Digital Signage Media Player Application: Media Decode Using Intel® Media SDK and Compositing Using DXVA-HD*

Table of Contents

1. Introduction

Intel® Core™ processors have an integrated GPU that provides hardware acceleration for various types of media processing, including video decoding. Developers of digital signage media player application software can take advantage of this capability to significantly increase graphics performance. For example, testing at Intel showed the hardware acceleration reduced CPU loading from 70 percent to 10 percent as compared to software-only video decoding, around a seven times improvement (See Appendix A for platform configuration information).[1] This white paper describes how to transition from software-based media processing to hardware-based processing and provides several code examples to facilitate this migration. The white paper also proposes a mechanism for hardware-based compositing of various media content on a synchronized display to create a rich user experience.

The digital signage media player (DSMP) software architecture described in the following uses the Intel® Media SDK 2012 to invoke hardware-based video decode on Intel® Core™ processors, and Microsoft* Direct3D* and Microsoft DirectX Video Acceleration High Definition* (DXVA-HD*) for composition.

2. Digital Signage Media Player Application

2.1 A Short Description of Digital Signage Media Player Architecture

A generic digital signage media player (DSMP) architecture is shown in Figure 1. The software runs on an Intel Core processor-based platform with support for multiple displays, Intel® High Definition Audio and Intel® HD Graphics, which includes hardware-accelerated video decoding and post processing. The Intel Media SDK, Microsoft DXVA-HD and Microsoft Direct 3D provide APIs for accessing the media acceleration capabilities of the hardware platform. Developers are responsible for the compositor, media content creators and DSMP controller.

The following briefly describes the main components of the DSMP.

- DSMP controller – This handles the state machine of the DSMP.
- DirectShow* framework and Intel Media SDK – The video display framework of the DSMP is based on DirectShow. The Intel Media SDK provides the DirectShow plugin for media decoding and encoding.
- Media Content Creators – These contain the processing logic to create decoded media content (e.g., video, image and text) that can be rendered.
- Compositor and DXVA-HD – The media content is composed using DXVA-HD.
- Direct3D – The rendering pipeline of the DSMP communicates with the graphics driver using Direct3D.

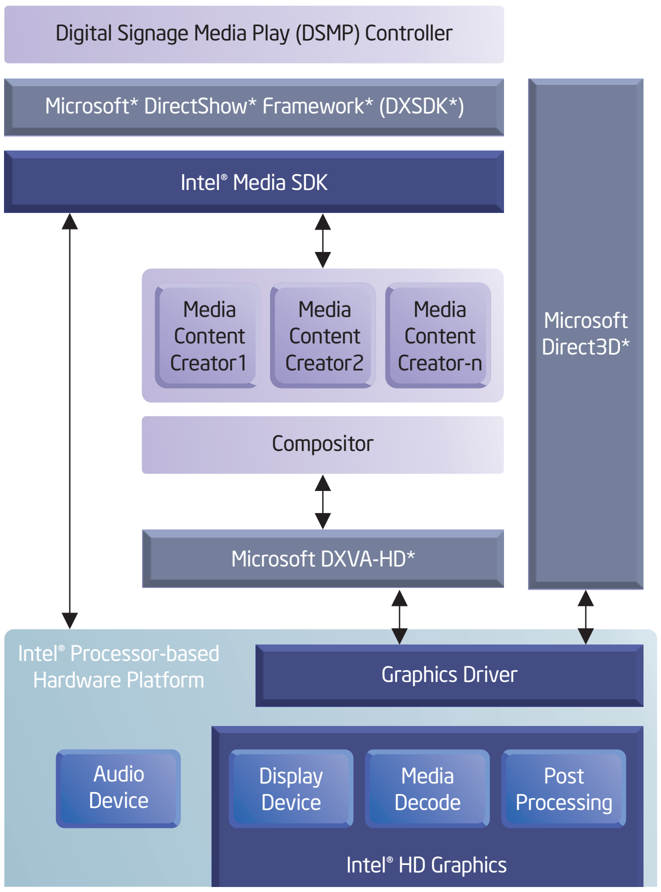

Figure 1. Digital Signage Media Player Architecture

3. Compositing

3.1 What Is Compositing and Why Is It Required?

Compositing is a technique of combining visual elements from separate sources into a single frame to create an illusion that all the elements are part of the same scene. This process involves a primary stream and multiple secondary streams. The secondary streams are mixed together with the primary stream to form a single frame that gets rendered to the display.

In the DSMP, various types of media content needs to be combined into a single frame in a synchronized way, hence compositing comes into play.

The following diagram shows the composition of multiple streams into a single stream.

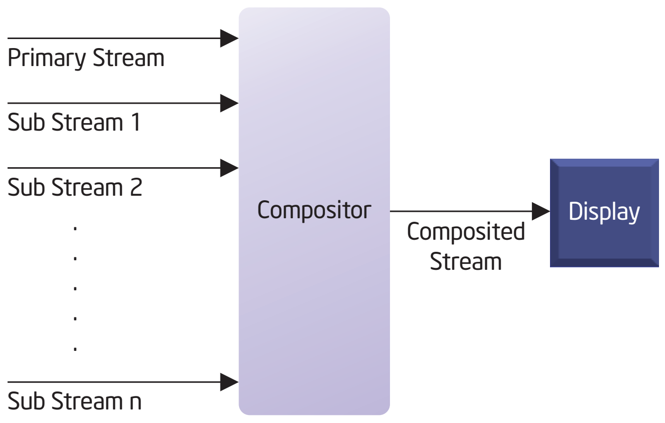

Figure 2. Compositing

3.2 Threading Architecture

The compositor in the DSMP is responsible for compositing the various media content.

