Migrating Your Apps to DirectX* 12 – Part 4

Chapter 4 DirectX 12 Features

4.0 Links to the Previous Chapters

4.1 Function and Usage of Multiplane Overlay

4.1.1 Introduction

Multiplane Overlays (MPO) is a new feature of WDDM1.3 (DX11.2) initially introduced in Windows 8.1, which is now extended to WDDM2.0 (DX12) in Windows 10. MPO supports using the original resolution to display gorgeous 2D art and UI elements, while drawing 3D scenes into a smaller and stretchable frame buffer. The stretching and composition of surfaces at different resolutions are automatically implemented by the system, and transparent to applications.

The major role of MPO is to allow a game to maintain a stable and appropriate frame rate under different circumstances, thereby improving the overall game experience. On one hand, the resolution almost always has a significant impact on the performance of modern games. With the popularity of high-definition 4K screens, the resolution of game window also increases. Increasing pixels also increases texture sampling and the bandwidth of render target, which brings challenges to the performance of games. Secondly, post-processing technology is increasingly used for rendering 3D scenes, which increases the complexity of a shader, so the trend will be that a trend that the cost per pixel determines games performanc. Therefore, setting the appropriate resolution is critical to maintaining game performance.

Furthermore, for games like role playing, real-time strategy and multiplayer online, rendering GUI components at the initial window resolution is also very important. For example, even on low-end platforms, players will want to text chat with teammates during game play. GUI components are mainly 2D graphics with relatively low rendering overhead, thus rendering at the window resolution can provide the best visual experience.

MPO provides a compromise solution for high-resolution display and gaming performance. Using MPO can alleviate the rapid performance degradation of games in high-resolution rendering, allowing the games to run smoothly on the majority of hardware platforms. MPO supports not only rendering GUI layer and 3D content layer in different resolutions, bu also rendering in different frame rates. For instance, for a GUI layer, just present when a change happened. It benefits to performance and power efficiency. Fiture 1 illustracts the swap chain composition of MPO.

Because MPO is a new feature of driver model, graphics driver provides the implementation of MPO to the upper-level D3D runtime or DXGI runtime. MPO can be implemented by WDDM1.3 or higher version drivers through software or by graphics hardware that uses a dedicated pipeline for the GPU in order to reduce consuming so many CPU and GPU resources, helping to further enhance game performance and lower power consumption. Intel will add hardware support for MPO in 6th generation Intel® Core™ processors (codenamed SkyLake).

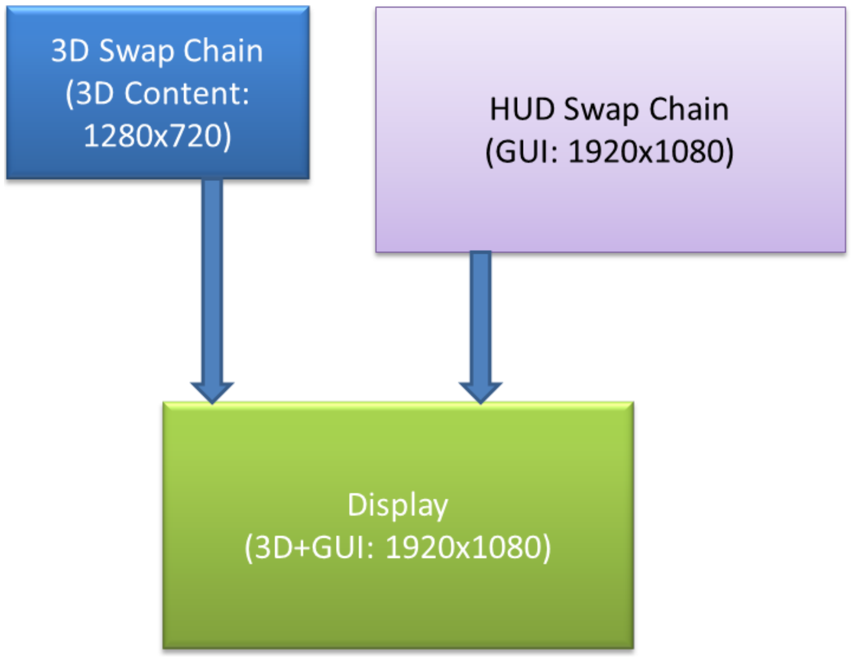

Figure 1: Swap Chain Composition of Multi-Plane Overlay

4.1.2 Application scenarios

In a game, Multiplane Overlays are typically applied in the following scenarios.

1. Scenes where the performance (such as frame rate) is below a certain threshold.

   Depending on the type of the game, when the frame rate is below a certain value, the playability of the game is degraded. For example, in battle scenes where a large number of game characters appear, the frame rate of the game decreases significantly. To avoid performance degradation to the point where the game can't be played, the game will automatically switch to the MPO rendering mode to maintain playable performance when it detects that the frame rate is lower than the threshold.

