

Oculus Rift Developer Guide

Version 1.23

Copyrights and Trademarks

© 2017 Oculus VR, LLC. All Rights Reserved.

OCULUS VR, OCULUS, and RIFT are trademarks of Oculus VR, LLC. (C) Oculus VR, LLC. All rights reserved. BLUETOOTH is a registered trademark of Bluetooth SIG, Inc. All other trademarks are the property of their respective owners. Certain materials included in this publication are reprinted with the permission of the copyright holder.

Contents

LibOVR Integration 5
Overview of the SDK 5

Initialization and Sensor Enumeration 6
Head Tracking and Sensors 8
Position Tracking 10
User Input Integration 12
Health and Safety Warning 13

Rendering to the Oculus Rift 14
Rendering to the Oculus Rift 15
Rendering Setup Outline 16
Texture Swap Chain Initialization 17
Frame Rendering 21
Frame Timing 23
Rendering on Different Threads 24
Layers 25
Working with HMD Eye Poses 31
Asynchronous TimeWarp 32
Adaptive Queue Ahead 33

Advanced Rendering Configuration 35
Using Oculus Dash 35
Coping with Graphics API or Hardware Render Target Granularity 37
Forcing a Symmetrical Field of View 38
Improving Performance by Decreasing Pixel Density 40
Improving Performance by Decreasing Field of View 41
Improving Performance by Rendering in Mono 42
Using Octilinear Rendering for NVIDIA Lens Matched Shading 42
Protecting Content 43

VR Focus Management 45

Oculus Guardian System 48

Rift Audio 51

Oculus Touch Controllers 57
Controller Data 58
Hand Tracking 59
Button State 59
Button Touch State 62
Haptic Feedback 62
Emulating Gamepad Input with Touch 67

SDK Samples and Gamepad Usage 72

Optimizing Your Application 73

VR Performance Optimization Guide 73
Guidelines for VR Performance Optimization 73
Workflows: The process flows you should follow 75
Performance Optimization Tools 76
Tutorial: Optimizing a Sample Application 79
Additional Resources 121
Lost Frame Capture Tool 122
SDK Performance Statistics 129
Oculus Debug Tool 132
Performance Profiler 137
Performance Head-Up Display 139
Performance Indicator 146
Compositor Mirror 148

Pairing the Oculus Touch Controllers 155

Asynchronous SpaceWarp 159

Mixed Reality Capture 161

VRC Validator 165

LibOVR Integration

The Oculus SDK is designed to be as easy to integrate as possible. This guide outlines a basic Oculus integration with a C/C++ game engine or application.

We'll discuss initializing the LibOVR, HMD device enumeration, head tracking, frame timing, and rendering for the Rift.

Many of the code samples below are taken directly from the OculusRoomTiny demo source code (available in `Oculus/LibOVR/Samples/OculusRoomTiny`). OculusRoomTiny and OculusWorldDemo are great places to view sample integration code when in doubt about a particular system or feature.

Overview of the SDK

There are three major phases when using the SDK: setup, the game loop, and shutdown.

To add Oculus support to a new application, do the following:

1. Initialize LibOVR through `ovr_Initialize`.
2. Call `ovr_Create` and check the return value to see if it succeeded. You can periodically poll for the presence of an HMD with ovr_GetHmdDesc(nullptr).
3. Integrate head-tracking into your application's view and movement code. This involves:
    a. Obtaining predicted headset orientation for the frame through a combination of the `GetPredictedDisplayTime` and `ovr_GetTrackingState` calls.
    b. Applying Rift orientation and position to the camera view, while combining it with other application controls.
    c. Modifying movement and game play to consider head orientation.
4. Initialize rendering for the HMD.
    a. Select rendering parameters such as resolution and field of view based on HMD capabilities.
        - See: `ovr_GetFovTextureSize` and`ovr_GetRenderDesc`.
    b. Configure rendering by creating D3D/OpenGL-specific swap texture sets to present data to the headset.
        - See: `ovr_CreateTextureSwapChainDX` and`ovr_CreateTextureSwapChainGL`.
5. Modify application frame rendering to integrate HMD support and proper frame timing:
    a. Make sure your engine supports rendering stereo views.
    b. Add frame timing logic into the render loop to obtain correctly predicted eye render poses.
    c. Render each eye's view to intermediate render targets.
    d. Submit the rendered frame to the headset by calling the functions `ovr_WaitToBeginFrame`, `ovr_BeginFrame`, and `ovr_EndFrame`.
6. Customize UI screens to work well inside of the headset.
7. Destroy the created resources during shutdown.
    - See: `ovr_DestroyTextureSwapChain`, `ovr_Destroy`, and `ovr_Shutdown`.

