OpenVR - Virtual Display

Documentation for the IVRVirtualDisplay driver component interface

Aaron Leiby, June 2017

OVERVIEW & PURPOSE

The IVRVirtualDisplay interface is provided to allow OpenVR driver authors access to the final composited backbuffer intended for the headset's display. The primary expected use case is for wireless transport, though this could also be used for saving output to disk or streaming video. From the perspective of the runtime, the VR compositor is interfacing with a *virtual* rather than an *actual* display. This makes it a critical piece of the render pipeline and is imperative to get the timing correct in order to continue providing a proper experience to the user.

CONTENTS

API

IVRVirtualDisplay is expected to be implemented as a driver component. Driver components in OpenVR are returned by the GetComponent accessor of vr::ITrackedDeviceServerDriver and registered with the runtime via vr::VRServerDriverHost()->TrackedDeviceAdded. This function requires a unique string to identify the device (to persist identity across reconnects) and a device class. For IVRVirtualDisplay, use vr::TrackedDeviceClass_DisplayRedirect.

The latest OpenVR driver header can be found here:
https://github.com/ValveSoftware/openvr/blob/master/headers/openvr_driver.h

IVRVirtualDisplay requires implementation of the following functions:

```
void Present(vr::SharedTextureHandle_t backbufferTextureHandle)
```

This function will be called once a frame after all rendering has been submitted to the gpu. On Windows, the provided vr::SharedTextureHandle_t can be cast to a HANDLE and passed into your D3D device's OpenSharedResource interface to obtain a D3D texture interface (e.g. ID3D11Texture2D). These interfaces should be cached to avoid calling OpenSharedResource every frame. The handles passed to Present are currently double-buffered (i.e. they will alternate between two different handles every other frame), but you should not assume the number of buffers used. This texture represents the backbuffer that should be transmitted and displayed on the remove display.

To ensure proper gpu scheduling across processes, you should use QueryInterface on the texture interface to get its IDXGIKeyedMutex interface and wrap your usage in AcquireSync/ReleaseSync using zero (0) as your key.

Present should avoid blocking on any operations such as reading the texture, but instead queue up any required gpu commands and return as soon as possible. This will allow multiple IVRVirtualDisplay implementations to kick off in parallel. A separate interface, WaitForPresent, is provided for blocking until those operations complete.

`void` **`WaitForPresent()`**

As mentioned above, WaitForPresent is where the driver should block on the operations kicked off in Present. In the simplest case, it would wait until the frame started scanning out on the physical remote device. However, in practice wireless systems introduce some amount of latency. To ensure the compositor accounts for this latency in its predictions, the driver should set the property vr::Prop_SecondsFromVsyncToPhotons_Float. This is the amount of additional time (in seconds) that the driver is adding to the render pipeline. This value is currently only read on startup, and is not dynamic, so a general worse case value should be used. Once established, the driver should keep track of a virtual vsync that is offset by this amount relative to the physical device's vsync timings. Given actual vsyncs v_0 .. v_1 .. v_2 and additional latency a, the driver should keep track of virtual vsyncs $v_0 - a$ .. $v_1 - a$ .. $v_2 - a$. Then, a Present that happens between $v_i - a$ and $v_{i+1} - a$, WaitForPresent should block until $v_{i+1} - a$.

With that established, as an optimization step, rather than having the driver spin wait until the next virtual vsync, it can simply return early and reflect this in the values returned by GetTimeSinceLastVsync instead. Why then even have a WaitForPresent?

The other, and most important thing that WaitForPresent needs to do is to wait until rendering is complete. This obviously needs to happen before the frame can start scanning out, and in order to allow the application to start working on the next frame before the previous frame finishes (see "running start" - http://www.gdcvault.com/play/1021771/Advanced-VR) it's up to the display interface to handle this. Usually this is the job of the DXGI SwapChain (extended mode) or NvAPI/LiquidVR direct mode implementations. In the case of a VirtualDisplay, however, the responsibility falls on the display redirect driver.

Therefore, while it is okay for WaitForPresent to return before the virtual vsync, it still needs to block until we know the Presented frame has finished rendering. The easiest way to do this is to copy a pixel (e.g. CopySubresourceRegion) from the provided backbuffer to a staging texture that can be read from (e.g. D3D11_CPU_ACCESS_READ) using Map/Unmap. If this doesn't finish in the same virtual frame interval that the Present was intended for, then the next call to GetTimeSinceLastVsync should reflect this.

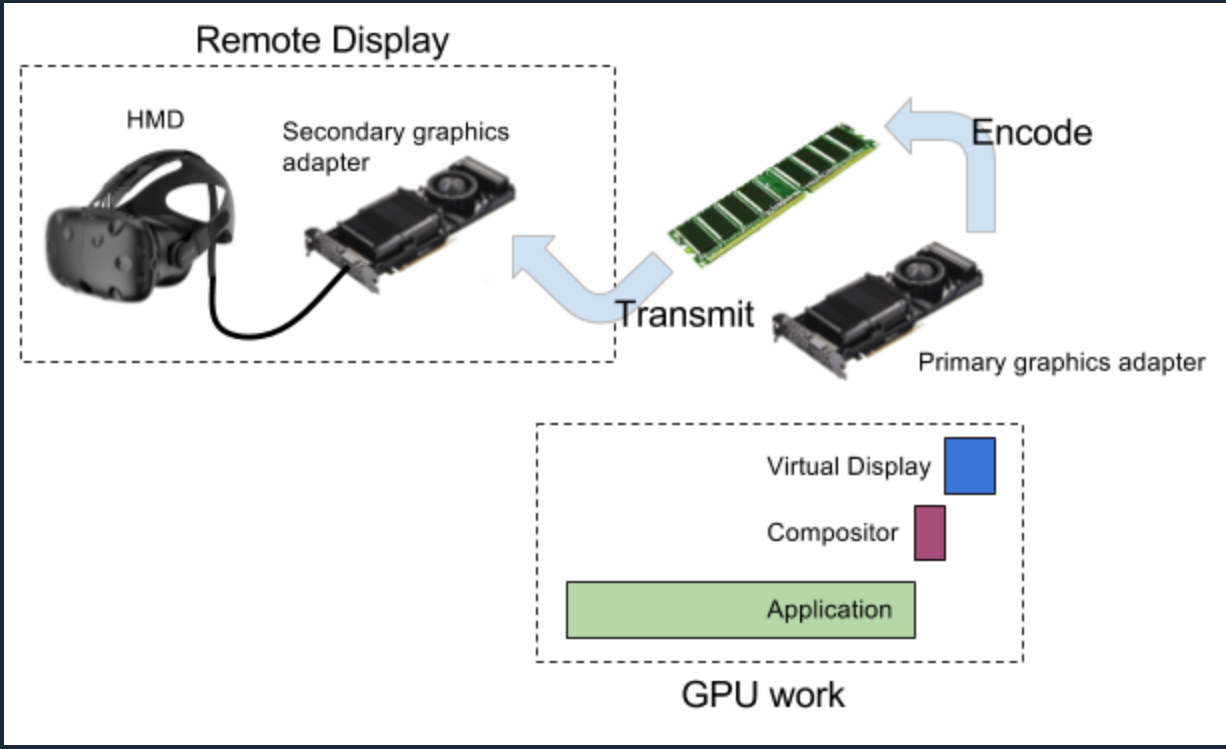
`bool GetTimeSinceLastVsync(float *pfSecondsSinceLastVsync, uint64_t *pulFrameCounter)`

This function will be called after WaitForPresent returns. It should return the time in seconds since the last *virtual* vsync event. Remember the *virtual* vsync event is offset from the physical device timing by the additional latency reported by the property vr::Prop_SecondsFromVsyncToPhotons_Float. The second parameter is a monotonically increasing value representing the frame count. This count should reflect the virtual vsync count rather than the number of frames presented in order for the runtime to detect dropped frames. For example, if a Present comes between frame v_i and v_{i+1}, the next call to GetTimeSinceLastVsync should return *i+1*. However, if the rendering of that frame does not finish until after v_{i+1}, it should return *i+2* (or whatever the next frame count after the rendering finishes that frame will start actually scanning out on).