Compositor as a thread

The compositor is designed as a thread, and it gets media data from various media content creators. As a thread, the compositor performs concurrent processing of media data and outputs the composed stream to the display. The compositor runs in a loop, obtaining media data from the various content creators at timed intervals and assembling a final frame to be displayed on the screen.

Refresh interval of compositor

The refresh rate of the compositor is the number of times in a second the compositor draws into the display hardware. Generally, this is equal to the display refresh rate.

Compositor in real-time

It is necessary to ensure the compositor assembles a frame in real time. A pre-defined threshold [in time] is set for the compositor to render the current frame. When this threshold is exceeded, the compositor skips a presentation cycle.

Media content creator

This entity is responsible for all the processing involved in the generation of its media content. Media content could be any of the following: image, video, text ticker, etc.

Individual media content creator as threads

All media content creators run as separate threads. Each thread is responsible for creating and preparing its media content with the help of a dedicated D3D surface. The compositor, when compositing, collects the surfaces from all content creators.

3.3 Usage of Direct 3D*

Direct3D exposes a set of API's for rendering two and three dimensional graphics in applications where performance is important. Direct3D uses hardware acceleration when available, allowing hardware acceleration of the entire or partial rendering pipeline.

*Device and surface creation in Direct3D**

The platform hardware is enumerated for Direct3D capabilities. Based on these capabilities, a Direct3D device context is created in the compositor. If creating a context with hardware acceleration fails, then a software-based context needs to be created. This device is shared across all content creators and is enumerated for available surface formats. A8R8G8B8 format is generally used as a sub-stream format. This format provides the flexibility of using alpha blending for different contents. Appropriate memory pool and surface format is selected for surface creation. The compositor considers the surface corresponding to one media content as primary and all the surfaces corresponding to the other media content as secondary.

Code snippet for device creation

```
HRESULT CreateD3DDevice(HWND hWnd, IDirect3DDevice9Ex **ppD3DDevice)
{
  // PresentationInterval specifies how frequently the back buffer
  // should be presented to the primary surface. Use parameter:
  // D3DPRESENT_INTERVAL_IMMEDIATE if you would like to render
  // as fast as possible" above this line "d3dpp.PresentationInterval =
  // D3DPRESENT_INTERVAL_ONE
  // D3D object creation for subsequent D3DDevice creation.
  if (FAILED(hr = Direct3DCreate9Ex(D3D_SDK_VERSION, &pD3D)))
  {
    *ppD3DDevice = NULL;
    return hr;
  }

  // Set up the structure used to create the D3DDevice.
  D3DPRESENT_PARAMETERS d3dpp;
  ZeroMemory(&d3dpp, sizeof(D3DPRESENT_PARAMETERS) );
  d3dpp.Windowed = TRUE;
  d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
  d3dpp.BackBufferFormat = D3DFMT_A8R8G8B8;
  d3dpp.BackBufferCount = BACK_BUFFER_COUNT;
  d3dpp.hDeviceWindow = hWnd;
  d3dpp.Windowed = TRUE;
  d3dpp.Flags = D3DPRESENTFLAG_VIDEO;
  // For windowed mode,the refresh rate must be 0 that is
  // D3DPRESENT_RATE_DEFAULT
  d3dpp.FullScreen_RefreshRateInHz = D3DPRESENT_RATE_DEFAULT;

  // D3DPRESENT_INTERVAL_ONE improves the quality of vertical sync, but
  // consumes slightly more processing time. This parameter attempts to
  // synchronize vertically.
  d3dpp.PresentationInterval = D3DPRESENT_INTERVAL_ONE;

  // Direct3D device creation.
  if (FAILED( hr = pD3D->CreateDeviceEx(AdapterOrdinal,
    D3DDEVTYPE_HAL,
    hWnd,
    D3DCREATE_SOFTWARE_VERTEXPROCESSING |
    D3DCREATE_MULTITHREADED |
    D3DCREATE_FPU_PRESERVE,
    &d3dpp,
    NULL,
    &pDevice )))
  {
    *ppD3DDevice = NULL;
    return hr;
  }

  *ppD3DDevice = pDevice;
  return hr;
}
```

