II. Direct3D Foundations
4. Direct3D Initialization

Game Graphic Programming

Kyung Hee University

Software Convergence

Prof. Daeho Lee

Environment Setup

- Direct3D 12 programming environment setup
  - https://learn.microsoft.com/en-us/windows/win32/direct3d12/directx-12-programming-environment-set-up
  - 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.
  - To use the Direct3D 12 API, include D3d12.h and link to D3d12.lib, or query the entry points directly in D3d12.dll.

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

Direct3D & COM (1)

- Direct3D is a low-level graphics API (application programming interface) used to control and program the GPU (graphics processing unit) from our application, thereby allowing us to render virtual 3D worlds using hardware acceleration.

- COM
  - **Component Object Model (COM)** is the technology that allows DirectX to be programming-language independent and have backwards compatibility.
  - We usually refer to a COM object as an interface, which for our purposes can be thought of and used as a C++ class.
  - All COM interfaces inherit functionality from the **`IUnknown`** COM interface.

Direct3D & COM (2)

- **`Comptr`** class
  - Template smart-pointer for COM objects
  - **`Comptr`** automatically maintains a reference count for the underlying interface pointer and releases the interface when the reference counter goes to zero.
  - **`Microsoft::WRL::ComPtr`** class (**`#include <wrl.h>`**)
  - WRL: Windows runtime C++ template library
  - The three main **`ComPtr`** methods:
    - **`Get`**: It returns a pointer to the underlying COM interface.
    - **`GetAddressOf`**: It returns the address of the pointer to the underlying COM interface.
    - **`Reset`**: It sets the **`ComPtr`** instance to **`nullptr`** and decrements the reference counter of the underlying COM interface.

Textures Formats (1)

- Textures formats
  - A 2D texture is a matrix of data elements. One use for 2D textures is to store 2D image data, where each element in the texture stores the color of a pixel.

R:red, G: green, B: blue, A: alpha

The alpha channel or alpha component is generally used to control transparency.

U: unsigned, S: signed, NORM: normalized to [0,1] (unsigned) or [-1,1] (signed)

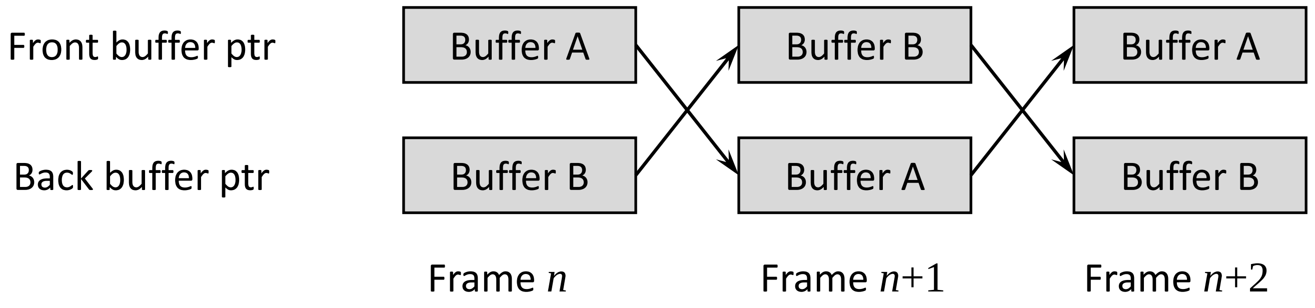
  - DXGI: DirectX Graphics Infrastructure (Direct3D 10/11/12)
  - **`DXGI_FORMAT`**
    - **`DXGI_FORMAT_R32G32B32_FLOAT`**: Each element has three 32-bit floating-point components.
    - **`DXGI_FORMAT_R16G16B16A16_UNORM`**: Each element has four 16-bit components mapped to the [0, 1] range.
    - **`DXGI_FORMAT_R32G32_UINT`**: Each element has two 32-bit unsigned integer components.

Textures Formats (2)

- **`DXGI_FORMAT_R8G8B8A8_UNORM`**: Each element has four 8-bit unsigned components mapped to the [0, 1] range.
- **`DXGI_FORMAT_R8G8B8A8_SNORM`**: Each element has four 8-bit signed components mapped to the [-1, 1] range.
- **`DXGI_FORMAT_R8G8B8A8_SINT`**: Each element has four 8-bit signed integer components mapped to the [-128, 127] range.
- **`DXGI_FORMAT_R8G8B8A8_UINT`**: Each element has four 8-bit unsigned integer components mapped to the [0, 255] range
- **`DXGI_FORMAT_R16G16B16A16_TYPELESS`**: Typeless format reserves elements with four 16-bit components, but does not specify the data type (e.g., integer, floating-point, unsigned integer).

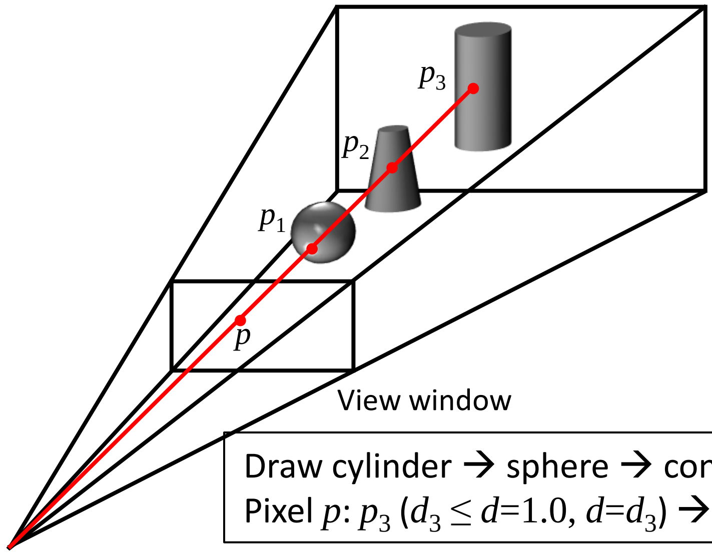
Swap Chain & Page Flipping

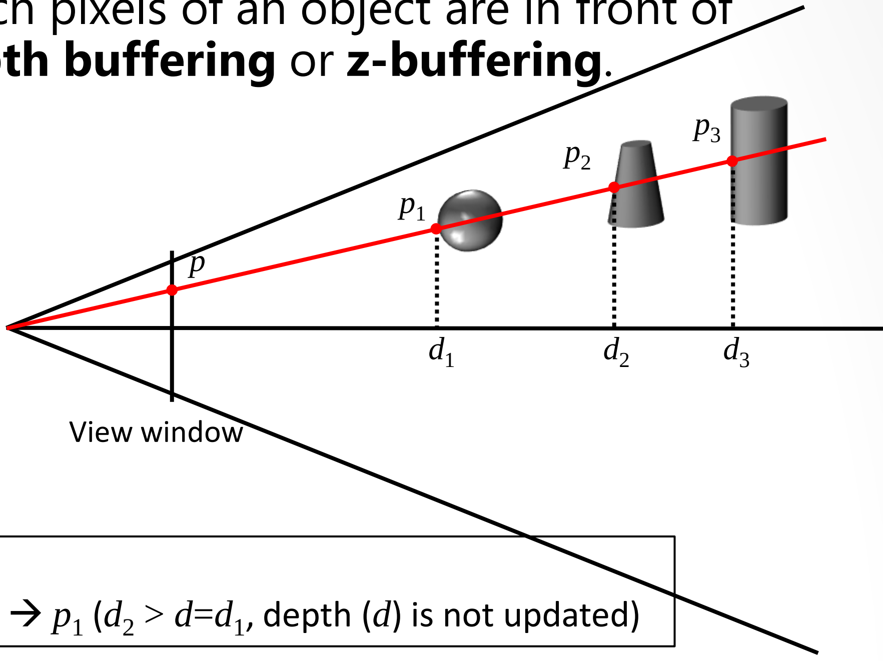
- Swap chain and page flipping
  - To avoid flickering in animation, it is best to draw an entire frame of animation into an off-screen texture called the back buffer.
  - Two texture buffers are maintained by the hardware, one called the front buffer and a second called the back buffer.
  - The front buffer stores the image data currently being displayed on the monitor, while the next frame of animation is being drawn to the back buffer.
  - After the frame has been drawn to the back buffer, the roles of the back buffer and front buffer are reversed.
  - Swapping the roles of the back and front buffers is called **presenting**.
  - In Direct3D, a swap chain is represented by the **`IDXGISwapChain`** interface.



Depth Buffering (1)

- Depth buffering (z-buffering)
  - The depth buffer is an example of a texture that does not contain image data, but rather depth information about a particular pixel.
  - The possible depth values range from 0.0 to 1.0, where 0.0 denotes the closest an object in the view frustum can be to the viewer and 1.0 denotes the farthest an object in the view frustum can be from the viewer.
  - In order for Direct3D to determine which pixels of an object are in front of another, it uses a technique called **depth buffering** or **z-buffering**.





Draw cylinder → sphere → conical frustum
Pixel p: p_3 ($d_3 \le d=1.0$, $d=d_3$) → p_1 ($d_1 \le d=d_3$, $d=d_1$) → p_1 ($d_2 > d=d_1$, depth (d) is not updated)

Depth Buffering (2)

- The depth buffer is a texture, so it must be created with certain data formats.
- The formats used for depth buffering are as follows:
  - An application is not required to have a stencil buffer, but if it does, the stencil buffer is always attached to the depth buffer. For example, the format **`DXGI_FORMAT_D24_UNORM_S8_UINT`** uses 24 bits for the depth buffer and 8 bits for the stencil buffer.
  - **`DXGI_FORMAT_D32_FLOAT_S8X24_UINT`**: Specifies a 32-bit floating-point depth buffer, with 8 bits (unsigned integer) reserved for the stencil buffer mapped to the [0, 255] range and 24-bits not used for padding.
  - **`DXGI_FORMAT_D32_FLOAT`**: Specifies a 32-bit floating-point depth buffer.
  - **`DXGI_FORMAT_D24_UNORM_S8_UINT`**: Specifies an unsigned 24-bit depth buffer mapped to the [0, 1] range with 8 bits (unsigned integer) reserved for the stencil buffer mapped to the [0, 255] range.
  - **`DXGI_FORMAT_D16_UNORM`**: Specifies an unsigned 16-bit depth buffer mapped to the [0, 1] range.

A stencil buffer is used to mask pixels in an image, to produce special effects, including dissolves, decaling, and outlining.

Resources & Descriptors (1)

- **Resources** and **descriptors**
  - During the rendering process, the GPU will write to resources (e.g., the back buffer, the depth/stencil buffer), and read from resources (e.g., textures that describe the appearance of surfaces, buffers that store the 3D positions of geometry in the scene). Before we issue a draw command, we need to bind (or link) the resources to the rendering pipeline that are going to be referenced in that draw call.
  - Some of the resources may change per draw call, so we need to update the bindings per draw call if necessary. However, GPU resources are not bound directly. Instead, a resource is referenced through a **descriptor** object, which can be thought of as **a lightweight structure that describes the resource to the GPU**.

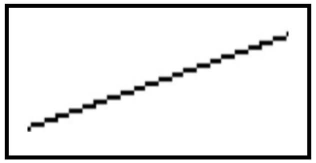
Resources & Descriptors (2)

- The types of descriptors used in this lecture:
  - Constant buffer views (CBVs), unordered access views (UAVs), shader resource views (SRVs) → CBV/SRV/UAV descriptor
  - Samplers (used in texturing) → sampler descriptor
  - Render target views (RTVs) → RTV descriptor
  - Depth stencil views (DSV) → DSV descriptor

The term "**view**" is used to mean "**data in the required format**".
For example, a Constant Buffer View (CBV) would be constant buffer data correctly formatted.

Multisampling

- Multisampling theory
  - Because the pixels on a monitor are not infinitely small, an arbitrary line cannot be represented perfectly on the computer monitor.
    - Stair-step (aliasing) effect
  - Shrinking the pixel sizes by increasing the monitor resolution can alleviate the problem significantly to where the stair-step effect goes largely unnoticed.
  - When increasing the monitor resolution is not possible or not enough, we can apply antialiasing techniques.
  - One technique, called supersampling, works by making the back buffer and depth buffer 4X bigger than the screen resolution. The 3D scene is then rendered to the back buffer at this larger resolution. Then, when it comes time to present the back buffer to the screen, the back buffer is resolved (or downsampled) such that 4-pixel block colors are averaged together to get an averaged pixel color.