EXAMPLE IMPLEMENTATION OVERVIEW

As discussed above, timing is critical to the proper implementation of the Virtual Display interface. An example implementation has been provided in order to help guide your own implementation. It can be found here: https://github.com/ValveSoftware/virtual_display

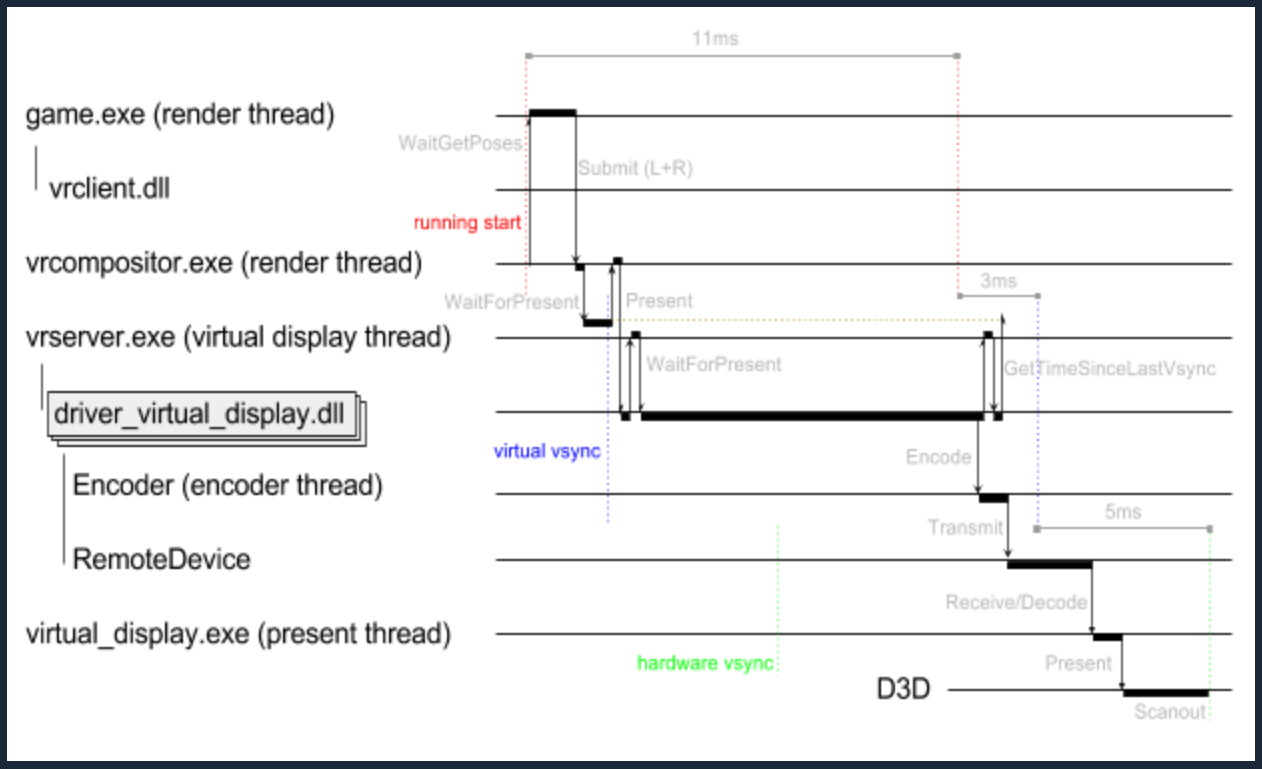
This example uses the headset (in Extended Mode) driven by a separate process (as opposed to the Compositor) to stand in for its "remote display". Since the separate process cannot interrupt the rendering of the application in order to guarantee the backbuffer is Presented on the appropriate vsync, we require it runs on a separate graphics adapter and copy the backbuffer through system memory. This simulates the encode/transmit/decode path you would find in a typical wireless setup, and is where the additional latency comes into play.



The application submits its rendering work to the primary graphics adapter for a frame, the compositor then takes the left and right images; composites them with any visible overlays, render models (e.g. controllers for interacting with the dashboard) and chaperone bounds; applies distortion, chromatic, and other screen-space corrections; then hands the final backbuffer image off to vrserver.

Device drivers are loaded by vrserver.exe. Our example driver is driver_virtual_display.dll. A thread in vrserver.exe is dedicated to responding to Virtual Display Present events from the compositor and dispatching to any implementations of IVRVirtualDisplay. These calls are multiplexed such that Present is called on each driver, followed by WaitForPresent on each, followed by GetTimeSinceLastVsync. Only the values of the first implementation of GetTimeSinceLastVsync that returns true will be used. If none return true, then we fire off a VsyncEvent internally at that time.

Here is an overview of the entire chain of events for a single frame:



A frame begins at what we refer to as “running start”. This is typically 3ms prior to vsync. In the Virtual Display case, it is 3ms prior to the virtual vsync. This is signaled by IVRCompositor’s WaitGetPoses function returning control to the game’s render thread.

A game typically takes 1-5ms to submit its rendering work for the frame to the gpu. When it is finished it calls IVRCompositor’s Submit function for both the left and right render targets. Since this may happen before the previous frame starts scanning out, the compositor will WaitForPresent. The bar for this above is illustrated as partway between vrcompositor.exe and vrserver.exe as it is an inter-process mutex that vrserver signals at the end of the frame (follow the yellow dotted line).

Once we know we are on the other side of vsync, we call Present. Normally, this would be serviced by a DXGI SwapChain (in Extended Mode) or an IHV-specific Direct Mode interface (i.e. NvAPI or LiquidVR). In the Virtual Display case, this gets handed off to vrserver.exe to dispatch to the drivers that implement IVRVirtualDisplay to handle (explained above).

The majority of the time is spent in WaitForPresent. This blocks waiting for the gpu to finish rendering the frame at which point the backbuffer can be copied into shared memory (our example encode/transmit step). This is handled by a separate thread in the driver so vrserver's virtual display thread which calls into each driver can respond to the next frame.

In our example, our Encoder doesn't actually do any encoding, instead it simply hands off the bits to our RemoteDevice object. The RemoteDevice object is our client interface to our remote display; it manages the lifespan of virtual_display.exe, sets up shared memory to transfer the backbuffer texture data, and signals the separate process when new data is available.

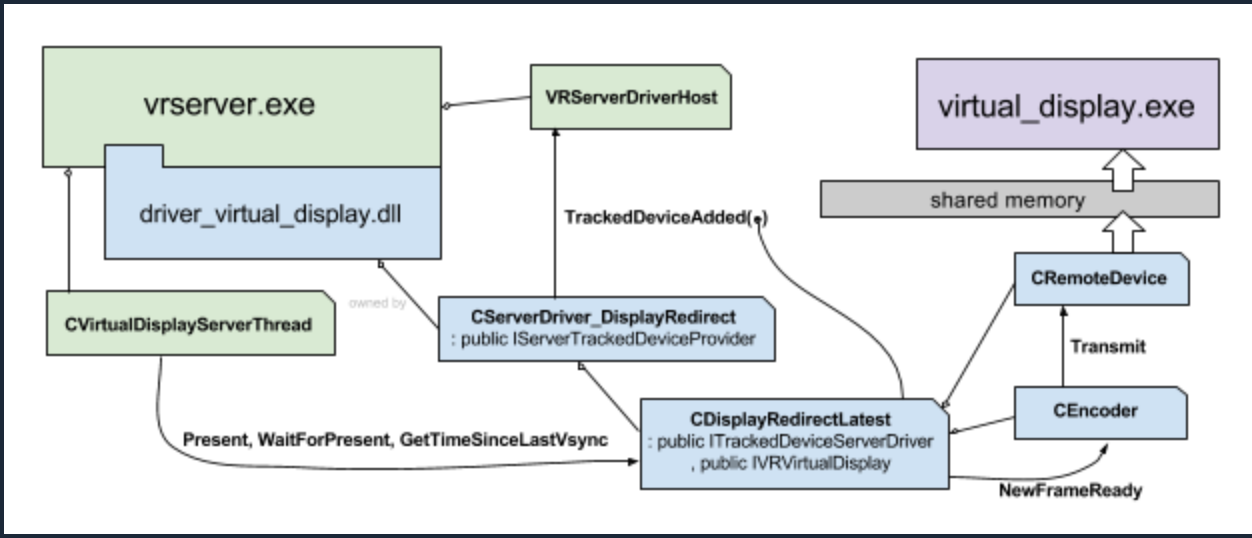
Finally, our separate process creates a fullscreen window on the headset and waits to be signaled by driver_virtual_display.dll that new data is available. When signaled, it copies that data out of shared memory, uploads it to its separate graphics adapter, and Presents it to its fullscreen window to start scanning out on the next actual vsync interval. It then waits for scanout to start and queries for vsync timing data to communicate back to driver_virtual_display.dll (not shown). This last bit is critical to ensuring the poses used to render each frame are predicted properly in order to minimize swim.

One important observation is that GetTimeSinceLastVsync is reporting the timing of the virtual vsync for which its hardware vsync hasn't happened yet. As such, the timing will always be behind a frame. This offset is treated as static to avoid any hysteresis; changes to SecondsFromVsyncToPhotons after initialization are ignored.

This offset between virtual vsync and hardware vsync (the 5ms interval in the illustration above) is needed to cover the additional latency added by the encode/transmit/receive/decode steps which are not present in cases where the compositor interfaces with the headset display directly. In our example this is specified in settings: additionalLatencyInSeconds. Drivers can provide their own default settings which can then be overridden in steamvr.vrsettings in the Steam/config folder.

EXAMPLE IMPLEMENTATION DETAILS

This section will delve into the implementation details of the example driver.



All OpenVR device drivers share a common entry point: HmdDriverFactory. Here we return a global instance of our IServerTrackedDeviceProvider implementation CServerDriver_DisplayRedirect.

