DirectX Graphics Articles

Article • 04/27/2021

Purpose

This section contains technical articles that describe WARP and Windows 7's newly added support for Flip Mode Present and its associated Present Statistics in Direct3D 9Ex and Desktop Window Manager. This section also contains technical articles about Windows graphics APIs, porting Direct3D 9 APIs to Microsoft DirectX Graphics Infrastructure (DXGI) APIs, and how to deploy Direct3D 11.

For more information about Direct3D, see Direct3D.

Developer audience

These technical articles are for DirectX graphics application developers.

In this section

Expand table

Topic	Description
Platform Update for Windows 7	Describes what components of the Windows 8 graphics stack become available, and to what extent, through the Platform Update for Windows 7.
Direct3D 9Ex Improvements	Describes Windows 7's newly added support for Flip Mode Present and its associated Present Statistics in Direct3D 9Ex and Desktop Window Manager. Target applications include video or framerate-based presentation applications. Applications using Direct3D 9Ex Flip Mode Present reduce the system resource load when DWM is enabled. Present Statistics enhancements associated with Flip Mode Present enable Direct3D 9Ex applications to better control the rate of presentation by providing real-time feedback and correction mechanisms. Detailed explanations and pointers to sample resources are included.
Surface Sharing Between Windows Graphics APIs	Provides a technical overview of interoperability using surface sharing between Windows graphics APIs, including Direct3D 11, Direct2D, Direct3D 10, and Direct3D 9Ex. If you already have a working knowledge of these APIs, this paper can help you use multiple APIs to render to the same surface in an application designed for the Windows 7 or Windows Vista operating systems. This topic also provides best practice guidelines and pointers to additional resources.
WARP Guide	Provides a guide of Windows Advanced Rasterization Platform (WARP), a high speed software rasterizer.

Topic	Description
Graphics APIs in Windows	Discusses Windows graphics features and APIs.
Direct3D 11 Deployment for Game Developers	Describes how to deploy the Direct3D 11 components on a system.
DirectX Graphics Infrastructure (DXGI): Best Practices	Discusses issues about porting Direct3D 9 APIs to DXGI APIs and features of various versions of DXGI.
Using DirectX with high dynamic range displays and advanced color	This topic provides a technical overview of rendering high dynamic range Direct3D 11 and Direct3D 12 content to an HDR10 display using Windows 10 advanced color support.

Platform Update for Windows 7

Article • 08/19/2020

This topic describes improvements to components of the Windows 7 graphics stack that become available through the Platform Update for Windows 7.

When installed on Windows 7, the Platform Update for Windows 7 updates Windows 7 with functionality available in Windows 8. For example, these Windows 8 components become available with full functionality:

- Direct2D 1.1 (including Direct2D Effects)
- DirectWrite
- Windows Imaging Component (WIC)

These provide partial functionality:

- Direct3D 11.1
- DXGI 1.2

And, for example, this component is not available:

- DirectComposition (DComp)

See these topics for info about Direct2D, DirectWrite, and WIC with the platform update:

- What's new in Direct2D for Windows 8 (Windows)
- What's new in DirectWrite for Windows 8 (Windows)
- What's New for WIC in Windows 8 (Windows)

See these topics for info about Direct3D and DXGI with the platform update:

- D3D11.1 Features
- DXGI 1.2 Improvements

After the platform update has been installed, the interfaces introduced in Direct3D11.1 and DXGI 1.2 will be available with partial functionality. The features of these graphics components are tied directly to the graphics kernel components, graphics drivers, and graphics hardware. Before using Direct3D11.1 on Windows 7, be familiar with these specifics:

- Windows 8 introduced the WDDM 1.2 driver model, which provided improvements across the associated API surface for all feature levels. When reading the Direct3D11.1 documentation, understand that *new drivers* means WDDM 1.2 drivers. These updated driver versions, as well as most optional features exposed

through **CheckFeatureSupport**, are unavailable on Windows 7. Since there is no guarantee that these optional features are available, make sure your applications have appropriate fallback behaviors in the event that the desired functionality is unavailable.

There's one important exception. Several features, such as **PSSetConstantBuffers1** with constant buffer offsets, require new drivers for feature level 10 and higher, but are actually emulated for feature level 9. This emulation is available on Windows 7 with the platform update. See **D3D11_FEATURE_DATA_D3D11_OPTIONS** for more info about which features are emulated.

- The Windows 8 WDDM 1.2 driver model supports a new generation of hardware, exposed through D3D feature level 11.1. Windows 7 with the platform update supports only the WDDM 1.1 driver model and therefore, feature level 11.1 hardware support is not available (through the platform update). On Windows 7 with the platform update, **D3D11CreateDevice** always returns a feature level of 11.0 or lower, except for with a reference device that can be used to test an 11.1 code path on Windows 7. Only use features available at your target feature levels, as described in the feature level reference.

- Some new methods introduced in DGXI 1.2 are not fully supported with the Platform Update for Windows 7.You can test for the availability of these functions by calling them directly and checking for an error code. Make sure your applications targeting Windows 7 with the platform update have a fallback in place when the desired functionality is unavailable. These classes of features are unavailable on Platform Update for Windows 7:
  - Stereo
  - Swapchains not targeting Hwnds
  - Occlusion status notifications
  - Desktop duplication
  - NT Handle resources

  Specifically, the following APIs will return DXGI_ERROR_UNSUPPORTED, DXGI_ERROR_INVALID_CALL, E_NOTIMPL, or E_INVALIDARG:
  - **IDXGIFactory2::CreateSwapChainForCoreWindow**
  - **IDXGIFactory2::CreateSwapChainForComposition**
  - **IDXGIFactory2::RegisterStereoStatusWindow**
  - **IDXGIFactory2::RegisterStereoStatusEvent**
  - **IDXGIFactory2::UnregisterStereoStatus**
  - **IDXGIFactory2::RegisterOcclusionStatusWindow**
  - **IDXGIFactory2::RegisterOcclusionStatusEvent**
  - **IDXGIFactory2::UnregisterOcclusionStatus**

    - IDXGISwapChain1::GetCoreWindow
    - IDXGISwapChain1::SetRotation
    - IDXGISwapChain1::GetRotation
    - IDXGIOutput1::DuplicateOutput
    - IDXGIDevice2::EnqueueSetEvent
    - IDXGIResource1::CreateSharedHandle
    - IDXGIFactory2::GetSharedResourceAdapterLuid
    - ID3D11Device1::OpenSharedResource1
    - ID3D11Device1::OpenSharedResourceByName

- These APIs have behavior differences, as noted:
    - IDXGIFactory2::CreateSwapChainForHwnd takes a DXGI_SWAP_CHAIN_DESC1 structure, which has a field for **Scaling**. DXGI_SCALING_NONE is not supported on Windows 7 with the platform update and causes **CreateSwapChainForHwnd** to return DXGI_ERROR_INVALID_CALL when called.
    - IDXGISwapChain1::SetBackgroundColor is only useful when set on a swapchain using DXGI_SCALING_NONE. Its value is still stored and can be retrieved, but it has no effect.
    - IDXGIDisplayControl::IsStereoEnabled, IDXGIFactory2::IsWindowedStereoEnabled, and IDXGISwapChain1::IsTemporaryMonoSupported all return **FALSE**.
    - IDXGIOutput1::GetDisplayModeList1 and **IDXGIOutput1**::FindClosestMatchingMode1 were added to facilitate stereo display modes. Stereo is not supported on the Platform Update for Windows 7 so this method is equivalent to IDXGIOutput::FindClosestMatchingMode as DXGI_MODE_DESC1.Stereo will always be FALSE.
    - IDXGIDevice2::OfferResources and **IDXGIDevice2**::ReclaimResources are not supported on the Platform Update for Windows 7. However, the runtime still allows them to be called, and performs validation that they are being used correctly on non-shared resources.
    - WARP devices only support feature level 11.0. That is, WARP devices created by passing D3D_DRIVER_TYPE_WARP in the *DriverType* parameter of D3D11CreateDevice don't support 11.1 nor do they support shared surfaces.

- For developers currently working on applications in Microsoft Visual Studio 2010 or earlier using the D3D11_CREATE_DEVICE_DEBUG flag, be aware that calls to D3D11CreateDevice will fail. This is because the D3D11.1 runtime now requires D3D11_1SDKLayers.dll instead of D3D11SDKLayers.dll. To get this new DLL (D3D11_1SDKLayers.dll), install the Windows 8 SDK, or Visual Studio 2012, or the Visual Studio 2012 remote debugging tools. See the Debug Layer documentation for more info.

Feedback

Was this page helpful? Yes No

Get help at Microsoft Q&A

Direct3D 9Ex improvements

Article • 08/19/2020

This topic describes Windows 7's added support for Flip Mode Present and its associated present statistics in Direct3D 9Ex and Desktop Window Manager. Target applications include video or frame rate-based presentation applications. Applications that use Direct3D 9Ex Flip Mode Present reduce the system resource load when DWM is enabled. Present statistics enhancements associated with Flip Mode Present enable Direct3D 9Ex applications to better control the rate of presentation by providing real-time feedback and correction mechanisms. Detailed explanations and pointers to sample resources are included.

This topic contains the following sections.

What's Improved about Direct3D 9Ex for Windows 7

Flip Mode Presentation of Direct3D 9Ex is an improved mode of presenting images in Direct3D 9Ex that efficiently hands off rendered images to Windows 7 Desktop Window Manager (DWM) for composition. Beginning in Windows Vista, DWM composes the entire Desktop. When DWM is enabled, windowed mode applications present their contents on the Desktop by using a method called Blt Mode Present to DWM (or Blt Model). With Blt Model, DWM maintains a copy of the Direct3D 9Ex rendered surface for Desktop composition. When the application updates, the new content is copied to the DWM surface through a blt. For applications that contain Direct3D and GDI content, the GDI data is also copied onto the DWM surface.

Available in Windows 7, Flip Mode Present to DWM (or Flip Model) is a new presentation method that essentially enables passing handles of application surfaces between windowed mode applications and DWM. In addition to saving resources, Flip Model supports enhanced present statistics.

Present statistics are frame-timing information that applications can use to synchronize video and audio streams and recover from video playback glitches. The frame-timing information in present statistics allows applications to adjust the presentation rate of their video frames for smoother presentation. In Windows Vista, where DWM maintains a corresponding copy of the frame surface for Desktop composition, applications can use present statistics provided by DWM. This method of obtaining present statistics will still be available in Windows 7 for existing applications.

In Windows 7, Direct3D 9Ex-based applications that adopt Flip Model should use D3D9Ex APIs to obtain present statistics. When DWM is enabled, windowed mode and full-screen exclusive mode Direct3D 9Ex applications can expect the same present statistics information when using Flip Model. Direct3D 9Ex Flip Model present statistics enable applications to query for present statistics in real time, rather than after the frame has been shown on screen; the same present statistics information is available for windowed-mode Flip-Model enabled applications as full-screen applications; an added flag in D3D9Ex APIs allows Flip Model applications to effectively discard late frames at presentation time.

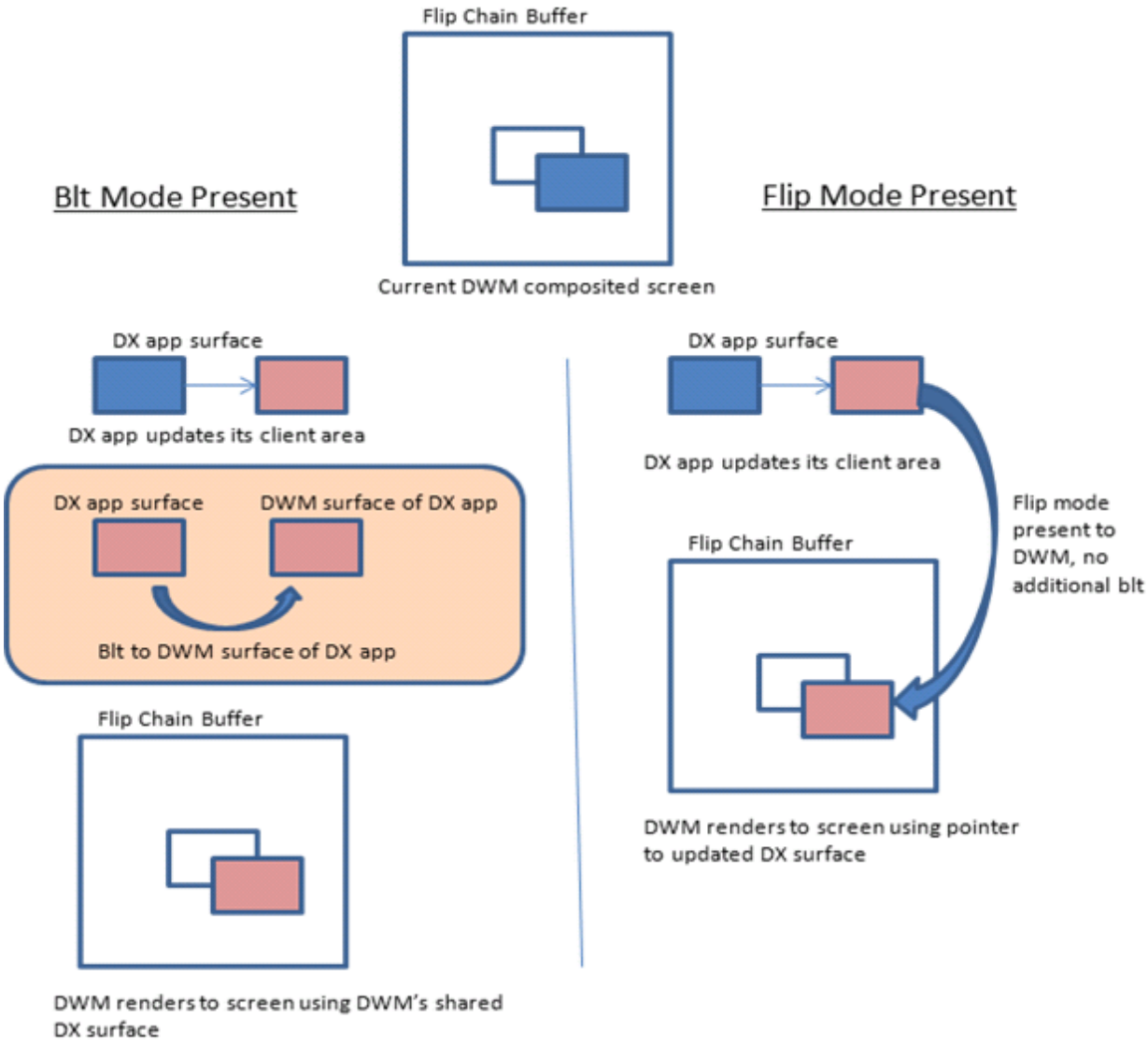
Direct3D 9Ex Flip Model should be used by new video or frame rate-based presentation applications that target Windows 7. Because of the synchronization between DWM and the Direct3D 9Ex runtime, applications that use Flip Model should specify between 2 to 4 backbuffers to ensure smooth presentation. Those applications that use present statistics information will benefit from using Flip Model enabled present statistics enhancements.

Direct3D 9EX Flip Mode Presentation

Performance improvements of Direct3D 9Ex Flip Mode Present are significant on the system when DWM is on and when the application is in windowed mode, rather than in full screen exclusive mode. The following table and illustration show a simplified comparison of memory bandwidth usages and system reads and writes of windowed applications that choose Flip Model versus the default usage Blt Model.

Blt mode present to DWM	**D3D9Ex Flip Mode Present to DWM**

Blt mode present to DWM	D3D9Ex Flip Mode Present to DWM
1. The application updates its frame (Write)	1. The application updates its frame (Write)
2. Direct3D runtime copies surface contents to a DWM redirection surface (Read, Write)	2. Direct3D runtime passes the application surface to DWM
3. After the shared surface copy is completed, DWM renders the application surface onto screen (Read, Write)	3. DWM renders the application surface onto screen (Read, Write)



Flip Mode Present reduces system memory usage by reducing the number of reads and writes by the Direct3D runtime for the windowed frame composition by DWM. This reduces the system power consumption and overall memory usage.

Applications can take advantage of Direct3D 9Ex Flip Mode present statistics enhancements when DWM is on, regardless of whether the application is in windowed mode or in full screen exclusive mode.

Programming Model and APIs

New video or frame rate-gauging applications that use Direct3D 9Ex APIs on Windows 7 can take advantage of the memory and power savings and of the improved presentation offered by Flip Mode Present when running on Windows 7. (When running on previous Windows versions, the Direct3D runtime defaults the application to Blt Mode Present.)

Flip Mode Present entails that the application can take advantage of real-time present statistics feedback and correction mechanisms when DWM is on. However, applications that use Flip Mode Present should be aware of limitations when they use concurrent GDI API rendering.

You can modify existing applications to take advantage of Flip Mode Present, with the same benefits and caveats as the newly developed applications.

How to Opt Into the Direct3D 9Ex Flip Model

Direct3D 9Ex applications that target Windows 7 can opt into the Flip Model by creating the swap chain with the **D3DSWAPEFFECT_FLIPEX** enumeration value. To opt into the Flip Model, applications specify the **D3DPRESENT_PARAMETERS** structure, and then pass a pointer to this structure when they call the **IDirect3D9Ex::CreateDeviceEx** API. This section describes how applications that target Windows 7 use **IDirect3D9Ex::CreateDeviceEx** to opt into the Flip Model. For more information about the **IDirect3D9Ex::CreateDeviceEx** API, see **IDirect3D9Ex::CreateDeviceEx on MSDN**.

For convenience, the syntax of **D3DPRESENT_PARAMETERS** and **IDirect3D9Ex::CreateDeviceEx** is repeated here.

syntax

```
HRESULT CreateDeviceEx(
  UINT Adapter,
  D3DDEVTYPE DeviceType,
  HWND hFocusWindow,
  DWORD BehaviorFlags,
  D3DPRESENT_PARAMETERS* pPresentationParameters,
  D3DDISPLAYMODEEX *pFullscreenDisplayMode,
  IDirect3DDevice9Ex **ppReturnedDeviceInterface
);
```

syntax

```
typedef struct D3DPRESENT_PARAMETERS {
    UINT BackBufferWidth, BackBufferHeight;
    D3DFORMAT BackBufferFormat;
```

```
    UINT BackBufferCount;
    D3DMULTISAMPLE_TYPE MultiSampleType;
    DWORD MultiSampleQuality;
    D3DSWAPEFFECT SwapEffect;
    HWND hDeviceWindow;
    BOOL Windowed;
    BOOL EnableAutoDepthStencil;
    D3DFORMAT AutoDepthStencilFormat;
    DWORD Flags;
    UINT FullScreen_RefreshRateInHz;
    UINT PresentationInterval;
} D3DPRESENT_PARAMETERS, *LPD3DPRESENT_PARAMETERS;
```

When you modify Direct3D 9Ex applications for Windows 7 to opt into the Flip Model, you should consider the following items about the specified members of D3DPRESENT_PARAMETERS:

BackBufferCount

(Windows 7 Only)

When **SwapEffect** is set to the new D3DSWAPEFFECT_FLIPEX swap chain effect type, the back buffer count should be equal or greater than 2, to prevent an application performance penalty as a result of waiting on the previous Present buffer to be released by DWM.

When the application also uses present statistics associated with D3DSWAPEFFECT_FLIPEX, we recommend that you set the back buffer count to from 2 to 4.

Using D3DSWAPEFFECT_FLIPEX on Windows Vista or previous operating system versions will return fail from CreateDeviceEx.

SwapEffect

(Windows 7 Only)

The new D3DSWAPEFFECT_FLIPEX swap chain effect type designates when an application is adopting Flip Mode Present to DWM. It allows the application more efficient usage of memory and power, and also enables the application to take advantage of full-screen present statistics in windowed mode. Full-screen application behavior is not affected. If Windowed is set to **TRUE** and **SwapEffect** is set to D3DSWAPEFFECT_FLIPEX, the runtime creates one extra back buffer and rotates whichever handle belongs to the buffer that becomes the front buffer at presentation time.

Flags

(Windows 7 Only)

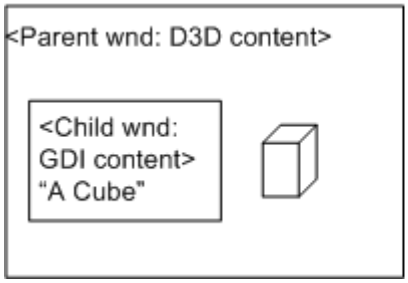
The D3DPRESENTFLAG_LOCKABLE_BACKBUFFER flag cannot be set if **SwapEffect** is set to the new **D3DSWAPEFFECT_FLIPEX** swap chain effect type.

Design Guidelines for Direct3D 9Ex Flip Model Applications

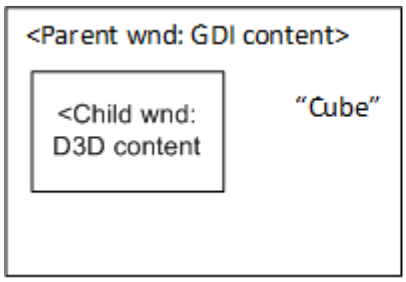
Use the guidelines in the following sections to design your Direct3D 9Ex Flip Model applications.

Use Flip Mode Present in a Separate HWND from Blt Mode Present

Applications should use Direct3D 9Ex Flip Mode Present in an HWND that is not also targeted by other APIs, including Blt Mode Present Direct3D 9Ex, other versions of Direct3D, or GDI. Flip Mode Present can be used to present to child windows; that is, applications can use Flip Model when it is not mixed with Blt Model in the same HWND, as shown in the following illustrations.

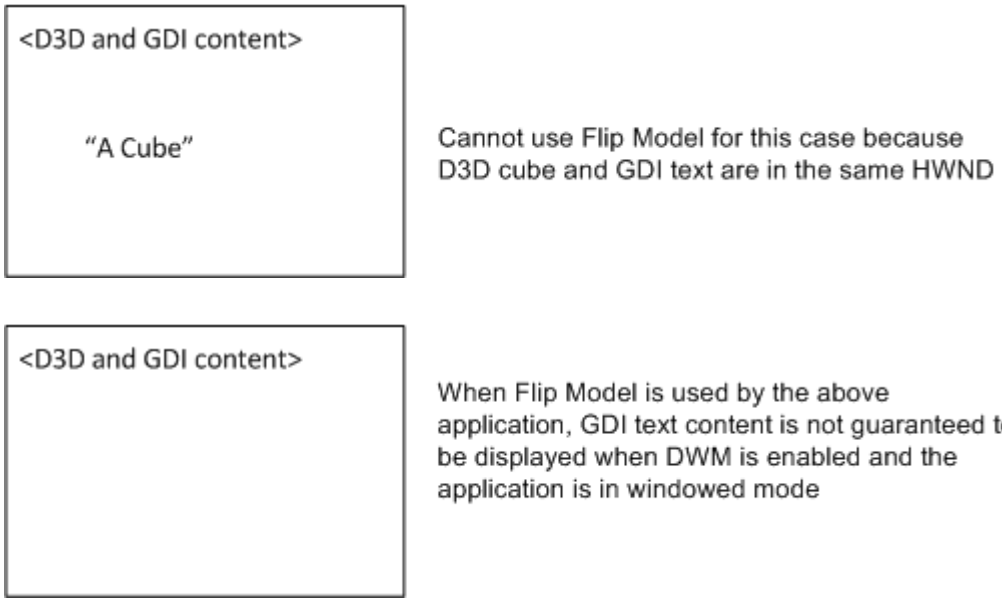


D3D9Ex Flip Model and GDI content each has its own HWND



D3D9Ex Flip Model and GDI content each has its own HWND

Because Blt Model maintains an additional copy of the surface, GDI and other Direct3D contents can be added to the same HWND through piecemeal updates from Direct3D and GDI. Using the Flip Model, only Direct3D 9Ex content in **D3DSWAPEFFECT_FLIPEX** swap chains that are passed to DWM will be visible. All other Blt Model Direct3D or GDI content updates will be ignored, as shown in the following illustrations.