Code snippet for surface creation for primary and secondary streams

```
HRESULT CreateSurfaces(IDirect3DSurface9 **pMainStream, IDirect3DSurface9
 **ppSubStream)
{
 //Surface creation for primary stream.
 if (FAILED( hr = m_pDXVAHD->CreateVideoSurface(
  rcMonitorWindowRect.right, // Width of display resolution
  rcMonitorWindowRect.bottom,// Height of display resolution
  D3DFMT_X8R8G8B8,
  caps.InputPool,        // Memory pool returned by DXVA-HD capability
  0,
  DXVAHD_SURFACE_TYPE_VIDEO_INPUT,
  1,
  pMainStream,
  NULL))
 {
  return hr;
 }

 // Texture surface creation for each secondary stream
 for (INT32 ui32Count = 0; ui32Count < ui32NoOfMediaContent; ui32Count++)
 {
  hr = m_D3D.m_pDevice->CreateTexture(
   ui32Width,
   ui32Height,
   1,
   D3DUSAGE_RENDERTARGET,
   D3DFMT_X8R8G8B8,
   D3DPOOL_DEFAULT,
   &m_Texture[ui32Count],
   NULL);

  // Get surface from texture
  if (m_Texture[ui32Count])
  {
   m_Texture[ui32ZoneNo]->GetSurfaceLevel(0, &ppSubStream[ui32ZoneNo]);
  }
 }
}
```

Media content creation

Individual content creators process media data and load this content onto Direct3D surfaces. These Direct3D surfaces are created by the individual content creators. The following is an example of video media content creation:

Code snippet for content generation

```
HRESULT VideoMediaContentCreation()
{
 // Get the device handle from compositor
 GetD3DDevice(&pID3DDevice9);

 // Check for file extension.

 // Depending on file extension create an instance of source filter.

 // Accordingly create an instance of Intel® Media SDK demuxer.

 // Connect source filter with Intel® Media SDK demuxer.

 // Check for output pin (audio/video) of Demuxer.

 // Depending on the video content, create an instance of Intel® Media
 // SDK decoder filter. Connect the Demuxer video output pin with input
  // pin of decoder.

 hr = CoCreateInstance(CLSID_EnhancedVideoRenderer, NULL,
         CLSCTX_INPROC_SERVER, IID_IBaseFilter, (void**)&pVideoRendr);

 hr = pVideoRendr->QueryInterface(IID_IMFVideoRenderer, (void**)
          &pIMFVidRend);
 if(hr == S_OK)
 {
    hr = pIMFVidRend->InitializeRenderer(NULL, pEVRPresenter);
         if(hr != S_OK)
         {
           return hr;
         }
 }

 // Create a child window and set its status to SH_HIDE.

 // Assign that window to EVR to render the output.
 VZM_SetWindow2EVR(pVideoRendr, hWnd);

 // Connect Decoder output pin to EVR input pin.
  hr = pGraph->ConnectDirect(DecoderOutPin, InputVidRndr, NULL);

  // Query the Decoder to get VPP interface. Set the VPP parameters.
  hr = VideoDecoder->QueryInterface(IID_IConfigureVPP, (void**)
                      &pVPPConfig);
  if (hr == S_OK)
  {
         pVPPConfig->SetVPPParams(&stVppParams);
  }

  // Set the pipeline state to Run
  hr = mediaControl->Run();
```

Data sharing between compositor and media content creator

When the compositor makes a request for media data, the latest available processed data from the media content creator's surface is blited onto the compositor's secondary surface to prepare the final frame. The blit takes care of the color conversion and scaling. The compositor loops at the display refresh rate to assemble the next frame. If media content creators are not ready with new data, the compositor assembles a frame using the data from the previous frame.