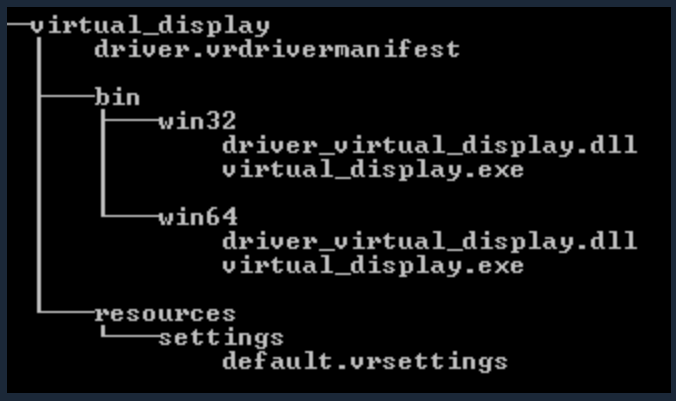
```
CServerDriver_DisplayRedirect g_serverDriverDisplayRedirect;

//-----------------------------------------------------------------------------
// Purpose: Entry point for vrserver when loading drivers.
//-----------------------------------------------------------------------------
extern "C" __declspec( dllexport )
void *HmdDriverFactory( const char *pInterfaceName, int *pReturnCode )
{
    if ( 0 == strcmp( vr::IServerTrackedDeviceProvider_Version, pInterfaceName ) )
    {
        return &g_serverDriverDisplayRedirect;
    }

    if( pReturnCode )
        *pReturnCode = vr::VRInitError_Init_InterfaceNotFound;

    return NULL;
}
```

For our purposes we only really need this for calls to Init and Cleanup (and versioning).

```
class CServerDriver_DisplayRedirect : public vr::IServerTrackedDeviceProvider
{
public:
    CServerDriver_DisplayRedirect()
        : m_pDisplayRedirectLatest( NULL )
    {}

    virtual vr::EVRInitError Init( vr::IVRDriverContext *pContext ) override;
    virtual void Cleanup() override;
    virtual const char * const *GetInterfaceVersions() override
        { return vr::k_InterfaceVersions;  }
    virtual const char *GetTrackedDeviceDriverVersion()
        { return vr::ITrackedDeviceServerDriver_Version; }
    virtual void RunFrame() override {}
    virtual bool ShouldBlockStandbyMode() override { return false; }
    virtual void EnterStandby() override {}
    virtual void LeaveStandby() override {}

private:
    CDisplayRedirectLatest *m_pDisplayRedirectLatest;
};

vr::EVRInitError CServerDriver_DisplayRedirect::Init( vr::IVRDriverContext *pContext )
{
    VR_INIT_SERVER_DRIVER_CONTEXT( pContext );

    m_pDisplayRedirectLatest = new CDisplayRedirectLatest();

    if ( m_pDisplayRedirectLatest->IsValid() )
    {
        vr::VRServerDriverHost()->TrackedDeviceAdded(
            m_pDisplayRedirectLatest->GetSerialNumber().c_str(),
            vr::TrackedDeviceClass_DisplayRedirect,
            m_pDisplayRedirectLatest );
    }

    return vr::VRInitError_None;
}

void CServerDriver_DisplayRedirect::Cleanup()
{
    delete m_pDisplayRedirectLatest;
    m_pDisplayRedirectLatest = NULL;

    VR_CLEANUP_SERVER_DRIVER_CONTEXT();
}
```

Init creates our tracked device object (CDisplayRedirectLatest) which implements the IVRVirtualDisplay component interface. However, it only registers it if valid. TrackedDeviceClass_DisplayRedirect is a new special type of tracked device. It doesn't actually have a pose associated with it like an hmd or controller, though there's nothing stopping you from providing one if you know the location of your device. Remember your device is the wireless transport (e.g. antenna) not the headset it is communicating with; the headset itself has its own driver (e.g. driver_lighthouse.dll). It's important to only register a DisplayRedirect device if you have successfully connected to the headset, because the presence of an active DisplayRedirect device causes the runtime to skip looking for the headset display and instead trusts that the DisplayRedirect driver will successfully deliver the backbuffer images handed to it to the actual display. We want to avoid the situation where simply installing a DisplayRedirect driver results in the headset *only* then working when connected through the associated device, and stops working when connected directly to the computer.

On disk, our example driver lives in a folder with the following layout:



This folder must be placed in one of the registered OpenVR driver paths. Use **vrpathreg** to determine the current paths, or register a new one for your driver. It can be found in your SteamVR install here:

```
<Steam install>/steamapps/common/SteamVR/bin/win32/vrpathreg.exe

adddriver <path>

removedriver <path>
```

You can call VR_RuntimePath() in your installer to get to the root of the SteamVR runtime.

Each driver provides a manifest to specify some meta-data. Here is our example driver's manifest.

```
{
    "alwaysActivate": true,
    "name" : "virtual_display",
    "redirectsDisplay": true,

    "hmd_presence" :
    [
        "*.*"
    ]
}
```

This is in JSON format, and no error checking is performed, so watch your trailing commas and use https://jsonlint.com/ for sanity checking.

Normally, SteamVR will stop activating drivers once it finds one that registers a headset. Setting *alwaysActivate* to true will allow your driver to activate in parallel to the headset's driver.

Setting *redirectsDisplay* to true tell the runtime to test your driver before those set to false. This allows the actual headset driver to skip failing because it cannot find its display physically attached to the computer by checking to see if any valid DisplayRedirect devices have been activated first.

Finally, *hmd_presence* is where you can specify the USB VID.PID for the devices you support so SteamVR will only initialize your driver if it first detects the hardware is present. For our example we use wildcards for both VID and PID to skip this check. Multiple entries can be added here separated by commas.

Drivers can also supply their own settings and defaults. Here is our example driver's default.vrsettings:

```
{
    "driver_virtual_display": {
        "serialNumber": "VD-001",
        "modelNumber": "Virtual Display",
        "additionalLatencyInSeconds": 0.008,
        "displayWidth": 2160,
        "displayHeight": 1200,
        "displayRefreshRateNumerator": 90,
        "displayRefreshRateDenominator": 1,
        "adapterIndex": -1
    }
}
```

Again this is in JSON format, and no error checking is performed.

These values can be accessed in code using the IVRSetting interface which CDisplayRedirectLatest's constructor does to find the display and initialize its subsystems.

```
vr::VRSettings()->GetString( k_pch_VirtualDisplay_Section,
    k_pch_Null_SerialNumber_String, m_rchSerialNumber, ARRAYSIZE( m_rchSerialNumber ) );
vr::VRSettings()->GetString( k_pch_VirtualDisplay_Section,
    k_pch_Null_ModelNumber_String, m_rchModelNumber, ARRAYSIZE( m_rchModelNumber ) );

m_flAdditionalLatencyInSeconds = max( 0.0f,
    vr::VRSettings()->GetFloat( k_pch_VirtualDisplay_Section,
        k_pch_VirtualDisplay_AdditionalLatencyInSeconds_Float ) );

int32_t nDisplayWidth = vr::VRSettings()->GetInt32(
    k_pch_VirtualDisplay_Section,
    k_pch_VirtualDisplay_DisplayWidth_Int32 );
int32_t nDisplayHeight = vr::VRSettings()->GetInt32(
    k_pch_VirtualDisplay_Section,
    k_pch_VirtualDisplay_DisplayHeight_Int32 );

int32_t nDisplayRefreshRateNumerator = vr::VRSettings()->GetInt32(
    k_pch_VirtualDisplay_Section,
    k_pch_VirtualDisplay_DisplayRefreshRateNumerator_Int32 );
int32_t nDisplayRefreshRateDenominator = vr::VRSettings()->GetInt32(
    k_pch_VirtualDisplay_Section,
    k_pch_VirtualDisplay_DisplayRefreshRateDenominator_Int32 );

int32_t nAdapterIndex = vr::VRSettings()->GetInt32(
    k_pch_VirtualDisplay_Section,
    k_pch_VirtualDisplay_AdapterIndex_Int32 );
```

Since we cannot interrupt application rendering in order to latently present the previous frame to the headset, this example simply requires a second graphics card to act as the remote display, and copies through system memory to emulate the wireless transfer.

The *adapterIndex* setting allows us to specify which graphics adapter to use as SteamVR's primary graphics card for rendering. By default, we choose the first adapter that the headset isn't plugged into. These are printed to the log for debugging purposes.

After that, it creates a CRemoteDevice which launches virtual_display.exe, and finally constructs the CEncoder which spins up a separate thread for handling our overlapping transmission of the frame data.

```
// Spawn our separate process to manage headset presentation.
m_pRemoteDevice = new CRemoteDevice();
if ( !m_pRemoteDevice->Initialize(
    nDisplayX, nDisplayY, nDisplayWidth, nDisplayHeight,
    nDisplayRefreshRateNumerator, nDisplayRefreshRateDenominator ) )
{
    return;
}

// Spin up a separate thread to handle the overlapped encoding/transmit step.
m_pEncoder = new CEncoder( m_pD3DRender, m_pRemoteDevice );
m_pEncoder->Start();
```

When registered as a tracked device, the runtime will assign our CDisplayRedirectLatest object an ObjectId via our overridden Activate function. We can in turn use this to set some properties.

```
virtual vr::EVRInitError Activate( uint32_t unObjectId ) override
{
    m_unObjectId = unObjectId;

    vr::PropertyContainerHandle_t ulContainer =
        vr::VRProperties()->TrackedDeviceToPropertyContainer( unObjectId );

    vr::VRProperties()->SetStringProperty( ulContainer,
        vr::Prop_ModelNumber_String, m_rchModelNumber );
    vr::VRProperties()->SetFloatProperty( ulContainer,
        vr::Prop_SecondsFromVsyncToPhotons_Float, m_flAdditionalLatencyInSeconds );
    vr::VRProperties()->SetUint64Property( ulContainer,
        vr::Prop_GraphicsAdapterLuid_Uint64, m_nGraphicsAdapterLuid );

    return vr::VRInitError_None;
}
```

Here we specify our model number, but more importantly is where we tell the runtime how much additional latency we are introducing into the render pipeline. This uses the existing property Prop_SecondsFromVsyncToPhotons_Float, but because it is associated with a DisplayRedirect device, the system knows to add this additional latency as opposed to using it directly. This is also where we specify which graphics adapter the runtime should use.

