Synchronous and Asynchronous Swaps on Xbox 360

By Claude Marais
Sr. Software Development Engineer
Advanced Technology Group (ATG)

Published: March 8, 2011

Abstract

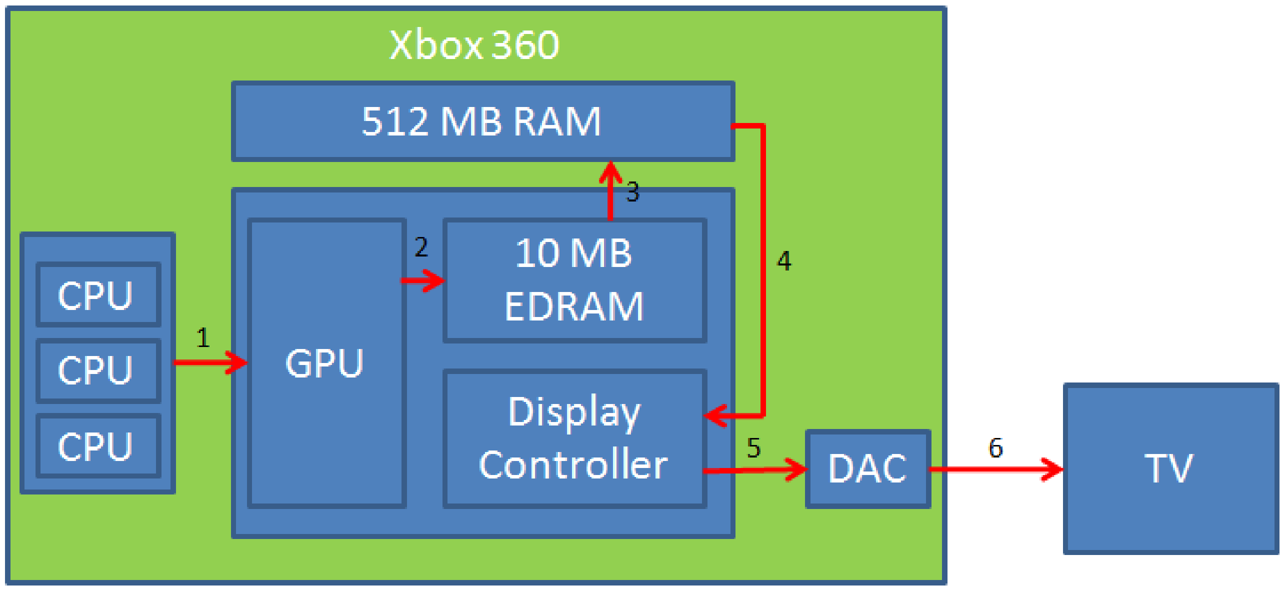
A typical game renders frames that are presented on a display device such as a TV. On the Xbox 360, the Swap or Present (which calls **Swap** internally) functions handle the presentation of the frame to the display device. The **Swap** function lets the system know that the newly rendered frame is ready to become the front buffer. The execution of the **Swap** function can either be synchronous or asynchronous. This white paper explains how synchronous and asynchronous swaps work, how to use both appropriately, and what asynchronous swaps can be used for.

Contents

Presenting a Frame on a Display Device

Before explaining how swaps work, it is useful to understand how a frame rendered on Xbox 360 is sent to your TV for display.

Figure 1. A diagram showing the data flow from the point a frame is rendered (1) by Direct3D to the point it is sent to the display device (6)



1 The title uses the CPU to update frame data, and then renders the frame by using Direct3D functions such as DrawIndexedPrimitive. These draw calls are queued, and then are sent to the GPU for processing. After the CPU finishes processing the frame, the title calls Resolve and **Swap** on the CPU, which queues a resolve and swap command to be processed on the GPU.

2 The GPU processes the draw commands and outputs a 2D frame. This 2D frame, which is stored in extended dynamic random access memory (EDRAM), is made up of shaded pixels.

3 After the GPU finishes processing the draw commands, it potentially could be stalled in order to synchronize with the display device. For synchronous swaps, synchronization is done by using the SynchronizeToPresentationInterval function. After the potential synchronization, the GPU executes the resolve command that copies the 2D frame from EDRAM into a front buffer texture stored in main memory.

4 The GPU now executes the swap command. In the case of synchronous swaps, the GPU swaps the current front-buffer pointer with the pointer to the texture that was resolved in step 3. When using a single front buffer, this will be the same pointer in memory for each frame, but **Swap** still needs to be called each frame. For asynchronous swaps, the pointer is swapped on the CPU in a callback function.

5 In parallel with the CPU and GPU, the display controller continuously reads the front-buffer pointer, and generates digital video signals. These signals include not only the front-buffer data, but also horizontal synchronization (HSync) signals and a vertical synchronization (VSync) signal. HSync signals are used to notify the display device where each row of pixels ends so that it can prepare to display the next row of pixels. The VSync signal is used to notify the display device that an entire new frame is ready to be displayed. VSyncs are generated at a regular interval of 16.6 milliseconds (ms) (20 ms in the case of 50 Hz PAL), and are generated during the vertical blanking (VBlank) interval. The VBlank interval is the time interval between the display of the last pixel of the current frame, and the display of the first pixel of the next frame.

6 If the user has selected an analog output mode, the digital video signal generated by the video controller has to be converted to an analog signal. This is done by the digital-to-analog converter (DAC). If the display device is digital and a digital connection like High-Definition Multimedia Interface (HDMI) is used, the DAC is not needed and the digital video signal can be sent directly to the display device. Finally, the rendered frame can now be displayed on the display device.

Synchronous Swaps

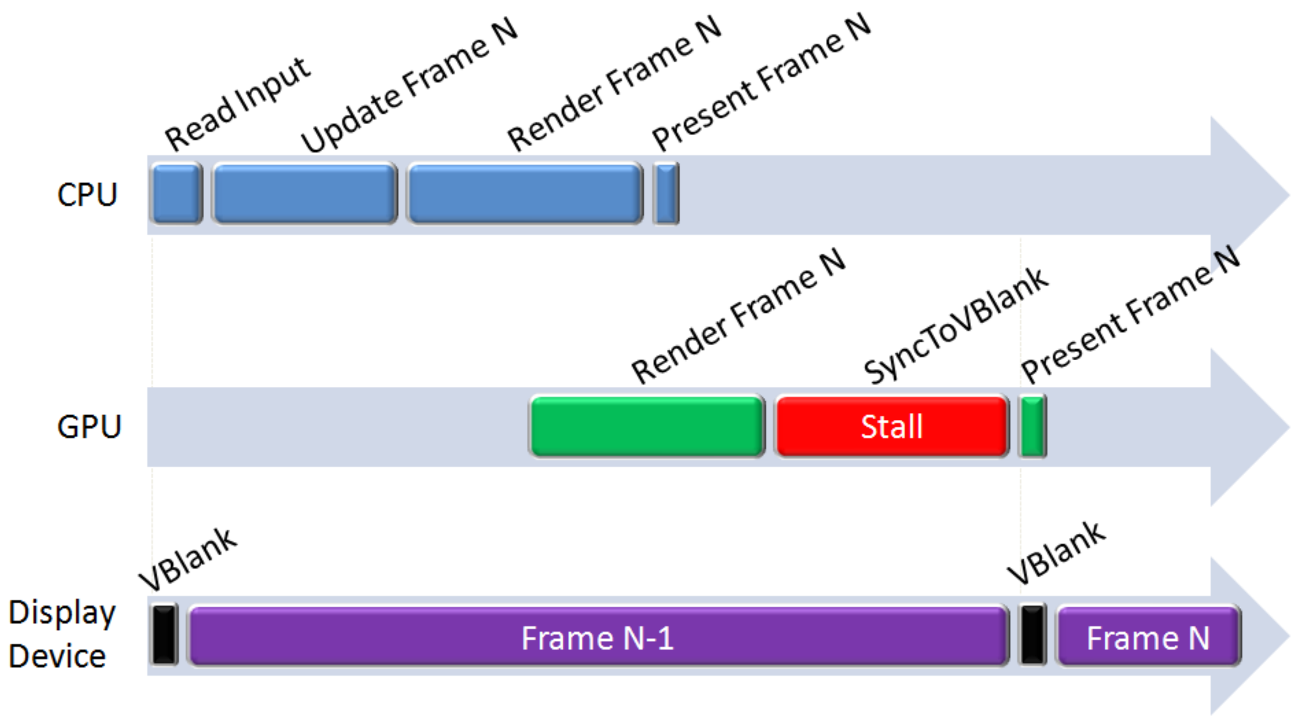
Now that the process of how a rendered frame is transferred to the TV is explained, how synchronous swaps work can be discussed in more detail.

As its name suggests, a synchronous swap is a blocking operation that forces events to occur in a specific order. Specifically, it blocks the GPU from performing the end-of-frame resolve before it can continue processing the next frame.