Multisampling in Direct3D (1)

- Multisampling in Direct3D
  - Getting information about the features that are supported by the current graphics driver.

    -
      ```
      HRESULT CheckFeatureSupport(
                       D3D12_FEATURE Feature,
           [in, out] void          *pFeatureSupportData,
                       UINT          FeatureSupportDataSize
      );
      ```
      - **Feature** is a constant from the **D3D12_FEATURE** enumeration describing the feature(s) that you want to query for support.
      - **pFeatureSupportData** is a pointer to a data structure that corresponds to the value of the Feature parameter.

        -
          ```
          typedef enum D3D12_FEATURE {
            D3D12_FEATURE_D3D12_OPTIONS = 0,
            // …
            D3D12_FEATURE_MULTISAMPLE_QUALITY_LEVELS = 4,
            D3D12_FEATURE_FORMAT_INFO = 5,
            // …
            D3D12_FEATURE_D3D12_OPTIONS21
          } ;
          ```
      - **FeatureSupportDataSize** is the size of the structure pointed to by the **pFeatureSupportData** parameter.

HRESULT:S_OK(0x00000000), 0x8…(E_ABORT, E_FAIL, …)

The **SUCCEEDED** macro returns **TRUE** for a success code and **FALSE** for a failure code.

```
 #define SUCCEEDED(hr) (((HRESULT)(hr)) >= 0)
```

The **FAILED** macro returns **TRUE** for a failure code and **FALSE** for a success code.

Multisampling in Direct3D (2)

-
  ```
  typedef struct
  D3D12_FEATURE_DATA_MULTISAMPLE_QUALITY_LEVELS {
   DXGI_FORMAT Format;
   UINT SampleCount;
   D3D12_MULTISAMPLE_QUALITY_LEVELS_FLAG Flags;
   UINT NumQualityLevels;
  } D3D12_FEATURE_DATA_MULTISAMPLE_QUALITY_LEVELS;
  ```

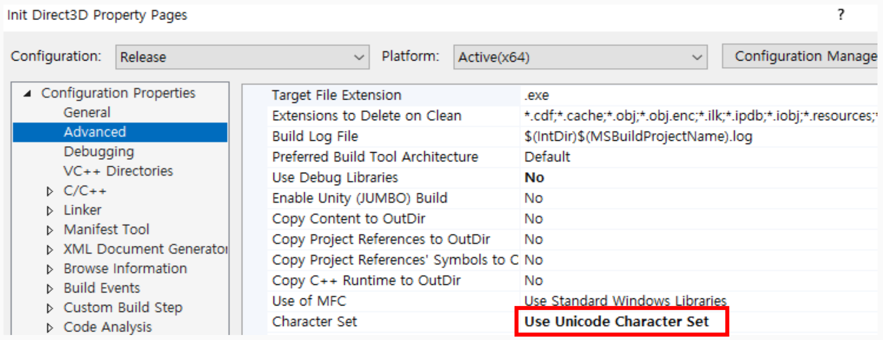
```
D3D12_FEATURE_DATA_MULTISAMPLE_QUALITY_LEVELS msQualityLevels;
msQualityLevels.Format = mBackBufferFormat;
// DXGI_FORMAT mBackBufferFormat = DXGI_FORMAT_R8G8B8A8_UNORM;
msQualityLevels.SampleCount = 4;
msQualityLevels.Flags = D3D12_MULTISAMPLE_QUALITY_LEVELS_FLAG_NONE;
msQualityLevels.NumQualityLevels = 0;

ThrowIfFailed(md3dDevice->CheckFeatureSupport(
  D3D12_FEATURE_MULTISAMPLE_QUALITY_LEVELS,
  &msQualityLevels,
  sizeof(msQualityLevels)));
```

ThrowIfFailed (1)

```
// d3dUtil.h
#ifndef ThrowIfFailed
#define ThrowIfFailed(x)                                              \
{                                                                     \
  HRESULT hr__ = (x);                                                 \
  std::wstring wfn = AnsiToWString(__FILE__);                         \
  if(FAILED(hr__)) { throw DxException(hr__, L#x, wfn, __LINE__); }  \
} // try … catch → WinMain
#endif
inline std::wstring AnsiToWString(const std::string& str) {
  WCHAR buffer[512];
  MultiByteToWideChar(CP_ACP, 0, str.c_str(), -1, buffer, 512);
  return std::wstring(buffer);
}

class DxException {
std::wstring ToString()const;
// …
};
```



Configuration: Release   Platform: Active(x64)

Target File Extension	.exe
Extensions to Delete on Clean	*.cdf;*.cache;*.obj;*.obj.enc;*.ilk;*.ipdb;*.iobj;*.resources;
Build Log File	$(IntDir)$(MSBuildProjectName).log
Preferred Build Tool Architecture	Default
Use Debug Libraries	**No**
Enable Unity (JUMBO) Build	No
Copy Content to OutDir	No
Copy Project References to OutDir	No
Copy Project References' Symbols to C	No
Copy C++ Runtime to OutDir	No
Use of MFC	Use Standard Windows Libraries
Character Set	**Use Unicode Character Set**

ThrowIfFailed (2)

```
// InitDirect3DApp.cpp
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE prevInstance,
                  PSTR cmdLine, int showCmd) {
// …
    try {
        InitDirect3DApp theApp(hInstance);
        if(!theApp.Initialize())
            return 0;
        return theApp.Run();
    }
    catch(DxException& e) {
        MessageBox(nullptr, e.ToString().c_str(), L"HR Failed", MB_OK);
        return 0;
    }
}
```

Direct3D Feature Levels

- Direct3D feature levels
  - To handle the diversity of video cards in new and existing machines, Microsoft Direct3D 11 introduces the concept of feature levels.
  - Feature level: shader model, raytracing, mesh shader, resource binding, ….
  -
```
typedef enum D3D_FEATURE_LEVEL {
  D3D_FEATURE_LEVEL_1_0_GENERIC,
  D3D_FEATURE_LEVEL_1_0_CORE,
  D3D_FEATURE_LEVEL_9_1,
  D3D_FEATURE_LEVEL_9_2,
  D3D_FEATURE_LEVEL_9_3,
  D3D_FEATURE_LEVEL_10_0,
  D3D_FEATURE_LEVEL_10_1,
  D3D_FEATURE_LEVEL_11_0,
  D3D_FEATURE_LEVEL_11_1,
  D3D_FEATURE_LEVEL_12_0,
  D3D_FEATURE_LEVEL_12_1,
  D3D_FEATURE_LEVEL_12_2
} D3D_FEATURE_LEVEL;
```

In this lecture, we always require support for feature level **`D3D_FEATURE_LEVEL_11_0`**.

DXGI

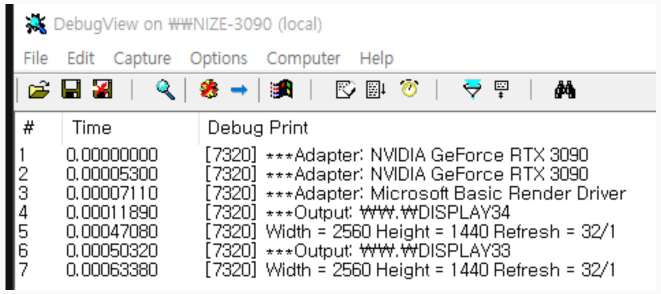
- DirectX graphics infrastructure
  - DirectX Graphics Infrastructure (DXGI) is an API used along with Direct3D.
  - DXGI handles other common graphical functionality like full-screen mode transitions, enumerating graphical system information like display adapters, monitors, and supported display modes (resolution, refresh rate, and such); it also defines the various supported surface formats (**`DXGI_FORMAT`**).

  - Direct3D initialization
    - One of the key DXGI interfaces is the **`IDXGIFactory`** interface, which is primarily used to create the **`IDXGISwapChain`** interface and enumerate display adapters. An **`IDXGIFactory`** interface implements methods for generating DXGI objects
    - A display adapter is represented by the **`IDXGIAdapter`** interface.

An **`IDXGIFactory`** interface implements methods for generating DXGI objects.
An **`IDXGISwapChain`** interface implements one or more surfaces for storing rendered data before presenting it to an output.
The **`IDXGIAdapter`** interface represents a display subsystem.

IDXGIAdapter (1)

```
// d3dApp.cpp
bool D3DApp::InitDirect3D() {
// …
#ifdef _DEBUG
    LogAdapters();
#endif
// …
}
```

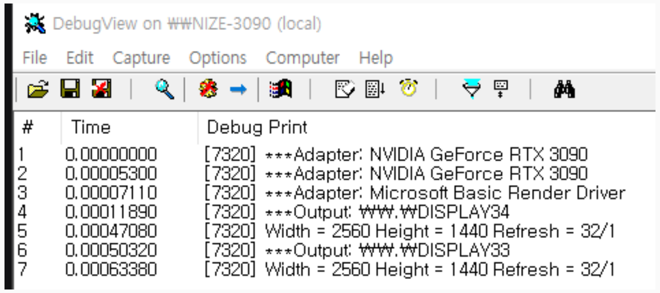

DebugView on ₩₩NIZE-3090 (local)
File Edit Capture Options Computer Help

```
#   Time         Debug Print
1   0.00000000   [7320] ***Adapter: NVIDIA GeForce RTX 3090
2   0.00005300   [7320] ***Adapter: NVIDIA GeForce RTX 3090
3   0.00007110   [7320] ***Adapter: Microsoft Basic Render Driver
4   0.00011890   [7320] ***Output: ₩₩.₩DISPLAY34
5   0.00047080   [7320] Width = 2560 Height = 1440 Refresh = 32/1
6   0.00050320   [7320] ***Output: ₩₩.₩DISPLAY33
7   0.00063380   [7320] Width = 2560 Height = 1440 Refresh = 32/1
```

https://download.sysinternals.com/files/DebugView.zip

IDXGIAdapter (2)

```
// d3dApp.cpp
void D3DApp::LogAdapters() {
    UINT i = 0;
    IDXGIAdapter* adapter = nullptr;
    std::vector<IDXGIAdapter*> adapterList;
// Microsoft::WRL::ComPtr<IDXGIFactory4> mdxgiFactory; // d3dApp.h
// Created in bool D3DApp::InitDirect3D()
    while(mdxgiFactory->EnumAdapters(i, &adapter) != DXGI_ERROR_NOT_FOUND) {
        DXGI_ADAPTER_DESC desc;
        adapter->GetDesc(&desc);
        std::wstring text = L"***Adapter: ";
        text += desc.Description; text += L"\n";
        OutputDebugString(text.c_str());
        adapterList.push_back(adapter);
        ++i;
    }
    for(size_t i = 0; i < adapterList.size(); ++i) {
        LogAdapterOutputs(adapterList[i]); ReleaseCom(adapterList[i]);
    }
}
```


DebugView on ₩₩NIZE-3090 (local)
File Edit Capture Options Computer Help

```
#   Time        Debug Print
1   0.00000000  [7320] ***Adapter: NVIDIA GeForce RTX 3090
2   0.00005300  [7320] ***Adapter: NVIDIA GeForce RTX 3090
3   0.00007110  [7320] ***Adapter: Microsoft Basic Render Driver
4   0.00011890  [7320] ***Output: ₩₩.₩DISPLAY34
5   0.00047080  [7320] Width = 2560 Height = 1440 Refresh = 32/1
6   0.00050320  [7320] ***Output: ₩₩.₩DISPLAY33
7   0.00063380  [7320] Width = 2560 Height = 1440 Refresh = 32/1
```

A display adapter is represented by the **IDXGIAdapter** interface.