Another important aspect is to register our device's IVRVirtualDisplay interface as a driver component. This is performed by overriding ITrackedDeviceServerDriver's GetComponent function.

```
virtual void *GetComponent( const char *pchComponentNameAndVersion ) override
{
    if ( !_stricmp( pchComponentNameAndVersion, vr::IVRVirtualDisplay_Version ) )
    {
        return static_cast< vr::IVRVirtualDisplay * >( this );
    }
    return NULL;
}
```

Each frame our IVRVirtualDisplay object will receive a Present call with a shared handle to the backbuffer surface. A finite number of buffers are used, so these should be cached to avoid repeatedly opening the same handle.

```
virtual void Present( vr::SharedTextureHandle_t backbufferTextureHandle ) override
{
    // Open and cache our shared textures to avoid re-opening every frame.
    ID3D11Texture2D *pTexture = m_pD3DRender->GetSharedTexture( ( HANDLE )backbufferTextureHandle );
```

```
ID3D11Texture2D *CD3DRender::GetSharedTexture( HANDLE hSharedTexture )
{
    if ( !hSharedTexture )
        return NULL;

    for ( SharedTextures_t::iterator it = m_SharedTextureCache.begin();
        it != m_SharedTextureCache.end(); ++it )
    {
        if ( it->m_hSharedTexture == hSharedTexture )
        {
            return it->m_pTexture;
        }
    }

    ID3D11Texture2D *pTexture;
    if ( SUCCEEDED( m_pD3D11Device->OpenSharedResource(
        hSharedTexture, __uuidof( ID3D11Texture2D ), ( void ** )&pTexture ) ) )
    {
        SharedTextureEntry_t entry { hSharedTexture, pTexture };
        m_SharedTextureCache.push_back( entry );
        return pTexture;
    }

    return NULL;
}
```

Also, for this to work we must use DXGI 1.1 by calling CreateDXGIFactory1 (note the 1).

```
// Need to use DXGI 1.1 for shared texture support.
IDXGIFactory1 *pDXGIFactory1;
if ( FAILED( CreateDXGIFactory1( __uuidof( IDXGIFactory1 ), ( void ** )&pDXGIFactory1 ) ) )
{
    Log( "Failed to create DXGIFactory1!\n" );
    return false;
}
else if ( FAILED( pDXGIFactory1->QueryInterface( __uuidof( IDXGIFactory ), ( void ** )&m_pDXGIFactory ) ) )
{
    pDXGIFactory1->Release();
    Log( "Failed to get DXGIFactory interface!\n" );
    return false;
}
pDXGIFactory1->Release();
```

Access to the shared texture must be wrapped in AcquireSync/ReleaseSync to ensure the compositor has finished rendering to it before it gets used. This enforces scheduling of work on the gpu between processes.

```
IDXGIKeyedMutex *pKeyedMutex = NULL;
if ( SUCCEEDED( pTexture->QueryInterface( __uuidof( IDXGIKeyedMutex ), ( void ** )&pKeyedMutex ) ) )
{
    if ( pKeyedMutex->AcquireSync( 0, 10 ) != S_OK )
    {
        pKeyedMutex->Release();
        return;
    }
}
```

We use an arbitrary 10ms timeout here (less than a full frame), and then bail if the Acquire fails for any reason.

Once we have access to the texture, we make two copies. The first copy is a single pixel that we can read back to know when all rendering is finished.

```
if ( m_pFlushTexture == NULL )
{
    D3D11_TEXTURE2D_DESC srcDesc;
    pTexture->GetDesc( &srcDesc );

    // Create a second small texture for copying and reading a single pixel from
    // in order to block on the cpu until rendering is finished.
    D3D11_TEXTURE2D_DESC flushTextureDesc;
    ZeroMemory( &flushTextureDesc, sizeof( flushTextureDesc ) );
    flushTextureDesc.Width = 32;
    flushTextureDesc.Height = 32;
    flushTextureDesc.MipLevels = 1;
    flushTextureDesc.ArraySize = 1;
    flushTextureDesc.Format = srcDesc.Format;
    flushTextureDesc.SampleDesc.Count = 1;
    flushTextureDesc.Usage = D3D11_USAGE_STAGING;
    flushTextureDesc.BindFlags = 0;
    flushTextureDesc.CPUAccessFlags = D3D11_CPU_ACCESS_READ;

    if ( FAILED( m_pD3DRender->GetDevice()->CreateTexture2D( &flushTextureDesc, NULL, &m_pFlushTexture ) ) )
    {
        DriverLog( "Failed to create flush texture!\n" );
        return;
    }
}

if ( m_pFlushTexture )
{
    // Copy a single pixel so we can block until rendering is finished in WaitForPresent.
    D3D11_BOX box = { 0, 0, 0, 1, 1, 1 };
    m_pD3DRender->GetContext()->CopySubresourceRegion( m_pFlushTexture, 0, 0, 0, 0, pTexture, 0, &box );

    // This can go away, but is useful to see it as a separate packet on the gpu in traces.
    m_pD3DRender->GetContext()->Flush();
}
```

The second is to queue up transferring the entire backbuffer into a readable buffer, followed by releasing access to the shared texture before we return.

```
// Copy entire texture to staging so we can read the pixels to send to remote device.
m_pEncoder->CopyToStaging( pTexture );
m_pD3DRender->GetContext()->Flush();

if ( pKeyedMutex )
{
    pKeyedMutex->ReleaseSync( 0 );
    pKeyedMutex->Release();
}
```

```
bool CopyToStaging( ID3D11Texture2D *pTexture )
{
    // Create a staging texture to copy frame data into that can in turn
    // be read back (for blocking until rendering is finished).
    if ( m_pStagingTexture == NULL )
    {
        D3D11_TEXTURE2D_DESC srcDesc;
        pTexture->GetDesc( &srcDesc );

        D3D11_TEXTURE2D_DESC stagingTextureDesc;
        ZeroMemory( &stagingTextureDesc, sizeof( stagingTextureDesc ) );
        stagingTextureDesc.Width = srcDesc.Width;
        stagingTextureDesc.Height = srcDesc.Height;
        stagingTextureDesc.Format = srcDesc.Format;
        stagingTextureDesc.MipLevels = 1;
        stagingTextureDesc.ArraySize = 1;
        stagingTextureDesc.SampleDesc.Count = 1;
        stagingTextureDesc.Usage = D3D11_USAGE_STAGING;
        stagingTextureDesc.CPUAccessFlags = D3D11_CPU_ACCESS_READ;

        if ( FAILED( m_pD3DRender->GetDevice()->CreateTexture2D( &stagingTextureDesc, NULL, &m_pStagingTexture ) ) )
        {
            DriverLog( "Failed to create staging texture!\n" );
            return false;
        }
    }

    m_pD3DRender->GetContext()->CopyResource( m_pStagingTexture, pTexture );

    return true;
}
```

Recall that multiple IVRVirtualDisplay devices may be active simultaneously, so we want to return from Present as quickly as possible so other instances can fire off their work as well.

Then in WaitForPresent, we start by blocking on that first pixel readback.

```
virtual void WaitForPresent() override
{
    // First wait for rendering to finish on the gpu.
    if ( m_pFlushTexture )
    {
        D3D11_MAPPED_SUBRESOURCE mapped = { 0 };
        if ( SUCCEEDED( m_pD3DRender->GetContext()->Map( m_pFlushTexture, 0, D3D11_MAP_READ, 0, &mapped ) ) )
        {
            m_pD3DRender->GetContext()->Unmap( m_pFlushTexture, 0 );
        }
    }
```

Note, it's tempting to use a QueryEvent here and poll at 1ms intervals to avoid having the driver spin wait until the data is ready. Unfortunately, we need more accuracy than that or we'll wind up reporting dropped frames when there was still plenty of time left in the frame.

