 DWORD each, since they are 32-bit values.
- Root descriptors (64-bit GPU virtual addresses) cost 2 DWORDs each.

Using Constants Directly in the Root Signature

Applications can define root constants in the root signature, each as a set of 32-bit values. They appear in High Level Shading Language (HLSL) as a constant buffer. Note that constant buffers for historical reasons are viewed as sets of 4x32-bit values.

Each set of user constants is treated as a scalar array of 32 -bit values, dynamically indexable and read-only from the shader. Out of bounds indexing a given set of root constants produces undefined results. In HLSL, data structure definitions can be provided for the user constants to give them types. For example, if the root signature defines a set of 4 root constants, HLSL can overlay the following structure on them.

```
struct DrawConstants {
    uint foo;
    float2 bar;
    int moo;
};
ConstantBuffer<DrawConstants> myDrawConstants : register(b1, space0);
```

Arrays are not permitted in constant buffers that get mapped onto root constants since dynamic indexing in the root signature space is not supported. For example, it is invalid to have an entry in the constant buffer like `float myArray[2];`. A constant buffer that is mapped to root constants cannot itself be an array; therefore, it is invalid to map `cbuffer myCBArray[2]` into root constants.

Constants can be partially set. For example, if the root signature defines four 32-bit values at *RootTableBindSlot* 2, then any subset of the four constants can be set at a time (the others remain unchanged). This can be useful in bundles that inherit root signature state and can partially change it.

The following APIs (from the **ID3D12GraphicsCommandList** interface) are for setting constants directly on the root signature:

- SetGraphicsRoot32BitConstant
- SetGraphicsRoot32BitConstants
- SetComputeRoot32BitConstant
- SetComputeRoot32BitConstants

Also, refer to the **D3D12_ROOT_CONSTANTS** structure.

Using Descriptors Directly in the Root Signature

Applications can put descriptors directly in the root signature to avoid having to go through a descriptor heap. These descriptors take a lot of space in the root signature (see the root signature limits section), so applications have to use them sparingly.

An example usage would be to place a CBV that is changing per draw in the root layout so that descriptor heap space doesn't have to be allocated by the application per draw (and save pointing a descriptor table at the new location in the descriptor heap). By putting something in the root signature, the application is merely handing the versioning responsibility to the driver, but this is infrastructure that they already have.

For rendering that uses extremely few resources, descriptor table / heap use may not be needed at all if all the needed descriptors can be placed directly in the root signature.

The only types of descriptors supported in the root signature are CBVs and SRV/UAVs of buffer resources, where the SRV/UAV format contains only 32 bit FLOAT/UINT/SINT components. There is no format conversion. UAVs in the root cannot have counters associated with them. Descriptors in the root signature appear each as individual separate descriptors - they cannot be dynamically indexed.

```
struct SceneData {
   uint foo;
   float bar[2];
   int moo;
};
ConstantBuffer<SceneData> mySceneData : register(b6);
```

In the above example, `mySceneData` cannot be declared as an array, as in `cbuffer mySceneData[2]` if it is going to be mapped onto a descriptor in the root signature, since indexing across descriptors is not supported in the root signature. The application can define separate individual constant buffers and define them each as a separate entry in the root signature if desired. Note that within `mySceneData` above, there is an array `bar[2]`. Dynamic indexing within the constant buffer is valid - a descriptor in the root signature behaves just like the same descriptor would behave if accessed through a descriptor heap. This is in contrast with inlining constants directly in the root signature, which also appears like a constant buffer except with the constraint that dynamic indexing within the inlined constants is not permitted, so `bar[2]` would not be allowed there.

The following APIs (from the **ID3D12GraphicsCommandList** interface) are for setting descriptors directly on the root signature:

- SetComputeRootConstantBufferView
- SetGraphicsRootConstantBufferView
- SetComputeRootShaderResourceView
- SetGraphicsRootShaderResourceView
- SetComputeRootUnorderedAccessView
- SetGraphicsRootUnorderedAccessView

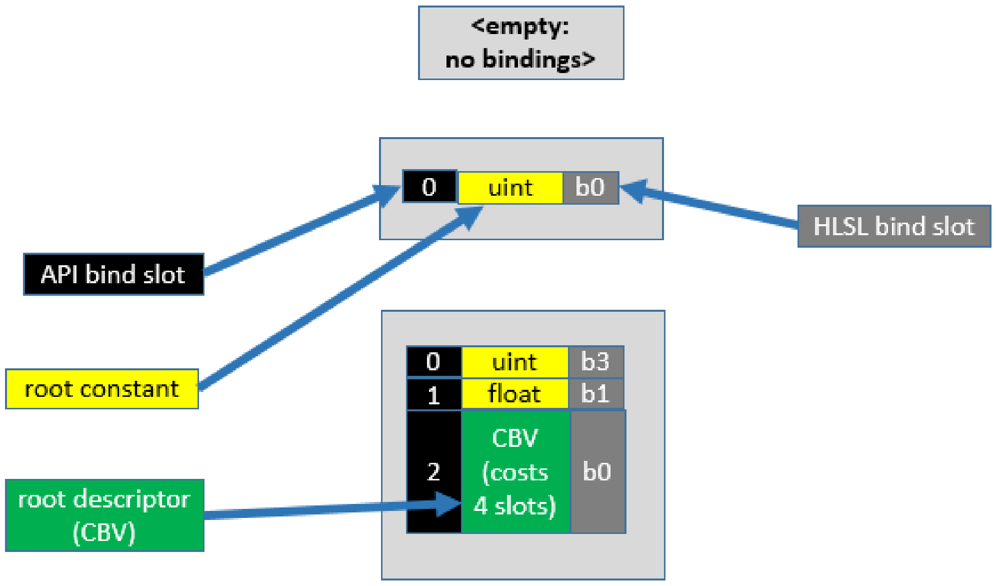
Example Root Signatures

The following section shows root signatures varying in complexity from empty to completely full.

- A root signature under construction
- A simple root signature
- A more complex root signature
- Related topics

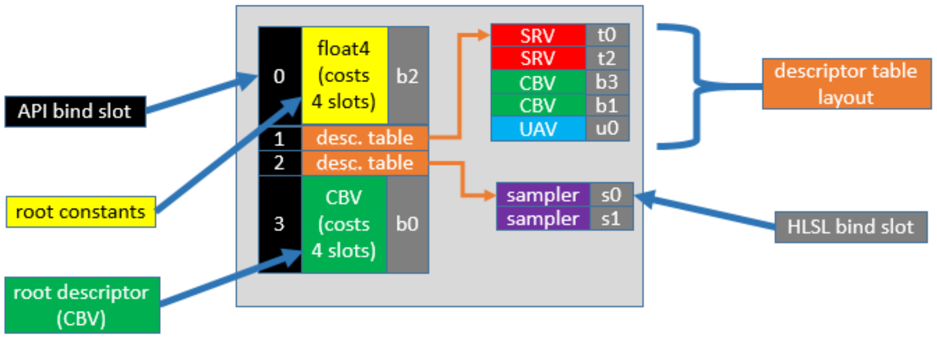
A root signature under construction

The following image shows the format of a root signature under construction, with some constants and a root descriptor being added to it.



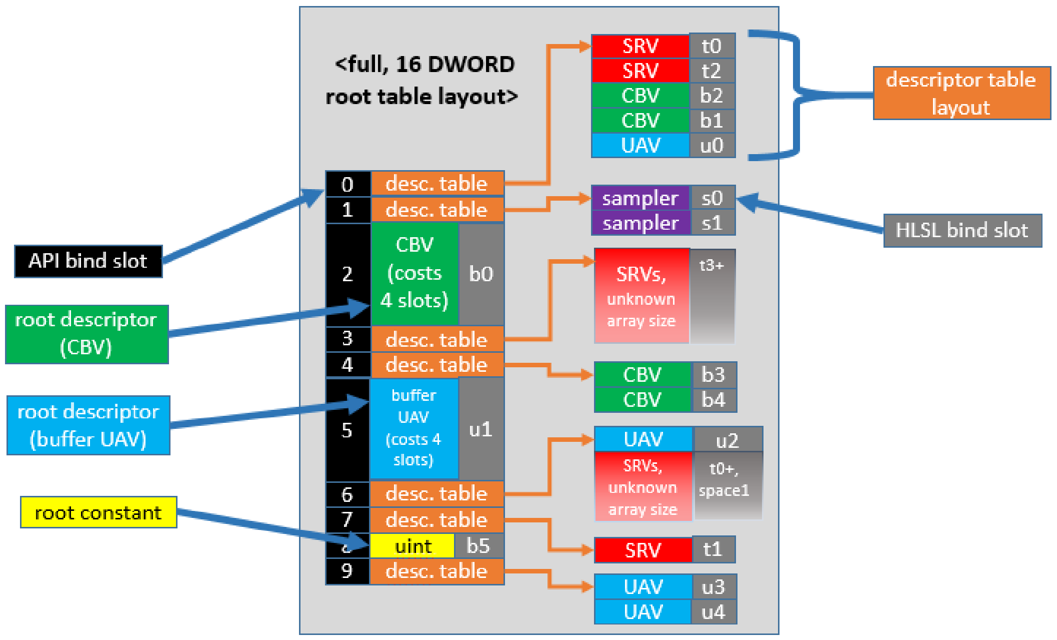
A simple root signature

The following example shows that descriptor tables in the root signature have their contents declared also as part of the root signature (the orange arrows). At execution time, when a descriptor table (which is simply a pointer into a descriptor heap and a length) has been defined, the application is responsible for placing the expected descriptor types from the layout declaration at identified descriptor heap range.



A more complex root signature

In the following example, there are two descriptor tables that contain unbounded arrays. In these cases, the descriptor table declaration is just stating that the shader doesn't know how many descriptors there will be when compiled. The shader will use dynamic indexing at execution time when the application decides how many descriptors it actually requires. The second unbounded array is declared at HLSL bind slot t0, space1. "space1" allows the unbounded array to not conflict with any other HLSL bindpoints. In this example, t0-t[infinity] in the default bind space, space0, are used up by other descriptor table declarations of t0, t1, t2 and t3+.



Specifying Root Signatures in HLSL

Specifying root signatures in HLSL Shader Model 5.1 is an alternative to specifying them in C++ code.

- Specifying Root Signatures in HLSL
  - RootFlags
  - Root Constants
  - Visibility
  - Root-level CBV

Specifying Root Signatures in HLSL

A root signature can be specified in HLSL as a string. The string contains a collection of comma-separated clauses that describe root signature constituent components. Here is an example:

```
#define MyRS1 "RootFlags( ALLOW_INPUT_ASSEMBLER_INPUT_LAYOUT | " \
                         "DENY_VERTEX_SHADER_ROOT_ACCESS), " \
              "CBV(b0, space = 1), " \
              "SRV(t0), " \
              "UAV(u0), " \
              "DescriptorTable( CBV(b1), " \
                               "SRV(t1, numDescriptors = 8), " \
                               "UAV(u1, numDescriptors = unbounded)), " \
              "DescriptorTable(Sampler(s0, space=1, numDescriptors = 4)), " \
              "RootConstants(num32BitConstants=3, b10), " \
              "StaticSampler(s1)," \
              "StaticSampler(s2, " \
                             "addressU = TEXTURE_ADDRESS_CLAMP, " \
                             "filter = FILTER_MIN_MAG_MIP_LINEAR )"
```

There are two mechanisms to compile an HLSL root signature. First, it is possible to attach a root signature string to a particular shader via the *RootSignature*attribute:

```
[RootSignature(MyRS1)]
float4 main(float4 coord : COORD) : SV_Target {
…
}
```

The compiler will create and verify the root signature blob for the shader and embed it alongside the shader byte code into the shader blob. The compiler supports root signature syntax for shader model 5.0 and higher. If a root signature is embedded in a shader model 5.0 shader and that shader is sent to the D3D11 runtime, as opposed to D3D12, the root signature portion will get silently ignored by D3D11.

The other mechanism is to create a standalone root signature blob, perhaps to reuse it with a large set of shaders, saving space. The compiler supports a new rootsig_1_0 shader model, and the name of the define string is specified via the usual /E argument. For example:

```
fxc.exe /T rootsig_1_0 MyRS1.hlsl /E MyRS1 /Fo MyRS1.fxo
```

Note that the root signature string define can also be passed on the command line, e.g, /D MyRS1="…".

The HLSL root signature language closely corresponds to the C++ root signature APIs and has equivalent expressive power. The root signature is specified as a sequence of clauses, separated by comma. The order of clauses is important, as the order of parsing determines the slot position in the root signature. Each clause takes one or more named parameters. The order of parameters is not important, however.

RootFlags

The optional RootFlags clause takes either 0 (the default value to indicate no flags), or one or several of predefined root flags values, connected via the OR '|' operator. The allowed root flag values are defined by D3D12_ROOT_SIGNATURE_FLAGS.

For example:

```
RootFlags(0)    // default value – no flags
RootFlags(ALLOW_INPUT_ASSEMBLER_INPUT_LAYOUT)
RootFlags(ALLOW_INPUT_ASSEMBLER_INPUT_LAYOUT | DENY_VERTEX_SHADER_ROOT_ACCESS)
```

Root Constants

The *RootConstants* clause specifies root constants in the root signature. Two mandatory parameters are: *num32BitConstants* and *bReg* (the register corresponding to *BaseShaderRegister* in C++ APIs) of the *cbuffer*. The space (*RegisterSpace* in C++ APIs) and visibility (*ShaderVisibility* in C++) parameters are optional, and the default values are:

```
RootConstants(num32BitConstants=N, bReg [, space=0, visibility=SHADER_VISIBILITY_ALL ])
```

Visibility

Visibility is an optional parameter that can have one the values from D3D12_SHADER_VISIBILITY.

Root-level CBV

The CBV (constant buffer view) clause specifies a root-level constant buffer b-register Reg entry. Note that this is a scalar entry; it is not possible to specify a range for the root level.

```
CBV(bReg [, space=0, visibility=VISIBILITY_ALL ])
```

Root-level SRV

The SRV (shader resource view) clause specifies a root-level SRV t-register Reg entry. Note that this is a scalar entry; it is not possible to specify a range for the root level.

```
SRV(tReg [, space=0, visibility=VISIBILITY_ALL ])
```

Root-level UAV

The UAV (unordered access view) clause specifies a root-level UAV u-register Reg entry. Note that this is a scalar entry; it is not possible to specify a range for the root level.

```
UAV(uReg [, space=0, visibility=VISIBILITY_ALL ])
```

Descriptor Table

The DescriptorTable clause is itself a list of comma-separated descriptor table clauses, as well as an optional visibility parameter. The *DescriptorTable* clauses include CBV, SRV, UAV, and Sampler. Note that their parameters differ from those of the root-level clauses.

```
DescriptorTable( DTClause1, [ DTClause2, … DTClauseN, visibility=SHADER_VISIBILITY_ALL ] )
```

The Descriptor Table CBV has the following syntax:

```
CBV(bReg [, numDescriptors=1, space=0, offset=0 ])
```

The mandatory parameter *bReg* specifies the start Reg of the cbuffer range. The *numDescriptors* parameter specifies the number of descriptors in the contiguous cbuffer range; the default value being 1. The entry declares a cbuffer range [Reg, Reg + numDescriptors - 1], when *numDescriptors* is a number. If*numDescriptors* is equal to 'unbounded', the range is [Reg, UINT_MAX]. The offset field represents the *OffsetInDescriptorsFromTableStart* field in C++ APIs.

The Descriptor Table SRV has the following syntax:

```
SRV(tReg [, numDescriptors=1, space=0, offset=0 ])
```

This is similar to the descriptor table CBV entry, except the specified range is for shader resource views.

The Descriptor Table UAV has the following syntax:

```
UAV(uReg [, numDescriptors=1, space=0, offset=0 ])
```

This is similar to the descriptor table CBV entry, except the specified range is for unordered access views.

The Descriptor Table `Sampler` has the following syntax:

```
Sampler(sReg [, numDescriptors=1, space=0, offset=0 ])
```

This is similar to the descriptor table CBV entry, except the specified range is for shader samplers. Note that Samplers can't be mixed with other types of descriptors in the same descriptor table (since they are in a separate descriptor heap).

Static Sampler

The static sampler represents the D3D12_STATIC_SAMPLER_DESC structure. The mandatory parameter for *StaticSampler* is a scalar, sampler s-register Reg. Other parameters are optional with default values given above. Each field accepts a set of predefined enums.

```
StaticSampler( sReg,
              [ filter = FILTER_ANISOTROPIC,
                addressU = TEXTURE_ADDRESS_WRAP,
                addressV = TEXTURE_ADDRESS_WRAP,
                addressW = TEXTURE_ADDRESS_WRAP,
                maxAnisotropy = 16,
                comparisonFunc = COMPARISON_LESS_EQUAL,
                borderColor = STATIC_BORDER_COLOR_OPAQUE_WHITE,
                space = 0,
                visibility = SHADER_VISIBILITY_ALL ])
```

The filter field can be one of D3D12_FILTER.

The address fields can each be one of D3D12_TEXTURE_ADDRESS_MODE.

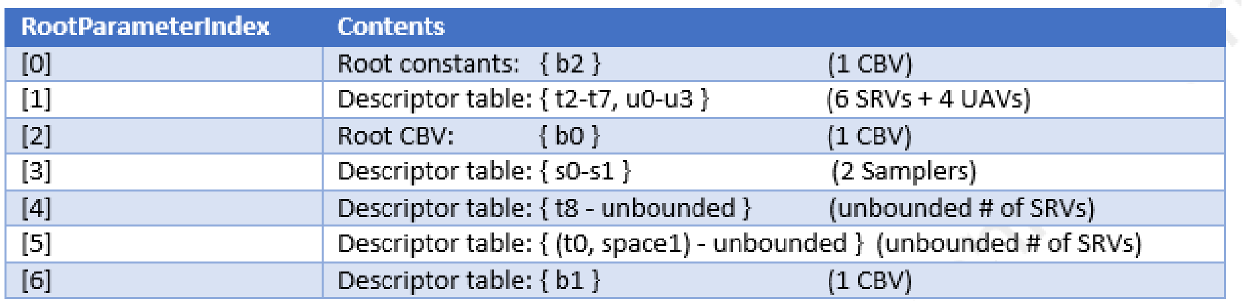
The comparison function can be one of D3D12_COMPARISON_FUNC.

The border color field can be one of D3D12_STATIC_BORDER_COLOR.

Visibility can be one of D3D12_SHADER_VISIBILITY.

Code for Defining a Root Signature

The example below shows how to create a root signature with the following format:



RootParameterIndex	Contents
[0]	Root constants: { b2 } (1 CBV)
[1]	Descriptor table: { t2-t7, u0-u3 } (6 SRVs + 4 UAVs)
[2]	Root CBV: { b0 } (1 CBV)
[3]	Descriptor table: { s0-s1 } (2 Samplers)
[4]	Descriptor table: { t8 - unbounded } (unbounded # of SRVs)
[5]	Descriptor table: { (t0, space1) - unbounded } (unbounded # of SRVs)
[6]	Descriptor table: { b1 } (1 CBV)

If most parts of the root signature get used most of the time it can be better than having to switch the root signature too frequently. Applications should sort entries in the root signature from most frequently changing to least. When an app changes the bindings to any part of the root signature, the driver may have to make a copy of some or all of root signature state, which can become a nontrivial cost when multiplied across many state changes.

In addition, the root signature will define a static sampler that does anisotropic texture filtering at shader register s3.

Once this root signature is bound, descriptor tables, root CBV and constants can be assigned to the [0..6] parameter space. e.g. descriptor tables (ranges in a descriptor heap) can be bound at each of root parameters [1] and [3..6].

```
D3D12_DESCRIPTOR_RANGE DescRange[6];

DescRange[0].Init(D3D12_DESCRIPTOR_RANGE_TYPE_SRV,6,2); // t2-t7
DescRange[1].Init(D3D12_DESCRIPTOR_RANGE_TYPE_UAV,4,0); // u0-u3
DescRange[2].Init(D3D12_DESCRIPTOR_RANGE_TYPE_SAMPLER,2,0); // s0-s1
DescRange[3].Init(D3D12_DESCRIPTOR_RANGE_TYPE_SRV,-1,8); // t8-unbounded
DescRange[4].Init(D3D12_DESCRIPTOR_RANGE_TYPE_SRV,-1,0,1); // (t0,space1)-unbounded
DescRange[5].Init(D3D12_DESCRIPTOR_RANGE_TYPE_CBV,1,1); // b1

D3D12_ROOT_PARAMETER RP[7];

RP[0].InitAsConstants(3,2); // 3 constants at b2
RP[1].InitAsDescriptorTable(2,&DescRange[0]); // 2 ranges t2-t7 and u0-u3
RP[2].InitAsConstantBufferView(0); // b0
RP[3].InitAsDescriptorTable(1,&DescRange[2]); // s0-s1
RP[4].InitAsDescriptorTable(1,&DescRange[3]); // t8-unbounded
RP[5].InitAsDescriptorTable(1,&DescRange[4]); // (t0,space1)-unbounded
RP[6].InitAsDescriptorTable(1,&DescRange[5]); // b1

D3D12_STATIC_SAMPLER_DESC StaticSamplers[1];
StaticSamplers[0].Init(3, D3D12_FILTER_ANISOTROPIC); // s3
D3D12_ROOT_SIGNATURE RootSig(7,RP,1,StaticSamplers);
ID3DBlob* pSerializedRootSig;
D3D12SerializeRootSignature(&RootSig,pSerializedRootSig);

ID3D12RootSignature* pRootSignature;
hr = pDevice->CreateRootSignature(
                   pSerializedRootSig->GetBufferPointer(),
                   pSerializedRootSig->GetBufferSize(),
                  __uuidof(ID3D12RootSignature),
                  &pRootSignature);
```

The following code illustrates how the above root signature might be used on a graphics command list.

```
InitializeMyDescriptorHeapContentsAheadOfTime(); // for simplicity of the
                                                 // example
CreatePipelineStatesAhreadOfTime(pRootSignature); // The root signature is passed into
                                    // shader/pipeline state creation
...

ID3D12DescriptorHeap* pHeaps[2] = {pCommonHeap, pSamplerHeap};
pGraphicsCommandList->SetDescriptorHeaps(pHeaps, 2);
pGraphicsCommandList->SetGraphicsRootSignature(pRootSignature);
pGraphicsCommandList->SetGraphicsRootDescriptorTable(
                        6, heapOffsetForMoreData,DescRange[5].NumDescriptors);
pGraphicsCommandList->SetGraphicsRootDescriptorTable(5, heapOffsetForMisc, 5000);
pGraphicsCommandList->SetGraphicsRootDescriptorTable(4, heapOffsetForTerrain, 20000);
pGraphicsCommandList->SetGraphicsRootDescriptorTable(
                        3, heapOffsetForSamplers,DescRange[2].NumDescriptors);
pGraphicsCommandList->SetComputeRootConstantBufferView(2, pDynamicCBHeap,&CBVDesc);

MY_PER_DRAW_STUFF stuff;
InitMyPerDrawStuff(&stuff);
pGraphicsCommandList->SetSetGraphicsRoot32BitConstants(
                        0,
                         &stuff,
                         0,
                         RTSlot[0].Constants.Num32BitValues);
```

```
SetMyRTVAndOtherMiscBindings();

for(UINT i = 0; i < numObjects; i++) {
    pGraphicsCommandList->SetPipelineState(PSO[i]);
    pGraphicsCommandList->SetGraphicsRootDescriptorTable(
                    1,
                     heapOffsetForFooAndBar[i],DescRange[1].NumDescriptors);
    pGraphicsCommandList->SetGraphicsRoot32BitConstant(0, &i, 1, drawIDOffset);
    SetMyIndexBuffers(i);
    pGraphicsCommandList->DrawIndexedInstanced(...);
}
```

Capability Querying

Applications can discover the level of support for resource binding, and many other features, via the ID3D12Device::CheckFeatureSupport call.

- Resource binding tiers
- Check feature support
  - D3D12_FEATURE_DATA_D3D12_OPTIONS
  - D3D12_FEATURE_DATA_ARCHITECTURE
  - D3D12_FEATURE_DATA_FEATURE_LEVELS
  - D3D12_FEATURE_DATA_FORMAT_SUPPORT
  - D3D12_FEATURE_DATA_MULTISAMPLE_QUALITY_LEVELS
  - D3D12_FEATURE_DATA_FORMAT_INFO
  - D3D12_FEATURE_DATA_GPU_VIRTUAL_ADDRESS_SUPPORT
- Hardware support for DXGI Formats
- Related topics

Resource binding tiers

For resource binding, each tier is a superset of lower tiers in functionality, so code that works on a given tier works on any higher tier unchanged.

The number of resource binding tiers is stored in the D3D12_RESOURCE_BINDING_TIER enum.

To check the resource binding tier, use code such as

```
D3D12_FEATURE_DATA_D3D12_OPTIONS options;
mDevice->CheckFeatureSupport(D3D12_FEATURE_D3D12_OPTIONS,&options, sizeof(options));
switch (options.ResourceBindingTier) {
    case D3D12_RESOURCE_BINDING_TIER_1:
        // Tier 1 is supported
        break;

    case D3D12_RESOURCE_BINDING_TIER_2:
        // Tier 1 and 2 are supported
        break;

    case D3D12_RESOURCE_BINDING_TIER_3:
        // Tier 1, 2 and 3 are supported
        break;
}
```

The following section shows the full range of options for checking feature support.

Check feature support

Select one member of the enum D3D12_FEATURE as input to CheckFeatureSupport to determine what features to request support information on. It can be one of the following.

- D3D12_FEATURE_DATA_D3D12_OPTIONS
- D3D12_FEATURE_DATA_ARCHITECTURE
- D3D12_FEATURE_DATA_FEATURE_LEVELS
- D3D12_FEATURE_DATA_FORMAT_SUPPORT
- D3D12_FEATURE_DATA_MULTISAMPLE_QUALITY_LEVELS
- D3D12_FEATURE_DATA_FORMAT_INFO
- D3D12_FEATURE_DATA_GPU_VIRTUAL_ADDRESS_SUPPORT
- D3D12_FEATURE_DATA_D3D12_OPTIONS
- D3D12_FEATURE_DATA_D3D12_OPTIONS

The D3D12_FEATURE_DATA_D3D12_OPTIONS structure holds a range of supported feature data.

```
typedef struct D3D12_FEATURE_DATA_D3D12_OPTIONS {
    /* [annotation] */
    _Out_  BOOL DoublePrecisionFloatShaderOps;
    /* [annotation] */
    _Out_  BOOL OutputMergerLogicOp;
    /* [annotation] */
    _Out_  D3D12_SHADER_MIN_PRECISION_SUPPORT MinPrecisionSupport;
    /* [annotation] */
    _Out_  D3D12_TILED_RESOURCES_TIER TiledResourcesTier;
    /* [annotation] */
    _Out_  D3D12_RESOURCE_BINDING_TIER ResourceBindingTier;
    /* [annotation] */
    _Out_  BOOL PSSpecifiedStencilRefSupported;
    /* [annotation] */
    _Out_  BOOL TypedUAVLoadAdditionalFormats;
    /* [annotation] */
    _Out_  BOOL ROVsSupported;
    /* [annotation] */
    _Out_  D3D12_CONSERVATIVE_RASTERIZATION_TIER ConservativeRasterizationTier;
    /* [annotation] */
    _Out_  UINT MaxGPUVirtualAddressBitsPerResource;
    /* [annotation] */
    _Out_  BOOL StandardSwizzle64KBSupported;
} D3D12_FEATURE_DATA_D3D12_OPTIONS;
```

Refer to the following enums.

- D3D12_SHADER_MIN_PRECISION_SUPPORT
- D3D12_TILED_RESOURCES_TIER
- D3D12_RESOURCE_BINDING_TIER
- D3D12_CONSERVATIVE_RASTERIZATION_TIER

D3D12_FEATURE_DATA_ARCHITECTURE

The D3D12_FEATURE_DATA_ARCHITECTURE structure holds three booleans referencing tile based rendering and UMA support.

D3D12_FEATURE_DATA_FEATURE_LEVELS

The D3D12_FEATURE_DATA_FEATURE_LEVELS structure holds Direct3D feature level support information.

D3D12_FEATURE_DATA_FORMAT_SUPPORT

The D3D12_FEATURE_DATA_FORMAT_SUPPORT structure holds data format support details, and refer to the following enums.

- D3D12_FORMAT_SUPPORT1
- D3D12_FORMAT_SUPPORT2

Refer to Typed Unordered Access View Loads for an example use of this structure.

D3D12_FEATURE_DATA_MULTISAMPLE_QUALITY_LEVELS

The D3D12_FEATURE_DATA_MULTISAMPLE_QUALITY_LEVELS structure holds multi-sampling quality level support information, and refer to theD3D12_MULTISAMPLE_QUALITY_LEVELS_FLAG enum.

D3D12_FEATURE_DATA_FORMAT_INFO

The D3D12_FEATURE_DATA_FORMAT_INFO structure describes the supported DXGI_FORMAT and plane count.

For example,

C++

```cpp
inline UINT8 D3D12GetFormatPlaneCount(_In_ ID3D12Device* pDevice, DXGI_FORMAT Format) {
    D3D12_FEATURE_DATA_FORMAT_INFO formatInfo = {Format};
    if (FAILED(pDevice->CheckFeatureSupport(D3D12_FEATURE_FORMAT_INFO, &formatInfo, sizeof(formatInfo)))) {
        return 0;
    }
    return formatInfo.PlaneCount;
}
```

D3D12_FEATURE_DATA_GPU_VIRTUAL_ADDRESS_SUPPORT

The D3D12_FEATURE_DATA_GPU_VIRTUAL_ADDRESS_SUPPORT structure provides the maximum number of bits for a virtual GPU address for resources, and for processes.

Hardware support for DXGI Formats

To view tables of DXGI formats and hardware features, refer to:

- DXGI Format Support for Direct3D Feature Level 12.1 Hardware
- DXGI Format Support for Direct3D Feature Level 12.0 Hardware
- DXGI Format Support for Direct3D Feature Level 11.1 Hardware
- DXGI Format Support for Direct3D Feature Level 11.0 Hardware
- Hardware Support for Direct3D 10Level9 Formats
- Hardware Support for Direct3D 10.1 Formats
- Hardware Support for Direct3D 10 Formats

Resource Binding in HLSL

This section describes some specific features of using High Level Shading Language (HLSL) Shader Model 5.1 with D3D12.

- Resource Types
- Resource Aliasing
- Divergence and derivatives
- UAVs in pixel shaders

Resource Types

Shader Model 5 (SM5.0) resource syntax uses the "register" keyword to relay important information about the resource to the HLSL compiler. For example, the following statement declares an array of four textures bound at slots t3, t4, t5, and t6.:

```
Texture2D<float4> tex1[4] : register(t3)
```

Shader Model 5.1 (SM5.1) resource syntax in HLSL is based on existing register resource syntax, to allow easier porting. D3D12 resources in HLSL are bound to virtual registers within logical register spaces:

- t – for shader resource views (SRV)
- s – for samplers
- u – for unordered access views (UAV)
- b – for constant buffer views (CBV)

A resource declaration may be a scalar, a 1D array, or a multidimensional array:

```
Texture2D<float4> tex1        : register(t3,  space0)
Texture2D<float4> tex2[4]        : register(t10)
Texture2D<float4> tex3[7][5][3]    : register(t20, space1)
```

SM5.1 uses the same resource types and element types as SM5.0. Declaration limits are much more permissive now and constrained by the runtime/hardware limits, which makes the maximum range size to be 227 entries. The "space" keyword specifies to which logical register space the declared variable is bound too. If the "space" keyword is omitted, the default space index of 0 is implicitly assigned to the range (so the `tex2` range above resides in`space0`).

An array resource may have an unbounded size, which is declared by specifying the very first dimension to be empty or 0:

```
Texture2D<float4> tex2[]        : register(t0, space0)
Texture2D<float4> tex3[0][5][3]    : register(t5, space1)
```

Aliasing of resource ranges is not allowed. In other words, for each resource type (t, s, u, b), declared register ranges must not overlap. This includes unbounded ranges too. Ranges declared in different register spaces never overlap. Note that unbounded tex2 resides in space0, while unbounded tex3 resides in space1, such that they do not overlap.

Resource Aliasing

The resource ranges specified in the HLSL shaders are logical ranges. They are be bound to concrete heap ranges at runtime via the root signature mechanism. Normally, a logical range maps to a heap range that does not overlap with other heap ranges. However, the root signature mechanism makes it possible to alias (overlap) heap ranges of compatible types. For example, `tex2` and `tex3` ranges from the above example may be mapped to the same (or overlapping) heap range, which has the effect of aliasing textures in the HLSL program. If such aliasing is desired, the shader must be compiled with D3D10_SHADER_RESOURCES_MAY_ALIAS option (*/res_may_alias* option for fxc.exe). The option makes the compiler produce correct code by preventing certain load/store optimizations under the assumption that resources may alias.

Divergence and derivatives

SM5.1 does not impose limitations on the resource index; i.e., `tex2[idx].Sample(…)` – the index idx can be a literal constant, a cbuffer constant, or an interpolated value. While the programming model provides such great flexibility, there are issues to be aware of:

- If index diverges across a quad, the hardware-computed derivative and derived quantities such as LOD may be undefined. The HLSL compiler makes the best effort to issue a warning in this case, but will not prevent a shader from compiling. This behavior is similar to computing derivatives in divergent control flow.
- If the resource index is divergent, the performance is diminished compared to the case of a uniform index, because the hardware needs to perform operations on several resources. For better performance, it is recommended to structure the code in such a way that compilers can figure out (e.g., via SSA-based analysis) what index values are uniform.

UAVs in pixel shaders

SM5.1 does not impose constraints on UAV ranges in pixel shaders as was the case for SM5.0.

Constant Buffers

SM5.1 constant buffers (cbuffer) syntax has changed from SM5.0 to enable developers to index constant buffers. To enable indexable constant buffers, SM5.1 introduces the `ConstantBuffer` "template" construct:

```
struct Foo {
    float4 a;
    int2 b;
};
ConstantBuffer<Foo> myCB1[2][3] : register(b2, space1);
ConstantBuffer<Foo> myCB2 : register(b0, space1);
```

The preceding code declares constant buffer variable `myCB1` of type `Foo` and size 6, and a scalar, constant buffer variable `myCB2`. A constant buffer variable can now be indexed in the shader as:

```
myCB1[i][j].a.xyzw
myCB2.b.yy
```

Fields 'a' and 'b' do not become global variables, but rather must be treated as fields. For backward compatibility, SM5.1 supports the old cbuffer concept for scalar cbuffers. The following statement makes 'a' and 'b' global, read-only variables as in SM5.0. However, such an old-style cbuffer cannot be indexable.

```
cbuffer : register(b1) {
    float4 a;
    int2 b;
};
```

Currently, the shader compiler supports the `ConstantBuffer` template for user-defined structures only.

For compatibility reasons, the HLSL compiler may automatically assign resource registers for ranges declared in `space0`. If 'space' is omitted in the register clause, the default `space0` is used. The compiler uses the first-hole-fits heuristic to assign the registers. The assignment can be retrieved via the reflection API, which has been extended to add the *Space* field for space, while the *BindPoint* field indicates the lower bound of the resource register range.

Bytecode changes in SM5.1

SM5.1 changes how resource registers are declared and referenced in instructions. The syntax involves declaring a register “variable”, similar to how it is done for group shared memory registers:

```
Texture2D<float4> tex0          : register(t5,  space0);
Texture2D<float4> tex1[][5][3]  : register(t10, space0);
Texture2D<float4> tex2[8]       : register(t0,  space1);
SamplerState samp0              : register(s5, space0);

float4 main(float4 coord : COORD) : SV_TARGET {
    float4 r = coord;
    r += tex0.Sample(samp0, r.xy);
    r += tex2[r.x].Sample(samp0, r.xy);
    r += tex1[r.x][r.y][r.z].Sample(samp0, r.xy);
    return r;
}

This will disassemble to:

// Resource Bindings:
//
// Name                                 Type  Format         Dim Space Slot  Elements
// ------------------------------ ---------- ------- ----------- ----- ---- ---------
// samp0                             sampler      NA          NA     0    5         1
// tex0                              texture  float4          2d     0    5         1
// tex1[0][5][3]                     texture  float4          2d     0   10 unbounded
// tex2[8]                           texture  float4          2d     1    0         8
//
//
//
// Input signature:
//
// Name                 Index   Mask Register SysValue  Format   Used
// -------------------- ----- ------ -------- -------- ------- ------
// COORD                    0   xyzw        0     NONE   float   xyzw
//
//
// Output signature:
//
// Name                 Index   Mask Register SysValue  Format   Used
// -------------------- ----- ------ -------- -------- ------- ------
// SV_TARGET                0   xyzw        0   TARGET   float   xyzw
//
ps_5_1
dcl_globalFlags refactoringAllowed
dcl_sampler s0[5:5], mode_default, space=0
dcl_resource_texture2d (float,float,float,float) t0[5:5], space=0
dcl_resource_texture2d (float,float,float,float) t1[10:*], space=0
dcl_resource_texture2d (float,float,float,float) t2[0:7], space=1
dcl_input_ps linear v0.xyzw
dcl_output o0.xyzw
dcl_temps 2
sample r0.xyzw, v0.xyxx, t0[0].xyzw, s0[5]
add r0.xyzw, r0.xyzw, v0.xyzw
ftou r1.x, r0.x
sample r1.xyzw, r0.xyxx, t2[r1.x + 0].xyzw, s0[5]
add r0.xyzw, r0.xyzw, r1.xyzw
ftou r1.xyz, r0.zyxz
imul null, r1.yz, r1.zzyz, l(0, 15, 3, 0)
iadd r1.y, r1.z, r1.y
iadd r1.x, r1.x, r1.y
sample r1.xyzw, r0.xyxx, t1[r1.x + 10].xyzw, s0[5]
add o0.xyzw, r0.xyzw, r1.xyzw
```

```
ret
// Approximately 12 instruction slots are used.
```

Each shader resource range now has an ID (a name) in the shader bytecode. For example, tex1 texture array becomes ‘t1’ in the shader byte code. Giving unique IDs to each resource range allows two things:

- Unambiguously identify which resource range (see dcl_resource_texture2d) is being indexed in an instruction (see sample instruction).
- Attach a set of attributes to the declaration, e.g., element type, stride size, raster operation mode, etc.

Note that the ID of the range is not related to the HLSL lower bound declaration.

The order of reflection resource bindings and shader declaration instructions is the same to aid in identifying the correspondence between HLSL variables and bytecode IDs.

