**Shared Surface Queue** – How To Use Guide

**Windows Experience – Graphics Infrastructure APIs**

## EXECUTIVE OVERVIEW

There are an increasing number of cases where app developers would like to use the various runtime apis (d3d9, d3d10 and d3d11) in combination to provide to best experience for their users. The app developer can choose the API that is most suitable for their needs. Examples of specific customer’s needs have largely been focused around video. D3D9 is the API of choice for video decoding (DXVA) and secure video presentation but the newer APIs provide additional rendering capabilities. App developers would like to decode and present video using d3d9 but do additional processing and rendering using d3d10 and/or d3d11. Specifically, d3d10 allows the developer to use d2d to render UI components on top of the video and d3d11 allows the use of compute shaders to do additional processing on the video.

Vista introduced the notion of shared handles, which allows resources be shared between devices. Shared handles allows cross-API devices to access the same resource in an efficient manner. Shared handles are also supported in D3D11. Shared handles allow developers to efficiently share the resources but no obvious way to synchronize using the resources. Unfortunately, getting the synchronization correct is difficult. The synchronization is between the CPU and GPU which cannot be implemented simply using the windows synchronization primitives (mutexes, events, etc). Furthermore, the errors will often manifest as race conditions causing slight rendering artifacts, errors that are typically difficult to diagnose.

To further complicate the issue, the features around surface sharing have a history of bugs that makes some of it just not work. The design and implementation of this utility attempts to navigate around these bugs. Known bugs will be mentioned in the design and code.

This utility will try to solve the synchronization problem by providing a building block that allows the app to safely pass shared resources from one device to another. This is a fairly high level synchronization tool but we expect most apps to be able to take advantage of it.

## PLATFORM OVERVIEW

The most fundamental task is passing a single surface from the first device (device A) to the second (device B) such that when device B acquires a handle on the surface, it is guaranteed that device A's rendering has completed. Device B can use this surface without worry. This is very similar to the classical producer-consumer problem and we will model the problem as such. The first device that uses the surface and then relinquishes it will be the producer (Device A), and the device initialing waiting will be the consumer (Device B). Apps that are more sophisticated (we expect all of them to be) will chain together multiple of these producer-consumer building blocks to get their desired functionality.

Like the producer-consumer problem, we will solve this by using a queue of surfaces. Surfaces are enqueued by the producer and dequeued by the consumer. The utility introduces 3 COM interfaces, ISurfaceQueue, ISurfaceProducer and ISurfaceConsumer.

### High Level Overview of Utility

The ISurfaceQueue object will be the building block for using the shared surfaces. It is created with an initialized d3d device and a description to create a fixed number of shared surfaces. The queue object manages all of the resource creation/opening code. The number and type of surfaces are fixed; once the surfaces are created, it’s not possible to add or remove them.

Each instance of the ISurfaceQueue object provides a one-way street which can be used to send surfaces from the producing device to the consuming device. Multiple of these one-way streets can be used to solve the developer’s specific problem.

#### Creation/Object Lifetime

There are two ways to create the queue object, through CreateSurfaceQueue and Clone. Clone will be discussed later. Since the interfaces are COM objects, standard COM lifetime management will apply.

#### Producer/Consumer Model:

Enqueue (): The producer calls this function to indicate it is done with the surface and should become available to another device. Upon returning from this function, the producer device no longer has rights to the surface and it is unsafe to continue using it.

Dequeue (): The consuming device calls this function to get a shared surface. The API guarantees that any dequeued surfaces are ready to be used.

#### MetaData:

The API supports associating metadata with the shared surfaces.

Enqueue() has the option of specifying additional metadata that will be passed to the consuming device. The metadata must be less than a maximize known at creation time.

Dequeue() can optionally pass a buffer and a pointer to the size of the buffer. The queue will fill in the buffer with the meta data from the corresponding Enqueue call.

#### Cloning:

Each ISurfaceQueue object will solve a one-way synchronization. We assume that the vast majority of apps using this API will be interested in a closed system. The simplest closed system with 2 devices sending surfaces back and forth will require 2 queues. The ISurfaceQueue object has a Clone() method to make it possible to create multiple queues that are all part of the same larger pipeline.

Clone creates an ISurfaceQueue from an existing one, sharing all the opened resources between them. The resulting object will have identical surfaces as the source queue. Cloned queues can have different meta data sizes from each other.