This is also where it gets a bit tricky, even though the logic itself is pretty straightforward. WaitForPresent *should* wait until the backbuffer submitted to Present starts scanning out. This doesn't *actually* happen until the *real* vsync on the headset, but waiting until that point would

introduce too much latency into the system. Instead, we trust the reliability and consistency of the components of the render pipeline beyond this point (encode, transmit and display) to finish with the specified AdditionalLatency timeframe. This then implies that WaitForPresent should return at the *virtual* vsync (real vsync minus AdditionalLatency), but we can go even one better. Since we are relying on the remaining portion of the render pipeline to finish within AdditionalLatency seconds, the only variable portion will be the application plus compositor gpu work, and we only need to know which *virtual* frame interval that landed in. The remainder of the logic in WaitForPresent calculates this (for returning in GetTimeSinceLastVsync), and then returns *early*. This means that the value returned by GetTimeSinceLastVsync is usually negative (i.e. in the future).

The first thing we do after waiting for rendering to finish is to tell the encoder thread to start its work on the frame (the copy we made to staging in Present). The encoder uses the shared D3D context which is not thread-safe, so it's important to wait until we are done before allowing it to continue.

```
// Now that we know rendering is done, we can fire off our thread that reads the
// backbuffer into system memory.  We also pass in the earliest time that this frame
// should get presented.  This is the real vsync that starts our frame.
m_pEncoder->NewFrameReady( m_flLastVsyncTimeInSeconds + m_flAdditionalLatencyInSeconds );
```

You will notice that we pass in a time here as well. This is the earliest that we want the remote device to present the frame. This will ensure that it gets scanned out on the correct *real* vsync interval with the displays illuminating at the expected time (matching the values used to predict the poses that were used for rendering and reprojection).

Note that we store a common base ticks between processes in shared memory in order for system timings to be comparable.

Next, we sync up with the remote device timing. This isn't necessarily the timing of the last presented frame depending on the amount of AdditionalLatency or when Present is called (which in turn can be affected by the current scene application's performance), but it's the latest timing info we have, and therefore the most accurate.

```
// Get latest timing info to work with.  This gets us sync'd up with the hardware in
// the first place, and also avoids any drifting over time.
double flLastVsyncTimeInSeconds;
uint32_t nVsyncCounter;
m_pRemoteDevice->GetTimingInfo( &flLastVsyncTimeInSeconds, &nVsyncCounter );

// Account for encoder/transmit latency.
// This is where the conversion from real to virtual vsync happens.
flLastVsyncTimeInSeconds -= m_flAdditionalLatencyInSeconds;
```

With that info, we can account for any drift by counting the number of frames between our last vsync timing and this new reference vsync timing, and apply that delta to the new reference vsync timing.

```
float flFrameIntervalInSeconds = m_pRemoteDevice->GetFrameIntervalInSeconds();

// Realign our last time interval given updated timing reference.
int32_t nTimeRefToLastVsyncFrames =
    ( int32_t )roundf( float( m_flLastVsyncTimeInSeconds - flLastVsyncTimeInSeconds ) / flFrameIntervalInSeconds );
m_flLastVsyncTimeInSeconds = flLastVsyncTimeInSeconds + flFrameIntervalInSeconds * nTimeRefToLastVsyncFrames;

// We could probably just use this instead, but it seems safer to go off the system timer calculation.
assert( m_nVsyncCounter == nVsyncCounter + nTimeRefToLastVsyncFrames );
```

With this, we can then determine the very next *virtual* vsync interval. Remember that GetTimeSinceLastVsync will normally return negative values since we are cheating and returning early. This means that when WaitForPresent gets called, it's possible for *last* vsync to still be in the future. Since we've already queued up a frame for that vsync interval, we clamp the *last* to *next* frame count to zero. This is the max(nLastVsyncToNextVsyncFrames, 0) seen in the following code.

```
double flNow = SystemTime::GetInSeconds();

// Find the next frame interval (keeping in mind we may get here during running start).
int32_t nLastVsyncToNextVsyncFrames =
    ( int32_t )( float( flNow - m_flLastVsyncTimeInSeconds ) / flFrameIntervalInSeconds );
nLastVsyncToNextVsyncFrames = max( nLastVsyncToNextVsyncFrames, 0 ) + 1;

// And store it for use in GetTimeSinceLastVsync (below) and updating our next frame.
m_flLastVsyncTimeInSeconds += flFrameIntervalInSeconds * nLastVsyncToNextVsyncFrames;
m_nVsyncCounter = nVsyncCounter + nTimeRefToLastVsyncFrames + nLastVsyncToNextVsyncFrames;
```

This just leaves us with GetTimeSinceLastVsync itself, which is straightforward.

```
virtual bool GetTimeSinceLastVsync( float *pfSecondsSinceLastVsync, uint64_t *pulFrameCounter ) override
{
    *pfSecondsSinceLastVsync = ( float )( SystemTime::GetInSeconds() - m_flLastVsyncTimeInSeconds );
    *pulFrameCounter = m_nVsyncCounter;
    return true;
}
```

We store an absolute time, but this interface takes a relative time. This is to avoid having to deal with synchronizing timebases between the runtime and drivers.

And that's it for the important bits. The encoder simply maps the copy of the texture we made in Present and hands it off to the remote device to "transmit", which in turn just performs a memcopy into shared memory and signals our helper process. The helper process then copies the data out, uploads it to the gpu (recall it is using a different graphics adapter to avoid interfering with the rest of the processes) waits until the proper vsync interval, then presents to the headset for scanout and display. It does have one last important role to play, however, and that is communicating vsync timing data back to the driver. It does this by calling GetFrameStatistics on the swap chain until the PresentCount matches, indicating the frame has started scanning out, and then uses that data (SyncQPCTime and SyncRefreshCount) to update shared memory. Actual hardware devices will have to source this through some other method.

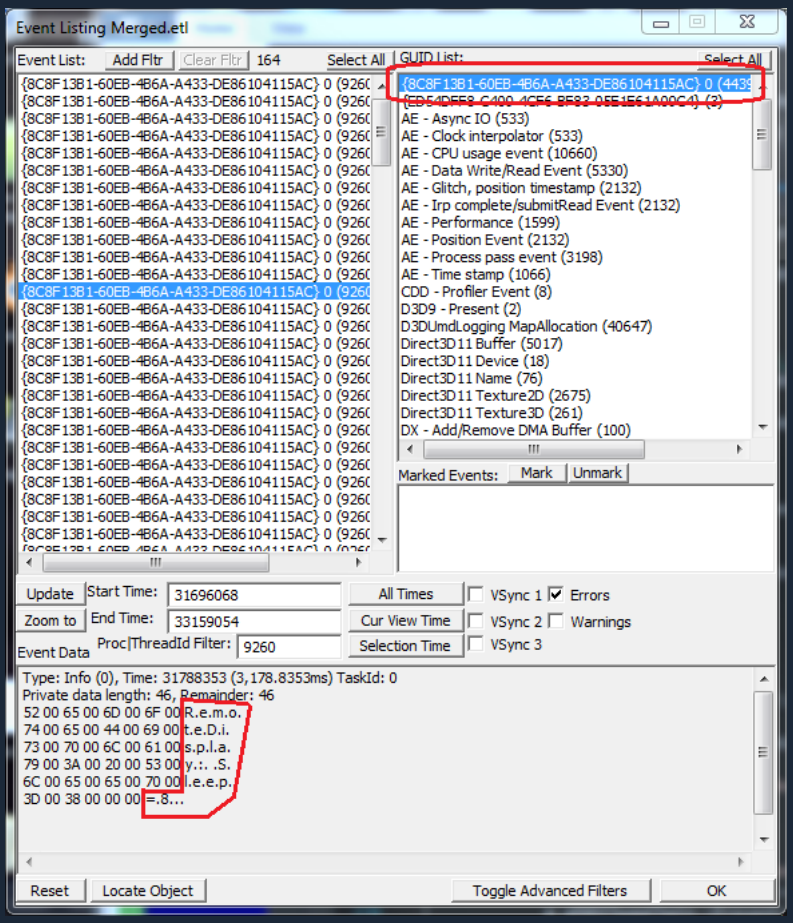
EXAMPLE IMPLEMENTATION EXECUTION

This section will take a look at the example driver in action to get a better feel for the flow of events on both the cpu and gpu, and how frames overlap one to the next.

For this, we will use gpuview to capture and analyze activity across the entire system. The OpenVR runtime injects various events which helps to annotate its workings. In the example driver implementation, these are calls to EventWriteString.