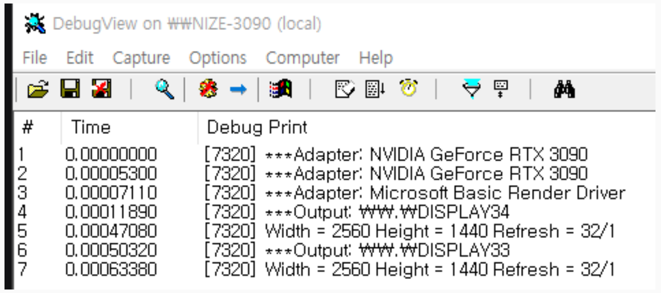
IDXGIAdapter (3)

```
void D3DApp::LogAdapterOutputs(IDXGIAdapter* adapter) {
    UINT i = 0;
    IDXGIOutput* output = nullptr;
    while(adapter->EnumOutputs(i, &output) != DXGI_ERROR_NOT_FOUND){
        DXGI_OUTPUT_DESC desc;
        output->GetDesc(&desc);

        std::wstring text = L"***Output: ";
        text += desc.DeviceName;
        text += L"\n";
        OutputDebugString(text.c_str());

        LogOutputDisplayModes(output, mBackBufferFormat);
        ReleaseCom(output);

        ++i;
    }
}
```


DebugView on ₩₩NIZE-3090 (local)
File Edit Capture Options Computer Help

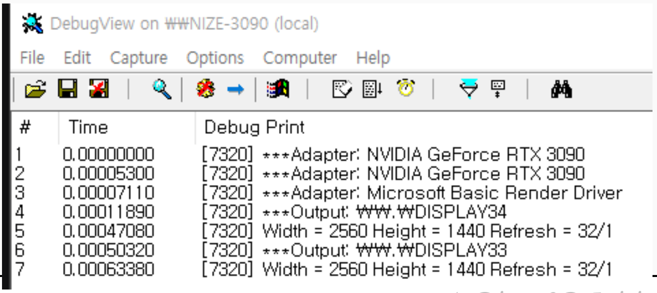
```
#    Time          Debug Print
1    0.00000000    [7320] ***Adapter: NVIDIA GeForce RTX 3090
2    0.00005300    [7320] ***Adapter: NVIDIA GeForce RTX 3090
3    0.00007110    [7320] ***Adapter: Microsoft Basic Render Driver
4    0.00011890    [7320] ***Output: ₩₩.₩DISPLAY34
5    0.00047080    [7320] Width = 2560 Height = 1440 Refresh = 32/1
6    0.00050320    [7320] ***Output: ₩₩.₩DISPLAY33
7    0.00063380    [7320] Width = 2560 Height = 1440 Refresh = 32/1
```

IDXGIAdapter (4)

```
void D3DApp::LogOutputDisplayModes(IDXGIOutput* output, DXGI_FORMAT format) {
    UINT count = 0;
    UINT flags = 0;
    // Call with nullptr to get list count.
    output->GetDisplayModeList(format, flags, &count, nullptr);

    std::vector<DXGI_MODE_DESC> modeList(count);
    output->GetDisplayModeList(format, flags, &count, &modeList[0]);
    for(auto& x : modeList) {
        UINT n = x.RefreshRate.Numerator;
        UINT d = x.RefreshRate.Denominator;
        std::wstring text =
            L"Width = " + std::to_wstring(x.Width) + L" " +
            L"Height = " + std::to_wstring(x.Height) + L" " +
            L"Refresh = " + std::to_wstring(n) + L"/" + std::to_wstring(d) +
            L"\n";

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


DebugView on ₩₩NIZE-3090 (local)
File Edit Capture Options Computer Help

```
#   Time        Debug Print
1   0.00000000  [7320] ***Adapter: NVIDIA GeForce RTX 3090
2   0.00005300  [7320] ***Adapter: NVIDIA GeForce RTX 3090
3   0.00007110  [7320] ***Adapter: Microsoft Basic Render Driver
4   0.00011890  [7320] ***Output: ₩₩.₩DISPLAY34
5   0.00047080  [7320] Width = 2560 Height = 1440 Refresh = 32/1
6   0.00050320  [7320] ***Output: ₩₩.₩DISPLAY33
7   0.00063380  [7320] Width = 2560 Height = 1440 Refresh = 32/1
```

IDXGIAdapter (5)

-
  ```
  typedef struct DXGI_MODE_DESC {
    UINT Width;                            // Resolution width
    INT Height;                            // Resolution height
    XGI_RATIONAL RefreshRate;
    DXGI_FORMAT Format;                    // Display format
    XGI_MODE_SCANLINE_ORDER ScanlineOrdering;
           //Progressive vs. interlaced
    DXGI_MODE_SCALING Scaling;
          // How the image is stretched over the monitor.
  } DXGI_MODE_DESC;
  ```

-
  ```
  typedef struct DXGI_RATIONAL {
    UINT Numerator;
    UINT Denominator;
  } DXGI_RATIONAL;
  ```

IDXGIAdapter (6)

-
  ```
  typedef enum DXGI_MODE_SCANLINE_ORDER {
    DXGI_MODE_SCANLINE_ORDER_UNSPECIFIED       = 0,
    XGI_MODE_SCANLINE_ORDER_PROGRESSIVE        = 1,
    DXGI_MODE_SCANLINE_ORDER_UPPER_FIELD_FIRST = 2,
    DXGI_MODE_SCANLINE_ORDER_LOWER_FIELD_FIRST = 3
  } DXGI_MODE_SCANLINE_ORDER;
  ```

-
  ```
  typedef enum DXGI_MODE_SCALING {
    DXGI_MODE_SCALING_UNSPECIFIED  = 0,
    DXGI_MODE_SCALING_CENTERED     = 1,
    DXGI_MODE_SCALING_STRETCHED    = 2
  } DXGI_MODE_SCALING;
  ```

Residency (1)

- Residency
  - A complex game will use a lot of resources such as textures and 3D meshes, but many of these resources will not be needed by the GPU all the time. For example, if we imagine a game with an outdoor forest that has a large cave in it, the cave resources will not be needed until the player enters the cave, and when the player enters the cave, the forest resources will no longer be needed.
  - In Direct3D 12, applications manage resource residency (essentially, whether a resource is in GPU memory) by evicting resources from GPU memory and then making them resident on the GPU again as needed.
  -
```
HRESULT MakeResident(
            UINT           NumObjects,
    [in] ID3D12Pageable * const *ppObjects
);
```
    - It makes objects resident for the device.
    - **`ppObjects`**: A pointer to a memory block that contains an array of **`ID3D12Pageable`** interface pointers for the objects.

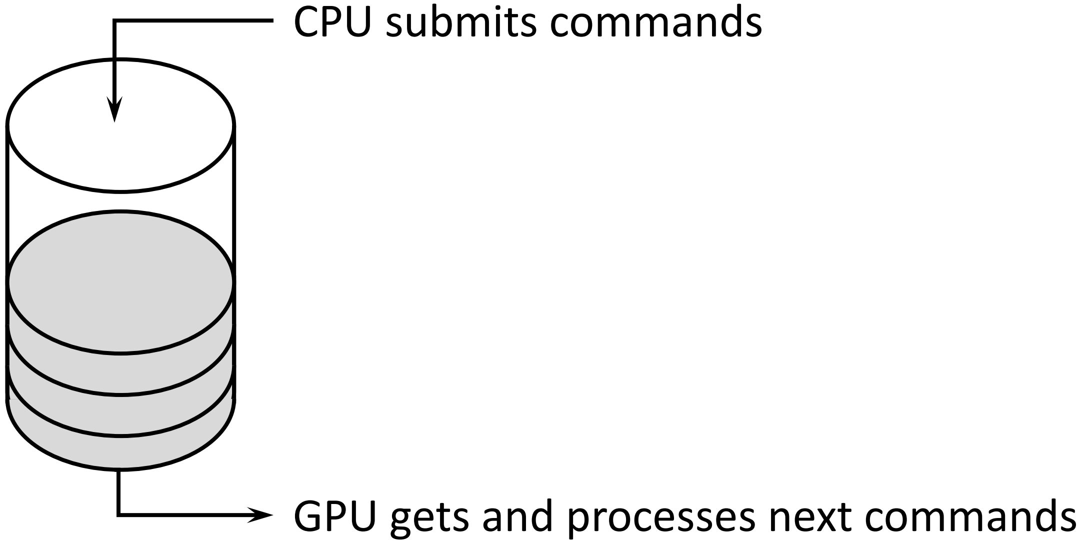
Residency (2)

-
  ```
  HRESULT Evict(
         UINT              NumObjects,
    [in] ID3D12Pageable * const *ppObjects
  );
  ```
    - It enables the page-out of data, which precludes GPU access of that data.

- In this lecture, for simplicity and due to our demos being small compared to a game, **we do not manage residency**.

The Command Queue & Command Lists (1)

- CPU/GPU work in parallel and sometimes need to be synchronized.
- For optimal performance, the goal is to keep both busy for as long as possible and minimize synchronizations.
- The command queue and command lists
  - The GPU has a command queue.
  - The CPU submits commands to the queue through the Direct3D API using command lists.



The Command Queue & Command Lists (2)

- Creating a command queue
  - **`ID3D12CommandQueue`** interface provides methods for submitting command lists, synchronizing command list execution, instrumenting the command queue, and updating resource tile mappings.
  - **`D3D12_COMMAND_QUEUE_DESC`** describes a command queue
    -
      ```
      typedef struct D3D12_COMMAND_QUEUE_DESC {
        D3D12_COMMAND_LIST_TYPE   Type;
        INT                       Priority;
        D3D12_COMMAND_QUEUE_FLAGS Flags;
        UINT                      NodeMask;
      } D3D12_COMMAND_QUEUE_DESC;
      ```
  - Using **`ID3D12Device::CreateCommandQueue`** with **`ID3D12CommandQueue`** and **`D3D12_COMMAND_QUEUE_DESC`**
    -
      ```
      HRESULT CreateCommandQueue(
        const D3D12_COMMAND_QUEUE_DESC *pDesc,
        REFIID riid, // GUID (globally unique identifier)
        void **ppCommandQueue
      );
      ```

The Command Queue & Command Lists (3)

```
// d3dApp.h
Microsoft::WRL::ComPtr<ID3D12CommandQueue> mCommandQueue;

// d3dApp.cpp - void D3DApp::CreateCommandObjects()
D3D12_COMMAND_QUEUE_DESC queueDesc = {};
queueDesc.Type = D3D12_COMMAND_LIST_TYPE_DIRECT;
queueDesc.Flags = D3D12_COMMAND_QUEUE_FLAG_NONE;

ThrowIfFailed(md3dDevice->CreateCommandQueue(
  &queueDesc, IID_PPV_ARGS(&mCommandQueue)));
// Microsoft::WRL::ComPtr<ID3D12Device> md3dDevice;
// ID3D12Device represents a virtual adapter;
//      it is used to create command allocators, command lists, command queues,
//      fences, resources, pipeline state objects, heaps, root signatures,
//      samplers, and many resource views.
// md3dDevice is created by D3D12CreateDevice in bool D3DApp::InitDirect3D()


// combaseapi.h
#define IID_PPV_ARGS(ppType) \
__uuidof(**(ppType)), IID_PPV_ARGS_Helper(ppType)
```

The Command Queue & Command Lists (4)

- Submitting an array of command lists for execution (**Adding the commands in the command lists to the queue**).

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

  - The command lists are executed in order starting with the first array element.

The Command Queue & Command Lists (5)

- A command list for graphics is represented by the **`ID3D12GraphicsCommandList`** interface which inherits from the **`ID3D12CommandList`** interface.
  - The **`ID3D12GraphicsCommandList`** interface has numerous methods for adding commands to the command list.
    - For example, the following code adds commands that set the viewport, clear the render target view, and issue a draw call:

    -
      ```
      // mCommandList pointer to ID3D12CommandList
      mCommandList->RSSetViewports(1, &mScreenViewport);
      mCommandList->ClearRenderTargetView(mBackBufferView,
          Colors::LightSteelBlue, 0, nullptr);
      ```
  - When we are done adding commands to a command list, we must indicate that we are finished recording commands by calling the **`ID3D12GraphicsCommandList::Close`** method:

    -
      ```
      // Done recording commands.
      mCommandList->Close();
      ```
  - **`ID3D12GraphicsCommandList`**
    - **`Reset`** → Add command lists → **`Close`** → **`ID3D12CommandQueue::ExecuteCommandLists`**

The Command Queue & Command Lists (6)

- Command allocator
  - Associated with a command list is a memory backing class called an **`ID3D12CommandAllocator`**.
  - As commands are recorded to the command list, they will actually be stored in the associated command allocator.
  - When a command list is executed via **`ID3D12CommandQueue::ExecuteCommandLists`**, the command queue will reference the commands in the allocator.
  - A command allocator is created from the ID3D12Device:

    -
      ```
      HRESULT ID3D12Device::CreateCommandAllocator(
        [in] D3D12_COMMAND_LIST_TYPE type,
        REFIID riid,
        [out] void **ppCommandAllocator);
      ```
      - **`type`**: The type of command lists that can be associated with this allocator. **`D3D12_COMMAND_LIST_TYPE_DIRECT`**, **`D3D12_COMMAND_LIST_TYPE_BUNDLE`**, …
      - **`riid`**: The COM ID of the ID3D12CommandAllocator interface.
      - **`ppCommandAllocator`**: Outputs a pointer to the created command allocator.

The Command Queue & Command Lists (7)

- Creating a command list
  - **`HRESULT ID3D12Device::CreateCommandList(`**
    ```
    HRESULT ID3D12Device::CreateCommandList(
      UINT nodeMask,
      D3D12_COMMAND_LIST_TYPE type,
      ID3D12CommandAllocator *pCommandAllocator,
      ID3D12PipelineState *pInitialState,
      REFIID riid,
      void **ppCommandList);
    ```
    - **`nodeMask`**: Set to 0 for single GPU system. Otherwise, the node mask identifies the physical GPU this command list is associated with.
    - **`pInitialState`**: It specifies the initial pipeline state of the command list.
- Resetting a command list back to its initial state (as if a new command list was just created).
  -
    ```
    HRESULT ID3D12CommandList::Reset(
      ID3D12CommandAllocator *pAllocator,
      ID3D12PipelineState *pInitialState);
    ```