A typical game engine architecture can be simplified to:

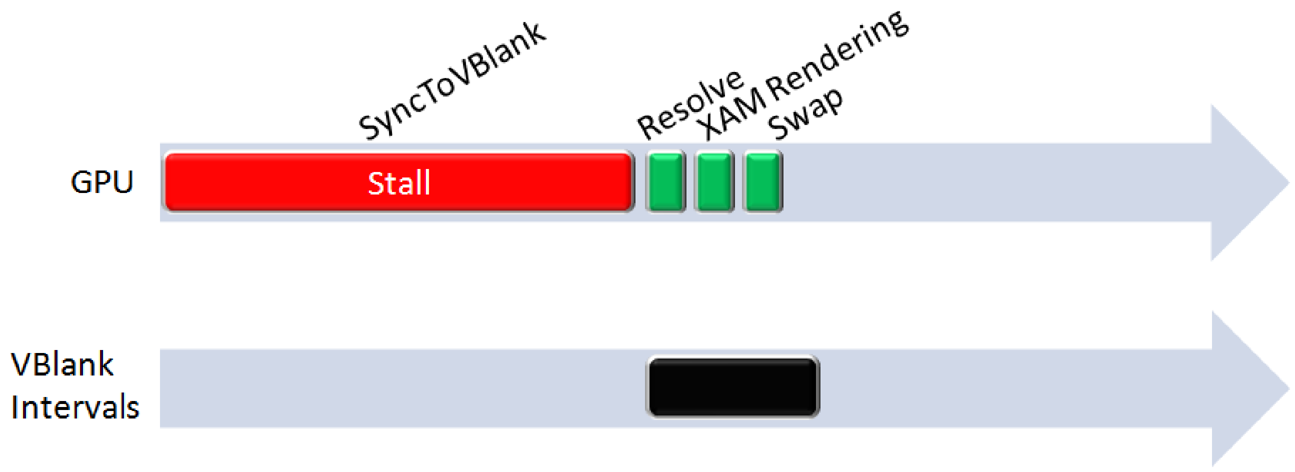
- Read input
- Update frame
- Render frame
    - **DrawIndexedPrimitive**, RunCommandBuffer, etc.
- Present frame
    - **SynchronizeToPresentationInterval**
    - **Resolve**(*pFrontBufferTexture*)
    - **Swap**(*pFrontBufferTexture*)

Figure 2. Diagram showing simplified game engine architecture with three independent timelines



When using synchronous swaps, a call to **Swap** means that the GPU is responsible for swapping the pointer of a texture—for example, *pFrontBufferTexture*—to be the current front buffer. This pointer swapping happens only once the GPU reaches the swap command in the GPU command buffer. When using a single front-buffer texture, the same pointer in memory will be used, therefore no pointer swapping occurs. Because **Swap** still needs to be called each frame, the term swap in this case continues to be used.

Figure 3. Diagram showing more details about Figure 2, including the resolve and swap commands that gets executed when presenting a frame, and also the Guide and system notifications rendered before the swap



Preventing Tearing

Tearing is a visual artifact seen when data from two or more different frames are shown on the same screen draw of the display device. It is noticeable where edges of objects fail to line up between one frame and another. Two kinds of tearing exist:

- Swap tearing

  Tearing can be caused by swapping the video output pointers in the middle of displaying a frame. This causes the video controller to read data from a different memory location while still busy presenting a frame on the display device. This kind of tearing is referred to as swap tearing.

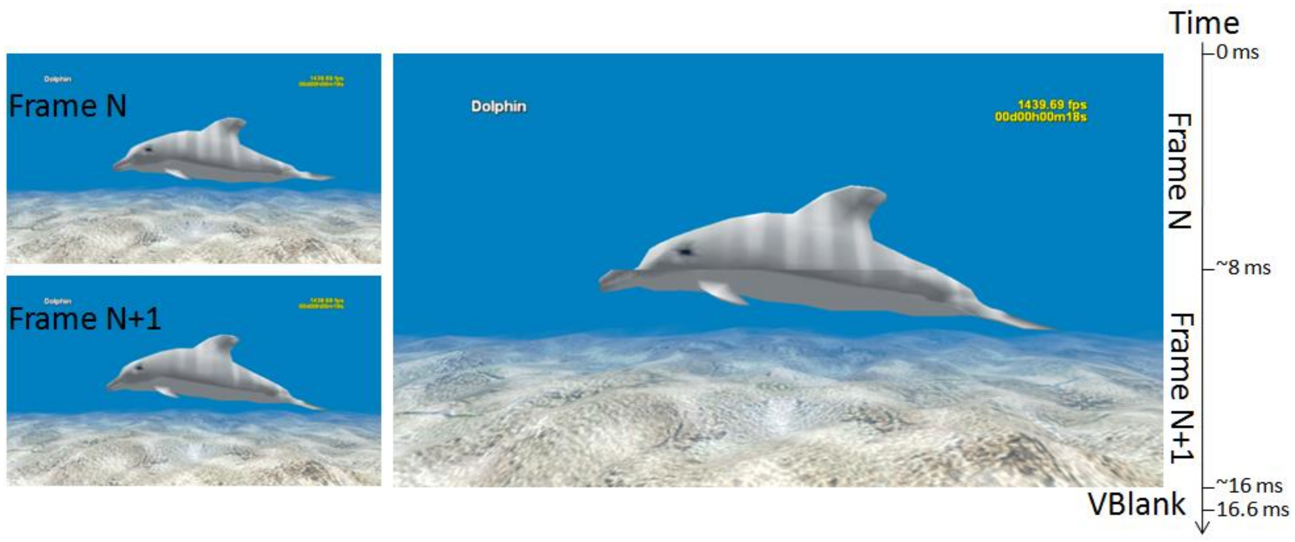
  Swap tearing happens between the current and next frame.

- Resolve tearing

  Tearing can also be caused by overwriting the currently visible front buffer or pending front buffer. Although the same video output pointer is being used, the data at that memory location has changed. We'll refer to this kind of tearing as resolve tearing.

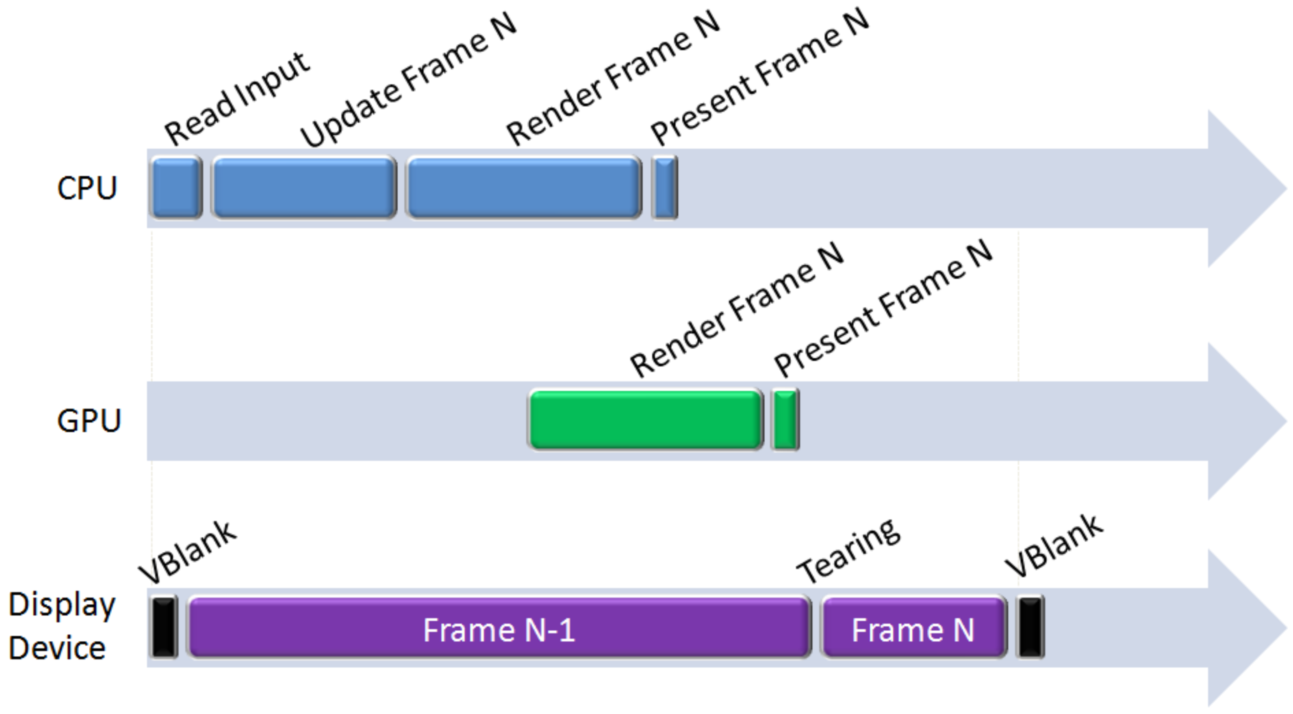
  When multiple front buffers are being used, resolve tearing can cause frames to appear on the screen out of order.