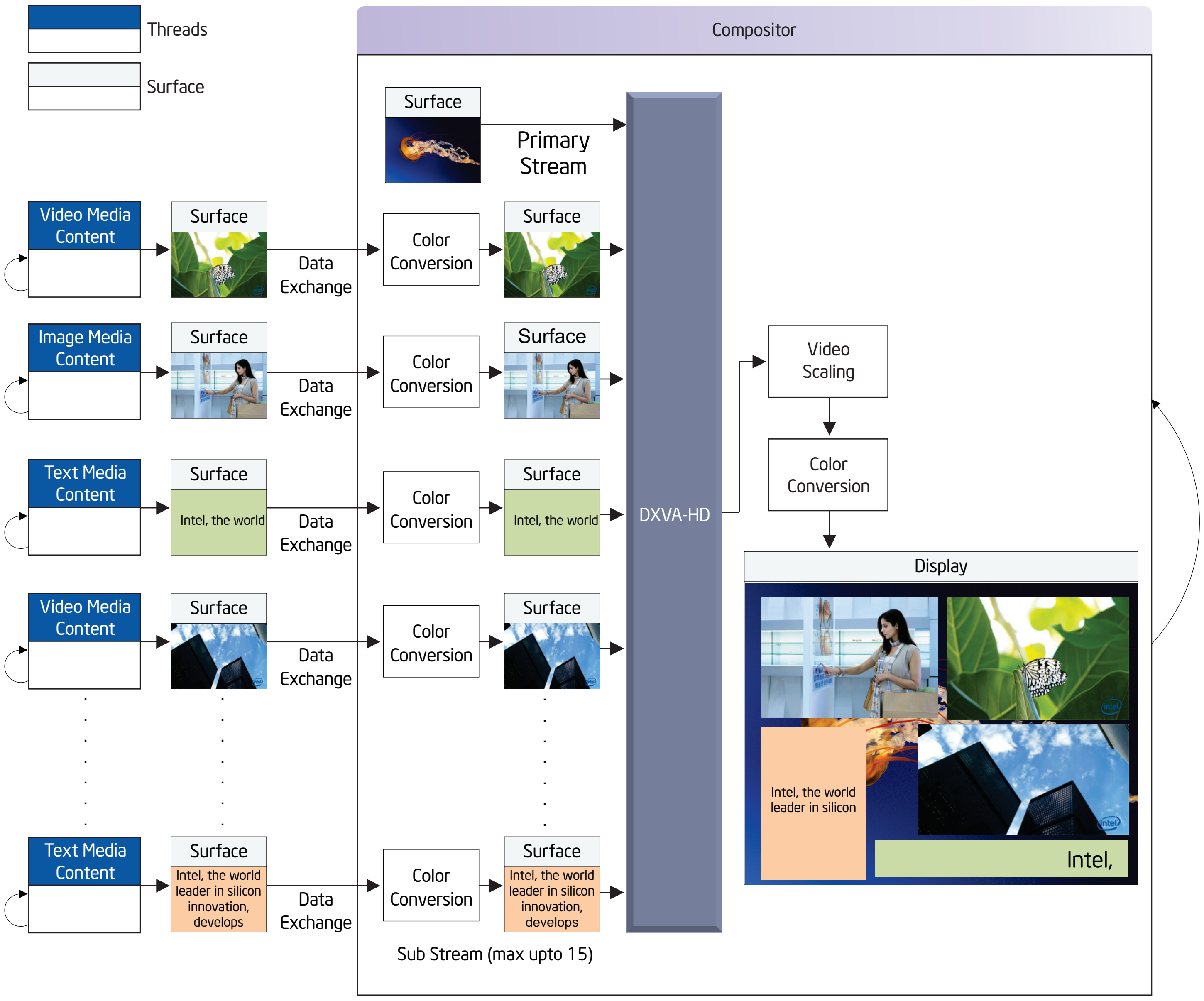

Figure 3. Data Sharing Between the Media Creator and the Compositor

3.4 Compositing using DXVA-HD*

The Microsoft DXVA-HD is an API for hardware-accelerated video processing. DXVA-HD encapsulates the functions of the graphics hardware that are devoted to uncompressed data processing such as compositing, deinterlacing, color-space conversion, alpha blending, frame rate conversion, luma keying, etc.

DXVA-HD provides a flexible composition model that enables the compositor to composite different media contents into a single video stream before rendering, thus enhancing the user experience. DXVA-HD is also used for color conversion between different color formats. DXVA-HD helps perform multiple blits in one go.

In DXVA-HD, the main video processing operation is the video processing blit. A video processing blit combines the primary stream and the secondary stream to create an output frame. The API used for performing a video processing blit is IDXVAHD_VideoProcessor::VideoProcessBltHD().

The following sequence of operation is performed before calling video processing blit.

- The compositor gets the back buffer handle.
- The DXVA-HD stream data structure is initialized.
- DXVA-HD treats the first entry of the 'DXVA-HD Stream Data' structure as the primary stream and all other entries as secondary streams. The stacking order during video processing blit will be as per the secondary stream order.
- The stream data structure is initialized for the primary stream.
- The primary stream destination rectangle is set to the display screen resolution.
- The stream data structure is initialized for each of the secondary streams.
- DXVA-HD properties, like planar alpha, video format, frame format and luma key, for each of the secondary streams are set.
- The latest data from each of the media content creators is obtained for the secondary stream surfaces.
- The video processing blit does the composition of all secondary streams and the primary stream.
- The composed frame is presented for display.

All sub-streams can have per-pixel transparency information, as well as any specified z-order. The primary stream cannot have transparent pixels, and it has a fixed z-order position at the back of all streams. The final frame is assembled onto a single surface by coloring each pixel according to the color and transparency of the corresponding pixel in all the streams.