Therefore, Flip Model should be enabled for swap chain buffers surfaces where the Direct3D 9Ex Flip Model alone renders to the entire HWND.

Do Not Use Flip Model with GDI's ScrollWindow or ScrollWindowEx

Some Direct3D 9Ex applications use GDI's ScrollWindow or ScrollWindowEx functions to update window contents when a user scroll event is triggered. ScrollWindow and ScrollWindowEx perform blts of window contents on screen as a window is scrolled. These functions also require Blt Model updates for GDI and Direct3D 9Ex content. Applications that use either function will not necessarily display visible window contents scrolling on screen when the application is in windowed mode and DWM is enabled. We recommend that you not use GDI's ScrollWindow and ScrollWindowEx APIs in your applications, and instead redraw their contents on screen in response to scrolling.

Use One D3DSWAPEFFECT_FLIPEX Swap Chain per HWND

Applications that use Flip Model should not use multiple Flip Model swap chains targeting the same HWND.

Frame Synchronization of Direct3D 9Ex Flip Model Applications

Present statistics are frame timing information that media applications use to synchronize video and audio streams and recover from video playback glitches. To enable present statistics availability, the Direct3D 9Ex application must ensure that the

BehaviorFlags parameter that the application passes to **IDirect3D9Ex::CreateDeviceEx** contains the device behavior flag D3DCREATE_ENABLE_PRESENTSTATS.

For convenience, the syntax of **IDirect3D9Ex::CreateDeviceEx** is repeated here.

syntax

```
HRESULT CreateDeviceEx(
  UINT Adapter,
  D3DDEVTYPE DeviceType,
  HWND hFocusWindow,
  DWORD BehaviorFlags,
  D3DPRESENT_PARAMETERS* pPresentationParameters,
  D3DDISPLAYMODEEX *pFullscreenDisplayMode,
  IDirect3DDevice9Ex **ppReturnedDeviceInterface
);
```

Direct3D 9Ex Flip Model adds the D3DPRESENT_FORCEIMMEDIATE presentation flag that enforces the D3DPRESENT_INTERVAL_IMMEDIATE presentation-flag behavior. The Direct3D 9Ex application specifies these presentation flags in the *dwFlags* parameter that the application passes to **IDirect3DDevice9Ex::PresentEx**, as shown here.

syntax

```
HRESULT PresentEx(
  CONST RECT *pSourceRect,
  CONST RECT *pDestRect,
  HWND hDestWindowOverride,
  CONST RGNDATA *pDirtyRegion,
  DWORD dwFlags
);
```

When you modify your Direct3D 9Ex application for Windows 7, you should consider the following information about the specified D3DPRESENT presentation flags:

D3DPRESENT_DONOTFLIP

This flag is available only in full-screen mode or

(Windows 7 Only)

when the application sets the **SwapEffect** member of **D3DPRESENT_PARAMETERS** to **D3DSWAPEFFECT_FLIPEX** in a call to **CreateDeviceEx**.

D3DPRESENT_FORCEIMMEDIATE

(Windows 7 Only)

This flag can be specified only if the application sets the **SwapEffect** member of D3DPRESENT_PARAMETERS to D3DSWAPEFFECT_FLIPEX in a call to CreateDeviceEx. The application can use this flag to immediately update a surface with several frames later in the DWM Present queue, essentially skipping intermediate frames.

Windowed FlipEx-enabled applications can use this flag to immediately update a surface with a frame that is later in the DWM Present queue, skipping intermediate frames. This is especially useful for media applications that want to discard frames that have been detected as late and present subsequent frames at composition time. IDirect3DDevice9Ex::PresentEx returns invalid parameter error if this flag is improperly specified.

To obtain present statistics information, the application obtains the D3DPRESENTSTATS structure by calling the IDirect3DSwapChain9Ex::GetPresentStatistics API.

The D3DPRESENTSTATS structure contains statistics about IDirect3DDevice9Ex::PresentEx calls. The device must be created by using a IDirect3D9Ex::CreateDeviceEx call with the D3DCREATE_ENABLE_PRESENTSTATS flag. Otherwise, the data returned by GetPresentStatistics is undefined. A Flip-Model-enabled Direct3D 9Ex swap chain provides present statistics information in both windowed and full-screen modes.

For Blt-Model-enabled Direct3D 9Ex swap chains in windowed mode, all D3DPRESENTSTATS structure values will be zeroes.

For FlipEx present statistics, GetPresentStatistics returns D3DERR_PRESENT_STATISTICS_DISJOINT in the following situations:

- First call to GetPresentStatistics ever, which indicates the beginning of a sequence
- DWM transition from on to off
- Mode change: either windowed mode to or from full screen or full screen to full screen transitions

For convenience, the syntax of GetPresentStatistics is repeated here.

syntax

```
HRESULT GetPresentStatistics(
  D3DPRESENTSTATS * pPresentationStatistics
);
```

The IDirect3DSwapChain9Ex::GetLastPresentCount method returns the last PresentCount, that is, the Present ID of the last successful Present call that was made by a display device that is associated with the swap chain. This Present ID is the value of the

PresentCount member of the D3DPRESENTSTATS structure. For Blt-Model-enabled Direct3D 9Ex swap chains, while in windowed mode, all **D3DPRESENTSTATS** structure values will be zeroes.

For convenience, the syntax of IDirect3DSwapChain9Ex::GetLastPresentCount is repeated here.

syntax

```
HRESULT GetLastPresentCount(
  UINT * pLastPresentCount
);
```

When you modify your Direct3D 9Ex application for Windows 7, you should consider the following information about the D3DPRESENTSTATS structure:

- The PresentCount value that GetLastPresentCount returns does not update when a PresentEx call with D3DPRESENT_DONOTWAIT specified in the *dwFlags* parameter returns failure.
- When PresentEx is called with D3DPRESENT_DONOTFLIP, a GetPresentStatistics call succeeds but does not return an updated D3DPRESENTSTATS structure when the application is in windowed mode.
- **PresentRefreshCount** versus **SyncRefreshCount** in D3DPRESENTSTATS:
  - **PresentRefreshCount** is equal to **SyncRefreshCount** when the application presents on every vsync.
  - **SyncRefreshCount** is obtained on the vsync interval when the present was submitted, **SyncQPCTime** is approximately the time associated with the vsync interval.

syntax

```
typedef struct _D3DPRESENTSTATS {
    UINT PresentCount;
    UINT PresentRefreshCount;
    UINT SyncRefreshCount;
    LARGE_INTEGER SyncQPCTime;
    LARGE_INTEGER SyncGPUTime;
} D3DPRESENTSTATS;
```

Frame Synchronization for Windowed Applications When DWM is Off

When DWM is off, windowed applications display directly to the monitor screen without going through a flip chain. In Windows Vista, there is no support for obtaining frame statistics information for windowed applications when DWM is off. To maintain an API where applications need not be DWM- aware, Windows 7 will return frame statistics information for windowed applications when DWM is off. The frame statistics returned when DWM is off are estimations only.

Walk-Through of a Direct3D 9Ex Flip Model and Present Statistics Sample

To opt into FlipEx presentation for Direct3D 9Ex sample

1. Ensure the sample application is running on Windows 7 or later operating system version.
2. Set the **SwapEffect** member of **D3DPRESENT_PARAMETERS** to **D3DSWAPEFFECT_FLIPEX** in a call to **CreateDeviceEx**.

```
C++
```

```
    OSVERSIONINFO version;
    ZeroMemory(&version, sizeof(version));
    version.dwOSVersionInfoSize = sizeof(version);
    GetVersionEx(&version);

    // Sample would run only on Win7 or higher
    // Flip Model present and its associated present statistics behavior are
only available on Windows 7 or higher operating system
    bool bIsWin7 = (version.dwMajorVersion > 6) ||
        ((version.dwMajorVersion == 6) && (version.dwMinorVersion >= 1));

    if (!bIsWin7)
    {
        MessageBox(NULL, L"This sample requires Windows 7 or higher", NULL,
MB_OK);
        return 0;
    }
```

To also opt into FlipEx associated Present Statistics for Direct3D 9Ex sample

- Set D3DCREATE_ENABLE_PRESENTSTATS in the *BehaviorFlags* parameter of **CreateDeviceEx**.

```
C++
```

```
    // Set up the structure used to create the D3DDevice
    D3DPRESENT_PARAMETERS d3dpp;
```

```
    ZeroMemory(&d3dpp, sizeof(d3dpp));

    d3dpp.Windowed = TRUE;
    d3dpp.SwapEffect = D3DSWAPEFFECT_FLIPEX;        // Opts into Flip Model
present for D3D9Ex swapchain
    d3dpp.BackBufferFormat = D3DFMT_X8R8G8B8;
    d3dpp.BackBufferWidth = 256;
    d3dpp.BackBufferHeight = 256;
    d3dpp.BackBufferCount = QUEUE_SIZE;
    d3dpp.PresentationInterval = D3DPRESENT_INTERVAL_ONE;

    g_iWidth = d3dpp.BackBufferWidth;
    g_iHeight = d3dpp.BackBufferHeight;

    // Create the D3DDevice with present statistics enabled - set
D3DCREATE_ENABLE_PRESENTSTATS for behaviorFlags parameter
    if(FAILED(g_pD3D->CreateDeviceEx(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL,
hWnd,
                                     D3DCREATE_HARDWARE_VERTEXPROCESSING |
D3DCREATE_ENABLE_PRESENTSTATS,
                                     &d3dpp, NULL, &g_pd3dDevice)))
    {
        return E_FAIL;
    }
```

To avoid, detect and recover from glitches

1. Queue Present calls: recommended backbuffer count is from 2 to 4.
2. Direct3D 9Ex sample adds an implicit backbuffer, actual Present queue length is backbuffer count + 1.
3. Create helper Present queue structure to store all successfully submitted Present's Present ID (PresentCount) and associated, calculated/expected PresentRefreshCount.
4. To detect glitch occurrence:
    - Call GetPresentStatistics.
    - Get the Present ID (PresentCount) and vsync count where the frame is shown (PresentRefreshCount) of the frame whose present statistics is obtained.
    - Retrieve the expected PresentRefreshCount (TargetRefresh in sample code) associated with the Present ID.
    - If actual PresentRefreshCount is later than expected, a glitch has occurred.
5. To recover from glitch:
    - Calculate how many frames to skip (g_ iImmediates variable in sample code).
    - Present the skipped frames with interval D3DPRESENT_FORCEIMMEDIATE.

Considerations for glitch detection and recovery

1. Glitch recovery takes N (g_iQueueDelay variable in sample code) number of Present calls where N (g_iQueueDelay) equals g_iImmediates plus length of the Present queue, that is:
    - Skipping frames with Present interval D3DPRESENT_FORCEIMMEDIATE, plus
    - Queued Presents that need to be processed
2. Set a limit to the glitch length (GLITCH_RECOVERY_LIMIT in sample). If the sample application cannot recover from a glitch that is too long (that is say, 1 second, or 60 vsyncs on 60Hz monitor), jump over the intermittent animation and reset the Present helper queue.

C++

```
VOID Render()
{
    g_pd3dDevice->Clear(0, NULL, D3DCLEAR_TARGET, D3DCOLOR_XRGB(0, 0, 0),
1.0f, 0);

    g_pd3dDevice->BeginScene();

    // Compute new animation parameters for time and frame based animations

    // Time-based is a difference between base and current SyncRefreshCount
    g_aTimeBasedHistory[g_iBlurHistoryCounter] = g_iStartFrame +
g_LastSyncRefreshCount - g_SyncRefreshCount;
    // Frame-based is incrementing frame value
    g_aFrameBasedHistory[g_iBlurHistoryCounter] = g_iStartFrame +
g_iFrameNumber;

    RenderBlurredMesh(TRUE);    // Time-based
    RenderBlurredMesh(FALSE);   // Frame-based

    g_iBlurHistoryCounter = (g_iBlurHistoryCounter + 1) % BLUR_FRAMES;

    DrawText();

    g_pd3dDevice->EndScene();

    // Performs glitch recovery if glitch was detected
    if (g_bGlitchRecovery && (g_iImmediates > 0))
    {
        // If we have present immediates queued as a result of glitch
detected, issue forceimmediate Presents for glitch recovery
        g_pd3dDevice->PresentEx(NULL, NULL, NULL, NULL,
D3DPRESENT_FORCEIMMEDIATE);
        g_iImmediates--;
        g_iShowingGlitchRecovery = MESSAGE_SHOW;
    }
```

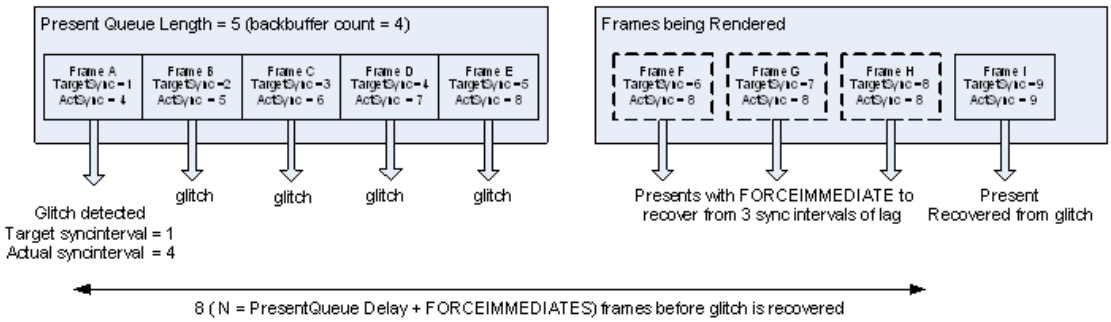
```
    // Otherwise, Present normally
    else
    {
        g_pd3dDevice->PresentEx(NULL, NULL, NULL, NULL, 0);
    }

    // Add to helper Present queue: PresentID + expected present refresh
count of last submitted Present
    UINT PresentCount;
    g_pd3dSwapChain->GetLastPresentCount(&PresentCount);
    g_Queue.QueueFrame(PresentCount, g_TargetRefreshCount);

    // QueueDelay specifies # Present calls to be processed before another
glitch recovery attempt
    if (g_iQueueDelay > 0)
    {
        g_iQueueDelay--;
    }

    if (g_bGlitchRecovery)
    {
        // Additional DONOTFLIP presents for frame conversions, which
basically follows the same logic, but without rendering
        for (DWORD i = 0; i < g_iDoNotFlipNum; i++)
        {
            if (g_TargetRefreshCount != -1)
            {
                g_TargetRefreshCount++;
                g_iFrameNumber++;
                g_aTimeBasedHistory[g_iBlurHistoryCounter] = g_iStartFrame +
g_LastSyncRefreshCount - g_SyncRefreshCount;
                g_aFrameBasedHistory[g_iBlurHistoryCounter] = g_iStartFrame
+ g_iFrameNumber;
                g_iBlurHistoryCounter = (g_iBlurHistoryCounter + 1) %
BLUR_FRAMES;
            }

            if (g_iImmediates > 0)
            {
                g_pd3dDevice->PresentEx(NULL, NULL, NULL, NULL,
D3DPRESENT_FORCEIMMEDIATE | D3DPRESENT_DONOTFLIP);
                g_iImmediates--;
            }
            else
            {
                g_pd3dDevice->PresentEx(NULL, NULL, NULL, NULL,
D3DPRESENT_DONOTFLIP);
            }
            UINT PresentCount;
            g_pd3dSwapChain->GetLastPresentCount(&PresentCount);
            g_Queue.QueueFrame(PresentCount, g_TargetRefreshCount);

            if (g_iQueueDelay > 0)
            {
                g_iQueueDelay--;
```

```
            }
        }
    }

    // Check Present Stats info for glitch detection
    D3DPRESENTSTATS PresentStats;

    // Obtain present statistics information for successfully displayed
presents
    HRESULT hr = g_pd3dSwapChain->GetPresentStats(&PresentStats);

    if (SUCCEEDED(hr))
    {
        // Time-based update
        g_LastSyncRefreshCount = PresentStats.SyncRefreshCount;
        if ((g_SyncRefreshCount == -1) && (PresentStats.PresentCount != 0))
        {
            // First time SyncRefreshCount is reported, use it as base
            g_SyncRefreshCount = PresentStats.SyncRefreshCount;
        }

        // Fetch frame from the queue...
        UINT TargetRefresh =
g_Queue.DequeueFrame(PresentStats.PresentCount);

        // If PresentStats returned a really old frame that we no longer
have in the queue, just don't do any glitch detection
        if (TargetRefresh == FRAME_NOT_FOUND)
            return;

        if (g_TargetRefreshCount == -1)
        {
            // This is first time issued frame is confirmed by present
stats, so fill target refresh count for all frames in the queue
            g_TargetRefreshCount =
g_Queue.FillRefreshCounts(PresentStats.PresentCount, g_SyncRefreshCount);
        }
        else
        {
            g_TargetRefreshCount++;
            g_iFrameNumber++;

            // To determine whether we're glitching, see if our estimated
refresh count is confirmed
            // if the frame is displayed later than the expected vsync count
            if (TargetRefresh < PresentStats.PresentRefreshCount)
            {
                // then, glitch is detected!

                // If glitch is too big, don't bother recovering from it,
just jump animation
                if ((PresentStats.PresentRefreshCount - TargetRefresh) >
GLITCH_RECOVERY_LIMIT)
                {
                    g_iStartFrame += PresentStats.SyncRefreshCount -
```

```
g_SyncRefreshCount;
                    ResetAnimation();
                    if (g_bGlitchRecovery)
                        g_iGlitchesInaRow++;
                }
                // Otherwise, compute number of immediate presents to
recover from it -- if we?re not still trying to recover from another glitch
                else if (g_iQueueDelay == 0)
                {
                      // skip frames to catch up to expected refresh count
                    g_iImmediates = PresentStats.PresentRefreshCount -
TargetRefresh;
                    // QueueDelay specifies # Present calls before another
glitch recovery
                    g_iQueueDelay = g_iImmediates + QUEUE_SIZE;
                    if (g_bGlitchRecovery)
                        g_iGlitchesInaRow++;
                }
            }
            else
            {
                // No glitch, reset glitch count
                g_iGlitchesInaRow = 0;
            }
        }
    }
    else if (hr == D3DERR_PRESENT_STATISTICS_DISJOINT)
    {
        // D3DERR_PRESENT_STATISTICS_DISJOINT means measurements should be
started from the scratch (could be caused by mode change or DWM on/off
transition)
        ResetAnimation();
    }

    // If we got too many glitches in a row, reduce framerate conversion
factor (that is, render less frames)
    if (g_iGlitchesInaRow == FRAMECONVERSION_GLITCH_LIMIT)
    {
        if (g_iDoNotFlipNum < FRAMECONVERSION_LIMIT)
        {
            g_iDoNotFlipNum++;
        }
        g_iGlitchesInaRow = 0;
        g_iShowingDoNotFlipBump = MESSAGE_SHOW;
    }
}
```

Sample scenario

- The following illustration shows an application with backbuffer count of 4. The actual Present queue length is therefore 5.



Frame A is targeted to go on screen on sync interval count of 1 but was detected that it was shown on sync interval count of 4. Therefore a glitch has occurred. Subsequent 3 frames are presented with D3DPRESENT_INTERVAL_FORCEIMMEDIATE. The glitch should take a total of 8 Present calls before it is recovered - the next frame will be shown as per its targeted sync interval count.

Summary of Programming Recommendations for Frame Synchronization

- Create a backup list of all the LastPresentCount IDs (obtained via **GetLastPresentCount**) and associated estimated PresentRefreshCount of all the Presents submitted.

  > ⓘ **Note**
  >
  > When the application calls **PresentEx** with D3DPRESENT_DONOTFLIP, the **GetPresentStatistics** call succeeds but does not return an updated **D3DPRESENTSTATS** structure when the application is in windowed mode.

- Call **GetPresentStatistics** to obtain the actual PresentRefreshCount associated with each Present ID of frames shown, to make sure that the application handles failure returns from the call.

- If actual PresentRefreshCount is later than estimated PresentRefreshCount, a glitch is detected. Compensate by submitting lagging frames' Present with D3DPRESENT_FORCEIMMEDIATE.

- When one frame is presented late in the Present queue, all subsequent queued frames will be presented late. D3DPRESENT_FORCEIMMEDIATE will correct only the next frame to be presented after all the queued frames. Therefore, the Present queue or backbuffer count should not be too long -- so there are less frame glitches to catch up with. The optimal backbuffer count is 2 to 4.

- If estimated PresentRefreshCount is later than the actual PresentRefreshCount, DWM throttling might have occurred. The following solutions are possible:
  - reducing Present queue length
  - reducing GPU memory requirements with any other means besides reducing Present Queue length (that is, decreasing quality, removing effects, and so on)
  - specifying **DwmEnableMMCSS** to prevent DWM throttling in general

- Verify application display functionality and frame statistics performance in the following scenarios:
  - with DWM on and off
  - fullscreen exclusive and windowed modes
  - lower capability hardware

- When applications cannot recover from large numbers of glitched frames with D3DPRESENT_FORCEIMMEDIATE Present, they can potentially perform the following operations:
  - reduce CPU and GPU usage by rendering with less workload.
  - in the case of video decode, decode faster by reducing quality and, therefore, CPU and GPU usage.

Conclusion about Direct3D 9Ex Improvements

On Windows 7, applications that display video or gauge frame rate during presentation can opt into Flip Model. The present statistics improvements that are associated with Flip Model Direct3D 9Ex can benefit applications that synchronize presentation per frame rate, with real time feedback for glitch detection and recovery. Developers that adopt the Direct3D 9Ex Flip Model should take targeting a separate HWND from GDI content and frame rate synchronization into account. Refer to details in this topic, and MSDN documentation. For additional documentation, see DirectX Developer Center on MSDN.

Call to Action

We encourage you to use Direct3D 9Ex Flip Model and its present statistics on Windows 7 when you create applications that attempt to synchronize presentation frame rate or recover from display glitches.

Related topics

DirectX Developer Center on MSDN

Feedback

Was this page helpful? Yes No

Get help at Microsoft Q&A

Surface sharing between Windows graphics APIs

Article • 05/24/2021

This topic provides a technical overview of interoperability using surface sharing between Windows graphics APIs, including Direct3D 11, Direct2D, DirectWrite, Direct3D 10, and Direct3D 9Ex. If you already have a working knowledge of these APIs, this paper can help you use multiple APIs to render to the same surface in an application designed for the Windows 7 or Windows Vista operating systems. This topic also provides best practice guidelines and pointers to additional resources.

> **Note**
>
> For Direct2D and DirectWrite interoperability on the DirectX 11.1 runtime, you can use **Direct2D devices and device contexts** to render directly to Direct3D 11 devices.

This topic contains the following sections.

Introduction

In this document, Windows graphics API interoperability refers to the sharing of the same rendering surface by different APIs. This kind of interoperability enables applications to create compelling displays by leveraging multiple Windows graphics APIs, and to ease migration to new technologies by maintaining compatibility with existing APIs.