Figure 4. An example of tearing



At left are the rendered frames N and N+1. At right is the larger image showing the end result on the display device. The top of the image on the display device is displaying frame N, while the bottom of the image is displaying frame N+1. By examining the timeline to the right of the larger image, we can see that a resolve tear occurred somewhere in the middle of the frame.

Figure 5. Shows a diagram of a simplified game engine architecture that introduces tearing (assuming only one front buffer is being used) since no synchronizing to a VBlank interval is done



As illustrated in Figure 5, the resolve and swap commands on the GPU are not occurring during a VBlank interval. The **Swap** command on the GPU occurs during the presentation (Present Frame N) on the time of Figure 5; initiation of the command for the GPU is shown in Figure 3.

To prevent swap tearing if multiple front buffers are used, the swap command on the GPU needs to be executed during the VBlank interval so that the display device always presents a single frame during the same screen draw.

Depending on the presentation interval settings specified by the title, the call to **SynchronizeToPresentationInterval** could cause Direct3D to stall the GPU until there is a VBlank interval. The presentation interval settings also control the VBlank interval at which the GPU is stalled. In a PIX Timing Capture the **SyncToVBlank** event indicates this GPU stalling. When using a single front buffer, this synchronization point also prevents resolve tearing because it now is safe to overwrite the front-buffer texture with the new frame in EDRAM by using **Resolve**. Using D3DPRESENT_INTERVAL_IMMEDIATE causes tearing because the GPU will not be stalled, which causes the **Resolve** function to overwrite the front buffer before the display device finishes displaying the previous frame. In order to avoid tearing, synchronization to a VBlank interval is essential.

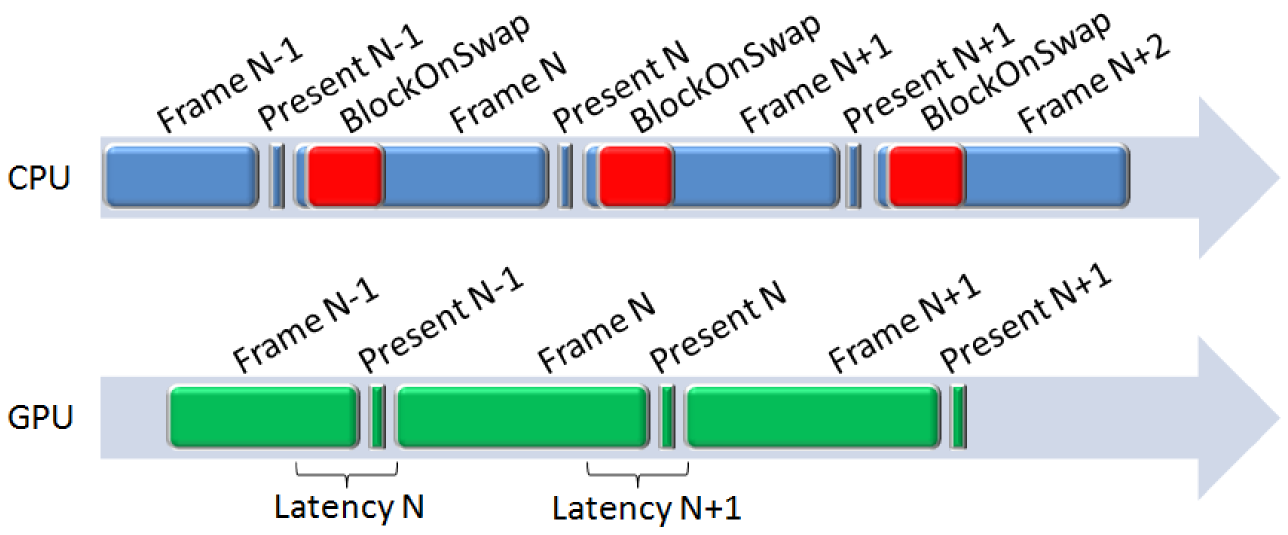
Throttling

Throttling can be defined as the act of waiting on a particular timeline (A) to reduce the latency with respect to another timeline (B)—we will call this "throttling A to B." For example, when throttling the CPU to the GPU at the start of a frame, the CPU is stalled to minimize the time interval occurring between the start of the frame on the CPU and the start of the frame on the GPU.

When using synchronous swaps, Direct3D implicitly throttles the CPU to the GPU by inserting a BlockOnSwap event on the CPU. If the title is using D3DCREATE_BUFFER_2_FRAMES, the BlockOnSwap event occurs immediately after the **Swap**. Otherwise, the BlockOnSwap event occurs at the end of the first ring-buffer segment of the frame after the **Swap**. The BlockOnSwap throttling ensures that the CPU never finishes a frame until the GPU is done processing the previous frame. In the case where the title is using D3DCREATE_BUFFER_2_FRAMES, Direct3D allows the CPU to queue two frames before throttling the CPU to the GPU. The title can also explicitly throttle the CPU to the GPU by using the InsertFence and BlockOnFence functions. **SynchronizeToPresentationInterval** throttles the GPU to the VBlank intervals of the display device unless PRESENT_INTERVAL_IMMEDIATE or D3DRS_PRESENTIMMEDIATETHRESHOLD are used.

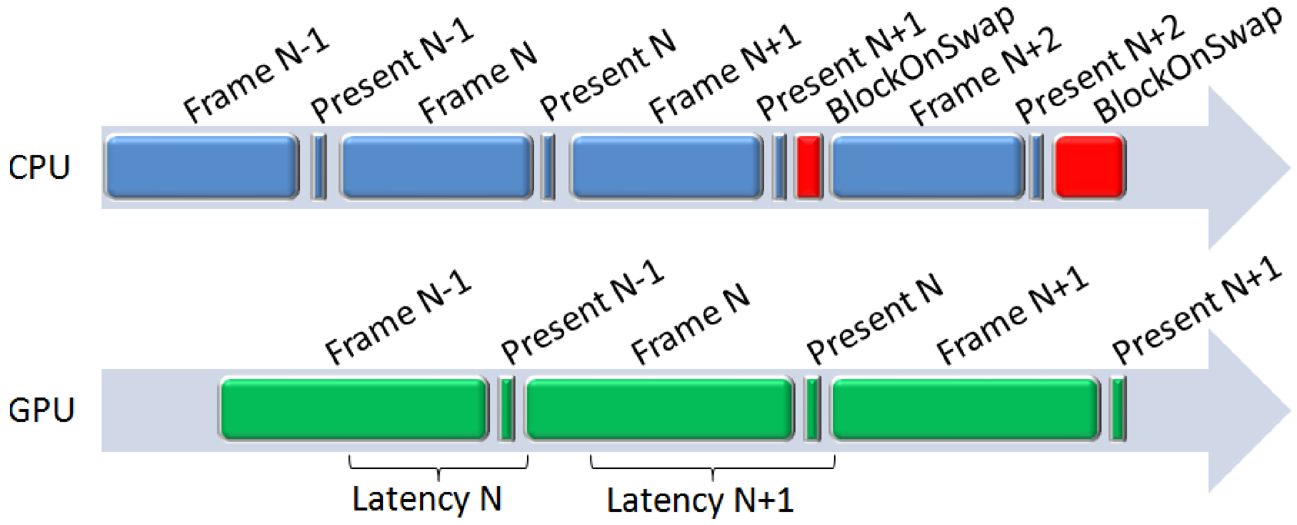
When examining a PIX timing capture, CPU throttling is indicated by the **BlockOnSwap** event and GPU throttling by the **SyncToVBlank** event. CPU throttling can also be seen by using the PIX System Monitor. The **%Frame D3D Throttled** counter indicates the percentage of the total frame time that the CPU was blocked by Direct3D on a BlockOnSwap event. When using the SetBlockCallback function, CPU throttling by Direct3D is labeled as D3DBLOCKTYPE_SWAP_THROTTLE.