Each declaration instruction in SM5.1 uses a 3D operand to define: range ID, lower and upper bounds. An additional token is emitted to specify the register space. Other tokens may be emitted as well to convey additional properties of the range, e.g., cbuffer or structured buffer declaration instruction emits the size of the cbuffer or structure. The exact details of encoding can be found in d3d12TokenizedProgramFormat.h and **D3D10ShaderBinary::CShaderCodeParser**.

SM5.1 instructions will not emit additional resource operand information as part of the instruction (as in SM5.0). This information is now in the declaration instructions. In SM5.0, instructions indexing resources required resource attributes to be described in extended opcode tokens, since indexing obfuscated the association to the declaration. In SM5.1, each ID (such as ‘t1’) is unambiguously associated with a single declaration that describes the required resource information. Therefore, the extended opcode tokens used on instructions to describe resource information are no longer emitted.

In non-declaration instructions, a resource operand for samplers, SRVs, and UAVs is a 2D operand. The first index is a literal constant that specifies the range ID. The second index represents the linearized value of the index. The value is computed relative to the beginning of the corresponding register space (not relative to the beginning of the logical range) to better correlate with the root signature and to reduce the driver compiler burden of adjusting the index.

A resource operand for CBVs is a 3D operand, containing: literal ID of the range, index of the constant buffer, offset into the particular instance of constant buffer.

Example HLSL Declarations

HLSL programs do not need to know anything about root signatures. They can assign bindings to the virtual “register” binding space, t# for SRVs, u# for UAVs, b# for CBVs, s# for samplers, or rely on the compiler to pick assignments (and query the resulting mappings using shader reflection afterwards). The root signature maps descriptor tables, root descriptors, and root constants to this virtual register space.

The following are some example declarations an HLSL shader might have. Note that there are no references to root signatures or descriptor tables.

```
Texture2D foo[5] : register(t2);
Buffer bar : register(t7);
RWBuffer dataLog : register(u1);

Sampler samp : register(s0);

struct Data {
    UINT index;
    float4 color;
};
ConstantBuffer<Data> myData : register(b0);

Texture2D terrain[] : register(t8); // Unbounded array
Texture2D misc[] : register(t0,space1); // Another unbounded array
                                        // space1 avoids overlap with above t#

struct MoreData {
    float4x4 xform;
};
ConstantBuffer<MoreData> myMoreData : register(b1);

struct Stuff {
    float2 factor;
    UINT drawID;
};
ConstantBuffer<Stuff> myStuff[][3][8] : register(b2, space3)
```

Default Texture Mapping and Standard Swizzle

The use of default texture mapping reduces copying and memory usage while sharing image data between the GPU and the CPU. However, it should only be used in specific situations. The standard swizzle layout avoids copying or swizzling data in multiple layouts.

- Overview
- APIs
- Related topics

Overview

Mapping default textures should not be the first choice for developers. Developers should code in a discrete-GPU friendly way first (that is, do not have any CPU access for the majority of textures and upload with CopyTextureRegion). But, for certain cases, the CPU and GPU may interact so frequently on the same data, that mapping default textures becomes helpful to save power, or to speed up a particular design on particular adapters or architectures. Applications should detect these cases and optimize out the unnecessary copies.

In D3D12, textures created with D3D12_TEXTURE_LAYOUT_UNKNOWN and no CPU access are the most efficient for frequent GPU rendering and sampling. When performance testing, those textures should be compared against D3D12_TEXTURE_LAYOUT_UNKNOWN with CPU access, and D3D12_TEXTURE_LAYOUT_STANDARD_SWIZZLE with CPU access, and D3D12_TEXTURE_LAYOUT_ROW_MAJOR for cross-adapter support.

Using D3D12_TEXTURE_LAYOUT_UNKNOWN with CPU access enables the methods WriteToSubresource, ReadFromSubresource, Map (precluding application access to pointer), and Unmap; but can sacrifice efficiency of GPU access. Using D3D12_TEXTURE_LAYOUT_STANDARD_SWIZZLE with CPU access enables

WriteToSubresource, **ReadFromSubresource**, **Map** (which returns a valid pointer to application), and **Unmap**. It can also sacrifices efficiency of GPU access more than D3D12_TEXTURE_LAYOUT_UNKNOWN with CPU access.

In general, applications should create the majority of textures as GPU-only-accessible, and with D3D12_TEXTURE_LAYOUT_UNKNOWN. Developers should only start entertaining better performance when D3D12_FEATURE_DATA_ARCHITECTURE::UMA is TRUE. Then, the application should pay attention to*CacheCoherentUMA* if deciding which CPU cache properties to choose on the heap.

D3D12 (and D3D11.3) introduce a standard multi-dimensional data layout. This is done to enable multiple processing units to operate on the same data without copying the data or swizzling the data between multiple layouts. A standardized layout enables efficiency gains through network effects and allows algorithms to make short-cuts assuming a particular pattern.

Note though that this standard swizzle is a hardware feature, and may not be supported by all GPUs.

APIs

Unlike D3D11.3, D3D12 supports texture mapping by default, so there is no need to query D3D12_FEATURE_DATA_D3D12_OPTIONS. However D3D12 does not always support standard swizzle - this feature will need to be queried for with a call to CheckFeatureSupport and checking the*StandardSwizzle64KBSupported* field of **D3D12_FEATURE_DATA_D3D12_OPTIONS**.

The following APIs reference texture mapping:

Enums

- D3D12_TEXTURE_LAYOUT : controls the swizzle pattern of default textures and enable map support on default textures.

Structures

- D3D12_RESOURCE_DESC : describes a resource, such as a texture, this is an extensively used structure.
- D3D12_HEAP_DESC : describes a heap.

Methods

- ID3D12Device::CreateCommittedResource : creates a single resource and backing heap of the right size and alignment.
- ID3D12Device::CreateHeap : creates a heap for a buffer or texture.
- ID3D12Device::CreatePlacedResource : creates a resource that is placed in a specific heap, usually a faster method of creating a resource thanCreateHeap.
- ID3D12Device::CreateReservedResource : creates a resource that is reserved but not yet committed or placed in a heap.
- ID3D12CommandQueue::UpdateTileMappings : updates mappings of tile locations in tiled resources to memory locations in a resource heap.
- ID3D12Resource::Map : gets a pointer to the specified data in the resource, and denies the GPU access to the subresource.
- ID3D12Resource::GetDesc : gets the resource properties.
- ID3D12Heap::GetDesc gets the heap properties.
- ReadFromSubresource : copies data from a texture which was mapped using Map.

- WriteToSubresource : copies data into a texture which was mapped using Map.

Resources and parent heaps have alignment requirements: 4MB for multi-sample textures, 64KB for single sample textures and buffers. Linear subresource copying must be aligned to 512 bytes (with row pitch being aligned to 128 bytes). Constant buffer views must be aligned to 256 bytes.

Textures smaller than 64KB should be processed through CreateCommittedResource.

With dynamic textures (textures that change every frame) the CPU will write linearly to the upload heap, followed by a GPU copy operation.

Typically to create dynamic resources (that changes every frame) create a large buffer in an upload heap (refer to Suballocation Within Buffers). To create staging resources, create a large buffer in a readback heap. To create default static resources, create adjacent resources in a default heap. To create default aliased resources, create overlapping resources in a default heap.

WriteToSubresource and ReadFromSubresource rearrange texture data between a row-major layout and an undefined resource layout. The operation is synchronous, so the application should keep CPU scheduling in mind. The application can always break up the copying into smaller regions or schedule this operation in another task. MSAA resources and depth-stencil resources with opaque resource layouts are not supported by these CPU copy operations, and will cause a failure. Formats which don't have a power-of-two element size are also not supported and will also cause a failure. Out of memory return codes can occur.

Typed Unordered Access View (UAV) Loads

Unordered Access View (UAV) Typed Load is the ability for a shader to read from a UAV with a specific DXGI_FORMAT.

Overview

An unordered access view (UAV) is a view of an unordered access resource (which can include buffers, textures, and texture arrays, though without multi-sampling). A UAV allows temporally unordered read/write access from multiple threads. This means that this resource type can be read/written simultaneously by multiple threads without generating memory conflicts. This simultaneous access is handled through the use of Atomic Functions.

D3D12 (and D3D11.3) expands on the list of formats that can be used with typed UAV loads.

Supported formats and API calls

Previously, the following three formats supported typed UAV loads, and were required for D3D11.0 hardware. They are supported for all D3D11.3 and D3D12 hardware.

- R32_FLOAT
- R32_UINT
- R32_SINT

The following formats are supported as a set on D3D12 or D3D11.3 hardware, so if any one is supported, all are supported.

- R32G32B32A32_FLOAT
- R32G32B32A32_UINT
- R32G32B32A32_SINT
- R16G16B16A16_FLOAT
- R16G16B16A16_UINT
- R16G16B16A16_SINT
- R8G8B8A8_UNORM
- R8G8B8A8_UINT
- R8G8B8A8_SINT
- R16_FLOAT
- R16_UINT
- R16_SINT
- R8_UNORM
- R8_UINT
- R8_SINT

The following formats are optionally and individually supported on D3D12 and D3D11.3 hardware, so a single query would need to be made on each format to test for support.

- R16G16B16A16_UNORM
- R16G16B16A16_SNORM
- R32G32_FLOAT
- R32G32_UINT
- R32G32_SINT
- R10G10B10A2_UNORM
- R10G10B10A2_UINT
- R11G11B10_FLOAT
- R8G8B8A8_SNORM
- R16G16_FLOAT
- R16G16_UNORM
- R16G16_UINT
- R16G16_SNORM
- R16G16_SINT
- R8G8_UNORM
- R8G8_UINT
- R8G8_SNORM
- 8G8_SINT
- R16_UNORM
- R16_SNORM
- R8_SNORM
- A8_UNORM
- B5G6R5_UNORM
- B5G5R5A1_UNORM
- B4G4R4A4_UNORM

To determine the support for any additional formats, call CheckFeatureSupport with the D3D12_FEATURE_DATA_D3D12_OPTIONS structure as the first parameter (refer to Capability Querying). The `TypedUAVLoadAdditionalFormats` field will be set if the "supported as a set" list above is supported. Make a second call to **CheckFeatureSupport**, using a D3D12_FEATURE_DATA_FORMAT_SUPPORT structure (checking the returned structure against the D3D12_FORMAT_SUPPORT2_UAV_TYPED_LOAD member of the D3D12_FORMAT_SUPPORT2 enum) to determine support in the list of optionally supported formats listed above, for example:

```
D3D12_FEATURE_DATA_D3D12_OPTIONS FeatureData;
ZeroMemory(&FeatureData, sizeof(FeatureData));
HRESULT hr = pDevice->CheckFeatureSupport(D3D12_FEATURE_D3D12_OPTIONS, &FeatureData, sizeof(FeatureData));
if (SUCCEEDED(hr)) {
    // TypedUAVLoadAdditionalFormats contains a Boolean that
    // tells you whether the feature is supported or not
    if (FeatureData.TypedUAVLoadAdditionalFormats) {
        // Can assume “all-or-nothing” subset is supported (e.g. R32G32B32A32_FLOAT)
        // Cannot assume other formats are supported, so we check:
        D3D12_FEATURE_DATA_FORMAT_SUPPORT FormatSupport = {
            DXGI_FORMAT_R32G32_FLOAT,
            D3D12_FORMAT_SUPPORT1_NONE,
            D3D12_FORMAT_SUPPORT2_NONE
        };
        hr = pDevice->CheckFeatureSupport(D3D12_FEATURE_FORMAT_SUPPORT,
                                          &FormatSupport,
                                          sizeof(FormatSupport));
        if (SUCCEEDED(hr) && (FormatSupport.Support2 & D3D12_FORMAT_SUPPORT2_UAV_TYPED_LOAD) != 0) {
            // DXGI_FORMAT_R32G32_FLOAT supports UAV Typed Load!
        }
    }
}
```

Using typed UAV loads from HLSL

For typed UAVs, the HLSL flag is D3D_SHADER_REQUIRES_TYPED_UAV_LOAD_ADDITIONAL_FORMATS.

Here is example shader code to process a typed UAV load:

```
RWTexture2D<float4> uav1;
uint2 coord;
float4 main() : SV_Target {
  return uav1.Load(coord);
}
```

Volume Tiled Resources

Volume (3D) textures can be used as tiled resources, noting that tile resolution is three-dimensional.

- Overview
- Tiled Resource APIs
- Related topics

Overview

Tiled Resources decouple a D3D Resource object from its backing memory (resources in the past had a 1:1 relationship with their backing memory). This allows for a variety of interesting scenarios such as streaming in texture data and reusing or reducing memory usage

2D texture tiled resources are supported in D3D11.2. D3D12 and D3D11.3 add support for 3D tiled textures.

The typical resource dimensions used in tiling are 4 x 4 tiles for 2D textures, and 4 x 4 x 4 tiles for 3D textures.

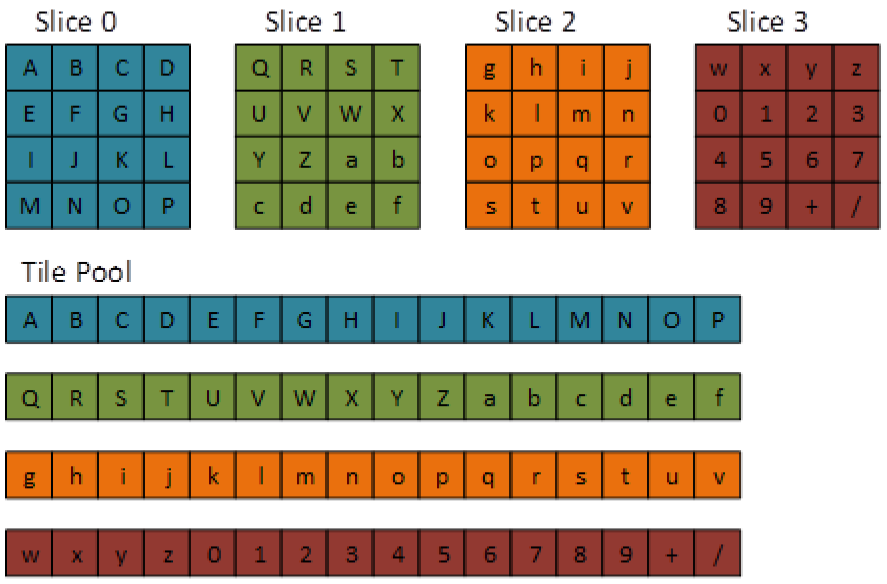
Bits/pixel (1 sample/pixel)	**Tile dimensions (pixels, w x h x d)**
8	64x32x32
16	32x32x32
32	32x32x16
64	32x16x16
128	16x16x16
BC 1,4	128x64x16
BC 2,3,5,6,7	64x64x16

Note the following formats are not supported with tiled resources: 96bpp formats, video formats, R1_UNORM, R8G8_B8G8_UNORM, R8R8_G8B8_UNORM.

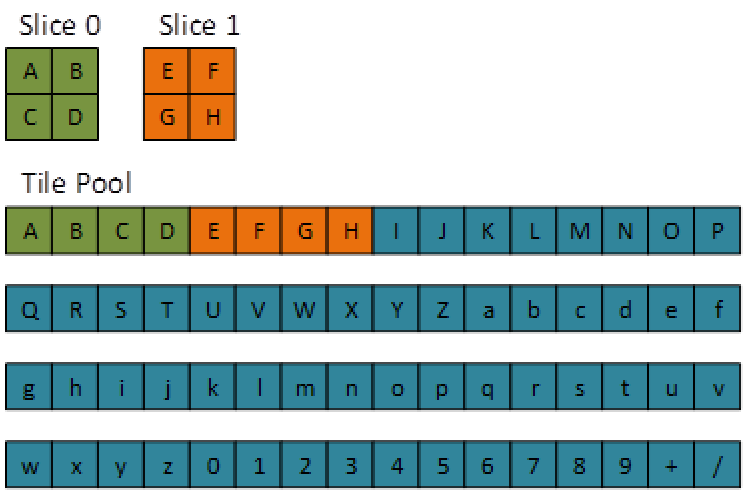
In the diagrams below dark gray represents NULL tiles.

- Texture 3D Tiled Resource default mapping (most detailed mip)
- Texture 3D Tiled Resource default mapping (second most detailed mip)
- Texture 3D Tiled Resource (most detailed mip)
- Texture 3D Tiled Resource (second most detailed mip)
- Texture 3D Tiled Resource (Single Tile)
- Texture 3D Tiled Resource (Uniform Box)

Texture 3D Tiled Resource default mapping (most detailed mip)



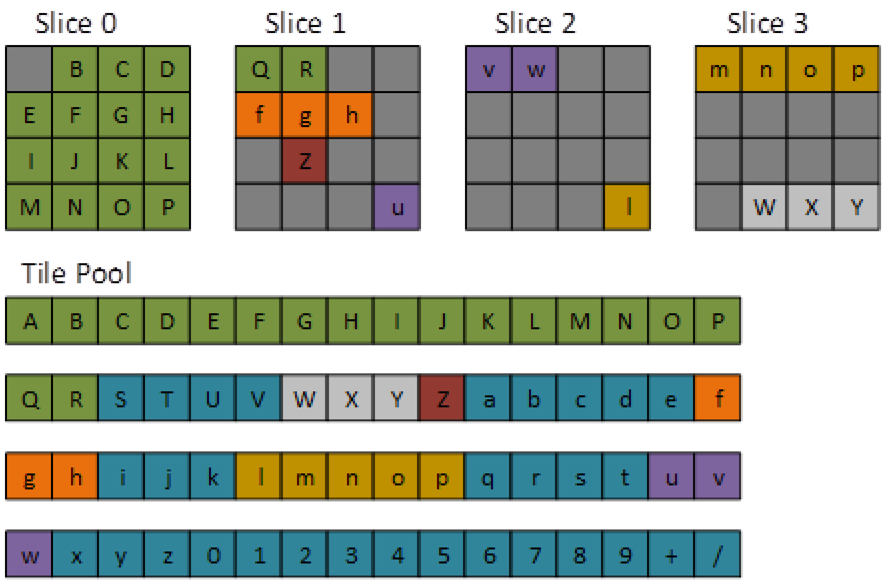
Texture 3D Tiled Resource default mapping (second most detailed mip)


Slice 0
A
B
C
D
Slice 1
E
F
G
H
Tile Pool
A B C D E F G H I J K L M N O P
Q R S T U V W X Y Z a b c d e f
g h i j k l m n o p q r s t u v
w x y z 0 1 2 3 4 5 6 7 8 9 + /

Texture 3D Tiled Resource (most detailed mip)

The following code sets up a 3D tiled resource at the most detailed mip.

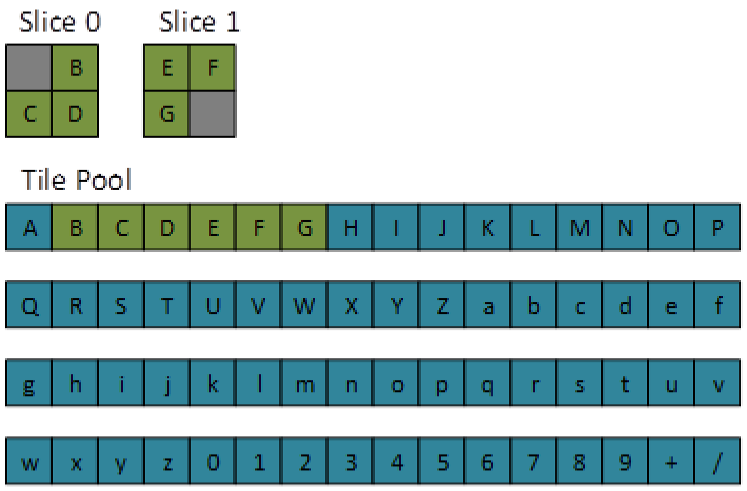
```
D3D12_TILED_RESOURCE_COORDINATE trCoord;
trCoord.X = 1;
trCoord.Y = 0;
trCoord.Z = 0;
trCoord.Subresource = 0;

D3D12_TILE_REGION_SIZE trSize;
trSize.bUseBox = false;
trSize.NumTiles = 63;
```


Slice 0
B C D
E F G H
I J K L
M N O P
Slice 1
Q R
f g h
Z
u
Slice 2
v w
l
Slice 3
m n o p
W X Y
Tile Pool
A B C D E F G H I J K L M N O P
Q R S T U V W X Y Z a b c d e f
g h i j k l m n o p q r s t u v
w x y z 0 1 2 3 4 5 6 7 8 9 + /

Texture 3D Tiled Resource (second most detailed mip)

The following code sets up a 3D tiled resource, and the second most detailed mip:

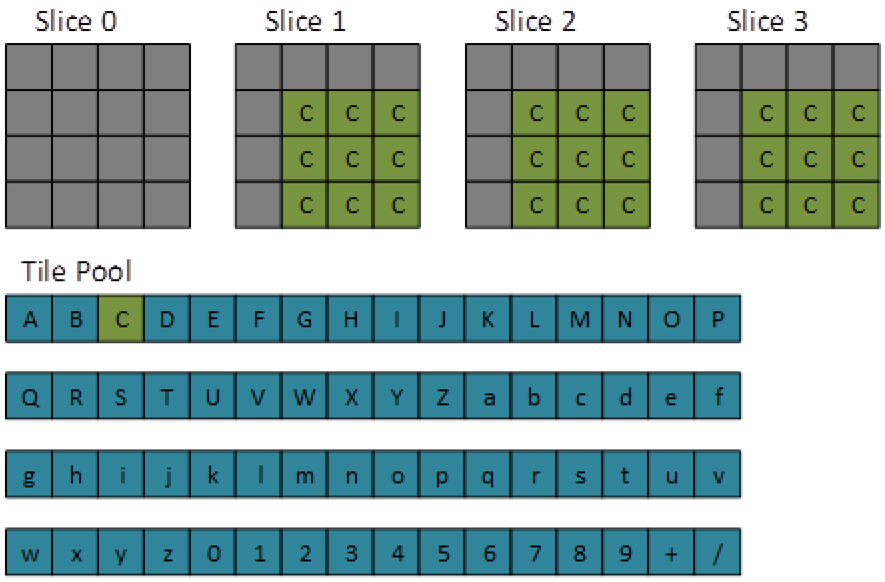
```
D3D12_TILED_RESOURCE_COORDINATE trCoord;
trCoord.X = 1;
trCoord.Y = 0;
trCoord.Z = 0;
trCoord.Subresource = 1;

D3D12_TILE_REGION_SIZE trSize;
trSize.bUseBox = false;
trSize.NumTiles = 6;
```


Slice 0
B
C D
Slice 1
E F
G
Tile Pool
A B C D E F G H I J K L M N O P
Q R S T U V W X Y Z a b c d e f
g h i j k l m n o p q r s t u v
w x y z 0 1 2 3 4 5 6 7 8 9 + /

Texture 3D Tiled Resource (Single Tile)

The following code sets up a Single Tile resource:

```
D3D12_TILED_RESOURCE_COORDINATE trCoord;
trCoord.X = 1;
trCoord.Y = 1;
trCoord.Z = 1;
trCoord.Subresource = 0;

D3D12_TILE_REGION_SIZE trSize;
trSize.bUseBox = true;
trSize.NumTiles = 27;
trSize.Width = 3;
trSize.Height = 3;
trSize.Depth = 3;
```

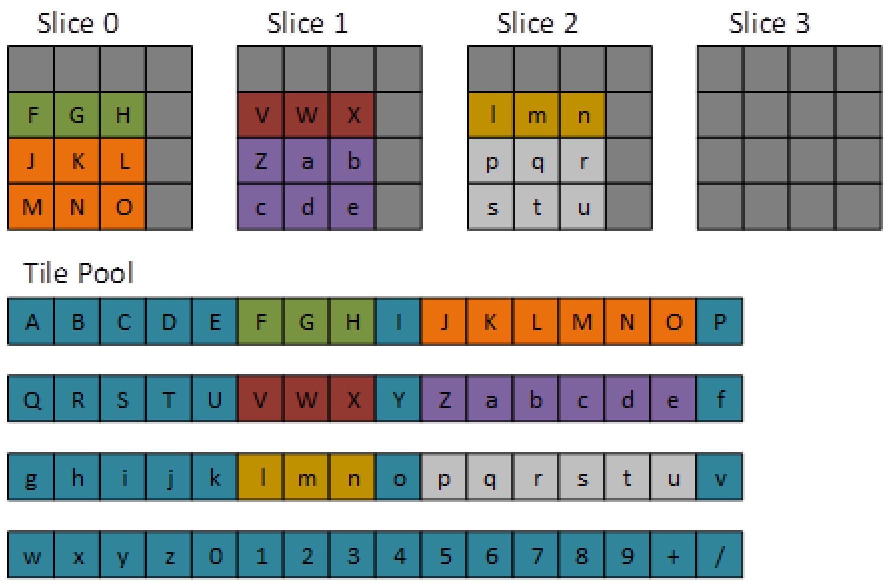

Slice 0
Slice 1
C C C
C C C
C C C
Slice 2
C C C
C C C
C C C
Slice 3
C C C
C C C
C C C
Tile Pool
A B C D E F G H I J K L M N O P
Q R S T U V W X Y Z a b c d e f
g h i j k l m n o p q r s t u v
w x y z 0 1 2 3 4 5 6 7 8 9 + /

Texture 3D Tiled Resource (Uniform Box)

The following code sets up a Uniform Box tiled resource (note the statement `trSize.bUseBox = true;`) :

```
D3D12_TILED_RESOURCE_COORDINATE trCoord;
trCoord.X = 0;
trCoord.Y = 1;
trCoord.Z = 0;
trCoord.Subresource = 0;
```

```
D3D12_TILE_REGION_SIZE trSize;
trSize.bUseBox = true;
trSize.NumTiles = 27;
trSize.Width = 3;
trSize.Height = 3;
trSize.Depth = 3;
```



Slice 0
Slice 1
Slice 2
Slice 3
F G H
J K L
M N O
V W X
Z a b
c d e
l m n
p q r
s t u
Tile Pool
A B C D E F G H I J K L M N O P
Q R S T U V W X Y Z a b c d e f
g h i j k l m n o p q r s t u v
w x y z 0 1 2 3 4 5 6 7 8 9 + /

Tiled Resource APIs

The same API calls are used for both 2D and 3D tiled resources:

Enums

- D3D12_TILED_RESOURCES_TIER : determines the level of tiled resource support.
- D3D12_FORMAT_SUPPORT2 : used to test for tiled resource support.
- D3D12_MULTISAMPLE_QUALITY_LEVEL_FLAGS : determines tiled resource support in a multi-sampling resource.
- D3D12_TILE_COPY_FLAGS : holds flags for copying to and from swizzled tiled resources and linear buffers.

Structs

- D3D12_TILED_RESOURCE_COORDINATE : holds the x, y, and z co-ordinate, and subresource reference.
- D3D12_TILE_REGION_SIZE : specifies the size, and number of tiles, of the tiled region.
- D3D12_TILE_SHAPE : the tile shape as a width, height and depth in texels.
- D3D12_FEATURE_DATA_D3D12_OPTIONS : holds the supported tile resource tier level and a boolean, *VolumeTiledResourcesSupported*, indicated whether volume tiled resources are supported.

Methods

- ID3D12Device::CheckFeatureSupport : used to determine what features, and at what tier, are supported by the current hardware.
- ID3D12GraphcisCommandList::CopyTiles : copies tiles from buffer to tiled resource or vice versa.
- ID3D12CommandQueue::UpdateTileMappings : updates mappings of tile locations in tiled resources to memory locations in a resource heap.
- ID3D12CommandQueue::CopyTileMappings : copies mappings from a source tiled resource to a destination tiled resource.

- ID3D12CommandQueue::GetResourceTiling : gets info about how a tiled resource is broken into tiles.

Subresources

Describes how a resource is divided into subresources, and how to reference a single, multiple or slice of subresources.

Example subresources

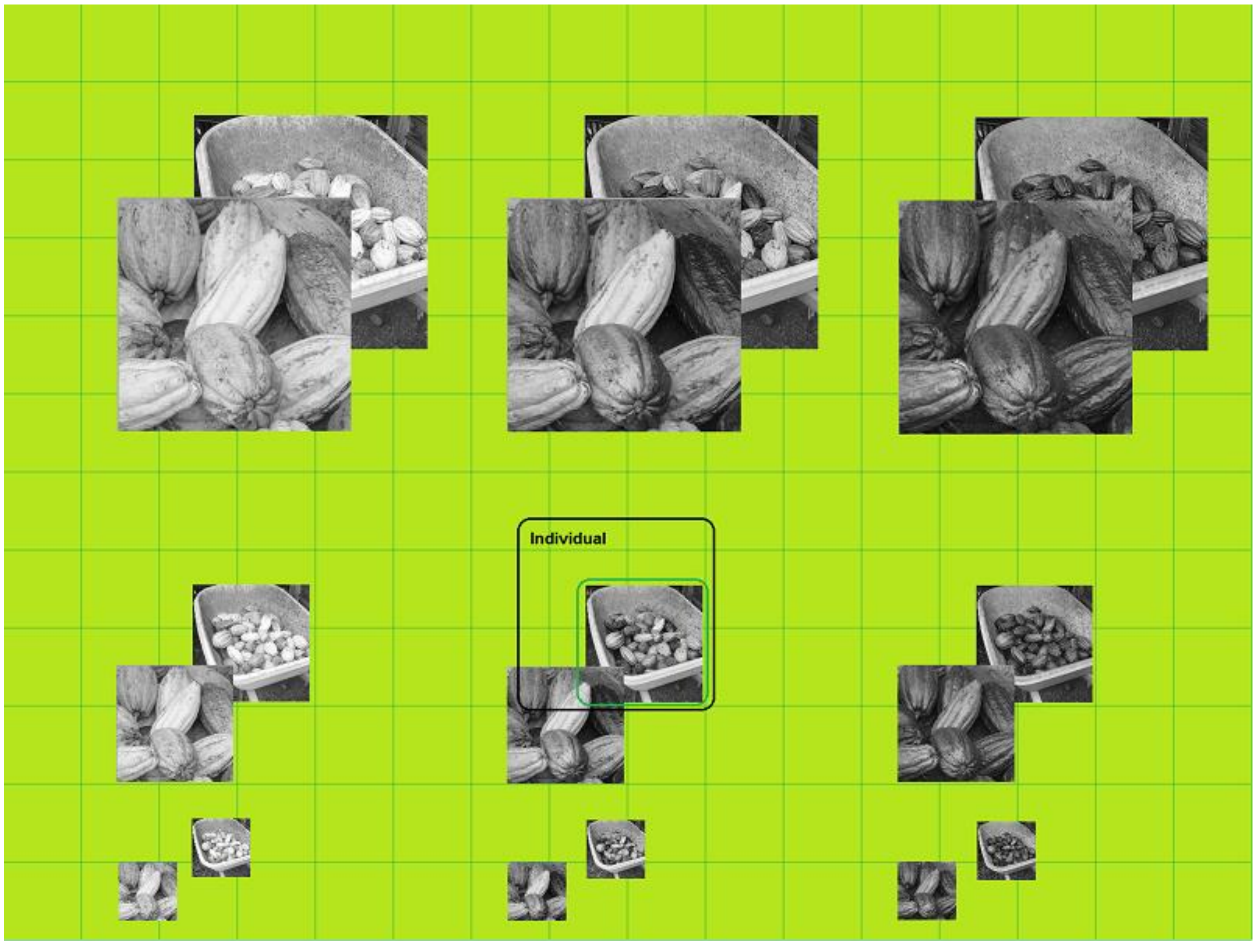
If a resource contains a buffer, then it simply contains one subresource with an index of 0. If the resource contains a texture (or texture array), then referencing the subresources is more complex.

The following examples are based on a ID3D12Resource containing the following texture array of two RGB images. Each texture has a red, green and blue component (no alpha component in these examples), and there are three mipmap levels.

In each example, the subresources outlined in their color component form the selection.

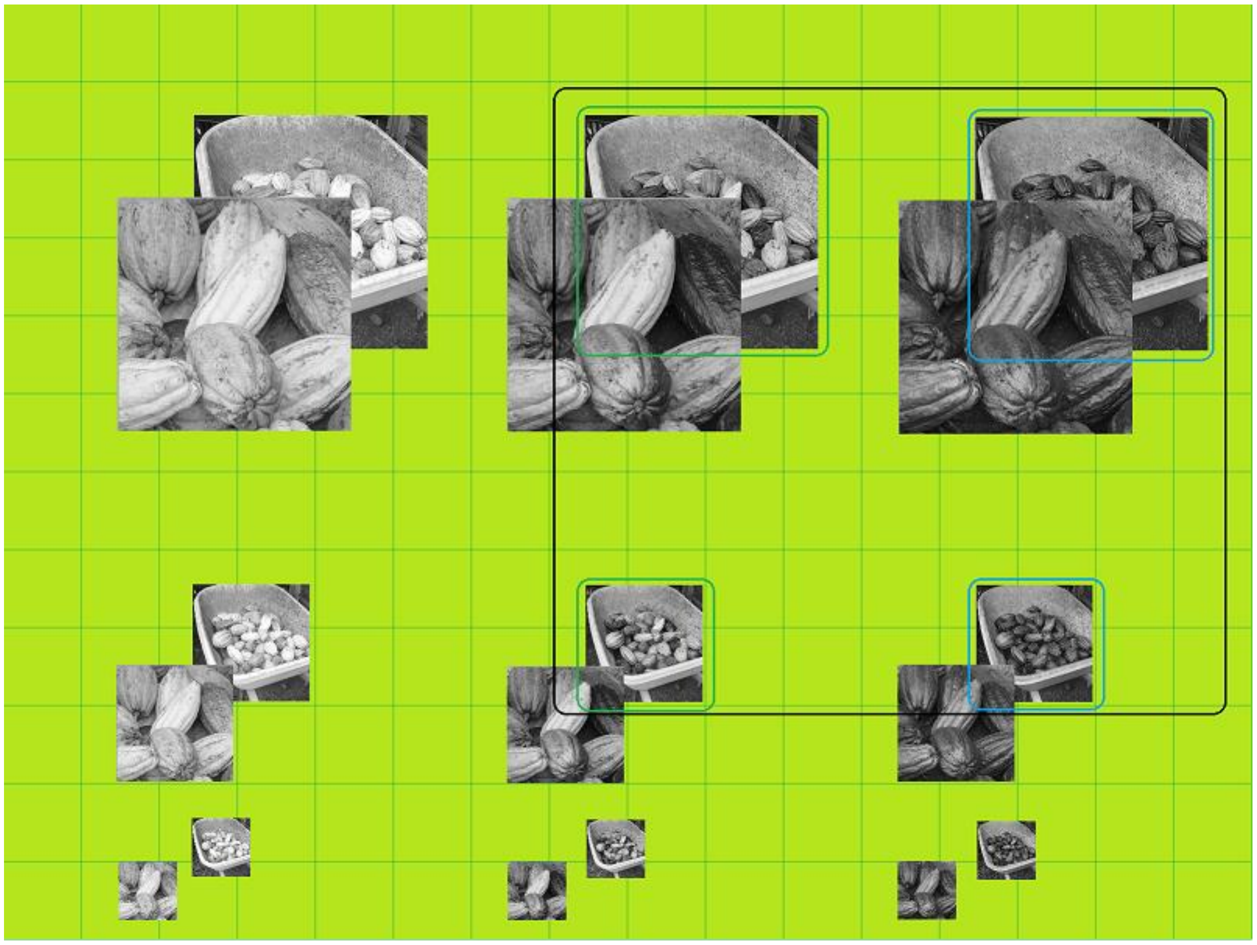
Individual subresource

An individual subresource, located by its index, would locate one of the red, green or blue components at one mip level.



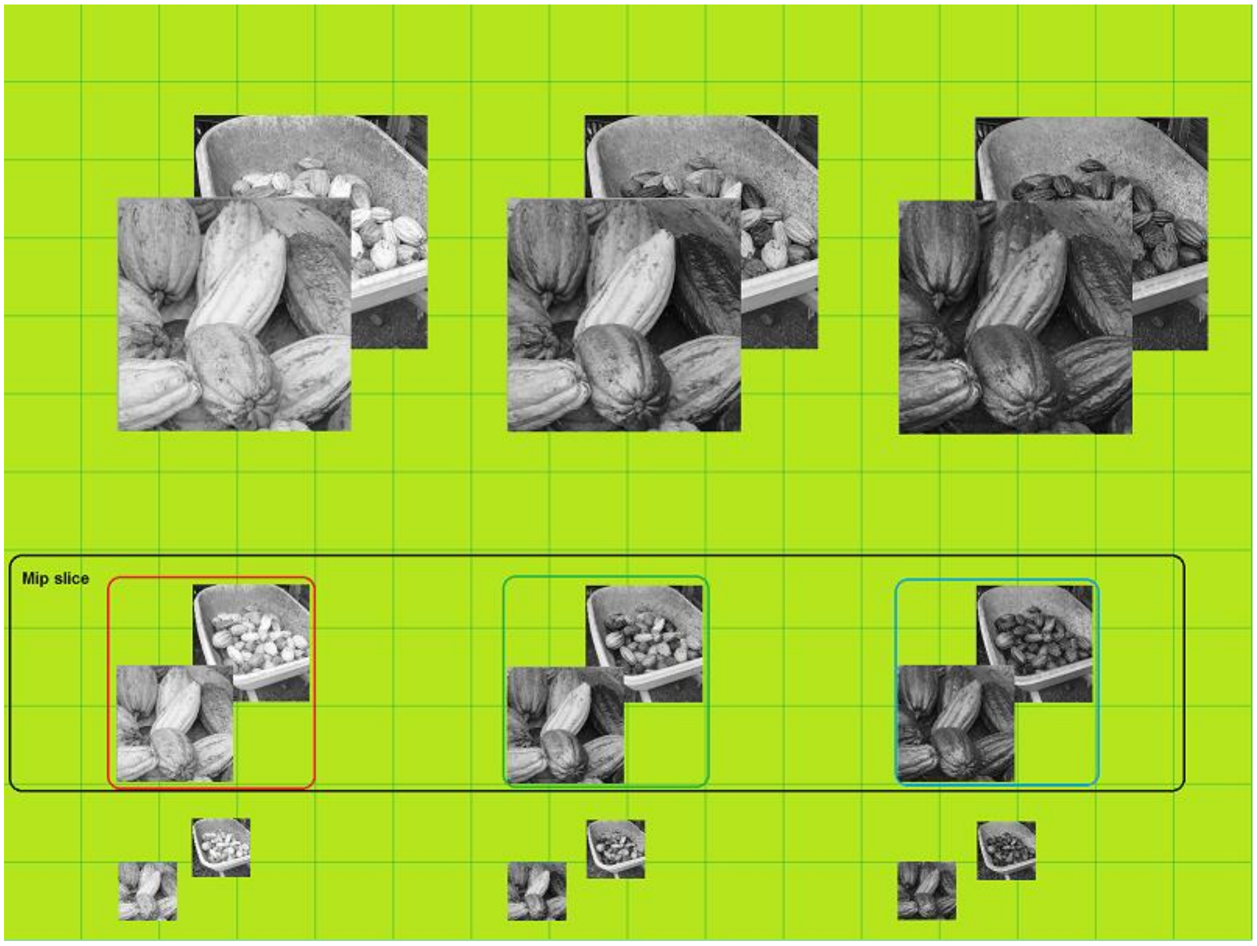
Multiple subresources

A shader-resource view can select any rectangular region of subresources, as shown in the image.