The Command Queue & Command Lists (8)

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

The Command Queue & Command Lists (9)

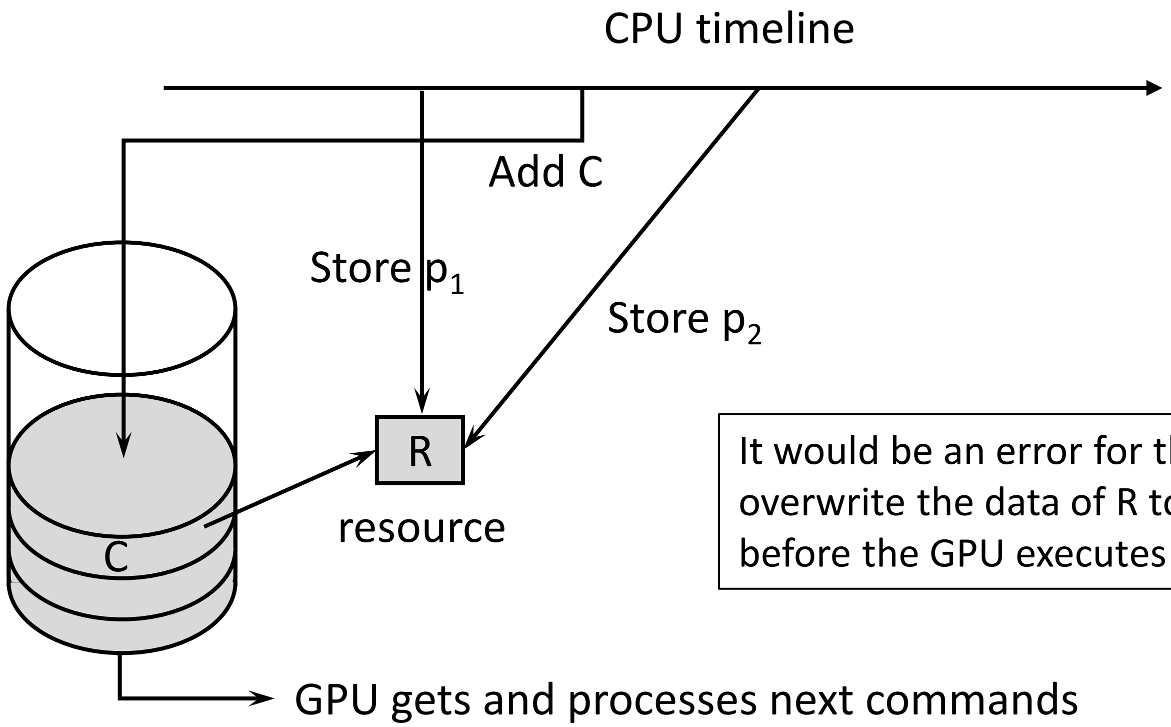
- Command queue processing
  - Members
    - `Microsoft::WRL::ComPtr<ID3D12Device> md3dDevice;`
    - `Microsoft::WRL::ComPtr<ID3D12CommandQueue> mCommandQueue;`
    - `Microsoft::WRL::ComPtr<ID3D12CommandAllocator> mDirectCmdListAlloc;`
    - `Microsoft::WRL::ComPtr<ID3D12GraphicsCommandList> mCommandList;`
  - Initialization
    - `D3D12CreateDevice`
    - `ID3D12Device::CreateCommandQueue`
    - `ID3D12Device::CreateCommandAllocator`
    - `ID3D12Device::CreateCommandList`
    - `ID3D12GraphicsCommandList::Close`

The Command Queue & Command Lists (10)

- Draw
  - **`ID3D12CommandAllocator::Reset`**
  - **`ID3D12GraphicsCommandList::Reset(ID3D12CommandAllocator.Get(), nullptr);`**
  - **`ID3D12GraphicsCommandList (Adding commanding)`**
    - **`ID3D12GraphicsCommandList::ResourceBarrier`**
    - **`ID3D12GraphicsCommandList::RSSetViewports`**
    - …
  - **`ID3D12GraphicsCommandList::Close`**
  - **`ID3D12CommandQueue::ExecuteCommandLists`**
  - **`ID3D12CommandQueue::Signal`**

CPU/GPU Synchronization

- CPU/GPU synchronization



It would be an error for the CPU to continue on and overwrite the data of R to store a new position p_2 before the GPU executes the draw command C.

CPU/GPU Synchronization: Fence (1)

- Using fence
  - One solution to previous situation is to force the CPU to wait until the GPU has finished processing all the commands in the queue up to a specified fence point.
  - We call this flushing the command queue. We can do this using a fence.
  - A fence is represented by the **`ID3D12Fence`** interface and is used to synchronize the GPU and CPU.
  - A fence object can be created with the following method:
    - **`HRESULT ID3D12Device::CreateFence(UINT64 InitialValue, D3D12_FENCE_FLAGS Flags, REFIID riid, void **ppFence);`**
    - **`// Microsoft::WRL::ComPtr<ID3D12Fence> mFence;`**

    - A fence object maintains a **`UINT64`** value, which is just an integer to identify a fence point in time. We start at value zero and every time we need to mark a new fence point, we just increment the integer.

CPU/GPU Synchronization: Fence (2)

This code shows how we can use a fence to flush the command queue.

```
// UINT64 mCurrentFence = 0;
void D3DApp::FlushCommandQueue() {
// Advance the fence value to mark commands up to this fence point.
   mCurrentFence++;
// Add an instruction to the command queue to set a new fence point. Because we
// are on the GPU timeline, the new fence point won't be set until the GPU finishes
// processing all the commands prior to this Signal().
  ThrowIfFailed(mCommandQueue->Signal(mFence.Get(), mCurrentFence));

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

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

Resource Transitions (1)

- Resource transitions
  - To implement common rendering effects, it is common for the GPU to write to a resource R in one step, and then, in a later step, read from the resource R. **However, it would be a resource hazard to read from a resource if the GPU has not finished writing to it or has not started writing at all.**
  - To solve this problem, Direct3D associates a state to resources. Resources are in a default state when they are created, and it is up to the application to tell Direct3D any state transitions. This enables the GPU to do any work it needs to do to make the transition and prevent resource hazards.

    -
      ```
      void ID3D12GraphicsCommandList::ResourceBarrier(
        [in] UINT NumBarriers,
        [in] const D3D12_RESOURCE_BARRIER *pBarriers
      );
      ```
      - This notifies the driver that it needs to synchronize multiple accesses to resources.
      - `NumBarriers`: The number of submitted barrier descriptions.
      - `pBarriers`: Pointer to an array of barrier descriptions.

Resource Transitions (2)

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

-
  ```
  struct CD3DX12_RESOURCE_BARRIER : public D3D12_RESOURCE_BARRIER {
     CD3DX12_RESOURCE_BARRIER();
  // …
     CD3DX12_RESOURCE_BARRIER static inline Transition
       (ID3D12Resource* pResource,
        D3D12_RESOURCE_STATES stateBefore,
        D3D12_RESOURCE_STATES stateAfter,
        UINT subresource = D3D12_RESOURCE_BARRIER_ALL_SUBRESOURCES,
        D3D12_RESOURCE_BARRIER_FLAGS flags =
           D3D12_RESOURCE_BARRIER_FLAG_NONE);
  // …
  };
  ```
  - A helper structure to enable easy initialization of a **D3D12_RESOURCE_BARRIER** structure.

Multithreading with Commands

- Multithreading with commands
  - Direct3D 12 was designed for efficient multithreading. The command list design is one way Direct3D takes advantage of multithreading.
  - For large scenes with lots of objects, building the command list to draw the entire scene can take CPU time. So the idea is to build command lists in parallel.
  - For example, you might spawn four threads, each responsible for building a command list to draw 25% of the scene objects.

Initializing Direct3D

- Initializing Direct3D can be broken down into the following steps:
    - 1. Create the **`ID3D12Device`** using the **`D3D12CreateDevice`** function.
    - 2. Create an **`ID3D12Fence`** object and query descriptor sizes.
    - 3. Check 4X MSAA quality level support.
    - 4. Create the command queue, command list allocator, and main command list.
    - 5. Describe and create the swap chain.
    - 6. Create the descriptor heaps the application requires.
    - 7. Resize the back buffer and create a render target view to the back buffer.
    - 8. Create the depth/stencil buffer and its associated depth/stencil view.
    - 9. Set the viewport and scissor rectangles.

Interface Pointers

```
// d3dApp.h
// Interface pointers
class D3DApp { // …
   Microsoft::WRL::ComPtr<IDXGIFactory4> mdxgiFactory;
   Microsoft::WRL::ComPtr<IDXGISwapChain> mSwapChain;
   Microsoft::WRL::ComPtr<ID3D12Device> md3dDevice;

   Microsoft::WRL::ComPtr<ID3D12Fence> mFence;
   UINT64 mCurrentFence = 0;

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

   static const int SwapChainBufferCount = 2;
   int mCurrBackBuffer = 0;
   Microsoft::WRL::ComPtr<ID3D12Resource>
      mSwapChainBuffer[SwapChainBufferCount];
   Microsoft::WRL::ComPtr<ID3D12Resource> mDepthStencilBuffer;

   Microsoft::WRL::ComPtr<ID3D12DescriptorHeap> mRtvHeap; // Render Target View
   Microsoft::WRL::ComPtr<ID3D12DescriptorHeap> mDsvHeap; // Depth Stencil View
//…
};
```

Create the Device (1)

- Create the device
  - Initializing Direct3D begins by creating the Direct3D 12 device (**`ID3D12Device`**).
  - The device represents a display adapter. Usually, the display adapter is a physical piece of3D hardware (e.g., graphics card); however, a system can also have a software display adapter that emulates 3D hardware functionality (e.g., the WARP adapter).

    -
      ```
      HRESULT D3D12CreateDevice(
        [in, optional]  IUnknown *pAdapter,
               D3D_FEATURE_LEVEL MinimumFeatureLevel,
        [in]  REFIID riid,
        [out, optional] void **ppDevice
      );
      ```

Create the Device (2)

```
// d2dApp.cpp
bool D3DApp::InitDirect3D() {
#if defined(DEBUG) || defined(_DEBUG)       // Enable the D3D12 debug layer.
{   ComPtr<ID3D12Debug> debugController;
    ThrowIfFailed(D3D12GetDebugInterface(IID_PPV_ARGS(&debugController)));
    debugController->EnableDebugLayer();    }
#endif
    ThrowIfFailed(CreateDXGIFactory1(IID_PPV_ARGS(&mdxgiFactory)));
    // Try to create a hardware device.
    HRESULT hardwareResult = D3D12CreateDevice(
        nullptr,                // default adapter
        D3D_FEATURE_LEVEL_11_0,
        IID_PPV_ARGS(&md3dDevice));
    // Fallback to WARP device.
    if(FAILED(hardwareResult)) {
      ComPtr<IDXGIAdapter> pWarpAdapter;
      ThrowIfFailed(mdxgiFactory->EnumWarpAdapter(IID_PPV_ARGS(&pWarpAdapter)));
      ThrowIfFailed(D3D12CreateDevice(
           pWarpAdapter.Get(),
           D3D_FEATURE_LEVEL_11_0,
           IID_PPV_ARGS(&md3dDevice)));
    }
     // …
}
```

WARP(Windows Advanced Rasterization Platform): Removing the need for custom software rasterizers, enabling maximum performance from graphics hardware, enabling rendering when Direct3D hardware is not available, …

Create the Fence & Descriptor Sizes (1)

-
  ```
  HRESULT ID3D12Device::CreateFence(
              UINT64             InitialValue,
              D3D12_FENCE_FLAGS Flags,
              REFIID             riid,
        [out] void               **ppFence
  );
  ```
  - This creates a fence object.