A more complete summary of rendering details is provided in the *Rendering Setup Outline* on page 16 section.

Initialization and Sensor Enumeration

This example initializes LibOVR and requests information about the available HMD.

Review the following code:

```
// Include the OculusVR SDK
#include <OVR_CAPI.h>
void Application()
{
   ovrResult result = ovr_Initialize(nullptr);
   if (OVR_FAILURE(result))
       return;

   ovrSession session;
   ovrGraphicsLuid luid;
   result = ovr_Create(&session, &luid);
   if (OVR_FAILURE(result))
   {
      ovr_Shutdown();
      return;
   }

   ovrHmdDesc desc = ovr_GetHmdDesc(session);
   ovrSizei resolution = desc.Resolution;

   ovr_Destroy(session);
   ovr_Shutdown();
}
```

As you can see, `ovr_Initialize` is called before any other API functions and `ovr_Shutdown` is called to shut down the library before you exit the program. In between these function calls, you are free to create HMD objects, access tracking state, and perform application rendering.

In this example, `ovr_Create(&session &luid)` creates the HMD. Use the LUID returned by ovr_Create() to select the IDXGIAdapter on which your ID3D11Device or ID3D12Device is created. Finally, `ovr_Destroy` must be called to clear the HMD before shutting down the library.

You can use ovr_GetHmdDesc() to get a description of the HMD.

If no Rift is plugged in, ovr_Create(&session, &luid) returns a failed ovrResult unless a virtual HMD is enabled through RiftConfigUtil. Although the virtual HMD will not provide any sensor input, it can be useful for debugging Rift-compatible rendering code and for general development without a physical device.

The description of an HMD (`ovrHmdDesc`) handle can be retrieved by calling `ovr_GetHmdDesc(session)`. The following table describes the fields:

Field	Type	Description
`Type`	ovrHmdType	Type of the HMD, such as `ovr_DK2` or `ovr_DK2` .
`ProductName`	char[]	Name of the product as a string.
`Manufacturer`	char[]	Name of the manufacturer.
`VendorId`	short	Vendor ID reported by the headset USB device.
`ProductId`	short	Product ID reported by the headset USB device.

Field	Type	Description
`SerialNumber`	char[]	Serial number string reported by the headset USB device.
`FirmwareMajor`	short	The major version of the sensor firmware.
`FirmwareMinor`	short	The minor version of the sensor firmware.
`AvailableHmdCaps`	unsigned int	Capability bits described by `ovrHmdCaps` which the HMD currently supports.
`DefaultHmdCaps`	unsigned int	Default capability bits described by `ovrHmdCaps` for the current HMD.
`AvailableTrackingCaps`	unsigned int	Capability bits described by `ovrTrackingCaps` which the HMD currently supports.
`DefaultTrackingCaps`	unsigned int	Default capability bits described by `ovrTrackingCaps` for the current HMD.
`DefaultEyeFov`	ovrFovPort[]	Recommended optical field of view for each eye.
`MaxEyeFov`	ovrFovPort[]	Maximum optical field of view that can be practically rendered for each eye.
`Resolution`	ovrSizei	Resolution of the full HMD screen (both eyes) in pixels.
`DisplayRefreshRate`	float	Nominal refresh rate of the HMD in cycles per second at the time of HMD creation.