In Windows 7 (and Windows Vista SP2 with Windows 7 Interop Pack, Vista 7IP), the graphics rendering APIs are Direct3D 11, Direct2D, Direct3D 10.1, Direct3D 10.0,

Direct3D 9Ex, Direct3D 9c and earlier Direct3D APIs, as well GDI and GDI+. Windows Imaging Component (WIC) and DirectWrite are related technologies for image processing, and Direct2D performs text rendering. DirectX Video Acceleration API (DXVA), based on Direct3D 9c and Direct3D 9Ex, is used for video processing.

As Windows graphics APIs evolve towards being Direct3D-based, Microsoft is investing more effort in ensuring interoperability across APIs. Newly developed Direct3D APIs and higher level APIs based on Direct3D APIs also provide support where needed for bridging compatibility with older APIs. To illustrate, Direct2D applications can use Direct3D 10.1 by sharing a Direct3D 10.1 device. Also, Direct3D 11, Direct2D, and Direct3D 10.1 APIs can all take advantage of DirectX Graphics Infrastructure (DXGI) 1.1, which enables synchronized shared surfaces that fully support interoperability among these APIs. DXGI 1.1-based APIs interoperate with GDI, and with GDI+ by association, by obtaining the GDI device context from a DXGI 1.1 surface. For more information, see the DXGI and GDI interoperability documentation available on MSDN.

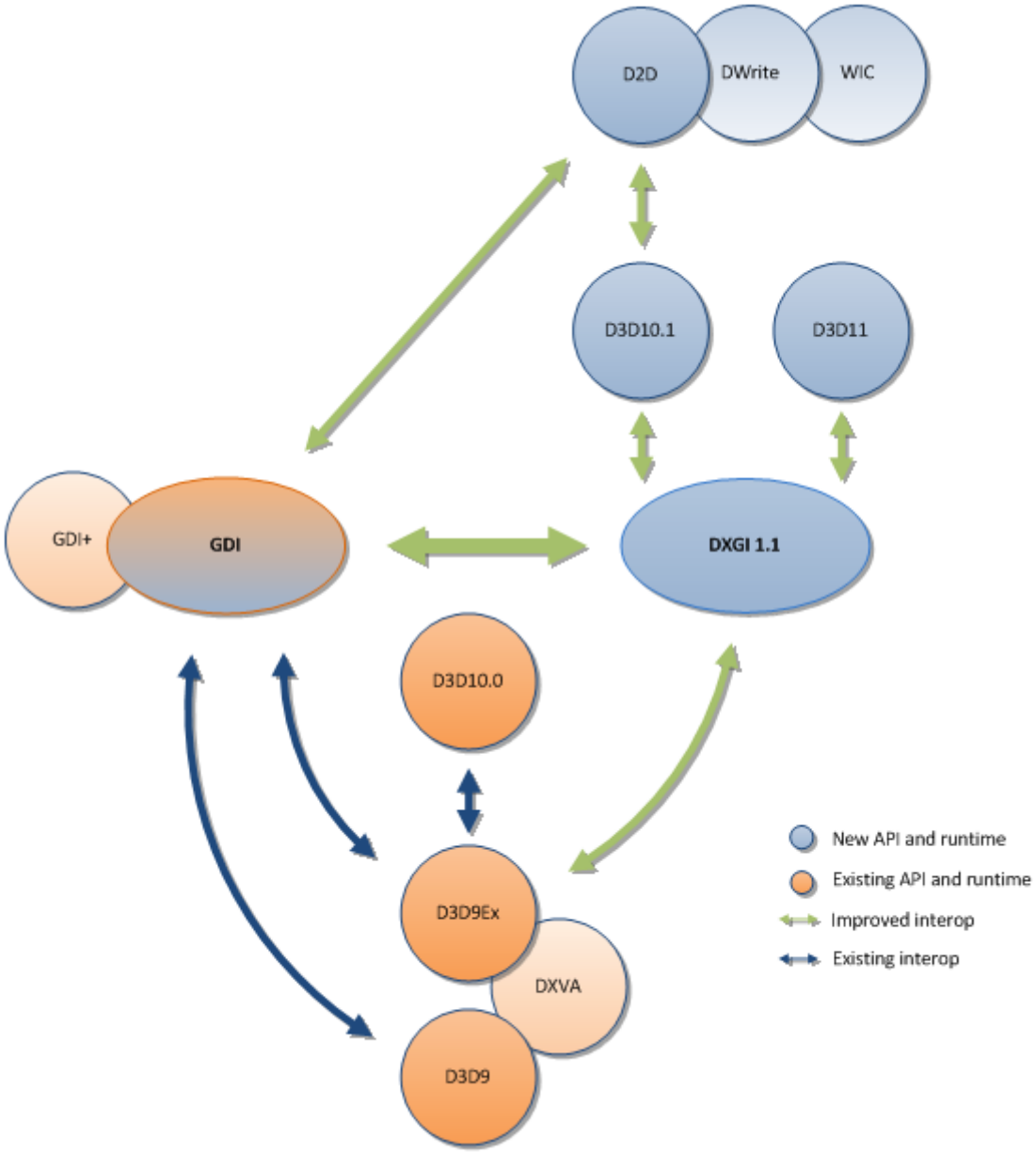
Unsynchronized surface sharing is supported by the Direct3D 9Ex runtime. DXVA-based video applications can use the Direct3D 9Ex and DXGI interoperability helper for Direct3D 9Ex-based DXVA interoperability with Direct3D 11 for compute shader, or can interoperate with Direct2D for 2D controls or text rendering. WIC and DirectWrite also interoperate with GDI, Direct2D, and by association, other Direct3D APIs.

Direct3D 10.0, Direct3D 9c, and older Direct3D runtimes do not support shared surfaces. System memory copies will continue to be used for interoperability with GDI or DXGI-based APIs.

Note that the interoperability scenarios within this document refer to multiple graphics APIs rendering to a shared rendering surface, rather than to the same application window. Synchronization for separate APIs targeting different surfaces that are then composited onto the same window is outside the scope of this paper.

API Interoperability Overview

Surface sharing interoperability of Windows graphics APIs can be described in terms of API-to-API scenarios and the corresponding interoperability functionality. As of Windows 7 and beginning with Windows Vista SP2 with 7IP, new APIs and associated runtimes include Direct2D and related technologies: Direct3D 11 and DXGI 1.1. GDI performance was also improved in Windows 7. Direct3D 10.1 was introduced in Windows Vista SP1. The following diagram shows the interoperability support between APIs.



In this diagram, arrows show interoperability scenarios in which the same surface can be accessible by the connected APIs. Blue arrows indicate interoperability mechanisms introduced in Windows Vista. Green arrows indicate interoperability support for new APIs or improvements that help older APIs to interoperate with newer APIs. For example, green arrows represent device sharing, synchronized shared surface support, Direct3D 9Ex/DXGI synchronization helper, and obtaining a GDI device context from a compatible surface.

Interoperability Scenarios

As of Windows 7 and Windows Vista 7IP, mainstream offerings from Windows graphics APIs support multiple APIs rendering to the same DXGI 1.1 surface.

Direct3D 11, Direct3D 10.1, Direct2D — Interoperability with each other

Direct3D 11, Direct3D 10.1 and Direct2D APIs (and its related APIs such as DirectWrite and WIC) can interoperate with each other using either Direct3D 10.1 device sharing or synchronized shared surfaces.

Direct3D 10.1 Device Sharing with Direct2D

Device sharing between Direct2D and Direct3D 10.1 allows an application to use both APIs to seamlessly and efficiently render onto the same DXGI 1.1 surface, using the same underlying Direct3D device object. Direct2D provides the ability to call Direct2D APIs using an existing Direct3D 10.1 device, leveraging the fact that Direct2D is built on top of Direct3D 10.1 and DXGI 1.1 runtimes. The following code snippets illustrate how Direct2D obtains the Direct3D 10.1 device render target from a DXGI 1.1 surface associated with the device. The Direct3D 10.1 device render target can execute Direct2D drawing calls between BeginDraw and EndDraw APIs.

C++

```
// Direct3D 10.1 Device and Swapchain creation
HRESULT hr = D3D10CreateDeviceandSwapChain1(
                pAdapter,
                DriverType,
                Software,
                D3D10_CREATE_DEVICE_BGRA_SUPPORT,
                featureLevel,
                D3D10_1_SDK_VERSION,
                pSwapChainDesc,
                &pSwapChain,
                &pDevice
                );

hr = pSwapChain->GetBuffer(
        0,
        __uuidof(IDXGISurface),
        (void **)&pDXGIBackBuffer
        ));

// Direct3D 10.1 API rendering calls
...

hr = D2D1CreateFactory(
        D2D1_FACTORY_TYPE_SINGLE_THREADED,
        &m_spD2DFactory
        ));

pD2DFactory->CreateDxgiSurfaceRenderTarget(
        pDXGIBackBuffer,
        &renderTargetProperties,
        &pD2DBackBufferRenderTarget
        ));
```

```
...

pD2DBackBufferRenderTarget->BeginDraw();
//Direct2D API rendering calls
...

pD2DBackBufferRenderTarget->EndDraw();

pSwapChain->Present(0, 0);
```

Remarks

- The associated Direct3D 10.1 device must support BGRA format. That device was created by calling D3D10CreateDevice1 with parameter D3D10_CREATE_DEVICE_BGRA_SUPPORT. BGRA format is supported starting with Direct3D 10 feature level 9.1.
- The application should not create multiple ID2D1RenderTargets associating to the same Direct3D10.1 device.
- For optimal performance, keep at least one resource around at all times, such as textures or surfaces associated with the device.

Device sharing is suitable for in-process, single-threaded usage of one rendering device shared by both Direct3D 10.1 and Direct2D rendering APIs. Synchronized shared surfaces enable multi-threaded, in-process and out-of-process usage of multiple rendering devices used by Direct3D 10.1, Direct2D and Direct3D 11 APIs.

Another method of Direct3D 10.1 and Direct2D interoperability is by using ID3D1RenderTarget::CreateSharedBitmap, which creates an ID2D1Bitmap object from IDXGISurface. You can write a Direct3D10.1 scene to the bitmap and render it with Direct2D. For more information, see ID2D1RenderTarget::CreateSharedBitmap Method.

Direct2D Software Rasterization

Device sharing with Direct3D 10.1 is not supported when using the Direct2D software renderer, for example, by specifying D2D1_RENDER_TARGET_USAGE_FORCE_SOFTWARE_RENDERING in D2D1_RENDER_TARGET_USAGE when creating a Direct2D render target.

Direct2D can use the WARP10 software rasterizer to share device with Direct3D 10 or Direct3D 11, but the performance declines significantly.

DXGI 1.1 Synchronized Shared Surfaces

Direct3D 11, Direct3D 10.1 and Direct2D APIs all use DXGI 1.1, which provides the functionality to synchronize reading from and writing to the same video memory surface (DXGISurface1) by two or more Direct3D devices. The rendering devices using synchronized shared surfaces can be Direct3D 10.1 or Direct3D 11 devices, each running in the same process or cross-processes.

Applications can use synchronized shared surfaces to interoperate between any DXGI 1.1-based devices, such as Direct3D 11 and Direct3D 10.1, or between Direct3D 11 and Direct2D, by obtaining the Direct3D 10.1 device from Direct2D render target object.

In Direct3D 10.1 and later APIs, to use DXGI 1.1, ensure that the Direct3D device is created using a DXGI 1.1 adapter object, which is enumerated from the DXGI 1.1 factory object. Call CreateDXGIFactory1 to create the IDXGIFactory1 object, and EnumAdapters1 to enumerate the IDXGIAdapter1 object. The IDXGIAdapter1 object needs to be passed in as part of D3D10CreateDevice or D3D10CreateDeviceAndSwapChain call. For more information on DXGI 1.1 APIs, please refer to the Programming Guide for DXGI.

APIs

D3D10_RESOURCE_MISC_SHARED_KEYEDMUTEX

When creating the synchronized shared resource, set D3D10_RESOURCE_MISC_SHARED_KEYEDMUTEX in D3D10_RESOURCE_MISC_FLAG.

C++

```
typedef enum D3D10_RESOURCE_MISC_FLAG {
    D3D10_RESOURCE_MISC_GENERATE_MIPS      = 0x1L,
    D3D10_RESOURCE_MISC_SHARED             = 0x2L,
    D3D10_RESOURCE_MISC_TEXTURECUBE        = 0x4L,
    D3D10_RESOURCE_MISC_SHARED_KEYEDMUTEX  = 0x10L,
    D3D10_RESOURCE_MISC_GDI_COMPATIBLE     = 0x20L,
}   D3D10_RESOURCE_MISC_FLAG;
```

D3D10_RESOURCE_MISC_SHARED_KEYEDMUTEX

Enables the resource created to be synchronized using the IDXGIKeyedMutex::AcquireSync and ReleaseSync APIs. The following resource creation Direct3D 10.1 APIs that all take a D3D10_RESOURCE_MISC_FLAG parameter have been extended to support the new flag.

- ID3D10Device1::CreateTexture1D
- ID3D10Device1::CreateTexture2D
- ID3D10Device1::CreateTexture3D
- ID3D10Device1::CreateBuffer

If any of the listed functions are called with the D3D10_RESOURCE_MISC_SHARED_KEYEDMUTEX flag set, the interface returned can be queried for an IDXGIKeyedMutex interface, which implements AcquireSync and ReleaseSync APIs to synchronize access to the surface. The device creating the surface and any other device opening the surface (using OpenSharedResource) is required to call IDXGIKeyedMutex::AcquireSync before any rendering commands to the surface, and IDXGIKeyedMutex::ReleaseSync when it is done rendering.
WARP and REF devices do not support shared resources. Attempting to create a resource with this flag on either a WARP or REF device will cause the create method to return an E_OUTOFMEMORY error code.

IDXGIKEYEDMUTEX INTERFACE

A new interface in DXGI 1.1, IDXGIKeyedMutex, represents a keyed mutex, which allows exclusive access to a shared resource that is used by multiple devices. For reference documentation about this interface and its two methods, AcquireSync and ReleaseSync, see IDXGIKeyedMutex.

Sample: Synchronized Surface Sharing Between two Direct3D 10.1 Devices

The example below illustrates sharing a surface between two Direct3D 10.1 devices. The synchronized shared surface is created by a Direct3D10.1 device.

C++

```
// Create Sync Shared Surface using Direct3D10.1 Device 1.
D3D10_TEXTURE2D_DESC desc;
ZeroMemory( &desc, sizeof(desc) );
desc.Width = width;
desc.Height = height;
desc.MipLevels = 1;
desc.ArraySize = 1;
// must match swapchain format in order to CopySubresourceRegion.
desc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
desc.SampleDesc.Count = 1;
desc.Usage = D3D10_USAGE_DEFAULT;
// creates 2D texture as a Synchronized Shared Surface.
desc.MiscFlags = D3D10_RESOURCE_MISC_SHARED_KEYEDMUTEX;
desc.BindFlags = D3D10_BIND_RENDER_TARGET | D3D10_BIND_SHADER_RESOURCE;
ID3D10Texture2D* g_pShared = NULL;
g_pd3dDevice1->CreateTexture2D( &desc, NULL, &g_pShared );

// QI IDXGIResource interface to synchronized shared surface.
IDXGIResource* pDXGIResource = NULL;
g_pShared->QueryInterface(__uuidof(IDXGIResource), (LPVOID*)
&pDXGIResource);

// obtain handle to IDXGIResource object.
```

```
pDXGIResource->GetSharedHandle(&g_hsharedHandle);
pDXGIResource->Release();
if ( !g_hsharedHandle )
    return E_FAIL;

// QI IDXGIKeyedMutex interface of synchronized shared surface's resource
handle.
hr = g_pShared->QueryInterface( __uuidof(IDXGIKeyedMutex),
    (LPVOID*)&g_pDXGIKeyedMutex_dev1 );
If ( FAILED( hr ) || ( g_pDXGIKeyedMutex_dev1 == NULL ) )
    return E_FAIL;
```

The same Direct3D10.1 device can obtain the synchronized shared surface for rendering by calling AcquireSync, then releasing the surface for the other device's rendering by calling ReleaseSync. When not sharing the synchronized shared surface with any other Direct3D device, the creator can obtain and release the synchronized shared surface (to start and end rendering) by acquiring and release using the same key value.

C++

```
// Obtain handle to Sync Shared Surface created by Direct3D10.1 Device 1.
hr = g_pd3dDevice2->OpenSharedResource(
g_hsharedHandle,__uuidof(ID3D10Texture2D),
                                        (LPVOID*) &g_pdev2Shared);
if (FAILED (hr))
    return hr;
hr = g_pdev2Shared->QueryInterface( __uuidof(IDXGIKeyedMutex),
                                    (LPVOID*) &g_pDXGIKeyedMutex_dev2);
if( FAILED( hr ) || ( g_pDXGIKeyedMutex_dev2 == NULL ) )
    return E_FAIL;

// Rendering onto Sync Shared Surface from D3D10.1 Device 1 using D3D10.1
Device 2.
UINT acqKey = 1;
UINT relKey = 0;
DWORD timeOut = 5;
DWORD result = g_pDXGIKeyedMutex_dev2->AcquireSync(acqKey, timeOut);
if ( result == WAIT_OBJECT_0 )
    // Rendering calls using Device 2.
else
    // Handle unable to acquire shared surface error.
result = g_pDXGIKeyedMutex_dev2->ReleaseSync(relKey));
if (result == WAIT_OBJECT_0)
    return S_OK;
```

The second Direct3D10.1 device can obtain the synchronized shared surface for rendering by calling AcquireSync, then releasing the surface for the first device's rendering by calling ReleaseSync. Note that device 2 is able to acquire the synchronized

shared surface using the same key value as the one specified in the ReleaseSync call by device 1.

C++

```
// Rendering onto Sync Shared Surface from D3D10.1 Device 1 using D3D10.1
Device 1.
UINT acqKey = 0;
UINT relKey = 1;
DWORD timeOut = 5;
DWORD result = g_pDXGIKeyedMutex_dev1->AcquireSync(acqKey, timeOut);
if (result == WAIT_OBJECT_0)
    // Rendering calls using Device 1.
else
    // Handle unable to acquire shared surface error.
result = g_pDXGIKeyedMutex_dev1->ReleaseSync(relKey));
if ( result == WAIT_OBJECT_0 )
    return S_OK;
```

Additional devices sharing the same surface can take turns acquiring and releasing the surface by using additional keys, as shown in the following calls.

C++

```
// Within Device 1's process/thread:
// Rendering onto Sync Shared Surface from D3D10.1 Device 1 using D3D10.1
Device 1
result = g_pDXGIKeyedMutex_dev1->AcquireSync(0, timeOut);
// Rendering calls using Device 1
...
result = g_pDXGIKeyedMutex_dev1->ReleaseSync(1);
...
////////////////////////////////////////////////////////////////////////
// Within Device 2's process/thread:
// Rendering onto Sync Shared Surface from D3D10.1 Device 1 using D3D10.1
Device 2
result = g_pDXGIKeyedMutex_dev2->AcquireSync(1, timeOut);
// Rendering calls using Device 2
...
result = g_pDXGIKeyedMutex_dev1->ReleaseSync(2);

////////////////////////////////////////////////////////////////////////
// Within Device 3's process/thread:
// Rendering onto Sync Shared Surface from D3D10.1 Device 1 using D3D10.1
Device 3
result = g_pDXGIKeyedMutex_dev1->AcquireSync(2, timeOut);
// Rendering calls using Device 3
...
result = g_pDXGIKeyedMutex_dev1->ReleaseSync(0);
...
```

Note that a real-world application might always render to an intermediate surface that is then copied into the shared surface to prevent any one device waiting on another device that shares the surface.

Using Synchronized Shared Surfaces with Direct2D and Direct3D 11

Similarly, for sharing between Direct3D 11 and Direct3D 10.1 APIs, a synchronized shared surface can be created from either API device and shared with the other API device(s), in or out of process.

Applications that use Direct2D can share a Direct3D 10.1 device and use a synchronized shared surface to interoperate with Direct3D 11 or other Direct3D 10.1 devices, whether they belong to the same process or different processes. However, for single-process, single-thread applications, device sharing is the most high-performance and efficient method of interoperability between Direct2D and either Direct3D 10 or Direct3D 11.

Software Rasterizer

Synchronized shared surfaces are not supported when applications use the Direct3D or Direct2D software rasterizers, including the reference rasterizer and WARP, instead of using graphics hardware acceleration.

Interoperability between Direct3D 9Ex and DXGI based APIs

Direct3D 9Ex APIs included the notion of surface sharing to allow other APIs to read from the shared surface. In order to share reading and writing to a Direct3D 9Ex shared surface, you must add manual synchronization to the application itself.

Direct3D 9Ex Shared Surfaces Plus Manual Synchronization Helper

The most fundamental task in Direct3D 9Ex and Direct3D 10 or 11 interoperability 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, device A's rendering is guaranteed to have completed. Therefore, device B can use this surface without worry. This is very similar to the classic producer-consumer problem and this discussion models the problem that way. The first device that uses the surface and then relinquishes it is the producer (Device A), and the device that is initially waiting is the consumer (Device B).

Any real-world application is more sophisticated than this, and will chain together multiple producer-consumer building blocks to create the desired functionality.

The producer-consumer building blocks are implemented in the helper by using a queue of surfaces. Surfaces are enqueued by the producer and dequeued by the consumer. The helper introduces three COM interfaces: ISurfaceQueue, ISurfaceProducer, and ISurfaceConsumer.

High-Level Overview of Helper

The ISurfaceQueue object is the building block for using the shared surfaces. It is created with an initialized Direct3D device and a description to create a fixed number of shared surfaces. The queue object manages creation of resources and opening of code. The number and type of surfaces are fixed; once the surfaces are created, the application cannot add or remove them.

Each instance of the ISurfaceQueue object provides a sort of one-way street that can be used to send surfaces from the producing device to the consuming device. Multiple such one-way streets can be used to enable surface sharing scenarios between devices of specific applications.

Creation/Object Lifetime
There are two ways to create the queue object: through CreateSurfaceQueue, or through the Clone method of ISurfaceQueue. Because the interfaces are COM objects, standard COM lifetime management applies.
Producer/Consumer Model
Enqueue (): The producer calls this function to indicate it is done with the surface, which can now 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 maximum known at creation time.
Dequeue() can optionally pass a buffer and a pointer to the size of the buffer. The queue fills in the buffer with the metadata from the corresponding Enqueue call.
Cloning
Each ISurfaceQueue object solves a one-way synchronization. We assume that the vast majority of applications using this API will use a closed system. The simplest closed system with two devices sending surfaces back and forth requires two 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 a new ISurfaceQueue object from an existing one, and shares all the opened resources between them. The resulting object has exactly the same surfaces as the source queue. Cloned queues can have different metadata sizes from each other.
Surfaces
The ISurfaceQueue takes responsibility for 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 a 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 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 that open the producing device and a consuming device. Only one of each can be opened at any time.

The queue exposes the following APIs:

API	Description
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	Creates an ISurfaceQueue object that shares surfaces with the root queue object.

CreateSurfaceQueue

```
C++
```

```
typedef struct SURFACE_QUEUE_DESC {
  UINT            Width;
  UINT            Height;
  DXGI_FORMAT     Format;
  UINT            NumSurfaces;
  UINT            MetaDataSize;
```

```
  DWORD           Flags;
} SURFACE_QUEUE_DESC;
```

Members

Width, **Height** The dimensions of the shared surfaces. All shared surfaces must have the same dimensions.
Format The format of the shared surfaces. All shared surfaces must have the same format. The valid formats depend on the devices that will be used, because different pairs of devices can share different format types.
NumSurfaces The number of surfaces that are part of the queue. This is a fixed number.
MetaDataSize The maximum size of the metadata buffer.
Flags Flags to control the behavior of the queue. See Remarks.