Figure 6. Shows a graph of the throttling behavior that Direct3D introduces when D3DPRESENT_INTERVAL_IMMEDIATE is used without D3DCREATE_BUFFER_2_FRAMES



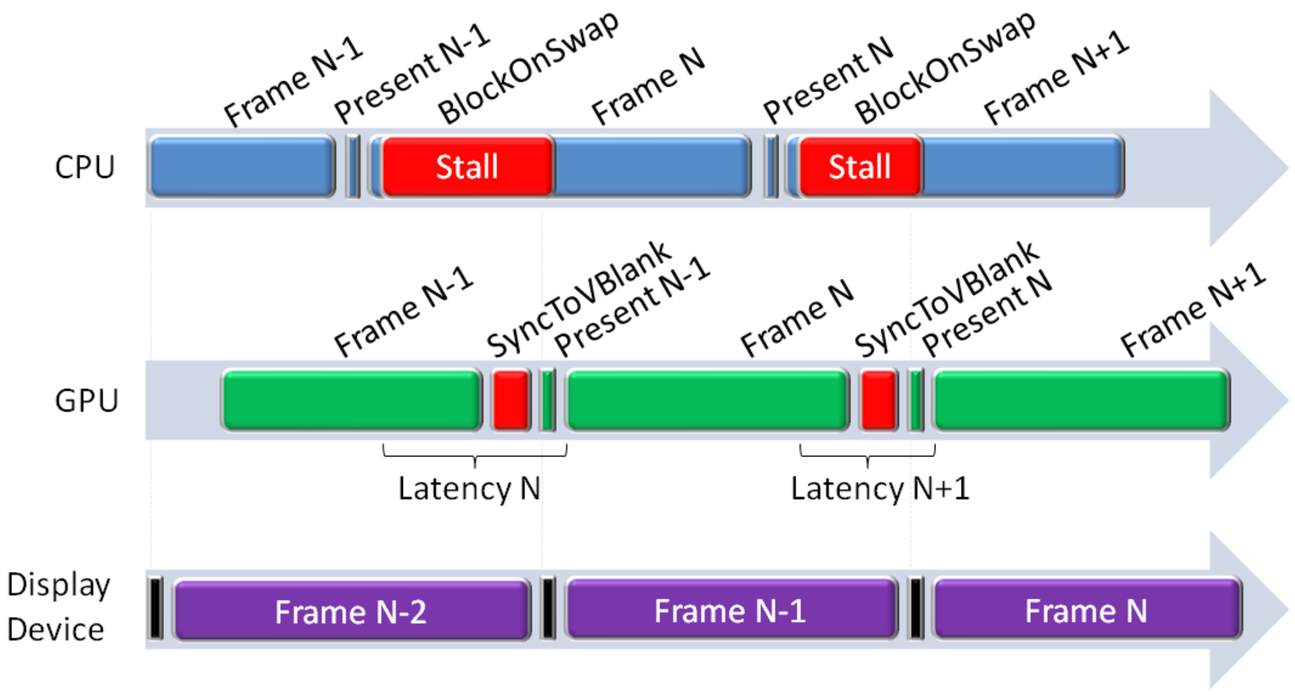
In Figure 6, Direct3D stalls the CPU until the GPU is done with the previous frame. Note how the CPU is allowed to queue only one segment of processing for frame N before it is stalled until the GPU finishes processing frame N–1.

Figure 7. Shows a graph of the throttling behavior that Direct3D introduces when D3DPRESENT_INTERVAL_IMMEDIATE and D3DCREATE_BUFFER_2_FRAMES are used



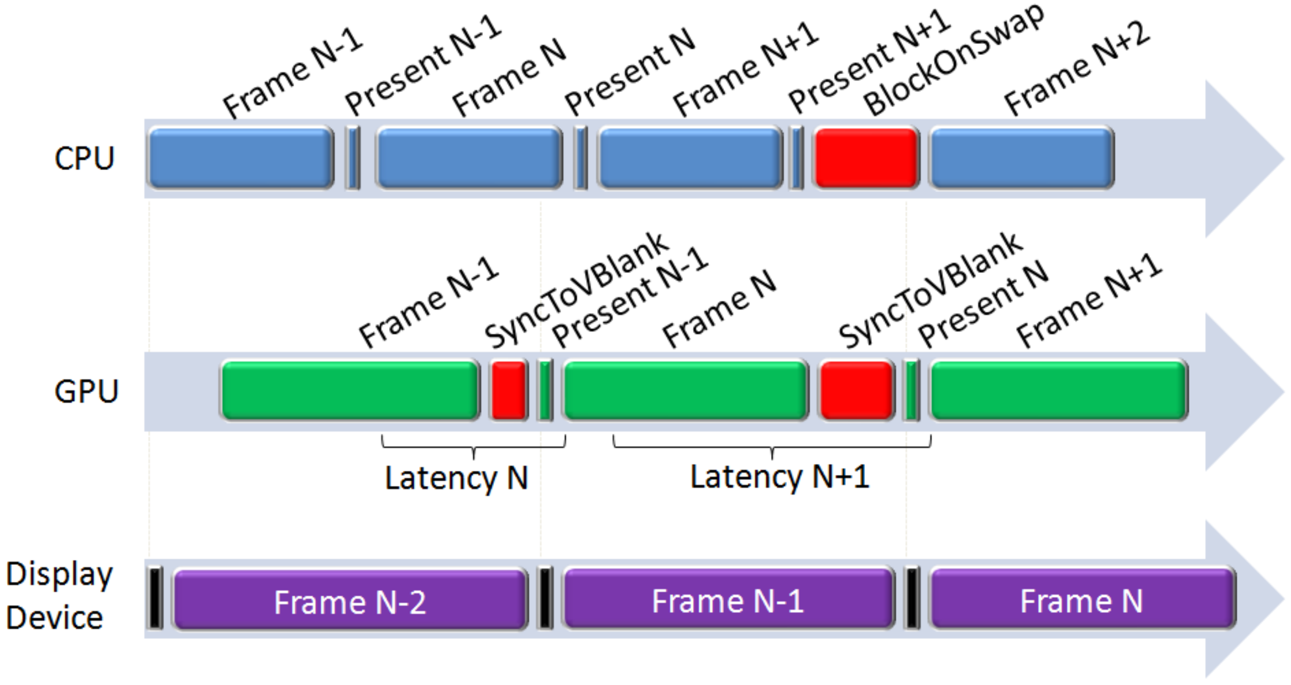
As illustrated in Figure 7, Direct3D allows the title to queue commands for two complete frames before the CPU is stalled. This could increase the latency between the start of the frame on the CPU and on the GPU when the title is GPU bound.

Figure 8. Shows a graph of throttling when D3DPRESENT_INTERVAL_ONE is used without D3DCREATE_BUFFER_2_FRAMES



As shown in Figure 8, Direct3D inserts a **SyncToVBlank** event on the GPU in order to throttle the GPU to the VBlank intervals. Indirectly, this also causes the CPU to be throttled to the VBlank intervals. Direct3D allows the CPU to fill one segment of processing for frame N before it starts throttling the CPU to the GPU. The CPU is stalled until the GPU has presented frame N−1.

Figure 9. Shows a graph of throttling when D3DPRESENT_INTERVAL_ONE and D3DCREATE_BUFFER_2_FRAMES are used



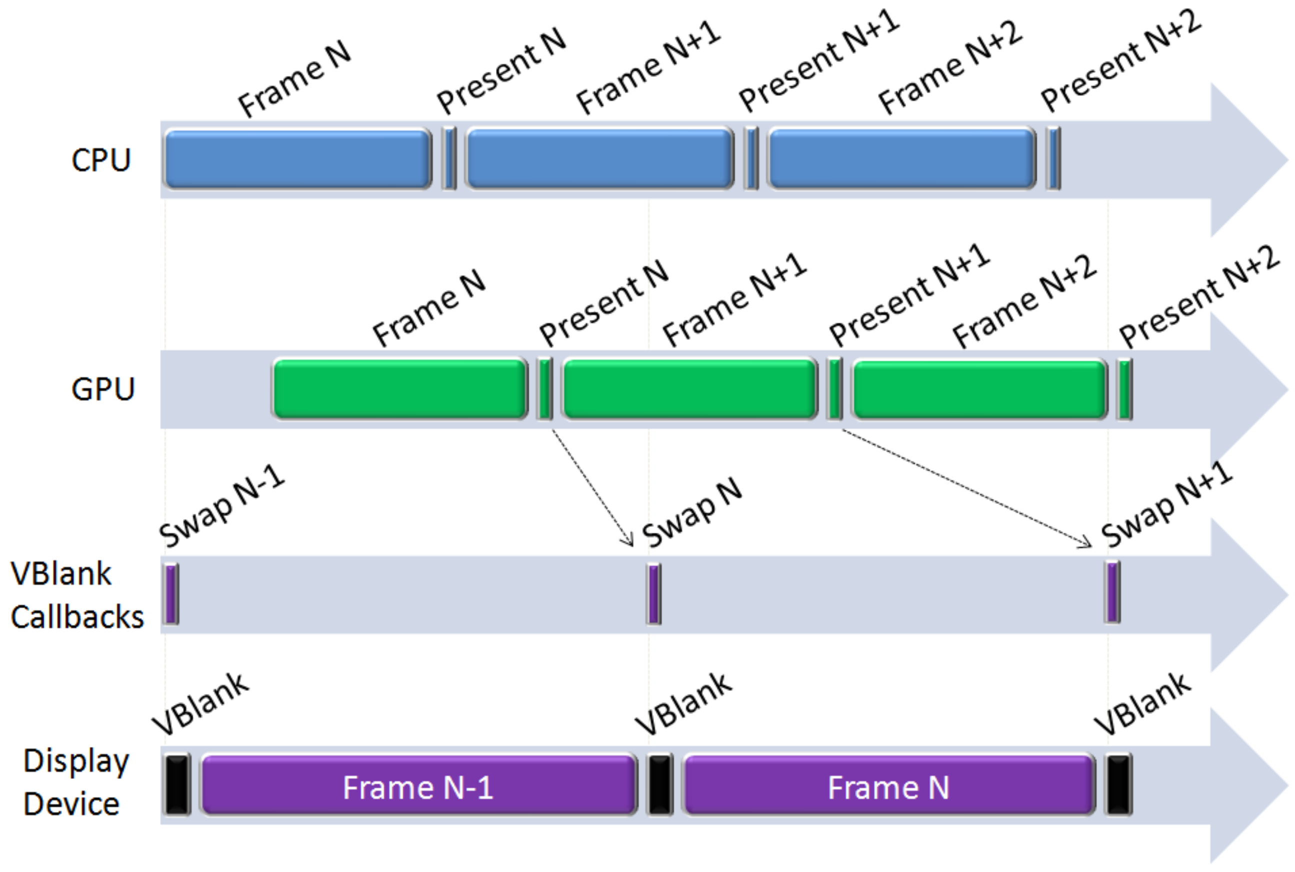
Direct3D, as shown in Figure 9, allows the title to queue commands for two complete frames before the CPU is stalled. Note how the CPU is allowed to start processing frame N+1 before frame N has been presented on the GPU. It does, however, cause extra latency between the start of frame N+1 on the CPU and on the GPU when the title is GPU bound.