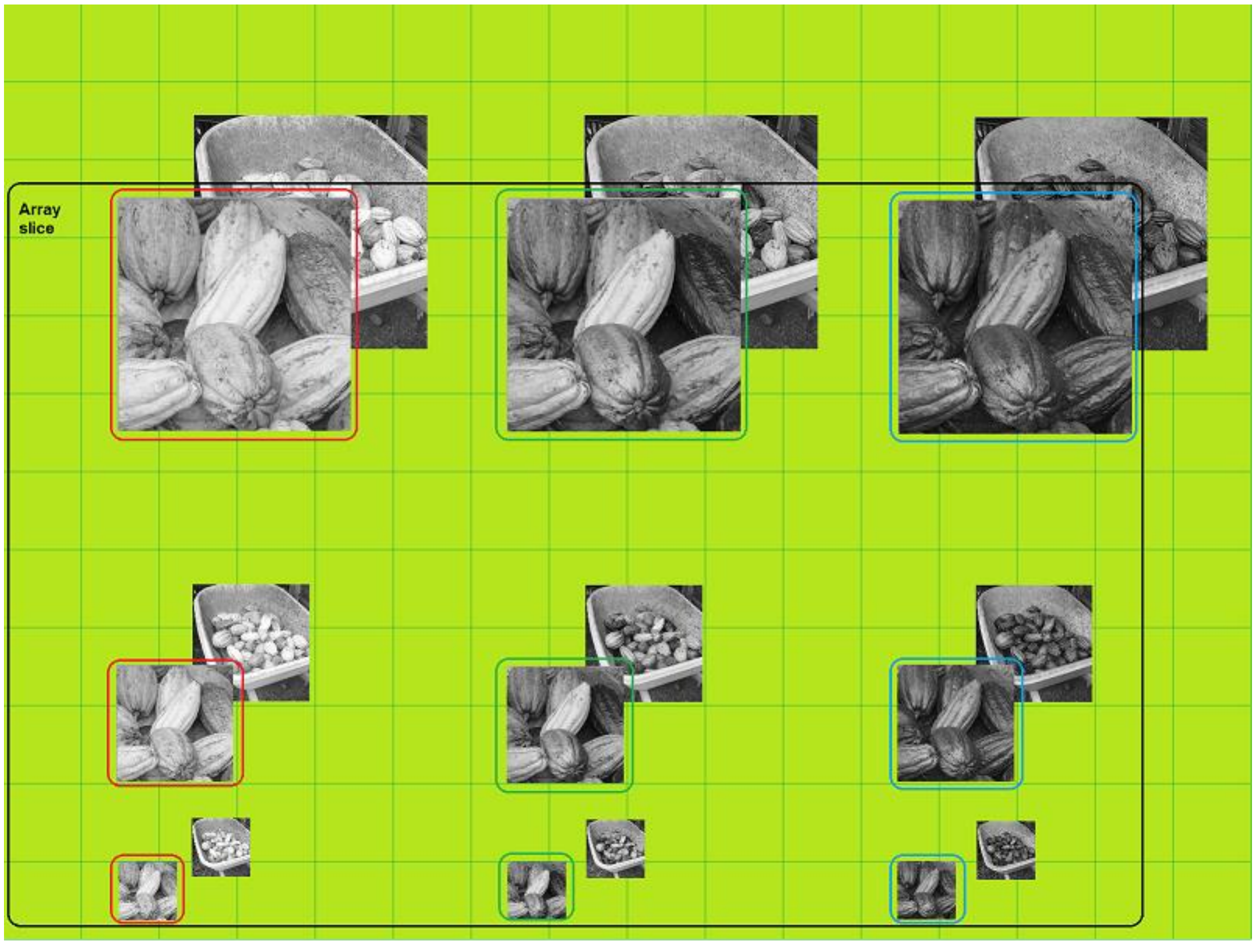
Mip slice

A mip slice includes one mipmap level for every texture in an array, as shown in the following image. A render-target view can only use a single subresource or mip slice and cannot include subresources from more than one mip slice. That is, every texture in a render-target view must be the same size.



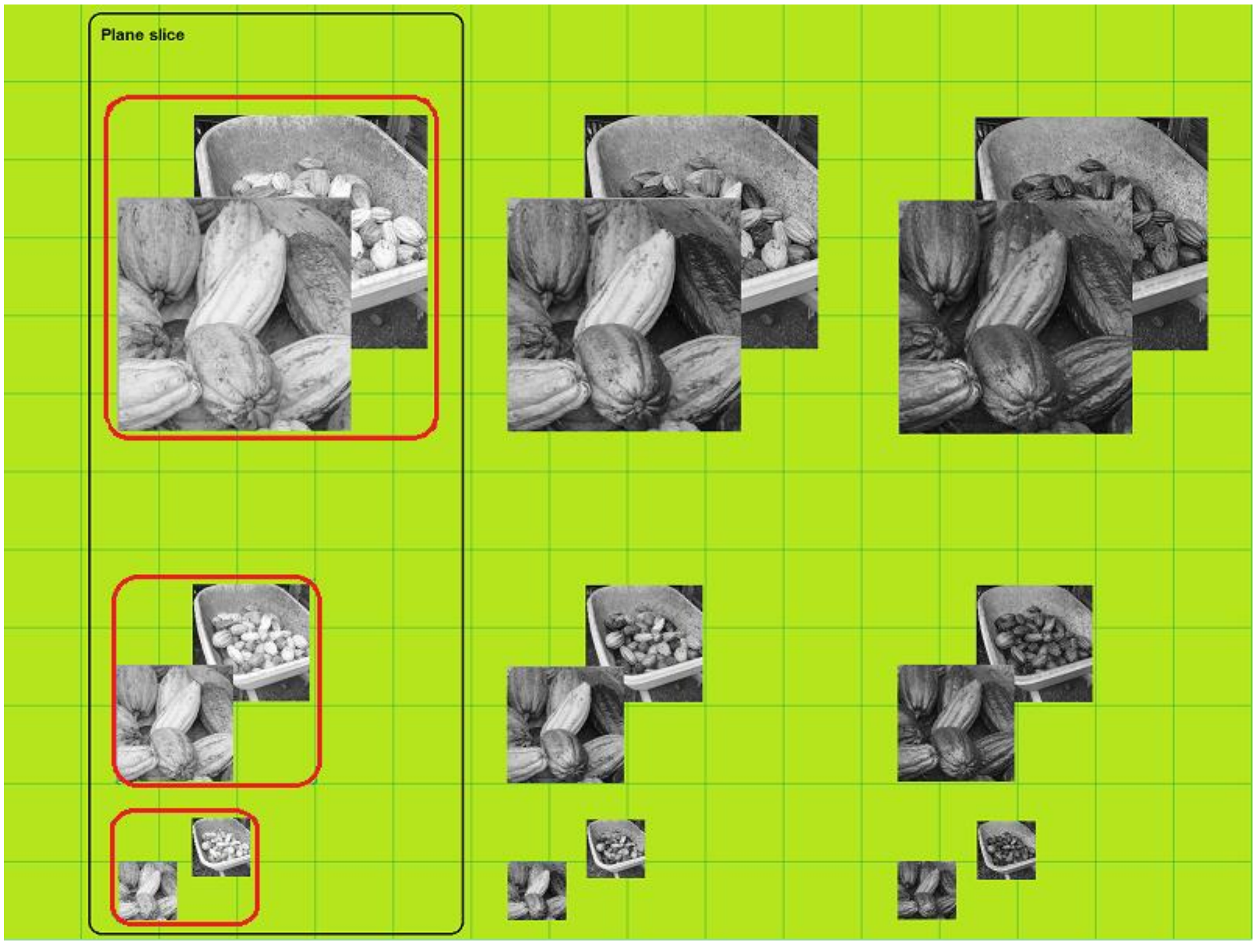
Array slice

Given an array of textures, each texture with mipmaps, an array slice includes one texture and all of its subresources, as shown in the following image.



Plane slice

A plane slice includes all the subresource in one plane (for an RGB based format, each of red, green and blue are one plane).



Subresource APIs

The following APIs reference and work with subresources:

Enumerations:

- D3D12_TEXTURE_COPY_TYPE

The following structures contain *PlaneSlice* indexes, most contain *MipSlice* indexes.

- D3D12_TEX2D_SRV
- D3D12_TEX2D_ARRAY_SRV
- D3D12_TEX2D_UAV
- D3D12_TEX2D_ARRAY_UAV
- D3D12_TEX2D_RTV
- D3D12_TEX2D_ARRAY_RTV

The following structures contain *ArraySlice* indexes, most contain *MipSlice* indexes.

- D3D12_TEX1D_ARRAY_SRV
- D3D12_TEX2D_ARRAY_SRV
- D3D12_TEX2DMS_ARRAY_SRV
- D3D12_TEX1D_ARRAY_UAV
- D3D12_TEX2D_ARRAY_UAV
- D3D12_TEX1D_ARRAY_RTV
- D3D12_TEX2D_ARRAY_RTV

- D3D12_TEX2DMS_ARRAY_RTV
- D3D12_TEX1D_ARRAY_DSV
- D3D12_TEX2D_ARRAY_DSV
- D3D12_TEX2DMS_ARRAY_DSV

The following structures contain *MipSlice* indexes, but neither *ArraySlice* nor *PlaneSlice* indexes.

- D3D12_TEX1D_DSV
- D3D12_TEX2D_DSV
- D3D12_TEX1D_UAV
- D3D12_TEX3D_UAV
- D3D12_TEX1D_RTV
- D3D12_TEX3D_RTV
- D3D12_TEX1D_DSV
- D3D12_TEX2D_DSV

The following structures also reference subresources:

- D3D12_SUBRESOURCE_DATA
- D3D12_SUBRESOURCE_INFO
- D3D12_TILED_RESOURCE_COORDINATE : describes the coordinates of a tiled resource.
- D3D12_SUBRESOURCE_TILING : describes a tiled subresource volume (refer to **Volume Tiled Resources**).
- D3D12_RESOURCE_TRANSITION_BARRIER : describes the transition of subresources between different usages (shader resource, render target, etc.).
- D3D12_SUBRESOURCE_FOOTPRINT
- D3D12_PLACED_SUBRESOURCE_FOOTPRINT
- D3D12_TEXTURE_COPY_LOCATION
- D3D12_DISCARD_REGION

Methods:

- ID3D12Resource::Map : returns a pointer to the specified data in the resource, and denies the GPU access to the subresource.
- ID3D12Resource::Unmap : unmaps the specified range of memory and invalidates the pointer to the resource. Reinstates GPU access to the subresource.
- ID3D12Resource::WriteToSubresource : copies data to a subresource, or a rectangular region of a subresource.
- ID3D12Resource::ReadFromSubresource : copies data from a subresource, or a rectangular region of a subresource.
- ID3D12GraphicsCommandList::ResolveSubresource : copies a multi-sampled subresource into a non-multi-sampled subresource.
- ID3D12Device::GetCopyableFootprints : gets information on a resource, to enable a copy to be made.
- ID3D12Device::GetResourceTiling : gets info about how a tiled resource is broken into tiles.

Memory Management in Direct3D 12

There are two major aspects that applications must manage for D3D12: synchronization and physical memory residency.

- Synchronization Issues
- Physical memory residency
- Related topics

Synchronization Issues

The application must prevent race-conditions between multiple queues, multiple adapters, and the CPU threads. D3D12 no longer synchronizes the CPU and GPU during calls to Map, nor supports convenient mechanisms for resource renaming or multi-buffering (e.g. D3D11_MAP_WRITE_DISCARD). Fences must be used to avoid multiple processing units from over-writing memory before another processing unit finishes using it.

The application must ensure memory is allocated while the GPU may use it. D3D12 no longer ref counts objects when command lists reference them. Fences must be used to ensure the last reference to an object is not released until the GPU finishes referencing an object.

Physical memory residency

The application must ensure data is resident in memory while the GPU reads it. This is simple for applications which never use MakeResident or Evict. Memory used by each object is made resident during the creation of the object. Applications which call these methods must use fences to ensure the GPU doesn't access objects which are evicted.

Resource barriers are another type of synchronization needed (refer to Using Resource Barriers to Synchronize Resource States in Direct3D 12, and to theResourceBarrier method.

In this section

Topic	Description
Suballocation Within Buffers	Buffers have all the features necessary in D3D12 for applications to transfer a large range of transient data from the CPU to the GPU. This section covers four common scenarios for the use and management of resources and buffers.
Suballocation Within Heaps	Resource heaps transfer data from the CPU to the GPU (upload), and from the GPU to the CPU (read back).
Residency	An object is considered to be *resident* when it is accessible by the GPU.

Suballocation Within Buffers

Buffers have all the features necessary in D3D12 for applications to transfer a large range of transient data from the CPU to the GPU. This section covers four common scenarios for the use and management of resources and buffers.

Similar to D3D11, applications in D3D12 still need to declare the usage of memory when allocating buffers in D3D12 compared to Dynamic/Staging Resources in D3D11, but in D3D12, developers have more flexibility and tighter control over memory usage. Buffers, through suballocation, have all the features necessary for low-level memory management.

In this section

Topic	Description
Uploading Different Types of Resources	Shows how to use one buffer to upload both constant buffer data and vertex buffer data to the GPU, and how to properly sub-allocate and place data within buffers. The use of one single buffer increases memory usage flexibility and provides applications with tighter control of memory usage. Also shows the differences between the D3D11 and D3D12 models for uploading different types of resources.
Uploading Texture Data Through Buffers	Uploading 2D or 3D texture data is similar to uploading 1D data, except that applications need to pay closer attention to data alignment related to row pitch. Buffers can be used orthogonally and concurrently from multiple parts of the graphics pipeline, and are very flexible.
Readback Data Through Buffers	Reading back data from the GPU, such as capturing a screen shot, involves the use of the Readback heap.
Fence-Based Resource Management	Shows how to manage resource data life-span by tracking GPU progress via fences. Memory can be effectively re-used with fences carefully managing the availability of free space in memory, such as in a ring buffer implementation for an Upload heap.

Uploading Resources

[Some information relates to pre-released product which may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.]

Shows how to use one buffer to upload both constant buffer data and vertex buffer data to the GPU, and how to properly sub-allocate and place data within buffers. The use of one single buffer increases memory usage flexibility and provides applications with tighter control of memory usage. Also shows the differences between the D3D11 and D3D12 models for uploading different types of resources.

- Upload Different Types of Resources

Upload Different Types of Resources

In D3D12, applications create one buffer to accommodate different types of resource data for uploading, and copy resource data to the same buffer in a similar way for different resource data. Individual views are then created to bind those resource data to the graphics pipeline in the new resource binding model.

In D3D11, applications create separate buffers for different types of resource data (note the different `BindFlags` used in the D3D11 sample code below), explicitly binding each resource buffer to the graphics pipeline, and update the resource data with different methods based on different resource types.

In both D3D12 and D3D11, applications should only use upload resources where the CPU will write the data once and the GPU will read it once. If the GPU will read the data multiple times, the GPU will not read the data linearly, or the rendering is significantly GPU-limited already. The better option may be to use ID3D12GraphicsCommandList::CopyTextureRegion or ID3D12GraphicsCommandList::CopyBufferRegion to copy the upload buffer data to a default resource. A default resource can reside in physical video memory on discrete GPUs.

Code Example: D3D11

```
// D3D11: Separate buffers for each resource type.

void main() {
    // ...

    // Create a constant buffer.
    float constantBufferData[] = ...;

    D3D11_BUFFER_DESC constantBufferDesc = {0};
    constantBufferDesc.ByteWidth = sizeof(constantBufferData);
    constantBufferDesc.Usage = D3D11_USAGE_DYNAMIC;
    constantBufferDesc.BindFlags = D3D11_BIND_CONSTANT_BUFFER;
    constantBufferDesc.CPUAccessFlags = D3D11_CPU_ACCESS_WRITE;

    ComPtr<ID3D11Buffer> constantBuffer;
    d3dDevice->CreateBuffer(
        &constantBufferDesc,
        NULL,
        &constantBuffer
        );

    // Create a vertex buffer.
    float vertexBufferData[] = ...;

    D3D11_BUFFER_DESC vertexBufferDesc = { 0 };
    vertexBufferDesc.ByteWidth = sizeof(vertexBufferData);
    vertexBufferDesc.Usage = D3D11_USAGE_DYNAMIC;
    vertexBufferDesc.BindFlags = D3D11_BIND_VERTEX_BUFFER;

    ComPtr<ID3D11Buffer> vertexBuffer;
    d3dDevice->CreateBuffer(
        &vertexBufferDesc,
        NULL,
        &vertexBuffer
        );

    // ...
}

void DrawFrame() {
    // ...

    // Bind buffers to the graphics pipeline.
    d3dDeviceContext->VSSetConstantBuffers(0, 1, constantBuffer.Get());
    d3dDeviceContext->IASetVertexBuffers(0, 1, vertexBuffer.Get(), ...);

    // Update the constant buffer.
    D3D11_MAPPED_SUBRESOURCE mappedResource;
    d3dDeviceContext->Map(
        constantBuffer.Get(),
        0,
        D3D11_MAP_WRITE_DISCARD,
        0,
        &mappedResource
        );
    memcpy(mappedResource.pData, constantBufferData, sizeof(contatnBufferData));
    d3dDeviceContext->Unmap(constantBuffer.Get(), 0);

    // Update the vertex buffer.
    d3dDeviceContext->UpdateSubresource(
        vertexBuffer.Get(),
        0,
        NULL,
        vertexBufferData,
        sizeof(vertexBufferData),
        0
    );

    // ...
}
```

Code Example: D3D12

```
// D3D12: One buffer to accommodate different types of resources

ComPtr<ID3D12Resource> m_spUploadBuffer;
UINT8* m_pDataBegin = nullptr;    // starting position of upload buffer
UINT8* m_pDataCur = nullptr;      // current position of upload buffer
UINT8* m_pDataEnd = nullptr;      // ending position of upload buffer

void main() {
    // Initialize an upload buffer
    InitializeUploadBuffer(64 * 1024);

    // ...
}

void DrawFrame() {
    // ...

    // Set vertices data to the upload buffer.
    float vertices[] = ...;
    UINT verticesOffset = 0;
    ThrowIfFailed(
        SetDataToUploadBuffer(
            vertices, sizeof(float), sizeof(vertices) / sizeof(float),
            sizeof(float),
            verticesOffset
            ));
    // Set constant data to the upload buffer.
    float constants[] = ...;
    UINT constantsOffset = 0;
    ThrowIfFailed(
        SetDataToUploadBuffer(
            constants, sizeof(float), sizeof(constants) / sizeof(float),
            D3D12_CONSTANT_BUFFER_DATA_PLACEMENT_ALIGNMENT,
            constantsOffset
            ));

    // Create vertex buffer views for the new binding model.
    D3D12_VERTEX_BUFFER_VIEW vertexBufferViewDesc = {
        m_spUploadBuffer->GetGPUVirtualAddress() + verticesOffset,
        sizeof(vertices), // size
        sizeof(float) * 4,  // stride
    };
    commandList->IASetVertexBuffers(
        0,
        1,
        &vertexBufferViewDesc,
        ));

    // Create constant buffer views for the new binding model.
    D3D12_CONSTANT_BUFFER_VIEW_DESC constantBufferViewDesc = {
        m_spUploadBuffer->GetGPUVirtualAddress() + constantsOffset,
        sizeof(constants) // size
         };

    commandList->CreateConstantBufferView(
        &constantBufferViewDesc,
        ...
        ));

    // Continue command list building and execution ...
}

// Create an upload heap and keep it always mapped.
HRESULT InitializeUploadHeap(SIZE_T uSize) {
    HRESULT hr = d3dDevice->CreateCommittedResource(
        &CD3DX12_HEAP_PROPERTIES( D3D12_HEAP_TYPE_UPLOAD ),
        D3D12_HEAP_FLAG_NONE,
        &CD3DX12_RESOURCE_DESC::Buffer( uSize ),
        D3D12_RESOURCE_STATE_GENERIC_READ, nullptr,
        IID_PPV_ARGS( &m_spUploadBuffer ) );

    if (SUCCEEDED(hr)) {
        void* pData;
        m_spUploadBuffer->Map( 0, nullptr, &pData );
        m_pDataCur = m_pDataBegin = reinterpret_cast< UINT8* >( pData );
        m_pDataEnd = m_pDataBegin + uSize;
    }

    return hr;
}

// Sub-allocate from Heap, with offset aligned.
HRESULT SuballocateFromHeap(SIZE_T uSize, UINT uAlign) {
    m_pDataCur = reinterpret_cast< UINT8* >(
        Align(reinterpret_cast< SIZE_T >(m_pDataCur), uAlign)
        );

    return (m_pDataCur + uSize > m_pDataEnd) ? E_INVALIDARG : S_OK;
}

// Place and copy data to the upload heap.
HRESULT SetDataToUploadHeap(
    const void* pData,
    UINT bytesPerData,
    UINT dataCount,
    UINT alignment,
    UINT& byteOffset
    )
{
    SIZE_T byteSize = bytesPerData * dataCount;
    HRESULT hr = SuballocateFromHeap(byteSize, alignment);
    if (SUCCEEDED(hr)) {
        byteOffset = UINT(m_pDataCur - m_pDataBegin);
        memcpy(m_pDataCur, pData, byteSize);
        m_pDataCur += byteSize;
    }
    return hr;
}

// Align uLocation to the next multiple of uAlign.
UINT Align(UINT uLocation, UINT uAlign)
{
    if ( (0 == uAlign) || (uAlign & (uAlign-1)) ) {
        ThrowException("non-pow2 alignment");
    }

    return ( (uLocation + (uAlign-1)) & ~(uAlign-1) );
}
```

Constants

To set constants, vertices and indexes within an Upload or Readback heap, use the following APIs:

- ID3D12Device::CreateConstantBufferView
- ID3D12GraphicsCommandList::IASetVertexBuffers
- ID3D12GraphicsCommandList::IASetIndexBuffer

Resources

Resources are the D3D concept which abstracts the usage of GPU physical memory. Resources require GPU virtual address space to access physical memory. Resource creation is free-threaded.

There are three types of resources with respect to virtual address creation and flexibility in D3D12:

- Committed resources
- Committed resources are the most common idea of D3D resources over the generations. Creating such a resource allocates virtual address range, an implicit heap large enough to fit the whole resource, and commits the virtual address range to the physical memory encapsulated by the heap. The implicit heap properties must be passed to match functional parity with previous D3D versions. Refer to ID3D12Device::CreateCommittedResource.
- Reserved resources
- Reserved resources are equivalent to D3D11 tiled resources. On their creation, only a virtual address range is allocated and not mapped to any heap. The application will map such resources to heaps later. The capabilities of such resources are currently unchanged over D3D11, as they can be mapped to a heap at a 64KB tile granularity with UpdateTileMappings. Refer to ID3D12Device::CreateReservedResource
- Placed resources

- New for D3D12, applications may create heaps separate from resources. Afterward, the application may locate multiple resources within a single heap. This can be done without creating tiled or reserved resources, enabling the capabilities for all resource types able to be created directly by applications. Multiple resources may overlap, and the application must use the `TiledResourceBarrier` to re-use physical memory correctly. Refer to**ID3D12Device::CreatePlacedResource**

Resource size reflection

Applications must use resource size reflection to understand how much room textures with unknown texture layouts require in heaps. Buffers are also supported, but mostly as a convenience.

Applications should be aware of major alignment discrepancies, to help pack resources more densely.

For example, a single-element array with a one-byte-buffer returns a Size of 64KB and an Alignment of 64KB, as buffers currently can only be 64KB aligned.

Also, a three element array with two single-texel 64KB aligned textures and a single-texel 4MB aligned texture reports differing sizes based on the order of the array. If the 4MB aligned textures is in the middle, the resulting Size is 12MB. Otherwise, the resulting Size is 8MB. The Alignment returned would always be 4MB, the super-set of all alignments in the resource array.

Reference the following APIs:

- **D3D12_RESOURCE_ALLOCATION_INFO**
- **GetResourceAllocationInfo**

Buffer alignment

Buffer alignment restrictions have not changed from D3D11, notably:

- 4 MB for multi-sample textures.
- 64 KB for single-sample textures and buffers.

Uploading Texture Data

Uploading 2D or 3D texture data is similar to uploading 1D data, except that applications need to pay closer attention to data alignment related to row pitch. Buffers can be used orthogonally and concurrently from multiple parts of the graphics pipeline, and are very flexible.

- **Upload Texture Data via Buffers**
- **Copying**
- **Mapping and unmapping**
- **Buffer alignment**
- **Related topics**

Upload Texture Data via Buffers

Applications must upload data via **ID3D12GraphicsCommandList::CopyTextureRegion** or **ID3D12GraphicsCommandList::CopyBufferRegion**. Texture data is much more likely to be larger, accessed repeatedly, and benefit from the improved cache-coherency of non-linear memory layouts than other resource data. When buffers are used in D3D12, applications have full control on data placement and arrangement associated with copying resource data around, as long as the memory alignment requirements are satisfied.

The sample highlights where the application simply flattens 2D data into 1D before placing it in the buffer. For the mipmap 2D scenario, the application can either flatten each sub-resource discretely and quickly use a 1D sub-allocation algorithm, or, use a more complicated 2D sub-allocation technique to minimize video memory utilization. The first technique is expected to be used more often as it is simpler. The second technique may be useful when packing data onto a disk or across a network. In either case, the application must still call the copy APIs for each sub-resource.

```
// Prepare a pBitmap in memory, with bitmapWidth, bitmapHeight, and pixel format of
DXGI_FORMAT_B8G8R8A8_UNORM.

// Sub-allocate from the heap for texture data.
D3D12_SUBRESOURCE_FOOTPRINT pitchedDesc = { 0 };
pitchedDesc.Format = DXGI_FORMAT_B8G8R8A8_UNORM;
pitchedDesc.Width = bitmapWidth;
pitchedDesc.Height = bitmapHeight;
pitchedDesc.Depth = 1;
pitchedDesc.RowPitch = Align(bitmapWidth * sizeof(DWORD), D3D12_TEXTURE_DATA_PITCH_ALIGNMENT);

// Note that the helper function UpdateSubresource in D3DX12.h, and ID3D12Device::GetCopyableFootprints
// can help applications fill out these structures.

SuballocateFromHeap(pitchedDesc.Height * pitchedDesc.RowPitch, D3D12_TEXTURE_DATA_PLACEMENT_ALIGNMENT);

D3D12_PLACED_SUBRESOURCE_FOOTPRINT placedTexture2D = { 0 };
placedTexture2D.Offset = m_pDataCur - m_pDataBegin;
placedTexture2D.Footprint = pitchedDesc;

// Copy texture data from pBitmap to the heap.
for (UINT y = 0; y < pitchedDesc.Height; y++) {
    UINT8 *pScan = m_pDataBegin + placedTexture2D. + y * pitchedDesc.RowPitch;
    memcpy(pScan, &(pBitmap->pixels[y * pitchedDesc.Width]), sizeof(DWORD) * pitchedDesc.Width );
}

// Create default texture2D resource.
D3D12_RESOURCE_DESC  textureDesc { ... };

CComPtr<ID3D12Resource> texture2D;
d3dDevice->CreateCommittedResource(
        &CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_DEFAULT),
        D3D12_HEAP_FLAG_NONE, &textureDesc,
        D3D12_RESOURCE_STATE_COPY_DEST,
        nullptr,
        IID_PPV_ARGS(&texture2D) );

// Copy heap data to texture2D.
commandList->CopyTextureRegion(
        &CD3DX12_TEXTURE_COPY_LOCATION( texture2D, 0 ),
        0, 0, 0,
        &CD3DX12_TEXTURE_COPY_LOCATION( m_spUploadHeap, placedTexture2D ),
        nullptr );
```

Note the use of the helper structures CD3DX12_HEAP_PROPERTIES and CD3DX12_TEXTURE_COPY_LOCATION.

Copying

D3D12 methods enable applications to replace D3D11 UpdateSubresource, CopySubresourceRegion, and resource initial data. A single 3D subresource worth of row-major texture data may be located in buffer resources. CopyTextureRegion can copy that texture data from the buffer to a texture resource with an unknown texture layout, and vice versa. Applications should prefer this type of technique to populate frequently accessed GPU resources, by create large buffers in a UPLOAD heap while creating the frequently accessed GPU resource in a DEFAULT heap that has no CPU access. Such a technique efficiently supports discrete GPUs and their large amounts of CPU-inaccessible memory, without commonly impairing UMA architectures.

Note the following two constants:

```
const UINT D3D12_TEXTURE_DATA_PITCH_ALIGNMENT = 256;
const UINT D3D12_TEXTURE_DATA_PLACEMENT_ALIGNMENT = 512;
```

- D3D12_SUBRESOURCE_FOOTPRINT
- D3D12_PLACED_SUBRESOURCE_FOOTPRINT
- D3D12_TEXTURE_COPY_LOCATION
- D3D12_TEXTURE_COPY_TYPE
- ID3D12Device::GetCopyableFootprints
- ID3D12GraphicsCommandList::CopyResource
- ID3D12GraphicsCommandList::CopyTextureRegion
- ID3D12GraphicsCommandList::CopyBufferRegion
- ID3D12GraphicsCommandList::CopyTiles
- ID3D12CommandQueue::UpdateTileMappings

Mapping and unmapping

Map and Unmap can be called by multiple threads safely. The first call to **Map** allocates a CPU virtual address range for the resource. The last call to **Unmap**deallocates the CPU virtual address range. The CPU virtual address is commonly returned to the application.

Whenever data is passed between the CPU and GPU through resources in readback heaps, Map and Unmap must be used to support all systems D3D12 is supported on. Keeping the ranges as tight as possible maximizes efficiency on the systems that require ranges (refer to D3D12_RANGE).

The performance of debugging tools benefit not only from the accurate usage of ranges on all Map / Unmap calls, but also from applications unmapping resources when CPU modifications will no longer be made.

The D3D11 method of using Map (with the DISCARD parameter set) to rename resources is not supported in D3D12. Applications must implement resource renaming themselves. All **Map** calls are implicitly NO_OVERWRITE and multi-threaded. It is the application's responsibility to ensure that any relevant GPU work contained in command lists is finished before the accessing data with the CPU. D3D12 calls to **Map** do not implicitly flush any command buffers, nor do they block waiting for the GPU to finish work. As a result, **Map** and Unmap may even be optimized out in some scenarios.

Buffer alignment

Buffer alignment restrictions:

- Linear subresource copying must be aligned to 512 bytes (with the row pitch aligned to D3D12_TEXTURE_DATA_PITCH_ALIGNMENT bytes).
- Constant data reads must be a multiple of 256 bytes from the beginning of the heap (i.e. only from addresses that are 256-byte aligned).
- Index data reads must be a multiple of the index data type size (i.e. only from addresses that are naturally aligned for the data).
- ID3D12GraphicsCommandList::DrawInstanced and ID3D12GraphicsCommandList::DrawIndexedInstanced data must be from offsets that are multiples of 4 (i.e. only from addresses that are DWORD aligned).

Readback of Data Using Buffers

Reading back data from the GPU, such as capturing a screen shot, involves the use of the Readback heap.

- Read Back Data via Buffers
- Related topics

Read Back Data via Buffers

Reading back data via buffers is similar to uploading data via buffers as shown before, with a few differences:

- Applications need to create a heap with the D3D12_HEAP_TYPE set to D3D12_HEAP_TYPE_READBACK instead of D3D12_HEAP_TYPE_UPLOAD.
- Applications need to use fences to detect when the GPU completes processing a frame and when the writing of data back to the buffer is complete. TheMap method no longer synchronizes with the GPU, as it did in D3D11. All D3D12 **Map** calls now behave as if you called D3D11 Map with the NO_OVERWRITE flag.
- After the data is ready, applications must call Map to see the results.

Fence-Based Resource Synchronization

Shows how to manage resource data life-span by tracking GPU progress via fences. Memory can be effectively re-used with fences carefully managing the availability of free space in memory, such as in a ring buffer implementation for an Upload heap.

- Ring buffer scenario
- Ring buffer sample
- Related topics

Ring buffer scenario

The following is an example in which an app experiences a rare demand for upload heap memory.

A ring buffer is one way to manage an upload heap. The ring buffer holds data required for the next few frames. The app maintains a current data input pointer, and a frame offset queue to record each frame and starting offset of resource data for that frame.

An app creates a ring-buffer based upon a buffer to upload data to the GPU for each frame. Currently frame 2 has been rendered, the ring buffer wraps around the data for frame 4, all the data required for frame 5 is present, and a large constant buffer required for frame 6 needs to be sub-allocated.

Figure 1 : the app tries to sub-allocate for the constant buffer, but finds insufficient free memory.

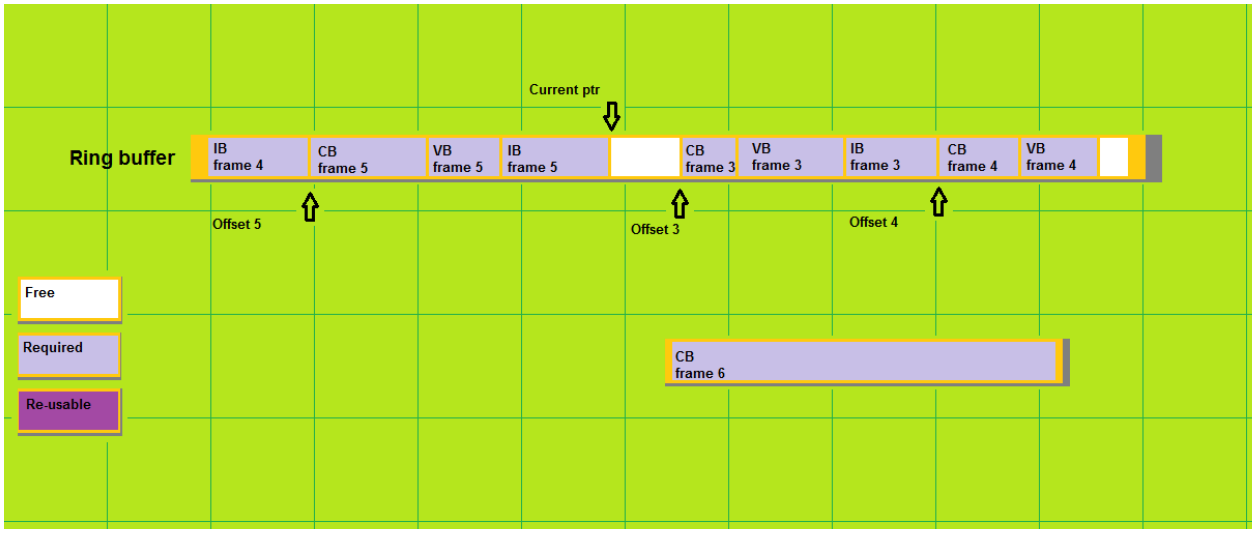


Figure 2 : through fence polling, the app discovers that frame 3 has been rendered, the frame offset queue is then updated, and the current state of the ring buffer follows - however, free memory is still not large enough to accommodate the constant buffer.

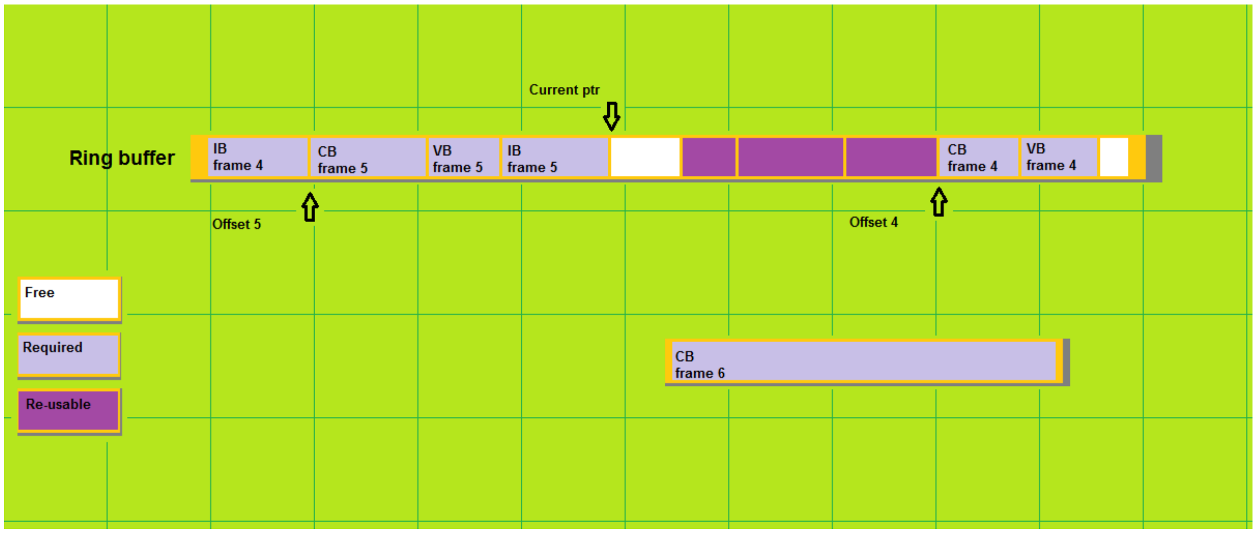


Figure 3 : given the situation, the CPU blocks itself (via fence waiting) until frame 4 has been rendered, which frees up the memory sub-allocated for frame 4.