C++

```
HRESULT CreateSurfaceQueue(
  [in]   SURFACE_QUEUE_DESC *pDesc,
  [in]   IUnknown *pDevice,
  [out]  IDXGIXSurfaceQueue **ppQueue
);
```

Parameters

pDesc [in] The description of the shared surface queue to be created.

pDevice [in] The device that should be used to create the shared surfaces. This is an explicit parameter because of a feature in Windows Vista. For surfaces shared between Direct3D 9 and Direct3D 10, the surfaces must be created with Direct3D 9.

ppQueue [out] On return, contains a pointer to the ISurfaceQueue object.

Return Values

If *pDevice* is not capable of sharing resources, this function returns DXGI_ERROR_INVALID_CALL. This function creates the resources. If it fails, it returns an error. If it succeeds, it returns S_OK.

Remarks

Creating the queue object also creates 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 developer can specify a flag that indicates whether the queue will be accessed by multiple threads. If no flags are set (**Flags** == 0), the queue will be used by multiple threads. The developer can specify single threaded access, which turns off the synchronization code and provides a performance improvement for those cases. Each cloned queue has its own flag, so it is possible for different queues in the system to have different synchronization controls.

Open a Producer

```
C++

HRESULT OpenProducer(
  [in]   IUnknown *pDevice,
  [out]  IDXGIXSurfaceProducer **ppProducer
);
```

Parameters

pDevice [in]

The producer device that enqueues surfaces onto the surface queue.

ppProducer [out] Returns an object to the producer interface.

Return Values

If the device is not capable of sharing surfaces, returns DXGI_ERROR_INVALID_CALL.

Open a Consumer

```
C++

HRESULT OpenConsumer(
  [in]   IUnknown *pDevice,
  [out]  IDXGIXSurfaceConsumer **ppConsumer
);
```

Parameters

pDevice [in]
The consumer device that dequeues surfaces from the surface queue. *ppConsumer* [out] Returns an object to the consumer interface.

Return Values

If the device is not capable of sharing surfaces, returns DXGI_ERROR_INVALID_CALL.

Remarks

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

Cloning an IDXGIXSurfaceQueue

```
C++
```

```
typedef struct SHARED_SURFACE_QUEUE_CLONE_DESC {
  UINT         MetaDataSize;
  DWORD        Flags;
} SHARED_SURFACE_QUEUE_CLONE_DESC;
```

Members MetaDataSize and **Flags** have the same behavior as they do for CreateSurfaceQueue.

```
C++
```

```
HRESULT Clone(
  [in]   SHARED_SURFACE_QUEUE_CLONE_DESC *pDesc,
  [out]  IDXGIXSurfaceQueue **ppQueue
);
```

Parameters

pDesc [in] A struct that provides a description of the Clone object to be created. This parameter should be initialized.
ppQueue [out] Returns the initialized object.

Remarks

You can clone from any existing queue object, even if it is not the root.

IDXGIXSurfaceConsumer

```
C++
```

```
HRESULT Dequeue(
  [in]      REFIID    id,
  [out]     void      **ppSurface,
  [in,out]  void      *pBuffer,
  [in,out]  UINT      *pBufferSize,
  [in]      DWORD     dwTimeout
);
```

Parameters

id [in]
The REFIID of a 2D surface of the consuming device.

- For a IDirect3DDevice9, the REFIID should be __uuidof(IDirect3DTexture9).
- For a ID3D10Device, the REFIID should be __uuidof(ID3D10Texture2D).
- For a ID3D11Device, the REFIID should be __uuidof(ID3D11Texture2D).

ppSurface [out] Returns a pointer to the surface.
pBuffer [in, out] An optional parameter and if not **NULL**, on return, contains the metadata that was passed in on the corresponding enqueue call.
pBufferSize [in, out] The size of *pBuffer*, in bytes. Returns the number of bytes returned in *pBuffer*. If the enqueue call did not provide metadata, *pBuffer* is set to 0.
dwTimeout [in] Specifies a timeout value. See the Remarks for more detail.

Return Values

This function can return WAIT_TIMEOUT if a timeout value is specified and the function does not return before the time out value. See Remarks. If no surfaces are available, the function returns with *ppSurface* set to **NULL**, *pBufferSize* set to 0 and the return value is 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, such as WaitForSingleObject. For non-blocking behavior, use a timeout of 0.

ISurfaceProducer

This interface provides two methods that allow the app to enqueue surfaces. After a surface is enqueued, the surface pointer is no longer valid and is not safe to use. The only action that the application should perform with the pointer is to release it.

Method	Description
ISurfaceProducer::Enqueue	Enqueues a surface to the queue object. After this call completes, the producer is done with the surface and the surface is ready for another device.
ISurfaceProducer::Flush	Used if the applications should have non-blocking behavior. See Remarks for details.

Enqueue

```
C++

HRESULT Enqueue(
  [in]  IUnknown *pSurface,
  [in]  void *pBuffer,
  [in]  UINT BufferSize,
  [in]  DWORD Flags
);
```

Parameters

pSurface [in]
The surface of the producing device that needs to be enqueued. This surface must be a dequeued surface from the same queue network. *pBuffer* [in] An optional parameter, which is used to pass in metadata. It should point to the data that will be passed on to the dequeue call.
BufferSize [in] The size of *pBuffer*, in bytes.
Flags [in] An optional parameter that controls 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 (*Flags* == 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 function puts the surface on the queue. If the application does not specify SURFACE_QUEUE_FLAG_DO_NOT_WAIT, this function is blocking and will do a GPU-CPU synchronization to assure that all the rendering on the enqueued surface is complete. If this function succeeds, a surface will be available for dequeue. If you want non-blocking behavior, use the DO_NOT_WAIT flag. See Flush() for details.
- As per the COM reference counting rules, the surface returned by Dequeue will be AddRef() so the application does not need to do this. After calling Enqueue, the application must Release the surface because they are no longer using it.

Flush

```
C++

HRESULT Flush(
  [in]  DWORD Flags,
  [out] UINT *nSurfaces
);
```

Parameters

Flags [in]
The only flag is SURFACE_QUEUE_FLAG_ DO_NOT_WAIT. See Remarks. *nSurfaces* [out]
Returns 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 returns S_OK if any surfaces were successfully flushed. This function returns DXGI_ERROR_WAS_STILL_DRAWING only if no surfaces were flushed. Together, the return value and *nSurfaces* indicate to the application what work has been done and if any work is left to do.

Remarks

Flush is meaningful only if the previous call to enqueue used the DO_NOT_WAIT flag; otherwise, it will be a no-op. If the call to enqueue used the DO_NOT_WAIT flag, enqueue returns 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 attempts 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 non-blocking. Surfaces that have finished the GPU-CPU sync will be ready for the consumer device. Surfaces that are still pending will be unaffected. The function returns the number of surfaces that still need to be flushed.

> ⓘ **Note**
>
> Flush 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.

Direct3D 9Ex and DXGI Interop Helper: How To Use

We expect most of the usage cases to involve two devices sharing a number of surfaces. Because this also happens to be the simplest scenario, this paper details how to use the APIs to achieve this goal, discusses a non-blocking variation, and ends with a brief section about initializing for three devices.

Two Devices

The example application that uses this helper can use Direct3D 9Ex and Direct3D 11 together. The application can process content with both devices, and present content using Direct3D 9. Processing could mean rendering content, decoding video, running compute shaders, and so on. For every frame, the application will first process with Direct3D 11, then process with Direct3D 9, and finally present with Direct3D 9. Furthermore, the processing with Direct3D 11 will produce some metadata that the Direct3D 9 present will need to consume. This section covers the helper usage in three parts that correspond to this sequence: Initialization, Main Loop, and Cleanup.

Initialization
Initialization involves the following steps:

1. Initialize both devices.
2. Create the Root Queue: m_11to9Queue.
3. Clone from the Root Queue: m_9to11Queue.
4. Call OpenProducer/OpenConsumer on both queues.

The queue names use the numbers 9 and 11 to indicate which API is the producer and which is the consumer: **m_*producer*to*consumer*Queue**. Accordingly, m_11to9Queue indicates a queue for which the Direct3D 11 device produces surfaces that the Direct3D 9 device consumes. Similarly, m_9to11Queue indicates a queue for which Direct3D 9 produces surfaces that Direct3D 11 consumes.
The Root queue is initially full and all cloned queues are initially empty. This should not be a problem for the application except for the first cycle of the Enqueues and Dequeues and the availability of metadata. If a dequeue asks for metadata but none was set (either because nothing is there initially or the enqueue did not set anything), dequeue sees that no metadata was received.

1. **Initialize Both Devices.**

```
C++

m_pD3D9Device = InitializeD3D9ExDevice();
m_pD3D11Device = InitializeD3D11Device();
```

2. **Create the Root Queue.**
   This step also creates the surfaces. Size and format restrictions are identical to creating any shared resource. The size of the metadata 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. Performance will vary depending on the scenario. Having multiple surfaces increases the chances that devices are busy. For example, if there is 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 degrade performance. This example uses two surfaces.

```
C++

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);
```

3. **Clone the Root Queue.**
   Each cloned queue must use the same surfaces but can have different metadata buffer sizes and different flags. In this case, there is no metadata from Direct3D 9 to Direct3D 11.

```
C++

SURFACE_QUEUE_CLONE_DESC Desc;
Desc.MetaDataSize = 0;
Desc.Flags        = 0;

m_11to9Queue->Clone(&Desc, &m_9to11Queue);
```

4. **Open the Producer and Consumer Devices.**
   The application must perform this step before calling Enqueue and Dequeue. Opening a producer and consumer returns interfaces which contain the enqueue/dequeue APIs.

```
C++

// 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. Think of this from a per-device perspective. Each device must perform these steps: dequeue to get a surface from its consuming queue, process on the surface, and then enqueue onto its producing queue. For the Direct3D 11 device, the Direct3D 9 usage is almost identical.

C++

```
// Direct3D 9 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.
    PresentD3D9(pSurface9, metaData, metaDataSize);

    // Enqueue surface.
    m_pD3D9Producer->Enqueue(pSurface9, NULL, 0, 0);
}
```

Cleaning Up

This step is very straightforward. In addition to the normal steps for cleaning up Direct3D APIs, the application must release the return COM interfaces.

C++

```
m_pD3D9Producer->Release();
m_pD3D9Consumer->Release();
m_pD3D11Producer->Release();
m_pD3D11Consumer->Release();
m_p9to11Queue->Release();
m_p11to9Queue->Release();
```

Non-Blocking Use

The previous example makes sense for a multithreaded usage case in which each device has its own thread. The example uses the blocking versions of the APIs: INFINITE for timeout, and no flag to enqueue. If you want to use the helper in a non-blocking way, you need to make only a few changes. This section shows non-blocking use with both devices on one thread.

Initialization

Initialization is identical except for the flags. Because the application is single-threaded, use that flag for creation. This turns off some of the synchronization code, which potentially improves performance.

C++

```
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);
```

C++

```
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 the same as in the blocking example.

Using the Queue

There are many ways of using the queue in a non-blocking fashion with various performance characteristics. The following example is simple but has poor performance due to excessive spinning and polling. Despite these problems, the example shows how to use the helper. 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 application simply tries again the next loop.

C++

```
// Direct3D 11 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 Direct3D 9 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.
        PresentD3D9(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);
}
```

A more complex solution could check the return value from enqueue and from flush to determine if flushing is necessary.

Three Devices

Extending the previous examples to cover multiple devices is straightforward. The following code performs the initialization. After the Producer/Consumer objects have been created, the code to use them is the same. This example has three devices and therefore three queues. Surfaces flow from Direct3D 9 to Direct3D 10 to Direct3D 11.

C++

```
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 earlier, cloning works the same way, no matter which queue is cloned. For example, the second Clone call could have been off of the m_9to10Queue object.

C++