Asynchronous Swaps

As its name suggests, an asynchronous swap is a non-blocking operation that allows events to happen at the same time. Specifically, the GPU does not have to block until the VBlank interval to swap in a new front buffer.

When using asynchronous swaps, a call to **Swap** means that the GPU is no longer responsible for swapping the pointer on the display hardware, but the GPU still needs to signal that the rendered front buffer is ready. The pointer swapping happens on the CPU during the VBlank callback deferred procedure call (DPC). Asynchronous swaps allow Direct3D to use multiple front buffers in order to render frames ahead of the current display without the need to stall the GPU until the VBlank interval. This decouples the GPU from the VBlank intervals so that the GPU can immediately start processing the next frame.

Figure 10. Diagram showing how asynchronous swaps operate by decoupling the GPU from the VBlank intervals



Compare the diagram in Figure10 with the one in Figure 2. Notice how the GPU can continue processing frame N+1 without having to stall until the VBlank interval where the swap happens for frame N.

With extra freedom comes extra responsibility. The title is now responsible for:

- Allocating and using at least two front-buffer textures so that one texture can be resolved into while the other is being displayed.
- Implementing a mechanism to prevent resolve tearing. Resolve on the GPU no longer occurs during a VBlank interval. For this reason, there is an increased risk of tearing that Direct3D cannot prevent.
- Implementing a throttling mechanism because all throttling in Direct3D is disabled.

Table 1. Shows differences in responsibilities between synchronous and asynchronous swaps

	Synchronous swaps	**Asynchronous swaps**
Swap video output pointers	GPU (swap command)	CPU (VBlank callback)
Prevent swap tearing	Direct3D (SyncToVBlank)	Direct3D (VBlank callback)
Prevent resolve tearing	Direct3D (SyncToVBlank)	Title
CPU throttling	Direct3D (BlockOnSwap)	Title
GPU throttling	Direct3D (SyncToVBlank)	Title

To enable asynchronous swaps, use either the D3DCREATE_ASYNCHRONOUS_SWAPS flag when calling **CreateDevice**, or use the **SetSwapMode** function to switch between synchronous and asynchronous at run time.

Preventing Tearing

Two types of tearing—swap tearing and resolve tearing—were defined. When using asynchronous swaps, swap tearing is automatically prevented because pointer swapping happens in the VBlank callback that occurs during a VBlank interval. Note that neither the CPU, nor the GPU needs to be blocked until there is a VBlank interval to avoid swap tearing.

Resolve tearing has to be prevented by the title. This is accomplished by implementing a mechanism that ensures the GPU does not overwrite a front buffer that is currently visible or pending. The mechanism, which can be implemented in several ways, must block the GPU until it is safe to overwrite the front buffer. Two different implementations are discussed next. One uses a spin-lock, and the other uses a block on asynchronous resources.

Use Spin Lock to Prevent Resolve Tearing

The main goal of the resolve-tearing prevention mechanism is to stall the GPU so that the resolve can occur during the earliest VBlank interval that is safe for overwriting the front buffer. The diagrams in Figure 11 and Figure 13 show a spin-lock mechanism that uses two front buffers.

Figure 11. Diagram showing a spin-lock mechanism to prevent resolve tearing when two front buffers are used