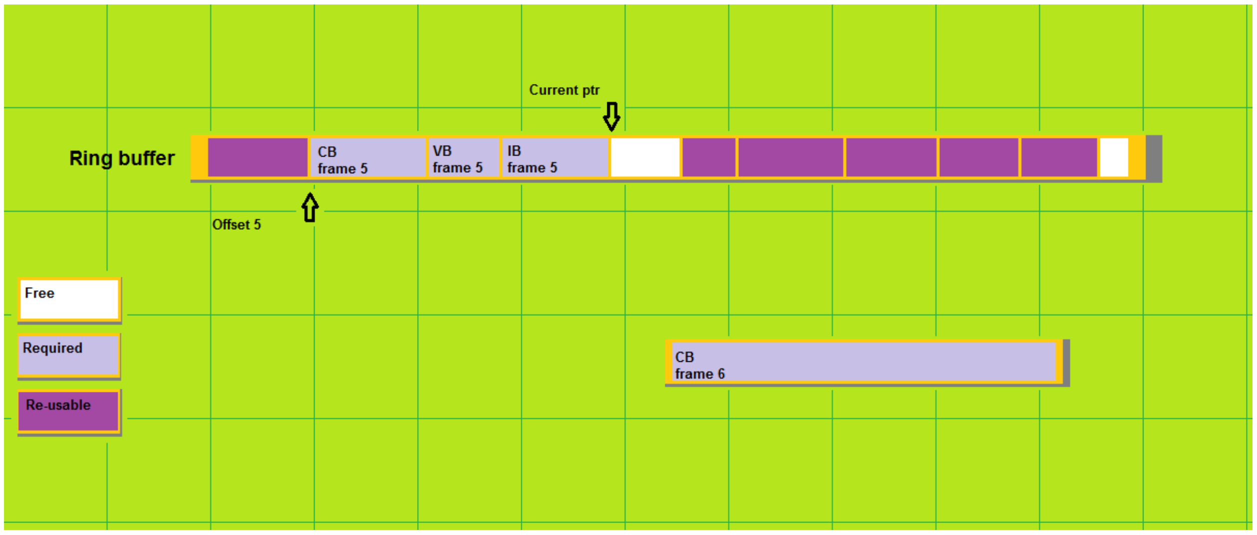
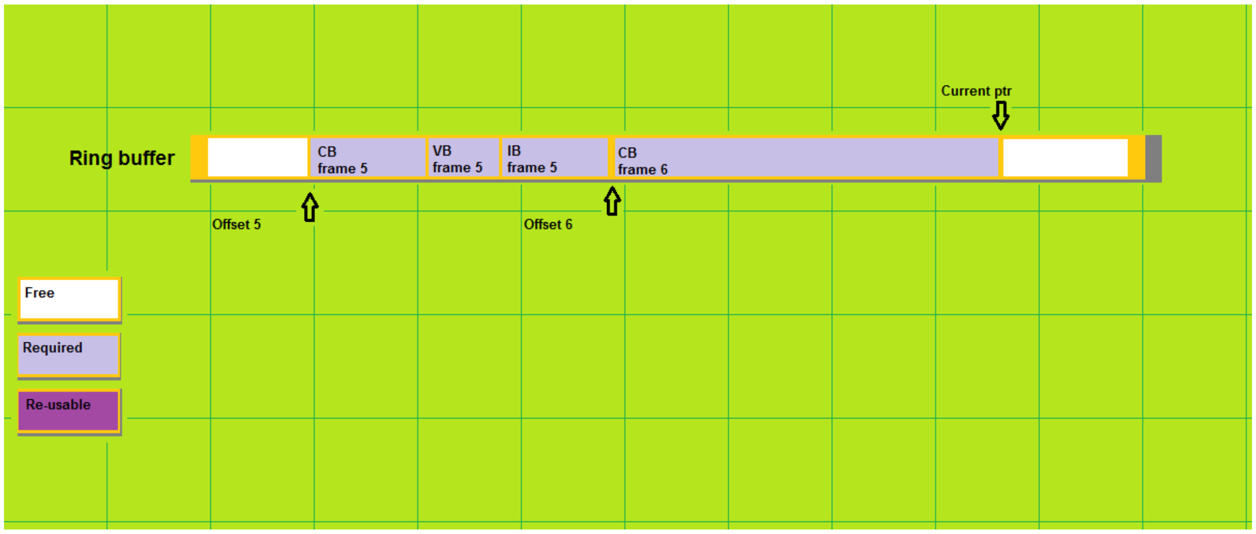


Figure 4 : now free memory is large enough for the constant buffer, and sub-allocation succeeds; the app copies the big constant buffer data to memory previously used by resource data for both frames 3 and 4. The current input pointer is finally updated.



If an app implements a ring buffer, the ring buffer must be large enough to cope with the worse-case scenario of the sizes of resource data.

Ring buffer sample

The following sample code shows how a ring buffer can be managed, paying attention to the sub-allocation routine that handles fence polling and waiting. For simplicity, the sample uses NOT_SUFFICIENT_MEMORY to hide the details of “not sufficient free memory found in the heap” since that logic (based on*m_pDataCur* and offsets inside *FrameOffsetQueue*) is not tightly related to heaps or fences. The sample is simplified to sacrifice frame rate instead of memory utilization.

Note that, ring-buffer support is expected to be a popular scenario; however, the heap design does not preclude other usage, such as command list parameterization and re-use.

```
struct FrameResourceOffset {
    UINT frameIndex;
    UINT8* pResourceOffset;
};
std::queue<FrameResourceOffset> frameOffsetQueue;

void DrawFrame() {
    float vertices[] = ...;
    UINT verticesOffset = 0;
    ThrowIfFailed(
        SetDataToUploadHeap(
            vertices, sizeof(float), sizeof(vertices) / sizeof(float),
            4, // Max alignment requirement for vertex data is 4 bytes.
            verticesOffset
            ));

    float constants[] = ...;
    UINT constantsOffset = 0;
    ThrowIfFailed(
        SetDataToUploadHeap(
            constants, sizeof(float), sizeof(constants) / sizeof(float),
            D3D12_CONSTANT_BUFFER_DATA_PLACEMENT_ALIGNMENT,
            constantsOffset
            ));

    // Create vertex buffer views for the new binding model.
    // Create constant buffer views for the new binding model.
    // ...

    commandQueue->Execute(commandList);
    commandQueue->AdvanceFence();
}

HRESULT SuballocateFromHeap(SIZE_T uSize, UINT uAlign) {
    if (NOT_SUFFICIENT_MEMORY(uSize, uAlign)) {
        // Free up resources for frames processed by GPU; see Figure 2.
        UINT lastCompletedFrame = commandQueue->GetLastCompletedFence();
        FreeUpMemoryUntilFrame( lastCompletedFrame );

        while (NOT_SUFFICIENT_MEMORY(uSize, uAlign) && !frameOffsetQueue.empty()) {
            // Block until a new frame is processed by GPU, then free up more memory; see Figure 3.
            UINT nextGPUFrame = frameOffsetQueue.front().frameIndex;
            commandQueue->SetEventOnFenceCompletion(nextGPUFrame, hEvent);
            WaitForSingleObject(hEvent, INFINITE);
            FreeUpMemoryUntilFrame( nextGPUFrame );
        }
    }

    if (NOT_SUFFICIENT_MEMORY(uSize, uAlign)) {
        // Apps need to create a new Heap that is large enough for this resource.
        return E_HEAPNOTLARGEENOUGH;
    } else {
        // Update current data pointer for the new resource.
        m_pDataCur = reinterpret_cast<UINT8*>(Align(reinterpret_cast<SIZE_T>(m_pHDataCur), uAlign));

        // Update frame offset queue if this is the first resource for a new frame; see Figure 4.
        UINT currentFrame = commandQueue->GetCurrentFence();
        if (frameOffsetQueue.empty() || frameOffsetQueue.back().frameIndex < currentFrame) {
            FrameResourceOffset offset = {currentFrame, m_pDataCur};
            frameOffsetQueue.push(offset);
        }

        return S_OK;
    }
}

void FreeUpMemoryUntilFrame(UINT lastCompletedFrame) {
    while (!frameOffsetQueue.empty() && frameOffsetQueue.first().frameIndex <= lastCompletedFrame) {
        frameOffsetQueue.pop();
    }
}
```

Suballocation Within Heaps

[Some information relates to pre-released product which may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.]

Resource heaps transfer data from the CPU to the GPU (upload), and from the GPU to the CPU (read back).

In this section

Topic	Description
Memory Aliasing and Data Inheritance	Placed and reserved resource may alias physical memory within a heap. Placed resources enable more data inheritance scenarios than reserved resources when the heap has the shared flag set or when the aliased resources have fully defined memory layouts.

Shared Heaps	Sharing is useful for multi-process and multi-adapter architectures.

Memory Aliasing and Data Inheritance

[Some information relates to pre-released product which may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.]

Placed and reserved resource may alias physical memory within a heap. Placed resources enable more data inheritance scenarios than reserved resources when the heap has the shared flag set or when the aliased resources have fully defined memory layouts.

- **Aliasing**
- **Data Inheritance**
- **Related topics**

Aliasing

An aliasing barrier must be issued between the usage of two resources that share the same physical memory, even if data inheritance is not desired. Simple usage models must denote, at least, the destination resource involved in such an operation. See **CreatePlacedResource** for more details and advanced usage models.

After a resource is accessed, any resources which share physical memory with that resource become invalidated, unless data inheritance is allowed to occur. Reads of invalidated resources result in undefined resource contents. Writes to invalidated resources also result in undefined resource contents, unless two conditions occur:

- The resource does not have either the D3D12_RESOURCE_FLAG_ALLOW_RENDER_TARGET or D3D12_RESOURCE_FLAG_ALLOW_DEPTH_STENCIL.
- The write is a copy or clear operation to an entire subresource or tile. Tile-initialization is only available for resources with 64KB_TILE_UNDEFINED_SWIZZLE and 64KB_TILE_STANDARD_SWIZZLE.

Overlapping invalidations are scoped to smaller granularities, when layouts provide information on the location on the location of texel data and when resources are in certain transition barrier states. But, invalidations cannot go any smaller than resource alignment granularities.

A buffer alignment granularity is 64KB, and larger alignment granularity takes precedence. This is important when considering 4KB textures, as multiple 4KB textures can reside in a 64KB region without overlapping each other. But, a buffer aliasing the same 64KB region cannot be used in conjunction with any of those 4KB textures. The application cannot reliably keep the access to the buffer from intersecting the 4KB textures, as GPUs are allowed to swizzle 4KB texture data within the 64KB region in an undefined pattern.

64KB_TILE_UNDEFINED_SWIZZLE, 64KB_TILE_STANDARD_SWIZZLE, and ROW_MAJOR texture layouts inform the application which overlapping alignment granularities have become invalid. For example: An application can create a 2D render target texture array with 2 array slices, a single mip level, and the 64KB_TILE_UNDEFINED_SWIZZLE layout. Assume the application understands each array slice occupies 100 64KB tiles. The application can forgo using array slice 0, and re-use that memory for either a ~6MB buffer, a ~6MB texture with undefined layout, etc. Going further, assume the application no longer required the first tile of array slice 1. Then, the application could also locate a 64KB buffer there until rendering would again require the first tile of array slice 1. The application would have to do a full tile clear or copy in order to re-use the first tile with the texture array again.

However, even textures with defined layouts still have problematic cases. Texture resource sizes can significantly differ from what the application can calculate itself, because some adapter architectures allocate extra memory for textures to reduce the effective bandwidth during common rendering scenarios. Any invalidations into that extra memory region cause the entire resource to become invalidated. See GetResourceAllocationInfo for more details.

Data Inheritance

Placed resources enable the most data inheritance for textures, even with undefined memory layouts. Applications can mimic the data inheritance capabilities that shared committed resources enable by locating two textures with identical resource properties at the same offset in a shared heap. The entire resource description must be identical, including the optimized clear value and type of resource creation method (placed or reserved). But, both resources may have had different initial transition barrier states.

Reserved resources enable per-tile data inheritance; but restrictions commonly exist for resource transition barrier states.

To inherit data, both resources must be in a compatible resource transition barrier state:

- For buffers, simultaneous access textures, and cross-adapter textures, the resource transition state is not important and all states are "compatible".
- For reserved textures without the previous properties or other per-tile data inheritance through 64KB_TILE_UNDEFINED_SWIZZLE or 64KB_TILE_STANDARD_SWIZZLE, the resource transition barrier state including the tile must be in the common state.
- For all other textures, where the resource descriptions match exactly, the resource transition barrier state for each corresponding pair of subresources must:
  - Be in the common state.
  - Be equal when the state has the same GPU-write flag in them.

When the GPU supports standard swizzle, buffers and standard swizzle textures may be aliased to the same memory and inherit data between them. The application can manipulate texels from the buffer representation, because the standard swizzle pattern describes how texels are laid out in memory. The CPU-visible swizzle pattern is equivalent to the GPU-visible swizzle pattern seen in buffers.

Shared Heaps

Sharing is useful for multi-process and multi-adapter architectures.

- Sharing overview
- Sharing heaps across processes
- Sharing heaps across adapters
- Related topics

Sharing overview

Shared heaps enable two things: sharing data in a heap across one or more processes, and precluding a non-deterministic choice of undefined texture layout for resources placed within the heap. Sharing heaps across adapters also removes the need for CPU marshaling of the data.

Both heaps and committed resources can be shared. Sharing a committed resource actually shares the implicit heap along with the committed resource description, such that a compatible resource description can be mapped to the heap from another device.

All methods are free-threaded and inherit the existing D3D11 semantics of the NT handle sharing design.

- ID3D12Device::CreateSharedHandle
- ID3D12Device::OpenSharedHandle
- ID3D12Device::OpenSharedHandleByName

Sharing heaps across processes

Shared heaps are specified with the D3D12_HEAP_FLAG_SHARED member of the D3D12_HEAP_FLAGS enum.

Shared heaps are not supported on CPU-accessible heaps: D3D12_HEAP_TYPE_UPLOAD, D3D12_HEAP_TYPE_READBACK, and D3D12_HEAP_TYPE_CUSTOM without D3D12_CPU_PAGE_PROPERTY_NOT_AVAILABLE.

Precluding a non-deterministic choice of undefined texture layout can significantly impair deferred rendering scenarios on some GPUs, so it is not the default behavior for placed and committed resources. Deferred rendering is impaired on some GPU architectures because deterministic texture layouts decrease the effective memory bandwidth achieved when rendering simultaneously to multiple render target textures of the same format and size. GPU architectures are evolving away from leveraging non-deterministic texture layouts in order to support standardized swizzle patterns and standardized layouts efficiently for deferred rendering.

Shared heaps come with other minor costs as well:

- Shared heap data cannot be recycled as flexibly as in-process heaps due to information disclosure concerns, so physical memory is zero'ed more often.
- There is a minor additional CPU overhead and increased system memory usage during creation and destruction of shared heaps.

Sharing heaps across adapters

Shared heaps across adapters is specified with the D3D12_HEAP_FLAG_SHARED_CROSS_ADAPTER member of the D3D12_HEAP_FLAGS enum.

Cross adapter shared heaps enable multiple adapters to share data without the CPU marshaling the data between them. While varying adapter capabilities determine how efficient adapters can pass data between them, merely enabling GPU copies increases the effective bandwidth achieved. Some texture layouts are allowed on cross adapter heaps to support an interchange of texture data, even if such texture layouts are not otherwise supported. Certain restrictions may apply to such textures, such as only supporting copying.

Cross-adapter sharing works with heaps created by calling ID3D12Device::CreateHeap. Applications can then create resources via CreatePlacedResource. Cross-adapter sharing does not work with CreateCommittedResource and CreateReservedResource.

For cross-adapter sharing, all of the usual cross-queue resource sharing rules still apply. Applications must issue the appropriate barriers to ensure proper synchronization and coherence between the two adapters. Applications should use cross-adapter fences to coordinate the scheduling of command lists submitted to multiple adapters. There is no mechanism to share cross-adapter resources across D3D API versions. Cross-adapter shared resources are only supported in system memory. Cross-adapter shared heaps/resources are supported in D3D12_HEAP_TYPE_DEFAULT heaps and D3D12_HEAP_TYPE_CUSTOM heaps (with the L0 memory pool, and write-combine CPU page properties). Drivers must be sure that GPU read/write operations to cross-adapter shared heaps are coherent with other GPUs on the system. For example, the driver may need to exclude the heap data from residing in GPU caches that typically don't need to be flushed when the CPU cannot directly access the heap data.

Applications should confine the usage of cross adapter heaps to only those scenarios which require the functionality they provide. Cross-adapter heaps are located in D3D12_MEMORY_POOL_L0, which is not always what **GetCustomHeapProperties** suggests. That memory pool is not efficient for discrete/ NUMA adapter architectures. And, the most efficient texture layouts are not always available.

The following limitations also apply:

- The heap flags related to heap tiers must be D3D12_HEAP_FLAG_ALLOW_ALL_BUFFERS_AND_TEXTURES.
- D3D12_HEAP_FLAG_SHARED must also be set.
- Either D3D12_HEAP_TYPE_DEFAULT must be set or D3D12_HEAP_TYPE_CUSTOM with D3D12_MEMORY_POOL_L0 and D3D12_CPU_PAGE_PROPERTY_NOT_AVAILABLE must be set.
- Only resources with D3D12_RESOURCE_FLAG_ALLOW_CROSS_ADAPTER may be placed on cross-adapter heaps.

For more information on using multiple adapters, refer to the **Multi-Adapter** section.

Residency

An object is considered to be *resident* when it is accessible by the GPU.

- **Residency budget**
- **Heap resources**
- **Programming residency management**
- **Related topics**

Residency budget

GPUs do not yet support page-faulting, so applications must commit data into physical memory while the GPU could access it. This process is known as "making something resident", and must be done for both physical system memory and physical discrete video memory. In D3D12, most API objects encapsulate some amount of GPU-accessible memory. That GPU-accessible memory is made resident during the creation of the API object, and evicted on API object destruction.

The amount of physical memory available for the process is known as the video memory budget. The budget can fluctuate noticeably as background processes wake-up and sleep; and fluctuate dramatically when the user switches away to another application. The application can be notified when the budget changes and poll both the current budget and the currently consumed amount of memory. If an application doesn't stay within its budget, the process will be intermittently frozen to allow other applications to run and/or the creation APIs will return failure. The **IDXGIAdapter3** interface provides the methods pertaining to this functionality, in particular **QueryVideoMemoryInfo** and **RegisterVideoMemoryBudgetChangeNotificationEvent**.

Applications are encouraged to use a reservation to denote the amount of memory they cannot go without. Ideally, the user-specified "low" graphics settings, or something even lower, is the right value for such a reservation. Setting a reservation won't ever give an application a higher budget than it would normally receive.

Instead, the reservation information helps the OS kernel quickly minimize the impact of large memory pressure situations. Even the reservation is not guaranteed to be available to the application when the application isn't the foreground application.

Heap resources

While many API objects encapsulate some GPU-accessible memory, heaps & resources are expected to be the most significant way applications consume and manage physical memory. A heap is the lowest level unit to manage physical memory, so it's good to have some familiarity with their residency properties.

- Heaps cannot be made partially resident, but workarounds exists with reserved resources.
- Heaps should be budgeted as part of a particular pool. UMA adapters have one pool, while discrete adapters have two pools. While it is true that kernel can shift some heaps on discrete adapters from video memory to system memory, it does so only as an extreme last resort. Applications should not rely on the over-budget behavior of the kernel, and should focus on good budget management instead.
- Heaps can be evicted from residency, which allows their content to be paged out to disk. But, destruction of heaps is a more reliable technique to free up residency across all adapter architectures. On adapters where *theMaxGPUVirtualAddressBitsPerProcess* field of**D3D12_FEATURE_DATA_GPU_VIRTUAL_ADDRESS_SUPPORT** is near the budget size, **Evict** won't reliably reclaim residency.
- Heap creation can be slow; but it is optimized for background thread processing. It's recommended to create heaps on background threads to avoid glitching the render thread. In D3D12, multiple threads may safely call create routines concurrently.

D3D12 introduces more flexibility and orthogonality into its resource model in order to enable more options for applications. There are three high-level types of resources in D3D12: committed, placed, and reserved.

- Committed resources create both a resource and a heap at the same time. The heap is implicit and cannot be accessed directly. The heap is appropriately sized to locate the entire resource within the heap.
- Placed resources allow the placement of a resource at a non-zero offset within a heap. Offsets must typically be aligned to 64KB; but some exceptions exist in both directions. MSAA resources require 4MB offset alignment, and 4KB offset alignment is available for small textures. Placed resources cannot be relocated or remapped to another heap directly; but they enable simple relocation of the resource data between heaps. After creating a new placed resource in a different heap and copying the resource data, new resource descriptors will have to be used for the new resource data location.
- Reserved resources are only available when the adapter supports tiled resources tier 1 or greater. When available, they offer the most advanced residency management techniques available; but not all adapters currently support them. They enable remapping a resource without requiring regeneration of resource descriptors, partial mip level residency, and sparse texture scenarios, etc. Not all resources types are supported even when reserved resources are available, so a fully general page-based residency manager isn't yet feasible.

Programming residency management

Simple applications may be able to get by merely creating committed resources until experiencing out-of-memory failures. Upon failure, the application can destroy other committed resources or API objects to enable further resource creations to succeed. But, even simple applications are strongly recommended to watch for negative budget changes and destroy unused API objects roughly once a frame.

The complexity of a residency management design will go up when trying to optimize for adapter architectures. Discretely budgeting and managing two pools for discrete will be more complex than managing only one. Overflowing textures into system memory adds more complexity, as the wrong resource in system-memory can severely impact frame rate. And, there is no simple functionality to help identify the resources that would either benefit from higher GPU bandwidth or tolerate lower GPU bandwidth.

Even more complicated designs will query for the features of the current adapter. This information is available in D3D12_FEATURE_DATA_GPU_VIRTUAL_ADDRESS_SUPPORT, D3D12_FEATURE_DATA_ARCHITECTURE, D3D12_TILED_RESOURCES_TIER, and D3D12_RESOURCE_HEAP_TIER.

Multiple parts of an application will likely wind up using different techniques. For example, some large textures and rarely exercised code paths may use committed resources, while many textures may be designated with a streaming property and use a general placed-resource technique.

Multi-Engine and Multi-Adapter Synchronization

Provides an overview and lists APIs relevant to multi-engine (the 3D, compute, and copy engines), and multi-adapter.

In this section

Topic	Description
Synchronization and Multi-Engine	Most modern GPUs contain multiple independent engines that provide specialized functionality. Many have one or more dedicated copy engines, and a compute engine, usually distinct from the 3D engine. Each of these engines can execute commands in parallel with each other. Direct3D 12 provides granular access to the 3D, compute and copy engines, using queues and command lists.
Multi-Adapter	Describes support in D3D12 for multi-engine adapter systems, covering scenarios where applications explicitly target multiple GPU adapters, and scenarios where drivers implicitly use multiple GPU adapters on behalf of an application.

Synchronization and Multi-Engine

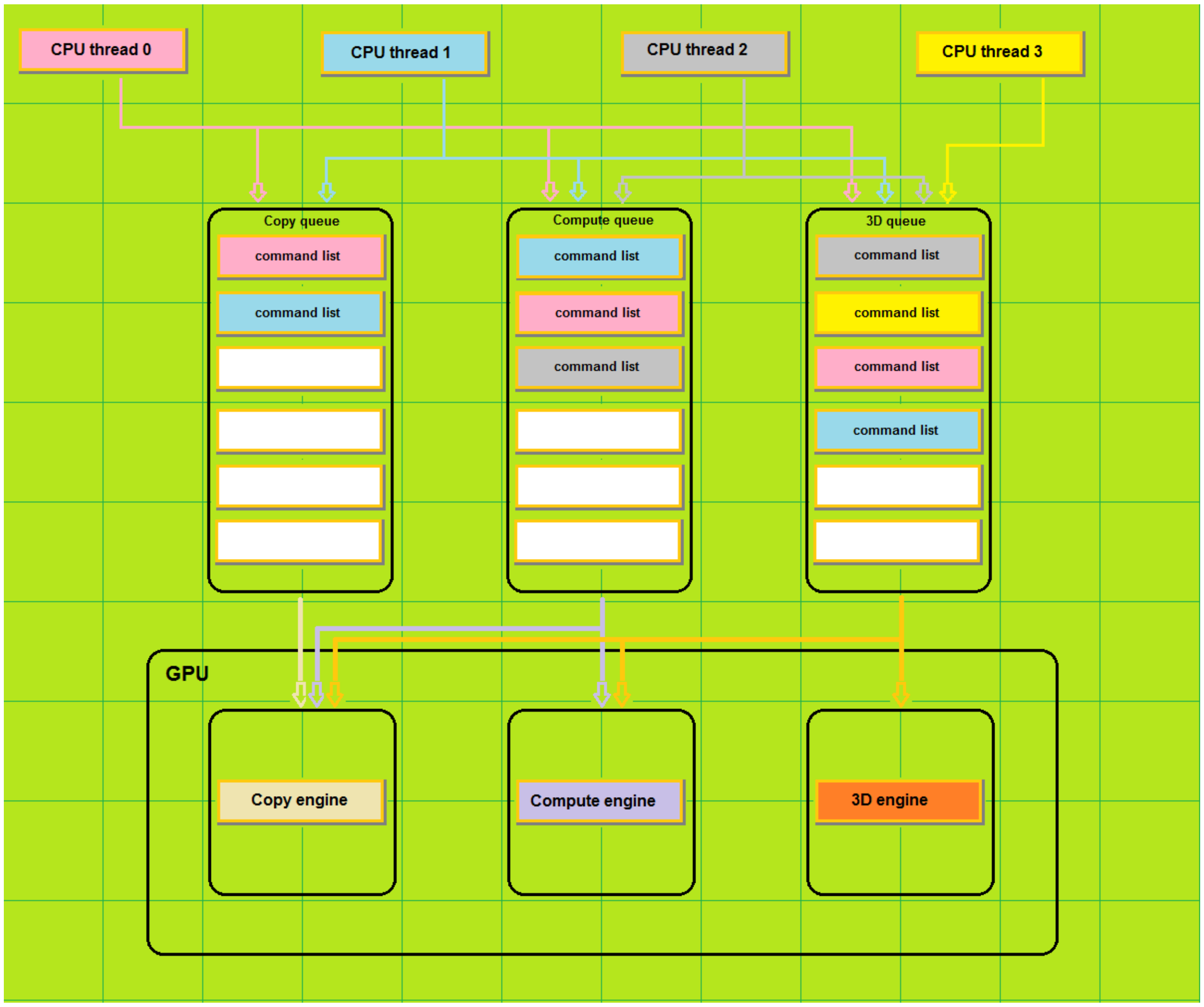
Most modern GPUs contain multiple independent engines that provide specialized functionality. Many have one or more dedicated copy engines, and a compute engine, usually distinct from the 3D engine. Each of these engines can execute commands in parallel with each other. Direct3D 12 provides granular access to the 3D, compute and copy engines, using queues and command lists.

- GPU engines
- Multi-engine scenarios
- Synchronization APIs
  - Devices and Queues
  - Copy and Compute command lists
- Pipelined compute and graphics example
- Asynchronous compute and graphics example

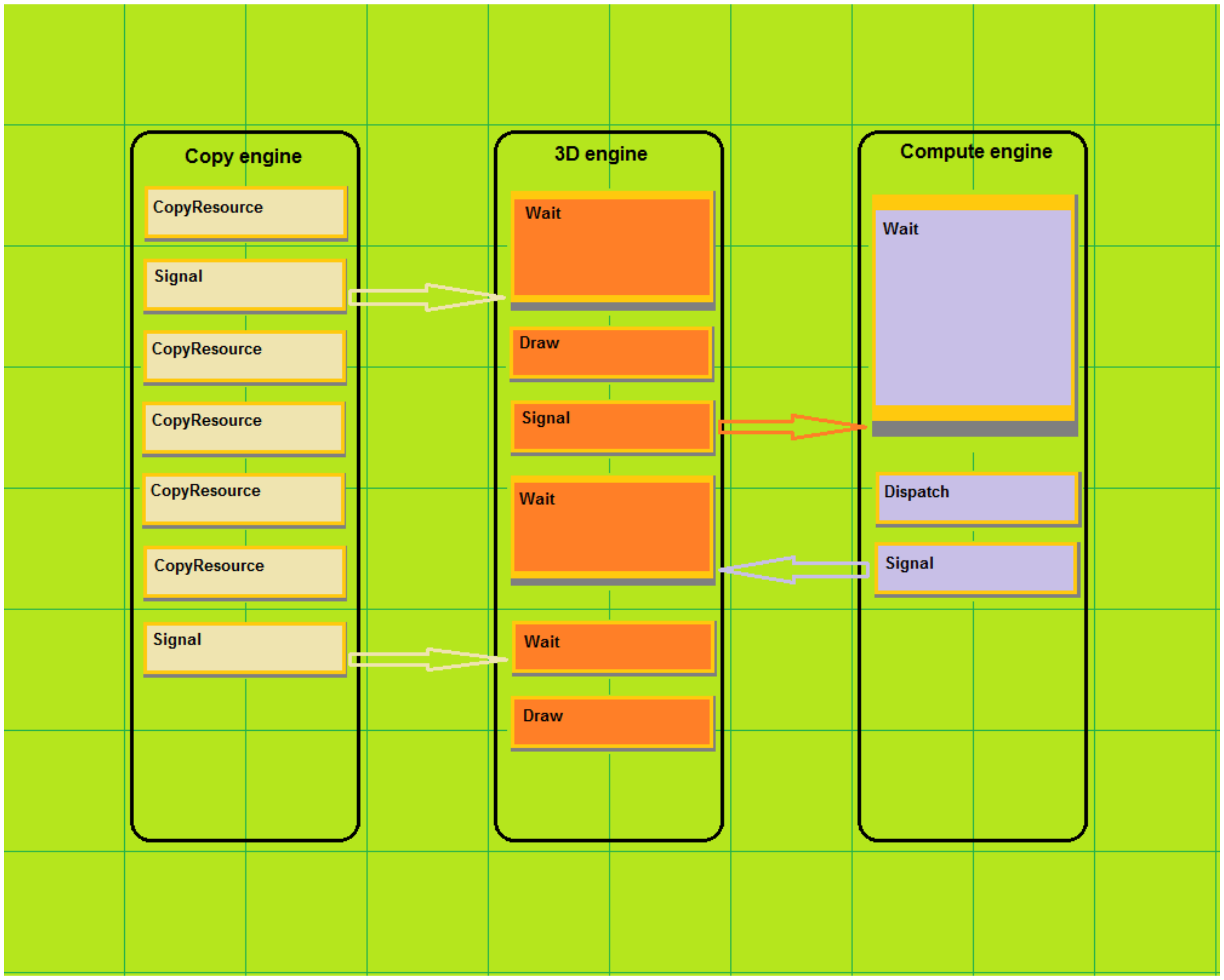
GPU engines

The following diagram shows a title's CPU threads, each populating one or more of the copy, compute and 3D queues. The 3D queue can drive all three GPU engines, the compute queue can drive the compute and copy engines, and the copy queue simply the copy engine.

As the different threads populate the queues, there can be no simple guarantee of the order of execution, hence the need for synchronization mechanisms - when the title requires them.



The following image illustrate how a title might schedule work across multiple GPU engines, including inter-engine synchronization where necessary: it shows the per-engine workloads with inter-engine dependencies. In this example, the copy engine first copies some geometry necessary for rendering. The 3D engine waits for these copies to complete, and renders a pre-pass over the geometry. This is then consumed by the compute engine. The results of the compute engine Dispatch, along with several texture copy operations on the copy engine, are consumed by the 3D engine for the final Draw call.



The following pseudo-code illustrates how a title might submit such a workload.

```
// Get per-engine contexts.  Note that multiple queues may be exposed
// per engine, however that design is not reflected here.
copyEngine = device->GetCopyEngineContext();
renderEngine = device->GetRenderEngineContext();
computeEngine = device->GetComputeEngineContext();
copyEngine->CopyResource(geometry, ...); // copy geometry
copyEngine->Signal(copyFence, 101);
copyEngine->CopyResource(tex1, ...); // copy textures
copyEngine->CopyResource(tex2, ...); // copy more textures
copyEngine->CopyResource(tex3, ...); // copy more textures
copyEngine->CopyResource(tex4, ...); // copy more textures
copyEngine->Signal(copyFence, 102);
renderEngine->Wait(copyFence, 101); // geometry copied
renderEngine->Draw(); // pre-pass using geometry only into rt1
renderEngine->Signal(renderFence, 201);
computeEngine->Wait(renderFence, 201); // prepass completed
computeEngine->Dispatch(); // lighting calculations on pre-pass (using rt1 as SRV)
computeEngine->Signal(computeFence, 301);
renderEngine->Wait(computeFence, 301); // lighting calculated into buf1
renderEngine->Wait(copyFence, 202); // textures copied
renderEngine->Draw(); // final render using buf1 as SRV, and tex[1-4] SRVs
```

The following pseudo-code illustrates synchronization between the copy and 3D engines to accomplish heap-like memory allocation via a ring buffer. Titles have the flexibility to choose the right balance between maximizing parallelism (via a large buffer) and reducing memory consumption and latency (via a small buffer).

```
device->CreateBuffer(&ringCB);
for(int i = 1; i++) {
    if(i > length) copyEngine->Wait(fence1, i - length);
    copyEngine->Map(ringCB, value%length, WRITE, pData); // copy new data
    copyEngine->Signal(fence2, i);
    renderEngine->Wait(fence2, i);
    renderEngine->Draw(); // draw using copied data
    renderEngine->Signal(fence1, i);
}

// example for length = 3:
// copyEngine->Map();
// copyEngine->Signal(fence2, 1); // fence2 = 1
// copyEngine->Map();
// copyEngine->Signal(fence2, 2); // fence2 = 2
// copyEngine->Map();
// copyEngine->Signal(fence2, 3); // fence2 = 3
// copy engine has exhausted the ring buffer, so must wait for render to consume it
// copyEngine->Wait(fence1, 1); // fence1 == 0, wait
// renderEngine->Wait(fence2, 1); // fence2 == 3, pass
// renderEngine->Draw();
// renderEngine->Signal(fence1, 1); // fence1 = 1, copy engine now unblocked
// renderEngine->Wait(fence2, 2); // fence2 == 3, pass
// renderEngine->Draw();
// renderEngine->Signal(fence1, 2); // fence1 = 2
// renderEngine->Wait(fence2, 3); // fence2 == 3, pass
// renderEngine->Draw();
// renderEngine->Signal(fence1, 3); // fence1 = 3
// now render engine is starved, and so must wait for the copy engine
// renderEngine->Wait(fence2, 4); // fence2 == 3, wait
```

Multi-engine scenarios

D3D12 allows developers to avoid accidentally running into inefficiencies caused by unexpected synchronization delays. It also allows developers to introduce synchronization at a higher level where the required synchronization can be determined with greater certainty. A second issue that multi-engine addresses is to make expensive operations more explicit, which includes transitions between 3D and video that were traditionally costly because of synchronization between multiple kernel contexts.

In particular, the following scenarios can be addressed with D3D12:

- Asynchronous and low priority GPU work. This enables concurrent execution of low priority GPU work and atomic operations that enable one GPU thread to consume the results of another unsynchronized thread without blocking.
- High priority compute work. With background compute it is possible to interrupt 3D rendering to do a small amount of high priority compute work. The results of this work can be obtained early for additional processing on the CPU.
- Background compute work. A separate low priority queue for compute workloads allows an application to utilize spare GPU cycles to perform background computation without negative impact on the primary rendering (or other) tasks. Background tasks may include decompression of resources or updating simulations or acceleration structures. Background tasks should be synchronized on the CPU infrequently (approximately once per frame) to avoid stalling or slowing foreground work.
- Streaming and uploading data. A separate copy queue replaces the D3D11 concepts of initial data and updating resources. Although the application is responsible for more details in the D3D12 model, this

responsibility comes with power. The application can control how much system memory is devoted to buffering upload data. The app can choose when and how (CPU vs GPU, blocking vs non-blocking) to synchronize, and can track progress and control the amount of queued work.
- Increased parallelism. Applications can use deeper queues for background workloads (e.g. video decode) when they have separate queues for foreground work.

In D3D12 the concept of a command queue is the API representation of a roughly serial sequence of work submitted by the application. Barriers and other techniques allow this work to be executed in a pipeline or out of order, but the application only sees a single completion timeline. This corresponds to the immediate context in D3D11.

Synchronization APIs

Devices and Queues

The D3D 12 device has methods to create and retrieve command queues of different types and priorities. Most applications should use the default command queues because these allow for shared usage by other components. Applications with additional concurrency requirements can create additional queues. Queues are specified by the command list type that they consume.

Refer to the following creation methods of **ID3D12Device**:

- **CreateCommandQueue** : creates a command queue based on information in a **D3D12_COMMAND_QUEUE_DESC** structure.
- **CreateCommandList** : creates a command list of type **D3D12_COMMAND_LIST_TYPE**.
- **CreateFence** : creates a fence, noting the flags in **D3D12_FENCE_FLAGS**. Fences are used to synchronize queues.

Queues of all types (3D, compute and copy) share the same interface and are all command-list based. Resource mapping operations remain on the queue interface, but are only allowed on 3D and compute queues (not copy).

Refer to the following methods of **ID3D12CommandQueue**:

- **ExecuteCommandLists** : submits an array of command lists for execution. Each command list being defined by **ID3D12CommandList**.
- **Signal** : sets a fence value when the queue (running on the GPU) reaches a certain point.
- **Wait** : the queue waits until the specified fence reaches the specified value.

Note that bundles are not consumed by any queues and therefore this type cannot be used to create a queue.

Fences

The multi-engine API provides explicit APIs to create and synchronize using fences. A fence is a synchronization construct determined by monotonically increasing a UINT64 value. Fence values are set by the application. A signal operation increases the fence value and a wait operation blocks until the fence has reached the requested value. An event can be fired when a fence reaches a certain value.

Refer to the methods of the **ID3D12Fence** interface:

- **GetCompletedValue** : returns the current value of the fence.
- **SetEventOnCompletion** : causes an event to fire when the fence reaches a given value.
- **Signal** : sets the fence to the given value.

Fences allow CPU access to the current fence value, and CPU waits and signals. Independent components can share the default queues but create their own fences and control their own fence values and synchronization.

The **Signal** method on the **ID3D12Fence** interface updates a fence from the CPU side. The **Signal** method on **ID3D12CommandQueue** updates a fence from the GPU side.

All nodes in a multi-engine setup can read and react to any fence reaching the right value.