The description of a sensor (`ovrTrackerDesc`) handle can be retrieved by calling `ovr_GetTrackerDesc(sensor)`. The following table describes the fields:

Field	Type	Description
`FrustumHFovInRadians`	float	The horizontal FOV of the position sensor frustum.
`FrustumVFovInRadians`	float	The vertical FOV of the position sensor frustum.
`FrustumNearZInMeters`	float	The distance from the position sensor to the near frustum bounds.
`FrustumNearZInMeters`	float	The distance from the position sensor to the far frustum bounds.

Head Tracking and Sensors

The Oculus Rift hardware contains a number of micro-electrical-mechanical (MEMS) sensors including a gyroscope, accelerometer, and magnetometer.

There is also a sensor to track headset position. The information from each of these sensors is combined through the sensor fusion process to determine the motion of the user's head in the real world and synchronize the user's view in real-time.

Once the ovrSession is created, you can poll sensor fusion for head position and orientation by calling `ovr_GetTrackingState`. These calls are demonstrated by the following code:

```
// Query the HMD for ts current tracking state.
ovrTrackingState ts = ovr_GetTrackingState(session, ovr_GetTimeInSeconds(), ovrTrue);

if (ts.StatusFlags & (ovrStatus_OrientationTracked | ovrStatus_PositionTracked))
{
    ovrPosef pose = ts.HeadPose.ThePose;
    ...
}
```

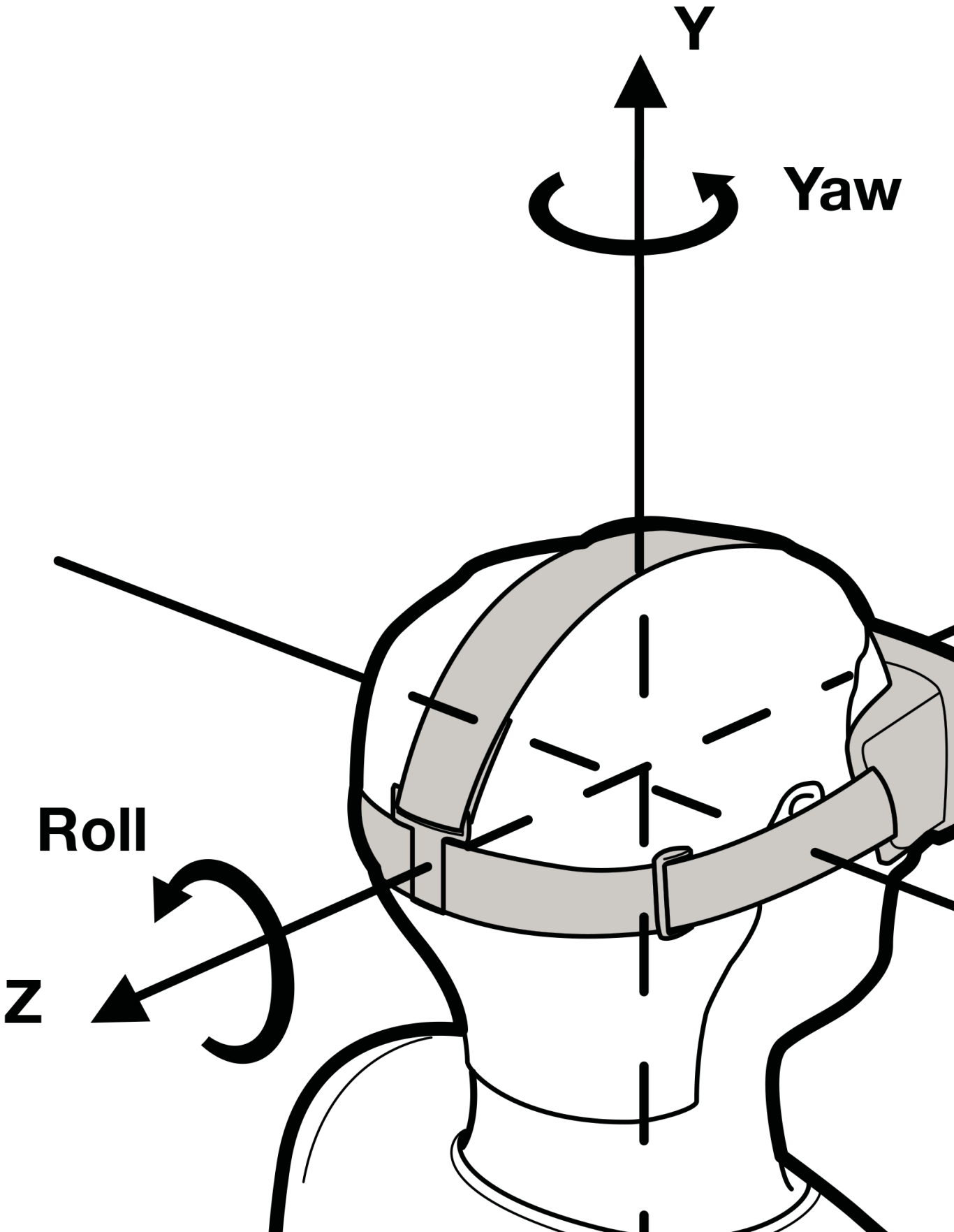
This example initializes the sensors with orientation, yaw correction, and position tracking capabilities. If the sensor is not available during the time of the call, but is plugged in later, the sensor is automatically enabled by the SDK.