```
// 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);
```

Conclusion

You can create solutions that use interoperability to employ the power of multiple DirectX APIs. Windows graphics API interoperability now offers a common surface management runtime DXGI 1.1. This runtime enables synchronized surface sharing support within newly developed APIs, such as Direct3D 11, Direct3D 10.1 and Direct2D. Interoperability improvements between new APIs and existing APIs aid application migration and backward compatibility. Direct3D 9Ex and DXGI 1.1 consumer APIs can interoperate, as shown with the synchronization mechanism provided through sample helper code on MSDN Code Gallery.

Feedback

Windows Advanced Rasterization Platform (WARP) Guide

This article describes Windows Advanced Rasterization Platform (WARP) and the following aspects of WARP.

What is WARP?

WARP is a high speed, fully conformant software rasterizer. It is a component of the DirectX graphics technology that was introduced by the Direct3D 11 runtime. The Direct3D 11 runtime is installed on Windows 7, Windows Server 2008 R2, and Windows Vista with the [KB971644] update. Starting with Windows 8, Windows includes Microsoft Basic Render Adapter and Microsoft Basic Display Adapter, which enable WARP to support cross-process shared resources, as well as being present in DXGI adapter enumeration and supporting seamless fallbacks for systems with no GPU. These systems also support Direct3D 9 and older running on WARP. Starting with Windows 10, Windows includes Direct3D 12 support as well.

WARP Benefits

WARP provides the following benefits:

Removing the Need for Custom Software Rasterizers

WARP simplifies development by removing the need to build a custom software rasterizer and to tune your application for it instead of tuning your application for hardware. By providing a single, general purpose software rasterizer, you no longer need to write image rendering algorithms in multiple ways to run on hardware or software with different features and capabilities. You can still implement algorithms in multiple ways to achieve better performance or scaling; however, you do not need to change the API or rendering architecture that is used to implement those algorithms. Instead, you can focus on creating a great application with Direct3D 10 or later that will look the same and perform well on hardware or in software.

Enabling Maximum Performance from Graphics Hardware

When an application is tuned to run efficiently on hardware, it will run efficiently on WARP as well. The converse is also true; any application that is tuned to run well on WARP will perform well on hardware. Applications that use Direct3D 10 and later inefficiently might not scale efficiently on different hardware. WARP has similar performance profiles to hardware, so tuning an application for large batches, minimizing state changes, removing synchronizing points or locks will benefit both hardware and WARP.

Enabling Rendering When Direct3D Hardware is Not Available

WARP allows fast rendering in a variety of situations where hardware implementations are undesirable or unavailable, including:

- When the user does not have any Direct3D-capable hardware
- When an application runs as a service or in a server environment
- When an application wants to reserve the Direct3D hardware resources for other uses
- When a video card is not installed
- When a video driver is not available, or is not working correctly
- When a video card is out of memory, hangs, or would take too many system resources to initialize

Leveraging Existing Resources for Software Rendering

There is a huge community, many books, Web sites, SDKs, samples, white papers, mailing lists and other resources that can help you take advantage of Direct3D 10 and later shader-based image rendering. With WARP as a software fallback, you can use existing knowledge about hardware to improve the performance of your application when it runs with hardware or software. In addition, many excellent tools from the graphics card vendors and in the DirectX SDK can help you design, build, develop, debug and analyze performance issues of graphics applications. These tools and knowledge can now benefit application development that targets both hardware and software when you use WARP.

Enabling Scenarios that Do Not Require Graphics Hardware

Various algorithms and applications (image processing algorithms, printing, remoting, Virtual PCs and other emulators, high quality font rendering, charts, graphs, and so on) have typically been optimized for the CPU because they are not dependent on hardware. With WARP, you can use a single architecture that runs these algorithms and applications and that can run fully in software; yet, if hardware acceleration is available, you can take advantage of it.

Completing the DirectX Graphics Platform

WARP allows you to access all Direct3D 10 and later graphics features even on computers without Direct3D 10 and later graphics hardware. Direct3D 10 removed capability bits (caps); that is, you no longer need to verify whether graphics capabilities are available from graphics hardware because Direct3D 10 and later guarantees this availability. You can now use all the features of a wide range of video cards knowing that their application will behave and look the same everywhere. You can scale the performance of these applications by simply disabling expensive graphics features on low end video cards or rendering to smaller targets.

Providing Driver Behavior Isolation

Ideally, graphics drivers would be perfect pieces of software, but in the real world, drivers or even GPUs can have bugs or quirks. By supporting a trivial switch to run on WARP, it can let developers or users try to isolate unexpected behaviors to determine whether they're coming from their own code or from the underlying graphics driver. There are also places where API specifications allow undefined behavior, which can take different forms across different vendors or generations of hardware, and WARP can help provide a more deterministic or debuggable result in some cases.

WARP Capabilities and Requirements

WARP fully supports all Direct3D features. For example, WARP has always supported the following most important features:

- All the precision requirements of the Direct3D 10 and 10.1 specification
- Direct3D 11 when used with feature levels 9_1, 9_2, 9_3, 10_0, and 10_1 (for more information about feature levels, see **D3D_FEATURE_LEVEL**)
- All optional texture formats, such as multisample render targets and sampling from float surfaces
- Antialiased, high quality rendering up to 8x multisample antialiasing (MSAA)
- Anisotropic filtering
- 32-bit and 64-bit applications and large address aware 32-bit applications

When you install the Platform Update for Windows 7 on Windows 7 SP1 or Windows Server 2008 R2 SP1, that operating system then includes the Direct3D 11.1 runtime and a version of WARP that supports Direct3D 11.x when used with feature levels 9_1, 9_2, 9_3, 10_0, 10_1, and 11_0.

Windows 8, Windows 10, Windows Server 2012 & above, and Windows RT include the Direct3D 11.1 runtime and a new version of WARP. This version supports Direct3D 11.x when used with feature levels 9_1, 9_2, 9_3, 10_0, 10_1, 11_0, and 11_1.

Windows 10 Fall Creators Update (1709) includes a new version of WARP that supports Direct3D 12 feature levels 12_0 and 12_1.

A future Windows 11 update will include a version of WARP that supports Direct3D 12 feature level 12_2. This functionality became available in preview form in Windows Insider build 27718.

The minimum computer requirements for WARP are the same as for Windows Vista, specifically:

- Minimum 800 MHz CPU
- MMX, SSE, or SSE2 is not required
- Minimum 512 MB of RAM

How to Use WARP

For Direct3D 12, creating a WARP device requires first identifying the WARP adapter. To facilitate this, DXGI 1.4 provides the **IDXGIFactory4::EnumWarpAdapter** method. The WARP adapter can then be provided to **D3D12CreateDevice** to create a WARP device.

Direct3D 10, 10.1, and 11 components can use an additional driver type that you can specify when you create the device (for example, when you call the **D3D11CreateDevice** function). That

driver type is **D3D10_DRIVER_TYPE_WARP** or **D3D_DRIVER_TYPE_WARP**. When you specify that driver type, the runtime creates a WARP device and does not initialize a hardware device.

Direct3D 9 and older components can only run on WARP when the Microsoft Basic Display Adapter is present, meaning that no GPU is present in the system.

Because WARP uses the same software interface to Direct3D as the reference rasterizer does, any Direct3D application that can support running with the reference rasterizer can be tested by using WARP. To use WARP, rename D3d10warp.dll to D3d10ref.dll and place it in the same folder as the sample or application. Next, when you switch to ref, you will see WARP rendering.

If you rename WARP to D3d10ref.dll and place it in C:\Program Files (x86)\Microsoft DirectX SDK (June 2010)\Samples\C++\Direct3D\Bin\x86, you can run all the DirectX samples against WARP, either by clicking the "Toggle Ref" button in the sample, or by running the sample with /ref specified on the command line.

WARP binaries are available for developers to use for testing from NuGet.org. These DLLs are available for developers or end users to try out changes or improvements to WARP without having to fully update your Windows operating system. Note that these DLLs cannot be redistributed, as there is no guarantee that future versions of Windows will maintain compatibility with them for use with Microsoft Basic Render Adapter. To use them, simply place D3d10warp.dll next to your application .exe file.

Recommended Application Types for WARP

All applications that can use Direct3D can use WARP. This includes the following types of applications:

- Casual Games
- Existing Non-Gaming Applications
- Advanced Rendering Games
- Other Applications

Casual Games

Games typically have simple rendering requirements. However, they also require the use of impressive visual effects that might need hardware acceleration. The majority of the best selling game titles for Windows are either simulations or casual games, neither of which requires high performance graphics. However, both styles of games greatly benefit from modern shader-based graphics and the ability to scale on hardware.

Existing Non-Gaming Applications

A large amount of graphical applications require a minimal number of code paths in their rendering layer. WARP enables these applications to implement a single Direct3D code path that can target a large number of computer configurations.

Advanced Rendering Games

Game developers might want to isolate graphics-card or driver-specific rendering errors. Therefore, all games, even extremely graphically demanding games, can benefit from being able to render their content by using WARP. You can use WARP to validate whether any visual artifacts that you find are rendering errors or problems with hardware or drivers.

Other Applications

The target applications for WARP also include those that might not currently use Direct3D 10 or Direct3D 10.1. These target applications include applications that must always work on all computers, image processing applications that do not write CPU and GPU versions of image processing algorithms, image processing algorithms where speed or use the GPU is not critical, such as printing, and emulators and virtual environments that display advanced 3D graphics.

WARP Architecture and Performance

WARP is based on the reference rasterizer codebase. Therefore, WARP uses the same software interface to both Direct3D 10 and later and DXGI. WARP is included in Windows 7 in the D3d10warp.dll, located in Windows systems folders. Two versions of WARP are installed on 64 bit machines, an x86 and x64 version. The x64 version might run faster in certain circumstances because the code generator contained in WARP can take advantage of the additional registers that are available when users run 64-bit applications. Windows on ARM64 also supports WARP, where ARM64X binaries can natively generate ARM64 CPU instructions in both native ARM64 and emulated x86 and x64 processes.

WARP contains the following two high-speed, real-time compilers:

- The high-level intermediate language compiler that converts HLSL bytecode and the current render state into an optimized stream of vector commands for the shader stages of the pipeline, including VS, HS, DS, GS, PS, MS, AS, CS, and raytracing shader stages.
- The high-performance just-in-time code generator that can take these commands and generate optimized SSE2, SSE4.1, AVX, AVX2, AVX512, and arm64 assembly code.

WARP uses the thread pool and complex task management and dependency tracking that was introduced in Windows Vista to allow all parts of the rendering pipeline to be distributed efficiently across available CPU cores.

WARP uses deferred rendering. That is, WARP can batch rendering commands so that rasterization occurs only when sufficient data is available to use all the CPU resources efficiently. Work on the main application thread is minimized to allow the application to submit commands as quickly as possible. If an application is also multi-threaded, and it uses the thread pool, work will be evenly distributed between WARP and the application.

The WARP code generator has been tuned to make best use of the modern CPU architecture. WARP runs on all computers that can run Windows Vista and later operating systems, even if the computer does not support SSE. However, WARP has been optimized for computers that support SSE2. It also contains optimizations for specific architectures of AMD and Intel processors, as well as support for the SSE 4.1 extensions. The WARP NuGet.org releases also add support for AVX, AVX2, and AVX512 CPU extensions.

WARP does not require graphics hardware to execute. It can execute even in situations where hardware is not available or cannot be initialized.

Applications and samples that were designed and built to run on Direct3D 10 and later hardware without any knowledge of WARP will likely run well by using WARP. However, we recommend that you lower the quality settings and resolution as much as possible to achieve usable frame rates. You can use WARP to develop and tune applications that run well on both hardware and software.

Historical Performance Data

Note: The performance data below is likely no longer accurate nor relevant given the speed of advancements in computing since the Windows 7 timeframe when this data was captured.

Because WARP uses multiple CPU cores for parallel execution, it performs best on modern multi-core CPUs. WARP also runs significantly faster on computers with SSE4.1 extensions installed. Microsoft performed significant testing and performance tuning on computers with eight or more cores and SSE4.1 because these high end computers will become more common during the lifetime of Windows 7 and later operating systems.

When WARP is running on the CPU, it is limited compared to a graphics card in a number of ways. The front-side bus speed of a CPU is typically around or under 10 GB/s. In contrast, a graphics card often has dedicated memory that uses 20 to 100GB/s or more of graphics bandwidth. Graphics hardware also has fixed-function units that can perform complex and expensive tasks, such as texture filtering, format decompression, or conversions, asynchronously with little overhead or power cost. Performing these operations on a typical CPU is expensive in terms of both power consumption and performance cycles.

The typical performance numbers for an Intel Penryn based 3.0GHz Quad Core machine show that WARP can in some cases outperform low-end integrated Direct3D 10 and later graphics

GPUs on a number of benchmarks. Low-end discrete graphics hardware is typically 4 to 5 times faster than WARP at running these benchmarks. These low-end integrated or discrete GPUs have minimal use of CPU resources. Mid-range or high-end graphics cards are significantly faster than WARP for many applications, particularly when an application can take advantage of the parallelism and memory bandwidth that these graphics cards provide.

WARP is not a replacement for graphics hardware, particularly as reasonably performing low-end Direct3D 10 and later discrete hardware is now inexpensive. The goal of WARP is to allow applications to target Direct3D-compatible level hardware without having significantly different code paths or testing requirements whether they run on hardware or in software.

The following two tables show WARP example data with various CPUs and graphics cards.

The first table shows WARP example data with Direct3D 10 Crysis running at 800x600 with all the quality settings on their lowest levels:

Expand table

CPU	Time	Ave FPS	Min FPS	Min Frame	Max FPS	Max Frame
Core i7 8 Core @ 3.0GHz	271.57	7.36	3.46	1966	15.01	995
Penryn 4 Core @ 3.0GHz	351.35	5.69	2.49	1967	10.95	980
Penryn 2 Core @ 3.0GHz	573.98	3.48	1.35	1964	6.61	988
Core 2 Duo @ 2.6GHz	707.19	2.83	0.81	1959	5.18	982
Core 2 Duo @ 2.4GHz	763.25	2.62	0.76	1964	4.70	984
Core 2 Duo @ 2.1GHz	908.87	2.20	0.64	1965	3.72	986
Xeon 8 Core @ 2.0GHz	424.04	4.72	1.84	1967	9.56	988
AMD FX74 4 Core @ 3.0GHz	583.12	3.43	1.41	1967	5.78	986
Phenom 9550 4 Core @ 2.2GHz	664.69	3.01	0.53	1959	5.46	987

The second table shows example data running the same test across a variety of graphics cards:

Expand table

Graphics Card	Time	Ave FPS	Min FPS	Min Frame	Max FPS	Max Frame
NVIDIA 8800 GTS	23.58	84.80	60.78	1957	130.83	1022
NVIDIA 8500 GT	47.63	41.99	25.67	1986	72.57	991

Graphics Card	Time	Ave FPS	Min FPS	Min Frame	Max FPS	Max Frame
NVIDIA Quadro 290	67.16	29.78	18.19	1969	49.87	1017
NVIDIA 8400 GS	59.01	33.89	21.22	1962	51.82	1021
ATI 3400	53.79	37.18	22.97	618	59.77	1021
ATI 3200	67.19	29.77	18.91	1963	45.74	980
ATI 2400 PRO	67.04	29.83	17.97	606	45.91	987
Intel DX10 Integrated	386.94	5.17	1.74	1974	16.22	995

WARP Conformance

WARP passes all the standard Windows Hardware Quality Labs (WHQL) conformance tests for validating Direct3D hardware devices.

WARP has been tested against a suite of Direct3D 10 and Direct3D 10.1 applications and benchmarks, and against SDK samples from DirectX, NVIDIA, and AMD.

WARP used the PIX debugging and analysis tool for Windows in its testing; Microsoft has a large library of single frame captures of applications that are used to compare between hardware and WARP. The majority of the images appear almost identical between hardware and WARP; where small differences sometimes occur, they are found to be within the tolerances defined by the Direct3D 10 specification.

Last updated on 07/24/2025

Graphics APIs in Windows

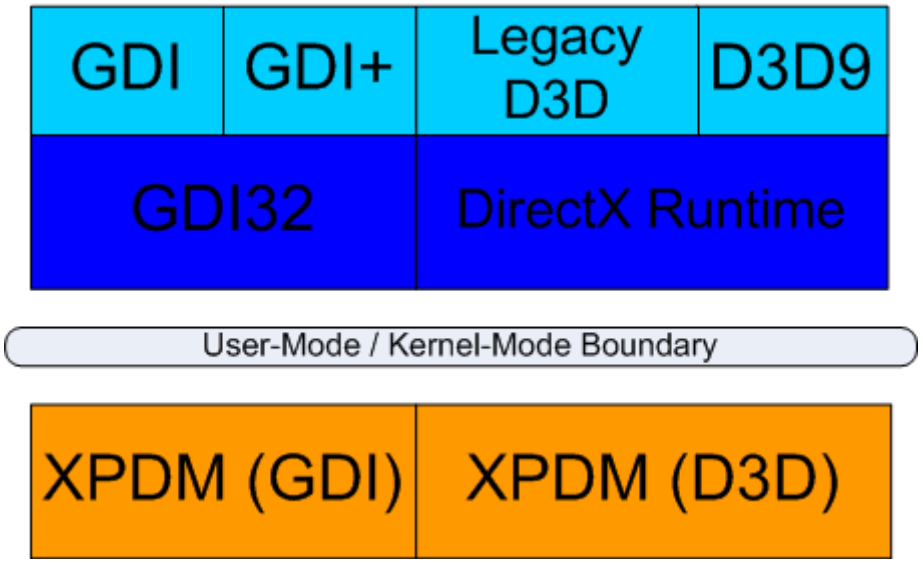
Article • 07/08/2024

Windows Vista includes support for an entirely new display driver model that represents a major revision in the design of video drivers since the introduction of the Windows Driver Model (WDM) for Windows 98. This redesigned model reflects the evolution of video hardware from the world of 2D raster operations and GDI applications to that of 3D games with fixed-function graphics hardware, and finally to that of the modern programmable graphics processing unit (GPU) that supports a wide-range of high-performance graphics applications. Windows 7 and Windows 8 build on the Windows Vista graphics infrastructure by providing additional graphics features and APIs. This articles discusses Windows graphics features and APIs.

Background

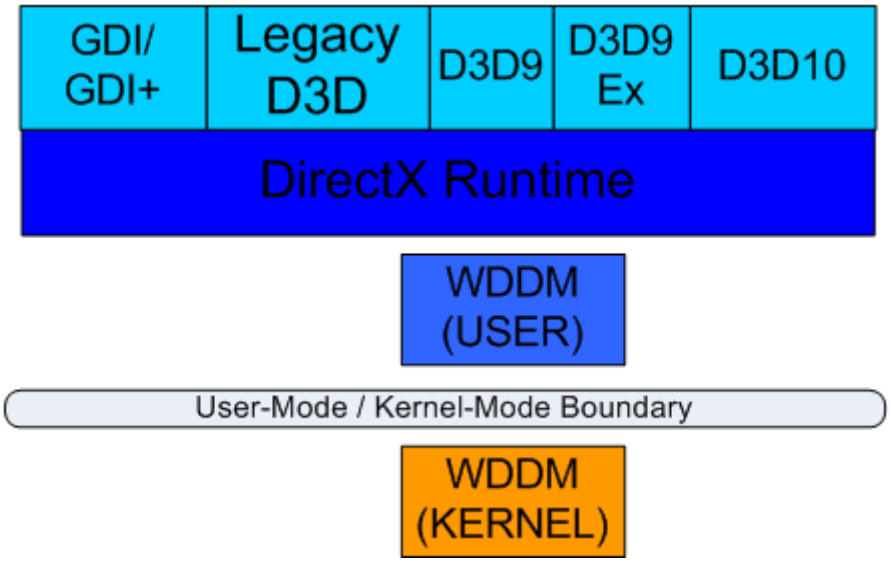
The primary API for programming graphics since the early days of Windows has been the Graphical Device Interface (GDI). This API was designed to handle numerous 2D output devices, and it formed the basis for the Windows user interface experience. DirectDraw and Direct3D were introduced as alternative APIs to support full-screen games and 3D rendering as extensions to the existing hardware of the time. Interactions with GDI were complicated. The effective intermixing of traditional GDI elements with Direct3D elements has been limited by this design. The Windows XP version of WDM, known as XPDM, reflects the side-by-side nature of GDI and Direct3D (see Figure 1).

Figure 1. Graphics APIs in Windows XP



Over the years, the power of 3D video cards has grown dramatically to the point where the vast majority of hardware is dedicated to this function. A new driver model, Windows Display Driver Model (WDDM), brings the GPU and Direct3D to the forefront, allowing the creation of an entirely new experience, the 3D desktop, that seamlessly blends the 2D world of GDI with the power of modern programmable GPUs. With WDDM, the video hardware is driven entirely by Direct3D, and all other graphics interfaces communicate with the video hardware via the new Direct3D–centric driver model (see Figure 2).

Figure 2. Graphics APIs in Windows Vista



For more information about the WDDM, see Windows Vista Display Driver Model (WDDM) Design Guide.

Direct3D 9

Version 9 of DirectX was first released for Windows in 2002, with subsequent updates in 2003 and 2004. This API represents a decade of evolution of the DirectX technologies, the introduction of more powerful shader programming models for Direct3D, and a maturity backed by thousands of shipping titles. Direct3D 9 is the primary graphics interface on Windows Vista. It remains the ideal API to use for writing 3D games and applications that need to run on the broad range of existing hardware and Windows releases. The details of the new driver model are hidden from applications using the Direct3D 9 interfaces, but behind the scenes the operating system is taking full advantage of the new capabilities to provide true multitasking of the GPU, more efficient resource management, and robust performance.

To ensure full compatibility with older versions of Windows, some quirks of the old driver model must be emulated even with the new Windows Vista display driver model. For example, when a full-screen application loses focus, it must assume it has lost all the resources in video memory (VRAM) and reload those it created as unmanaged resources even though the new driver model handles the resources transparently without evicting them from the device context. Even the concept of a managed vs. default resource type is specific to the old driver model. Another example is the expectation of failure when allocating unmanaged (default pool) resources in excess of the amount of VRAM available, even though the new driver model can provide a nearly unlimited amount of virtual video memory. Because of these requirements, Direct3D applications running on Windows Vista will still receive these error conditions. Thus, they are limited in their ability to use the basic Direct3D 9 interfaces to fully use some features of the new driver model.

While new systems shipping with Windows Vista will include video cards with WDDM drivers, and new drivers for a number of popular video cards are included in the box, Windows Vista continues to support the ability to use older XPDM drivers for upgrades and corporate editions. On systems using the old driver model, Direct3D 9 and older interfaces must be used, and the operation of the graphics system is very similar to that of Windows XP (Figure 1). WDDM is required for applications to use Direct3D 9Ex, Direct3D 10, and later versions.

Direct3D 9Ex

The Direct3D 9Ex interface provides access to a slight extension of the standard Direct3D 9 API that exposes the virtualized resource allocation, new lost device semantics, and some other new features available while running on Windows Vista. By creating this extended object, the Direct3D 9 API uses the new semantics, and therefore requires the application to use different logic (and therefore different code paths) for resource creation, management, and error handling for new kinds of conditions. This API

is only available on Windows Vista, and it requires WDDM drivers. Because Direct3D 9Ex uses a separate API and driver code path than Direct3D 9, supporting this API requires additional test cases for your application.

The primary reason for creating the new Direct3D 9Ex API was to allow full access to the new capabilities of WDDM while maintaining compatibility for existing Direct3D applications. The new 3D desktop and many Windows Vista-specific applications make use of this version of Direct3D 9, but they are not functional when running on older XPDM drivers. Because the Direct3D 9Ex API will never appear on older versions of Windows due to a lack of support for the WDDM, the standard Direct3D 9 interfaces cover a much broader set of systems. For high-performance applications that can take advantage of the next generation of video hardware, the entirely new version 10 of Direct3D provides many new capabilities not exposed by Direct3D 9Ex. As a result, for games and most other applications, Direct3D 9 or Direct3D 10 is the recommended API.

> **Note**
>
> The DirectX SDK does not provide samples, headers, or libraries for the Direct3D 9Ex interface. For more information about Direct3D 9Ex, see **DirectX for Windows Vista**.

Direct3D 10

To fully realize the potential of the new Windows Vista driver model and next-generation hardware, an entirely new version of the Direct3D API has been created. While WDDM eliminates some of the limitations on performance in the existing graphics system, Direct3D 10 goes further by removing design bottlenecks in the existing Direct3D API, and greatly simplifies the task of programming the GPU.

The new API completely eliminates all but a few fixed-function aspects, replacing them with programmable constructs and greatly streamlining the internal implementation. The hundreds of capability bits in previous versions of Direct3D have been completely eliminated and replaced with a well-defined, inclusive set of functionality that has only a few optional usage scenarios for specific resource formats. CPU-intensive resource creation and validation now have explicit semantics in the new API. This allows for much more predictable performance behavior, and greatly reduced per-draw overhead. Resources can be reconfigured into multiple forms to allow efficient use at various stages, and the feature set imposes far fewer restrictions on usage scenarios for formats. There are also new block-compressed normal-map texture formats.

In the new API, shader constants and device state are explicit resources, allowing for far more efficient caching on the hardware and greatly simplified driver validation. The programmable shader model has been unified across both vertex and pixel shaders, and made more expressive with a well-defined computational model and operator set. Also, a new geometry shader stage has been added to operate on primitives after the vertex shader stage. The results of the GPU's work in the vertex and geometry shader stages of the pipeline can be streamed out to video RAM for reuse, allowing for the possibility of extremely complex multi-pass GPU operations with minimal CPU interaction.

All of these enhancements enable next-generation graphics technology and expand the ability of applications to off-load work to the GPU. Offloading allows more complex GPU-based character skinning, accelerated morphing techniques, shadow volume generation and extrusion, particle and physics systems that are entirely GPU-based, more complex materials combined into efficient large-draw batches, procedural detailing, real-time ray-traced displacement mapping, single-pass cube-map generation, and many more techniques—all while freeing up CPU resources for more complex applications.

To provide this level of innovation in Direct3D 10, older hardware cannot be expressed as a partial implementation of a new interface. A video card is either capable of supporting all of the new features, or it's not a Direct3D 10–capable card. Therefore, while Direct3D 9 could drive DirectX7-era hardware with many missing capability bits and usage limitations, Direct3D 10 only works on a new generation of video cards. For an application to support older video hardware, it must also support the Direct3D 9 interfaces. Future versions of Direct3D will build on version 10, extending it to new versions of the API while ensuring a strict superset of Direct3D 10 functionality.

For more information about Direct3D 10, see Direct3D 10.

Direct3D 10.1

Windows Vista Service Pack 1 extends the Direct3D 10 API with Direct3D 10.1, which adds optional interfaces and an additional shader model to support new hardware features of video cards that support Direct3D 10.1. All hardware that is capable of supporting Direct3D 10.1 also fully supports all features of Direct3D 10, and game developers can make use of the additional features of Direct3D 10.1, when available.

> **Note**
>
> Direct3D 10.1 is the graphics API used by the Windows 7 desktop.

> **Note**
>
> Windows 7 and the Windows Vista update add support for DXGI 1.1, 10level9 feature levels, and the WARP10 device to the existing Direct3D 10.1 API.

Direct3D 11

Windows 7 supports a new revision of Direct3D, Direct3D 11, built on the design of Direct3D 10.1 API. New features of the API include multithreaded rendering and resource creation, Compute Shader, support for 10level9 feature levels and the WARP10 software rendering device, and new Direct3D 11 class hardware features such as tessellation using hull & domain shaders, BC6H and BC7 texture compression formats, Shader Model 5.0, and Dynamic Shader Linkage. The new API can use existing Direct3D 10 and 10.1 class video cards, some Direct3D 9 cards through the 10level9 feature levels with limited feature support, and the latest generation Direct3D 11 class video cards.

In addition to the Direct3D 11 API, Windows 7 includes DXGI 1.1, Direct2D, DirectWrite, and support for WDDM 1.1 drivers.

> **Note**
>
> The Direct3D 11 and related APIs are also available as an update to Windows Vista (see How to install the latest version of DirectX).

Direct3D 11.1

Windows 8 extends the Direct3D 11 API with Direct3D 11.1. Direct3D 11.1 supports all existing hardware that feature levels 11, 10_x, and 9_x support, as well as a new 11_1 feature level.

In addition to the Direct3D 11.1 API, Windows 8 includes DXGI 1.2, Direct2D device contexts, and support for WDDM 1.2 drivers.

> **Note**

If you want your Windows Store apps to program 3D graphics with DirectX, you can use the Direct3D 11.1 API. For more info about programming 3D graphics with DirectX, see **Introduction to 3D graphics with DirectX**.

Platform Update for Windows 7: Partial support is available for the Direct3D 11.1 API on Windows 7 or Windows Server 2008 R2 with the Platform Update for Windows 7 installed. For more info about the Platform Update for Windows 7, see Platform Update for Windows 7.

OpenGL

Windows Vista, Windows 7, and Windows 8 provide the same support as Windows XP for OpenGL, which allows video card manufactures to provide an installable client driver (ICD) for OpenGL that provides hardware-accelerated support. Note that newer versions of such ICDs are required to fully support Windows Vista, or Windows 7, or Windows 8. If no ICD is installed, the system will fall back to the OpenGL v1.1 software layer in most cases.

Application Compatibility, GDI, and older versions of Direct3D

The Windows Vista, Windows 7, and Windows 8 graphics systems are designed to support a broad range of hardware and usage scenarios to enable new technology while continuing to support existing systems. Existing graphics interfaces, such as GDI, GDI+, and older versions of Direct3D, continue to work on Windows Vista and Windows 7, but are internally remapped where possible. This means that the majority of existing Windows applications will continue to work.

Windows Vista, Windows 7, and Windows 8 continue to support the same Direct3D and DirectDraw interfaces as Windows XP, back to version 3 of DirectX (with the exception of Direct3D's Retained Mode, which has been removed). Just as with Windows XP Professional x64 Edition, 64-bit native applications on newer versions of Windows are limited to Direct3D9, DirectDraw7, or newer interfaces. High-performance applications should make use of Direct3D 9 or later to ensure that they have the closest match to the hardware capabilities.

Recommendations

Consider the following recommendations when selecting an API for your graphical application:

- Use Direct3D 9 if your application must support Windows XP or an earlier version of Windows.
- Use Direct3D 9 if you want to support Windows Vista or Windows 7 running with XPDM drivers. For Windows Vista or Windows 7 systems that lack Direct3D 10 or better video hardware, you can either choose to use the existing Windows XP Direct3D 9 code path or use the 10level9 feature levels through the Direct3D 10.1 or Direct3D 11 API.
- Use Direct3D 11 to take advantage of the next generation of video hardware on Windows Vista, Windows 7, and Windows 8. Windows Store apps must use Direct3D 11 or later.

Feedback

Direct3D 11 deployment for game developers

Article • 01/26/2022

This article describes how to deploy the Direct3D 11 components on a system if necessary.

Overview

The Direct3D 11 API extends the existing Direct3D 10.1 API with support for multithreaded rendering and resource creation, Compute Shader, hardware tessellation, BC6H/BC7 texture compression, and HLSL Shader Model 5.0 with Dynamic Shader Linkage. In addition to the Direct3D 11 component, a number of additional graphics components are included in the DirectX 11 runtime: Direct3D 11, DXGI 1.1, 10level9 feature levels, WARP10 software rendering device, Direct2D, DirectWrite, and an updated Direct3D 10.1 with support for 10level9 and WARP10. For information on these and other Windows graphics components, see Graphics APIs in Windows.

All of these new graphics components are built into the Windows 7 and Windows Server 2008 R2 operating systems. The Direct3D 11 API and related components can also be installed on Windows Vista by using a system update from Windows Update. This update requires Windows Vista and Service Pack 2. End-users with automatic updates enabled will, therefore, likely already have the Direct3D 11 components installed, as will all Windows 7 users.

The D3D11InstallHelper sample is designed to simplify detection of the Direct3D 11 API, automatically install the system update if applicable to an end-user's computer, and to provide appropriate messages to the end-user on manual procedure if a newer Service Pack is required.

> **Note**

The HLSL compiler (D3DCompile*.dll) and the D3DX utility library for Direct3D 11 (D3DX11*.dll) are not built into any version of the Windows operating system, but they can be deployed as part of an application's installer by using the existing DirectSetup technology; for more information about using DirectSetup, see **DirectX Installation for Game Developers**. "Effects 11" is available as a shared source support library at **Effects for Direct3D 11 Update**, and you can include it directly into an app (much like the DXUT utility library). Thus, it doesn't have any additional run-time redistribution requirements.

Direct3D 11.3

Windows 10 ships with the Direct3D 11.3 API built in. See Direct3D 11.3 Features for a list of new features in the Direct3D 11.3 API.

Direct3D 11.2

Windows 8.1 and Windows Server 2012 R2 ship with the Direct3D 11.2 API built in. See Direct3D 11.2 Features for a list of new features in the Direct3D 11.2 API.

Direct3D 11.1

Windows 8 and Windows Server 2012 ship with the Direct3D 11.1 API built in. Partial support for the Direct3D 11.1 API is available on Windows 7 or Windows Server 2008 R2 with the Platform Update for Windows 7 installed. For more info about the Platform Update for Windows 7, see Platform Update for Windows 7.

D3D11InstallHelper.dll

D3D11InstallHelper.dll hosts the core functionality for detecting Direct3D 11 components, and performing the system update through the Windows Update service if applicable. The DLL displays no messages or dialog boxes directly.

The DLL consists of the following entry points:

CheckDirect3D11Status

This function performs the necessary checks and returns the status of Direct3D 11 on this computer. This function does not require administrator rights.

- A status of D3D11IH_STATUS_INSTALLED indicates that Direct3D 11 is already installed on the computer and is ready for use.
- D3D11IH_STATUS_NOT_SUPPORTED indicates that this version of Windows does not support Direct3D 11 or related technologies.

- A status of D3D11IH_STATUS_NEED_LATEST_SP indicates that the latest Windows Vista Service Pack should be installed by the user.
- Finally, a status of D3D11IH_STATUS_REQUIRES_UPDATE indicates that the system does not have Direct3D 11 installed, but that the system update does apply to this version of Windows.

DoUpdateForDirect3D11

This function uses the Windows Update API to perform the system update for installing Direct3D 11 on this system, if applicable. Note that this function requires network connectivity to Windows Update to succeed, as well as administrative rights. It takes an optional progress callback function and user-context pointer, and returns a final result code when complete.

- The D3D11IH_RESULT_SUCCESS result indicates that the system update was applied and is ready for use, while D3D11IH_RESULT_SUCCESS_REBOOT indicates that the system update requires the computer is restarted before it is complete. Note that this function does not schedule a system restart.
- D3D11IH_RESULT_NOT_SUPPORTED indicates that the system update does not apply to this version of Windows. This result should not occur if this function is only called after getting a D3D11IH_STATUS_REQUIRES_UPDATE status from CheckDirect3D11Status.
- A result of D3D11IH_RESULT_UPDATE_NOT_FOUND indicates that the system update package was not found on the Windows Update servers.
- If the Windows Update download or installation fails, then D3D11IH_RESULT_UPDATE_DOWNLOAD_FAILED or D3D11IH_RESULT_UPDATE_INSTALL_FAILED will be returned as the result.
- If a network connectivity error is returned from the Windows Update API, then the D3D11IH_RESULT_WU_SERVICE_ERROR result is returned, indicating that the problem might be intermittent or related to network configuration or firewall settings. Trying the update function again may succeed.

For more information on the Windows Update API, see Windows Update Agent API.

D3D11Install.exe

> **Note**
>
> D3D11Install.exe requires D3D11InstallHelper.dll to execute.

D3D11Install.exe is a tool for using D3D11InstallHelper.dll as a stand-alone installer complete with UI and end-user messages, as well as acting as an example for proper use of the DLL. The process exits with a 0 if Direct3D 11 is already installed, if the system update applies successfully without requiring a system restart, if a Service Pack installation is required, or if Direct3D 11 is not supported by this computer. A 1 is returned if the system update is applied successfully and requires a system restart to complete. A 2 is returned for other error

conditions. Note that this executable file requires administrator rights to run, and it has a manifest that requests elevation when run on Windows Vista or Windows 7 with UAC enabled. D3D11Install.exe can be used as a stand-alone tool for deploying the Direct3D 11 update, or it can be used directly by installers.

It supports the following command-line switches:

/quiet

Displays no messages, prompts, progress dialog boxes, or error messages.

/passive

Displays no messages, prompts, or error messages, but will show the progress dialog box.

/minimal

Shows only minimal prompts.

/y

Suppresses prompting to confirm installing the update, if needed and applicable, for a standard and minimal installation.

/langid decimal

Forces which language identifier code to use when displaying end-user messages and dialog box resources. The default value is 1024, which uses the system default language setting.

/wu

Forces use of Windows Update rather than the system default, which may be Windows Server Update Services (WSUS) running on a managed server or some other non-standard configuration.

Integrating into installation programs

To comply with Support Easy Installation, Technical Requirement 3.1 for Games for Windows, care needs to be taken so that any end-user prompts are presented early in the installation process, and to ensure that there are not multiple UAC-related elevation prompts. There are three basic choices for achieving this goal:

1. The most basic method is to execute the D3D11Install.exe with the command-line switch **/minimal**. This should be done early in the installer Q&A, and the installation should use the return value of 1 to indicate that a restart should be scheduled at the end of the installation. Executing the program requires administrative rights.
2. Use D3D11InstallHelper.dll directly to detect the need for the update, providing any end-user messages necessary for the status D3D11IH_STATUS_NEED_LATEST_SP, where the

resolution requires manual user operations. The status result of D3D11IH_STATUS_NOT_SUPPORTED could be used to control installation of Direct3D 11–related assets, or as an error condition for Direct3D 11–only applications, but it is otherwise not necessarily a useful end-user message. For the status D3D11IH_STATUS_REQUIRES_UPDATE, the installer can directly use the DLL entry point DoUpdateForDirect3D11 to perform the update and handle the various resulting end-user messages. Examples of standard messages can be found by examining the D3D11Install.exe dialog box and string table resources. The update entry point requires administrator rights.

3. A hybrid approach is to check status with D3D11InstallHelper.dll, and in the case of the status code D3D11IH_STATUS_NEED_LATEST_SP or D3D11IH_STATUS_REQUIRES_UPDATE, D3D11Install.exe can be executed with the switches **/minimal** and **/y** to display the dialog box or to perform the update, as needed. These steps should be performed early in the installation process, usually immediately after the Q&A, and running the executable requires administrative rights.

Integrating into InstallShield

Handling the Direct3D 11 deployment from InstallShield's InstallScript is easily done using the D3D11InstallHelper sample. The steps required to integrate with InstallShield using InstallScript are as follows (using method 3, described in the previous section):

1. Open an InstallScript project in the InstallShield editor.

2. Add D3D11InstallHelper.dll and D3D11Install.exe to the project in **Support Files**.

   To add the files to the InstallShield Project
   a. On the **Installation Designer** tab, click **Support Files/Billboards** under **Behavior and Logic** in the navigation pane on the left.
   b. Click **Language Independent**, then right-click in the **Files** window and select **Insert Files**. Browse to add D3D11InstallHelper.dll and D3D11Install.exe. The default location for these files is: SDK root\Samples\C++\Misc\Bin\x86

3. In the InstallScript explorer, click the InstallScript file (usually Setup.rul) that will call the DLL or executable, located under **Behavior and Logic** in the navigation pane on the left.

4. Paste the following InstallScript into the file near the top:

```
syntax
```

#define D3D11IH_STATUS_INSTALLED 0 #define D3D11IH_STATUS_NOT_SUPPORTED 1 #define D3D11IH_STATUS_REQUIRES_UPDATE 2 #define D3D11IH_STATUS_NEED_LATEST_SP 3 #define D3D11IH_STATUS_ERROR -1 prototype NUMBER D3D11InstallHelper.CheckDirect3D11StatusIS();

#define D3D11IH_RESULT_SUCCESS 0 #define D3D11IH_RESULT_SUCCESS_REBOOT 1 #define D3D11IH_RESULT_NOT_SUPPORTED 2 #define D3D11IH_RESULT_UPDATE_NOT_FOUND 3 #define D3D11IH_RESULT_UPDATE_DOWNLOAD_FAILED 4 #define D3D11IH_RESULT_UPDATE_INSTALL_FAILED 5 #define D3D11IH_RESULT_WU_SERVICE_ERROR 6 #define D3D11IH_RESULT_ERROR -1 prototype NUMBER D3D11InstallHelper.DoUpdateForDirect3D11IS(BOOL); ```

5. Paste the following InstallScript into the file in the **OnFirstUIBefore** function, just before the return 0:

```
syntax

Dlg_D3D11:
    UseDLL( SUPPORTDIR ^ "D3D11InstallHelper.DLL" );
    nResult = D3D11InstallHelper.CheckDirect3D11StatusIS();
    UnUseDLL( SUPPORTDIR ^ "D3D11InstallHelper.DLL" );