Applications set their own fence values, a good starting point might be increasing a fence once per frame.

The fence APIs provide powerful synchronization functionality but can create potentially difficult to debug issues.

Copy and Compute command lists

All three types of command list use the **ID3D12GraphicsCommandList** interface, however only a subset of the methods are supported for copy and compute.

Copy and compute command lists can use the following methods:

- Close
- CopyBufferRegion
- CopyResource
- CopyTextureRegion
- CopyTiles
- Reset
- ResourceBarrier

Compute command lists can also use the following methods:

- ClearState
- ClearUnorderedAccessViewFloat
- ClearUnorderedAccessViewUint
- DiscardResource
- Dispatch
- ExecuteIndirect
- SetComputeRoot32BitConstant
- SetComputeRoot32BitConstants
- SetComputeRootConstantBufferView
- SetComputeRootDescriptorTable
- SetComputeRootShaderResourceView
- SetComputeRootSignature
- SetComputeRootUnorderedAccessView
- SetDescriptorHeaps
- SetPipelineState
- SetPredication

Compute command lists must set a compute PSO when calling **SetPipelineState**.

Bundles cannot be used with compute or copy command lists or queues.

Pipelined compute and graphics example

This example shows how fence synchronization can be used to create a pipeline of compute work on a queue (referenced by `pComputeQueue`) that is consumed by graphics work on queue `pGraphicsQueue`. The compute and graphics work is pipelined with the graphics queue consuming the result of compute work from several frames back, and a CPU event is used to throttle the total work queued over all.

```
void PipelinedComputeGraphics() {
    const UINT CpuLatency = 3;
    const UINT ComputeGraphicsLatency = 2;

    HANDLE handle = CreateEvent(nullptr, FALSE, FALSE, nullptr);

    UINT64 FrameNumber = 0;

    while (1) {
        if (FrameNumber > ComputeGraphicsLatency) {
            pComputeQueue->Wait(pGraphicsFence, FrameNumber - ComputeGraphicsLatency);
        }

        if (FrameNumber > CpuLatency) {
            pComputeFence->SetEventOnFenceCompletion(FrameNumber - CpuLatency, handle);
            WaitForSingleObject(handle, INFINITE);
        }

        ++FrameNumber;

        pComputeQueue->ExecuteCommandLists(1, &pComputeCommandList);
        pComputeQueue->Signal(pComputeFence, FrameNumber);
        if (FrameNumber > ComputeGraphicsLatency) {
            UINT GraphicsFrameNumber = FrameNumber - ComputeGraphicsLatency;
            pGraphicsQueue->Wait(pComputeFence, GraphicsFrameNumber);
            pGraphicsQueue->ExecuteCommandLists(1, &pGraphicsCommandList);
            pGraphicsQueue->Signal(pGraphicsFence, GraphicsFrameNumber);
        }
    }
}
```

To support this pipelining there must be a buffer of `ComputeGraphicsLatency+1` different copies of the data passing form the compute queue to the graphics queue. The command lists must use UAVs and indirection to read and write from the appropriate “version” of the data in the buffer. The compute queue must wait until the graphics queue has finished reading from the data for frame N before it can write frame `N+ComputeGraphicsLatency`.

Note that the amount of compute queued worked relative to the CPU does not depend directly on the amount of buffering required, however queuing GPU work beyond the amount of buffer space available is less valuable.

An alternative mechanism to avoid indirection would be to create multiple command lists corresponding to each of the “renamed” versions of the data. The next example uses this technique while extending the previous example to allow the compute and graphics queues to run more asynchronously.

Asynchronous compute and graphics example

This next example allows graphics to render asynchronously from the compute queue. There is still a fixed amount of buffered data between the two stages, however now graphics work proceeds independently and uses the most up-to-date result of the compute stage as known on the CPU when the graphics work is queued. This would be useful if the graphics work was being updated by another source, for example user input. There must be multiple command lists to allow the `ComputeGraphicsLatency` frames of graphics work to be in flight at a time, and the function `UpdateGraphicsCommandList` represents updating the command list to include the most recent input data and read from the compute data from the appropriate buffer.

The compute queue must still wait for the graphics queue to finish with the pipe buffers, but a third fence (`pGraphicsComputeFence`) is introduced so that the progress of graphics reading compute work versus graphics progress in general can be tracked. This reflects the fact that now consecutive graphics frames could read from the same compute result or could skip a compute result. A more efficient but slightly more complicated design would use just the single graphics fence and store a mapping to the compute frames used by each graphics frame.

```
void AsyncPipelinedComputeGraphics() {
    const UINT CpuLatency = 3;
    const UINT ComputeGraphicsLatency = 2;

    // Compute is 0, graphics is 1
    ID3D12Fence *rgpFences[] = { pComputeFence, pGraphicsFence };
    HANDLE handles[2];
    handles[0] = CreateEvent(nullptr, FALSE, TRUE, nullptr);
    handles[1] = CreateEvent(nullptr, FALSE, TRUE, nullptr);
    UINT FrameNumbers[] = { 0, 0 };

    ID3D12GraphicsCommandList *rgpGraphicsCommandLists[CpuLatency];
    CreateGraphicsCommandLists(ARRAYSIZE(rgpGraphicsCommandLists), rgpGraphicsCommandLists);

    // Graphics needs to wait for the first compute frame to complete, this is the
    // only wait that the graphics queue will perform.
    pGraphicsQueue->Wait(pComputeFence, 1);

    while (1) {
        for (auto i = 0; i < 2; ++i) {
            if (FrameNumbers[i] > CpuLatency) {
                rgpFences[i]->SetEventOnFenceCompletion(FrameNumbers[i] - CpuLatency, handles[i]);
            } else {
                SetEvent(handles[i]);
            }
        }

        auto WaitResult = WaitForMultipleObjects(2, handles, FALSE, INFINITE);
        auto Stage = WaitResult = WAIT_OBJECT_0;
        ++FrameNumbers[Stage];

        switch (Stage) {
        case 0: {
            if (FrameNumbers[Stage] > ComputeGraphicsLatency) {
                pComputeQueue->Wait(pGraphicsComputeFence, FrameNumbers[Stage] - ComputeGraphicsLatency);
            }
            pComputeQueue->ExecuteCommandLists(1, &pComputeCommandList);
            pComputeQueue->Signal(pComputeFence, FrameNumbers[Stage]);
            break;
        }
        case 1: {
            // Recall that the GPU queue started with a wait for pComputeFence, 1
            UINT64 CompletedComputeFrames = min(1, pComputeFence->GetCurrentFenceValue());
            UINT64 PipeBufferIndex = (CompletedComputeFrames - 1) % ComputeGraphicsLatency;
            UINT64 CommandListIndex = (FrameNumbers[Stage] - 1) % CpuLatency;

            // Update graphics command list based on CPU input and using the appropriate
            // buffer index for data produced by compute.
            UpdateGraphicsCommandList(PipeBufferIndex, rgpGraphicsCommandLists[CommandListIndex]);

            // Signal *before* new rendering to indicate what compute work
            // the graphics queue is DONE with
            pGraphicsQueue->Signal(pGraphicsComputeFence, CompletedComputeFrames - 1);
            pGraphicsQueue->ExecuteCommandLists(1, rgpGraphicsCommandLists + PipeBufferIndex);
```

```
                pGraphicsQueue->Signal(pGraphicsFence, FrameNumbers[Stage]);
                break;
            }
            }
        }
    }
```

Multi-queue resource access

To access a resource on more than one queue an application must adhere to the following rules.

- Resource access (refer to D3D12_RESOURCE_STATES) is determined by queue type class not queue object. There are two types of queue: Compute/3D queue is one type, Copy is a second type. So a resource that has a barrier to the NON_PIXEL_SHADER_RESOURCE state on one 3D queue can be used in that state on any 3D or Compute queue, subject to synchronization requirements which require (most) writes to be serialized. The resource states that are shared between the two type classes (COPY_SOURCE and COPY_DEST) are considered different states for each type class. So that if a resource transitions to COPY_DEST on a Copy queue it is not accessible as a copy destination from 3D or Compute queues and vice versa.
- The D3D12_RESOURCE_STATE_COMMON state is used for read-only access by all queue type classes and as an intermediate state for transferring write access between queue types classes. Any resource in the COMMON state can be accessed as through it were in a single state with one WRITE flag, or one or more READ flags - set from the resource's promotable flags. The promotable flags of the resource are all of the flags that apply to the resource and are marked as promotable in the following table:

State flag	Initializable	Promotable
VERTEX_AND_CONSTANT_BUFFER	Yes	No
INDEX_BUFFER	Yes	No
RENDER_TARGET	No	No
UNORDERED_ACCESS	Yes	No
DEPTH_WRITE	No	No
DEPTH_READ	No	No
NON_PIXEL_SHADER_RESOURCE	Yes	Yes
PIXEL_SHADER_RESOURCE	Yes	Yes
STREAM_OUT	No	No
INDIRECT_ARGUMENT	No	No
COPY_DEST	Yes	Yes
COPY_SOURCE	Yes	Yes
RESOLVE_DEST	No	No
RESOLVE_SOURCE	No	No

-

- When this access occurs the promotion acts like an implicit resource barrier. Subsequent to this access further resource barriers will be required to change the resource state. The promotion represents the fact that resources in the COMMON state should not require additional GPU work or driver tracking to support certain accesses.

- The COPY flags (COPY_DEST and COPY_SOURCE) used as initial states represent states in the 3D/Compute type class. To use a resource initially on a Copy queue it should start in the COMMON state. The COMMON state can be used for all usages on a Copy queue using the implicit state transitions.

- The flip side of common state promotion is decay. Resources that meet certain requirements are considered to be stateless and automatically return to the common state under certain conditions. To support the use of resource transition barriers to allow hazard tracking the resource state only decays when a GPU signal operation is queued. The following resources will decay when a GPU signal operation is queued:
  - Resources being accessed on a Copy queue.
  - Resources that have the D3D12_RESOURCE_FLAG_ALLOW_SIMULTANEOUS_ACCESS flag set.
- Although resource state is shared across all Compute and 3D queues, it is not generally permitted to write to the resource simultaneously on different queues. (Simultaneously here means unsynchronized, although simultaneous execution is not possible on some hardware.) There are two exceptions.
  - UAV accesses; when a resource is in the UNORDERED_ACCESS state it can be accessed for read/write by multiple queues simultaneously, including cross-device and cross-process access. UAV barriers (D3D12_RESOURCE_BARRIER_TYPE_UAV) apply to cross-queue access.
  - Resources that have the D3D12_RESOURCE_FLAG_ALLOW_SIMULTANEOUS_ACCESS flag set may be accessed for read/write simultaneously as long as the operations don't touch the same pixels. Conceptually these simultaneous accesses split the resource into independent areas and each queue can independently apply resource barriers to the section of the resource that it is operating upon. See the **Example of split barriers** below.
- Back buffers being presented must be in the D3D12_RESOURCE_STATE_COMMON state.
- There are two depth flags for read-only and read-write access (D3D12_RESOURCE_STATE_DEPTH_READ and D3D12_RESOURCE_STATE_DEPTH_WRITE respectively.) This serves to make the read or write distinction explicit.
- A resource in the most generic read state is readable from any number of queues simultaneously, including cross device and cross process access. The most generic read state for a resource is the subset of D3D12_RESOURCE_STATE_GENERIC_READ bits that are compatible with the resource description.
- Any write to a resource other than UAV access must be done exclusively by a single queue at a time. When a resource has transitioned to a writeable state on a queue it is considered exclusively owned by that queue and it must transition to the most generic read state before it can be accessed by another queue.

Validation of resource states happens during both command list recording and submission.

Example of split barriers

The following example shows how to use a split barrier to reduce pipeline stalls. The code that follows does not use split barriers:

```
D3D12_RESOURCE_BARRIER BarrierDesc = {};
    BarrierDesc.Type = D3D12_RESOURCE_BARRIER_TRANSITION;
    BarrierDesc.Flags = D3D12_RESOURCE_BARRIER_NONE;
    BarrierDesc.Transition.pResource = pResource;
    BarrierDesc.Transition.Subresource = 0;
    BarrierDesc.Transition.StateBefore = D3D12_RESOURCE_STATE_COMMON;
    BarrierDesc.Transition.StateAfter = D3D12_RESOURCE_STATE_RENDER_TARGET;

    pCommandList->ResourceBarrier( 1, &BarrierDesc );

    Write(pResource); // ... render to pResource
    OtherStuff(); // .. other gpu work

    // Transition pResource to PIXEL_SHADER_RESOURCE
    BarrierDesc.Transition.StateBefore = D3D12_RESOURCE_STATE_RENDER_TARGET;
    BarrierDesc.Transition.StateAfter = D3D12_RESOURCE_STATE_PIXEL_SHADER_RESOURCE;
    pCommandList->ResourceBarrier( 1, &BarrierDesc );

    Read(pResource); // ... read from pResource
```

The following code uses split barriers:

```
D3D12_RESOURCE_BARRIER BarrierDesc = {};
    BarrierDesc.Type = D3D12_RESOURCE_BARRIER_TRANSITION;
    BarrierDesc.Flags = D3D12_RESOURCE_BARRIER_NONE;
    BarrierDesc.Transition.pResource = pResource;
    BarrierDesc.Transition.Subresource = 0;
    BarrierDesc.Transition.StateBefore = D3D12_RESOURCE_STATE_COMMON;
    BarrierDesc.Transition.StateAfter = D3D12_RESOURCE_STATE_RENDER_TARGET;

    pCommandList->ResourceBarrier( 1, &BarrierDesc );

    Write(pResource); // ... render to pResource

    // Done writing to pResource. Start barrier to PIXEL_SHADER_RESOURCE and
    // then do other work
    BarrierDesc.Flags = D3D12_RESOURCE_BARRIER_BEGIN_ONLY;
    BarrierDesc.Transition.StateBefore = D3D12_RESOURCE_STATE_RENDER_TARGET;
    BarrierDesc.Transition.StateAfter = D3D12_RESOURCE_STATE_PIXEL_SHADER_RESOURCE;
    pCommandList->ResourceBarrier( 1, &BarrierDesc );

    OtherStuff(); // .. other gpu work

    // Need to read from pResource so end barrier
    BarrierDesc.Flags = D3D12_RESOURCE_BARRIER_END_ONLY;

    pCommandList->ResourceBarrier( 1, &BarrierDesc );
    Read(pResource); // ... read from pResource
```

Multi-Adapter

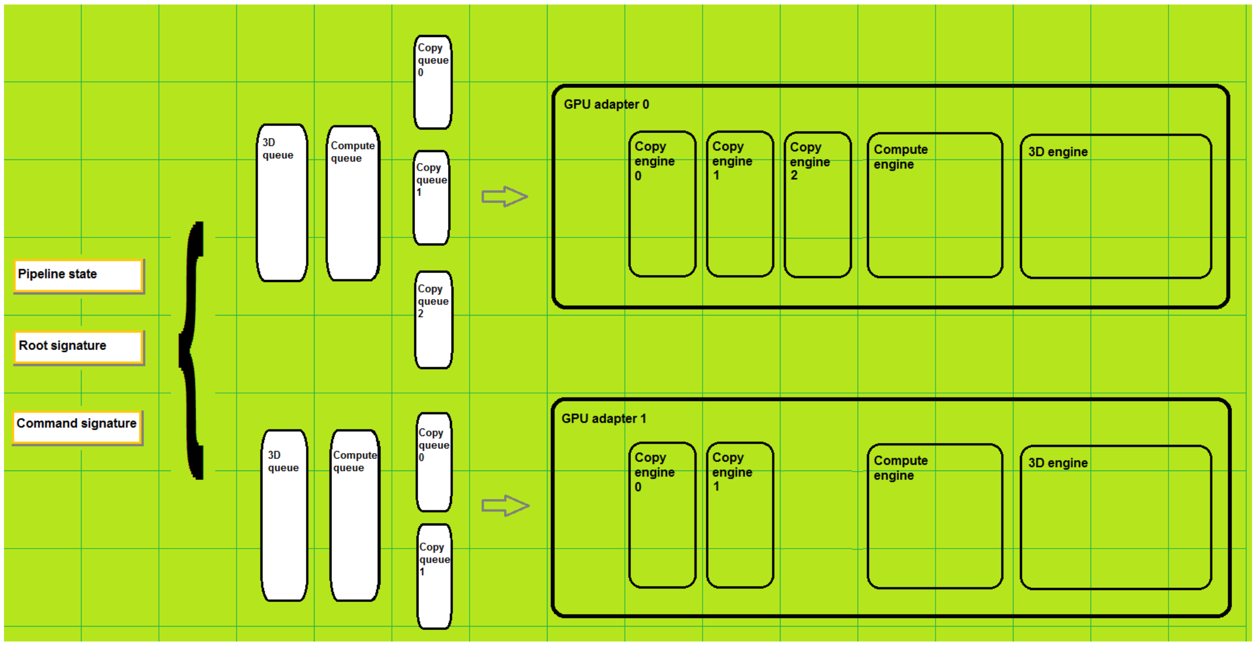
Describes support in D3D12 for multi-engine adapter systems, covering scenarios where applications explicitly target multiple GPU adapters, and scenarios where drivers implicitly use multiple GPU adapters on behalf of an application.

- Multi-adapter Overview
- Sharing heaps across adapters
- Multi-adapter APIs
  - Single nodes
  - Multiple nodes
  - Resource creation APIs
- Related topics

Multi-adapter Overview

A GPU adapter can be any graphics adapter (including an on-board adapter), from any manufacturer, that supports D3D12.

Multiple adapters are referenced as *nodes*. in the API. These nodes are indexed from zero, but in the bit masks used to refer to the nodes, zero translates to bit 1, 1 to bit 2, and so on. A number of elements, such as the queues, apply to each node, so if there are two nodes, there will be two default 3D queues. Other elements, such as the pipeline state and root and command signatures, can refer to one or more or all of the nodes, as shown in the following diagram.



Sharing heaps across adapters

Refer to the Shared Heaps section.

Multi-adapter APIs

Similar to previous D3D APIs, each set of linked adapters is enumerated as a single IDXGIAdapter3 object. All outputs attached to any adapter in the link are enumerated as attached to the single **IDXGIAdapter3** object.

Single nodes

Applications can determine the number of physical adapters associated with a given device with a call to ID3D12Device::GetNodeCount. Many APIs in D3D12 accept a *NodeMask*, which indicates the set of nodes which the API call refers to.

When calling the following (single node) APIs, applications specify a single node that the API call will be associated with. Most of the time this is specified by a *NodeMask*. Each bit in the mask corresponds to a single node. For all of the APIs described in this section, exactly 1 bit must be set in the *NodeMask*.

- D3D12_COMMAND_QUEUE_DESC : has a *NodeMask* member.
- CreateCommandQueue : creates a queue from a D3D12_COMMAND_QUEUE_DESC structure.
- CreateCommandList : takes a *nodeMask* parameter.
- D3D12_DESCRIPTOR_HEAP_DESC : has a *NodeMask* member.
- CreateDescriptorHeap : creates a descriptor heap from a D3D12_DESCRIPTOR_HEAP_DESC structure.
- D3D12_QUERY_HEAP_DESC : has a *NodeMask* member.
- CreateQueryHeap : creates a query heap from a D3D12_QUERY_HEAP_DESC structure.

Multiple nodes

When calling the following APIs, applications specify a set of nodes that the API call will be associated with. Node affinity is specified as a bit mask. If the application passes 0 for the bit mask, then the D3D12 driver converts this to the bit mask 1 (indicating that the object is associated with node 0).

- D3D12_CROSS_NODE_SHARING_TIER : determines the support for cross node sharing.
- D3D12_FEATURE_DATA_D3D12_OPTIONS : structure referencing D3D12_CROSS_NODE_SHARING_TIER.
- D3D12_FEATURE_DATA_ARCHITECTURE : contains a *NodeIndex* member.

- D3D12_GRAPHICS_PIPELINE_STATE_DESC : has a *NodeMask* member.
- CreateGraphicsPipelineState : creates a graphics pipeline state object from a D3D12_GRAPHICS_PIPELINE_STATE_DESC structure.
- D3D12_COMPUTE_PIPELINE_STATE_DESC : has a *NodeMask* member.
- CreateComputePipelineState : creates a compute pipeline state object from a D3D12_COMPUTE_PIPELINE_STATE_DESC structure.
- CreateRootSignature: takes a *nodeMask* parameter.
- D3D12_COMMAND_SIGNATURE_DESC: has a *NodeMask* member.
- CreateCommandSignature : creates a command signature object from a D3D12_COMMAND_SIGNATURE_DESC structure.

Resource creation APIs

The following APIs reference node masks:

- D3D12_HEAP_PROPERTIES : has both *CreationNodeMask* and *VisibleNodeMask* members.
- GetResourceAllocationInfo : has a *visibleMask* parameter.
- GetCustomHeapProperties : has a *nodeMask* parameter.

When creating reserved resource no node index or mask is specified. The reserved resource can be mapped onto a heap on any node (following the cross-node sharing rules).

The method **MakeResident** works internally with adapter queues, there is no need for the application to specify anything for this.

When calling the following **ID3D12Device** APIs, applications do not need to specify a set of nodes that the API call will be associated with because the API call applies to all nodes:

- CreateFence
- GetDescriptorHandleIncrementSize
- SetStablePowerState
- CheckFeatureSupport
- CreateSampler
- CopyDescriptors
- CopyDescriptorsSimple
- CreateSharedHandle
- OpenSharedHandleByName
- OpenSharedHandle : with a *fence* as a parameter. With a *resource* or a *heap* as parameters this method does not accept nodes as parameters because node masks are inherited from previously created objects.
- CreateCommandAllocator
- CreateConstantBufferView
- CreateRenderTargetView
- CreateUnorderedAccessView
- CreateDepthStencilView
- CreateShaderResourceView

Rendering

This section contains information about rendering features new to Direct3D 12 (and Direct3D 11.3).

In this section

Topic	Description
Conservative Rasterization	Conservative Rasterization adds some certainty to pixel rendering, which is helpful in particular to collision detection algorithms.
Indirect Drawing	Indirect drawing enables some scene-traversal and culling to be moved from the CPU to the GPU, which can improve performance. The command buffer doesn't necessarily need to be GPU-generated.
Rasterizer Ordered Views	Rasterizer ordered views (ROVs) allow pixel shader code to mark UAV bindings with a declaration that alters the normal requirements for the order of graphics pipeline results for UAVs. This enables Order Independent Transparency (OIT) algorithms to work, which give much better rendering results when multiple transparent objects are in line with each other in a view.
Shader Specified Stencil Reference Value	Enabling pixel shaders to output the Stencil Reference Value, rather than using the API-specified one, enables a very fine granular control over stencil operations.
Swap Chains	Swap chains control the back buffer rotation, forming the basis of graphics animation.

The following topics are also new to Direct3D 12 and Direct3D:

- Default Texture Mapping
- Typed Unordered Access View Loads
- Volume Tiled Resources

Conservative Rasterization

Conservative Rasterization adds some certainty to pixel rendering, which is helpful in particular to collision detection algorithms.

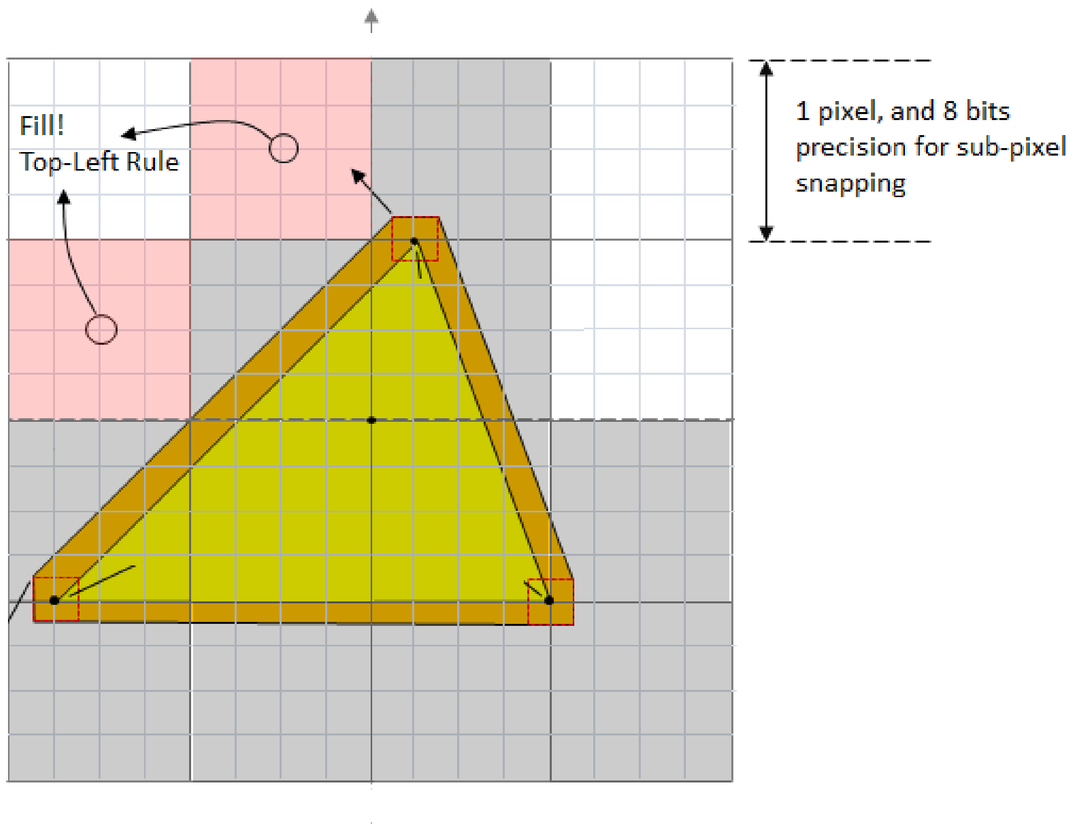
Overview

Conservative Rasterization means that all pixels that are at least partially covered by a rendered primitive are rasterized, which means that the pixel shader is invoked. Normal behavior is sampling, which is not used if Conservative Rasterization is enabled.

Conservative Rasterization is useful in a number of situations, including for certainty in collision detection, occlusion culling, and tiled rendering.

For example, the following figure shows a green triangle rendered using Conservative Rasterization, as it would appear in the rasterizer (that is, using 16.8 fixed point vertex coordinates). The brown area is known as an "uncertainty region" - a conceptual region that represents the extended bounds of the triangle, required to ensure the primitive in the rasterizer is conservative with respect to the original floating point vertex coordinates. The red squares at each vertex shows how the uncertainty region is calculated: as a swept square.

The large gray squares show the pixels that will be rendered. The pink squares show pixels rendered using the "Top-Left Rule", which comes into play as the edge of the triangle crosses the edge of the pixels. There can be false positives (pixels set that should not have been) which the system will normally but not always cull.



Interactions with the pipeline

Rasterization Rules interaction

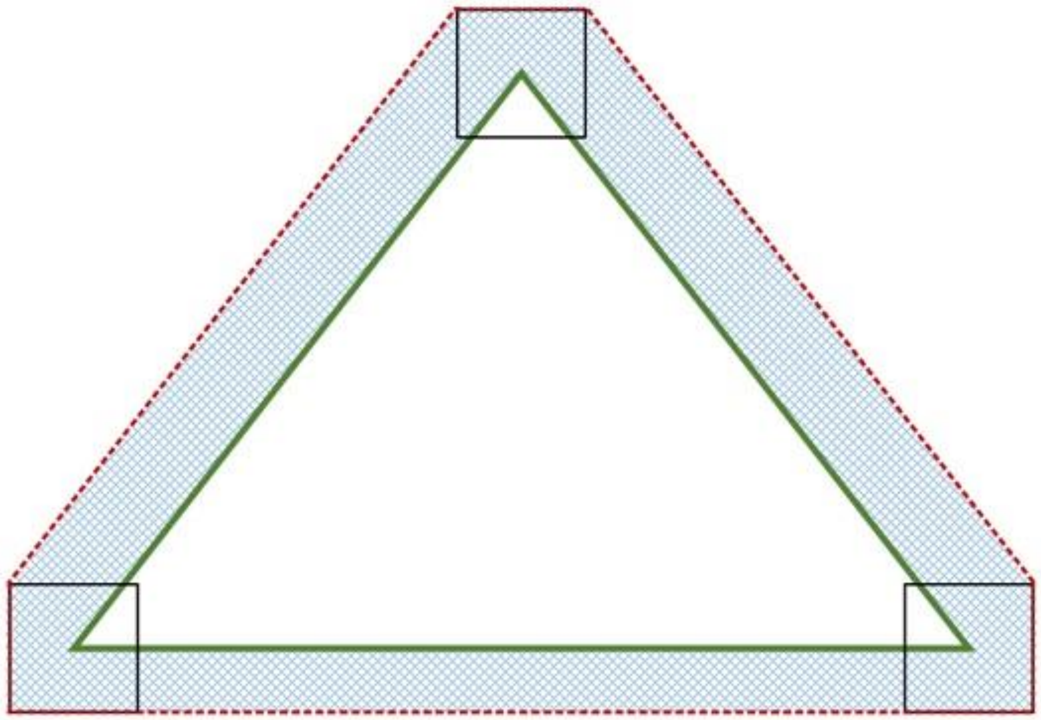
In Conservative Rasterization mode, Rasterization Rules apply the same way as when Conservative Rasterization mode is not enabled with exceptions for the Top-Left Rule, described above, and Pixel Coverage. 16.8 Fixed-Point Rasterizer precision must be used.

Pixels that would not be covered if hardware was using full floating point vertex coordinates may only be included if they are within an uncertainty region no larger half a pixel in the fixed point domain. Future hardware is expected to reach the tightened uncertainty region specified in Tier 2. Note that this requirement prevents sliver triangles from extending further than necessary.

A similar valid uncertainty region applies to `InnerCoverage` as well, but it is tighter since no implementations require a larger uncertainty region for this case. See **InnerCoverage interaction** for more detail.

Inner and outer uncertainty regions must be greater than or equal to the size of half the sub-pixel grid, or 1/512 of a pixel, in the fixed point domain. This is the minimum valid uncertainty region. 1/512 comes from the 16.8 fixed point Rasterizer coordinate representation and the round-to-nearest rule that applies when converting floating point vertex coordinates to 16.8 fixed point coordinates. 1/512 can change if the Rasterizer precision changes. If an implementation implements this minimum uncertainty region, then they must follow the Top-Left Rule when an edge or corner of the uncertainty region falls along the edge or corner of a pixel. The clipped edges of the uncertainty region should be treated as the closest vertex, meaning that it counts as two edges: the two that join at the associated vertex. Top-Left Rule is required when the minimum uncertainty region is used because if it is not, then a Conservative Rasterization implementation would fail to rasterize pixels that could be covered when Conservative Rasterization mode is disabled.

The following diagram illustrates a valid outer uncertainty region produced by sweeping a square around the edges of the primitive in the fixed point domain (i.e. the vertices have been quantized by the 16.8 fixed point representation). The dimensions of this square are based on the valid outer uncertainty region size: for the 1/2 of a pixel, the square is 1 pixel in width and height, for 1/512 of a pixel, the square is 1/256 of a pixel in width and height. The green triangle represents a given primitive, the red dotted line represents the bound on Overestimated Conservative Rasterization, the solid black squares represent the square that is swept along the primitive edges, and the blue checkered area is the outer uncertainty region:



Multisampling interaction

Regardless of the number of samples in **RenderTarget**/**DepthStencil** surfaces (or whether *ForcedSampleCount* is being used or not), all samples are covered for pixels rasterized by Conservative Rasterization. Individual sample locations are not tested for whether they fall in the primitive or not.

SampleMask interaction

The *SampleMask* Rasterizer State applies the same way as when Conservative Rasterization is not enabled for `InputCoverage`, but does not affect `InnerCoverage`(i.e. it is not AND'ed into an input declared with `InnerCoverage`). This is because `InnerCoverage` is unrelated to whether MSAA samples are masked out: `0InnerCoverage` only means that the pixel is not guaranteed to be fully covered, not that no samples will be updated.

Depth/Stencil Test interaction

Depth/Stencil Testing proceeds for a conservatively rasterized pixel the same way as if all samples are covered when Conservative Rasterization is not enabled.

Proceeding with all samples covered can cause Depth Extrapolation, which is valid and must be clamped to the viewport as specified when Conservative Rasterization is not enabled. This is similar to when pixel-frequency interpolation modes are used on a **RenderTarget** with sample count greater than 1, although in the case of

Conservative Rasterization, it is the depth value going into the fixed function depth test that can be extrapolated.

Early Depth culling behavior with Depth Extrapolation is undefined. This is because some Early Depth culling hardware cannot properly support extrapolated depth values. However, Early Depth culling behavior in the presence of Depth Extrapolation is problematic even with hardware that can support extrapolated depth values. This issue can be worked around by clamping the Pixel Shader input depth to the min and max depth values of the primitive being rasterized and writing that value to `oDepth` (the pixel shader output depth register). Implementations are required to disable Early Depth culling in this case, due to the`oDepth` write.

Helper Pixel interaction

Helper Pixel rules apply the same way as when Conservative Rasterization is not enabled. As part of this, all pixels including Helper Pixels must report`InputCoverage` accurately as specified in the `InputCoverage` interaction section. So fully non-covered pixels report 0 coverage.

Output Coverage interaction

Output Coverage (`oMask`) behaves for a conservatively rasterized pixel as it does when Conservative Rasterization is not enabled with all samples covered.

InputCoverage interaction

In Conservative Rasterization mode, this input register is populated as if all samples are covered when Conservative Rasterization is not enabled for a given conservatively rasterized pixel. That is to say, all existing interactions apply (e.g. *SampleMask* is applied), and the first n bits in `InputCoverage` from the LSB are set to 1 for a conservatively rasterized pixel, given an n sample per pixel **RenderTarget** and/or **DepthStencil** buffer is bound at the **Output Merger**, or an n sample *ForcedSampleCount*. The rest of the bits are 0.

This input is available in a shader regardless of the use of Conservative Rasterization, though Conservative Rasterization changes its behavior to only show all samples covered (or none for Helper Pixels).

InnerCoverage interaction

This feature is required by, and only available in, Tier 3. The runtime will fail shader creation for shaders that use this mode when an implementation supports a Tier less than Tier 3.

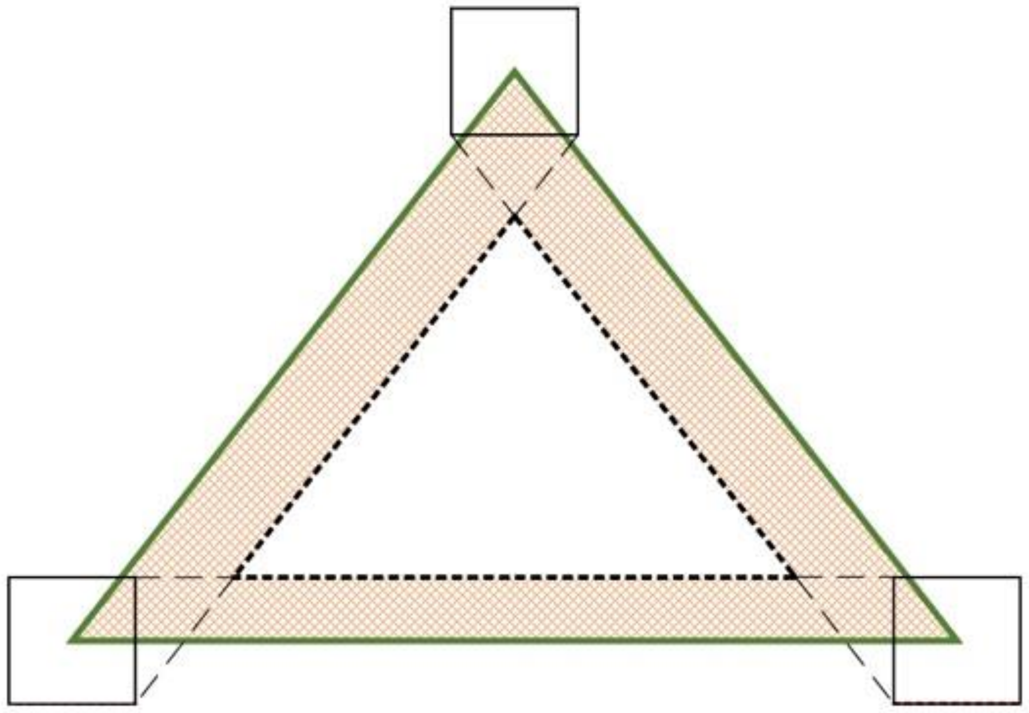
The Pixel Shader has a 32-bit scalar integer System Generate Value available: `InnerCoverage`. This is a bit-field that has bit 0 from the LSB set to 1 for a given conservatively rasterized pixel, only when that pixel is guaranteed to be entirely inside the current primitive. All other input register bits must be set to 0 when bit 0 is not set, but are undefined when bit 0 is set to 1 (essentially, this bit-field represents a Boolean value where false must be exactly 0, but true can be any odd (i.e. bit 0 set) non-zero value). This input is used for underestimated Conservative Rasterization information. It informs the Pixel Shader whether the current pixel lies completely inside the geometry.

This must account for snapping error at resolutions greater than or equal to the resolution at which the current Draw is operating. There must not be false positives (setting `InnerCoverage` bits when the pixel is not fully covered for any snapping error at resolutions greater than or equal to the resolution at which the current Draw is operating), but false negatives are allowed. In summary, the implementation must not incorrectly identify pixels as fully covered that would not be with full floating point vertex coordinates in the Rasterizer.

Pixels that would be fully covered if hardware was using full floating point vertex coordinates may only be omitted if they intersect the inner uncertainty region, which must be no larger than the size of the sub-pixel grid, or 1/256 of a pixel, in the fixed point domain. Said another way, pixels entirely within the inner boundary of the inner uncertainty region must be marked as fully covered. The inner boundary of the uncertainty region is illustrated in the diagram below by the bold black dotted line. 1/256 comes from the 16.8 fixed point Rasterizer coordinate representation, which can change if the Rasterizer precision changes. This uncertainty region is enough to account for snapping error caused by the conversion of floating point vertex coordinates to fixed point vertex coordinates in the Rasterizer.

The same 1/512 minimum uncertainty region requirements defined in Rasterization Rules interaction apply here as well.