After the sensors are initialized, the sensor state is obtained by calling `ovr_GetTrackingState`. This state includes the predicted head pose and the current tracking state of the HMD as described by `StatusFlags`. This state can change at runtime based on the available devices and user behavior. For example, the `ovrStatus_PositionTracked` flag is only reported when `HeadPose` includes the absolute positional tracking data from the sensor.

The reported `ovrPoseStatef` includes full six degrees of freedom (6DoF) head tracking data including orientation, position, and their first and second derivatives. The pose value is reported for a specified absolute point in time using prediction, typically corresponding to the time in the future that this frame's image will be displayed on screen. To facilitate prediction, `ovr_GetTrackingState` takes absolute time, in seconds, as a second argument. The current value of absolute time can be obtained by calling `ovr_GetTimeInSeconds`. If the time passed into `ovr_GetTrackingState` is the current time or earlier, the tracking state returned will be based on the latest sensor readings with no prediction. In a production application, however, you should use the real-time computed value returned by `GetPredictedDisplayTime`. Prediction is covered in more detail in the section on Frame Timing.

As already discussed, the reported pose includes a 3D position vector and an orientation quaternion. The orientation is reported as a rotation in a right-handed coordinate system, as illustrated in the following figure.

Figure 1: Rift Coordinate System



The x-z plane is aligned with the ground regardless of camera orientation.

As seen from the diagram, the coordinate system uses the following axis definitions:

- Y is positive in the up direction.
- X is positive to the right.
- Z is positive heading backwards.

Rotation is maintained as a unit quaternion, but can also be reported in yaw-pitch-roll form. Positive rotation is counter-clockwise (CCW, direction of the rotation arrows in the diagram) when looking in the negative direction of each axis, and the component rotations are:

- Pitch is rotation around X, positive when pitching up.
- Yaw is rotation around Y, positive when turning left.
- Roll is rotation around Z, positive when tilting to the left in the XY plane.

The simplest way to extract yaw-pitch-roll from `ovrPose` is to use the C++ OVR Math helper classes that are included with the library. The following example uses direct conversion to assign ovrPosef to the equivalent C ++ Posef class. You can then use the `Quatf::GetEulerAngles<>` to extract the Euler angles in the desired axis rotation order.

All simple C math types provided by OVR such as `ovrVector3f` and `ovrQuatf` have corresponding C++ types that provide constructors and operators for convenience. These types can be used interchangeably.