    if ( nResult = D3D11IH_STATUS_REQUIRES_UPDATE
         || nResult = D3D11IH_STATUS_NEED_LATEST_SP) then
        nResult = LaunchAppAndWait(
SUPPORTDIR^"D3D11Install.exe",
"/minimal /y", WAIT);
        if ( nResult < 0 ) then
            MessageBox("Unable to launch D3D11Install.exe",
 SEVERE);
        elseif ( nResult == 1 ) then
            BATCH_INSTALL = 1;
        endif;
    endif;
```

Integrating into an MSI package

The following is a high-level description of the steps required to integrate Direct3D 11 deployment using MSI custom actions (using method 3, described earlier in this topic):

1. Add a property to the MSI Property table called **RelativePathToD3D11IH** that contains the relative path to D3D11Install.exe and D3D11InstallHelper.dll during installation (this is typically in the media image). This also sets an MSI property D3D11IH_STATUS to the status returned by CheckDirect3D11Status (a string property equal to the enumeration symbol or "ERROR").
2. After the CostFinalize action, call the D3D11InstallHelper.dll function **SetD3D11InstallMSIProperties** as an immediate custom action to set the appropriate MSI properties for the other custom actions.
3. Upon installation, trigger a deferred custom action after the InstallFiles action that calls the D3D11InstallHelper.dll function **DoD3D11InstallUsingMSI**. The custom action must set the flag msidbCustomActionTypeNoImpersonate to run in an elevated context.
4. After the InstallFinalize action, call the D3D11InstallHelper.dll function **FinishD3D11InstallUsingMSI** as an immediate custom action to handle the successful

reboot request result code, if needed.

This procedure is described in detail in the following instructions, which describe a process that can be done using an MSI editor, such as the Orca editor. Some MSI editors have wizards that simplify some of these configuration steps.

To configure an MSI package for integration with D3D11InstallHelper.dll

1. Open the MSI package in Orca.
2. Add the row shown in the following table to the Binary table in the MSI package.

Name	Data
D3D11IH	File path to the DLL\D3D11InstallHelper.dll

> **Note**
>
> This file will be embedded in the MSI package, so you must do this step every time that you recompile D3D11InstallHelper.dll.

3. Add the rows shown in the following table to the CustomAction table in the MSI package.

Action	Type	Source	Target
Direct3D11SetProps	msidbCustomActionTypeDll + msidbCustomActionTypeBinaryData + msidbCustomActionTypeContinue = 65	D3D11IH	SetD3D11InstallMSIProperties
Direct3D11DoInstall	msidbCustomActionTypeDll + msidbCustomActionTypeBinaryData + msidbCustomActionTypeContinue + msidbCustomActionTypeInScript + msidbCustomActionTypeNoImpersonate = 3137	D3D11IH	DoD3D11InstallUsingMSI
Direct3D11Finish	msidbCustomActionTypeDll + msidbCustomActionTypeBinaryData + msidbCustomActionTypeContinue = 65	D3D11IH	FinishD3D11InstallUsingMSI

4. Add the values shown for Action, Condition, and Sequence in the following table to the InstallExecuteSequence table in the MSI package.

Action	Condition	Sequence	Notes
Direct3D11SetProps		1016	The sequence number places the action soon after CostFinalize.
Direct3D11DoInstall	NOT Installed	4004	This custom action will only happen during a new installation for all users. The sequence number places the action after InstallFiles and after the rollbacks.
Direct3D11Finish		6615	The sequence number places the action soon after InstallFinalize.

5. Add the row shown in the following table to the **Property** table in the MSI package.

Property	Value
RelativePathToD3D11IH	relative file path that contains D3D11Install.exe and D3D11InstallHelper.dll

> **Note**
>
> The location specified by the path is relative to the location specified by the installation path—for example, "redist\".

6. Save the MSI package. For more detailed information about MSI packages and Windows Installer, see Windows Installer.

Debugging tips

Both D3D11InstallHelper.dll and D3D11Install.exe can be built with the Debug configuration in Visual Studio, and these versions will print messages to the standard Windows debug output mechanism.

Corporate settings

The D3D11InstallHelper sample is designed for standard deployment through Windows Update, which is the most common scenario for installation of a game by consumers. However, Many game developers, working for publishers and in development studios, do so in enterprise settings that have a locally managed server providing software updates by using Windows

Server Update Services (WSUS) technology. In this type of environment, the local IT administrator has approval control over which updates are made available to computers within the corporate network, and the standard consumer version of update KB 971644 is not available.

There are three basic solutions for deploying DirectX 11 in corporate/enterprise settings:

- In some configurations, it is possible to directly check Windows Update rather than use the locally managed WSUS server. For this reason, D3D11InstallHelper supports the **/wu** command-line switch. However, not all corporate networks allow connections to the public Microsoft servers.
- The local IT administrator can approve KB 971512, an enterprise-supported update deployed from WSUS, that includes the Direct3D 11 API. This is the only option for a Standard User to obtain the Direct3D 11 update in an environment that is fully locked down.
- Alternatively, KB 971512 can be manually installed.

It is very rare that a gamer's computer can only get updates from a locally managed WSUS server, and it is only developers in large organizations who are likely to be in such environments.

Related articles

Windows Firewall for Game Developers

Windows Games Explorer for Game Developers

Feedback

DirectX Graphics Infrastructure (DXGI): Best Practices

Article • 01/26/2022

Microsoft DirectX Graphics Infrastructure (DXGI) is a new subsystem that was introduced with Windows Vista that encapsulates some of the low-level tasks that are needed by Direct3D 10, 10.1, 11, and 11.1. From the perspective of a Direct3D 9 programmer, DXGI encompasses most of the code for enumeration, swap-chain creation, and presentation that previously was packed into the Direct3D 9 APIs. When you port an app to DXGI and Direct3D 10.x and Direct3D 11.x, you need to take some considerations into account to ensure that the process runs smoothly.

This article discusses key porting issues.

Full-Screen Issues

In porting from Direct3D 9 to DXGI and to Direct3D 10.x or Direct3D 11.x, issues associated with moving from windowing to full-screen mode often may cause headaches for developers. The main problems arise because Direct3D 9 applications, unlike DXGI applications, require a more hands-on approach to tracking window styles and window states. When the mode-changing code is ported to run on DXGI, it often causes unexpected behavior.

Often, Direct3D 9 applications handled the transition into full-screen mode by setting the resolution of the front buffer, forcing the device into full-screen exclusive mode, and then setting the back buffer resolutions to match. A separate path was used for changes to window size because they had to be managed from the window process whenever the application received a WM_SIZE message.

DXGI attempts to simplify this approach by combining the two cases. For example, when the window border is dragged in windowed mode, the application receives a WM_SIZE message. DXGI intercepts this message and automatically resizes the front buffer. All

that the application needs to do is call IDXGISwapChain::ResizeBuffers to resize the back buffer to the size that was passed as parameters in WM_SIZE. Similarly, when the application needs to switch between full-screen and windowed mode, the application can simply call IDXGISwapChain::SetFullscreenState. DXGI resizes the front buffer to match the newly selected full-screen mode, and it sends a WM_SIZE message to the application. The application again calls **ResizeBuffers**, just as it would if the window border was dragged.

The methodology of the preceding explanation follows a very particular path. DXGI set the full-screen resolution to the desktop resolution by default. Many applications, however, switch to a preferred full-screen resolution. In such a case, DXGI provides IDXGISwapChain::ResizeTarget. This should be called before calling SetFullscreenState. Although these methods can be called in the opposite order (**SetFullscreenState** first, followed by **ResizeTarget**), doing so causes an extra WM_SIZE message to be sent to the application. (Doing so can also cause flickering, since DXGI could be forced to perform two mode changes.) After calling **SetFullscreenState**, it is advisable to call **ResizeTarget** again with the **RefreshRate** member of DXGI_MODE_DESC zeroed out. This amounts to a no-operation instruction in DXGI, but it can avoid issues with the refresh rate, which are discussed next.

When in full-screen mode, the Desktop Window Manager (DWM) is disabled. DXGI can perform a flip to present the back buffer contents instead of doing a blit, which it would do in windowed mode. This performance gain can be undone, however, if certain requirements are not met. To ensure that DXGI does a flip instead of a blit, the front buffer and back buffer must be sized identically. If the application correctly handles its WM_SIZE messages, this should not be a problem. Also, the formats must be identical.

The problem for most applications is the refresh rate. The refresh rate that is specified in the call to ResizeTarget must be a refresh rate that is enumerated by the IDXGIOutput object that the swap chain is using. DXGI can automatically calculate this value if the application zeroes out the **RefreshRate** member of DXGI_MODE_DESC that is passed into **ResizeTarget**. It is important not to assume that certain refresh rates will always be supported and to simply hard-code a value. Often, developers choose 60 Hz as the refresh rate, not knowing that the enumerated refresh rate from the monitor is approximately 60,000 / 1,001 Hz from the monitor. If the refresh rate does not match the expected refresh rate of 60, DXGI is forced to perform a blit in full-screen mode instead of a flip.

The last issue that developers often face is how to change full-screen resolutions while remaining in full-screen mode. Calling ResizeTarget and SetFullscreenState sometimes succeeds, but the full-screen resolution remains the desktop resolution. Also, developers may create a full-screen swap chain and give a specific resolution, only to find that DXGI

defaults to the desktop resolution regardless of the numbers passed in. Unless otherwise instructed, DXGI defaults to the desktop resolution for full-screen swap chains. When creating a full-screen swap chain, the **Flags** member of the DXGI_SWAP_CHAIN_DESC structure must be set to DXGI_SWAP_CHAIN_FLAG_ALLOW_MODE_SWITCH to override DXGI's default behavior. This flag also can be passed to **ResizeTarget** to enable or disable this functionality dynamically.

Multiple Monitors

When using DXGI with multiple monitors, there are two rules to follow.

The first rule applies to the creation of two or more full-screen swap chains on multiple monitors. When creating such swap chains, it is best to create all swap chains as windowed, and then to set them to full-screen. If swap chains are created in full-screen mode, the creation of a second swap chain causes a mode change to be sent to the first swap chain, which could cause termination of full-screen mode.

The second rule applies to outputs. Be watchful of outputs used when creating swap chains. With DXGI, the IDXGIOutput object controls which monitor the swap chain uses when becoming full-screen. Unlike DXGI, Direct3D 9 had no concept of outputs.

Window Styles and DXGI

Direct3D 9 applications had a lot of work to do when switching between full-screen and windowed modes. Much of this work involved changing window styles to add and remove borders, to add scrollbars, and so on. When applications are ported to DXGI and Direct3D 10.x or Direct3D 11.x, this code often is left in place. Depending on the changes being made, switching between modes can cause unexpected behavior. For example, when switching to windowed mode, the application might no longer have a window frame or window border despite having code that specifically sets these styles. This occurs because DXGI now handles much of this style changing on its own. Manual setting of window styles can interfere with DXGI, and this can cause unexpected behavior.

The recommended behavior is to do as little work as possible, and to let DXGI handle most of the interaction with the windows. However, if the application needs to handle its own windowing behavior, IDXGIFactory::MakeWindowAssociation can be used to tell DXGI to disable some of its automatic window handling.

Multithreading and DXGI

Special care must be taken when using DXGI in a multithreaded application to ensure that deadlocks do not occur. Because of DXGI's close interaction with windowing, it occasionally sends window messages to the associated application window. DXGI needs the windowing changes to occur before it can continue, so it will use **SendMessage**, which is a synchronous call. The application must process the window message before **SendMessage** returns.

In an application where DXGI calls and the message pump are on the same thread (or a single-threaded application), little needs to be done. When the DXGI call is on the same thread as the message pump, **SendMessage** calls the window's *WindowProc*. This bypasses the message pump, and allows execution to continue after the call to **SendMessage**. Remember that **IDXGISwapChain** calls, such as **IDXGISwapChain::Present**, are also considered DXGI calls; DXGI may defer work from **ResizeBuffers** or **ResizeTarget** until **Present** is called.

If the DXGI call and message pump are on different threads, care must be taken to avoid deadlocks. When the message pump and SendMessage are on different threads, **SendMessage** adds a message to the window's message queue, and waits for the window to process that message. If the window procedure is busy or is not called by the message pump, the message may never get processed and DXGI will wait indefinitely.

For example, if an application that has its message pump on one thread and its rendering on another, it may want to change modes. The message pump thread tells the rendering thread to change modes, and waits until the mode change is complete. However, the rendering thread calls DXGI functions, which in turn call **SendMessage**, which blocks until the message pump processes the message. A deadlock occurs because both threads now are blocked, and are waiting on each other. To avoid this, never block the message pump. If a block is unavoidable, then all DXGI interaction should occur on the same thread as the message pump.

Gamma and DXGI

Although gamma may be best handled in Direct3D 10.x or Direct3D 11.x by using SRGB textures, the gamma ramp still can be useful to developers who want a different gamma value than 2.2 or who are using a render target format that does not support SRGB. Be aware of two issues when setting the gamma ramp through DXGI. The first issue is that the ramp values passed into **IDXGIOutput::SetGammaControl** are float values, not **WORD** values. Also, ensure that code ported from Direct3D 9 does not try to convert to **WORD** values before passing these to **SetGammaControl**.

The second issue is that, after changing to full-screen mode, **SetGammaControl** may not appear to work, dependent on the **IDXGIOutput** object being used. When changing to full-screen mode, DXGI creates a new output object, and uses the object for all subsequent operations on the output. If calling **SetGammaControl** on an output that is enumerated before a full-screen mode switch, the call is not directed toward the output that DXGI is using currently. To avoid this, call **IDXGISwapChain::GetContainingOutput** to get the current output, and then call **SetGammaControl** off this output to get the correct behavior.

For info about using gamma correction, see Using gamma correction.

DXGI 1.1

The Direct3D 11 runtime included in Windows 7 and installed onto Windows Vista includes version 1.1 of DXGI. This update adds definitions for a number of new formats (particularly BGRA, 10-bit X2 bias, and Direct3D 11's BC6H and BC7 texture compression), as well as a new version of the DXGI factory and adapter interfaces (**CreateDXGIFactory1**, **IDXGIFactory1**, **IDXGIAdapter1**) for enumerating remote desktop connections.

When you use Direct3D 11, the runtime will use DXGI 1.1 by default when calling **D3D11CreateDevice** or **D3D11CreateDeviceAndSwapChain** with a NULL **IDXGIAdapter** pointer. Mixing use of DXGI 1.0 and DXGI 1.1 in the same process is not supported. Mixing DXGI object instances from different factories in the same process also is not supported. Therefore, when you use DirectX 11, any explicit use of the DXGI interfaces uses a **IDXGIFactory1** created by the **CreateDXGIFactory1** entry-point in “DXGI.DLL” to ensure the application is always using DXGI 1.1.

DXGI 1.2

The Direct3D 11.1 runtime that is included in Windows 8 also includes version 1.2 of DXGI.

DXGI 1.2 enables these features:

- stereo rendering
- 16 bit-per-pixel formats
  - DXGI_FORMAT_B5G6R5_UNORM and DXGI_FORMAT_B5G5R5A1_UNORM are now fully supported
  - a new DXGI_FORMAT_B5G5R5A1_UNORM format was added

- video formats
- new DXGI interfaces

For more info about DXGI 1.2 features, see DXGI 1.2 Improvements.

Feedback

Use DirectX with Advanced Color on high/standard dynamic range displays

Article • 10/18/2022

This topic shows how to use DirectX with Advanced Color scenarios—including high dynamic range (HDR), wide color gamut (WCG) with automatic system color management, and high bit-depth. Premium personal computer (PC) displays with at least one of the above enhancements are becoming widespread, providing significantly higher color fidelity than traditional standard dynamic range (SDR) displays.

In this topic, you'll get an overview of the key technical concepts behind Windows Advanced Color support. You'll learn requirements and instructions for rendering HDR, WCG, and high bit-depth DirectX content to one of these displays. If you have a color-managed app (for example, using ICC profiles), then you'll learn how auto color management enables better color accuracy for your scenarios.

Introduction to Advanced Color in Windows

Advanced Color is an umbrella term of operating system (OS) technologies for displays with significantly higher color fidelity than standard displays. The predominant extended capabilities are described in the sections below. Advanced Color capabilities were first introduced for HDR displays with Windows 10, version 1709 (Fall Creators Update), and for specially provisioned SDR displays with the Windows 11, version 22H2 (10.0; Build 22621) release.

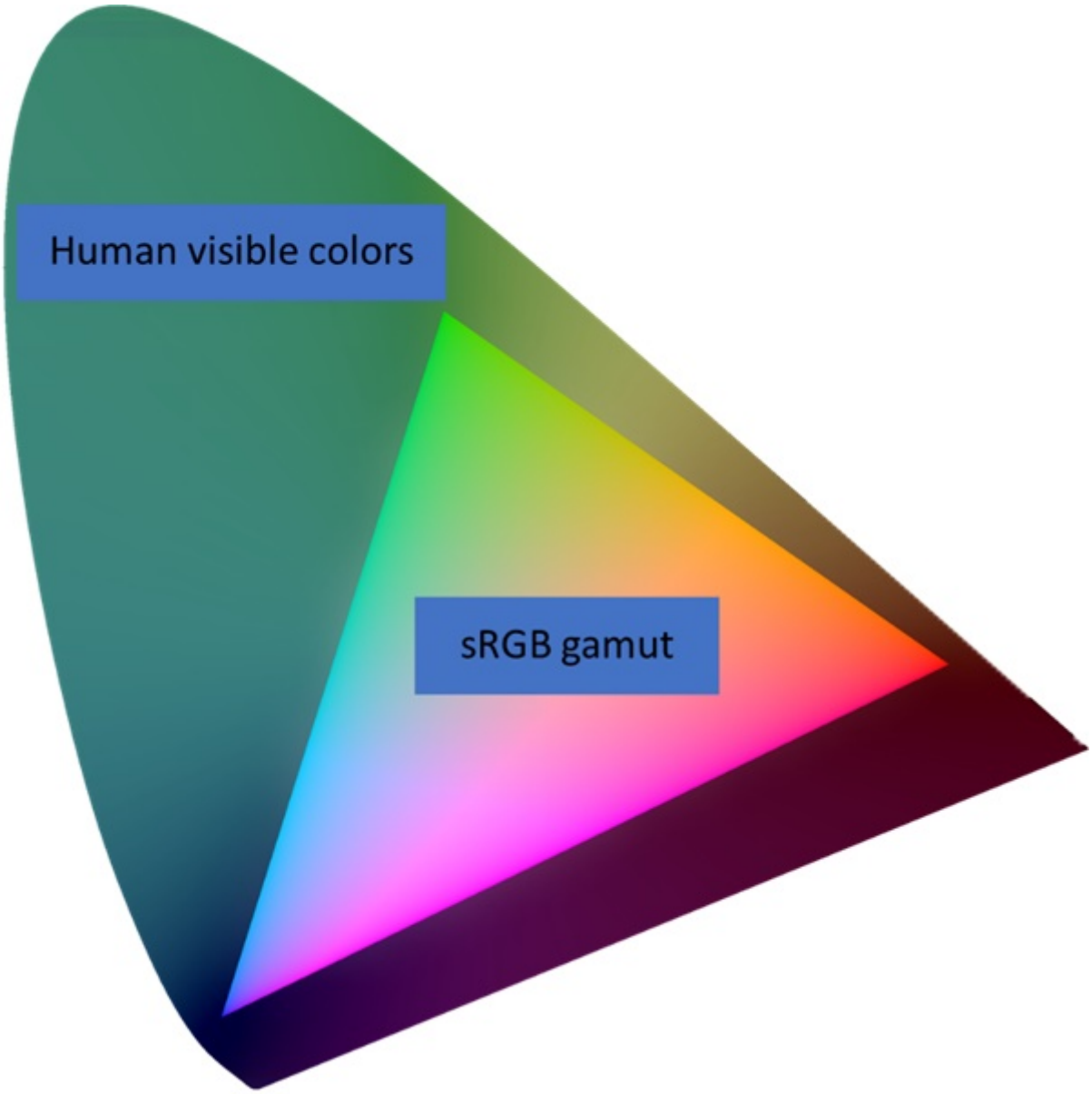
High dynamic range

Dynamic range refers to the difference between the maximum and minimum luminance in a scene; this is often measured in nits (candelas per square centimeter). Real world scenes, such as this sunset, often have dynamic ranges of 10 orders of magnitude of luminance; the human eye can discern an even greater range after adaptation.

Ever since Direct3D 9, graphics engines have been able to internally render their scenes with this level of physically accurate fidelity. However, a typical standard dynamic range display can reproduce only a little more than 3 orders of magnitude of luminance, and therefore any HDR-rendered content had to be tonemapped (compressed) into the limited range of the display. New HDR displays, including those that comply with the HDR10 (BT.2100) standard, break through this limitation; for example, high quality self-emissive displays can achieve greater than 6 orders of magnitude.

Wide color gamut

Color gamut refers to the range and saturation of hues that a display can reproduce. The most saturated natural colors the human eye can perceive consist of pure, monochromatic light such as that produced by lasers. However, mainstream consumer displays can often reproduce colors only within the sRGB gamut, which represents only about 35% of all human-perceivable colors. The diagram below is a representation of the human "spectral locus", or all perceivable colors (at a given luminance level), where the smaller triangle is the sRGB gamut.



High end, professional PC displays have long supported color gamuts that are significantly wider than sRGB, such as Adobe RGB and DCI-P3 which cover around half of human-perceivable colors. And these wide gamut displays are becoming more common.

Automatic system color management

Color management is the technology and practice of ensuring accurate and consistent color reproduction across devices. If you're a digital content creator, its crucial for the colors in your visual content—such as a photo, a product image, or a logo—to appear the same on your display as it does on your audience's wide variety of digital devices.

Windows has provided color management support APIs since Windows 2000 with the Image Color Management (ICM) and later Windows Color System (WCS) APIs. However, those APIs were only helpers for apps that wished/required to do color management; while most apps and digital content simply assumed the industry standard sRGB color

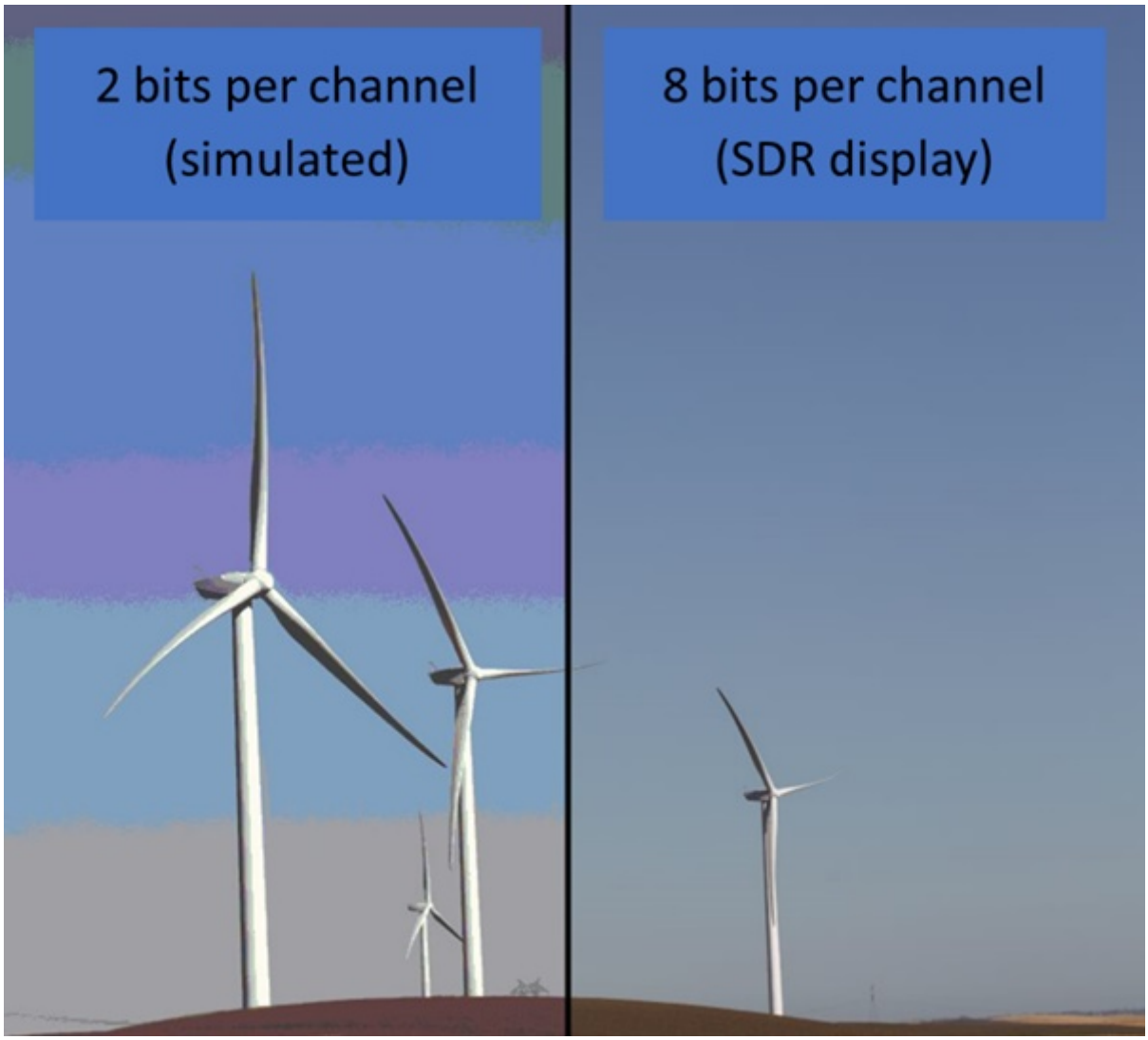
space, and were never color-managed by the OS. That was a reasonable assumption in the past, but high-quality wide gamut displays are becoming much more common.

New versions of Windows support automatic system color management; that ensures that all colors in every Windows app, whether or not they are color-aware, appear accurately and consistently on every supported display.

> **Note**
>
> Auto color management is not a property of the display hardware; instead, it's a Windows feature to properly support displays that have larger color gamuts than sRGB.

Deep precision/bit depth

Numerical precision, or bit depth, refers to the amount of information used to uniquely identify colors. Higher bit depth means that you can distinguish between very similar colors without artifacts such as banding. Mainstream PC displays support 8 bits per color channel, while the human eye requires at least 10-12 bits of precision to avoid perceivable distortions.

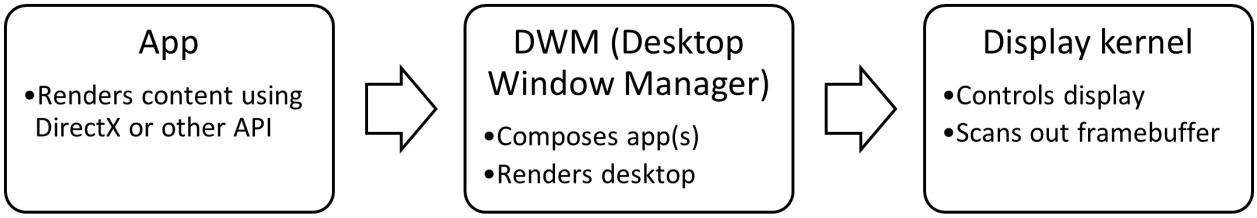


Prior to Advanced Color, the Desktop Window Manager (DWM) restricted windowed apps to output content at only 8 bits per color channel, even if the display supported a higher bit depth. When Advanced Color is enabled, the DWM performs its composition using IEEE half-precision floating point (FP16), eliminating any bottlenecks, and allowing the full precision of the display to be used.

Windows Advanced Color system architecture

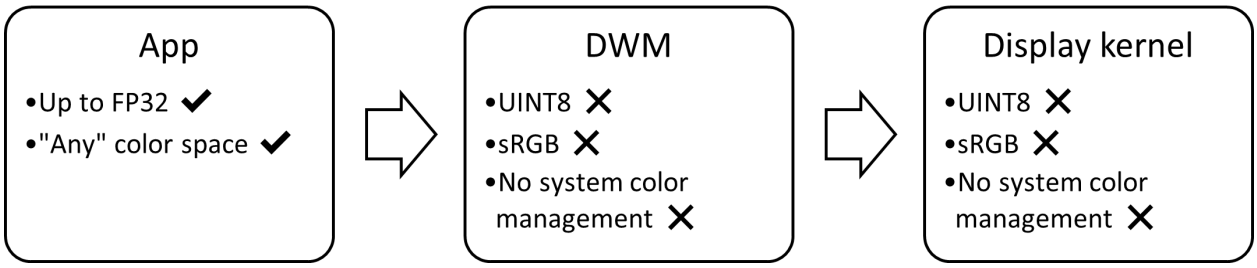
The info in this section is optional for building Advanced Color apps; but it's helpful to understand how the technology works in order to optimize your app's rendering and behavior.

In this section, we'll use a simplified diagram to describe the relevant components of the Windows graphics stack:



Existing Windows: 8-bit / sRGB displays

For decades, consumer displays and the Windows graphics stack were based around 8 bits per channel (24 bits per pixel) sRGB content. Apps using graphics APIs such as DirectX could perform internal rendering using high bit-depths and extended color spaces; however, the OS supported only 8-bit integer with implicit sRGB and no system color management:

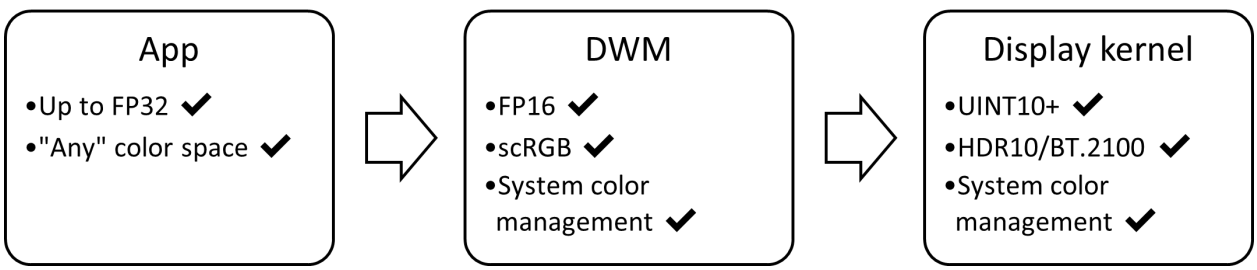


That meant that any additional color data rendered by an app would be lost when being displayed; and that the app had to perform color management itself to ensure accurate reproduction on a display.

Windows 10, version 1703: HDR displays with Advanced Color

Windows 10, version 1703 introduced the first version of Advanced Color capabilities for HDR displays. That required several significant advances in the OS graphics stack:

- HDR display signaling support
- System composition using a high bit-depth, canonical color space
- Automatic system color management



Each advancement is covered in the sub-sections below. The net result is that extended app color data is now correctly preserved by the OS, and accurately reproduced on HDR displays.

HDR display signaling support

HDR signaling over display connectors such as DisplayPort and HDMI primarily uses 10 bits per channel precision (or greater) and the BT.2100 ST.2084 color space. The display kernel, display driver, and underlying GPU hardware all need to support detecting, selecting, and driving this signaling mode.

System composition using a high bit-depth, canonical color space

The BT.2100 ST.2084 color space is an efficient standard for encoding HDR colors, but it's not well suited for many rendering and composition (blending) operations. We also want to future proof the OS to support technologies and color spaces well beyond BT.2100, which covers less than 2/3 of human-visible colors. Finally, where possible we want to minimize GPU resource consumption in order to improve power and performance.

When in HDR mode, the Desktop Window Manager (DWM) uses a canonical composition color space (CCCS) defined as:

- scRGB color space (BT.709/sRGB primaries with linear gamma)
- IEEE half precision (FP16 bit depth)

That provides a good balance between all of the above goals. CCCS allows color values outside of the [0, 1] numeric range; given the range of valid FP16 values, it can represent orders of magnitude more colors than the natural human visual range, including luminance values over 5 million nits. FP16 has excellent precision for linear gamma blend operations, but costs half the GPU memory consumption and bandwidth of traditional single precision (FP32) with no perceivable quality loss.

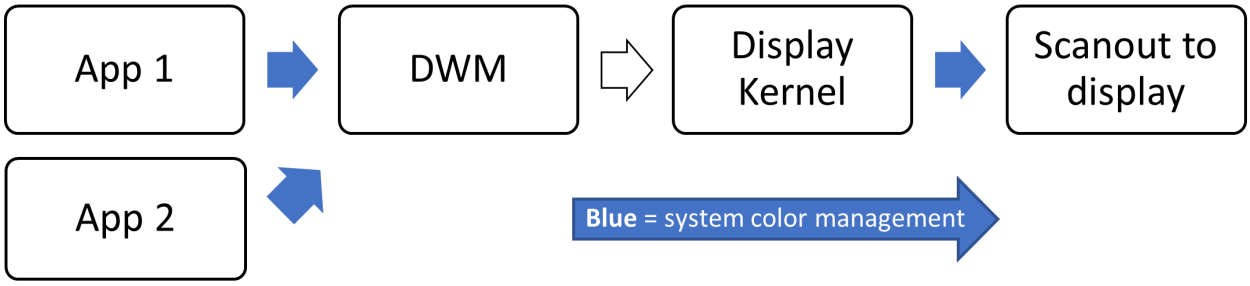
Automatic system color management

Windows is a multitasking environment where the user can run any number of SDR and HDR apps at the same time with overlapping windows. Therefore, it's crucial that all types of content look correct and at maximum quality when output to a display; for example, an sRGB (SDR) productivity app with a BT.2100 ST.2084 (HDR) video window playing over it.

When in HDR mode, Windows performs color management operations in two stages:

1. The DWM converts each app from its native color space to CCCS before blending.
2. The display kernel converts the OS framebuffer from CCCS to the wire format color space (BT.2100 ST.2084).

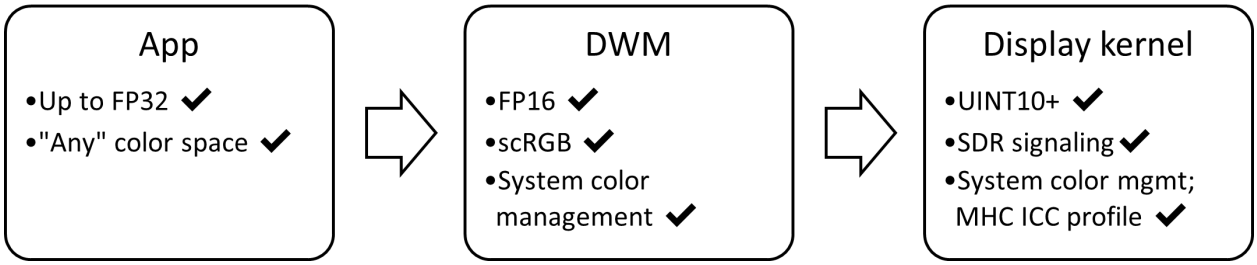
    - Optionally, the display kernel works with the display driver to perform additional display color calibration; for more info, see Windows hardware display color calibration pipeline.



> **Note**
>
> In both stages, the color management operation consists of a color space conversion (matrix and 1DLUT). Colors that exceed the display's target color gamut are numerically clipped.

Windows 11, version 22H2: SDR displays with Advanced Color

While the prevalence of HDR displays is growing rapidly, SDR displays will remain important for years to come. HDR support in Windows 10, version 1703 laid most of the groundwork needed to enhance SDR displays too. Windows 11, version 22H2 extends Advanced Color and auto color management capabilities to certain qualifying SDR displays. The graphics block diagram for Advanced Color SDR displays looks very similar to HDR:



SDR display signaling support with high bit-depth

The underlying signaling for SDR displays is unchanged, although the Windows 11m version 22H2 release supports 10 bits per channel and greater, depending on the

display's capabilities.

System composition using a high bit-depth, canonical color space

DWM Advanced Color functionality including blending in CCCS is almost entirely unchanged from HDR displays. The main difference is that DWM uses display-referred luminance with SDR displays, and scene-referred luminance with HDR displays. This changes the way that your Advanced Color rendered content is interpreted by the OS:

Display kind	Luminance behavior	How is 1.0f interpreted
SDR	Display-referred	As the reference white level of the display
HDR	Scene-referred	As 80 nits (nominal reference white)

Automatic system color management

OS system color management capabilities are also mostly unchanged from HDR displays. The main difference is that the display kernel converts to the display-referred color space as defined by the display's colorimetry and calibration data, instead of the standard BT.2100 ST.2084 color space for HDR displays.

Display provisioning required

Accurate data from an MHC ICC profile is needed to define the display kernel's output color management operation. Therefore, only SDR displays that have been specifically provisioned by the manufacturer or a display calibration provider with a valid profile are eligible for auto color management. See ICC profile behavior with Advanced Color for more info.

System requirements and operating system support

Windows 10, version 1709 first shipped Advanced Color support for HDR displays. The Windows 11, version 22H2 release adds Advanced Color support for SDR displays that have accurate provisioning data.

This topic assumes that your app is targeting Windows 10, version 2004 (or later) for HDR displays, and the Windows 11, version 22H2 release (or later) for SDR displays.

Display

A high dynamic range display must implement the HDR10, or BT.2100 ST.2084, standard. HDR display quality can vary greatly, and we strongly recommend displays that are certified, such as VESA DisplayHDR. Starting with the Windows 11, version 22H2 release, Windows displays the certification status of known displays in the **Settings** app.

A standard dynamic range display must have accurate color provisioning data for Advanced Color support. In the Windows 11, version 22H2 release, the only supported method to override this data is via an MHC ICC profile; in addition, the user or display manufacturer must have enabled auto color management. For more info, see ICC profile behavior with Advanced Color.

Graphics processor (GPU)

For full Advanced Color functionality on both SDR and HDR displays, a recent GPU is required:

- AMD Radeon RX 400 series (Polaris), or newer
- NVIDIA GeForce 10 series (Pascal), or newer
- Selected Intel Core 10th gen (Ice Lake), or newer*

> **Note**
>
> Intel code name Comet Lake (5 digit model code) chipsets don't provide full functionality.

Additional hardware requirements might apply, depending on the scenarios, including hardware codec acceleration (10 bit HEVC, 10 bit VP9, etc.) and PlayReady support (SL3000). Contact your GPU vendor for more specific information.

Graphics driver (WDDM)

The latest available graphics driver is strongly recommended, either from Windows Update or from the GPU vendor or PC manufacturer's website. This topic relies on driver functionality from WDDM 2.7 (Windows 10, version 2004) for HDR displays, and WDDM 3.0 (Windows 11, version 21H2) for SDR displays.

Supported rendering APIs

Windows 10 supports a wide variety of rendering APIs and frameworks. Advanced color support fundamentally relies on your app being able to perform modern presentation using either DXGI or the Visual Layer APIs.

- DirectX Graphics Infrastructure (DXGI)
- Visual Layer (Windows.UI.Composition)

Therefore, any rendering API that can output to one of those presentations methods can support Advanced Color. That includes (but is not limited to) the below.

- Direct3D 11
- Direct3D 12
- Direct2D
- Win2D
  - Requires using the lower level **CanvasSwapChain** or **CanvasSwapChainPanel** APIs.
- Windows.UI.Input.Inking
  - Supports custom dry ink rendering using DirectX.
- XAML
  - Supports playback of HDR videos using **MediaPlayerElement**.
  - Supports decode of JPEG XR images using **Image** element.
  - Supports DirectX interop using **SwapChainPanel**.

Handling dynamic display capabilities

Windows 10 supports an enormous range of Advanced Color-capable displays, from power-efficient integrated panels to high-end gaming monitors and TVs. Windows users expect that your app will seamlessly handle all of those variations, including ubiquitous existing SDR displays.

Windows 10 provides control over HDR and Advanced Color capabilities to the user. Your app must detect the current display's configuration, and respond dynamically to any changes in capability. That could occur for many reasons, for example, because the user enabled or disabled a feature, or moved the app between different displays, or the system's power state changed.

Option 1: AdvancedColorInfo

> **Note**

The **AdvancedColorInfo** Windows Runtime API is usable independently of the rendering API, supports Advanced Color for SDR displays, and uses events to signal when capabilities change. However, it's available only for Universal Windows Platform (UWP) apps; desktop apps (which don't have a **CoreWindow**) can't use it. For more info, see **Windows Runtime APIs not supported in desktop apps**.

First, obtain an instance of **AdvancedColorInfo** from **DisplayInformation::GetAdvancedColorInfo**.

To check what Advanced Color kind is currently active, use the **AdvancedColorInfo::CurrentAdvancedColorKind** property. That's the most important property to check, and you should configure your render and presentation pipeline in response to the active kind:

Advanced color kind	Display capabilities
SDR	SDR display with no Advanced Color capabilities
WCG	SDR display with high bit-depth and auto color management
HDR	HDR display with all Advanced Color capabilities

To check what Advanced Color kinds are supported, but not necessarily active, call **AdvancedColorInfo::IsAdvancedColorKindAvailable**. You could use that information, for example, to prompt the user to navigate to the Windows **Settings** app so that they can enable HDR or auto color management.

The other members of **AdvancedColorInfo** provide quantitative information about the panel's physical color volume (luminance and chrominance), corresponding to SMPTE ST.2086 static HDR metadata. Even though ST.2086 was originally designed for HDR displays, that info is useful, and is available for both HDR and SDR displays. You should use that information to configure your app's tone mapping and gamut mapping.

To handle changes in Advanced Color capabilities, register for the **DisplayInformation::AdvancedColorInfoChanged** event. That event is raised if any parameter of the display's Advanced Color capabilities changes for any reason.

Handle that event by obtaining a new instance of **AdvancedColorInfo**, and checking which values have changed.

IDXGIOutput6

> **Note**

The DirectX Graphics Infrastructure **IDXGIOutput6** interface is available for any app that uses DirectX, whether it's desktop or Universal Windows Platform (UWP). However, **IDXGIOutput6** *doesn't* support SDR displays with Advanced Color capabilities such as auto color management; it can identify only HDR displays.

If you're writing a Win32 desktop app, and using DirectX to render, then use **DXGI_OUTPUT_DESC1** to get display capabilities. Obtain an instance of that struct via **IDXGIOutput6::GetDesc1**.

To check what Advanced Color kind is currently active, use the **ColorSpace** property, which is of type **DXGI_COLOR_SPACE_TYPE**, and contains one of the following values:

DXGI_COLOR_SPACE_TYPE	Display capabilities
DXGI_COLOR_SPACE_RGB_FULL_G22_NONE_P709	SDR display with no Advanced Color capabilities
DXGI_COLOR_SPACE_RGB_FULL_G2084_NONE_P2020	HDR display with all Advanced Color capabilities

> ⓘ **Note**
>
> SDR displays with Advanced Color capabilities are also reported as **DXGI_COLOR_SPACE_RGB_FULL_G22_NONE_P709**; DXGI doesn't allow you to distinguish between the two types.

> ⓘ **Note**
>
> DXGI doesn't let you check what Advanced Color kinds are supported-but-not-active at the moment.

Most of the other members of **DXGI_OUTPUT_DESC1** provide quantitative information about the panel's physical color volume (luminance and chrominance), corresponding to SMPTE ST.2086 static HDR metadata. Even though ST.2086 was originally designed for HDR displays, that info is useful and is available for both HDR and SDR displays. You should use that information to configure your app's tone mapping and gamut mapping.

Win32 desktop apps don't have a native mechanism to respond to Advanced Color capability changes. Instead, if your app uses a render loop, then you should query **IDXGIFactory1::IsCurrent** with each frame. If it reports **FALSE**, then you should obtain a new **DXGI_OUTPUT_DESC1**, and check which values have changed.

In addition, your Win32 message pump should handle the **WM_SIZE** message, which indicates that your app might have moved between different displays.

> **Note**
>
> To obtain a new **DXGI_OUTPUT_DESC1**, you must obtain the current display. However, you shouldn't call **IDXGISwapChain::GetContainingOutput**. That's because swap chains return a stale DXGI output once **DXGIFactory::IsCurrent** is false; and recreating the swap chain to get a current output results in a temporarily black screen. Instead, we recommend that you enumerate through the bounds of all DXGI outputs, and determine which one has the greatest intersection with your app window's bounds.

The following example code comes from the Direct3D 12 HDR sample app on GitHub.

C++

```
// Retrieve the current default adapter.
ComPtr<IDXGIAdapter1> dxgiAdapter;
ThrowIfFailed(m_dxgiFactory->EnumAdapters1(0, &dxgiAdapter));

// Iterate through the DXGI outputs associated with the DXGI adapter,
// and find the output whose bounds have the greatest overlap with the
// app window (i.e. the output for which the intersection area is the
// greatest).