  -
    ```
    typedef enum D3D12_FENCE_FLAGS {
      D3D12_FENCE_FLAG_NONE = 0,
      D3D12_FENCE_FLAG_SHARED = 0x1,
      D3D12_FENCE_FLAG_SHARED_CROSS_ADAPTER = 0x2,
      D3D12_FENCE_FLAG_NON_MONITORED = 0x4
    } ;
    ```
    - This specifies fence options.

Create the Fence & Descriptor Sizes (2)

- **`UINT ID3D12Device::GetDescriptorHandleIncrementSize(`**
  `[in] D3D12_DESCRIPTOR_HEAP_TYPE DescriptorHeapType`
  `);`
  - This gets the size of the handle increment for the given type of descriptor heap.
  - **typedef enum D3D12_DESCRIPTOR_HEAP_TYPE {**

```
typedef enum D3D12_DESCRIPTOR_HEAP_TYPE {
  D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV = 0,
  D3D12_DESCRIPTOR_HEAP_TYPE_SAMPLER,
  D3D12_DESCRIPTOR_HEAP_TYPE_RTV,
  D3D12_DESCRIPTOR_HEAP_TYPE_DSV,
  D3D12_DESCRIPTOR_HEAP_TYPE_NUM_TYPES
} ;
```

    - This specifies a type of descriptor heap.

Create the Fence & Descriptor Sizes (3)

```
// d2dApp.cpp
bool D3DApp::InitDirect3D() {
// …
ThrowIfFailed(md3dDevice->CreateFence(0, D3D12_FENCE_FLAG_NONE,
    IID_PPV_ARGS(&mFence)));

mRtvDescriptorSize = md3dDevice->
   GetDescriptorHandleIncrementSize(D3D12_DESCRIPTOR_HEAP_TYPE_RTV);
mDsvDescriptorSize = md3dDevice->
   GetDescriptorHandleIncrementSize(D3D12_DESCRIPTOR_HEAP_TYPE_DSV);
mCbvSrvUavDescriptorSize = md3dDevice->
   GetDescriptorHandleIncrementSize(D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV);
   // constant-buffer, shader-resource, and unordered-access views.
// …
}
```

Check 4x MSAA Quality Support

```
// d2dApp.cpp
bool D3DApp::InitDirect3D() {
// …
// Multisample anti-aliasing (MSAA)
    D3D12_FEATURE_DATA_MULTISAMPLE_QUALITY_LEVELS msQualityLevels;
    msQualityLevels.Format = mBackBufferFormat;
    msQualityLevels.SampleCount = 4;
    msQualityLevels.Flags = D3D12_MULTISAMPLE_QUALITY_LEVELS_FLAG_NONE;
    msQualityLevels.NumQualityLevels = 0;
    ThrowIfFailed(md3dDevice->CheckFeatureSupport(
        D3D12_FEATURE_MULTISAMPLE_QUALITY_LEVELS,
        &msQualityLevels,
        sizeof(msQualityLevels)));

     m4xMsaaQuality = msQualityLevels.NumQualityLevels;
    assert(m4xMsaaQuality > 0 && "Unexpected MSAA quality level.");
// …
}
```

Create Command Queue & Command List

```
// d2dApp.cpp
void D3DApp::CreateCommandObjects() {
    D3D12_COMMAND_QUEUE_DESC queueDesc = {};
    queueDesc.Type = D3D12_COMMAND_LIST_TYPE_DIRECT;
    queueDesc.Flags = D3D12_COMMAND_QUEUE_FLAG_NONE;
    ThrowIfFailed(md3dDevice->CreateCommandQueue(&queueDesc,
        IID_PPV_ARGS(&mCommandQueue)));

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

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

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

Describe & Crate the Swap Chain (1)

```
// d2dApp.cpp
void D3DApp::CreateSwapChain() {
    // Release the previous swapchain we will be recreating.
    mSwapChain.Reset();
    DXGI_SWAP_CHAIN_DESC sd;
    sd.BufferDesc.Width = mClientWidth;
    sd.BufferDesc.Height = mClientHeight;
    sd.BufferDesc.RefreshRate.Numerator = 60;
    sd.BufferDesc.RefreshRate.Denominator = 1;
    sd.BufferDesc.Format = mBackBufferFormat;
    sd.BufferDesc.ScanlineOrdering = DXGI_MODE_SCANLINE_ORDER_UNSPECIFIED;
    sd.BufferDesc.Scaling = DXGI_MODE_SCALING_UNSPECIFIED;
    sd.SampleDesc.Count = m4xMsaaState ? 4 : 1;
    sd.SampleDesc.Quality = m4xMsaaState ? (m4xMsaaQuality - 1) : 0;
    sd.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;
    sd.BufferCount = SwapChainBufferCount;
    sd.OutputWindow = mhMainWnd;     sd.Windowed = true;
    sd.SwapEffect = DXGI_SWAP_EFFECT_FLIP_DISCARD;
    sd.Flags = DXGI_SWAP_CHAIN_FLAG_ALLOW_MODE_SWITCH;

    // Note: Swap chain uses queue to perform flush.
    ThrowIfFailed(mdxgiFactory->CreateSwapChain(
       mCommandQueue.Get(), &sd, mSwapChain.GetAddressOf()));
}
```

Describe & Crate the Swap Chain (2)

```
• typedef struct DXGI_SWAP_CHAIN_DESC {
    DXGI_MODE_DESC BufferDesc;
    DXGI_SAMPLE_DESC SampleDesc;
    DXGI_USAGE BufferUsage;
    UINT BufferCount;
    HWND OutputWindow;
    BOOL Windowed;
    DXGI_SWAP_EFFECT SwapEffect;
    UINT Flags;
  } DXGI_SWAP_CHAIN_DESC;
```

`HWND`: A handle to a window. **`typedef HANDLE HWND;`**
Handle is a variable that identifies an object; an indirect reference to an operating system resource.

Starting with Direct3D 11.1, we (Microsoft) recommend not to use **`CreateSwapChain`** anymore to create a swap chain. Instead, use **`CreateSwapChainForHwnd`**, **`CreateSwapChainForCoreWindow`**, or **`CreateSwapChainForComposition`** depending on how you want to create the swap chain.

Create the Descriptor Heaps (1)

- Create the descriptor heaps
  - We need **`SwapChainBufferCount`** many render target views (RTVs) to describe the buffer resources in the swap chain we will render into, and one depth/stencil view (DSV) to describe the depth/stencil buffer resource for depth testing. Therefore, we need a heap for storing **`SwapChainBufferCount`** RTVs, and we need a heap for storing one DSV.

Create the Descriptor Heaps (2)

-
  ```
  HRESULT ID3D12Device::CreateDescriptorHeap(
    [in] const D3D12_DESCRIPTOR_HEAP_DESC
                 *pDescriptorHeapDesc,
         REFIID riid,
    [out] void **ppvHeap
  );
  ```
  - This creates a descriptor heap object.

  -
    ```
    typedef struct D3D12_DESCRIPTOR_HEAP_DESC {
      D3D12_DESCRIPTOR_HEAP_TYPE Type;
      UINT NumDescriptors;
      D3D12_DESCRIPTOR_HEAP_FLAGS Flags;
      UINT NodeMask;
    } D3D12_DESCRIPTOR_HEAP_DESC;
    ```
    - This describes the descriptor heap.

    -
      ```
      typedef enum D3D12_DESCRIPTOR_HEAP_FLAGS {
        D3D12_DESCRIPTOR_HEAP_FLAG_NONE = 0,
        D3D12_DESCRIPTOR_HEAP_FLAG_SHADER_VISIBLE = 0x1
      };
      ```
    - **NodeMask**: For single-adapter operation, set this to zero. If there are multiple adapter nodes, set a bit to identify the node (one of the device's physical adapters) to which the descriptor heap applies.

Create the Descriptor Heaps (3)

```
// d2dApp.cpp
void D3DApp::CreateRtvAndDsvDescriptorHeaps() {
    D3D12_DESCRIPTOR_HEAP_DESC rtvHeapDesc;
    rtvHeapDesc.NumDescriptors = SwapChainBufferCount;
    rtvHeapDesc.Type = D3D12_DESCRIPTOR_HEAP_TYPE_RTV;
    rtvHeapDesc.Flags = D3D12_DESCRIPTOR_HEAP_FLAG_NONE;
    rtvHeapDesc.NodeMask = 0;
    ThrowIfFailed(md3dDevice->CreateDescriptorHeap(
        &rtvHeapDesc, IID_PPV_ARGS(mRtvHeap.GetAddressOf())));

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

Create the Descriptor Heaps (4)

```
// d2dApp.cpp
ID3D12Resource* D3DApp::CurrentBackBuffer()const {
    return mSwapChainBuffer[mCurrBackBuffer].Get();
} // It returns an ID3D12Resource to the current back buffer in the swap chain.

D3D12_CPU_DESCRIPTOR_HANDLE D3DApp::CurrentBackBufferView()const {
    return CD3DX12_CPU_DESCRIPTOR_HANDLE(
        mRtvHeap->GetCPUDescriptorHandleForHeapStart(),
        mCurrBackBuffer,
        mRtvDescriptorSize);
} // It returns the RTV (render target view) to the current back buffer.
// CD3DX12_CPU_DESCRIPTOR_HANDLE: A helper structure to enable easy
// initialization of a D3D12_CPU_DESCRIPTOR_HANDLE structure.

D3D12_CPU_DESCRIPTOR_HANDLE D3DApp::DepthStencilView()const {
    return mDsvHeap->GetCPUDescriptorHandleForHeapStart();
} // It returns the DSV (depth/stencil view) to the main depth/stencil buffer.
```

Kyung Hee University
nize@khu.ac.kr

Create the Render Target View

- Create the render target view for accessing resource data

  -
    ```
    void ID3D12Device::CreateRenderTargetView(
      [in, optional] ID3D12Resource *pResource,
      [in, optional] const D3D12_RENDER_TARGET_VIEW_DESC *pDesc,
      [in] D3D12_CPU_DESCRIPTOR_HANDLE DestDescriptor
    );
    ```

```
void D3DApp::OnResize() { // …
    // Resize the swap chain.
    ThrowIfFailed(mSwapChain->ResizeBuffers(
        SwapChainBufferCount,
        mClientWidth, mClientHeight,
        mBackBufferFormat,
        DXGI_SWAP_CHAIN_FLAG_ALLOW_MODE_SWITCH));
    mCurrBackBuffer = 0;
    CD3DX12_CPU_DESCRIPTOR_HANDLE rtvHeapHandle(
        mRtvHeap->GetCPUDescriptorHandleForHeapStart());
    for (UINT i = 0; i < SwapChainBufferCount; i++) {
        ThrowIfFailed(mSwapChain->GetBuffer(i, IID_PPV_ARGS(&mSwapChainBuffer[i])));
        md3dDevice->CreateRenderTargetView(mSwapChainBuffer[i].Get(), nullptr,
            rtvHeapHandle);
        rtvHeapHandle.Offset(1, mRtvDescriptorSize);
    }
// …
}
```

Create the Depth/Stencil Buffer & View (1)

- A depth buffer is a kind of GPU resource, so we create one by filling out a **`D3D12_RESOURCE_DESC`** structure describing the texture resource and then calling the **`ID3D12Device::CreateCommittedResource`** method.
  -
```
typedef struct D3D12_RESOURCE_DESC {
  D3D12_RESOURCE_DIMENSION Dimension;
  UINT64 Alignment;
  UINT64 Width;
  UINT Height;
  UINT16 DepthOrArraySize;
  UINT16 MipLevels;
  DXGI_FORMAT Format;
  DXGI_SAMPLE_DESC SampleDesc;
  D3D12_TEXTURE_LAYOUT Layout;
  D3D12_RESOURCE_FLAGS Flags;
} D3D12_RESOURCE_DESC;
```

MIP maps or pyramids are pre-calculated, optimized sequences of images, each of which is a progressively lower-resolution representation of the previous.