The following diagram illustrates a valid inner uncertainty region produced by sweeping a square around the edges of the primitive in the fixed point domain (i.e. the vertices have been quantized by the 16.8 fixed point representation). The dimensions of this square are based on the valid inner uncertainty region size: for 1/256 of a pixel, the square is 1/128 of a pixel in width and height. The green triangle represents a given primitive, the bold black dotted line represents the boundary of the inner uncertainty region, the solid black squares represent the square that is swept along the primitive edges, and the orange checkered area is the inner uncertainty region:



The use of `InnerCoverage` does not affect whether a pixel is conservatively rasterized, i.e. using one of these `InputCoverage` modes does not affect which pixels are rasterized when Conservative Rasterization mode is enabled. Therefore, when `InnerCoverage` is used and the Pixel Shader is processing a pixel that is not completely covered by the geometry its value will be 0, but the Pixel Shader invocation will have samples updated. This is different from when `InputCoverage`is 0, meaning that no samples will be updated.

This input is mutually exclusive with `InputCoverage` : both cannot be used.

To access `InnerCoverage`, it must be declared as a single component out of one of the Pixel Shader input registers. The interpolation mode on the declaration must be constant (interpolation does not apply).

The `InnerCoverage` bit-field is not affected by depth/stencil tests, nor is it ANDed with the *SampleMask* Rasterizer state.

This input is only valid in Conservative Rasterization mode. When Conservative Rasterization is not enabled, `InnerCoverage` produces an undefined value.

Pixel Shader invocations caused by the need for Helper Pixels, but otherwise not covered by the primitive, must have the `InnerCoverage` register set to 0.

Attribute Interpolation interaction

Attribute interpolation modes are unchanged and proceed the same way as when Conservative Rasterization is not enabled, where the viewport-scaled and fixed-point-converted vertices are used. Because all samples in a conservatively rasterized pixel are considered covered, it is valid for values to be extrapolated, similar to when pixel-frequency interpolation modes are used on a **RenderTarget** with sample count greater than 1. Centroid interpolation modes produce results identical to the corresponding non-centroid interpolation mode; the notion of centroid is meaningless in this scenario – where sample coverage is only either full or 0.

Conservative Rasterization allows for degenerate triangles to produce Pixel Shader invocations, therefore, degenerate triangles must use the values assigned to Vertex 0 for all interpolated values.

Clipping interaction

When Conservative Rasterization mode is enabled and depth clip is disabled (when the *DepthClipEnable* Rasterizer State is set to FALSE), there may be variances in attribute interpolation for segments of a primitive that fall outside the 0 <= z <= w range, depending on implementation: either constant values are used from a point where the primitive intersects the relevant plane (near or far), or attribute interpolation behaves as when Conservative Rasterization mode is disabled. However, the depth value behavior is the same regardless of Conservative Rasterization mode, i.e. primitives that fall outside of the depth range must still be given the value of the nearest limit of the viewport depth range. Attribute interpolation behavior inside the 0 <= z <= w range must remain unchanged.

Clip Distance interaction

Clip Distance is valid when Conservative Rasterization mode is enabled, and behaves for a conservatively rasterized pixel as it does when Conservative Rasterization is not enabled with all samples covered.

Note that Conservative Rasterization can cause extrapolation of the W vertex coordinate, which may cause W <= 0. This could cause per-pixel Clip Distance implementations to operate on a Clip Distance that has been Perspective Divided by an invalid W value. Clip Distance implementations must guard against invoking rasterization for pixels where vertex coordinate W <= 0 (e.g. due to extrapolation when in Conservative Rasterization mode).

Target Independent Rasterization interaction

Conservative Rasterization mode is compatible with Target Independent Rasterization (TIR). TIR rules and restrictions apply, behaving for a conservatively rasterized pixel as if all samples are covered.

IA Primitive Topology interaction

Conservative Rasterization is not defined for line or point primitives. Therefore, Primitive Topologies that specify points or lines produce undefined behavior if they are fed to the rasterizer unit when Conservative Rasterization is enabled.

The debug layer validation verifies applications do not use these Primitive Topologies.

Query interaction

For a conservatively rasterized pixel, queries behave as they do when Conservative Rasterization is not enabled when all samples are covered. For example, for a conservatively rasterized pixel, D3D12_QUERY_TYPE_OCCLUSION and D3D12_QUERY_TYPE_PIPELINE_STATISTICS (from **D3D12_QUERY_TYPE**) must behave as they would when Conservative Rasterization is not enabled when all samples are covered.

Pixel Shader invocations should increment for every conservatively rasterized pixel in Conservative Rasterization mode.

Cull State interaction

All Cull States are valid in Conservative Rasterization mode and follow the same rules as when Conservative Rasterization is not enabled.

When comparing Conservative Rasterization across resolutions to itself or without Conservative Rasterization enabled, there is the possibility that some primitives may have mismatched facedness (i.e. one back facing, the other front facing). Applications can avoid this uncertainty by using D3D12_CULL_MODE_NONE (from **D3D12_CULL_MODE**) and not using the `IsFrontFace` System Generated Value.

IsFrontFace interaction

The `IsFrontFace` System Generated Value is valid to use in Conservative Rasterization mode, and follows the behavior defined when Conservative Rasterization is not enabled.

Fill Modes interaction

The only valid **D3D12_FILL_MODE** for Conservative Rasterization is D3D12_FILL_SOLID, any other fill mode is an invalid parameter for the Rasterizer State.

This is because D3D12 functional specification specifies that wireframe fill mode should convert triangle edges to lines and follow the line rasterization rules and conservative line rasterization behavior has not been defined.

Implementation details

The type of rasterization supported in Direct3D 12 is sometimes referred to as "Overestimated Conservative Rasterization". There is also the concept of "Underestimated Conservative Rasterization", which means that only pixels that are fully covered by a rendered primitive are rasterized. Underestimated Conservative Rasterization information is available through the pixel shader through the use of input coverage data, and only overestimated Conservative Rasterization is available as a rasterizing mode.

If any part of a primitive overlaps a pixel, then that pixel is considered covered and is then rasterized. When an edge or corner of a primitive falls along the edge or corner of a pixel, the application of the "top-left rule" is implementation-specific. However, for implementations that support degenerate triangles, a degenerate triangle along an edge or corner must cover at least one pixel.

Conservative Rasterization implementations can vary on different hardware, and do produce false positives, meaning that they can incorrectly decide that pixels are covered. This can occur because of implementation-specific details like primitive growing or snapping errors inherent in the fixed-point vertex coordinates used in rasterization. The reason false positives (with respect to fixed point vertex coordinates) are valid is because some amount of false positives are needed to allow an implementation to do coverage evaluation against post-snapped vertices (i.e. vertex coordinates that have been converted from floating point to the 16.8 fixed-point used in the rasterizer), but honor the coverage produced by the original floating point vertex coordinates.

Conservative Rasterization implementations do not produce false negatives with respect to the floating-point vertex coordinates for non-degenerate post-snap primitives: if any part of a primitive overlaps any part of a pixel, then that pixel is rasterized.

Triangles that are degenerate (duplicate indices in an index buffer or collinear in 3D), or become degenerate after fixed-point conversion (collinear vertices in the rasterizer), may or may not be culled; both are valid behaviors. Degenerate triangles must be considered back facing, so if a specific behavior is required by an application, it can use back-face culling or test for front facing. Degenerate triangles use the values assigned to Vertex 0 for all interpolated values.

There are three tiers of hardware support, in addition to the possibility that the hardware does not support this feature.

- Tier 1 enforces a maximum 1/2 pixel uncertainty region and does not support post-snap degenerates. This is good for tiled rendering, a texture atlas, light map generation and sub-pixel shadow maps.
- Tier 2 reduces the maximum uncertainty region to 1/256 and requires post-snap degenerates not be culled. This tier is helpful for CPU-based algorithm acceleration (such as voxelization).
- Tier 3 maintains a maximum 1/256 uncertainty region and adds support for inner input coverage. Inner input coverage adds the new value`SV_InnerCoverage` to High Level Shading Language (HLSL). This is a 32-bit scalar integer that can be specified on input to a pixel shader, and represents the underestimated Conservative Rasterization information (that is, whether a pixel is guaranteed-to-be-fully covered). This tier is helpful for occlusion culling.

API summary

The following methods, structures, enums, and helper classes reference Conservative Rasterization:

- D3D12_RASTERIZER_DESC : structure holding the rasterizer description.
- D3D12_CONSERVATIVE_RASTERIZATION_MODE : enum values for the mode (on or off).
- D3D12_FEATURE_DATA_D3D12_OPTIONS : structure holding the tier of support.
- D3D12_CONSERVATIVE_RASTERIZATION_TIER : enum values for each tier of support by the hardware.
- CheckFeatureSupport : method to access the supported features.
- CD3DX12_RASTERIZER_DESC : helper class for creating rasterizer descriptions.

Indirect Drawing

Indirect drawing enables some scene-traversal and culling to be moved from the CPU to the GPU, which can improve performance. The command buffer doesn't necessarily need to be GPU-generated.

- Command Signatures
- Indirect Argument Buffer Structures
- Command Signature Creation

o No Argument Changes

o Root Constants and Vertex Buffers

- Related topics

Command Signatures

The command signature object (ID3D12CommandSignature) enables apps to specify indirect drawing, in particular setting the following:

- The indirect argument buffer format.
- The command type that will be used (from the ID3D12GraphicsCommandList methods DrawInstanced, DrawIndexedInstanced, or Dispatch).
- The set of resource bindings which will change per-command call versus the set which will be inherited.

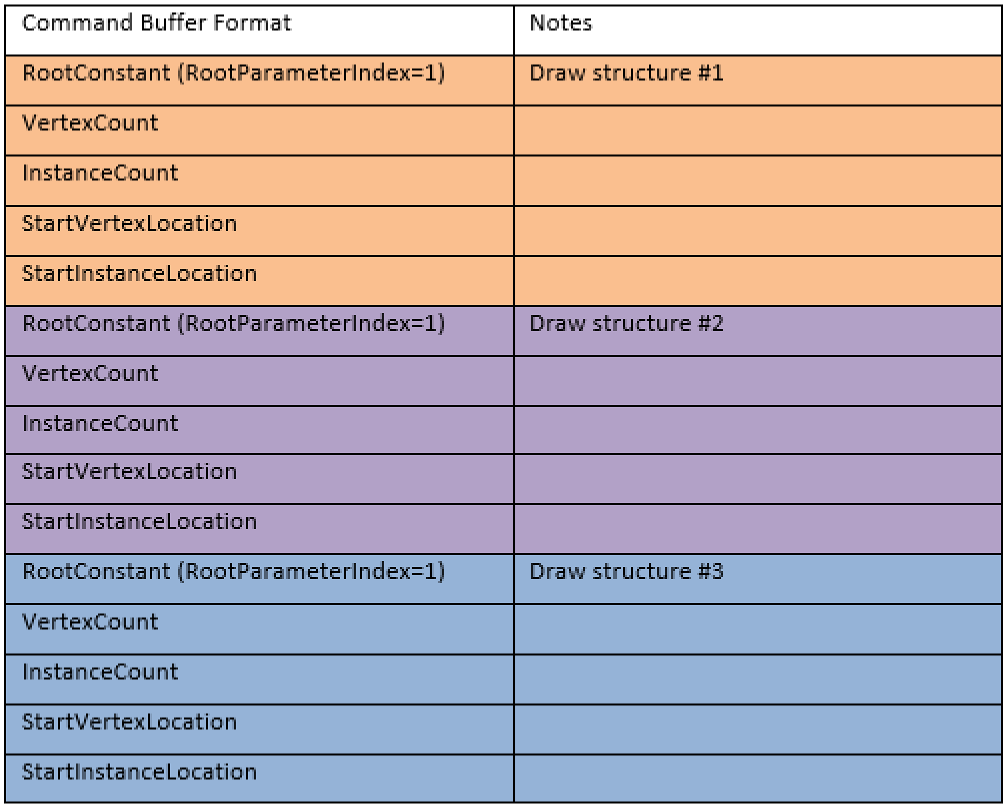
At startup, an app creates a small set of command signatures. At runtime, the application fills a buffer with commands (via whatever means the app developer chooses). The app then uses D3D12 command list APIs to set state (render target bindings, PSO, etc), and then uses ExecuteIndirect to instruct the GPU to interpret the contents of the indirect argument buffer according to the format defined by a particular command signature.

For example, suppose an app developer wants a unique root constant to be specified per-draw call in the indirect argument buffer. The app would create a command signature that enables the indirect argument buffer to specify the following parameters per draw call:

- The value of one root constant.
- The draw arguments (vertex count, instance count, etc).

The indirect argument buffer generated by the application would contain an array of fixed-size records. Each structure corresponds to one draw call. Each structure contains the drawing arguments, and the value of the root constant. The number of draw calls is specified in a separate GPU-visible buffer.

An example command buffer generated by the app follows:



Command Buffer Format	Notes
RootConstant (RootParameterIndex=1)	Draw structure #1
VertexCount	
InstanceCount	
StartVertexLocation	
StartInstanceLocation	
RootConstant (RootParameterIndex=1)	Draw structure #2
VertexCount	
InstanceCount	
StartVertexLocation	
StartInstanceLocation	
RootConstant (RootParameterIndex=1)	Draw structure #3
VertexCount	
InstanceCount	
StartVertexLocation	
StartInstanceLocation	

Indirect Argument Buffer Structures

The following structures define how particular arguments appear in an indirect argument buffer. These structures do not appear in any D3D12 API. Applications use these definitions when writing to an indirect argument buffer (with the CPU or GPU):

- D3D12_DRAW_ARGUMENTS
- D3D12_DRAW_INDEXED_ARGUMENTS
- D3D12_DISPATCH_ARGUMENTS
- D3D12_VERTEX_BUFFER_VIEW
- D3D12_INDEX_BUFFER_VIEW
- D3D12_GPU_VIRTUAL_ADDRESS (a typedef'd synonym of UINT64).
- D3D12_CONSTANT_BUFFER_VIEW

Command Signature Creation

To create a command signature, use the following API items:

- ID3D12Device::CreateCommandSignature (outputs an ID3D12CommandSignature)
- D3D12_INDIRECT_ARGUMENT_TYPE
- D3D12_INDIRECT_ARGUMENT_DESC
- D3D12_COMMAND_SIGNATURE_DESC

The ordering of arguments within an indirect argument buffer is defined to exactly match the order of arguments specified in the pArguments parameter ofD3D12_COMMAND_SIGNATURE_DESC. All of the arguments for one draw/dispatch call within an indirect argument buffer are tightly packed. However, applications are allowed to specify an arbitrary byte stride between draw/dispatch commands in an indirect argument buffer.

The root signature must be specified if and only if the command signature changes one of the root arguments.

For root SRV/UAV/CBV, the application specified size is in bytes. The debug layer will validate the following restrictions on the address:

- CBV – address must be a multiple of 256 bytes.
- Raw SRV/UAV – address must be a multiple of 4 bytes.
- Structured SRV/UAV – address must be a multiple of the structure byte stride (declared in the shader).

A given command signature is either a draw or a compute command signature. If a command signature contains a drawing operation, then it is a graphics command signature. Otherwise, the command signature must contain a dispatch operation, and it is a compute command signature.

The following sections show some example command signatures.

No Argument Changes

In this example, the indirect argument buffer generated by the application holds an array of 36-byte structures. Each structure only contains the five parameters passed to DrawIndexedInstanced (plus padding).

The code to create the command signature description follows:

```
D3D12_INDIRECT_ARGUMENT_DESC Args[1];
Args[0].Type = D3D12_INDIRECT_ARGUMENT_TYPE_DRAW_INDEXED;

D3D12_COMMAND_SIGNATURE_DESC ProgramDesc;
```

```
ProgramDesc.ByteStride = 36;
ProgramDesc.ArgumentCount = 1;
ProgramDesc.pArguments = Args;
```

The layout of a single structure within an indirect argument buffer is:

Bytes 0:3	**IndexCountPerInstance**
Bytes 4:7	InstanceCount
Bytes 8:11	StartIndexLocation
Bytes 12:15	BaseVertexLocation
Bytes 16:19	StartInstanceLocation
Bytes 20:35	Padding

Root Constants and Vertex Buffers

In this example, each structure in an indirect argument buffer changes two root constants, changes one vertex buffer binding, and performs one drawing non-indexed operation. There is no padding between structures.

The code to create the command signature description is:

```
D3D12_INDIRECT_ARGUMENT_DESC Args[4];
Args[0].Type = D3D12_INDIRECT_ARGUMENT_TYPE_CONSTANT;
Args[0].Constant.RootParameterIndex = 2;

Args[1].Type = D3D12_INDIRECT_ARGUMENT_TYPE_CONSTANT;
Args[1].Constant.RootParameterIndex = 6;

Args[2].Type = D3D12_INDIRECT_ARGUMENT_TYPE_VERTEX_BUFFER_VIEW;
Args[2].VertexBuffer.VBSlot = 3;

Args[3].Type = D3D12_INDIRECT_ARGUMENT_TYPE_DRAW_INSTANCED;

D3D12_COMMAND_SIGNATURE_DESC ProgramDesc;
ProgramDesc.ByteStride = 40;
ProgramDesc.ArgumentCount = 4;
ProgramDesc.pArguments = Args;
```

The layout of a single structure within the indirect argument buffer is the following:

Bytes 0:3	**Data for root parameter index 2**
Bytes 4:7	Data for root parameter index 6
Bytes 8:15	Virtual address of VB (64-bit)
Bytes 16:19	VB stride
Bytes 20:23	VB size
Bytes 24:27	VertexCountPerInstance
Bytes 28:31	InstanceCount
Bytes 32:35	StartVertexLocation
Bytes 36:39	StartInstanceLocation

Rasterizer Ordered Views

Rasterizer ordered views (ROVs) allow pixel shader code to mark UAV bindings with a declaration that alters the normal requirements for the order of graphics pipeline results for UAVs. This enables Order Independent Transparency (OIT) algorithms to work, which give much better rendering results when multiple transparent objects are in line with each other in a view.

- Overview

Overview

Standard graphics pipelines may have trouble correctly compositing together multiple textures that contain transparency. Objects such as wire fences, smoke, fire, vegetation, and colored glass use transparency to get the desired effect. Problems arise when multiple textures that contain transparency are in line with each other (smoke in front of a fence in front of a glass building containing vegetation, as an example). Rasterizer ordered views (ROVs) enable the underlying OIT algorithms to use features of the hardware to try to resolve the transparency order correctly. Transparency is handled by the pixel shader.

Rasterizer ordered views (ROVs) allow pixel shader code to mark UAV bindings with a declaration that alters the normal requirements for the order of graphics pipeline results for UAVs.

ROVs guarantee the order of UAV accesses for any pair of overlapping pixel shader invocations. In this case “overlapping” means that the invocations are generated by the same draw calls and share the same pixel coordinate when in pixel-frequency execution mode, and the same pixel and sample coordinate in sample-frequency mode.

The order in which overlapping ROV accesses of pixel shader invocations are executed is identical to the order in which the geometry is submitted. This means that, for overlapping pixel shader invocations, ROV writes performed by a pixel shader invocation must be available to be read by a subsequent invocation and must not affect reads by a previous invocation. ROV reads performed by a pixel shader invocation must reflect writes by a previous invocation and must not reflect writes by a subsequent invocation. This is important for UAVs because they are explicitly omitted from the output-invariance guarantees normally set by the fixed order of graphics pipeline results.

Implementation details

Rasterizer ordered views (ROVs) are declared with the following new High Level Shader Language (HLSL) objects, and are only available to the pixel shader:

- `RasterizerOrderedBuffer`
- `RasterizerOrderedByteAddressBuffer`
- `RasterizerOrderedStructuredBuffer`
- `RasterizerOrderedTexture1D`
- `RasterizerOrderedTexture1DArray`
- `RasterizerOrderedTexture2D`
- `RasterizerOrderedTexture2DArray`
- `RasterizerOrderedTexture3D`

Use these objects in the same manner as other UAV objects (such as `RWBuffer` etc.).

API summary

ROVs are an HLSL-only construct that applies different behavior semantics to UAVs. All APIs relevant to UAVs are also relevant to ROVs. Note that the following method, structures, and helper class reference the rasterizer:

- D3D12_RASTERIZER_DESC : structure holding the rasterizer description.
- D3D12_FEATURE_DATA_D3D12_OPTIONS : structure holding a boolean, indicating the support.
- CheckFeatureSupport : method to access the supported features.

- D3D12_GRAPHICS_PIPELINE_STATE_DESC : structure holding the pipeline state.

Shader Specified Stencil Reference Value

Enabling pixel shaders to output the Stencil Reference Value, rather than using the API-specified one, enables a very fine granular control over stencil operations.

The Stencil Reference Value is normally specified by the ID3D12GraphicsCommandList::OMSetStencilRef method. This method sets the stencil reference value on a per-draw granularity. However, this value can be overwritten by the pixel shader.

This D3D12 (and D3D11.3) feature enables developers to read and use the Stencil Reference Value (*SV_StencilRef*) that is output from a pixel shader, enabling a per-pixel or per-sample granularity.

The shader specified value replaces the API-specified reference value for that invocation, which means the change affects both the stencil test, and when the stencil operation D3D12_STENCIL_OP_REPLACE (one member of D3D12_STENCIL_OP) is used to write the reference value to the stencil buffer.

This feature is optional in both D3D12 and D3D11.3. To test for its support, check the *PSSpecifiedStencilRefSupported* boolean field ofD3D12_FEATURE_DATA_D3D12_OPTIONS using CheckFeatureSupport.

Here is an example of the use of *SV_StencilRef* in a pixel shader:

```
float main2(float4 c : COORD) : SV_StencilRef {
    return c;
}
```

Swap Chains

Swap chains control the back buffer rotation, forming the basis of graphics animation.

Overview

The programming model for swap chains in D3D12 is not identical to that in earlier versions of D3D. The programming convenience, for example, of supporting automatic resource rotation that was present in D3D10 and D3D11 is not now supported. Automatic resource rotation enabled apps to render the same API object while the actual surface being rendered changes each frame. The behavior of swap chains is changed with D3D12 to enable other features of D3D12 to have low CPU overhead.

Buffer lifetime

Apps are allowed to store pre-created descriptors which reference back buffers This is enabled by ensuring that the set of buffers owned by a swap chain never changes for the lifetime of the swap chain. The set of buffers returned by IDXGISwapChain::GetBuffer does not change until certain APIs are called:

- IDXGISwapChain::ResizeTarget
- IDXGISwapChain::ResizeBuffers

The order of buffers returned by GetBuffer never changes.

IDXGISwapChain3::GetCurrentBackBufferIndex returns the index of the current back buffer to the app.

Swap effects

The only supported swap effect is FLIP_SEQUENTIAL, which requires the buffer count to be greater than one.

Transitioning between windowed and full-screen modes

D3D12 maintains the restriction that applications must call ResizeBuffers after transitioning between windowed and full-screen modes (D3D11 flip-model swap chains have the same restrictions).

The IDXGISwapChain::SetFullscreenState transitions do not change the set of app-visible buffers in the swap chain. Only the ResizeBuffers andResizeTarget calls create or destroy app-visible buffers.

When IDXGISwapChain::Present is called, the back buffer to be presented must be in the D3D12_RESOURCE_STATE_PRESENT state. Present will fail with DXGI_ERROR_INVALID_CALL if this is not the case.

Full-screen swap chains continue to have the restriction that SetFullscreenState(FALSE, NULL) must be called before the final release of the swap chain.**SetFullscreenState**(FALSE) succeeds on swap chains running on D3D12 devices.

Present operations occur on the default 3D queue associated with the device., and apps are free to concurrently present multiple swap chains, and record and execute command lists.

Example

The following example code would be present in the main rendering loop:

```
CComPtr<IDXGISwapChain3> spSwapChain3;
m_spSwapChain->QueryInterface(&spSwapChain3);
UINT backBufferIndex = spSwapChain3->GetCurrentBackBufferIndex();

CComPtr<ID3D12Resource> spBackBuffer;
m_spSwapChain->GetBuffer(backBufferIndex, IID_PPV_ARGS(&spBackBuffer));

// Record and execute a command list referencing spBackBuffer
m_spSwapChain->Present(0, 0);
```

Counters, Queries and Performance Measurement

The following sections describe features for use in performance testing and improvement, such as queries, counters, timing, and predication.

In this section

Topic	Description
Stream-Output Counters, UAV Counters, Queries, and Predication	Stream output and UAV counters operate in Direct3D 12 in a similar method to Direct3D 11, although now memory for the counters must be allocated by the app, the driver does not do it. Queries in Direct3D 12 are more different from those in Direct3D 11, with the addition of fences and other processes that remove the need for some query types.
Timing	This section covers timestamps, and calibrating the GPU and CPU clocks.
Predication	Predication is a feature that enables the GPU rather than the CPU to determine to not draw, copy or dispatch an object.

Stream-Output Counters, UAV Counters, Queries, and Predication

Stream output and UAV counters operate in Direct3D 12 in a similar method to Direct3D 11, although now memory for the counters must be allocated by the app, the driver does not do it. Queries in Direct3D 12 are more different from those in Direct3D 11, with the addition of fences and other processes that remove the need for some query types.

In this section

Topic	Description
Stream Output Counters	Stream output is the ability of the GPU to write vertices to a buffer. The stream output counters monitor progress.
UAV Counters	UAV counters can be used to associate a 32-bit atomic counter with an unordered-access-view (UAV).
Queries	In Direct3D 12, queries are grouped into arrays of queries called a query heap. A query heap has a type which defines the valid types of queries that can be used with that heap.

Stream Output Counters

Stream output is the ability of the GPU to write vertices to a buffer. The stream output counters monitor progress.

- Differences in Stream Counters from Direct3D 11 to Direct3D 12
- BufferFilledSize
- Related topics

Differences in Stream Counters from Direct3D 11 to Direct3D 12

As a part of the stream output process, the GPU has to know the current location in the buffer that it is writing to. In Direct3D 11, memory to store this location is allocated by the driver and the only way for applications to manipulate this value is via the **SOSetTargets** method. In Direct3D 12, apps allocate memory to store this current location. There are no special ways to manipulate this value, and apps are free to read/write the value with the CPU or GPU.

BufferFilledSize

The application is responsible for allocating storage for a 32-bit quantity called the *BufferFilledSize*. This contains the number of bytes of data in the stream-output buffer. This storage can be placed in the same, or a different, resource as the one that contains the stream-output data. This value is accessed by the GPU in the stream-output stage to determine where to append new vertex data in the buffer. Additionally, this value is accessed by the GPU to determine when overflow has occurred.

Refer to the structure D3D12_STREAM_OUTPUT_DESC.

The debug layer will validate the following in ID3D12GraphicsCommandList::SOSetTargets:

- *BufferFilledSize* falls in the range implied by {*OffsetInBytes*, *SizeInBytes*}, if a non-NULL resource is specified.
- *BufferFilledSizeOffsetInBytes* is a multiple of 4.
- *BufferFilledSizeOffsetInBytes* is within the range of the containing resource.

- The specified resource is a buffer.

The runtime will not validate the heap type associated with the stream output buffer, as stream output is supported in all heap types.

Root signatures must specify if stream output will be used, by using the D3D12_ROOT_SIGNATURE_FLAGS flags.

D3D12_ROOT_SIGNATURE_FLAG_ALLOW_STREAM_OUTPUT can be specified for root signatures authored in HLSL, in a manner similar to how the other flags are specified.

CreateGraphicsPipelineState will fail if the geometry shader contains stream-output but the root signature does not have the D3D12_ROOT_SIGNATURE_FLAG_ALLOW_STREAM_OUTPUT flag set.

When a resource is used as a stream-output target, the resources used must be in the D3D12_RESOURCE_STATE_STREAM_OUT state. This applies to both the vertex data and the *BufferFilledSize* (which can be in the same or separate resources).

There are no special APIs to set stream-output buffer offsets because applications can write to the *BufferFilledSize* with the CPU or GPU directly.

UAV Counters

UAV counters can be used to associate a 32-bit atomic counter with an unordered-access-view (UAV).

- Differences in UAV Counters from Direct3D 11 to Direct3D 12
- Using UAV Counters
- Related topics

Differences in UAV Counters from Direct3D 11 to Direct3D 12

In Direct3D 12 apps use the same HLSL shader functions as Direct3D 11 to access the UAV counters:

- **IncrementCounter**
- **DecrementCounter**
- **Append**
- **Consume**

Outside of the shaders Direct3D 11 uses API methods to access the counters, in Direct3D 12 the 32-bit values are allocated by the app so the 32-bit values can be read and written to by the CPU or GPU just like any other Direct3D 12 resource.

Using UAV Counters

The app is responsible for allocating 32-bits of storage for UAV counters. This storage can be allocated in a different resource as the one that contains data accessible via the UAV.

Refer to CreateUnorderedAccessView, D3D12_BUFFER_UAV_FLAGS and D3D12_BUFFER_UAV.

If *pCounterResource* is specified in the call to CreateUnorderedAccessView, then there is a counter associated with the UAV. In this case:

- *StructureByteStride* must be greater than zero
- Format must be DXGI_FORMAT_UNKNOWN
- The RAW flag must not be set

- Both of the resources must be buffers
- *CounterOffsetInBytes* must be a multiple of 4 bytes
- *CounterOffsetInBytes* must be within the range of the counter resource
- *pDesc* cannot be NULL
- *pResource* cannot be NULL

And note the following use cases:

- If *pCounterResource* is not specified, then *CounterOffsetInBytes* must be 0.
- If the RAW flag is set then the format must be DXGI_FORMAT_R32_TYPELESS and the UAV resource must be a buffer.
- If *pCounterResource* is not set, then *CounterOffsetInBytes* must be 0.
- If the RAW flag is not set and *StructureByteStride* = 0, then the format must be a valid UAV format.

Direct3D 12 removes the distinction between append and counter UAVs (although the distinction still exists in HLSL bytecode).

The core runtime will validate these restrictions inside of **CreateUnorderedAccessView**.

During Draw/Dispatch, the counter resource must be in the state **D3D12_RESOURCE_STATE_UNORDERED_ACCESS**. Also, within a single Draw/Dispatch call, it is invalid for an application to access the same 32-bit memory location via two separate UAV counters. The debug layer will issue errors if either of these is detected.

There are no "SetUnorderedAccessViewCounterValue" or "CopyStructureCount" methods because apps can simply copy data to and from the counter value directly.

Dynamic indexing of UAVs with counters is supported.

If a shader attempts to access the counter of a UAV that does not have an associated counter, then the debug layer will issue a warning, and a GPU page fault will occur causing the apps's device to be removed.

UAV counters are supported in all heap types (default, upload, readback).

Queries

In Direct3D 12, queries are grouped into arrays of queries called a query heap. A query heap has a type which defines the valid types of queries that can be used with that heap.

Differences in Queries from Direct3D 11 to Direct3D 12

The following query types are no longer present in Direct3D 12, their functionality being incorporated into other processes:

- **Event queries** - event functionally is now handled by fences.
- **Disjoint timestamp queries** - GPU clocks can be set to a stable state in Direct3D 12 (see the **Timing** section). GPU clock comparisons are not meaningful if the GPU idled at all between the timestamps

(known as a disjoint query). With stable power two timestamp queries issued from different command lists are reliably comparable. Two timestamps within the same command list are always reliably comparable.
- **Steam output statistics queries** - in Direct3D 12 there is no single stream output (SO) overflow query for all the output streams. Apps need to issue multiple single-stream queries, and then correlate the results.
- **Stream output statistics predicate and occlusion predicate queries** - queries (which write to memory) and Predication (which reads from memory) are no longer coupled, and so these query types are not needed.

A new binary occlusion query type has been added to Direct3D 12.

Query Heaps

Queries can be one from a number of types (D3D12_QUERY_HEAP_TYPE), and are grouped into query heaps before being submitted to the GPU.

A new query type D3D12_QUERY_TYPE_BINARY_OCCLUSION is available and acts like D3D12_QUERY_TYPE_OCCLUSION except that it returns a binary 0/1 result: 0 indicates that no samples passed depth and stencil testing, 1 indicates that at least one sample passed depth and stencil testing. This enables occlusion queries to not interfere with any GPU performance optimization associated with depth/stencil testing.

Creating Query heaps

The APIs relevant to creating query heaps are the enum D3D12_QUERY_HEAP_TYPE, the struct D3D12_QUERY_HEAP_DESC, and the methodCreateQueryHeap.

The core runtime will validate that the query heap type is a valid member of the D3D12_HEAP_TYPE enumeration, and that the count is greater than 0.

Each individual query element within a query heap can be started and stopped separately.

The APIs for using the query heaps are the enum D3D12_QUERY_TYPE, and the methods BeginQuery and EndQuery.

D3D12_QUERY_TYPE_TIMESTAMP is the only query that supports EndQuery only. All other query types require BeginQuery and **EndQuery**.

The debug layer will validate the following:

- It is illegal to begin a query twice without ending it (for a given element). For queries which require both begin and end, it is illegal to end a query before the corresponding begin (for a given element).
- The query type passed to BeginQuery must match the query type passed to EndQuery.

The core runtime will validate the following:

- BeginQuery cannot be called on a timestamp query.
- For the query types which support both BeginQuery and EndQuery (all except for timestamp), a query for a given element must not span command list boundaries.
- *ElementIndex* must be within range.
- The query type is a valid member of the D3D12_QUERY_TYPE enum.

- The query type must be compatible with the query heap. The following table shows the query heap type required for each query type:

Query Type	Query Heap type
D3D12_QUERY_TYPE_OCCLUSION	D3D12_QUERY_TYPE_HEAP_TYPE_OCCLUSION
D3D12_QUERY_TYPE_BINARY_OCCLUSION	D3D12_QUERY_TYPE_HEAP_TYPE_OCCLUSION
D3D12_QUERY_TYPE_TIMESTAMP	D3D12_QUERY_TYPE_HEAP_TYPE_TIMESTAMP
D3D12_QUERY_TYPE_PIPELINE_STATISTICS	D3D12_QUERY_TYPE_HEAP_TYPE_PIPELINE_STATISTICS
D3D12_QUERY_TYPE_SO_STATISTICS_STREAM0	D3D12_QUERY_TYPE_HEAP_TYPE_SO_STATISTICS
D3D12_QUERY_TYPE_SO_STATISTICS_STREAM1	D3D12_QUERY_TYPE_HEAP_TYPE_SO_STATISTICS
D3D12_QUERY_TYPE_SO_STATISTICS_STREAM2	D3D12_QUERY_TYPE_HEAP_TYPE_SO_STATISTICS
D3D12_QUERY_TYPE_SO_STATISTICS_STREAM3	D3D12_QUERY_TYPE_HEAP_TYPE_SO_STATISTICS

-

- The query type is supported by the command list type. The following table shows which queries are supported on which command list types.

Query Type	Supported Command List Types
D3D12_QUERY_TYPE_OCCLUSION	Direct
D3D12_QUERY_TYPE_BINARY_OCCLUSION	Direct
D3D12_QUERY_TYPE_TIMESTAMP	Direct and Compute
D3D12_QUERY_TYPE_PIPELINE_STATISTICS	Direct
D3D12_QUERY_TYPE_SO_STATISTICS_STREAM0	Direct
D3D12_QUERY_TYPE_SO_STATISTICS_STREAM1	Direct
D3D12_QUERY_TYPE_SO_STATISTICS_STREAM2	Direct
D3D12_QUERY_TYPE_SO_STATISTICS_STREAM3	Direct

-

Extracting data from a query

The way to extract data from a query is to use the ResolveQueryData method. **ResolveQueryData** works with all heap types (default, upload, and readback).

Timing

This section covers timestamps, and calibrating the GPU and CPU clocks.

- Timestamp Frequency
- Clock Calibration
- SetStablePowerState
- Related topics

Timestamp Frequency

Applications can query the GPU timestamp clock frequency on a per-command queue basis (refer to the ID3D12CommandQueue::GetTimestampFrequencymethod).

The returned frequency is measured in Hz (ticks/sec). This API fails (and returns E_FAIL) if the specified command queue does not support timestamps (see the table in the Queries section).

Clock Calibration

D3D12 enables applications to correlate results obtained from timestamp queries with results obtained from calling `QueryPerformanceCounter`. This is enabled by the call ID3D12CommandQueue::GetClockCalibration.

GetClockCalibration samples the GPU clock for a given command queue and samples the CPU clock via `QueryPerformanceCounter` at nearly the same time. Again this API fails (returning E_FAIL) if the specified command queue does not support timestamps (see the table in the Queries section).

SetStablePowerState

In order for the clock calibration to be useful the application must be confident that the GPU timestamp clock will not stop ticking during idle periods. This is enabled by the method ID3D12Device::SetStablePowerState. This API is intended for development time use only. Therefore it is only allowed when the D3D12 SDK layers are present on the machine. The API fails with E_FAIL if the D3D12 SDK layers are not present.

The debug layer will issue a warning if the GetClockCalibration API is used without SetStablePowerState being called first.

This method is also useful in its own right for applications to ensure that GPU throttling (the scaling back of the GPU clock speed to meet an apparent lower demand) does not interfere with performance measurement.

Predication

Predication is a feature that enables the GPU rather than the CPU to determine to not draw, copy or dispatch an object.

- Overview
- SetPredication
- Related topics

Overview

The typical use of predication is with occlusion; if a bounding box is drawn and is occluded, there is obviously no point in drawing the object itself. In this situation the drawing of the object can be "predicated" , enabling its removal from actual rendering by the GPU.

Unlike Direct3D 11, predication is decoupled from queries, and is expanded in Direct3D 12 to enable an application to predicate objects based on any reasoning the app developer may decide on (not just occlusion).

SetPredication

Predication can be set based on the value of 64-bits within a buffer (refer to D3D12_PREDICATION_OP).

When the GPU executes a SetPredication command it snaps the value in the buffer. Future changes to the data in the buffer do not retroactively affect the predication state.

If the input parameter Buffer is NULL, then predication is disabled.

Predication hints are not present in the D3D12 API, and predication is allowed on direct and compute command lists. The source buffer can be in any heap type (default, upload, readback).

The core runtime will validate the following:

- *AlignedBufferOffset* is a multiple of 8 bytes
- The resource is a buffer
- The operation is a valid member of the enumeration
- SetPredication cannot be called from within a bundle
- The command list type supports predication

- The offset does not exceed the buffer size

The debug layer will issue an error if the source buffer is not in the D3D12_RESOURCE_STATE_VERTEX_AND_CONSTANT_BUFFER state.