2. Scenes that release particle effects.
   In MMOG, scenes that release a large number of particle effects often result in the sudden drop in frame rate and even cause the game stuttering in severe cases, which is a common performance bottleneck in games. Rendering particle effects involves a lot of pixel filling, while rendering on a high-resolution surface has more significant impact on performance. The game will switch to MPO mode when releasing particle effects, and render the particle effects to a render target with smaller resolution to reduce the amount of pixel filling. In addition, in the fierce battle scenes, the release of a large number of particle effects leads to very dramatic changes in the screen, and thus it is difficult for the player to see the details of each frame. Therefore, the impact of MPO stretching render target on the final visual quality is almost imperceptible. Using MPO in such scenes, your game can maintain the proper frame rate while keeping the visual experience, and addressing common performance problems.

3. Rendering on mobile platforms
   Now that mobile platforms are popular gaming terminal, game developer must worry about power consumption. When the platform switches from AC power mode to battery power mode, the system performance settings could cause the CPU or GPU frequency to decrease and force the game frame

rate to drop. With MPO, a game will automatically adjust the resolution of the render target without changing the window resolution to compensate for the overall performance, while reducing power consumption and allowing the game to run longer. This feature of MPO can be used to implement power-saving mode for the game. This mode is available as a game setting option for the players. For players, in the power-saving mode, slight difference on the screen is acceptable.

4. All of the above application scenarios mentioned focus on dynamically adjusting resolution to maintain FPS. In practice, it will probably be more common to just choose a fixed ratio for the duration of the game. For a 3D content layer, a natural choice for the scale ratio would be based on the device DPI. (e.g. scale = deviceDpi/96.0)

4.1.3 Sample Codes for API usage

To apply Multiplane Overlays (MPO) in a game, you first need to detect whether the software and hardware on the platform support it. The code for this is as follows:

Table 4.1: Detecting whether the hardware supports Multiplane Overlay

```
BOOL supportMultiPlaneOverlay = FALSE;
IDXGISwapChain* dxgiSwapChain;
IDXGIOutput* dxgiOutput;

dxgiSwapChain->GetContainingOutput(&dxgiOutput);
if (dxgiOutput)
{
    IDXGIOutput2* dxgiOutput2;
    dxgiOutput->QueryInterface(IID_PPV_ARGS(&dxgiOutput2));
    SAFE_RELEASE(dxgiOutput);
    if (dxgiOutput2)
    {
        supportMultiPlaneOverlay = dxgiOutput2->SupportsOverlays();
        SAFE_RELEASE(dxgiOutput2);
    }
}
```

IDXGIOutput2 : SupportsOverlays () API is used to detect whether the graphics hardware supports this feature. If it returns true, then the hardware supports this feature; if it returns false, then the hardware doesn't support this feature, but the driver supports it using software approach. Intel Skylake platform and later processor cores will support MPO on the hardware level.

In the game initialization phase, it's required to create a few key Multiplane Overlay API objects: such as Direct Composition device, scale transformer, foreground, background SwapChain, etc. The sample code for desktop apps is as follows~~:~~

Table 4.2: Initialization and Creation of Multiplane Overlay API Objects

```
IDXGIFactory2* dxgiFactory;
adapter->GetParent(IID_PPV_ARGS(&dxgiFactory));

if (dxgiFactory)
{
    // Create a Direct Composition device
    DCompositionCreateDevice(NULL, IID_PPV_ARGS(&m_directCompositionDevice));

     // The device creates a render target for the window
    m_directCompositionDevice->CreateTargetForHwnd(
   (HWND)_CORE_API->GetAppWindow(), false, &m_directCompositionTarget);

    // The device creates a multi-layer view
    m_directCompositionDevice->CreateVisual(&m_rootVisual);
    m_directCompositionDevice->CreateVisual(&m_mainVisual);
    m_directCompositionDevice->CreateVisual(&m_overlayVisual);

    // The device creates a scale transformer and sets it into the main view
    m_directCompositionDevice->CreateScaleTransform(&m_mainScaleTransform);
    m_mainVisual->SetTransform(m_mainScaleTransform);

    // When the main view is stretched, adopt the linear interpolation for filtering

       m_mainVisual->SetBitmapInterpolationMode(DCOMPOSITION_BITMAP_INTERPOLATION_MO
DE_LINEAR);

 // Add the main view and interface view to the root view, set the root
    view onto the render target
     m_rootVisual->AddVisual(m_mainVisual, FALSE, NULL);
     m_rootVisual->AddVisual(m_overlayVisual, FALSE, NULL);
    m_directCompositionTarget->SetRoot(m_rootVisual);

    // Prepare to create SwapChain
    DXGI_SWAP_CHAIN_DESC1 swapChainDesc = { 0 };
       swapChainDesc.Width = width; // Match the size of the window.
       swapChainDesc.Height = height;
       swapChainDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
       swapChainDesc.Stereo = false;
       swapChainDesc.SampleDesc.Count = 1; // Don't use multi-sampling.
       swapChainDesc.SampleDesc.Quality = 0;
       swapChainDesc.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;
       swapChainDesc.BufferCount = 2;
       swapChainDesc.SwapEffect = DXGI_SWAP_EFFECT_FLIP_DISCARD;
       swapChainDesc.Flags = DXGI_SWAP_CHAIN_FLAG_FRAME_LATENCY_WAITABLE_OBJECT;
       swapChainDesc.Scaling = DXGI_SCALING_STRETCH;
       swapChainDesc.AlphaMode = DXGI_ALPHA_MODE_PREMULTIPLIED;

    // DXGIFactory creates the foreground SwapChain
     hr = dxgiFactory->CreateSwapChainForComposition(
            m_HUDCommandQueue,
            &swapChainDesc,
            nullptr,
            &m_foregroundSwapChain
            );
    // Bind the interface view to the foreground SwapChain
     m_overlayVisual->SetContent(m_foregroundSwapChain);

    // Create the background SwapChain
     swapChainDesc.AlphaMode = DXGI_ALPHA_MODE_IGNORE;

     hr = dxgiFactory->CreateSwapChainForComposition(
            m_3DCommandQueue,
            &swapChainDesc,
            nullptr,
            &m_backgroundSwapChain
            );


      // Bind the main view to the background SwapChain
     m_mainVisual->SetContent(m_backgroundSwapChain);
     m_directCompositionDevice->Commit();
     SAFE_RELEASE(dxgiFactory);
}
```