Code snippet illustrating single blit

```
HRESULT MediaContentComposition(IDirect3DDevice9Ex* pDevice,
    IDXVAHD_VideoProcessor* pDXVAVP, INT ui32TotalMediaContents)
{
  // Allocate DXVA-HD stream data structure
  DXVAStreamData = new
      DXVAHD_STREAM_DATA[ui32TotalMediaContents+1];

  // Initialize the DXVA-HD stream data structure to 0

  // Get the render-target surface.
  hr = pDevice->GetBackBuffer(0, 0, D3DBACKBUFFER_TYPE_MONO,
&pRT);

  if (FAILED(hr))
  {
    // Handle error;
  }

  // Initialize the stream data structures for the primary video
  // stream
  DXVAStreamData[0].Enable = TRUE;
  DXVAStreamData[0].OutputIndex = 0;
  DXVAStreamData[0].InputFrameOrField = frame;
  // Primary video surface
  DXVAStreamData[0].pInputSurface = pMainStream;

  // Apply the destination rectangle for the main video stream.
  hr = DXVAHD_SetDestinationRect(
    pDXVAVP,   // DXVAHD_VideoProcessor Interface
    0,         // Stream no
    TRUE,
    Rect       // Display resolution
    );

  for (INT stream_count = 1; stream_count <=
         ui32TotalMediaContents; stream_count++)
  {
    // Initialize the stream data structures for the Secondary
    // video stream
    DXVAStreamData[stream_count].Enable = TRUE;
    DXVAStreamData[stream_count].OutputIndex = 0;
    DXVAStreamData[stream_count].InputFrameOrField = frame;
    // Assign the secondary stream surface.
    DXVAStreamData[stream_count].pInputSurface =
      ppSubStream[stream_count - 1];

    // Apply the destination rectangle for the secondary stream.
    SetRect( &rect, Substream_pos[stream_count].xLeft,
      Substream_pos[stream_count].yTop,
      Substream_pos[stream_count].xRight,
      Substream_pos[stream_count].yBottom);

    hr = DXVAHD_SetDestinationRect(
      pDXVAVP,      //DXVAHD_VideoProcessor Interface
      stream_count,  //Stream Number
      TRUE,
      rect
      );
    if (FAILED(hr))
    {
      // Handle error;
    }

    /* Set the DXVA-HD properties like planar alpha, video format,
       Frame format and Luma Key */

    hr = DXVAHD_SetStreamFormat(pDXVAVP, stream_count,
        enD3DFormat);
    if (FAILED(hr))
    {
      // Handle error;
    }

    // Frame format (progressive)
    hr = DXVAHD_SetFrameFormat(pDXVAVP, stream_count,
        DXVAHD_FRAME_FORMAT_PROGRESSIVE);
    if (FAILED(hr))
    {
      // Handle error;
    }

    hr = pDevice->SetRenderTarget(0, ppSubStream[stream_count]);
    hr = pDevice->BeginScene();
    /* Get the data from the particular media content creator. */
    hr = GetMediaContent();
    hr = pDevice->EndScene();
  }

  // Perform the blit.
  hr = pDXVAVP->VideoProcessBltHD(
    pRT,
    frame,
    ui32TotalMediaContents + 1,
    stream_data
    );

  if (FAILED(hr))
  {
    // Handle error;
  }

  // Present the frame.
  hr = pDevice->Present(NULL, NULL, NULL, NULL);
}
```

Multiple blits for handling large number of sub streams

Query the device capability with GetVideoProcessorDeviceCaps() to get the maximum number of streams ['MaxInputStreams'] that a video processing blit can handle in one go. In the case where the number of streams is less than 'MaxInputStreams-1' (one being the primary), then one blit operation is sufficient. If the number of streams is more than 'MaxInputStreams-1', the video processing blit operation must be performed multiple times, with the previously composed frame [output of previous blit] used as the primary stream. Each consecutive video processing blit must be able to handle the 'MaxInputStreams-1' number of sub streams.

Interoperability between DXVA-HD and Direct3D**

Surfaces created by a Direct3D device can be used by a DXVA-HD device and vice versa. This interoperability avoids a level of copying between the surfaces created by Direct3D and DXVA-HD devices. The compositor creates a Direct3D surface for all the media content that it needs to assemble. These surfaces are passed as sub-streams to the DXVA-HD device.

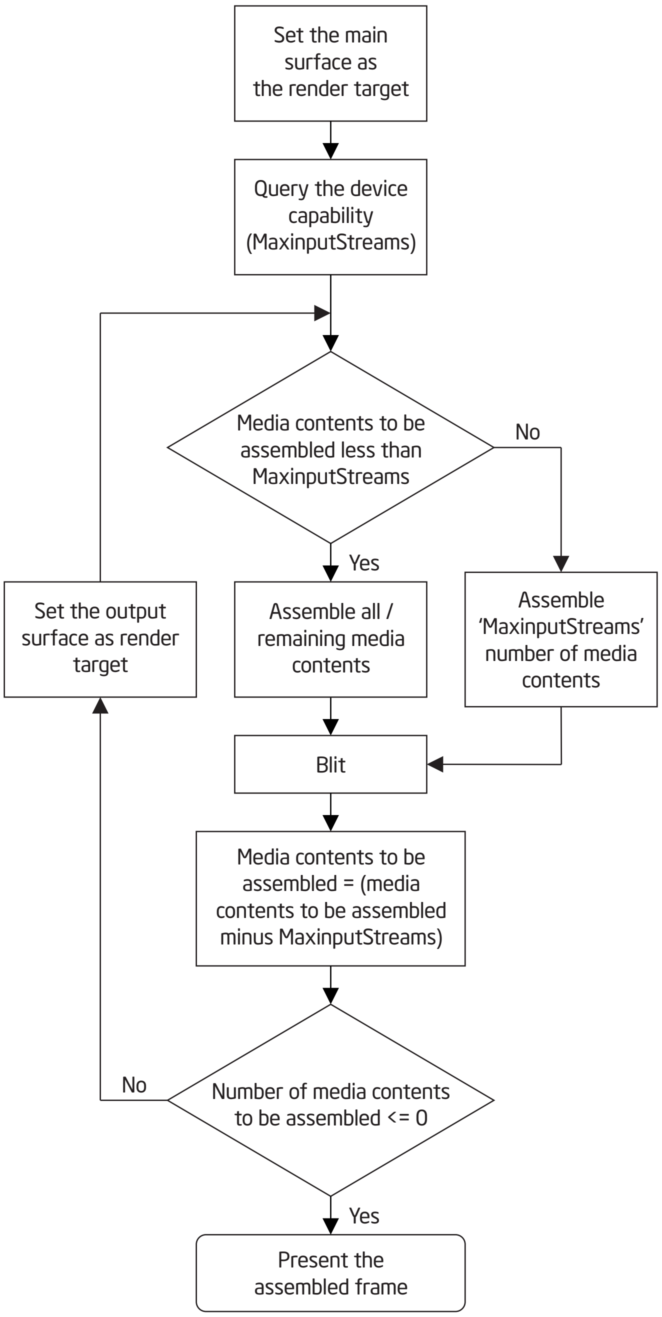
Set the main surface as the render target
Query the device capability (MaxinputStreams)
Media contents to be assembled less than MaxinputStreams
No
Yes
Set the output surface as render target
Assemble all / remaining media contents
Assemble 'MaxinputStreams' number of media contents
Blit
Media contents to be assembled = (media contents to be assembled minus MaxinputStreams)
No
Number of media contents to be assembled <= 0
Yes
Present the assembled frame