Create the Depth/Stencil Buffer & View (2)

- D3D12_RESOURCE_DIMENSION Dimension :
  - typedef enum D3D12_RESOURCE_DIMENSION {
    ```
      D3D12_RESOURCE_DIMENSION_UNKNOWN = 0,
      D3D12_RESOURCE_DIMENSION_BUFFER = 1,
      D3D12_RESOURCE_DIMENSION_TEXTURE1D = 2,
      D3D12_RESOURCE_DIMENSION_TEXTURE2D = 3,
      D3D12_RESOURCE_DIMENSION_TEXTURE3D = 4
    } ;
    ```
- DXGI_FORMAT Format: DXGI_FORMAT_R24G8_TYPELESS (Depth/Stencil)
- D3D12_RESOURCE_FLAGS Flags: D3D12_RESOURCE_FLAG_ALLOW_DEPTH_STENCIL

Create the Depth/Stencil Buffer & View (3)

```
void D3DApp::OnResize() { // …
   // Create the depth/stencil buffer and view.
   D3D12_RESOURCE_DESC depthStencilDesc;
   depthStencilDesc.Dimension = D3D12_RESOURCE_DIMENSION_TEXTURE2D;
   depthStencilDesc.Alignment = 0;
   depthStencilDesc.Width = mClientWidth;
   depthStencilDesc.Height = mClientHeight;
   depthStencilDesc.DepthOrArraySize = 1;
   depthStencilDesc.MipLevels = 1;

   depthStencilDesc.Format = DXGI_FORMAT_R24G8_TYPELESS;
   depthStencilDesc.SampleDesc.Count = m4xMsaaState ? 4 : 1;
   depthStencilDesc.SampleDesc.Quality = m4xMsaaState?(m4xMsaaQuality - 1) : 0;
   depthStencilDesc.Layout = D3D12_TEXTURE_LAYOUT_UNKNOWN;
   depthStencilDesc.Flags = D3D12_RESOURCE_FLAG_ALLOW_DEPTH_STENCIL;
```

Create the Depth/Stencil Buffer & View (4)

-
  ```
  HRESULT ID3D12Device::CreateCommittedResource(
    [in] const D3D12_HEAP_PROPERTIES *pHeapProperties,
    [in] D3D12_HEAP_FLAGS HeapFlags,
    [in] const D3D12_RESOURCE_DESC *pDesc,
    [in] D3D12_RESOURCE_STATES InitialResourceState,
    [in, optional] const D3D12_CLEAR_VALUE
          *pOptimizedClearValue,
    [in] REFIID riidResource,
    [out, optional] void **ppvResource
  );
  ```
  - This creates both a resource and an implicit heap.
  - **`const D3D12_CLEAR_VALUE *pOptimizedClearValue`**

    -
      ```
      typedef struct D3D12_CLEAR_VALUE {
        DXGI_FORMAT Format;
        union {
          FLOAT                    Color[4];
          D3D12_DEPTH_STENCIL_VALUE DepthStencil;
        };
      } D3D12_CLEAR_VALUE;
      ```
      - This describes a value used to optimize clear operations for a particular resource.

      -
        ```
        typedef struct D3D12_DEPTH_STENCIL_VALUE {
          FLOAT Depth;
          UINT8 Stencil;
        } D3D12_DEPTH_STENCIL_VALUE;
        ```

Kyung Hee University
nize@khu.ac.kr

Create the Depth/Stencil Buffer & View (5)

-
  ```
  Void ID3D12Device::CreateDepthStencilView(
    [in, optional] ID3D12Resource *pResource,
    [in, optional] const D3D12_DEPTH_STENCIL_VIEW_DESC
                                 *pDesc,
    [in] D3D12_CPU_DESCRIPTOR_HANDLE DestDescriptor
  );
  ```
  - This creates a depth-stencil view for accessing resource data.

Kyung Hee University
nize@khu.ac.kr

Create the Depth/Stencil Buffer & View (6)

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

// Create descriptor to mip level 0 of entire resource using the format of the resource.
    D3D12_DEPTH_STENCIL_VIEW_DESC dsvDesc;
    dsvDesc.Flags = D3D12_DSV_FLAG_NONE;
    dsvDesc.ViewDimension = D3D12_DSV_DIMENSION_TEXTURE2D;
    dsvDesc.Format = mDepthStencilFormat; dsvDesc.Texture2D.MipSlice = 0;
    md3dDevice->CreateDepthStencilView(mDepthStencilBuffer.Get(), &dsvDesc,
       DepthStencilView());

// Transition the resource from its initial state to be used as a depth buffer.
    mCommandList->ResourceBarrier(1,
&CD3DX12_RESOURCE_BARRIER::Transition(mDepthStencilBuffer.Get(),
       D3D12_RESOURCE_STATE_COMMON, D3D12_RESOURCE_STATE_DEPTH_WRITE));
// …
}
```

Set the Viewport (1)

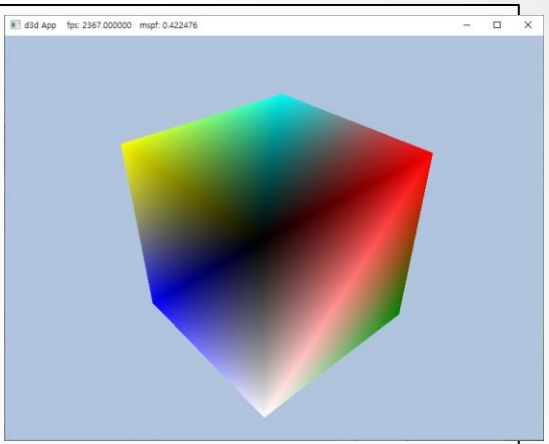
- Setting the viewport
  - Usually we like to draw the 3D scene to the entire back buffer, where the back buffer size corresponds to the entire screen (full-screen mode) or the entire client area of a window. However, sometimes we only want to draw the 3D scene into a subrectangle of the back buffer.
  - Structure for describing the dimensions of a viewport.

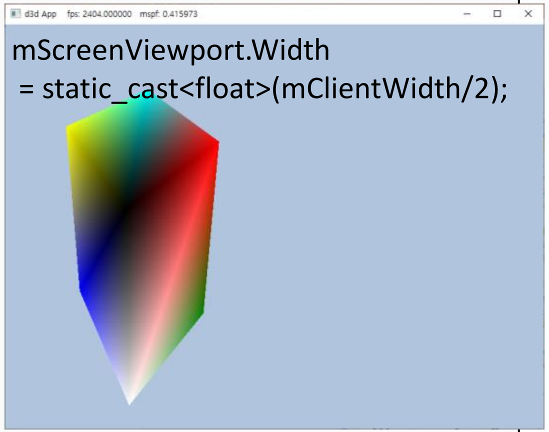
    -
      ```
      typedef struct D3D12_VIEWPORT {
        FLOAT TopLeftX;
        FLOAT TopLeftY;
        FLOAT Width;
        FLOAT Height;
        FLOAT MinDepth;
        FLOAT MaxDepth;
      } D3D12_VIEWPORT;
      ```
  - Binding an array of viewports to the rasterizer stage of the pipeline.

    -
      ```
      void ID3D11DeviceContext::RSSetViewports(
        [in]           UINT NumViewports,
        [in, optional] const D3D11_VIEWPORT *pViewports
      )
      ```

Set the Viewport (2)

```
// d3dApp.h
class D3DApp {// …
   virtual void Draw(const GameTimer& gt)=0;
// …
};
// d3dApp.cpp
void D3DApp::OnResize() { // …
   mScreenViewport.TopLeftX = 0;
   mScreenViewport.TopLeftY = 0;
   mScreenViewport.Width    = static_cast<float>(mClientWidth);
   mScreenViewport.Height   = static_cast<float>(mClientHeight);
   mScreenViewport.MinDepth = 0.0f;
   mScreenViewport.MaxDepth = 1.0f;
// …
}
// initDirect3DApp.cpp
void InitDirect3DApp::Draw(const GameTimer& gt) { // …
   mCommandList->RSSetViewports(1, &mScreenViewport);
// …
}
```





mScreenViewport.Width
= static_cast<float>(mClientWidth/2);

Kyung Hee University
nize@khu.ac.kr

Set the Scissor Rectangles (1)

- Set the scissor rectangles
  - We can define a scissor rectangle relative to the back buffer such that pixels outside this rectangle are culled (i.e., not rasterized to the back buffer). This can be used for optimizations.
  - RECT structure

    -
      ```
      typedef struct tagRECT {
        LONG  left;
        LONG  top;
        LONG  right;
        LONG  bottom;
      } RECT;
      ```
  - Binding an array of scissor rectangles to the rasterizer stage.

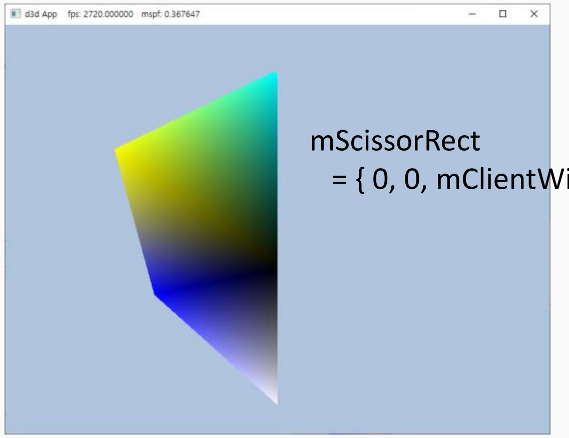
    -
      ```
      void RSSetScissorRects(
        [in]            UINT             NumRects,
        [in, optional]  const D3D11_RECT *pRects
      );
      ```

The viewport specifies how the normalized device coordinates are transformed into the pixel coordinates of the framebuffer. Scissor is the area where you can render.

Set the Scissor Rectangles (2)

```
// d3dApp.cpp
void D3DApp::OnResize() { // …
   mScissorRect = { 0, 0, mClientWidth, mClientHeight };
}

// initDirect3DApp.cpp
void InitDirect3DApp::Draw(const GameTimer& gt) { // …
   mCommandList->RSSetScissorRects(1, &mScissorRect);
// …
}
```



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

The Performance Timer (1)

- The performance timer
  - For accurate time measurements, we use the performance timer (or performance counter).
    - **`#include <windows.h>.`**
  - The performance timer measures time in units called counts.

    -
      ```
      BOOL QueryPerformanceCounter(
      [out] LARGE_INTEGER *lpPerformanceCount);
      ```

    -
      ```
      __int64 currTime;
      QueryPerformanceCounter((LARGE_INTEGER*)&currTime);
      ```
  - Get the frequency (counts per second) of the performance timer

    -
      ```
      BOOL QueryPerformanceFrequency(
        [out] LARGE_INTEGER *lpFrequency);
      ```

    -
      ```
      __int64 countsPerSec;
      QueryPerformanceFrequency((LARGE_INTEGER*)&countsPerSec);
      ```

The Performance Timer (2)

- The number of seconds (or fractions of a second) per count is just the reciprocal of the counts per second:
  - **`mSecondsPerCount = 1.0 / (double)countsPerSec;`**
- To convert a time reading **`valueInCounts`** to seconds, we just multiply it by the conversion factor **`mSecondsPerCount`**
  - **`valueInSecs = valueInCounts * mSecondsPerCount;`**
-
```
__int64 A = 0;
QueryPerformanceCounter((LARGE_INTEGER*)&A);
/* Do work */
__int64 B = 0;
QueryPerformanceCounter((LARGE_INTEGER*)&B);
Secs = (B-A)*mSecondsPerCount;
```

MSDN has the following remark about **`QueryPerformanceCounter`**: "On a multiprocessor computer, it should not matter which processor is called. However, you can get different results on different processors due to bugs in the basic input/output system (BIOS) or the hardware abstraction layer (HAL)." You can use the **`SetThreadAffinityMask`** function so that the main application thread does not get switch to another processor.

GameTimer Class (1)

```
// Game timer class // GameTimer.h
class GameTimer {
public:
    GameTimer();

    float TotalTime()const; // in seconds
    float DeltaTime()const; // in seconds
    void Reset(); // Call before message loop.
    void Start(); // Call when unpaused.
    void Stop();  // Call when paused.
    void Tick();  // Call every frame.
private:
    double mSecondsPerCount;
    double mDeltaTime;

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

    bool mStopped;
};
```

GameTimer Class (2)