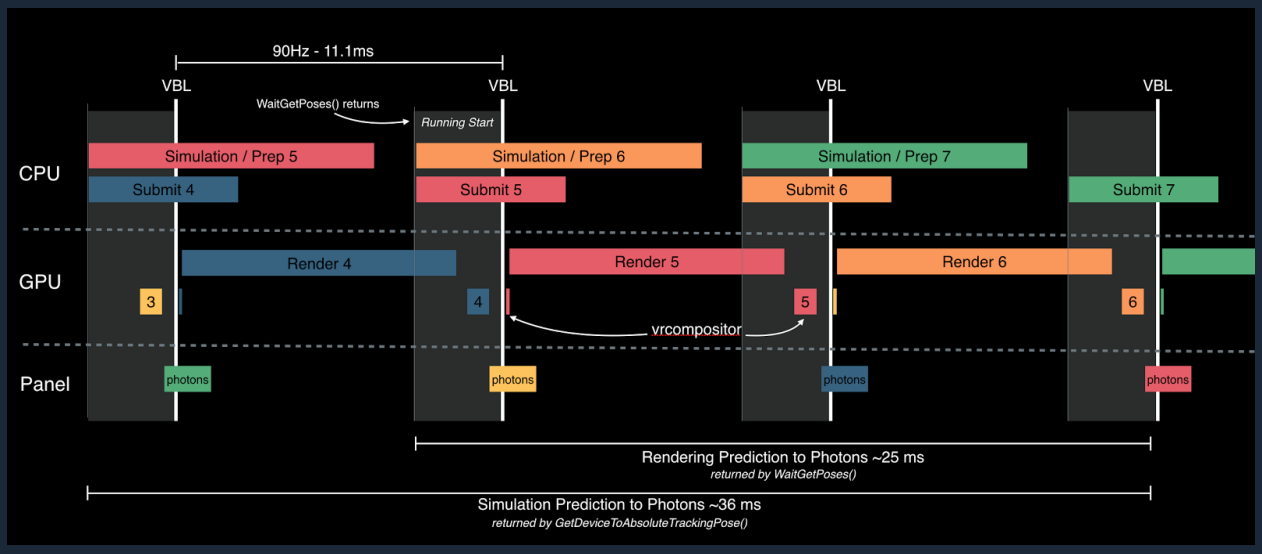
```
wchar_t buffer[ 255 ];
swprintf( buffer, ARRAYSIZE( buffer ), L"RemoteDisplay: Sleep=%d", nSleepMs );
EventWriteString( buffer );

Sleep( nSleepMs );

EventWriteString( L"RemoteDisplay: SleepEnd" );
```

These all use a single GUID and show up in gpuview's event list like so:



To install gpuview and enable capturing of the OpenVR events, refer to the instructions here: https://developer.valvesoftware.com/wiki/SteamVR/Installing_GPUView

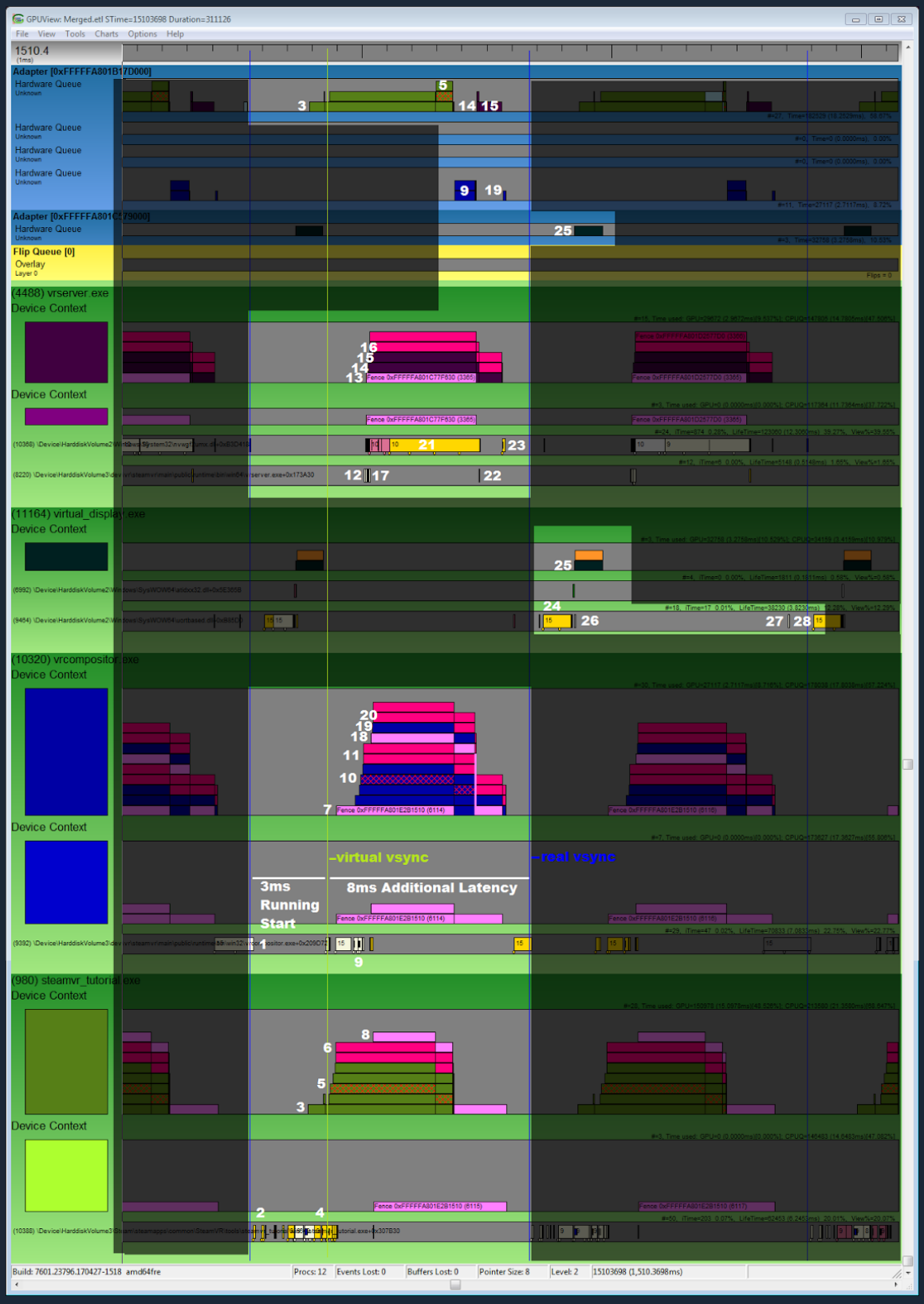
As a quick refresher to start, here is an overview of the normal operation of things:



Following the red frame through, we start on the left with the simulation of the frame and preparation for rendering (e.g. building up a command list of render operations). The application then waits (usually in a separate render thread so it can get started on the next frame's simulation in parallel) for WaitGetPoses to return new poses at the beginning of running start. Running Start is a compromise between giving the application time to submit enough work to get the gpu started working on the frame while not introducing too much error into the predicted poses. Once the application has the poses to render the frame, it can update its constant buffers and submit the work to the gpu. It's good to remember that these poses aren't polling the current pose, but being predicted ahead to when the photons leave the panel (for this frame on that last vblank line to the far right). The gpu spends most of the frame interval rendering the scene followed by the compositor warp right at the end. This needs to finish before the vblank that ends the frame interval in order to start scanning out at that time. The displays in the current generation of headsets are globally illuminated, and it takes nearly a full frame interval from the time rendering is completed to copy across the hdmi cable and loaded into the display buffers before the displays are flashed on for 1-2 milliseconds, and the process repeats.

As discussed earlier, a virtual display introduces additional latency into the pipeline. This requires the additional of a *virtual* vblank (a.k.a. vsync) which is offset from the *real* vblank by that amount.

Here’s an example capture of an application running in framerate using a virtual display:



Let's walk through each step for a single frame:

1. Running Start - Compositor predicts new poses and signals application.
2. WaitGetPoses returns, application begins submitting work to gpu/driver.
3. Works starts showing up on gpu. If this starts too late, it can often by fixed by adding Flushes to the application renderer in key places.
4. Application calls Submit providing the left and right scene textures to the Compositor, which lets it know it's safe to start submitting its own gpu work for the frame.
5. Red hatched blocks like this represent the application calling Present. This is so it can display its own "companion" window on the desktop for spectators. The work following this block is usually very small as most applications simply draw a single quad using one of the eye textures.
6. Application calls PostPresentHandoff after calling Present on their window. Applications that have a separate render thread can usually get away with just blocking on WaitGetPoses at this point (which then calls PostPresentHandoff internally). The dark pink boxes that show up here correspond to ReleaseSync being called on the mutex used to synchronize the render work between the application and compositor to enforce scheduling order across processes.
7. Compositor calls AcquireSync on the shared mutex to bookmark its render work for the frame.
8. Application (inside PostPresentHandoff) calls AcquireSync on the shared mutex to bookmark its render work for the next frame.
9. Compositor submits render work to the gpu/driver.
10. Red hatched blocks in the compositor process represent Present being called on its mirror window. We only update the mirror window every other frame, so you'll only see this show up accordingly (and only if the mirror window is shown).

So far none of this is any different than working with a real display except the fact that all the timing is centered around the virtual vsync rather than the real vsync. One potential point of confusion: Since we are using 8ms of Additional Latency in our example, running start just happens to line up with the previous frame's real vsync (8+3=11ms). This is just coincidental. Reducing AdditionalLatency (which can be overridden in your steamvr.vrsettings file) will pull all of this to the right, closer to real vsync.