Figure 4. Flowchart for Multiple Blit

3.5 Custom EVR Presenter for Video Composition

The DSMP creates video content via the DirectShow framework. A typical DirectShow video pipeline is depicted in Figure 5.

The renderer, in this case the Enhanced Video Renderer (EVR), is usually the last block in the pipeline. EVR filter is used to render video onto the display. Internally, the EVR uses a mixer object for mixing the streams. The assembled frame is handed off to a presenter object, which schedules them for display. In order to prevent the EVR from presenting the same data it passes to the compositor to assemble, the presenter module in the EVR requires customization. The custom presenter allows the sharing of video data between the video content creator and compositor. Customization involves maintaining a queue in the presenter, which contains frames ready for rendering. The frame is added to the queue when it is scheduled for rendering. Whenever the compositor is assembling the frame, it queries a frame from this queue.

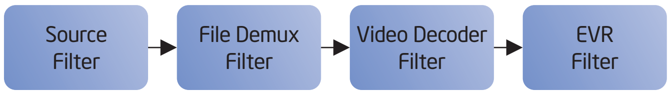

Figure 5. Typical DirectShow* Pipeline

Figure 6 illustrates the presenter queue implementation and data sharing with the compositor.

The mixer passes the decoded video frame received from the decoder to the presenter. The scheduler schedules the frame for presentation as per the presentation time. The frame remains in the scheduler queue until presentation time, after which the frame is added to the compositor queue for display. The compositor queries the presenter at the display refresh rate for the video frame. The frame handler blits the video frame to the compositor's surface and removes that frame from the compositor queue.

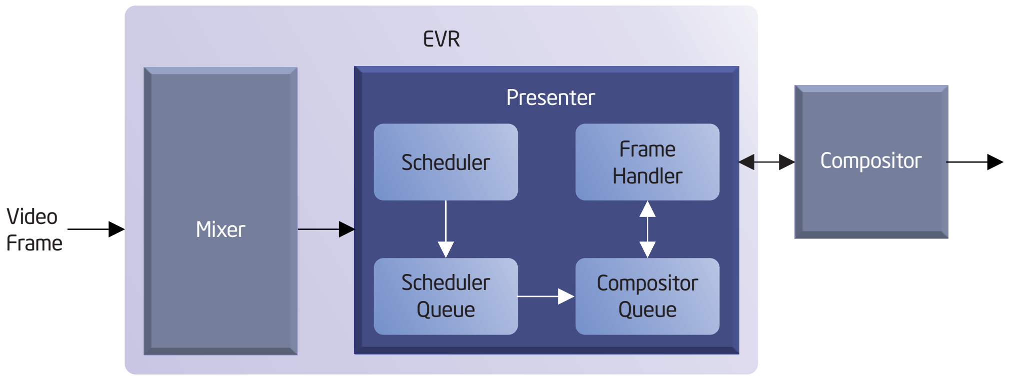

Figure 6. Custom Presenter

The following code snippet demonstrates video data sharing between the presenter and compositor:

```
  BOOL EVRCustomPresenter::GetVideoData(IDirect3DSurface9
*pCurrentSurface)
{
  HRESULT hr = S_FALSE;
  IMFSample *pSample = NULL;
  CComPtr<IDirect3DSurface9> pSurface = NULL;

  AutoLock lock(m_ObjectLock);

  // Video frame is available only when the video pipeline is in running
  // state
  if (IsActive() && pCurrentSurface != NULL)
  {
    pSurface = m_pD3DPresentEngine->GetSurface();
    if(pSurface != NULL)
    {
      hr = D3DXLoadSurfaceFromSurface(pCurrentSurface, NULL, NULL,
          pSurface, NULL, NULL, D3DX_FILTER_LINEAR, NULL);

            if (FAILED(hr))
            {
              goto done;
            }
    }
  }
  else
  {
    // This might be because the stop command has come.
    // We need to release all the surfaces in Queue
    while(1)
    {
      pSurface = m_pD3DPresentEngine->GetSurface();
            if(pSurface != NULL)
            {
                        pSurface.Release();
                        pSurface = NULL;
            }
            else
              break;
    }
  }

done:
  if (pSurface != NULL)
  {
    pSurface.Release();
    pSurface = NULL;
  }
  if(hr != S_OK)
    return FALSE;

  return TRUE;
}

IDirect3DSurface9* D3DPresentEngine::GetSurface()
{
  HRESULT hr = S_OK;
  IDirect3DSurface9 *pSurface = NULL;
  IMFMediaBuffer* pBuffer = NULL;
  IMFSample *pSample = NULL;

  AutoLock lock(m_ObjectLock);

  m_pCurrentSurface.Dequeue(&pSample);

  if (pSample)
  {
    // Get the buffer from the sample.
    hr = pSample->GetBufferByIndex(0, &pBuffer);
    if (FAILED(hr))
    {
      goto done;
    }

    // Get the surface from the buffer.
    hr = MFGetService(pBuffer, MR_BUFFER_SERVICE,
        __uuidof(IDirect3DSurface9), (void**)&pSurface);
  }

   done:
 SAFE_RELEASE(pSample);
 SAFE_RELEASE(pBuffer);
 return pSurface;
    }
```