```
// Game timer class // GameTimer.cpp

GameTimer::GameTimer()
: mSecondsPerCount(0.0), mDeltaTime(-1.0), mBaseTime(0),
  mPausedTime(0), mPrevTime(0), mCurrTime(0), mStopped(false)
{
    __int64 countsPerSec;
    QueryPerformanceFrequency((LARGE_INTEGER*)&countsPerSec);
    mSecondsPerCount = 1.0 / (double)countsPerSec;
}
```

GameTimer Class (3)

```
// Returns the total time elapsed since Reset() was called, NOT counting any
// time when the clock is stopped.
float GameTimer::TotalTime()const {
    // If we are stopped, do not count the time that has passed since we stopped.
    // Moreover, if we previously already had a pause, the distance
    // mStopTime - mBaseTime includes paused time, which we do not want to count.
    // To correct this, we can subtract the paused time from mStopTime:
    //
    //                     |<--paused time-->|
    // ----*---------------*-----------------*------------*------------*------> time
    //  mBaseTime       mStopTime        startTime     mStopTime    mCurrTime
    if( mStopped ) {
        return (float)(((mStopTime - mPausedTime)-mBaseTime)*mSecondsPerCount); }
    // The distance mCurrTime - mBaseTime includes paused time,
    // which we do not want to count.  To correct this, we can subtract
    // the paused time from mCurrTime:
    //  (mCurrTime - mPausedTime) - mBaseTime
    //
    //                     |<--paused time-->|
    // ----*---------------*-----------------*------------*------> time
    //  mBaseTime       mStopTime        startTime     mCurrTime
    else {
        return (float)(((mCurrTime-mPausedTime)-mBaseTime)*mSecondsPerCount); }
}
```

GameTimer Class (4)

```
float GameTimer::DeltaTime()const {
    return (float)mDeltaTime;
}
void GameTimer::Reset() {
    __int64 currTime;
    QueryPerformanceCounter((LARGE_INTEGER*)&currTime);

    mBaseTime = currTime;              mPrevTime = currTime;
    mStopTime = 0;                     mStopped  = false;
}
void GameTimer::Start() {
    __int64 startTime;
    QueryPerformanceCounter((LARGE_INTEGER*)&startTime);
    // Accumulate the time elapsed between stop and start pairs.
    //                    |<-------d------->|
    // ----*---------------*-----------------*------------> time
    //  mBaseTime       mStopTime        startTime
    if( mStopped ) {
        mPausedTime += (startTime - mStopTime);

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

GameTimer Class (5)

```
void GameTimer::Stop() {
    if( !mStopped ) {
        __int64 currTime;
        QueryPerformanceCounter((LARGE_INTEGER*)&currTime);
        mStopTime = currTime;                   mStopped  = true;
    }
}
void GameTimer::Tick() {
    if( mStopped ) { mDeltaTime = 0.0;      return; }
    __int64 currTime;
    QueryPerformanceCounter((LARGE_INTEGER*)&currTime);
    mCurrTime = currTime;
    // Time difference between this frame and the previous.
    mDeltaTime = (mCurrTime - mPrevTime)*mSecondsPerCount;
    // Prepare for next frame.
    mPrevTime = mCurrTime;
    // Force nonnegative.  The DXSDK's CDXUTTimer mentions that if the
    // processor goes into a power save mode or we get shuffled to another
    // processor, then mDeltaTime can be negative.
    if(mDeltaTime < 0.0) mDeltaTime = 0.0;
}
```

Time Elapsed Between Frames

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

Framework

- [VS Project] Init Direct3D
    - Header Files
        - d3dApp.h
            - **`D2DApp`** class
                - Functions for creating the main application window, running the application message loop, handling window messages, and initializing Direct3D.
        - d3dUtil.h
            - Debug utilities
            - **`DxException`** class
            - **`struct Light, Texture, Material, MeshGeometry`**, …
        - GameTimer.h
    - Source Files
        - d3dApp.cpp
        - d3dUtil.cpp
        - GemeTimer.cpp
        - InitDirect3DApp.cpp

D3DApp Class (1)

- Abstract class
  - The base application class
  - Functions for creating the main application window, running the application message loop, handling window messages, and initializing Direct3D.
  - Singleton pattern
  - Application source code:

    -
      ```
      class InitDirect3DApp : public D3DApp {// ..
          virtual void OnResize()override;
      // ..
      };
      int WINAPI WinMain(/* … */) {
      try {
              InitDirect3DApp theApp(hInstance);
              if(!theApp.Initialize()) return 0;
              return theApp.Run();
          }
          catch(DxException& e)  {
              MessageBox(/* … */);
              return 0;
          }
      }
      ```

D3DApp Class (2)

```
// d2dApp.h
class D3DApp {
protected:
    D3DApp(HINSTANCE hInstance);
    D3DApp(const D3DApp& rhs) = delete;
    D3DApp& operator=(const D3DApp& rhs) = delete;
    virtual ~D3DApp();
public:
    static D3DApp* GetApp();
    HINSTANCE AppInst()const;
    HWND      MainWnd()const;
    float     AspectRatio()const;

    bool Get4xMsaaState()const;
    void Set4xMsaaState(bool value);

    int Run();
    virtual bool Initialize();
    virtual LRESULT MsgProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam);
```

D3DApp Class (3)

```
// d2dApp.h
protected:
     virtual void CreateRtvAndDsvDescriptorHeaps();
     virtual void OnResize();
     virtual void Update(const GameTimer& gt)=0;
     virtual void Draw(const GameTimer& gt)=0;

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

D3DApp Class (4)

```
// d2dApp.h
protected:
    bool InitMainWindow();
    bool InitDirect3D();
    void CreateCommandObjects();
    void CreateSwapChain();
    void FlushCommandQueue();

    ID3D12Resource* CurrentBackBuffer()const;
    D3D12_CPU_DESCRIPTOR_HANDLE CurrentBackBufferView()const;
    D3D12_CPU_DESCRIPTOR_HANDLE DepthStencilView()const;

    void CalculateFrameStats();
    void LogAdapters();
    void LogAdapterOutputs(IDXGIAdapter* adapter);
    void LogOutputDisplayModes(IDXGIOutput* output, DXGI_FORMAT format);
```

D3DApp Class (5)

```
// d2dApp.h
protected:
    static D3DApp* mApp;
    HINSTANCE mhAppInst = nullptr; // application instance handle
    HWND      mhMainWnd = nullptr; // main window handle
    bool      mAppPaused = false;  // is the application paused?
    bool      mMinimized = false;  // is the application minimized?
    bool      mMaximized = false;  // is the application maximized?
    bool      mResizing = false;   // are the resize bars being dragged?
    bool      mFullscreenState = false;// fullscreen enabled
    // Set true to use 4X MSAA (§4.1.8).  The default is false.
    bool      m4xMsaaState = false;    // 4X MSAA enabled
    UINT      m4xMsaaQuality = 0;      // quality level of 4X MSAA
```

D3DApp Class (6)

```
// d2dApp.h
    GameTimer mTimer;
    // Used to keep track of the "delta-time" and game time (§4.4).

    Microsoft::WRL::ComPtr<IDXGIFactory4> mdxgiFactory;
    Microsoft::WRL::ComPtr<IDXGISwapChain> mSwapChain;
    Microsoft::WRL::ComPtr<ID3D12Device> md3dDevice;
    Microsoft::WRL::ComPtr<ID3D12Fence> mFence;
    UINT64 mCurrentFence = 0;

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

    static const int SwapChainBufferCount = 2;      int mCurrBackBuffer = 0;
    Microsoft::WRL::ComPtr<ID3D12Resource>
         mSwapChainBuffer[SwapChainBufferCount];
    Microsoft::WRL::ComPtr<ID3D12Resource> mDepthStencilBuffer;
    Microsoft::WRL::ComPtr<ID3D12DescriptorHeap> mRtvHeap;
    Microsoft::WRL::ComPtr<ID3D12DescriptorHeap> mDsvHeap;
```

D3DApp Class (7)

```
// d2dApp.h
    D3D12_VIEWPORT mScreenViewport;
    D3D12_RECT mScissorRect;
    UINT mRtvDescriptorSize = 0;
    UINT mDsvDescriptorSize = 0;
    UINT mCbvSrvUavDescriptorSize = 0;

    // Derived class should set these in the derived constructor
    //     to customize starting values.
    std::wstring mMainWndCaption = L"d3d App";
    D3D_DRIVER_TYPE md3dDriverType = D3D_DRIVER_TYPE_HARDWARE;
    DXGI_FORMAT mBackBufferFormat = DXGI_FORMAT_R8G8B8A8_UNORM;
    DXGI_FORMAT mDepthStencilFormat = DXGI_FORMAT_D24_UNORM_S8_UINT;
    int mClientWidth = 800;
    int mClientHeight = 600;
};
```

D3DApp Class (8)

```
// d3dApp.cpp

bool D3DApp::Initialize() {
    if(!InitMainWindow())
        return false;

    if(!InitDirect3D())
        return false;

     // Do the initial resize code.
     OnResize();

    return true;
}
```

D3DApp Class (9)

```
// d3dApp.cpp
bool D3DApp::InitMainWindow() {
    WNDCLASS wc;
    wc.style         = CS_HREDRAW | CS_VREDRAW;
    //…
    if( !RegisterClass(&wc) ) {
        MessageBox(0, L"RegisterClass Failed.", 0, 0);
        return false;
    }
    // Compute window rectangle dimensions based on requested client area dimensions.
    RECT R = { 0, 0, mClientWidth, mClientHeight };
     AdjustWindowRect(&R, WS_OVERLAPPEDWINDOW, false);
    int width  = R.right - R.left; int height = R.bottom - R.top;

    mhMainWnd = CreateWindow(L"MainWnd", mMainWndCaption.c_str(),
        WS_OVERLAPPEDWINDOW, /* … */, width, height, 0, 0, mhAppInst, 0);
    if( !mhMainWnd )       {
        MessageBox(0, L"CreateWindow Failed.", 0, 0);
        return false;
    }
    ShowWindow(mhMainWnd, SW_SHOW);UpdateWindow(mhMainWnd);
    return true;
}
```

D3DApp Class (10)

```
// d3dApp.cpp
bool D3DApp::InitDirect3D() {
// …
    HRESULT hardwareResult = D3D12CreateDevice(nullptr,
        D3D_FEATURE_LEVEL_11_0, IID_PPV_ARGS(&md3dDevice));
    if(FAILED(hardwareResult))      {
        ComPtr<IDXGIAdapter> pWarpAdapter;
        ThrowIfFailed(mdxgiFactory->
            EnumWarpAdapter(IID_PPV_ARGS(&pWarpAdapter)));
        ThrowIfFailed(D3D12CreateDevice(pWarpAdapter.Get(),
            D3D_FEATURE_LEVEL_11_0, IID_PPV_ARGS(&md3dDevice)));
    }
    ThrowIfFailed(md3dDevice->CreateFence(0, D3D12_FENCE_FLAG_NONE,
        IID_PPV_ARGS(&mFence)));
    mRtvDescriptorSize = md3dDevice->
        GetDescriptorHandleIncrementSize(D3D12_DESCRIPTOR_HEAP_TYPE_RTV);
    mDsvDescriptorSize = md3dDevice->
        GetDescriptorHandleIncrementSize(D3D12_DESCRIPTOR_HEAP_TYPE_DSV);
    mCbvSrvUavDescriptorSize = md3dDevice->
       GetDescriptorHandleIncrementSize(D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV);
```

D3DApp Class (11)

```
// d3dApp.cpp
    D3D12_FEATURE_DATA_MULTISAMPLE_QUALITY_LEVELS msQualityLevels;
    msQualityLevels.Format = mBackBufferFormat;
    msQualityLevels.SampleCount = 4;
    msQualityLevels.Flags = D3D12_MULTISAMPLE_QUALITY_LEVELS_FLAG_NONE;
    msQualityLevels.NumQualityLevels = 0;
    ThrowIfFailed(md3dDevice->
        CheckFeatureSupport(D3D12_FEATURE_MULTISAMPLE_QUALITY_LEVELS,
        &msQualityLevels, sizeof(msQualityLevels)));
    m4xMsaaQuality = msQualityLevels.NumQualityLevels;

    // …
    CreateCommandObjects();
    CreateSwapChain();
    CreateRtvAndDsvDescriptorHeaps();

    return true;
}
```

D3DApp Class (12)