If an application uses a left-handed coordinate system, it can use the `ovrPosef_FlipHandedness` function to switch any right-handed `ovrPosef` provided by `ovr_GetTrackingState`, `ovr_GetEyePoses`, or `ovr_CalcEyePoses` functions to be left-handed. Be aware that the `RenderPose` and `QuadPoseCenter`requested for the ovrLayers must still use the right-handed coordinate system.

Position Tracking

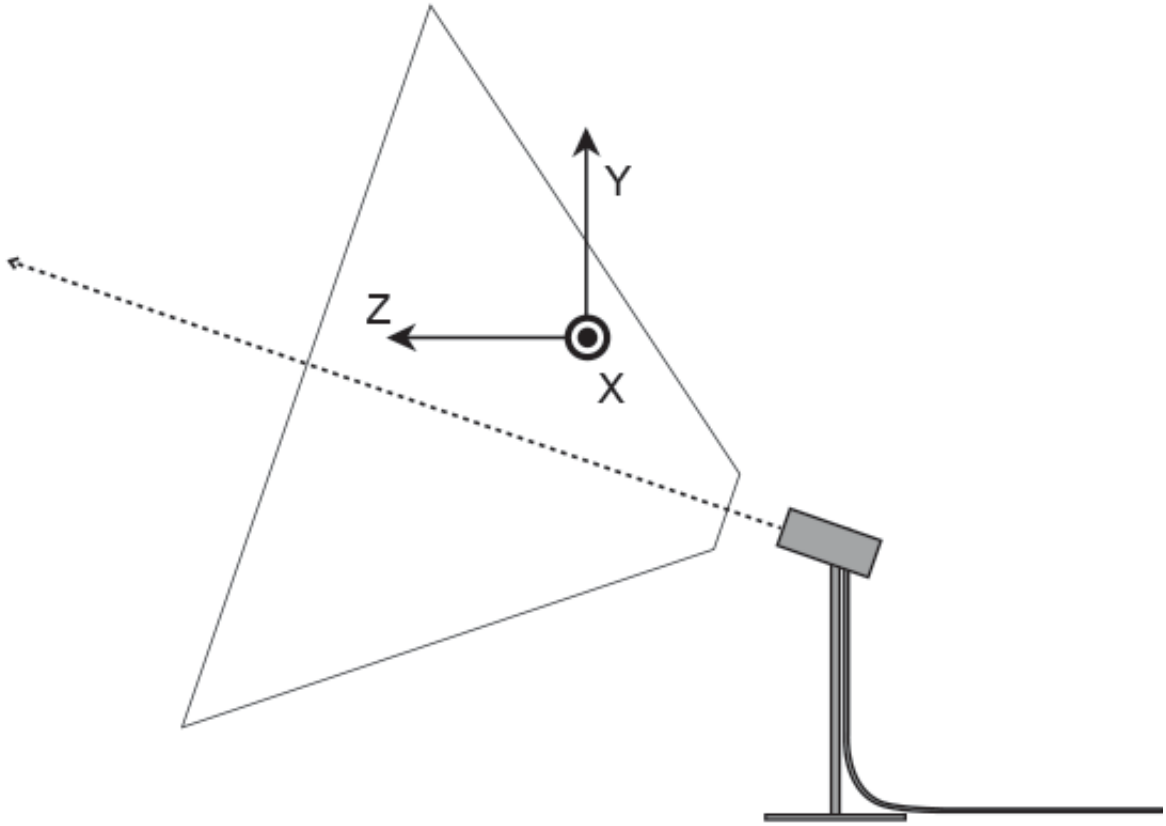
The frustum is defined by the horizontal and vertical FOV, and the distance to the front and back frustum planes.

Approximate values for these parameters can be accessed through the `ovrTrackerDesc` struct as follows:

```
ovrSession session;
ovrGraphicsLuid luid;
if(OVR_SUCCESS(ovr_Create(&session, &luid)))
{
    // Extract tracking frustum parameters.
    float frustomHorizontalFOV = session->CameraFrustumHFovInRadians;
    ...
```

The following figure shows the tracking sensor and a representation of the resulting tracking frustum.