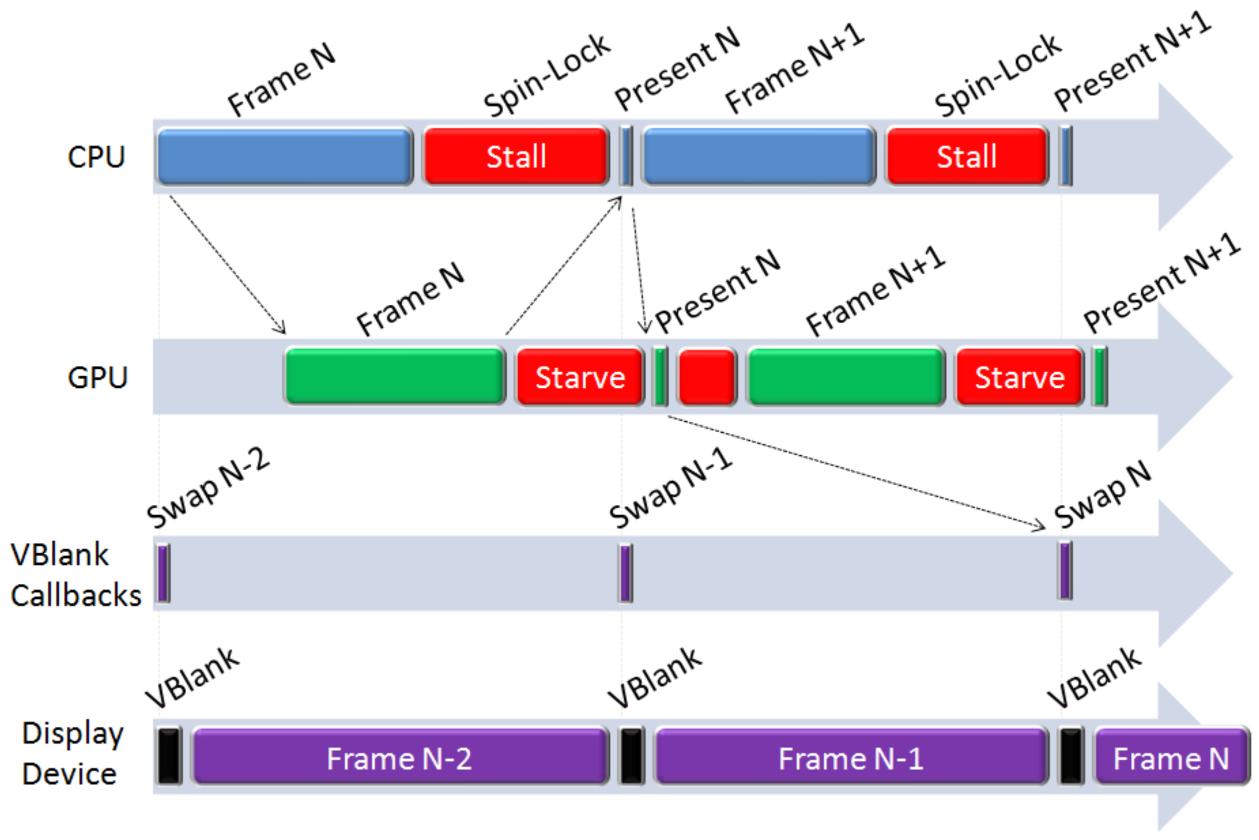


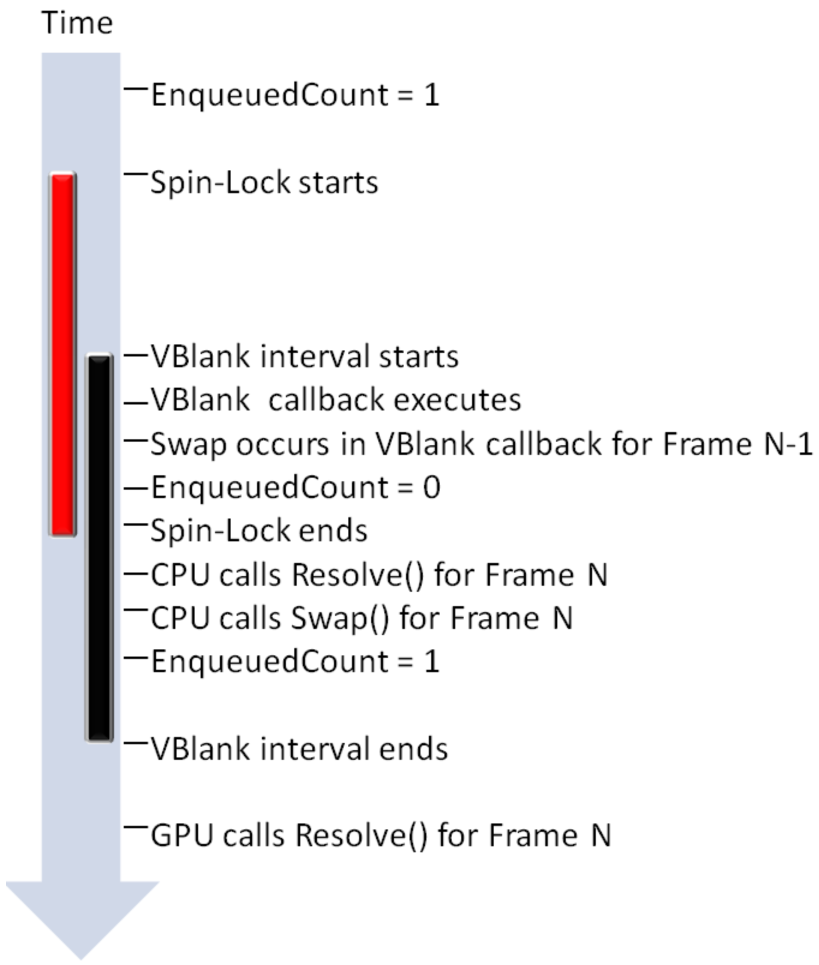
Figure 11 presents a case in which the GPU is within budget and gets starved.

The CPU processes frame N, and at some point, the work is kicked off to the GPU. When the CPU reaches the end of frame N, a spin lock is inserted before calling **Resolve** and **Swap** on the CPU. This spin lock can be implemented by using the QuerySwapStatus function. Generally, system functions such as WaitForSingleObject should be used to implement spin-lock mechanisms. In this case, however, there is no function that provides the needed functionality. Since this is a loop that just keeps the processor busy, consider using **SetHWThreadPriorityLow** for the duration of the spin lock.

```
D3DSWAP_STATUS status;
do
{
    pDevice->QuerySwapStatus( &status );
} while ( status.EnqueuedCount >= ( uNumFrontBuffers - 1 ) );
```

EnqueuedCount refers to the number of swaps that are currently pending, either because the GPU is still rendering the frame, or because the appropriate VBlank interval where the frame becomes visible has not occurred yet. In the case of two front buffers, we need to ensure that **Resolve** on the GPU for frame N occurs after the swap for frame N−1. Recall that when using asynchronous swaps, the swap occurs during the VBlank callback, which means that essentially this spin-lock implementation causes the CPU to wait until the VBlank interval during which the swap for frame N−1 occurs.

Figure 12. Shows a timeline with all relevant events occurring to ensure that the spin-lock mechanism prevents resolve tearing

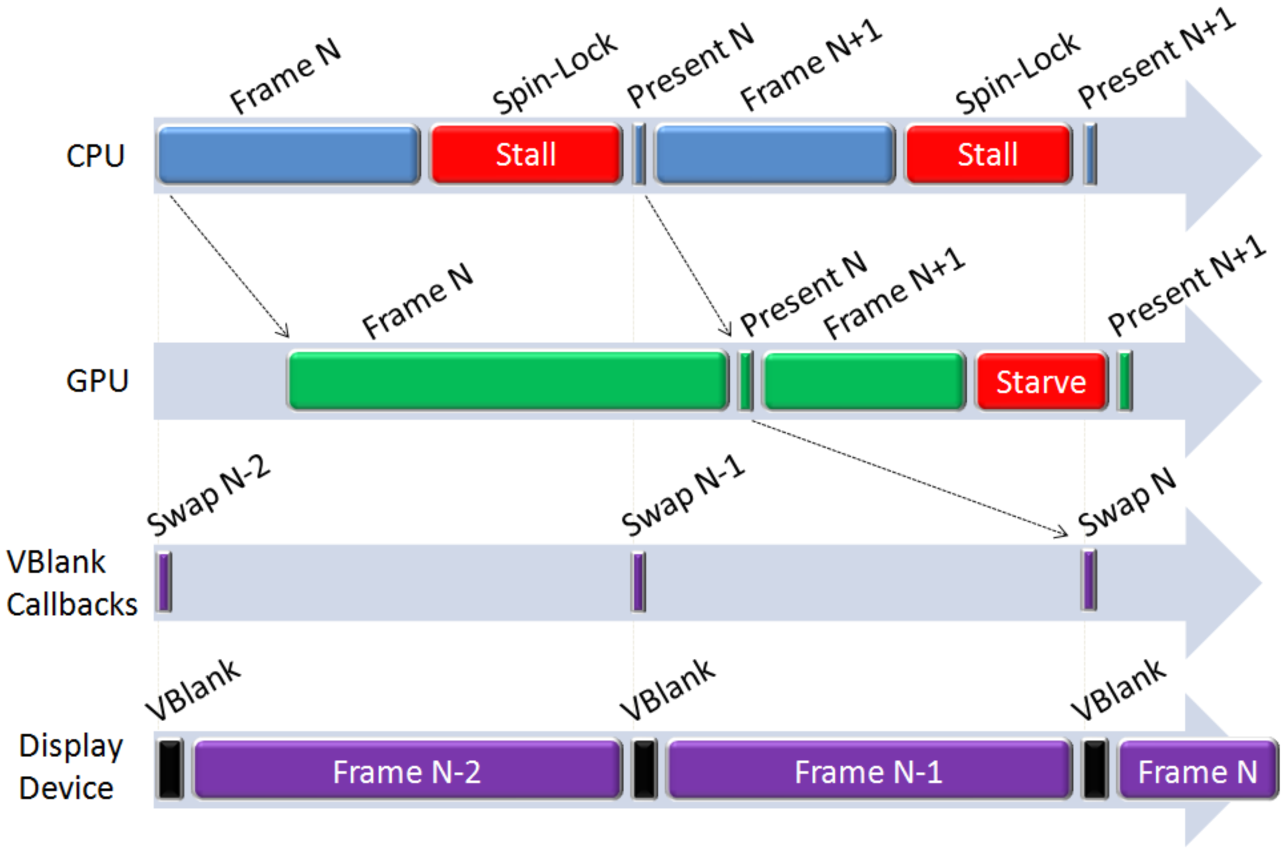


Note how the GPU identified in Figure 12 executes only after the swap in the VBlank callback for the previous frame.

After the spin lock, it is safe to overwrite the front buffer for frame N because it is not being displayed on the display device and is not pending. Note that the front buffer for frame N uses the same memory as frame N–*numFrontBuffers*. Resolve tearing is prevented because the GPU logically cannot resolve the front buffer until after the CPU calls **Resolve** for frame N.

Examining the diagram from Figure 11, it can be seen that if the GPU finishes processing frame N before the VBlank, it is starved until the CPU calls **Resolve** and **Swap**. This means that unless the CPU or GPU is over budget in such a way that other stalls consume this one, the GPU is essentially synchronized to the CPU, which is synchronized to the VBlank intervals. Putting the CPU in the middle in this way introduces a risk. If the CPU misses the VBlank interval and also ends up stalling the GPU, then GPU performance could be wasted. The GPU could have been resolving the frame, but did not because the CPU was late.