#### Surfaces

The ISurfaceQueue takes the responsibility of creating and managing its surfaces. It is not valid to enqueue arbitrary surfaces. Furthermore, a surface should only have one active “owner.” It should either be on specific queue or being used by a specific device. It is not valid to have it on multiple queues or for devices to continue using the surface after it is enqueued.

## API SURFACE AND PLATFORM DETAILS

### ISurfaceQueue

The queue is responsible for creating and maintaining the shared resources. It also provides the functionality to chain multiple queues using Clone. The queue has methods to open the producing device and a consuming device. Only one of each can be opened at any time.

APIs for the queue are:

|  |  |
| --- | --- |
| **CreateSurfaceQueue** | Creates an ISurfaceQueue object (the “root” queue) |
| **ISurfaceQueue::OpenConsumer** | Returns an interface for the consuming device to dequeue. |
| **ISurfaceQueue::OpenProducer** | Returns an interface for the producing device to enqueue |
| **ISurfaceQueue::Clone** | Create an ISurfaceQueue object that shares surfaces with the root queue object. |

#### CreateSurfaceQueue

|  |
| --- |
| typedef struct SURFACE\_QUEUE\_DESC  {  UINT Width;  UINT Height;  DXGI\_FORMAT Format;  UINT NumSurfaces;  UINT MetaDataSize;  DWORD Flags;  } SURFACE\_QUEUE\_DESC  HRESULT CreateSurfaceQueue(  IUnknown\* pDevice,  SHARED\_SURFACE\_QUEUE\_DESC,  ISurfaceQueue \*\*  ); |

#### Parameters:

pDevice: The device that should be used to create the shared surfaces. There is an explicit control for this because of a Vista RTM bug. For surfaces shared between DX9 and DX10, the surfaces must be created with DX9.

Width/Height: The size of the shared surfaces. They must all be the same size.

Format: The format for the shared surfaces. They must all be the same. The valid formats depend on the devices that will be used as different pairs of devices can share different format types.

NumSurfaces: The number of surfaces that are part of the queue. This is fixed size.

MetaDataSize: The maximum size of the meta data buffer.

Flags: Flags to control the behavior of the queue. See remarks.

ppQueue: On return this will contain a pointer to the ISurfaceQueue object.

#### Return Values:

If pDevice is not capable of sharing resources, this function will return DXGI\_ERROR\_INVALID\_CALL. This function creates the resources and can return any error that would be valid for those functions.

#### Remarks:

Creating the queue object will also create all of the surfaces. All surfaces are assumed to be 2D render targets and will be created with the D3D10\_BIND\_RENDER\_TARGET and D3D10\_BIND\_SHADER\_RESOURCE flags set (or the equivalent flags for the different runtimes).

The user can specify as a flag if the queue will not be accessed by multiple threads. If no flags are set (flag == 0), the queue is expecting to be used by multiple threads. The user can specify single threaded access which will turn off the synchronization code, providing a performance improvement for those cases. Cloned queues have their own flag so it is possible for different queues in the system to have different synchronization controls.

#### Open a Producer

|  |
| --- |
| HRESULT OpenProducer  (  IUnknown\* pDevice,  ISurfaceProducer\*\* ppProducer  ) |

#### Parameters:

pDevice is an in parameter. ppProducer will return a producer object on return.

#### Return Values:

If the device is not capable of sharing surfaces, DXGI\_ERROR\_INVALID\_CALL will be returned.

#### Open a Consumer

|  |
| --- |
| HRESULT OpenConsumer  (  IUnknown\* pDevice,  ISurfaceConsumer\*\* ppConsumer  ) |

#### Parameters:

pDevice is an in parameter. ppConsumer will return an producer object on return.

#### Return Values:

If the device is not capable of sharing surfaces, DXGI\_ERROR\_INVALID\_CALL will be returned.

#### Remarks:

This function will open all of the surfaces in the queue for the input device and cache them. Subsequent calls to Dequeue will simply go to the cache and not have to reopen the surfaces each time.

#### Cloning an ISurfaceQueue

|  |
| --- |
| typedef struct SHARED\_SURFACE\_QUEUE\_CLONE\_DESC  {  UINT MetaDataSize;  DWORD Flags;  } SHARED\_SURFACE\_QUEUE\_CLONE\_DESC;  HRESULT Clone  (  SHARED\_SURFACE\_QUEUE\_CLONE\_DESC\* pDesc,  ISurfaceQueue\*\* ppQueue  ) |

#### Parameters:

pDesc is an in parameter and should be initialized. ppQueue will return the initialized object on return. MetaDataSize and Flags have the same behavior as they do for CreateSurfaceQueue.