```
// d3dApp.cpp
int D3DApp::Run() {
    MSG msg = {0};
    mTimer.Reset();
    while(msg.message != WM_QUIT) {
        // If there are Window messages then process them.
        if(PeekMessage( &msg, 0, 0, 0, PM_REMOVE )) {
              TranslateMessage( &msg );
              DispatchMessage( &msg );
        }
        // Otherwise, do animation/game stuff.
        else {
            mTimer.Tick();
            if( !mAppPaused )         {
                CalculateFrameStats();
                Update(mTimer);
                Draw(mTimer);
            }
            else { Sleep(100); }
         }
     }
    return (int)msg.wParam;
}
```

D3DApp Class (13)

```
// d3dApp.cpp
LRESULT D3DApp::MsgProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam) {
    switch( msg ) {
    // WM_ACTIVATE is sent when the window is activated or deactivated.
    // We pause the game when the window is deactivated and unpause it
    // when it becomes active.
    case WM_ACTIVATE:
        if( LOWORD(wParam) == WA_INACTIVE ) {
            mAppPaused = true;
            mTimer.Stop();
        }
        else {
            mAppPaused = false;
            mTimer.Start();
        }
        return 0;
```

D3DApp Class (14)

```
// d3dApp.cpp - WM_SIZE is sent when the user resizes the window.
    case WM_SIZE: // Save the new client area dimensions.
        mClientWidth  = LOWORD(lParam);     mClientHeight = HIWORD(lParam);
        if( md3dDevice ) {
            if( wParam == SIZE_MINIMIZED ) {
                mAppPaused = true;   mMinimized = true;   mMaximized = false;
            }
            else if( wParam == SIZE_MAXIMIZED ) {
                mAppPaused = false;  mMinimized = false;   mMaximized = true;
                OnResize();
            }
            else if( wParam == SIZE_RESTORED ) {
                if( mMinimized ) { /* … */ } else if( mMaximized ) { /* … */ }
                else if( mResizing ) { }
                else { // API call such as SetWindowPos
                    OnResize();
                }
            }
        }
        return 0;
```

D3DApp Class (15)

```
// d3dApp.cpp
    // WM_EXITSIZEMOVE is sent when the user grabs the resize bars.
    case WM_ENTERSIZEMOVE:
        mAppPaused = true; mResizing  = true;        mTimer.Stop();
        return 0;

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

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

D3DApp Class (16)

```
// d3dApp.cpp
   // The WM_MENUCHAR message is sent when a menu is active
   // and the user presses a key that does not correspond
   // to any mnemonic or accelerator key.
   case WM_MENUCHAR:
       // Don't beep when we alt-enter.
       return MAKELRESULT(0, MNC_CLOSE);

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

D3DApp Class (17)

In this lecture, we use **GetAsyncKeyState** function for keyboard input.

```
SHORT GetAsyncKeyState( [in] int vKey );
```

Return value:

If the high-order bit is 1, the key is down; otherwise, it is up. (**0x8000**)

If the low-order bit is 1, the key is toggled. (**0x0001**)

```
// d3dApp.cpp
    case WM_LBUTTONDOWN:
    case WM_MBUTTONDOWN:
    case WM_RBUTTONDOWN:
        OnMouseDown(wParam, GET_X_LPARAM(lParam), GET_Y_LPARAM(lParam));
        return 0;
    case WM_LBUTTONUP:
    case WM_MBUTTONUP:
    case WM_RBUTTONUP:
        OnMouseUp(wParam, GET_X_LPARAM(lParam), GET_Y_LPARAM(lParam));
        return 0;
    case WM_MOUSEMOVE:
        OnMouseMove(wParam, GET_X_LPARAM(lParam), GET_Y_LPARAM(lParam));
        return 0;
     case WM_KEYUP:
         if(wParam == VK_ESCAPE) {
             PostQuitMessage(0);
         }
         else if((int)wParam == VK_F2)
             Set4xMsaaState(!m4xMsaaState);
         return 0;
    }
    return DefWindowProc(hwnd, msg, wParam, lParam);
}
```

```
// windowsx.h
#define GET_X_LPARAM(lp)  ((int)(short)LOWORD(lp))
#define GET_Y_LPARAM(lp)  ((int)(short)HIWORD(lp))
```

Application Class (1)

```
// InitDirect3DApp.cpp
#include "../../Common/d3dApp.h"
#include <DirectXColors.h>
using namespace DirectX;
class InitDirect3DApp : public D3DApp {
public:
    InitDirect3DApp(HINSTANCE hInstance);
    ~InitDirect3DApp();
    virtual bool Initialize()override;
private:
     virtual void OnResize()override;
     virtual void Update(const GameTimer& gt)override;
     virtual void Draw(const GameTimer& gt)override;
};
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE prevInstance,
    PSTR cmdLine, int showCmd) {
// Enable run-time memory check for debug builds. // …
     try { InitDirect3DApp theApp(hInstance);
             if(!theApp.Initialize()) return 0;
             return theApp.Run();     }
     catch(DxException& e) {
         MessageBox(nullptr, e.ToString().c_str(), L"HR Failed", MB_OK);
         return 0;     }
}
```

Application Class (2)

```
// InitDirect3DApp.cpp
InitDirect3DApp::InitDirect3DApp(HINSTANCE hInstance) : D3DApp(hInstance) { }
InitDirect3DApp::~InitDirect3DApp() { }
bool InitDirect3DApp::Initialize() {
    if(!D3DApp::Initialize()) return false;
    return true;
}

void InitDirect3DApp::OnResize() {
    D3DApp::OnResize();
}

void InitDirect3DApp::Update(const GameTimer& gt) {
}
```

Application Class (3)

```
// InitDirect3DApp.cpp
void InitDirect3DApp::Draw(const GameTimer& gt) {
    // Reuse the memory associated with command recording.
    // We can only reset when the associated command lists have finished
    // execution on the GPU.
    ThrowIfFailed(mDirectCmdListAlloc->Reset());

    // A command list can be reset after it has been added to the command queue
    // via ExecuteCommandList.
    // Reusing the command list reuses memory.
    ThrowIfFailed(mCommandList->Reset(mDirectCmdListAlloc.Get(), nullptr));

   // Indicate a state transition on the resource usage.
   mCommandList->ResourceBarrier(1,
      &CD3DX12_RESOURCE_BARRIER::Transition(CurrentBackBuffer(),
      D3D12_RESOURCE_STATE_PRESENT, D3D12_RESOURCE_STATE_RENDER_TARGET));
  // Set the viewport and scissor rect.
  // This needs to be reset whenever the command list is reset.
  mCommandList->RSSetViewports(1, &mScreenViewport);
  mCommandList->RSSetScissorRects(1, &mScissorRect);
```

Application Class (4)

```
// InitDirect3DApp.cpp
    // Clear the back buffer and depth buffer.
    mCommandList->ClearRenderTargetView(CurrentBackBufferView(),
        Colors::LightSteelBlue,
        0, nullptr);

    mCommandList->ClearDepthStencilView(DepthStencilView(),
        D3D12_CLEAR_FLAG_DEPTH |
        D3D12_CLEAR_FLAG_STENCIL, 1.0f, 0, 0, nullptr);

    // Specify the buffers we are going to render to.
    mCommandList->OMSetRenderTargets(1, &CurrentBackBufferView(), true,
        &DepthStencilView());

    // Indicate a state transition on the resource usage.
    mCommandList->ResourceBarrier(1,
        &CD3DX12_RESOURCE_BARRIER::Transition(CurrentBackBuffer(),
        D3D12_RESOURCE_STATE_RENDER_TARGET, D3D12_RESOURCE_STATE_PRESENT));
```

Application Class (5)

- **ID3D12GraphicsCommandList::OMSetRenderTargets**
  -
```
void OMSetRenderTargets(
  [in] UINT  NumRenderTargetDescriptors,
  [in, optional] const D3D12_CPU_DESCRIPTOR_HANDLE
        *pRenderTargetDescriptors,
  [in] BOOL  RTsSingleHandleToDescriptorRange,
  [in, optional] const D3D12_CPU_DESCRIPTOR_HANDLE
        *pDepthStencilDescriptor
);
```
    - This sets CPU descriptor handles for the render targets and depth stencil.

Application Class (6)

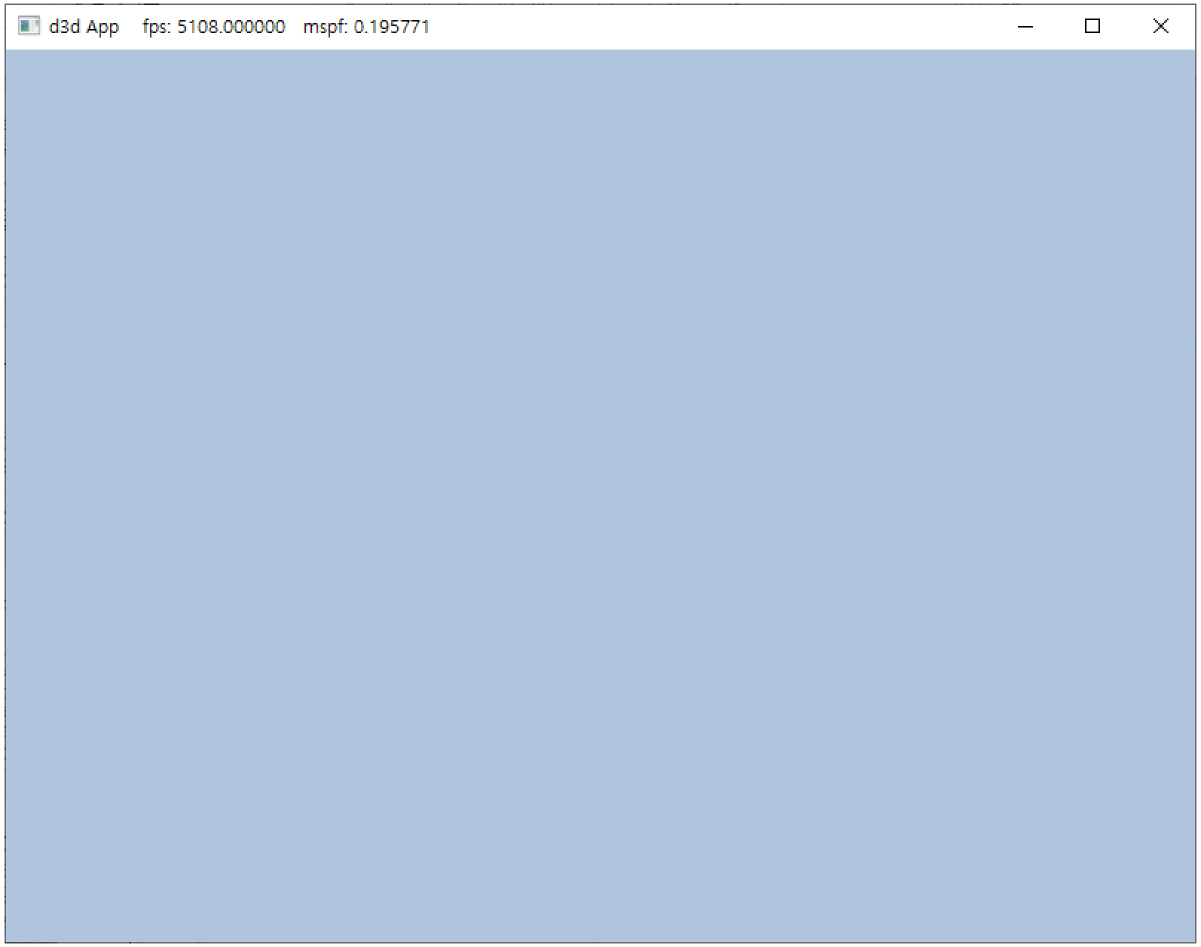
```
// InitDirect3DApp.cpp
    // Done recording commands.
    ThrowIfFailed(mCommandList->Close());

    // Add the command list to the queue for execution.
    ID3D12CommandList* cmdsLists[] = { mCommandList.Get() };
    mCommandQueue->ExecuteCommandLists(_countof(cmdsLists), cmdsLists);

    // swap the back and front buffers
    ThrowIfFailed(mSwapChain->Present(0, 0));
    mCurrBackBuffer = (mCurrBackBuffer + 1) % SwapChainBufferCount;

    // Wait until frame commands are complete.  This waiting is inefficient
    // and is done for simplicity.  Later we will show how to organize
    // our rendering code so we do not have to wait per frame.
    FlushCommandQueue();
}
```

The Demo Application


d3d App   fps: 5108.000000   mspf: 0.195771