Continuing on with the steps specific to Virtual Display drivers:

11. Compositor calls ReleaseSync on backbuffer shared mutex and hands off to vrserver.
12. Vrserver calls IVRVirtualDisplay::Present on registered driver components.
13. Driver calls AcquireSync on provided backbuffer's shared mutex to bookmark its access to that texture.
14. Driver copies a single pixel from provided backbuffer to a texture it can use to block on to determine when rendering is finished.
15. Driver copies entire backbuffer to a staging texture it can read into system memory. Note: This is really just queuing up a copy command on the gpu at this point.
16. Driver calls ReleaseSync on shared mutex and returns from Present. Elapsed cpu time less than 0.5ms.
17. Vrserver signals Compositor that Virtual Display(s) are finished with backbuffer texture.
18. Compositor calls AcquireSync on shared mutex.
19. Compositor copies single pixel from backbuffer.
20. Compositor calls ReleaseSync on shared mutex. These three steps result in the blue compositor work wrapping the driver's work (purple) in the vrserver process. This allows the compositor to time this work to provide feedback to applications so they can dynamically adjust their fidelity to account for any additional overhead the driver is adding.
21. Meanwhile, the driver is spinning in IVRVirtualDisplay::WaitForPresent as it waits for rendering to complete by calling Map on the "flush" texture it copied a single pixel to in Present.
22. Driver determines if we made framerate or not (finished before virtual vsync), updates its timing info and returns that data in IVRVirtualDisplay::GetTimeSinceLastVsync. Note that it did not actually block until that virtual vsync, so the value returned here will be negative, and the frame counter should have advanced by 1 (indicating no frames dropped).

All Virtual Display drivers should behave this way up to this point. Ideally, this commonality would be pushed into the OpenVR runtime itself, however, since it requires a D3D device and each driver will need to create their own device to open the shared texture handle, this unfortunately has to be handled inside the driver itself. That's really what this example is all about - getting this part of the pipeline all working correctly. What you do with the texture beyond this point is really up to your own requirements as long as it can be consistently finished in the specified Additional Latency timeframe.

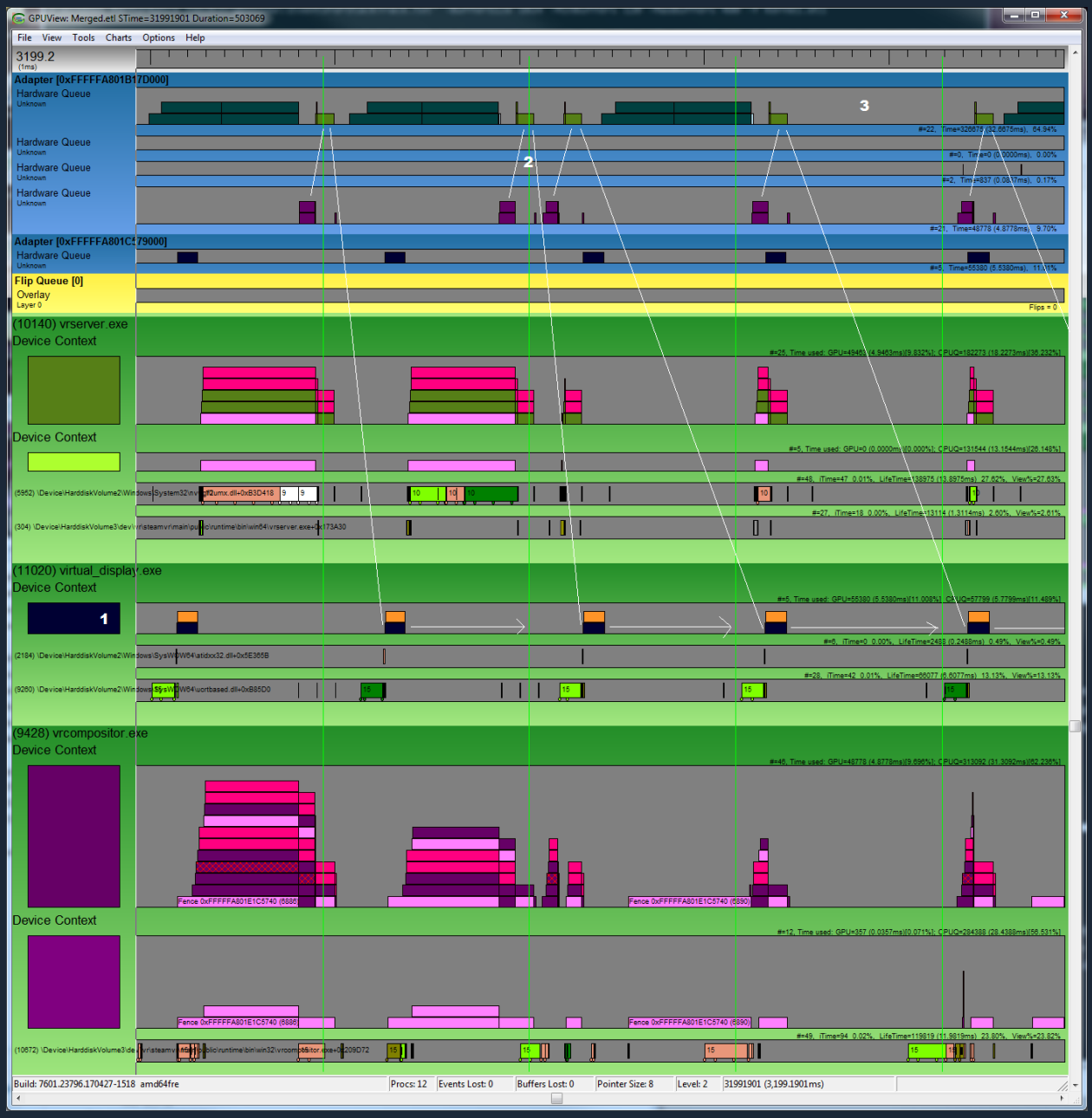
The rest of these steps are specific to the example's use of a second graphics adapter to mimic a "remote display".

23. Driver Maps "staging" copy of backbuffer, copies into shared memory and signals virtual_display.exe.
24. Virtual_display.exe copies texture data out of shared memory, and then calls Present on a fullscreen swap chain created on the headset display in extended mode. If the thread had been woken up early, it would have waited until it was on the right side of vsync before calling Present to ensure it gets scanned out and illuminated at the proper vsync interval that the poses used to render the scene were predicted to.
25. This block represents the time it takes to upload to the gpu. The orange block on top indicates that this packet contains a Present command.
26. We know when our next vsync should be, so we Sleep until then.
27. Then poll every 1ms to ensure we are on the right side of vsync.

```
UINT nLastPresentCount;
m_pDXGISwapChain->GetLastPresentCount( &nLastPresentCount );

DXGI_FRAME_STATISTICS stats;
m_pDXGISwapChain->GetFrameStatistics( &stats );

while ( stats.PresentCount != nLastPresentCount )
{
    Sleep( 1 );
    m_pDXGISwapChain->GetFrameStatistics( &stats );
}
```

28. And finally updating shared memory with the latest vsync timing info.

Now let's take a look at what happens when the application takes too long to render a frame:



A couple things to note here.

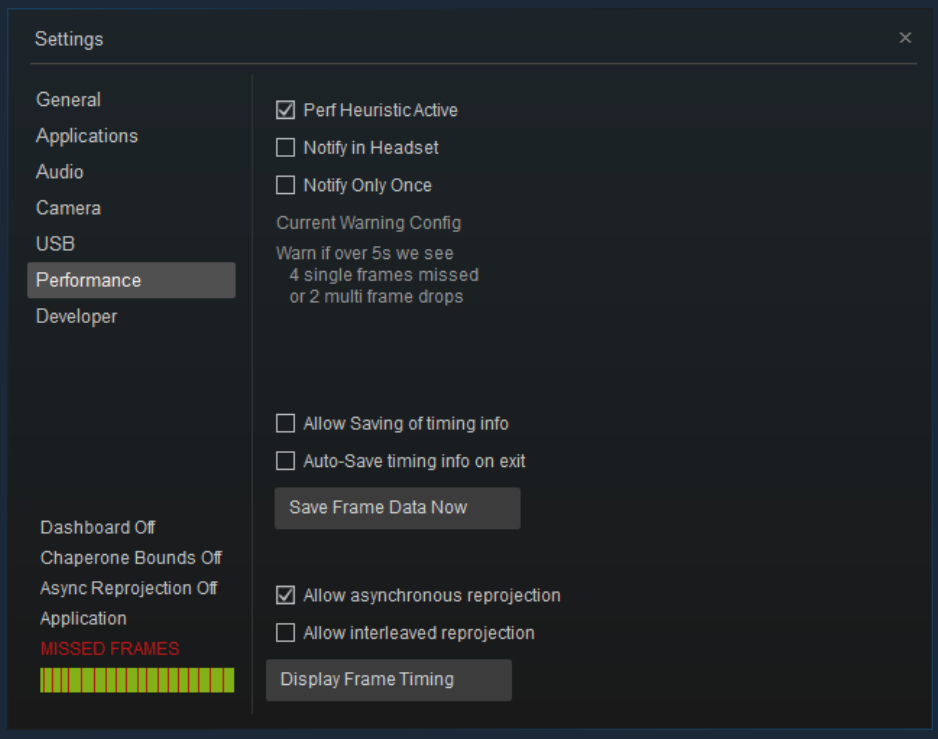
1. Virtual_display.exe never drops a frame. It just keeps chugging along presenting every frame even though the application drops down to half rate.
2. On the second frame, the compositor has made a decision that a frame drop is imminent. This is because the previous two frames were nearing our frame budget and their timing is used to extrapolate a prediction for how long the next frame will take, and

that was going to be over budget. At this point, it decides to present that second frame in the capture twice. This is why you see two sets of purple blocks on either side of the second (green) vsync (real) line in the hardware queue, and two light green blocks corresponding to the virtual display driver gpu work for each of those frames. The first present is expected to be queued up as normal, while the second is meant for the next frame.