#### Remarks:

You can clone from any existing queue object; it does not have to be the root. The behavior is identical and it does not make a difference.

### ISurfaceConsumer

|  |
| --- |
| HRESULT Dequeue  (  REFIID id,  void\*\* ppSurface,  void\* pBuffer,  UINT\* pBufferSize,  DWORD dwTimeout  ) |

#### Parameters:

id, pBuffer and dwTimeout are in parameters. ppSurface is an out parameter and pBufferSize is both an in and out parameter.

***id*** should be the REFIID for a 2D surface of the consuming device. If the consuming device is:

IDirect3DDevice9, the REFIID should be \_\_uuidof(IDirect3DTexture9)

ID3D10Device, the REFIID should be \_\_uuidof(ID3D10Texture2D)

ID3D11Device, the REFIID should be \_\_uuidof(ID3D11Texture2D)

***ppSurface*** upon returning will contain a pointer to the surface.

***pBuffer*** is an optional parameter and if not null, on return, will contain the meta data that was passed in on the corresponding enqueue call.

***pBufferSize*** is the size of pBuffer (in bytes). On return it will contain the number of bytes return to pBuffer. If the enqueue call did not provide metadata, pBuffer will be set to 0.

***dwTimeout*** specifics a timeout value for this function. See the remarks for more detail.

#### Return Values:

This function can return WAIT\_TIMEOUT is a timeout value is used. See remarks. If the function returns because no surfaces are availible, ppSurface will be set to NULL, pBufferSize will be set to 0 and the function will return 0x80070120 (WIN32\_TO\_HRESULT(WAIT\_TIMEOUT)).

#### Remarks:

This API can block if the queue is empty. The dwTimeout parameter works identically to the windows synchronization APIs (i.e. WaitForSingleObject). If the user wants non-blocking behavior, a timeout of 0 should be used.

### ISurfaceProducer

This interface provides 2 APIs that allows the app to enqueue surfaces. Once a surface is enqueued, the surface pointer is no longer valid and not safe to use. The only thing that app should do with the pointer is Release it.

|  |  |
| --- | --- |
| **ISurfaceProducer::Enqueue** | Enqueues a surface to the queue object. This indicates the producer is done with the surface and it is ready for another device. |
| **ISurfaceProducer::Flush** | This is for apps that want nonblocking behavior. See remarks for details. |

#### Enqueue

|  |
| --- |
| HRESULT Enqueue  (  IUnknown\* pSurface,  void\* pBuffer,  UINT BufferSize,  DWORD Flags  ) |

#### Parameters:

All parameters to this function are in parameters.

***pSurface*** should be the surface of the producing device that needs to be enqueued. This surface must be a dequeued surface from the same queue network.

***pBuffer*** is an optional parameter to pass in meta data. It should point to the data that will be passed on to the dequeue call.

***BufferSize*** is the size of pBuffer (in bytes).

***Flags*** can be passed to control the behavior of this function. The only flag is SURFACE\_QUEUE\_FLAG\_ DO\_NOT\_WAIT. See the remarks for Flush. If no flag is passed (flag == 0), then the default blocking behavior is used.

#### Return Values:

This function can return DXGI\_ERROR\_WAS\_STILL\_DRAWING if a SURFACE\_QUEUE\_FLAG\_ DO\_NOT\_WAIT flag is used.

#### Remarks:

This API will put the surface on the queue. If the user does not specify SURFACE\_QUEUE\_FLAG\_ DO\_NOT\_WAIT, this API is blocking and will do a GPU-CPU synchronization to assure that all the rendering on the enqueued surface is complete. If this function succeedes, there will be a surface available for dequeue. If the user wants non-blocking behavior, the DO\_NOT\_WAIT flag can be used. See Flush() for details.

As per the COM reference counting rules, the surface returned by Dequeue will be AddRef() so the user does not need to do this. After calling Enqueue, the user must Release the surface since they are no longer using it.

#### Flush

|  |
| --- |
| HRESULT Flush  (  DWORD Flags,  UINT\* nSurfaces  ) |

#### Parameters:

Flags is an input parameter. The only flag is SURFACE\_QUEUE\_FLAG\_ DO\_NOT\_WAIT. See remarks.

nSurfaces is an output parameter. On return, it will return the number of surfaces that are still pending and not flushed.