The set of operations which can be predicated are:

- DrawInstanced
- DrawIndexedInstanced
- Dispatch
- CopyTextureRegion
- CopyBufferRegion
- CopyResource
- CopyTiles
- ResolveSubresource
- ClearDepthStencilView
- ClearRenderTargetView
- ClearUnorderedAccessViewUint
- ClearUnorderedAccessViewFloat
- ExecuteIndirect

ExecuteBundle is not predicated itself. Instead, individual operations from the list above which are contained in side of the bundle are predicated.

The ID3D12GraphicsCommandList methods ResolveQueryData, BeginQuery and EndQuery are not predicated.

Direct3D 12 Interop

D3D12 can interop both with existing code written using other Windows graphics APIs, as well as with new code written using D3D12 which implements a portion of a graphics engine.

D3D12 can be very powerful, and allow applications to write graphics code with console-like efficiency, but not every application needs to reinvent the wheel and write the entirety of their rendering engine from scratch. In some cases, another component or library has already done it better, or in other cases, the performance of a portion of code is not as critical as its correctness and readability.

This section covers the following interop techniques:

- D3D12 and D3D12, on the same device
- D3D12 and D3D12, on different devices
- D3D12 and any combination of D3D11, D3D10, or D2D, on the same device
- D3D12 and any combination of D3D11, D3D10, or D2D, on different devices
- D3D12 and GDI, or D3D12 and D3D11 and GDI

In this section

Topic	Description
Interop Overview	D3D12 can be used to write componentized applications.
Direct3D 11 on 12	D3D11On12 is a mechanism by which developers can use D3D11 interfaces and objects to drive the D3D12 API. D3D11on12 enables components

	written using D3D11 (for example, D2D text and UI) to work together with components written targeting the D3D12 API. D3D11on12 also enables incremental porting of an application from D3D11 to D3D12, by enabling portions of the app to continue targeting D3D11 for simplicity while others target D3D12 for performance, while always having complete and correct rendering. D3D11On12 makes it simpler than using interop techniques to share resources and synchronize work between the two APIs.

Interop Overview

D3D12 can be used to write componentized applications.

Reasons for using interop

There are several reasons an application would want D3D12 interop with other APIs. Some examples:

- Incremental porting: wanting to port an entire application from D3D10 or D3D11 to D3D12, while having it functional at intermediate stages of the porting process (to enable testing and debugging).
- Black box code: wanting to leave a particular portion of an application as-is while porting the rest of the code. For example, there might be no need to port UI elements of a game.
- Unchangeable components: needing to use components which are not owned by the application, which are not written to target D3D12.
- A new component: not wanting to port the entire application, but wanting to use a new component which is written using D3D12.

There are four main techniques for interop in D3D12:

- An app can choose to provide an open command list to a component, which records some additional rendering commands to an already-bound render target. This is equivalent to providing a prepared device context to another component in D3D11, and is great for things like adding UI/text to an already bound back buffer.
- An app can choose to provide a command queue to a component, along with a desired destination resource. This is equivalent to using ClearState orDeviceContextState APIs in D3D11 to provide a clean device context to another component. This is how components like D2D operate.
- A component may opt for a model where it produces a command list, potentially in parallel, which the app is responsible for submission at a later time. At least one resource must be provided across component boundaries. This same technique is available in D3D11 using deferred contexts, though the performance in D3D12 is more desirable.

- Each component has its own queue(s) and/or device(s), and the app and components need to share resources and synchronization information across component boundaries. This is similar to the legacy `ISurfaceQueue`, and the more modern **IDXGIKeyedMutex**.

The differences between these scenarios is what exactly is shared between the component boundaries. The device is assumed to be shared, but since it is basically stateless, it is not really relevant. The key objects are the command list, the command queue, the sync objects, and the resources. Each of these have their own complications when sharing them.

Sharing a command list

The simplest method of interop requires sharing only a command list with a portion of the engine. Once the rendering operations have completed, the command list ownership goes back to the caller. The ownership of the command list can be traced through the stack. Since command lists are single threaded, there's no way for an app to do something unique or innovative using this technique.

Sharing a command queue

Probably the most common technique for multiple components sharing a device in the same process.

When the command queue is the unit of sharing, there needs to be a call to the component to let it know that all outstanding command lists need to be submitted to the command queue immediately (and any internal command queues need to be synchronized). This is equivalent to the D3D11 **Flush** API, and is the only way that the application can submit its own command lists or sync primitives.

Sharing sync primitives

The expected pattern for a component which operates on its own devices and/or command queues will be to accept an **ID3D12Fence** or shared handle, and UINT64 pair upon beginning its work, which it will wait on, and then a second ID3D12Fence or shared handle, and UINT64 pair which it will signal when all work is complete. This pattern matches the current implementation of both **IDXGIKeyedMutex** and the DWM/DXGI flip model synchronization design.

Sharing resources

By far the most complicated part of writing a D3D12 app which leverages multiple components is how to deal with the resources which are shared across component boundaries. This is mostly due to the concept of resource states. While some aspects of the resource state design are meant to deal with intra-command-list synchronization, others do have impact between command lists, affecting resource layout and either valid sets of operations or performance characteristics of accessing the resource data.

There are two patterns of dealing with this complication, both of which involve essentially a contract between components.

- The contract can be defined by the component developer and documented. This could be as simple as "the resource must be in the default state when work is started, and will be put back in the default state when work is done" or could have more complicated rules to allow things like sharing a depth buffer without forcing intermediate depth resolves.
- The contract can be defined by the application at runtime, at the time when the resource is shared across component boundaries. It consists of the same two pieces of information – the state the resource will be in when the component starts using it, and the state the component should leave it in when it finishes.

Choosing an interop model

For most D3D12 applications, sharing a command queue is probably the ideal model. It allows complete ownership of work creation and submission, without the additional memory overhead from having redundant queues, and without the perf impact of dealing with the GPU sync primitives.

Sharing sync primitives is required once the components need to deal with different queue properties, such as type or priority, or once the sharing needs to span process boundaries.

Sharing or producing command lists are not widely used externally by third party components, but might be widely used in components which are internal to a game engine.

Interop APIs

Interop is a work in progress.

Direct3D 11 on 12

D3D11On12 is a mechanism by which developers can use D3D11 interfaces and objects to drive the D3D12 API. D3D11on12 enables components written using D3D11 (for example, D2D text and UI) to work together with components written targeting the D3D12 API. D3D11on12 also enables incremental porting of an application from D3D11 to D3D12, by enabling portions of the app to continue targeting D3D11 for simplicity while others target D3D12 for performance, while always having complete and correct rendering. D3D11On12 makes it simpler than using interop techniques to share resources and synchronize work between the two APIs.

- **Initializing D3D11On12**
- **Example Usage**
- **Background**
- **Cleaning up**
- **Limitations**
- **APIs**
- **Related topics**

Initializing D3D11On12

To begin using D3D11On12, the first step is to create a D3D12 device and command queue. These objects are provided as input to the initialization method**D3D11On12CreateDevice**. You can think of this method as creating a D3D11 device with the imaginary driver type D3D_DRIVER_TYPE_11ON12, where the D3D11 driver is responsible for creating objects and submitting command lists to the D3D12 API.

After you have a D3D11 device and immediate context, you can `QueryInterface` off of the device for the **ID3D11On12Device** interface. This is the primary interface that is used for interop between D3D11 and D3D12. In order to have both the D3D11 device context and the D3D12 command lists operate on the same resources, it is necessary to create “wrapped resources” using the **CreateWrappedResource** API. This method “promotes” a D3D12 resource to be understandable in D3D11. A wrapped resource starts out in the “acquired” state, a property which is manipulated by the **AcquireWrappedResources** and**ReleaseWrappedResources** methods.

Example Usage

Typical usage of D3D11On12 would be to use D2D to render text or images on top of a D3D12 back buffer. See the D3D11On12 sample for example code. Here is a rough outline of the steps to take to do so:

- Create a D3D12 device (**D3D12CreateDevice**) and a D3D12 swap chain (**CreateSwapChain** with an **ID3D12CommandQueue** as an input).
- Create a D3D11On12 device using the D3D12 device and the same command queue as input.
- Retrieve the swap chain back buffers, and create D3D11 wrapped resources for each of them. The input state used should be the last way that D3D12 used it (e.g. RENDER_TARGET) and the output state should be the way that D3D12 will use it after D3D11 has finished (e.g. PRESENT).
- Initialize D2D, and provide the D3D11 wrapped resources to D2D to prepare for rendering.

Then, on each frame, do the following:

- Render into the current swap chain back buffer using a D3D12 command list, and execute it.
- Acquire the current back buffer's wrapped resource (**AcquireWrappedResources**).
- Issue D2D rendering commands.
- Release the wrapped resource (**ReleaseWrappedResources**).
- Flush the D3D11 immediate context.
- Present.

Background

D3D11On12 works systematically. Each D3D11 API call goes through the typical runtime validation and makes its way to the driver. At the driver layer, the special 11on12 driver records state and issues render operations to D3D12 command lists. These command lists are submitted as necessary (for example, a query `GetData` or resource `Map` might require commands to be flushed) or as requested by Flush. Creating a D3D11 object typically results in the corresponding D3D12 object being created. Some fixed function render operations in D3D11 such as `GenerateMips` or `DrawAuto` are not supported in D3D12, and so D3D11On12 emulates them using shaders and additional resources.

For interop, it's important to understand how D3D11On12 interacts with the D3D12 objects that the app has created and provided. In order to ensure that work happens in the correct order, the D3D11 immediate context must be flushed before additional D3D12 work can be submitted to that queue. It's also important to ensure that the queue given to D3D11On12 must be drainable at all times. That means that any waits on the queue must eventually be satisfied, even if the D3D11 render thread blocks indefinitely. Be wary not to take a dependency on when D3D11On12 inserts flushes or waits, as this may change with future releases. Additionally, D3D11On12 tracks and manipulates resource states on its own. The only way to ensure coherency of state transitions is to make use of the acquire/release APIs to manipulate the state tracking to match the app's needs.

Cleaning up

In order to release a D3D11On12 wrapped resource, two things need to happen in this order:

- All references to the resource, including any views of the resource, need to be released.
- Deferred destruction processing must take place. The simplest way to ensure this happens is to invoke the immediate context `Flush` API.

After both of those steps are completed, any references taken by the wrapped resource should be released, and the D3D12 resource becomes exclusively owned by the D3D12 component. Be aware that D3D12 still requires waiting for GPU completion before completely releasing a resource, so be sure to hold a reference on the resource before doing the two steps above, unless you've already confirmed that the GPU is no longer using the resource.

All other resources or objects created by D3D11On12 will be cleaned up at the appropriate time, when the GPU has finished using them, using D3D11's deferred destruction mechanism. However if you attempt to release the D3D11On12 device itself while the GPU is still executing, the destruction may block until the GPU completes.

Limitations

The D3D11On12 layer implements a very large subset of the D3D11 API, but there are some known gaps. In addition to bugs in the implementation which can cause incorrect rendering, the shader interfaces feature is currently unimplemented in D3D11On12. Attempting to use this feature will cause errors and debug messages. Additionally, swap chains are not currently supported on D3D11On12 devices.

D3D11On12 has not currently been optimized for performance. There will likely be moderate CPU overhead compared to a standard D3D11 driver, minimal GPU overhead, and there is known to be significant memory overhead. Therefore it is not recommended to use D3D11On12 for complicated 3D scenes, and it is instead recommended for simple scenes or 2D rendering.

APIs

APIs that make up the 11on12 layer are described in 11on12 Reference.

Working Samples

Working samples are available for download, showing the usage of a number of features of Direct3D 12.

Working samples

Working samples (in the form of Visual Studio 2015 projects) can be downloaded from DirectX-Graphics-Samples.

Note The exact list of samples available at this location will vary as samples are added and updated.

Sample title	Description
HelloWorld **HelloWindow** **HelloTriangle** **HelloBundles** **HelloConstBuffers** **HelloTexture**	The HelloWorld sample set contains the following simple projects to help you get started with Direct3D 12. Creates a window in preparation of rendering Direct3D 12 content. Renders a simple triangle using Direct3D 12. Demonstrates the usage of a bundle for rendering using Direct3D 12. Demonstrates how to use constant buffers to pass data to the GPU used for rendering in Direct3D 12. Demonstrates how to apply a texture to a triangle using Direct3D 12.
D3D12Bundles	Demonstrates frame buffering and synchronization best practices as well as rendering a simple mesh using bundles.
D3D12Multithreading	An example of how to build a multithreaded capable application.
D3D12nBodyGravity	Demonstrates how multi-engine can be used to do asynchronous compute work alongside 3D work on the same GPU.
D3D12PredicationQueries	Demonstrates occlusion culling using query heaps and predication.

D3D12DynamicIndexing	Demonstrates the dynamic indexing capabilities of DirectX 12 and HLSL.
D3D1211on12	Demonstrates basic usage of the 11on12 layer. This sample renders text using D2D using the Direct3D 11 API on a Direct3D 12 11on12 device.
D3D12ExecuteIndirect	Demonstrates compute engine culling in conjunction with the execute indirect feature to only render objects that pass the culling test.
D3D12PipelineStateCache	Demonstrates Pipeline State Object (PSO) caching.

D3D12 Code Walk-Throughs

This section provides code for sample scenarios. Many of the walk-throughs provide details on what coding is required to be added to a basic sample, to avoid repeating the basic component code for each scenario.

For the most basic component, refer to the Creating a Basic Direct3D 12 Component section. The following walk-throughs describe more advanced scenarios.

In this section

Topic	Description
D2D using D3D11on12	The **D3D1211on12** sample demonstrates how to render D2D content over D3D12 content by sharing resources between an 11 based device and a 12 based device.
Multi-engine n-body gravity simulation	The **D3D12nBodyGravity** sample demonstrates how to do compute work asynchronously. The sample spins up a number of threads each with a compute command queue and schedules compute work on the GPU that performs an n-body gravity simulation. Each thread operates on two buffers full of position and velocity data. With each iteration, the compute shader reads the current position and velocity data from one buffer and writes the next iteration into the other buffer. When the iteration completes, the compute shader swaps which buffer is the SRV for reading position/velocity data and which is the UAV for writing position/velocity updates by changing the resource state on each buffer.
Predication queries	The **D3D12PredicationQueries** sample demonstrates occlusion culling using DirectX 12 query heaps and predication. The walkthrough describes the additional code needed to extend the **HelloConstBuffer** sample to handle predication queries.
Dynamic Indexing using HLSL 5.1	The **D3D12DynamicIndexing** sample demonstrates some of the new HLSL features available in Shader Model 5.1 - particularly dynamic indexing and unbounded arrays - to render the same mesh multiple times, each time rendering it with a dynamically selected material. With dynamic indexing, shaders can now index

	into an array without knowing the value of the index at compile time. When combined with unbounded arrays, this adds another level of indirection and flexibility for shader authors and art pipelines.

D2D using D3D11on12

The **D3D1211on12** sample demonstrates how to render D2D content over D3D12 content by sharing resources between an 11 based device and a 12 based device.

- Create an ID3D11On12Device
- Create a D2D factory
- Create a render target for D2D
- Create basic D2D text objects
- Updating the main render loop
- Run the sample
- Related topics

Create an ID3D11On12Device

The first step is to create an ID3D11On12Device after the ID3D12Device has been created, which involves creating an ID3D11Device that is wrapped around the **ID3D12Device** via the API D3D11On12CreateDevice. This API also takes in, among other parameters, an ID3D12CommandQueue so that the 11On12 device can submit its commands. After the **ID3D11Device** is created, you can query the **ID3D11On12Device** interface from it. This is the primary device object that will be used to set up D2D.

In the **LoadPipeline** method, setup the devices.

```
// Create an 11 device wrapped around the 12 device and share
// 12's command queue.
ComPtr<ID3D11Device> d3d11Device;
ThrowIfFailed(D3D11On12CreateDevice(
    m_d3d12Device.Get(),
    d3d11DeviceFlags,
    nullptr,
    0,
    reinterpret_cast<IUnknown**>(m_commandQueue.GetAddressOf()),
    1,
    0,
    &d3d11Device,
    &m_d3d11DeviceContext,
    nullptr
));

// Query the 11On12 device from the 11 device.
ThrowIfFailed(d3d11Device.As(&m_d3d11On12Device));
```

Create a D2D factory

Now that we have an 11On12 device, we use it to create a D2D factory and device just as it would normally be done with D3D11.

Add to the **LoadAssets** method.

```
// Create D2D/DWrite components.
{
    D2D1_DEVICE_CONTEXT_OPTIONS deviceOptions = D2D1_DEVICE_CONTEXT_OPTIONS_NONE;
    ThrowIfFailed(D2D1CreateFactory(D2D1_FACTORY_TYPE_SINGLE_THREADED,
                                    __uuidof(ID2D1Factory3),
                                    &d2dFactoryOptions,
                                     &m_d2dFactory));
    ComPtr<IDXGIDevice> dxgiDevice;
    ThrowIfFailed(m_d3d11On12Device.As(&dxgiDevice));
    ThrowIfFailed(m_d2dFactory->CreateDevice(dxgiDevice.Get(), &m_d2dDevice));
    ThrowIfFailed(m_d2dDevice->CreateDeviceContext(deviceOptions, &m_d2dDeviceContext));
    ThrowIfFailed(DWriteCreateFactory(DWRITE_FACTORY_TYPE_SHARED, __uuidof(IDWriteFactory),
                                      &m_dWriteFactory));
}
```

Create a render target for D2D

D3D12 owns the swap chain, so if we want to render to the back buffer using our 11On12 device (D2D content), then we need to create wrapped resources of type ID3D11Resource from the back buffers of type ID3D12Resource. This links the **ID3D12Resource** with a D3D11 based interface so that it can be used with the 11On12 device. After we have a wrapped resource, we can then create a render target surface for D2D to render to, also in the **LoadAssets** method.

```
// Query the desktop's dpi settings, which will be used to create
// D2D's render targets.
float dpiX;
float dpiY;
m_d2dFactory->GetDesktopDpi(&dpiX, &dpiY);
D2D1_BITMAP_PROPERTIES1 bitmapProperties = D2D1::BitmapProperties1(
    D2D1_BITMAP_OPTIONS_TARGET | D2D1_BITMAP_OPTIONS_CANNOT_DRAW,
    D2D1::PixelFormat(DXGI_FORMAT_UNKNOWN, D2D1_ALPHA_MODE_PREMULTIPLIED),
    dpiX,
    dpiY
);

// Create frame resources.
{
    CD3DX12_CPU_DESCRIPTOR_HANDLE rtvHandle(m_rtvHeap->GetCPUDescriptorHandleForHeapStart());

    // Create a RTV, D2D render target, and a command allocator for each frame.
    for (UINT n = 0; n < FrameCount; n++) {
        ThrowIfFailed(m_swapChain->GetBuffer(n, IID_PPV_ARGS(&m_renderTargets[n])));
        m_d3d12Device->CreateRenderTargetView(m_renderTargets[n].Get(), nullptr, rtvHandle);

        // Create a wrapped 11On12 resource of this back buffer. Since we are
        // rendering all D3D12 content first and then all D2D content, we specify
        // the In resource state as RENDER_TARGET - because D3D12 will have last
        // used it in this state - and the Out resource state as PRESENT. When
        // ReleaseWrappedResources() is called on the 11On12 device, the resource
        // will be transitioned to the PRESENT state.
        D3D11_RESOURCE_FLAGS d3d11Flags = { D3D11_BIND_RENDER_TARGET };
        ThrowIfFailed(m_d3d11On12Device->CreateWrappedResource(
            m_renderTargets[n].Get(),
            &d3d11Flags,
            D3D12_RESOURCE_STATE_RENDER_TARGET,
            D3D12_RESOURCE_STATE_PRESENT,
            IID_PPV_ARGS(&m_wrappedBackBuffers[n])
        ));

        // Create a render target for D2D to draw directly to this back buffer.
        ComPtr<IDXGISurface> surface;
        ThrowIfFailed(m_wrappedBackBuffers[n].As(&surface));
        ThrowIfFailed(m_d2dDeviceContext->CreateBitmapFromDxgiSurface(
            surface.Get(),
            &bitmapProperties,
            &m_d2dRenderTargets[n]
        ));

        rtvHandle.Offset(1, m_rtvDescriptorSize);

        ThrowIfFailed(m_d3d12Device->CreateCommandAllocator(D3D12_COMMAND_LIST_TYPE_DIRECT,
                                                               IID_PPV_ARGS(&m_commandAllocators[n])));
        }
    }
```

Create basic D2D text objects

Now we have an ID3D12Device to render 3D content, an ID2D1Device that is shared with our 12 device via an ID3D11On12Device - which we can use to render 2D content - and they are both configured to render to the same swap chain. This sample simply uses the D2D device to render text over the 3D scene, similar to how games render their UI. For that, we need to create some basic D2D objects, still in the **LoadAssets** method.

```
// Create D2D/DWrite objects for rendering text.
{
    ThrowIfFailed(m_d2dDeviceContext->CreateSolidColorBrush(D2D1::ColorF(D2D1::ColorF::Black),
                   &m_textBrush));
    ThrowIfFailed(m_dWriteFactory->CreateTextFormat(
        L"Verdana",
        nullptr,
        DWRITE_FONT_WEIGHT_NORMAL,
        DWRITE_FONT_STYLE_NORMAL,
        DWRITE_FONT_STRETCH_NORMAL,
        50,
        L"en-us",
        &m_textFormat
    ));
    ThrowIfFailed(m_textFormat->SetTextAlignment(DWRITE_TEXT_ALIGNMENT_CENTER));
    ThrowIfFailed(m_textFormat->SetParagraphAlignment(DWRITE_PARAGRAPH_ALIGNMENT_CENTER));
}
```

Updating the main render loop

Now that the initialization of the sample is complete, we can move on to the main render loop.

```
// Render the scene.
void D3D1211on12::OnRender() {
    // Record all the commands we need to render the scene into the command list.
    PopulateCommandList();

    // Execute the command list.
    ID3D12CommandList* ppCommandLists[] = { m_commandList.Get() };
    m_commandQueue->ExecuteCommandLists(_countof(ppCommandLists), ppCommandLists);

    RenderUI();

    // Present the frame.
    ThrowIfFailed(m_swapChain->Present(0, 0));

    MoveToNextFrame();
}
```

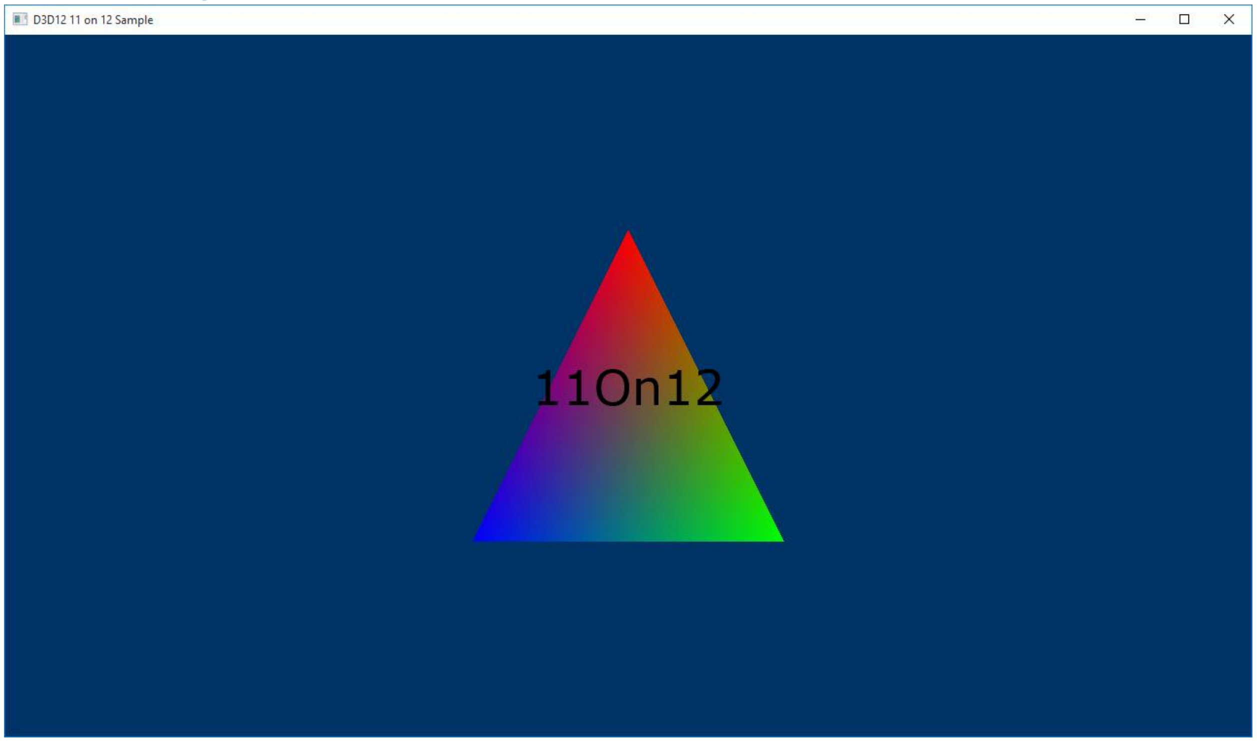
The only thing new to our render loop is the **RenderUI** call, which will use D2D to render our UI. Notice that we execute all of our D3D12 command lists first to render our 3D scene, and then we render our UI on top of that. Before we dive into **RenderUI**, we must look at a change to **PopulateCommandLists**. In other samples we commonly put a resource barrier on the command list prior to closing it to transition the back buffer from the render target state to the present state. However, in this sample we remove that resource barrier, because we still need to render to the back buffers with D2D. Note that when we created our wrapped resources of the back buffer that we specified the render target state as the “IN” state and the present state as the “OUT” state.

RenderUI is fairly straight-forward in terms of D2D usage. We set our render target and render our text. However, before using any wrapped resources on an 11On12 device, such as our back buffer render targets, we must call the AcquireWrappedResources API on the 11On12 device. After rendering we call the

ReleaseWrappedResources API on the 11On12 device. By calling **ReleaseWrappedResources** we incur a resource barrier behind the scenes that will transition the specified resource to the “OUT” state specified at creation time. In our case, this is the present state. Finally, in order to submit all of our commands performed on the 11On12 device to the shared ID3D12CommandQueue, we must call Flush on the ID3D11DeviceContext.

```
// Render text over D3D12 using D2D via the 11On12 device.
void D3D1211on12::RenderUI() {
    D2D1_SIZE_F rtSize = m_d2dRenderTargets[m_frameIndex]->GetSize();
    D2D1_RECT_F textRect = D2D1::RectF(0, 0, rtSize.width, rtSize.height);
    static const WCHAR text[] = L"11On12";

    // Acquire our wrapped render target resource for the current back buffer.
    m_d3d11On12Device->AcquireWrappedResources(m_wrappedBackBuffers[m_frameIndex].GetAddressOf(), 1);

    // Render text directly to the back buffer.
    m_d2dDeviceContext->SetTarget(m_d2dRenderTargets[m_frameIndex].Get());
    m_d2dDeviceContext->BeginDraw();
    m_d2dDeviceContext->SetTransform(D2D1::Matrix3x2F::Identity());
    m_d2dDeviceContext->DrawTextW(
        text,
        _countof(text) - 1,
        m_textFormat.Get(),
        &textRect,
        m_textBrush.Get()
        );
    ThrowIfFailed(m_d2dDeviceContext->EndDraw());

    // Release our wrapped render target resource. Releasing
    // transitions the back buffer resource to the state specified
    // as the OutState when the wrapped resource was created.
    m_d3d11On12Device->ReleaseWrappedResources(m_wrappedBackBuffers[m_frameIndex].GetAddressOf(), 1);

    // Flush to submit the 11 command list to the shared command queue.
    m_d3d11DeviceContext->Flush();
}
```

Run the sample



Multi-engine n-body gravity simulation

The **D3D12nBodyGravity** sample demonstrates how to do compute work asynchronously. The sample spins up a number of threads each with a compute command queue and schedules compute work on the GPU that performs an n-body gravity simulation. Each thread operates on two buffers full of position and velocity data. With each iteration, the compute shader reads the current position and velocity data from one buffer and writes the next iteration into the other buffer. When the iteration completes, the compute shader swaps which buffer is the SRV for reading position/velocity data and which is the UAV for writing position/velocity updates by changing the resource state on each buffer.

Create the root signatures

We start out by creating both a graphics and a compute root signature, in the **LoadAssets** method. Both root signatures have a root constant buffer view (CBV) and a shader resource view (SRV) descriptor table. The compute root signature also has an unordered access view (UAV) descriptor table.

```
// Create the root signatures.
{
    CD3DX12_DESCRIPTOR_RANGE ranges[2];
    ranges[0].Init(D3D12_DESCRIPTOR_RANGE_TYPE_SRV, 1, 0);
    ranges[1].Init(D3D12_DESCRIPTOR_RANGE_TYPE_UAV, 1, 0);

    CD3DX12_ROOT_PARAMETER rootParameters[RootParametersCount];
    rootParameters[RootParameterCB].InitAsConstantBufferView(0, 0, D3D12_SHADER_VISIBILITY_ALL);
    rootParameters[RootParameterSRV].InitAsDescriptorTable(1, &ranges[0], D3D12_SHADER_VISIBILITY_VERTEX);
    rootParameters[RootParameterUAV].InitAsDescriptorTable(1, &ranges[1], D3D12_SHADER_VISIBILITY_ALL);

    // The rendering pipeline does not need the UAV parameter.
    CD3DX12_ROOT_SIGNATURE_DESC rootSignatureDesc;
    rootSignatureDesc.Init(_countof(rootParameters) - 1,
                           rootParameters,
                           0,
                           nullptr,
                           D3D12_ROOT_SIGNATURE_FLAG_ALLOW_INPUT_ASSEMBLER_INPUT_LAYOUT);

    ComPtr<ID3DBlob> signature;
    ComPtr<ID3DBlob> error;
    ThrowIfFailed(D3D12SerializeRootSignature(&rootSignatureDesc, D3D_ROOT_SIGNATURE_VERSION_1, &signature,
                                              &error));
    ThrowIfFailed(m_device->CreateRootSignature(0,
                                                signature->GetBufferPointer(),
                                                signature->GetBufferSize(),
                                                IID_PPV_ARGS(&m_rootSignature)));

    // Create compute signature. Must change visibility for the SRV.
    rootParameters[RootParameterSRV].ShaderVisibility = D3D12_SHADER_VISIBILITY_ALL;

    CD3DX12_ROOT_SIGNATURE_DESC computeRootSignatureDesc(_countof(rootParameters),
                                                         rootParameters,
                                                         0,
                                                         nullptr);

    ThrowIfFailed(D3D12SerializeRootSignature(&computeRootSignatureDesc, D3D_ROOT_SIGNATURE_VERSION_1,
                                              &signature, &error));
    ThrowIfFailed(m_device->CreateRootSignature(0, signature->GetBufferPointer(), signature->GetBufferSize(),
                  IID_PPV_ARGS(&m_computeRootSignature)));
}
```

Create the SRV and UAV buffers

The SRV and UAV buffers consist of an array of position and velocity data.

```
// Position and velocity data for the particles in the system.
// Two buffers full of Particle data are utilized in this sample.
// The compute thread alternates writing to each of them.
// The render thread renders using the buffer that is not currently
// in use by the compute shader.
struct Particle {
    XMFLOAT4 position;
    XMFLOAT4 velocity;
};
```

Create the CBV and vertex buffers

For the graphics pipeline, the CBV is a **struct** containing two matrices used by the geometry shader. The geometry shader takes the position of each particle in the system and generates a quad to represent it using these matrices.

```
struct ConstantBufferGS {
    XMMATRIX worldViewProjection;
    XMMATRIX inverseView;

    // Constant buffers are 256-byte aligned in GPU memory. Padding is added
    // for convenience when computing the struct's size.
    float padding[32];
};
```

As a result, the vertex buffer used by the vertex shader actually does not contain any positional data.

```
// "Vertex" definition for particles. Triangle vertices are generated
// by the geometry shader. Color data will be assigned to those
// vertices via this struct.
struct ParticleVertex {
    XMFLOAT4 color;
};
```

For the compute pipeline, the CBV is a **struct** containing some constants used by the n-body gravity simulation in the compute shader.

```
struct ConstantBufferCS {
    UINT param[4];
    float paramf[4];
};
```

Synchronize the rendering and compute threads

After the buffers are all initialized, the rendering and compute work will begin. The compute thread will be changing the state of the two position/velocity buffers back and forth between SRV and UAV as it iterates on the simulation, and the rendering thread needs to ensure that it schedules work on the graphics pipeline that operates on the SRV. Fences are used to synchronize access to the two buffers.

On the Render thread:

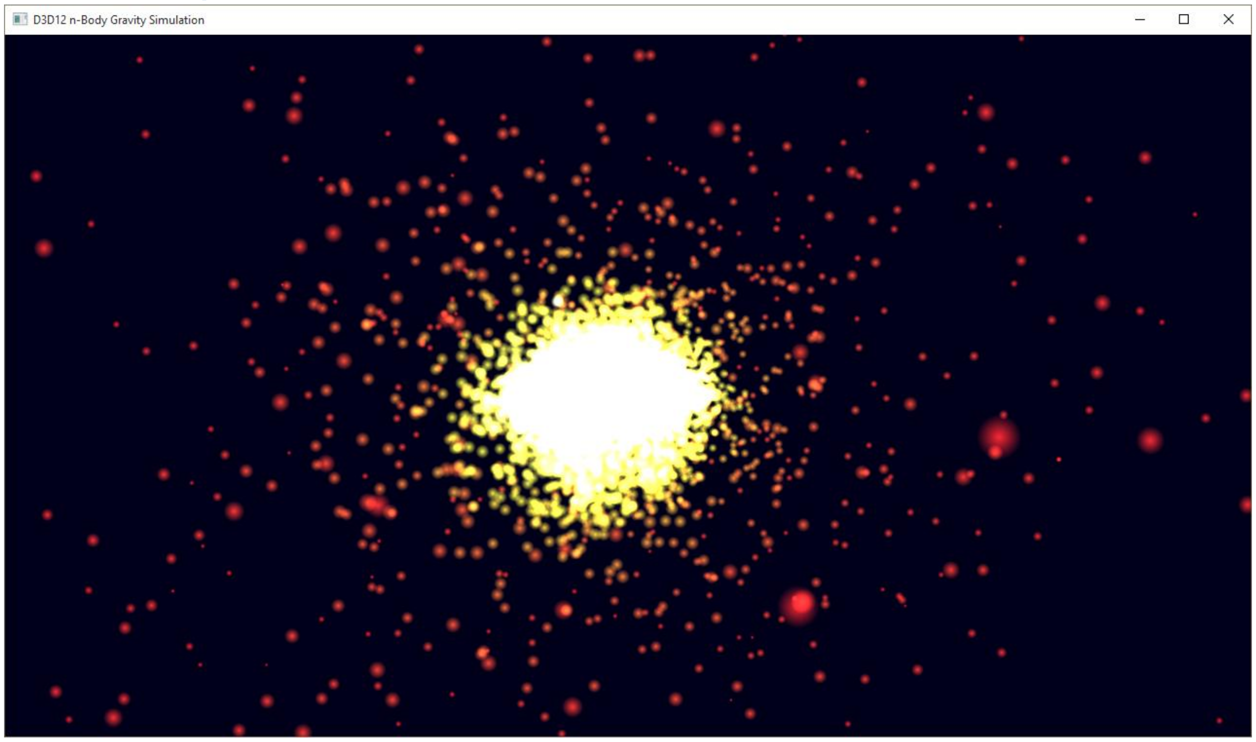
```
// Render the scene.
void D3D12nBodyGravity::OnRender() {
   // Let the compute thread know that a new frame is being rendered.
   for (int n = 0; n < ThreadCount; n++) {
       InterlockedExchange(&m_renderContextFenceValues[n], m_renderContextFenceValue);
   }

   // Compute work must be completed before the frame can render or else the SRV
   // will be in the wrong state.
   for (UINT n = 0; n < ThreadCount; n++) {
       UINT64 threadFenceValue = InterlockedGetValue(&m_threadFenceValues[n]);
       if (m_threadFences[n]->GetCompletedValue() < threadFenceValue) {
           // Instruct the rendering command queue to wait for the current
           // compute work to complete.
           ThrowIfFailed(m_commandQueue->Wait(m_threadFences[n].Get(), threadFenceValue));
       }
   }

   // Record all the commands we need to render the scene into the command list.
   PopulateCommandList();

   // Execute the command list.
   ID3D12CommandList* ppCommandLists[] = { m_commandList.Get() };
   m_commandQueue->ExecuteCommandLists(_countof(ppCommandLists), ppCommandLists);

   // Present the frame.
   ThrowIfFailed(m_swapChain->Present(0, 0));

   MoveToNextFrame();
}
```

To simplify the sample a bit, the compute thread waits for the GPU to complete each iteration before scheduling any more compute work. In practice, applications will likely want to keep the compute queue full to achieve maximum performance from the GPU.

On the Compute thread:

```
DWORD D3D12nBodyGravity::AsyncComputeThreadProc(int threadIndex) {
    ID3D12CommandQueue* pCommandQueue = m_computeCommandQueue[threadIndex].Get();
    ID3D12CommandAllocator* pCommandAllocator = m_computeAllocator[threadIndex].Get();
    ID3D12GraphicsCommandList* pCommandList = m_computeCommandList[threadIndex].Get();
    ID3D12Fence* pFence = m_threadFences[threadIndex].Get();

    while (0 == InterlockedGetValue(&m_terminating)) {
        // Run the particle simulation.
        Simulate(threadIndex);

        // Close and execute the command list.
        ThrowIfFailed(pCommandList->Close());
        ID3D12CommandList* ppCommandLists[] = { pCommandList };

        pCommandQueue->ExecuteCommandLists(1, ppCommandLists);

        // Wait for the compute shader to complete the simulation.
        UINT64 threadFenceValue = InterlockedIncrement(&m_threadFenceValues[threadIndex]);
        ThrowIfFailed(pCommandQueue->Signal(pFence, threadFenceValue));
        ThrowIfFailed(pFence->SetEventOnCompletion(threadFenceValue, m_threadFenceEvents[threadIndex]));
        WaitForSingleObject(m_threadFenceEvents[threadIndex], INFINITE);

        // Wait for the render thread to be done with the SRV so that
        // the next frame in the simulation can run.
        UINT64 renderContextFenceValue = InterlockedGetValue(&m_renderContextFenceValues[threadIndex]);
        if (m_renderContextFence->GetCompletedValue() < renderContextFenceValue) {
            ThrowIfFailed(pCommandQueue->Wait(m_renderContextFence.Get(), renderContextFenceValue));
            InterlockedExchange(&m_renderContextFenceValues[threadIndex], 0);
        }

        // Swap the indices to the SRV and UAV.
        m_srvIndex[threadIndex] = 1 - m_srvIndex[threadIndex];

        // Prepare for the next frame.
        ThrowIfFailed(pCommandAllocator->Reset());
        ThrowIfFailed(pCommandList->Reset(pCommandAllocator, m_computeState.Get()));
    }

    return 0;
}
```

Run the sample



Predication queries

The **D3D12PredicationQueries** sample demonstrates occlusion culling using DirectX 12 query heaps and predication. The walkthrough describes the additional code needed to extend the **HelloConstBuffer** sample to handle predication queries.