Figure 13. Diagram showing a spin-lock mechanism to prevent resolve tearing when two front buffers are used



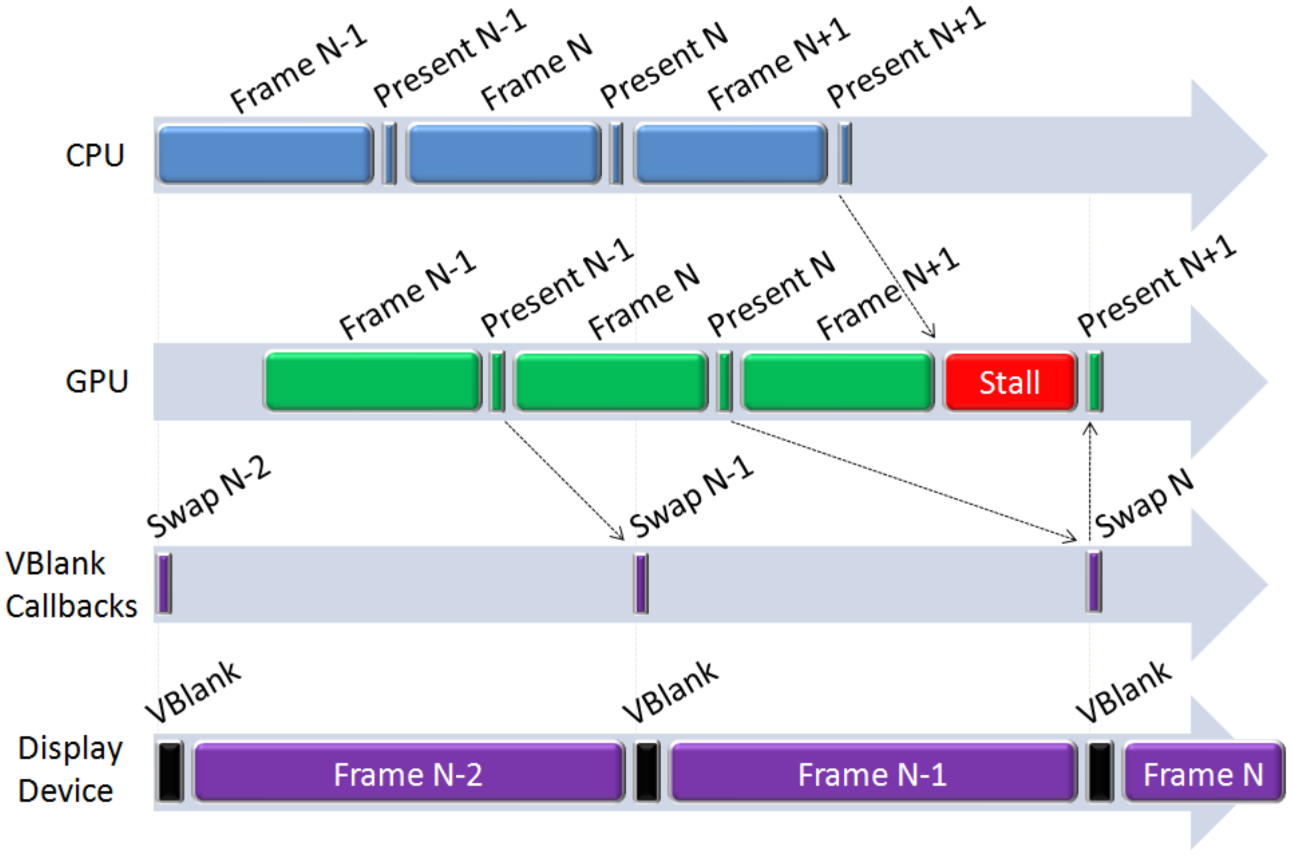
The diagram illustrated in Figure 13 shows that frame N on the GPU is over budget. Resolve tearing is prevented as long as the resolve for frame N occurs after the swap for frame N−1.

Examining the diagram from Figure 13, it can be seen that frame N on the GPU went over-budget. The mechanism still prevents resolve tearing because the resolve on the GPU occurs after the swap for frame N−1. What is interesting in this situation is that the title could decide to reduce latency by dropping frame N, and instead present frame N+1 because it has already been processed. The **SwapCallback** function could be used to dynamically decide whether to skip the frame. A practical example of this behavior would be if a title had to use the maximum allowed latency for a while, but then recovers and wants to be able to continue rending using normal latency.

Use Block on Asynchronous Resources to Prevent Resolve Tearing

At the cost of higher latency, a more efficient mechanism can be implemented by using the InsertBlockOnAsyncResources and SignalAsyncResources functions to block the GPU. This blocking method is illustrated in Figure 14.

Figure 14. Diagram showing a mechanism to prevent resolve tearing when two front buffers are used



As shown in Figure 14, a block on asynchronous resources is used to implement the mechanism.

The CPU processes frame N, and at some point, the work is passed to the GPU. When the CPU reaches the end of frame N, there is no need to stall the CPU. The CPU can immediately call **Resolve** and **Swap**, and can continue processing frame N+1. Because the end of frame N on the GPU occurs after the swap for frame N−1, the blocking time on the GPU before calling **Resolve** for frame N is zero. We do, however, need to block the GPU before calling **Resolve** for frame N+1 because the content of frame N−1 is still being shown by the display device. Calling **Resolve** without blocking the GPU until the next VBlank interval introduces tearing; the function overwrites the content of the frame buffer that is currently being displayed.

When the GPU reaches the end of frame N+1, it stalls until the **SignalAsyncResources** function is called. Because we want to stall the GPU until the VBlank interval, the most convenient place to call **SignalAsyncResources** is from within the VBlank callback. This does make the technique more complicated because the title has to safely transmit and track resources such as the handle to the asynchronous block between the rendering thread and the VBlank callback that is a DPC.

The GPU can be stalled by using the **InsertBlockOnAsyncResources** function before the call to **Resolve** on the CPU. Because multiple front buffers are used, the current swap count also needs to be recorded to ensure that the correct frame is unblocked from within the VBlank callback. The following code can be called from the title's rendering thread to insert a block on the GPU. To keep the code as simple as possible, it is assumed that only two front buffers are used.

```
// Constants for keeping track of visible frames on the display device
const D3DASYNCBLOCK AsyncBlockFrameIsVisible = 2;
const D3DASYNCBLOCK AsyncBlockFrameIsNotVisible = 4;

// In VC++ volatile ensures that the compiler don’t cache values for excessive
// time. This is useful since we read and write from different threads
D3DTexture* pFrontBuffer[ 2 ];
volatile D3DASYNCBLOCK AsyncBlock[ 2 ] = { AsyncBlockFrameIsNotVisible,
                                           AsyncBlockFrameIsNotVisible };
...

// Inserts a block on the GPU to avoid resolve tearing. This function should be
// called before Resolve and Swap
VOID BlockToAvoidTearing()
{
    PIXBeginNamedEvent( 0, __FUNCTION__ );

    // Use the Direct3D swap counter to select the current frame. Swap is only
    // called after this code, so the swap count here is from the previous
    // frame. In order to get the swap count for the currently rendered frame,
    // we need to use Swap + 1
    D3DSWAP_STATUS status;
    pDevice->QuerySwapStatus( &status );
    const DWORD dwCurrentFrameSwap = status.Swap + 1;
    const DWORD i = dwCurrentFrameSwap & 1;

    // Atomic read then compare: AsyncBlock[ i ] == AsyncBlockFrameIsVisible
    if ( InterlockedCompareExchange64( (PLONG64)&AsyncBlock[ i ], 0, 0 ) ==
                                        AsyncBlockFrameIsVisible )
    {
        // Insert a block on the GPU if the current frame buffer is still being displayed
        D3DASYNCBLOCK asyncBlock;
        asyncBlock = pDevice->InsertBlockOnAsyncResources( 0, NULL, 0, NULL, 0 );

        // If the frame remains visible, atomically assign AsyncBlock[ i ] = asyncBlock
        const D3DASYNCBLOCK previousAsyncBlock = InterlockedCompareExchange64(
                                                   (PLONG64)&AsyncBlock[ i ],
                                                   (LONG64)asyncBlock,
                                                    AsyncBlockFrameIsVisible );

        if ( previousAsyncBlock == AsyncBlockFrameIsNotVisible )
        {
            // This rarely happens. We are too late and the VBlank is gone.
            // The frame is now hidden, so we need to unblock the GPU here instead
            // of in the VBlank callback.
            pDevice->SignalAsyncResources( asyncBlock );
        }
    }

    PIXEndNamedEvent();
}


// Render a frame
VOID Render ()
{
    ...
    // Avoid resolve tearing
    BlockToAvoidTearing();

    // Resolve and swap
    pDevice->Resolve( 0, NULL, pFrontBuffer[ i ], NULL, 0, 0, NULL, 0, 0, NULL );
    pDevice->Swap( pFrontBuffer[ i ], NULL );
}
```

The following code shows an implementation of the VBlank callback that unblocks the GPU when it reaches the appropriate VBlank interval.