#### Return Values:

This function can return DXGI\_ERROR\_WAS\_STILL\_DRAWING if the SURFACE\_QUEUE\_FLAG\_ DO\_NOT\_WAIT flag is used. This function will return S\_OK if any surfaces were successfully flushed and only return DXGI\_ERROR\_WAS\_STILL\_DRAWING if no surfaces flushed. A combination of the return value and nSurfaces lets the app know what work has been done and if any work is left to do.

#### Remarks:

Flush is only meaningful if the previous call to enqueue used the DO\_NOT\_WAIT flag (it will be a no-op otherwise). If the call to enqueue used the DO\_NOT\_WAIT flag, enqueue will return immediately and the GPU-CPU synchronization is not guaranteed. The surface is still considered enqueued, the producing device cannot continue using it, but it is not available for dequeue. In order to try to commit the surface for dequeue, Flush must be called. Flush will attempt to commit all of the surfaces that are currently enqueued. If no flag is passed to Flush, it will block and clear out the entire queue, readying all surfaces in it for dequeue. If the DO\_NOT\_WAIT flag is used, the queue will check the surfaces to see if any of them are ready (this step is nonblocking). Surfaces that have finished the gpu-cpu sync will be ready for the consumer device and surfaces that are still pending will be left as is. The function will return the number of surfaces that still need to be flushed.

Note: this will not break the queue semantics. The API guarantees that surfaces enqueued first will be committed before surfaces enqueued later (regardless of when the gpu-cpu sync happens).

## How To Use

We expect most of the use cases to involve two devices sharing a number of surfaces between them. Since this also happens to be the simplest scenario, this doc will go into some detail on how to use the APIs to this task. There is a brief section on how to initialize for 3 devices at the end.

### Two Devices

The example app that wants to use this utility would like to use D3D9Ex and D3D11 together. The application wants to process content with both devices and present using D3D9. Process could mean render content, decode video, run compute shaders, etc. For every frame, the application will want:

Process with D3D11🡪Process with D3D9 🡪 Present with D3D9, in this order.

Furthermore, the processing with D3D11 will produce some meta data that D3D9’s present will need to consume.

We’ll break up the utilty usage into 3 parts, Initialization, Main Loop and Cleanup.

#### Initialization

Initialization involves:

1. Initialize both devices
2. Creating the Root Queue (call it m\_11to9Queue)
3. Cloning from the Root Queue (call it m\_9to11Queue).
4. Calling OpenProducer/OpenConsumer on both queues

Naming Convention: the m\_11to9Queue indicates a queue where d3d11 is producing surfaces for d3d9. For this queue, the D3D11 device would be the producer and the d3d9 device is the consumer. Similary, m\_9to11Queue indicates D3D9 is the producer and D3D11 is the consumer.

The queue has the behavior that Root queue is initially full and all cloned queues start of empty. This should not be a big deal to the app except for the first cycle of the Enqueues and Dequeues and the availability of meta data. If a dequeue asks for meta data but none was set (either because nothing is there initially or the enqueue did not set anything), dequeue will see that no meta data was received. Some care might be needed to make those scenarios work.

Code snippets for the 4 steps are below:

1. Devices are initialized as normal – this part is largely omitted

|  |
| --- |
| m\_pD3D9Device = InitializeD3D9ExDevice();  m\_pD3D11Device = InitializeD3D11Device(); |

1. Creating the root queue also creates the surfaces. Size and format restrictions are identical tto creating any shared resource. The size of the meta data buffer is fixed at create time, and in this case, we’ll just be passing a UINT.

The queue must be created with a fixed number of surfaces and performance will vary depending on the scenario. Having multiple surfaces increases the chances that devices are busy. For example, if there was only one surface, then there will be no parallelization between the two devices. On the other hand, increasing the number of surfaces increases the memory footprint which can cause bad perf. In this example, we’ll use two surfaces.

|  |
| --- |
| SURFACE\_QUEUE\_DESC Desc;  Desc.Width = 640;  Desc.Height = 480;  Desc.Format = DXGI\_FORMAT\_R16G16B16A16\_FLOAT;  Desc.NumSurfaces = 2;  Desc.MetaDataSize = sizeof(UINT);  Desc.Flags = 0;  CreateSurfaceQueue(&Desc, m\_pD3D9Device, &m\_11to9Queue); |