The presenter creates and maintains multiple swap chains with a single back buffer. The video sample object holds a pointer to the swap chain's back buffer surface. It is to this surface that the mixer renders. The surface is pushed to the scheduler queue from where it is added to the compositor queue for presentation. In the EVR presenter, the Direct3D device created by compositor is used instead of creating a new device instance.

4. Media Decode Using Intel® Media SDK

4.1 Why Intel® Media SDK?

The Intel Media SDK is the software development library that exposes Intel platforms' industry-leading media acceleration capabilities (encoding, decoding and transcoding). The library APIs are available with software fallback option. With a forward scalable interface, plus easy-to-use coding samples and documentation, application developers can gain a time-to-market, competitive advantage with respect to optimized power and performance characteristics. Applications built with the Intel Media SDK can consistently deliver a high-quality, feature-rich user experience across all platforms and devices. The latest version of the Intel Media SDK can be downloaded from http://software.intel.com/en-us/articles/vcsource-tools-media-sdk/.

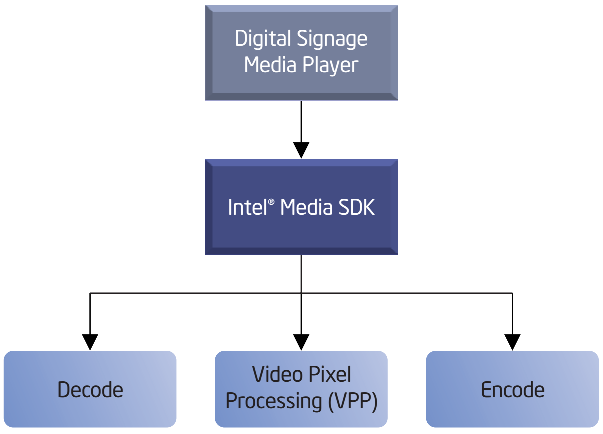

Figure 7. Intel® Media SDK

Advantage of using the Intel® Media SDK

Developers using the Intel Media SDK no longer have to write separate code paths to tap into platform-specific hardware acceleration to improve video performance. The Intel Media SDK features a single API that streamlines workflow and exploits hardware acceleration capabilities within Intel hardware. Additionally, applications integrating Intel Media SDK today will benefit from the hardware acceleration capabilities of future graphics processing solutions without requiring a program code rewrite.

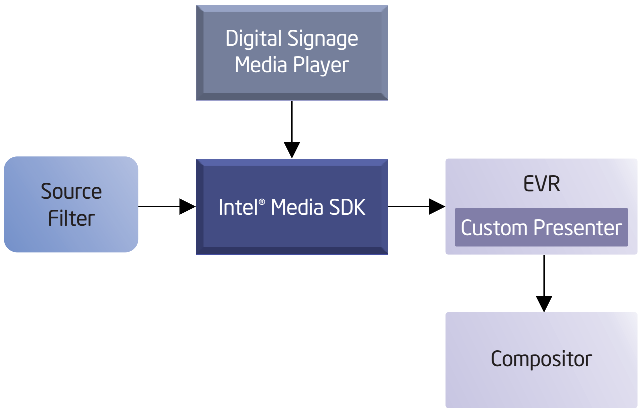

Figure 8. Intel® Media SDK-based DirectShow Filter Graph

The Intel Media SDK builds on top of standard APIs like Microsoft DXVA and Direct3D. Developers need not know the low-level details of these complex standard APIs. Surface types created by all three APIs are compatible.

The Intel Media SDK handles multiple simultaneous sessions for video processing. When hardware acceleration is not available for decode, the Intel Media SDK will fall back to software. Video decoding from H.264, MPEG-2 and VC-1 formats is supported.

4.2 Intel® Media SDK usage in DSMP

The DSMP uses Intel Media SDK based DirectShow filters for decoding video content. Figure 8 shows a sample DirectShow filter graph that illustrates the use of Intel Media SDK DirectShow filters for video playback.

Intel® Media SDK interaction with Direct3D and DXVA-HD**

The Intel Media SDK natively supports I/O from both system memory and Microsoft Direct3D9 surfaces. It uses Direct3D surfaces to handle uncompressed data during decode. In the DSMP, this uncompressed data is composed using DXVA-HD.