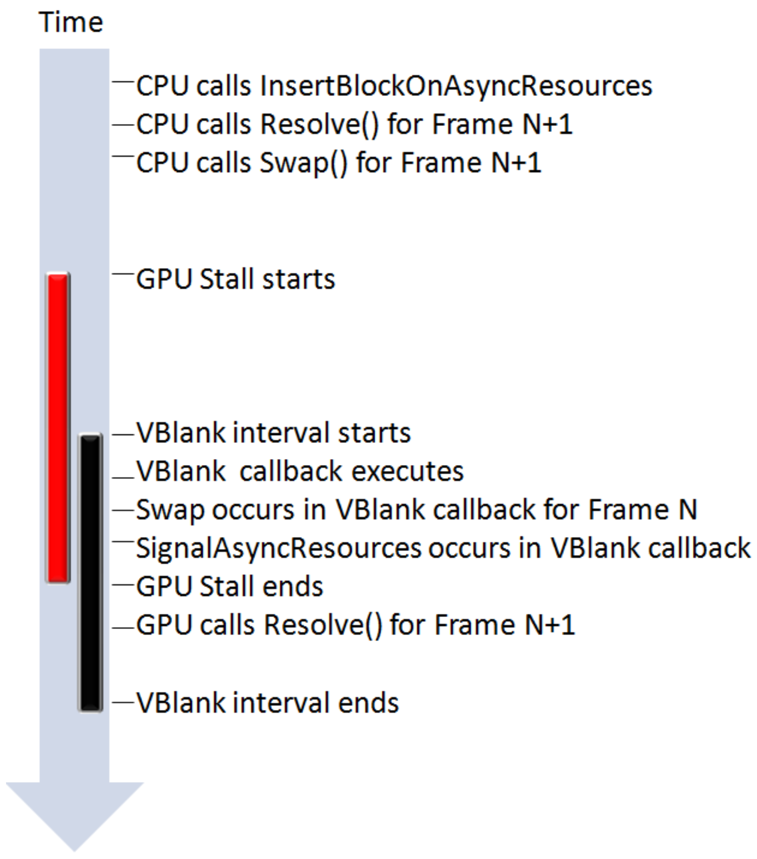
```
VOID VBlankCallback( D3DVBLANKDATA* pVBlankData )
{
    for ( UINT i = 0; i < 2; ++i )
    {
        if ( ( pVBlankData->Swap & 1 ) == i )
        {
            // The swap for frame i has just occurred so the display device
            // will now start displaying frame i, therefore we set the frame
            // to visible if it is currently not visible.
            InterlockedCompareExchange64( (PLONG64)&AsyncBlock[ i ],
                                          AsyncBlockFrameIsVisible,
                                          AsyncBlockFrameIsNotVisible );
        }
        else
        {
            // The display device has finished displaying frame i, so it is
            // safe to unblock the GPU

            // Atomic read: asyncBlock = AsyncBlock[ i ]
            // Atomic write: AsyncBlock[ i ] = AsyncBlockFrameIsNotVisible
            D3DASYNCBLOCK asyncBlock;
            asyncBlock = InterlockedExchange64( (PLONG64)&AsyncBlock[ i ],
                                                AsyncBlockFrameIsNotVisible );

            if ( asyncBlock != AsyncBlockFrameIsVisible &&
                 asyncBlock != AsyncBlockFrameIsNotVisible )
            {
                // Found a real D3DASYNCBLOCK for frame i, so we unblock the GPU
                pDevice->SignalAsyncResources( asyncBlock );
            }
        }
    }
}
```

Resolve tearing, therefore, is prevented by blocking the GPU until the previous frame's swap occurs before resolving the current frame.

Figure 15. Shows a timeline with all relevant events occurring to ensure that resolve tearing is prevented


Time
CPU calls InsertBlockOnAsyncResources
CPU calls Resolve() for Frame N+1
CPU calls Swap() for Frame N+1
GPU Stall starts
VBlank interval starts
VBlank callback executes
Swap occurs in VBlank callback for Frame N
SignalAsyncResources occurs in VBlank callback
GPU Stall ends
GPU calls Resolve() for Frame N+1
VBlank interval ends

Note in Figure 15 how the GPU is stalled up to the VBlank interval during which it is safe to call **Resolve** to overwrite the front buffer.

Table 2. Identifies the pros and cons of using an implementation using spin lock and block on asynchronous resources

	Spin-Lock	**Block on asynchronous resources**
Pros	• Easy to understand, implement, and debug.	• CPU not stalled. It is free to immediately start processing the next frame, and can recover more easily from a long frame.
Cons	• Stalls the CPU with a manual loop, essentially "processing" valuable CPU cycles as opposed to a system API such as **WaitForSingleObject**. • Stalls the CPU, which could cause the CPU to miss a VBlank interval, and essentially stall the GPU.	• More complex to implement and debug. • Extra latency could be introduced because the CPU is allowed to run farther ahead of the GPU. • Consumes at least one of the possible 64 pending slots of **InsertBlockOnAsyncResources**. Async command buffers could be used as an alternative.

Throttling

Direct3D does not throttle the CPU or the GPU when using asynchronous swaps. The GPU needs to be throttled to the VBlank intervals to avoid tearing as described in the previous section. CPU throttling can be implemented by the title in several ways, and ideally should occur just before updating a frame.

- Synchronize the CPU to the GPU by using **InsertFence** and **BlockOnFence**.
- **QuerySwapStatus** to spin lock on some previous frame's VBlank.
- Wait for a particular VBlank by using SetEvent in the VBlank callback. Refer to the Multirate Render Sample in the Xbox 360 Development Kit (XDK) for implementation details.

Useful Scenarios for Asynchronous Swaps

Now that how asynchronous swaps work has been explained, here are a few examples of where asynchronous swaps could be used:

- Games using video playback can use multiple front buffers to render multiple frames ahead of the current display device. For example, when using H.264 video compression, asynchronous swaps allow for very slow intra-coded (I) frames and very fast predicted (P) and bipredictive (B) frames. It also can help to smooth out video playback while streaming.

- Similar to video playback, games can also use multiple front buffers for playback of instant replays or in-engine cut-scenes.

- A game's main scene can be rendered at 30 Hertz (Hz), and the user interface can be rendered at 60 Hz. Refer to the Multirate Render Sample in the XDK.

- Games can more gracefully handle longer frames that slightly exceed the CPU or GPU budget. When this happens, the GPU can more easily recover because it does not need to stall waiting for a VBlank interval. Refer to the AsyncSwaps Sample in the XDK.

- Post-processing can be done on the CPU before the frame is presented. Precious GPU cycles could be traded for less precious CPU cycles. Some post-processing algorithms may be more efficient on the CPU. A good example would be a CPU implementation of a screen space antialiasing technique. While the CPU is still processing the antialiasing for frame N, the GPU can continue processing frame N+1.

- With special care, a game rendering at 30 Hz could do motion compensation to upsample to 60 Hz, increasing the perceived frame rate by the player.

Asynchronous Swaps and Kinect

Asynchronous swaps should not be used with Kinect titles. Kinect GPU processing is done during the execution of the **Present** function on the GPU timeline. The system relies on the fact that the GPU typically goes idle just before the VBlank that initiates the swap. With synchronous swaps, this timing is predictable, and data delivery from the Kinect sensor can be scheduled relative to this point to minimize Kinect processing latency. With asynchronous swaps, Kinect works, but the resulting variability of the timing of executing the **Present** function in the GPU's timeline can mean that frames from the Kinect sensor are dropped.

Furthermore, the difference between the start of a frame on the CPU and GPU can increase input latency.

Tips for Debugging Swaps and VBlanks

Use PIX timing captures to debug when and where swaps occur. PIX timing captures do not show where VBlank intervals occur, although the reserved XAM time intervals can be used as an indication. The best way to add extra debug information for swaps and VBlanks is to make use of the SetVerticalBlankCallback and SetSwapCallback functions. The VBlank callback is called at regular intervals—every 16.6 ms when the VBlank interval occurs. The swap callback is called after the swap command is executed on the GPU. Because these callback functions are DPCs, they should be handled with great care. Refer to the Kernel DPC Callbacks topic in the XDK. Listed below are some things that cannot be done within a DPC callback:

- Cannot set break points in Visual Studio
- Cannot use kernel blocking functions such as **WaitForSingleObject**
- Cannot use VMX or floating-point instructions

It is, however, safe to signal events within a DPC. Therefore, events can be signaled from the callback functions for a debug thread that can add PIX named events with valuable information. Refer to the Multirate Render Sample in the XDK for implementation details.

Conclusion

Synchronous swaps are well understood by developers. These swaps are easy and safe to use because Direct3D implements mechanisms for internally handling throttling and tearing prevention. To prevent tearing, the GPU is throttled to the VBlank intervals. Asynchronous swaps allow a title to render frames ahead of the current display without the need to stall the GPU until the VBlank interval. This can be useful for specific scenarios and allows a title to use the GPU more efficiently. Asynchronous swaps, however, should be used with great care. Without the proper knowledge and understanding of how to prevent tearing and how to implement throttling, it is easy to introduce visual artifacts, to add extra display latency, or to hang the GPU. Refer to the AsyncSwaps and Multirate Render samples in the XDK and the source code provided in this white paper for implementation details.