1. Now that the root queue is created, we can clone the second queue. Cloned queues must use the same surfaces but can have different meta data buffer sizes and different flags. In this case, there is no meta data from D3D9 to D3D11.

|  |
| --- |
| SURFACE\_QUEUE\_CLONE\_DESC Desc;  Desc.MetaDataSize = 0;  Desc.Flags = 0;  m\_11to9Queue->Clone(&Desc, &m\_9to11Queue); |

1. The last step to initialize is to open the producer and consumer devices. This must be done before calls to Enqueue and Dequeue. Opening a producer and consumer returns interfaces which contain the enqueue/dequeue APIs.

|  |
| --- |
| // Open for m\_p9to11Queue  m\_p9to11Queue->OpenProducer(m\_pD3D9Device, &m\_pD3D9Producer);  m\_p9to11Queue->OpenConsumer(m\_pD3D11Device, &m\_pD3D11Consumer);  // Open for m\_p11to9Queue  m\_p11to9Queue->OpenProducer(m\_pD3D11Device, &m\_pD3D11Producer);  m\_p11to9Queue->OpenConsumer(m\_pD3D9Device, &m\_pD3D9Consumer); |

#### Main Loop

The usage of the queue is modeled after the classical producer/consumer problem. It’s probably easiest to think from a per device perspective. Each device must: dequeue to get a surface from its consuming queue, process on the surface, enqueue onto its producing queue.

For the D3D11 Device:

|  |
| --- |
| // D3D11 Device  ID3D11Texture2D\* pSurface11 = NULL;  REFIID surfaceID11 = \_\_uuidof(ID3D11Texture2D);  UINT metaData;  while (!done)  {  // Dequeue surface  m\_pD3D11Consumer->Dequeue(surfaceID11, (void\*\*)&pSurface11,  NULL, 0, INFINITE);    // Process the surface and return some meta data.  ProcessD3D11(pSurface11, &metaData);  // Enqueue surface  m\_pD3D11Producer->Enqueue(pSurface11, &metaData,  sizeof(UINT), 0);  } |

The D3D9 usage is almost identical:

|  |
| --- |
| // D3D9 Device  IDirect3DTexture9\* pTexture9 = NULL;  REFIID surfaceID9 = \_uuidof(IDirect3DTexture9);  UINT metaData;  UINT metaDataSize;  while (!done)  {  // Dequeue surface  m\_pD3D9Consumer->Dequeue(surfaceID9, (void\*\*)&pSurface9,  &metaData, &metaDataSize, INFINITE);    // Process the surface.  ProcessD3D9(pSurface9);  // Present the surface using the meta data.  PreesntD3D9(pSurface9, metaData, metaDataSize);  // Enqueue surface  m\_pD3D9Producer->Enqueue(pSurface9, NULL, 0, 0);  } |

#### Cleaning up

Cleaning up is very straightforward. All of the returned come interfaces must be released in addition to what is normally necessary to cleanup D3D.

|  |
| --- |
| m\_pD3D9Producer->Release();  m\_pD3D9Consumer->Release();  m\_pD3D11Producer->Release();  m\_pD3D11Consumer->Release();  m\_p9to11Queue->Release();  m\_p11to9Queue->Release();  // cleanup d3d resources as normal |

#### Non-blocking use

The example provided above would be for a multithreaded use case where each device is on its own thread. It uses the blocking versions of the APIs: INFINITE for timeout and no flag to enqueue. Should the app want to use the utility in a non-blocking way, only a few changes have to be made. Here are the changes for a simple non-blocking use for the same scenario but both devices on one thread.

##### Initialization

Initialization is identical except for the flags. Since the app is single threaded, we can use that flag for creation. This will turn off some of the sychronization code potentially providing a performance benefit.

|  |
| --- |
| SURFACE\_QUEUE\_DESC Desc;  Desc.Width = 640;  Desc.Height = 480;  Desc.Format = DXGI\_FORMAT\_R16G16B16A16\_FLOAT;  Desc.NumSurfaces = 2;  Desc.MetaDataSize = sizeof(UINT);  Desc.Flags = SURFACE\_QUEUE\_FLAG\_SINGLE\_THREADED;  CreateSurfaceQueue(&Desc, m\_pD3D9Device, &m\_11to9Queue); |

|  |
| --- |
| SURFACE\_QUEUE\_CLONE\_DESC Desc;  Desc.MetaDataSize = 0;  Desc.Flags = SURFACE\_QUEUE\_FLAG\_SINGLE\_THREADED;  m\_11to9Queue->Clone(&Desc, &m\_9to11Queue); |

Opening the producer and consumer devices are identical to before.