Note: D3D12 and D3D11 take different objects as the first argument for CreateSwapChainForComposition: D3D12 takes CommandQueue, while D3D11 takes Device.

It is recommended to use separate queues for different swapchains. The reason is that DXGI will insert waits on your behalf in order to synchronize with DWM, so using the same queue for two swapchains could result in workloads targeting one swapchain being blocked, waiting for a buffer to become available from the other swapchain

.

In the game rendering process, the code logic based on MPO rendering is as follows:

Table 4.3: Multiplane Overlay-based Game Rendering Method

```
// According to changes in the number of frames, dynamically adjust the scale ratio
of the background SwapChain
m_mainScaleTransform->SetScaleX(scaleRatio);
m_mainScaleTransform->SetScaleY(scaleRatio);
m_directCompositionDevice->Commit();

// Before rendering 3D scenes, first set the corresponding RenderTarget of the
background SwapChain
//
Adjust the size of Viewport at the same time
OMSetRenderTargets(1, &m_backgroundRTV, m_backgroundDSV);
viewport.Width = foregroundSwapChain.Width / scaleRatio;
viewport.Height = foregroundSwapChain.Height / scaleRatio;
RSSetViewports(1, viewport);

// Draw 3D Scene...

// When performing full-screen post-processing, you need to change the texture
 sampling coordinates, control the addressing range, because the background Viewport
 may only partially filled the RenderTarget
VSSetConstantBuffer(scaleRatio);
// Draw Fullscreen PostProcess

// Before rendering the UI, first set the corresponding RenderTarget of the
foreground SwapChain
// Adjust the size of Viewport at the same time
OMSetRenderTargets(1, &m_foregroundRTV, m_foregroundDSV);
viewport.Width = foregroundSwapChain.Width;
viewport.Height = foregroundSwapChain.Height;
RSSetViewports(1, viewport);

// Draw UI..

// Finally submit SwapChain
m_backgroundResource->SetTransitionBarrier(D3D12_RESOURCE_STATE_PRESENT);
m_foregroundResource->SetTransitionBarrier(D3D12_RESOURCE_STATE_PRESENT);

m_commandList->Close();

m_backgroundSwapChain->Present(0, 0);
m_foregroundSwapChain->Present(0, 0);
```

4.1.4 Summary

Multiplane Overlays is a new graphic display function on the Windows 8.1 and later platforms. Win10 platform-based DirectX 12 games can easily use the DX12 API to render Multiplane Overlays, potentially solving the problem of sudden drop in frame rate due to high-resolution rendering and high load scenes for games, enabling smooth gaming experience on the majority of hardware platforms.

Coming Soon: Link to the Following Chapter

Chapter 5: DirectX 12 Optimization

Notices

All sample source code is released under the Intel Sample Source Code License Agreement.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

The products and services described may contain defects or errors known as errata which may cause deviations from published specifications. Current characterized errata are available on request.

Copies of documents which have an order number and are referenced in this document may be obtained by calling 1-800-548-4725 or by visiting www.intel.com/design/literature.htm.

Intel, the Intel logo, and Intel Core are trademarks of Intel Corporation in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others

© 2015 Intel Corporation.