UINT i = 0;
ComPtr<IDXGIOutput> currentOutput;
ComPtr<IDXGIOutput> bestOutput;
float bestIntersectArea = -1;

while (dxgiAdapter->EnumOutputs(i, &currentOutput) != DXGI_ERROR_NOT_FOUND)
{
    // Get the retangle bounds of the app window
    int ax1 = m_windowBounds.left;
    int ay1 = m_windowBounds.top;
    int ax2 = m_windowBounds.right;
    int ay2 = m_windowBounds.bottom;

    // Get the rectangle bounds of current output
    DXGI_OUTPUT_DESC desc;
    ThrowIfFailed(currentOutput->GetDesc(&desc));
    RECT r = desc.DesktopCoordinates;
    int bx1 = r.left;
    int by1 = r.top;
    int bx2 = r.right;
    int by2 = r.bottom;

    // Compute the intersection
```

```
    int intersectArea = ComputeIntersectionArea(ax1, ay1, ax2, ay2, bx1,
by1, bx2, by2);
    if (intersectArea > bestIntersectArea)
    {
        bestOutput = currentOutput;
        bestIntersectArea = static_cast<float>(intersectArea);
    }

    i++;
}

// Having determined the output (display) upon which the app is primarily
being
// rendered, retrieve the HDR capabilities of that display by checking the
color space.
ComPtr<IDXGIOutput6> output6;
ThrowIfFailed(bestOutput.As(&output6));

DXGI_OUTPUT_DESC1 desc1;
ThrowIfFailed(output6->GetDesc1(&desc1));
```

Setting up Your DirectX swap chain

Once you've determined that the display currently supports Advanced Color capabilities, configure your swap chain as follows.

Use a flip presentation model effect

When creating your swap chain using one of the **CreateSwapChainFor[Hwnd|Composition|CoreWindow]** methods, you must use the DXGI flip model by selecting either the **DXGI_SWAP_EFFECT_FLIP_SEQUENTIAL** or **DXGI_SWAP_EFFECT_FLIP_DISCARD** option, which makes your swap chain eligible for Advanced Color processing from DWM and various fullscreen optimizations. For more information, see For best performance, use DXGI flip model.

Option 1. Use FP16 pixel format and scRGB color space

Windows 10 supports two main combinations of pixel format and color space for Advanced Color. Select one based on your app's specific requirements.

We recommend that general-purpose apps use Option 1. It's the only option that works for all types of Advanced Color displays, content, and rendering APIs. When creating your swap chain, specify **DXGI_FORMAT_R16G16B16A16_FLOAT** in your **DXGI_SWAP_CHAIN_DESC1**. By default, a swap chain created with a floating point pixel

format is treated as if it uses the DXGI_COLOR_SPACE_RGB_FULL_G10_NONE_P709 color space. That's the same pixel format and color space used by the DWM.

That combination provides you with the numeric range and precision to specify any physically possible color, and perform arbitrary processing including blending.

However, that option consumes 64 bits per pixel, which doubles GPU bandwidth and memory consumption compared to the traditional UINT8 pixel formats. In addition, scRGB uses numeric values that are outside the normalized [0, 1] range to represent colors that are outside the sRGB gamut and/or greater than 80 nits of luminance. For example, scRGB (1.0, 1.0, 1.0) encodes the standard D65 white at 80 nits; but scRGB (12.5, 12.5, 12.5) encodes the same D65 white at a much brighter 1000 nits. Some graphics operations require a normalized numeric range, and you must either modify the operation, or re-normalize color values.

The way that luminance values are interpreted with that option differs between SDR and HDR displays; see below.

Option 2: Use UINT10/RGB10 pixel format and HDR10/BT.2100 color space

Option 2 is a performance optimization that's available only if your app meets all of the following conditions:

- Targets an HDR display
- Uses Direct3D 12 or Direct3D 11
- Swap chain doesn't require blending with alpha/transparency

If your app doesn't meet all of those conditions, then you must use Option 1.

But if your app qualifies for option 2, then that might provide better performance if your app is consuming HDR10-encoded content, such as a video player, or if it mainly will be used in fullscreen scenarios, such as a game. When creating your swap chain you should consider specifying DXGI_FORMAT_R10G10B10A2_UNORM in DXGI_SWAP_CHAIN_DESC1. By default, that's treated as using the sRGB color space; therefore, you must explicitly call IDXGISwapChain3::SetColorSpace1, and set as your color space DXGI_COLOR_SPACE_RGB_FULL_G2084_NONE_P2020, also known as HDR10/BT.2100.

This option consumes the same 32 bits per pixel as traditional UINT8 SDR pixel formats. In addition, on certain GPUs this eliminates some processing needed to convert the content to the HDR10 wire format.

Using an Advanced Color swap chain when the display is in SDR mode

You can use an Advanced Color swap chain even if the display doesn't support all Advanced Color capabilities. In those cases, the Desktop Window Manager (DWM) will downconvert your content to fit the display's capabilities by performing numeric clipping. For example, if you render to an FP16 scRGB swap chain, and target a standard display, then everything outside of the [0, 1] numeric range is clipped.

That downconversion behavior will also occur if your app window is straddling two or more displays with differing Advanced Color capabilities. **AdvancedColorInfo** and **IDXGIOutput6** are abstracted to report only the *main* display's characteristics (*main* being defined as the display containing the center of the window).

Match your app's reference white to the OS SDR reference white level

> **Note**
>
> Reference white applies only to HDR displays; for SDR Advanced Color displays, (1.0, 1.0, 1.0) always means the maximum white luminance that the display can reproduce.

In many scenarios, your app will want to render both SDR and HDR content; for example, rendering subtitles or transport controls over HDR video, or UI into a game scene. It's important to understand the concept of *SDR reference white level* to ensure that your SDR content looks correct on an HDR display. Reference white indicates the brightness at which a diffuse white object (such as a sheet of paper, or sometimes UI) appears in an HDR scene. Because HDR color values have *scene-referred brightness*, a particular color value should be displayed at an absolute luminance level, and not relative to the maximum possible panel value. For example, scRGB (1.0, 1.0, 1.0) and HDR10 (497, 497, 497) both encode exactly D65 white at 80 nits luminance. Windows allows the user to adjust the *SDR reference white level* to their preference; that's the luminance that Windows will render sRGB (1.0, 1.0, 1.0) at. On desktop HDR monitors, SDR reference white levels are typically set to around 200 nits.

Your HDR app must allow the user to either set their desired reference white level, or to read the value configured by the system. You must map your diffuse white color values in your scene to the SDR reference white level. That entails multiplying your app framebuffer in linear gamma space.

> **Note**
>
> On a display that supports a brightness control, such as on a laptop, Windows also adjusts the luminance of HDR (scene-referred) content to match the user's desired brightness level, but that's invisible to the app. Unless you're trying to guarantee bit-accurate reproduction of the HDR signal, you can generally ignore that.

If your app always renders SDR and HDR to separate surfaces, and relies on OS composition, then Windows will automatically perform the correct adjustment to boost SDR content to the desired white level. For example, if your app uses XAML, and renders HDR content to its own SwapChainPanel.

However, if your app performs its own composition of SDR and HDR content into a single surface, then you're responsible for performing the SDR reference white level adjustment yourself. Otherwise the SDR content might appear too dim under typical desktop viewing conditions. First, you must obtain the current SDR reference white level, and then you must adjust the color values of any SDR content you're rendering.

Step 1. Obtain the current SDR reference white level

Currently, only UWP apps can obtain the current SDR reference white level via AdvancedColorInfo.SdrWhiteLevelInNits. That API requires a CoreWindow.

Step 2. Adjust color values of SDR content

Windows defines the nominal, or default, reference white level at 80 nits. Therefore if you were to render a standard sRGB (1.0, 1.0, 1.0) white to an FP16 swap chain, then it would be reproduced at 80 nits luminance. In order to match the actual user-defined reference white level, you must adjust SDR content from 80 nits to the level specified via **AdvancedColorInfo.SdrWhiteLevelInNits**.

If you're rendering using FP16 and scRGB, or any color space that uses linear (1.0) gamma, then you can simply multiply the SDR color value by `AdvancedColorInfo.SdrWhiteLevelInNits / 80`. If you're using Direct2D, then there's a predefined constant D2D1_SCENE_REFERRED_SDR_WHITE_LEVEL, which has a value of 80.

C++

```
D2D1_VECTOR_4F inputColor; // Input SDR color value.
D2D1_VECTOR_4F outputColor; // Output color adjusted for SDR white level.
auto acInfo = ...; // Obtain an AdvancedColorInfo.
```

```
float sdrAdjust = acInfo->SdrWhiteLevelInNits /
D2D1_SCENE_REFERRED_SDR_WHITE_LEVEL;

// Normally in DirectX, color values are manipulated in shaders on GPU
textures.
// This example performs scaling on a CPU color value.
outputColor.r = inputColor.r * sdrAdjust; // Assumes linear gamma color
values.
outputColor.g = inputColor.g * sdrAdjust;
outputColor.b = inputColor.b * sdrAdjust;
outputColor.a = inputColor.a;
```

If you're rendering using a nonlinear gamma color space such as HDR10, then performing SDR white level adjustment is more complex. If you're writing your own pixel shader, then consider converting into linear gamma to apply the adjustment.

Adapt HDR content to the display's capabilities using tone mapping

HDR and Advanced Color displays vary greatly in terms of their capabilities. For example, in the minimum and maximum luminance and the color gamut that they're capable of reproducing. In many cases, your HDR content will contain colors that exceed the display's capabilities. For the best image quality it's important for you to perform HDR tone mapping, essentially compressing the range of colors to fit the display while best preserving the visual intent of the content.

The most important single parameter to adapt for is max luminance, also known as MaxCLL (content light level); more sophisticated tone mappers will also adapt min luminance (MinCLL) and/or color primaries.

Step 1. Get the display's color volume capabilities

Universal Windows Platform (UWP) apps

Use **AdvancedColorInfo** to get the display's color volume.

Win32 (desktop) DirectX apps

Use **DXGI_OUTPUT_DESC1** to get the display's color volume.

Step 2. Get the content's color volume information

Depending on where your HDR content came from, there are multiple potential ways to determine its luminance and color gamut information. Certain HDR video and image files contain SMPTE ST.2086 metadata. If your content was rendered dynamically, then you might be able to extract scene information from the internal rendering stages—for example, the brightest light source in a scene.

A more general but computationally expensive solution is to run a histogram or other analysis pass on the rendered frame. The Direct2D advanced color image rendering sample app on GitHub demonstrates how to do that using Direct2D; the most relevant code snippets are included below:

C++

```
// Perform histogram pipeline setup; this should occur as part of image
resource creation.
// Histogram results in no visual output but is used to calculate HDR
metadata for the image.
void D2DAdvancedColorImagesRenderer::CreateHistogramResources()
{
    auto context = m_deviceResources->GetD2DDeviceContext();

    // We need to preprocess the image data before running the histogram.
    // 1. Spatial downscale to reduce the amount of processing needed.
    DX::ThrowIfFailed(
        context->CreateEffect(CLSID_D2D1Scale, &m_histogramPrescale)
        );

    DX::ThrowIfFailed(
        m_histogramPrescale->SetValue(D2D1_SCALE_PROP_SCALE,
D2D1::Vector2F(0.5f, 0.5f))
        );

    // The right place to compute HDR metadata is after color management to
the
    // image's native colorspace but before any tonemapping or adjustments
for the display.
    m_histogramPrescale->SetInputEffect(0, m_colorManagementEffect.Get());

    // 2. Convert scRGB data into luminance (nits).
    // 3. Normalize color values. Histogram operates on [0-1] numeric range,
    //    while FP16 can go up to 65504 (5+ million nits).
    // Both steps are performed in the same color matrix.
    ComPtr<ID2D1Effect> histogramMatrix;
    DX::ThrowIfFailed(
        context->CreateEffect(CLSID_D2D1ColorMatrix, &histogramMatrix)
        );

    histogramMatrix->SetInputEffect(0, m_histogramPrescale.Get());

    float scale = sc_histMaxNits / sc_nominalRefWhite;
```

```
    D2D1_MATRIX_5X4_F rgbtoYnorm = D2D1::Matrix5x4F(
        0.2126f / scale, 0, 0, 0,
        0.7152f / scale, 0, 0, 0,
        0.0722f / scale, 0, 0, 0,
        0              , 0, 0, 1,
        0              , 0, 0, 0);
    // 1st column: [R] output, contains normalized Y (CIEXYZ).
    // 2nd column: [G] output, unused.
    // 3rd column: [B] output, unused.
    // 4th column: [A] output, alpha passthrough.
    // We explicitly calculate Y; this deviates from the CEA 861.3
definition of MaxCLL
    // which approximates luminance with max(R, G, B).

    DX::ThrowIfFailed(histogramMatrix-
>SetValue(D2D1_COLORMATRIX_PROP_COLOR_MATRIX, rgbtoYnorm));

    // 4. Apply a gamma to allocate more histogram bins to lower luminance
levels.
    ComPtr<ID2D1Effect> histogramGamma;
    DX::ThrowIfFailed(
        context->CreateEffect(CLSID_D2D1GammaTransfer, &histogramGamma)
        );

    histogramGamma->SetInputEffect(0, histogramMatrix.Get());

    // Gamma function offers an acceptable tradeoff between simplicity and
efficient bin allocation.
    // A more sophisticated pipeline would use a more perceptually linear
function than gamma.
    DX::ThrowIfFailed(histogramGamma-
>SetValue(D2D1_GAMMATRANSFER_PROP_RED_EXPONENT, sc_histGamma));
    // All other channels are passthrough.
    DX::ThrowIfFailed(histogramGamma-
>SetValue(D2D1_GAMMATRANSFER_PROP_GREEN_DISABLE, TRUE));
    DX::ThrowIfFailed(histogramGamma-
>SetValue(D2D1_GAMMATRANSFER_PROP_BLUE_DISABLE, TRUE));
    DX::ThrowIfFailed(histogramGamma-
>SetValue(D2D1_GAMMATRANSFER_PROP_ALPHA_DISABLE, TRUE));

    // 5. Finally, the histogram itself.
    HRESULT hr = context->CreateEffect(CLSID_D2D1Histogram,
&m_histogramEffect);

    if (hr == D2DERR_INSUFFICIENT_DEVICE_CAPABILITIES)
    {
        // The GPU doesn't support compute shaders and we can't run
histogram on it.
        m_isComputeSupported = false;
    }
    else
    {
        DX::ThrowIfFailed(hr);
        m_isComputeSupported = true;
```

```
        DX::ThrowIfFailed(m_histogramEffect-
>SetValue(D2D1_HISTOGRAM_PROP_NUM_BINS, sc_histNumBins));

        m_histogramEffect->SetInputEffect(0, histogramGamma.Get());
    }
}

// Uses a histogram to compute a modified version of MaxCLL (ST.2086 max
content light level).
// Performs Begin/EndDraw on the D2D context.
void D2DAdvancedColorImagesRenderer::ComputeHdrMetadata()
{
    // Initialize with a sentinel value.
    m_maxCLL = -1.0f;

    // MaxCLL is not meaningful for SDR or WCG images.
    if ((!m_isComputeSupported) ||
        (m_imageInfo.imageKind != AdvancedColorKind::HighDynamicRange))
    {
        return;
    }

    // MaxCLL is nominally calculated for the single brightest pixel in a
frame.
    // But we take a slightly more conservative definition that takes the
99.99th percentile
    // to account for extreme outliers in the image.
    float maxCLLPercent = 0.9999f;

    auto ctx = m_deviceResources->GetD2DDeviceContext();

    ctx->BeginDraw();

    ctx->DrawImage(m_histogramEffect.Get());

    // We ignore D2DERR_RECREATE_TARGET here. This error indicates that the
device
    // is lost. It will be handled during the next call to Present.
    HRESULT hr = ctx->EndDraw();
    if (hr != D2DERR_RECREATE_TARGET)
    {
        DX::ThrowIfFailed(hr);
    }

    float *histogramData = new float[sc_histNumBins];
    DX::ThrowIfFailed(
        m_histogramEffect->GetValue(D2D1_HISTOGRAM_PROP_HISTOGRAM_OUTPUT,
            reinterpret_cast<BYTE*>(histogramData),
            sc_histNumBins * sizeof(float)
            )
        );

    unsigned int maxCLLbin = 0;
    float runningSum = 0.0f; // Cumulative sum of values in histogram is
1.0.
```

```
    for (int i = sc_histNumBins - 1; i >= 0; i--)
    {
        runningSum += histogramData[i];
        maxCLLbin = i;

        if (runningSum >= 1.0f - maxCLLPercent)
        {
            break;
        }
    }

    float binNorm = static_cast<float>(maxCLLbin) / static_cast<float>
(sc_histNumBins);
    m_maxCLL = powf(binNorm, 1 / sc_histGamma) * sc_histMaxNits;

    // Some drivers have a bug where histogram will always return 0. Treat
this as unknown.
    m_maxCLL = (m_maxCLL == 0.0f) ? -1.0f : m_maxCLL;
}
```

Step 3. Perform the HDR tonemapping operation

Tonemapping is inherently a lossy process, and can be optimized for a number of perceptual or objective metrics, so there's no single standard algorithm. Windows provides a built-in HDR tonemapper effect as part of Direct2D as well as in the Media Foundation HDR video playback pipeline. Some other commonly used algorithms include ACES Filmic, Reinhard, and ITU-R BT.2390-3 EETF (electrical-electrical transfer function).

A simplified Reinhard tonemapper operator is shown in this next code example.

C++

```
// This example uses C++. A typical DirectX implementation would port this
to HLSL.
D2D1_VECTOR_4F simpleReinhardTonemapper(
    float inputMax, // Content's maximum luminance in scRGB values, e.g. 1.0
= 80 nits.
    float outputMax, // Display's maximum luminance in scRGB values, e.g.
1.0 = 80 nits.
    D2D1_VECTOR_4F input // scRGB color.
)
{
    D2D1_VECTOR_4F output = input;

    // Vanilla Reinhard normalizes color values to [0, 1].
    // This modification scales to the luminance range of the display.
    output.r /= inputMax;
    output.g /= inputMax;
    output.b /= inputMax;
```

```
    output.r = output.r / (1 + output.r);
    output.g = output.g / (1 + output.g);
    output.b = output.b / (1 + output.b);

    output.r *= outputMax;
    output.g *= outputMax;
    output.b *= outputMax;

    return output;
}
```

Capturing HDR and WCG screen content

APIs that support specifying pixel formats, such as those in the **Windows.Graphics.Capture** namespace, and the **IDXGIOutput5::DuplicateOutput1** method, provide the capability to capture HDR and WCG content without losing pixel information. Note that after acquiring content frames, additional processing is required. For example, HDR-to-SDR tone mapping (for example, SDR screenshot copy for internet sharing) and content saving with proper format (for example, JPEG XR).

Changes to legacy color management and ICC profile behavior

Advanced color and auto color management ensure consistent and colorimetrically accurate display color for all apps, legacy and modern. However, some apps may perform their own explicit color management using International Color Consortium (ICC) color profiles.

When Advanced Color is active on either SDR or HDR displays, the behavior of display ICC profiles changes in non-backwards compatible ways. If your app works with display ICC profiles, Windows offers compatibility helpers to ensure your app continues to get correct behavior.

More info about the changes to ICC profile behavior and how you can adapt your app to maximize compatibility with Advanced Color, refer to ICC profile behavior with Advanced Color.

Additional resources

- On GitHub, *Using HDR Rendering with the DirectX Tool Kit* for DirectX 11 / DirectX 12. Walkthrough of how to add HDR support to a DirectX app using the

DirectX Tool Kit (DirectXTK).

- Direct2D advanced color image rendering sample app. UWP SDK sample app implementing an Advanced Color-aware HDR and WCG image viewer using Direct2D. Demonstrates the full range of best practices for UWP apps, including responding to display capability changes and adjusting for SDR white level.
- Direct3D 12 HDR desktop sample app. Desktop SDK sample implementing a basic Direct3D 12 HDR scene.
- Direct3D 12 HDR UWP sample app. UWP equivalent of the above sample.
- SimpleHDR_PC. An Xbox ATG SimpleHDR PC sample app (a desktop sample app) implementing a basic Direct3D 11 HDR scene.

Feedback