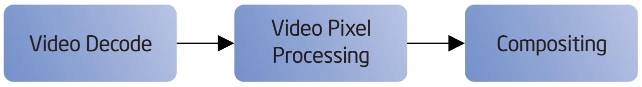

Figure 9. Video Pixel Processing using the Intel® Media

Video pixel processing using Intel® Media SDK

The Intel Media SDK supports video pixel processing functions that include inverse telecine, scene detection, deinterlacing, denoising, resizing and color conversion.

Intel® Media SDK Decoder Filter extension to support VPP

The sample Intel Media SDK decoder filters do not support video pixel processing (VPP). The DSMP extends the functionality by adding VPP support to the decoder filter. A VPP frame processor is created, which allows VPP for output video. The VPP frame processor takes raw video frames from the decoder as input. The output of the VPP frame processor is used for composition. The different pixel processing operations supported include:

- Deinterlacing
- Inverse telecine
- De-noising
- Resizing
- Scene detection
- Color conversion
- Frame rate conversion
- Crop and resize
- Detail filter
- ProcAmp

Code Snippet illustrating addition of VPP functionality in decoder

During the filter object creation, a VPP frame processor is created.

```
m_pmfxVPP = new MFXVideoVPP(m_mfxVideoSession);
```

The VPP external buffer and the VPP frame processor structure are initialized. By default all the VPP algorithms are disabled. Individual algorithms will be enabled based on user selection.

```
 // initialize vpp configuration parameters
mfxU32 DoNotUseAlg[4];

DoNotUseAlg[0] = MFX_EXTBUFF_VPP_DENOISE;
DoNotUseAlg[1] = MFX_EXTBUFF_VPP_SCENE_ANALYSIS;
DoNotUseAlg[2] = MFX_EXTBUFF_VPP_DETAIL;
DoNotUseAlg[3] = MFX_EXTBUFF_VPP_PROCAMP;
// fill VPP external buffer structure
mfxExtVPPDoNotUse VppExtDoNotUse;

VppExtDoNotUse.Header.BufferId = MFX_EXTBUFF_VPP_DONOTUSE;
VppExtDoNotUse.Header.BufferSz = sizeof(mfxExtVPPDoNotUse);
VppExtDoNotUse.NumAlg = 2;
VppExtDoNotUse.AlgList = (mfxU32* )&DoNotUseAlg;

// store the address of VPP external buffer structure
mfxExtBuffer* pVppExtBuf;

pVppExtBuf = (mfxExtBuffer* )&VppExtDoNotUse;

//Initialize VPP Frame Processor
VppParams.ExtParam = pVppExtBuf;

sts = m_pmfxVPP->Init(&VppParams);
```

During the surface creation, determine the number of surfaces required for VPP and allocate memory accordingly. The surface requirement will vary depending upon the algorithms selected for VPP.

```
sts = m_pmfxVPP->QueryIOSurf(&m_pVideoParamVPP, RequestVPP);

sts = m_pMfxAllocator->Alloc(m_pMfxAllocator->pthis, &RequestVPP,
&ResponseVpp);
```

Once the VPP is initialized and surfaces are allocated, the decoded video frame can be processed by the VPP frame processor. The generated output is then passed to downstream filter.

```
do
{
   sts = m_pmfxVPP->RunFrameVPPAsync(pInFrameSurface, *pOut-
FrameSurface,
        NULL, &syncp);

   if (MFX_WRN_DEVICE_BUSY == sts)
   {
      Sleep(1);
   }
} while (MFX_WRN_DEVICE_BUSY == sts);

if (MFX_ERR_NONE == sts)
{
   sts = m_mfxVideoSession.SyncOperation(syncp, 0xFFFF);
   MSDK_CHECK_RESULT(sts, MFX_ERR_NONE, sts);
}
```

5. Conclusion

Software vendors who write digital signage media player application software can easily switch from a software-based implementation to hardware-based processing using the Intel Media SDK. The development effort is streamlined by giving developers a means to implement code that uses hardware acceleration to increase graphics performance without the need for mastering the low-level details of the complex Microsoft APIs, like DXVA2* and Direct3D. The interoperability between the Intel Media SDK, DXVA2 and Direct3D leads to a DSMP architecture capable of delivering significant performance gains.

Compositing via DXVA-HD facilitates a rich user experience by assembling together a number of media streams and rendering them in a synchronized way.

6. Abbreviations

API	Application Programming Interface
DSMP	Digital Signage Media Player
DXVA-HD*	DirectX Video Acceleration – High Definition*
EVR	Enhanced Video Renderer
I/O	Input / Output
SDK	Software Development Kit
VPP	Video Pixel Processing

7. Appendix A: Platform Configuration

Appendix A: Platform Configuration

Platform	AOpen* MP67-D
Processor	Intel® Core™ i5-2520M Processor (2.5 GHz)
Memory	8GB DDR3
Operating System	Microsoft* Windows* 7 64-bit SP1
Graphics Driver	8.15.10.2669

[1] Performance results have been estimated based on internal Intel analysis and are provided for informational purposes only. Any difference in system hardware or software design or configuration may affect actual performance.

Copyright © 2012 Intel Corporation. All rights reserved. Intel, the Intel logo and Intel Core are trademarks of Intel Corporation in the United States and/or other countries.
*Other names and brands may be claimed as the property of others. Printed in USA 0512/TB/TM/PDF Please Recycle 327201-001US