##### Using the Queue

There are many ways of using the queue in a non-blocking fashion with various performance characteristics. Below a very simple but not very performant approach which does excessive of spinning and polling. It should however illustrate how to use the utility. The approach is to constantly sit in a loop and dequeue, process, enqueue and flush. If any of the steps fail because the resource is not available, the app will simply try again the next loop.

|  |
| --- |
| // D3D11 Device  ID3D11Texture2D\* pSurface11 = NULL;  REFIID surfaceID11 = \_\_uuidof(ID3D11Texture2D);  UINT metaData;  while (!done)  {  //  // D3D11 Portion  //  // Dequeue surface  hr = m\_pD3D11Consumer->Dequeue(surfaceID11,  (void\*\*)&pSurface11,  NULL, 0, 0);  // Only continue if we got a surface  if (SUCCEEDED(hr))  {  // Process the surface and return some meta data.  ProcessD3D11(pSurface11, &metaData);  // Enqueue surface  m\_pD3D11Producer->Enqueue(pSurface11, &metaData,  sizeof(UINT),  SURFACE\_QUEUE\_FLAG\_DO\_NOT\_WAIT);  }  // Flush the queue to check if any surfaces completed  m\_pD3D11Producer->Flush(NULL,SURFACE\_QUEUE\_FLAG\_DO\_NOT\_WAIT);  //  // Do the same with the D3D9 Device  //  // Dequeue surface  hr = m\_pD3D9Consumer->Dequeue(surfaceID9,  (void\*\*)&pSurface9,  &metaData,  &metaDataSize, 0);  // Only continue if we got a surface  if (SUCCEEDED(hr)))  {  // Process the surface.  ProcessD3D9(pSurface9);  // Present the surface using the meta data.  PreesntD3D9(pSurface9, metaData, metaDataSize);  // Enqueue surface  m\_pD3D9Producer->Enqueue(pSurface9, NULL, 0,  SURFACE\_QUEUE\_FLAG\_DO\_NOT\_WAIT);  }  // Flush the queue to check if any surfaces completed  m\_pD3D9Producer->Flush(NULL,SURFACE\_QUEUE\_FLAG\_DO\_NOT\_WAIT);    } |

More complex solutions could check the return value from enqueue and from flush to determine if flushing is necessary.

### 3 Devices

Extending to multiple devices is pretty straightforward. Below is some code that does the initialization. Once the Producer/Consumer objects have been created, the code to use them is the same.

In this example we will have 3 devices and therefore 3 queues. Surfaces will flow from D3D9 🡪 D3D10 🡪 D3D11.

|  |
| --- |
| SURFACE\_QUEUE\_DESC Desc;  Desc.Width = 640;  Desc.Height = 480;  Desc.Format = DXGI\_FORMAT\_R16G16B16A16\_FLOAT;  Desc.NumSurfaces = 2;  Desc.MetaDataSize = sizeof(UINT);  Desc.Flags = 0;  SURFACE\_QUEUE\_CLONE\_DESC Desc;  Desc.MetaDataSize = 0;  Desc.Flags = 0;  CreateSurfaceQueue(&Desc, m\_pD3D9Device, &m\_11to9Queue);  m\_11to9Queue->Clone(&Desc, &m\_9to10Queue);  m\_11to9Queue->Clone(&Desc, &m\_10to11Queue); |

As mentioned in the docs, cloning off of any queue is equivalent. The second Clone call could have been off of the m\_9to10Queue object.

|  |
| --- |
| // Open for m\_p9to10Queue  m\_p9to10Queue->OpenProducer(m\_pD3D9Device, &m\_pD3D9Producer);  m\_p9to10Queue->OpenConsumer(m\_pD3D10Device, &m\_pD3D10Consumer);  // Open for m\_p10to11Queue  m\_p10to11Queue->OpenProducer(m\_pD3D10Device, &m\_pD3D10Producer);  m\_p10to11Queue->OpenConsumer(m\_pD3D11Device, &m\_pD3D11Consumer);  // Open for m\_p11to9Queue  m\_p11to9Queue->OpenProducer(m\_pD3D11Device, &m\_pD3D11Producer);  m\_p11to9Queue->OpenConsumer(m\_pD3D9Device, &m\_pD3D9Consumer); |