- Create a depth stencil descriptor heap and an occlusion query heap
- Enable alpha blending
- Disable color and depth writes
- Create a buffer to store the results of the query
- Draw the quads and perform and resolve the occlusion query
- Run the sample
- Related topics

Create a depth stencil descriptor heap and an occlusion query heap

In the **LoadPipeline** method create a depth stencil descriptor heap.

```
// Describe and create a depth stencil view (DSV) descriptor heap.
D3D12_DESCRIPTOR_HEAP_DESC dsvHeapDesc = {};
dsvHeapDesc.NumDescriptors = 1;
dsvHeapDesc.Type = D3D12_DESCRIPTOR_HEAP_TYPE_DSV;
dsvHeapDesc.Flags = D3D12_DESCRIPTOR_HEAP_FLAG_NONE;
ThrowIfFailed(m_device->CreateDescriptorHeap(&dsvHeapDesc, IID_PPV_ARGS(&m_dsvHeap)));
```

In the **LoadAssets** method create a heap for occlusion queries.

```
// Describe and create a heap for occlusion queries.
D3D12_QUERY_HEAP_DESC queryHeapDesc = {};
queryHeapDesc.Count = 1;
queryHeapDesc.Type = D3D12_QUERY_HEAP_TYPE_OCCLUSION;
ThrowIfFailed(m_device->CreateQueryHeap(&queryHeapDesc, IID_PPV_ARGS(&m_queryHeap)));
```

Enable alpha blending

This sample draws two quads and illustrates a binary occlusion query. The quad in front animates across the screen, and the one in back will occasionally be occluded. In the **LoadAssets** method, alpha blending is enabled for this sample so that we can see at what point D3D considers the quad in back occluded.

```
// Enable alpha blending so we can visualize the occlusion query results.
CD3DX12_BLEND_DESC blendDesc(D3D12_DEFAULT);
blendDesc.RenderTarget[0] = {
    true,
    false,
    D3D12_BLEND_SRC_ALPHA,
    D3D12_BLEND_INV_SRC_ALPHA,
    D3D12_BLEND_OP_ADD,
    D3D12_BLEND_ONE,
    D3D12_BLEND_ZERO,
    D3D12_BLEND_OP_ADD,
    D3D12_LOGIC_OP_NOOP,
    D3D12_COLOR_WRITE_ENABLE_ALL,
};
```

Disable color and depth writes

The occlusion query is performed by rendering a quad that covers the same area as the quad whose visibility we want to test. In more complex scenes, the query would likely be a bounding volume, rather than a simple quad. In either case, a new pipeline state is created that disables writing to the render target and the z buffer so that the occlusion query itself does not affect the visible output of the rendering pass.

In the **LoadAssets** method, disable color writes and depth writes for the occlusion query's state.

```
// Disable color writes and depth writes for the occlusion query's state.
psoDesc.BlendState.RenderTarget[0].RenderTargetWriteMask = 0;
psoDesc.DepthStencilState.DepthWriteMask = D3D12_DEPTH_WRITE_MASK_ZERO;

ThrowIfFailed(m_device->CreateGraphicsPipelineState(&psoDesc, IID_PPV_ARGS(&m_queryState)));
```

Create a buffer to store the results of the query

In the **LoadAssets** method a buffer needs to be created to store the results of the query. Each query requires 8 bytes of space in GPU memory. This sample only performs one query and for simplicity and readability creates a buffer exactly that size (even though this function call will allocate a 64K page of GPU memory - most real apps would likely create a larger buffer).

```
// Create the query result buffer.
ThrowIfFailed(m_device->CreateCommittedResource(
                  &CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_DEFAULT),
                  D3D12_HEAP_FLAG_NONE,
                  &CD3DX12_RESOURCE_DESC::Buffer(8),
                  D3D12_RESOURCE_STATE_GENERIC_READ,
                  nullptr,
                  IID_PPV_ARGS(&m_queryResult))
);
```

Draw the quads and perform and resolve the occlusion query

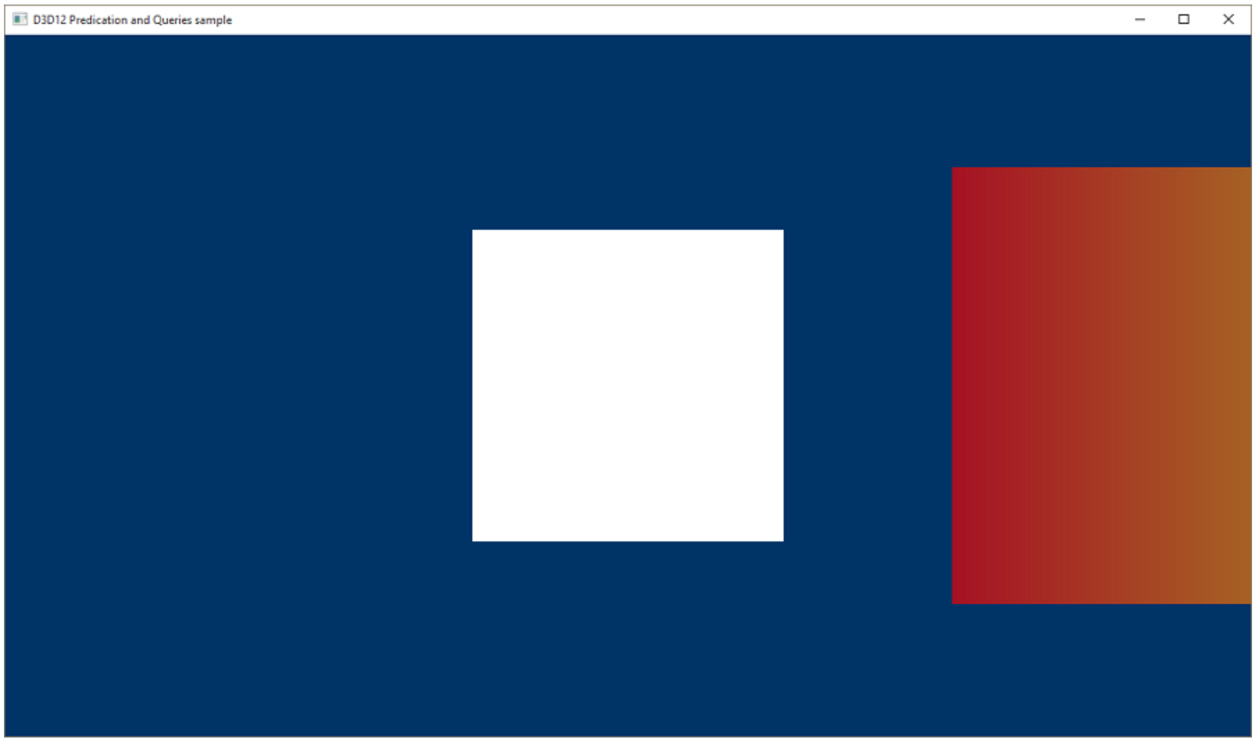
Having done the setup, the main loop is updated in the **PopulateCommandLists** method.

- Draw the quads from back to front to make the transparency effect work properly. Drawing the quad in back to front is predicated on the result of the previous frame's query and is a fairly common technique for this.
- Change the PSO to disable render target and depth stencil writes.
- Perform the occlusion query.
- Resolve the occlusion query.

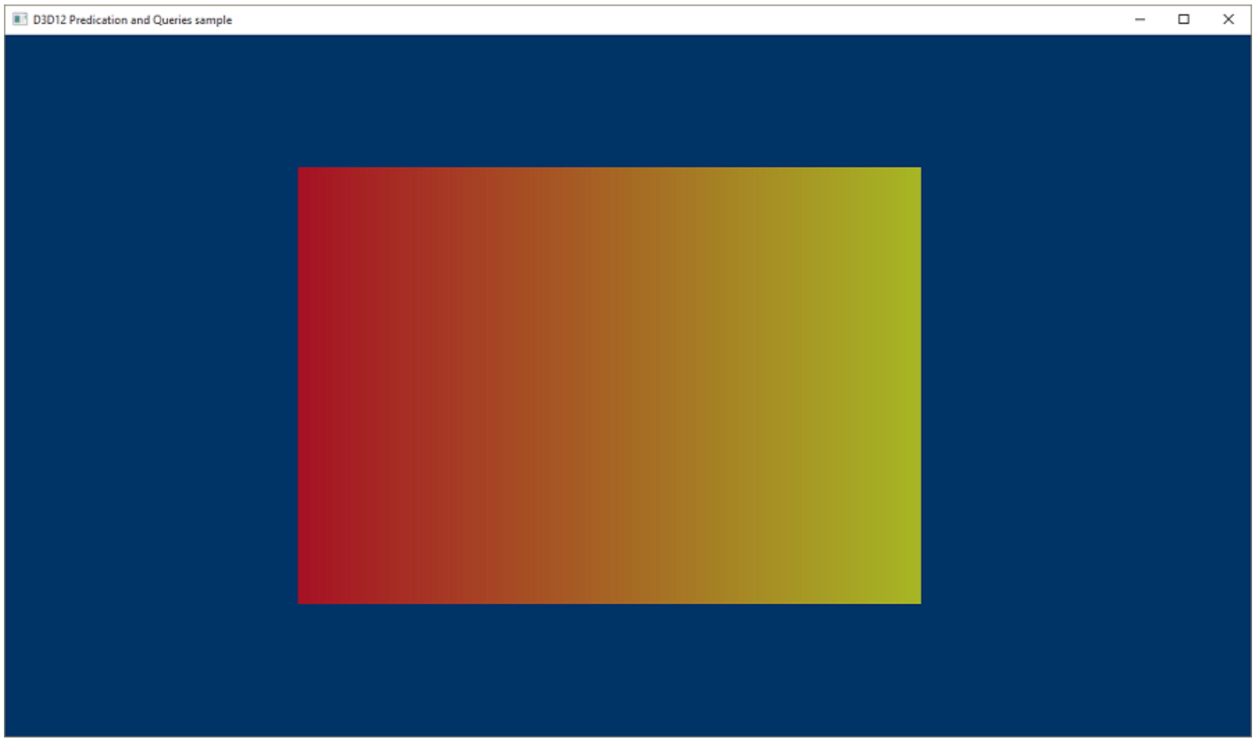
```
// Draw the quads and perform the occlusion query.
{
    CD3DX12_GPU_DESCRIPTOR_HANDLE cbvFarQuad(m_cbvHeap->GetGPUDescriptorHandleForHeapStart(),
                                              m_frameIndex * CbvCountPerFrame,
                                              m_cbvSrvDescriptorSize);
    CD3DX12_GPU_DESCRIPTOR_HANDLE cbvNearQuad(cbvFarQuad, m_cbvSrvDescriptorSize);

    m_commandList->IASetPrimitiveTopology(D3D_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP);
    m_commandList->IASetVertexBuffers(0, 1, &m_vertexBufferView);

    // Draw the far quad conditionally based on the result of the occlusion query
    // from the previous frame.
    m_commandList->SetGraphicsRootDescriptorTable(0, cbvFarQuad);
    m_commandList->SetPredication(m_queryResult.Get(), 0, D3D12_PREDICATION_OP_EQUAL_ZERO);
    m_commandList->DrawInstanced(4, 1, 0, 0);

    // Disable predication and always draw the near quad.
    m_commandList->SetPredication(nullptr, 0, D3D12_PREDICATION_OP_EQUAL_ZERO);
    m_commandList->SetGraphicsRootDescriptorTable(0, cbvNearQuad);
    m_commandList->DrawInstanced(4, 1, 4, 0);

    // Run the occlusion query with the bounding box quad.
    m_commandList->SetGraphicsRootDescriptorTable(0, cbvFarQuad);
    m_commandList->SetPipelineState(m_queryState.Get());
    m_commandList->BeginQuery(m_queryHeap.Get(), D3D12_QUERY_TYPE_BINARY_OCCLUSION, 0);
    m_commandList->DrawInstanced(4, 1, 8, 0);
    m_commandList->EndQuery(m_queryHeap.Get(), D3D12_QUERY_TYPE_BINARY_OCCLUSION, 0);

    // Resolve the occlusion query and store the results in the query result buffer
    // to be used on the subsequent frame.
    m_commandList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(m_queryResult.Get(),
                               D3D12_RESOURCE_STATE_GENERIC_READ, D3D12_RESOURCE_STATE_COPY_DEST));
    m_commandList->ResolveQueryData(m_queryHeap.Get(), D3D12_QUERY_TYPE_BINARY_OCCLUSION, 0, 1,
                                m_queryResult.Get(), 0);
    m_commandList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(m_queryResult.Get(),
                               D3D12_RESOURCE_STATE_COPY_DEST, D3D12_RESOURCE_STATE_GENERIC_READ));
}
```

Run the sample

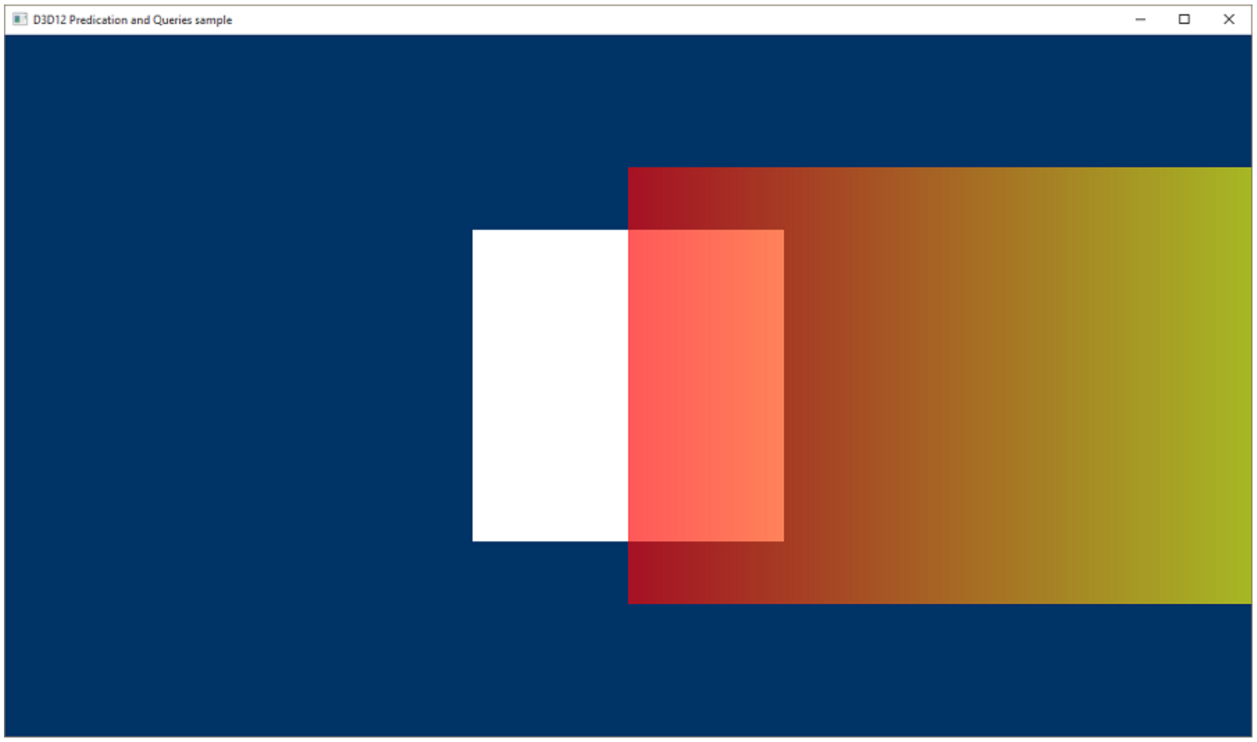
Not occluded:



Occluded:



Partially occluded:



Dynamic Indexing using HLSL 5.1

The **D3D12DynamicIndexing** sample demonstrates some of the new HLSL features available in Shader Model 5.1 - particularly dynamic indexing and unbounded arrays - to render the same mesh multiple times, each time rendering it with a dynamically selected material. With dynamic indexing, shaders can now index into an array without knowing the value of the index at compile time. When combined with unbounded arrays, this adds another level of indirection and flexibility for shader authors and art pipelines.

Setup the pixel shader

Let's first look at the shader itself, which for this sample is a pixel shader.

```
Texture2D g_txDiffuse : register(t0);
Texture2D g_txMats[] : register(t1);
SamplerState g_sampler : register(s0);

struct PSSceneIn {
    float4 pos : SV_Position;
    float2 tex : TEXCOORD0;
};

struct MaterialConstants {
    uint matIndex; // Dynamically set index for looking up from g_txMats[].
};
ConstantBuffer<MaterialConstants> materialConstants : register(b0, space0);
```

```
float4 PSSceneMain(PSSceneIn input) : SV_Target {
    float3 diffuse = g_txDiffuse.Sample(g_sampler, input.tex).rgb;
    float3 mat = g_txMats[materialConstants.matIndex].Sample(g_sampler, input.tex).rgb;
    return float4(diffuse * mat, 1.0f);
}
```

The unbounded array feature is illustrated by the `g_txMats[]` array as it does not specify an array size. Dynamic indexing is used to index into `g_txMats[]` with`matIndex`, which is defined as a root constant. The shader has no knowledge of the size or the array or the value of the index at compile-time. Both attributes are defined in the root signature of the pipeline state object used with the shader.

To take advantage of the dynamic indexing features in HLSL requires that the shader be compiled with SM 5.1. Additionally, to make use of unbounded arrays, the **/enable_unbounded_descriptor_tables** flag must also be used. The following command line options are used to compile this shader with fxc:

```
/Zi /E"PSSceneMain" /Od /Fo"dynamic_indexing_pixel.cso" /ps"_5_1" /nologo /enable_unbounded_descriptor_tables
```

Setup the root signature

Now, let's look at the root signature definition, particularly, how we define the size of the unbounded array and link a root constant to `matIndex`. For the pixel shader, we define three things: a descriptor table for SRVs (our Texture2Ds), a descriptor table for Samplers and a single root constant. The descriptor table for our SRVs contains `CityMaterialCount + 1` entries. `CityMaterialCount` is a constant that defines the length of `g_txMats[]` and the `+ 1` is for `g_txDiffuse`. The descriptor table for our Samplers contains only one entry and we only define one 32-bit root constant value via **InitAsConstants**(…)., in the **LoadAssets**method.

```
// Create the root signature.
{
    CD3DX12_DESCRIPTOR_RANGE ranges[3];
    // Diffuse texture + array of materials.
    ranges[0].Init(D3D12_DESCRIPTOR_RANGE_TYPE_SRV, 1 + CityMaterialCount, 0);
    ranges[1].Init(D3D12_DESCRIPTOR_RANGE_TYPE_SAMPLER, 1, 0);
    ranges[2].Init(D3D12_DESCRIPTOR_RANGE_TYPE_CBV, 1, 0);

    CD3DX12_ROOT_PARAMETER rootParameters[4];
    rootParameters[0].InitAsDescriptorTable(1, &ranges[0], D3D12_SHADER_VISIBILITY_PIXEL);
    rootParameters[1].InitAsDescriptorTable(1, &ranges[1], D3D12_SHADER_VISIBILITY_PIXEL);
    rootParameters[2].InitAsDescriptorTable(1, &ranges[2], D3D12_SHADER_VISIBILITY_VERTEX);
    rootParameters[3].InitAsConstants(1, 0, 0, D3D12_SHADER_VISIBILITY_PIXEL);

    CD3DX12_ROOT_SIGNATURE_DESC rootSignatureDesc;
    rootSignatureDesc.Init(_countof(rootParameters), rootParameters, 0, nullptr,
                           D3D12_ROOT_SIGNATURE_FLAG_ALLOW_INPUT_ASSEMBLER_INPUT_LAYOUT);

    ComPtr<ID3DBlob> signature;
    ComPtr<ID3DBlob> error;
    ThrowIfFailed(D3D12SerializeRootSignature(&rootSignatureDesc, D3D_ROOT_SIGNATURE_VERSION_1, &signature,
                                              &error));
    ThrowIfFailed(m_device->CreateRootSignature(0, signature->GetBufferPointer(), signature->GetBufferSize(),
                                                IID_PPV_ARGS(&m_rootSignature)));
    }
```

Upload the texture data

The contents of `g_txMats[]` are procedurally generated textures created in **LoadAssets**. Each city rendered in the scene shares the same diffuse texture but each also has its own procedurally generated texture. The array of textures span the rainbow spectrum to easily visualize the indexing technique. Texture data is uploaded to the GPU via an upload heap and SRVs are created for each and stored in an SRV descriptor heap. The diffuse texture, g_txDiffuse, is uploaded in a similar manner and also gets its own SRV, but the texture data is already defined in occcity.bin.

```
// Create the textures and sampler.
{
    // Procedurally generate an array of textures to use as city materials.
    {
        // All of these materials use the same texture desc.
        D3D12_RESOURCE_DESC textureDesc = {};
        textureDesc.MipLevels = 1;
        textureDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
        textureDesc.Width = CityMaterialTextureWidth;
        textureDesc.Height = CityMaterialTextureHeight;
        textureDesc.Flags = D3D12_RESOURCE_FLAG_NONE;
        textureDesc.DepthOrArraySize = 1;
        textureDesc.SampleDesc.Count = 1;
        textureDesc.SampleDesc.Quality = 0;
        textureDesc.Dimension = D3D12_RESOURCE_DIMENSION_TEXTURE2D;

        // The textures evenly span the color rainbow so that each city gets
        // a different material.
        float materialGradStep = (1.0f / static_cast<float>(CityMaterialCount));

        // Generate texture data.
        vector<vector<unsigned char>> cityTextureData;
        cityTextureData.resize(CityMaterialCount);
        for (int i = 0; i < CityMaterialCount; ++i) {
            ThrowIfFailed(m_device->CreateCommittedResource(
                &CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_DEFAULT),
                D3D12_HEAP_FLAG_NONE,
                &textureDesc,
                D3D12_RESOURCE_STATE_COPY_DEST,
                nullptr,
                IID_PPV_ARGS(&m_cityMaterialTextures[i])));

            // Fill the texture.
            float t = i * materialGradStep;
            cityTextureData[i].resize(CityMaterialTextureWidth * CityMaterialTextureHeight *
                                      CityMaterialTextureChannelCount);
            for (int x = 0; x < CityMaterialTextureWidth; ++x) {
                for (int y = 0; y < CityMaterialTextureHeight; ++y) {
                    // Compute the appropriate index into the buffer based on the x/y coordinates.
                    int pixelIndex = (y * CityMaterialTextureChannelCount * CityMaterialTextureWidth) + (x *
                                      CityMaterialTextureChannelCount);
                    // Determine this row's position along the rainbow gradient.
                    float tPrime = t + ((static_cast<float>(y) /
                                   static_cast<float>(CityMaterialTextureHeight)) * materialGradStep);

                    // Compute the RGB value for this position along the rainbow
                    // and pack the pixel value.
                    XMVECTOR hsl = XMVectorSet(tPrime, 0.5f, 0.5f, 1.0f);
                    XMVECTOR rgb = XMColorHSLToRGB(hsl);
                    cityTextureData[i][pixelIndex + 0] = static_cast<unsigned char>((255 *
                                                          XMVectorGetX(rgb)));
                    cityTextureData[i][pixelIndex + 1] = static_cast<unsigned char>((255 *
                                                          XMVectorGetY(rgb)));
                    cityTextureData[i][pixelIndex + 2] = static_cast<unsigned char>((255 *
                                                          XMVectorGetZ(rgb)));
                    cityTextureData[i][pixelIndex + 3] = 255;
                }
            }
        }

    // Upload texture data to the default heap resources.
    {
        const UINT subresourceCount = textureDesc.DepthOrArraySize * textureDesc.MipLevels;
        const UINT64 uploadBufferStep = GetRequiredIntermediateSize(m_cityMaterialTextures[0].Get(), 0,
                        subresourceCount); // All of our textures are the same size in this case.
        const UINT64 uploadBufferSize = uploadBufferStep * CityMaterialCount;


         ThrowIfFailed(m_device->CreateCommittedResource(
                            &CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_UPLOAD),
                            D3D12_HEAP_FLAG_NONE,
                            &CD3DX12_RESOURCE_DESC::Buffer(uploadBufferSize),
                            D3D12_RESOURCE_STATE_GENERIC_READ,
                            nullptr,
                            IID_PPV_ARGS(&materialsUploadHeap)));

        for (int i = 0; i < CityMaterialCount; ++i) {
            // Copy data to the intermediate upload heap and then schedule
            // a copy from the upload heap to the appropriate texture.
            D3D12_SUBRESOURCE_DATA textureData = {};
            textureData.pData = &cityTextureData[i][0];
            textureData.RowPitch = static_cast<LONG_PTR>((CityMaterialTextureChannelCount *
                                                       textureDesc.Width));
            textureData.SlicePitch = textureData.RowPitch * textureDesc.Height;

            UpdateSubresources(m_commandList.Get(), m_cityMaterialTextures[i].Get(),
                            materialsUploadHeap.Get(), i * uploadBufferStep, 0,
                            subresourceCount, &textureData);
            m_commandList->ResourceBarrier(
                                   1,
                                   &CD3DX12_RESOURCE_BARRIER::Transition(m_cityMaterialTextures[i].Get(),
                                   D3D12_RESOURCE_STATE_COPY_DEST,
                                   D3D12_RESOURCE_STATE_PIXEL_SHADER_RESOURCE));
        }
    }
}

// Load the occcity diffuse texture with baked-in ambient lighting.
// This texture will be blended with a texture from the materials
// array in the pixel shader.
{
    D3D12_RESOURCE_DESC textureDesc = {};
    textureDesc.MipLevels = SampleAssets::Textures[0].MipLevels;
    textureDesc.Format = SampleAssets::Textures[0].Format;
    textureDesc.Width = SampleAssets::Textures[0].Width;
    textureDesc.Height = SampleAssets::Textures[0].Height;
    textureDesc.Flags = D3D12_RESOURCE_FLAG_NONE;
    textureDesc.DepthOrArraySize = 1;
    textureDesc.SampleDesc.Count = 1;
    textureDesc.SampleDesc.Quality = 0;
    textureDesc.Dimension = D3D12_RESOURCE_DIMENSION_TEXTURE2D;

    ThrowIfFailed(m_device->CreateCommittedResource(
                &CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_DEFAULT),
                D3D12_HEAP_FLAG_NONE,
                &textureDesc,
                D3D12_RESOURCE_STATE_COPY_DEST,
                nullptr,
                IID_PPV_ARGS(&m_cityDiffuseTexture)));

    const UINT subresourceCount = textureDesc.DepthOrArraySize * textureDesc.MipLevels;
    const UINT64 uploadBufferSize = GetRequiredIntermediateSize(m_cityDiffuseTexture.Get(),
                                                             0,
                                                             subresourceCount);
    ThrowIfFailed(m_device->CreateCommittedResource(
                &CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_UPLOAD),
                D3D12_HEAP_FLAG_NONE,
                &CD3DX12_RESOURCE_DESC::Buffer(uploadBufferSize),
                D3D12_RESOURCE_STATE_GENERIC_READ,
                nullptr,
                IID_PPV_ARGS(&textureUploadHeap)));
    // Copy data to the intermediate upload heap and then schedule
    // a copy from the upload heap to the diffuse texture.
    D3D12_SUBRESOURCE_DATA textureData = {};
    textureData.pData = pMeshData + SampleAssets::Textures[0].Data[0].Offset;
    textureData.RowPitch = SampleAssets::Textures[0].Data[0].Pitch;
    textureData.SlicePitch = SampleAssets::Textures[0].Data[0].Size;

    UpdateSubresources(m_commandList.Get(), m_cityDiffuseTexture.Get(), textureUploadHeap.Get(), 0, 0,
                       subresourceCount, &textureData);
    m_commandList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(m_cityDiffuseTexture.Get(),
                           D3D12_RESOURCE_STATE_COPY_DEST, D3D12_RESOURCE_STATE_PIXEL_SHADER_RESOURCE));
}
```

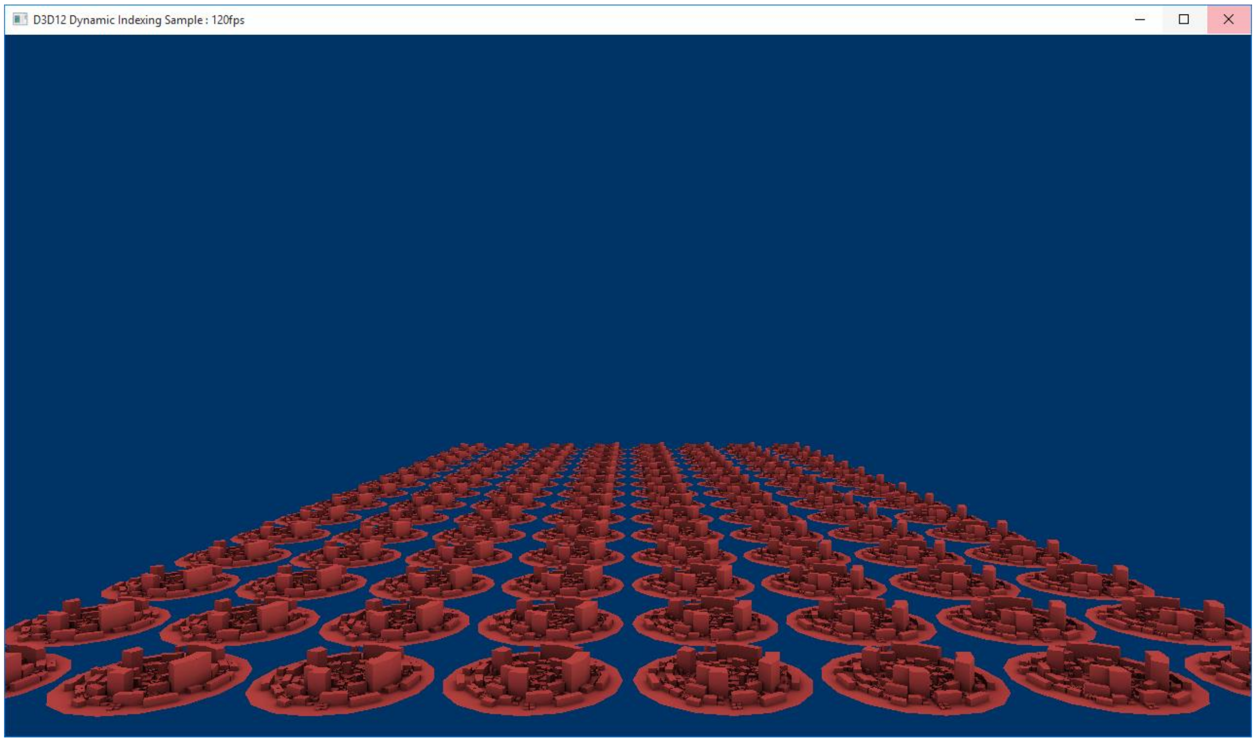
Create a sampler

Finally for **LoadAssets**, a single sampler is created to sample from either the diffuse texture or the texture array.

```
// Describe and create a sampler.
D3D12_SAMPLER_DESC samplerDesc = {};
samplerDesc.Filter = D3D12_FILTER_MIN_MAG_MIP_LINEAR;
samplerDesc.AddressU = D3D12_TEXTURE_ADDRESS_MODE_WRAP;
samplerDesc.AddressV = D3D12_TEXTURE_ADDRESS_MODE_WRAP;
samplerDesc.AddressW = D3D12_TEXTURE_ADDRESS_MODE_WRAP;
samplerDesc.MinLOD = 0;
samplerDesc.MaxLOD = D3D12_FLOAT32_MAX;
samplerDesc.MipLODBias = 0.0f;
samplerDesc.MaxAnisotropy = 1;
samplerDesc.ComparisonFunc = D3D12_COMPARISON_FUNC_ALWAYS;
m_device->CreateSampler(&samplerDesc, m_samplerHeap->GetCPUDescriptorHandleForHeapStart());

// Create SRV for the city's diffuse texture.
CD3DX12_CPU_DESCRIPTOR_HANDLE srvHandle(m_cbvSrvHeap->GetCPUDescriptorHandleForHeapStart(), 0,
                                        m_cbvSrvDescriptorSize);
D3D12_SHADER_RESOURCE_VIEW_DESC diffuseSrvDesc = {};
diffuseSrvDesc.Shader4ComponentMapping = D3D12_DEFAULT_SHADER_4_COMPONENT_MAPPING;
diffuseSrvDesc.Format = SampleAssets::Textures->Format;
diffuseSrvDesc.ViewDimension = D3D12_SRV_DIMENSION_TEXTURE2D;
diffuseSrvDesc.Texture2D.MipLevels = 1;
m_device->CreateShaderResourceView(m_cityDiffuseTexture.Get(), &diffuseSrvDesc, srvHandle);
srvHandle.Offset(m_cbvSrvDescriptorSize);

// Create SRVs for each city material.
for (int i = 0; i < CityMaterialCount; ++i) {
    D3D12_SHADER_RESOURCE_VIEW_DESC materialSrvDesc = {};
    materialSrvDesc.Shader4ComponentMapping = D3D12_DEFAULT_SHADER_4_COMPONENT_MAPPING;
    materialSrvDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
    materialSrvDesc.ViewDimension = D3D12_SRV_DIMENSION_TEXTURE2D;
    materialSrvDesc.Texture2D.MipLevels = 1;
    m_device->CreateShaderResourceView(m_cityMaterialTextures[i].Get(), &materialSrvDesc, srvHandle);
    srvHandle.Offset(m_cbvSrvDescriptorSize);
}
```

Dynamically change the root parameter index

If we were to render the scene now, all of the cities would appear the same, because we have not set the value of our root constant, `matIndex`. Each pixel shader would index into the 0th slot of `g_txMats` and the scene would look like this:

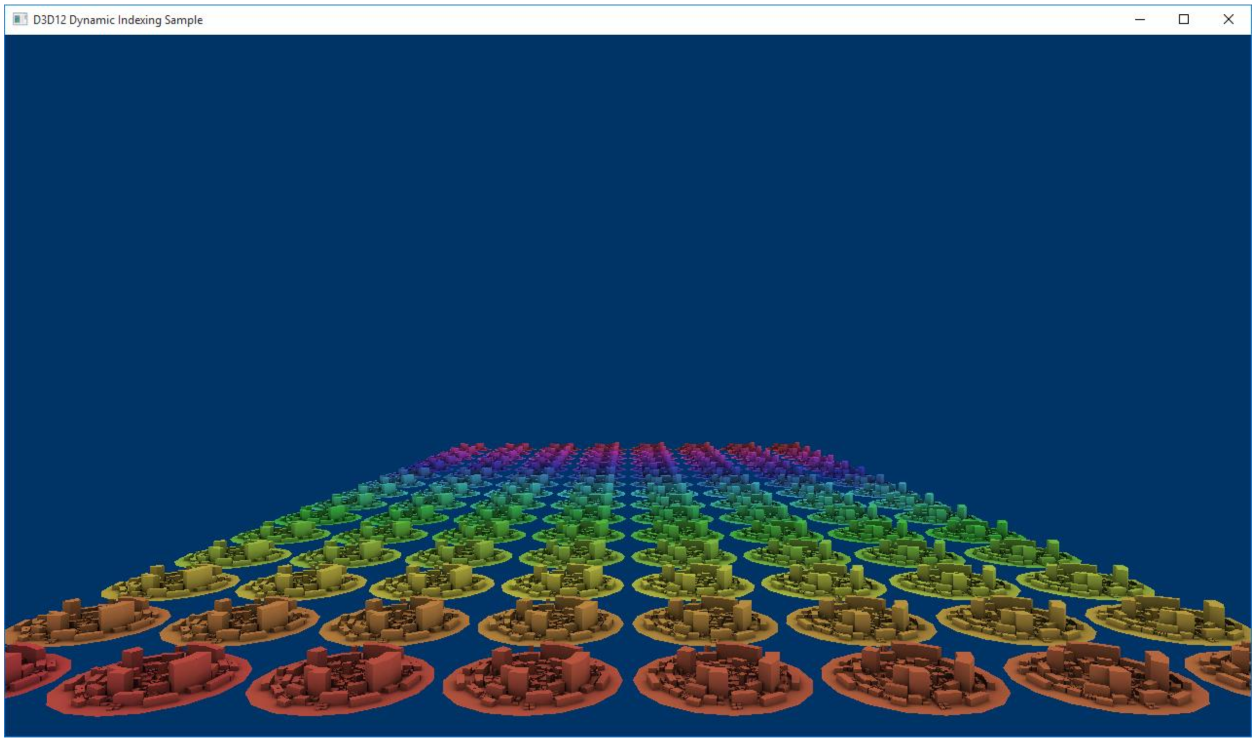


The value of the root constant is set in **FrameResource::PopulateCommandLists**. In the double **for** loop where a draw command is recorded for each city, we record a call to SetGraphicsRoot32BitConstants specifying our root parameter index in regards to the root signature – in this case 3 – the value of the dynamic index and an offset – in this case 0. Since the length of `g_txMats` is equal to the number of cities we render, the value of the index is incrementally set for each city.

```
for (UINT i = 0; i < m_cityRowCount; i++) {
    for (UINT j = 0; j < m_cityColumnCount; j++) {
        pCommandList->SetPipelineState(pPso);

        // Set the city's root constant for dynamically indexing into the material array.
        pCommandList->SetGraphicsRoot32BitConstant(3, (i * m_cityColumnCount) + j, 0);

        // Set this city's CBV table and move to the next descriptor.
        pCommandList->SetGraphicsRootDescriptorTable(2, cbvSrvHandle);
        cbvSrvHandle.Offset(cbvSrvDescriptorSize);

        pCommandList->DrawIndexedInstanced(numIndices, 1, 0, 0, 0);
    }
}
```

Run the sample

Now when we render the scene, each city will have a different value for `matIndex` and will thus look up a different texture from `g_txMats[]` making the scene look like this:


D3D12 Dynamic Indexing Sample