3. These presents have shifted closer to the start of the frame effectively making running start a full frame rather than just 3ms.

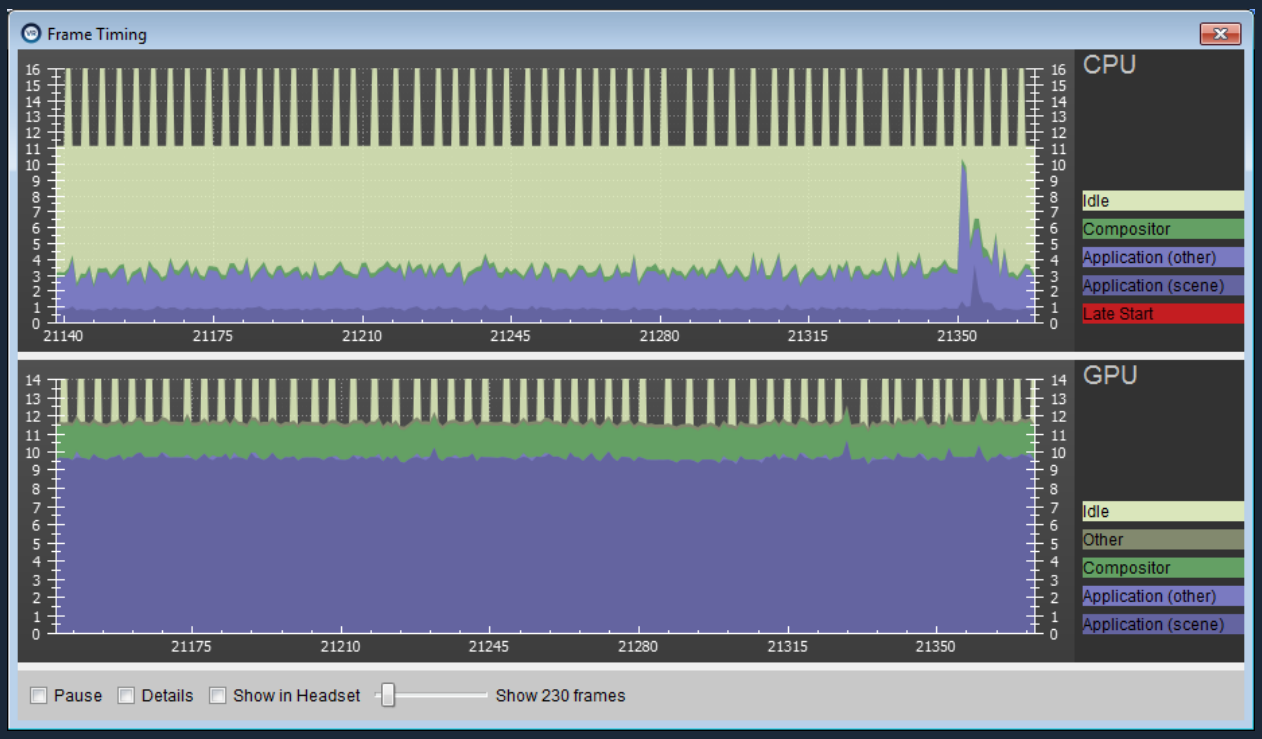
This is called Interleaved Reprojection. While active, each frame from the application is presented twice: once normally, and a second time we reprojection. Virtual Display drivers don't need to be concerned with whether a frame has reprojection applied to it or not, it just needs to be able to handle the changing cadence of Present calls.

If you disable Interleaved Reprojection (under Performance in the desktop SteamVR settings):

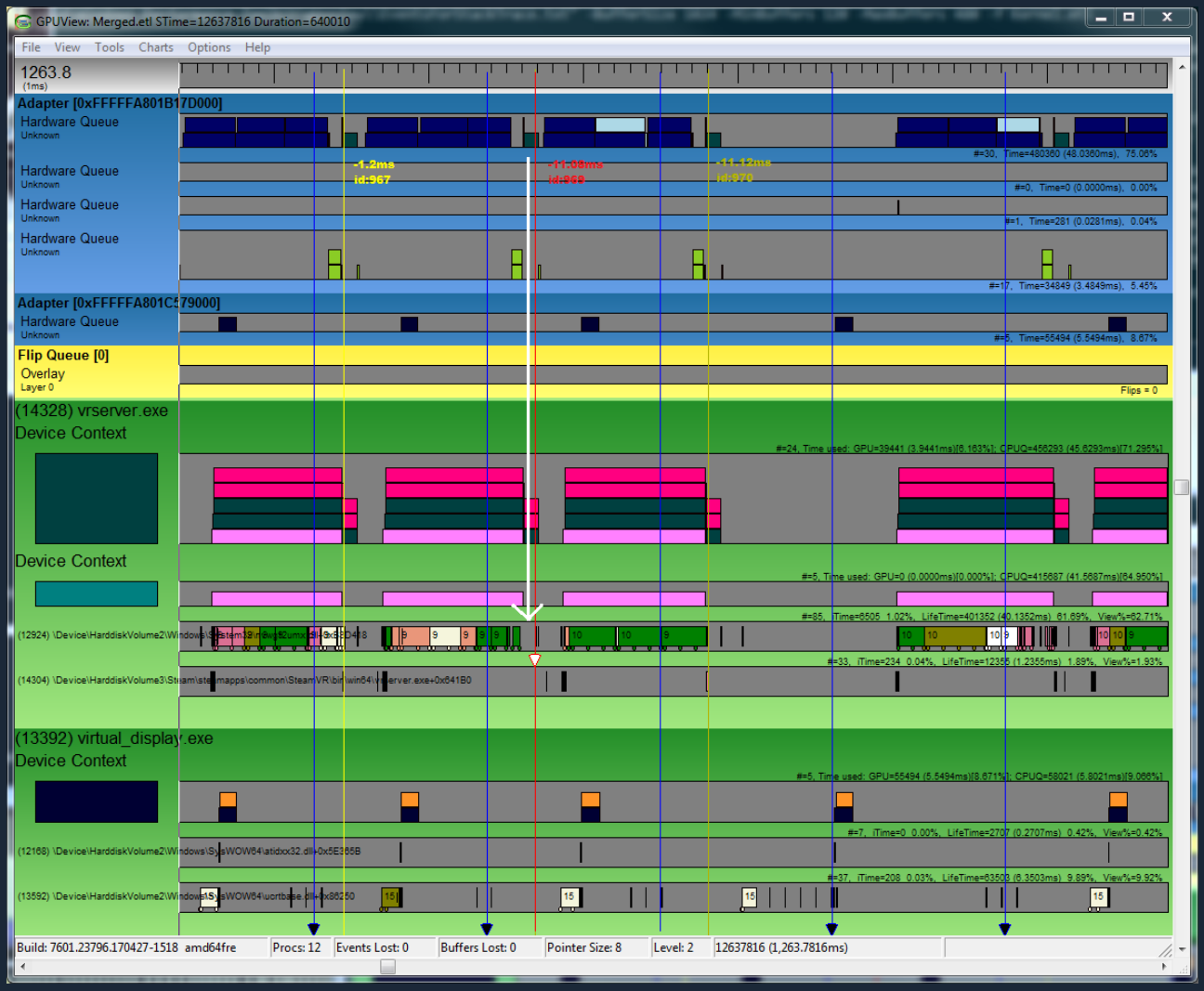


you should see red lines show up in the missed frames bar, if you are reporting dropped frame correctly. These red lines should correspond to hitches you see in the headset when looking around.

In the Frame Timing graph, you will see a yellow spike for each dropped frame like so:



Then in gpuview, you should expect to see something like this:



The first frame (bright yellow line) reported “last” vsync at -1.2ms, so it finished rendering with plenty of time. Note the frame id of 967, and that the yellow line shows up right after that first sliver of dark green work finishes (this is the copy single pixel discussed earlier).

The second frame (bright red) missed our virtual vsync. We know this because a) it calculated the next frame id after rendering finished to be 969 (we missed 968), and b) the “last” vsync time is a full 11ms away (reflecting the gap). It’s not clear what happened here. There is a gap (white arrow) between when the graphics driver finished spinning and when it notified the virtual display driver - probably a Windows scheduling issue. In any case, it caused the application rendering (or the driver’s ability to measure it) to miss the *virtual* deadline. What’s somewhat interesting here is that we still are able to copy the texture over to our “remote device” and

present it in time. We still report this as missed because we are being conservative with our Additional Latency estimate in order to handle variance on that side of things. We could have updated the time passed to the encoder for which *real* vsync interval to present the frame on, but that would just make things worse. This also means the next frame comes in early, and you can see on the bottom thread (13592) where virtual_display.exe holds off presenting until the next *real* vsync interval, at which point things go back to normal.

QUESTIONS

Please post questions here:

https://github.com/ValveSoftware/virtual_display/issues