Figure 2: Tracking Sensor and Tracking Frustum



The relevant parameters and typical values are list below:

Field	Type	Typical Value
`FrustumHFovInRadians`	float	1.292 radians (74 degrees)
`FrustumVFovInRadians`	float	0.942 radians (54 degrees)
`FrustumNearZInMeters`	float	0.4m
`FrustumFarZInMeters`	float	2.5m

These parameters provide application developers with a visual representation of the tracking frustum. The previous figure also shows the default tracking origin and associated coordinate system.

Note: Although the sensor axis (and hence the tracking frustum) are shown tilted downwards slightly, the tracking coordinate system is always oriented horizontally such that the axes are parallel to the ground.

By default, the tracking origin is located one meter away from the sensor in the direction of the optical axis but with the same height as the sensor. The default origin orientation is level with the ground with the negative axis pointing towards the sensor. In other words, a headset yaw angle of zero corresponds to the user looking towards the sensor.

This can be modified using the API call `ovr_RecenterTrackingOrigin`, which resets the tracking origin to the headset's current location and sets the yaw origin to the current headset yaw value. Additionally, it can be manually specified to any location using the API call `ovr_SpecifyTrackingOrigin`.

There are two types of tracking origins: floor-level and eye-level. Floor-level is recommended for room scale, when the user stands, and when the user is interacting with objects on the floor (although telekinesis/force grab/gaze grab is a better option for picking up objects). For most other experiences, especially when the user is seated, eye-level is preferred. To get the current origin, use `ovr_GetTrackingOriginType`. To set the origin, use `ovr_SetTrackingOriginType`.



Note: The tracking origin is set on a per application basis; switching focus between different VR apps also switches the tracking origin.

The head pose is returned by calling `ovr_GetTrackingState`. The returned `ovrTrackingState` struct contains several items relevant to position tracking:

- `HeadPose`—includes both head position and orientation.
- `Pose`—the pose of the sensor relative to the tracking origin.
- `CalibratedOrigin`—the pose of the origin previously calibrated by the user and stored in the profile reported in the new recentered tracking origin space. This value can change when the application calls `ovr_RecenterTrackingOrigin`, though it refers to the same location in real-world space. Otherwise it will remain as an identity pose. Different tracking origin types will report different CalibrateOrigin poses, as the calibration origin refers to a fixed position in real-world space but the two tracking origin types refer to different y levels.

The `StatusFlags` variable contains the following status bits relating to position tracking:

- `ovrStatus_PositionTracked`—flag that is set only when the headset is being actively tracked.

There are several conditions that may cause position tracking to be interrupted and for the flag to become zero:

- The headset moved wholly or partially outside the tracking frustum.
- The headset adopts an orientation that is not easily trackable with the current hardware (for example facing directly away from the sensor).
- The exterior of the headset is partially or fully occluded from the sensor's point of view (for example by hair or hands).
- The velocity of the headset exceeds the expected range.

Following an interruption, assuming the conditions above are no longer present, tracking normally resumes quickly and the `ovrStatus_PositionTracked` flag is set.

If you want to get the pose and leveled pose of a sensor, call `ovr_GetTrackerPose`. The returned `ovrTrackerPose` struct contains the following:

- `Pose`—the pose of the sensor relative to the tracking origin.
- `LeveledPose`— the pose of the sensor relative to the tracking origin but with roll and pitch zeroed out. You can use this as a reference point to render real-world objects in the correct place.

User Input Integration

To provide the most comfortable, intuitive, and usable interface for the player, head tracking should be integrated with an existing control scheme for most applications.

For example, in a first person shooter (FPS) game, the player generally moves forward, backward, left, and right using the left joystick, and looks left, right, up, and down using the right joystick. When using the Rift, the player can now look left, right, up, and down, using their head. However, players should not be required to frequently turn their heads 180 degrees since this creates a bad user experience. Generally, they need a way to reorient themselves so that they are always comfortable (the same way in which we turn our bodies if we want to look behind ourselves for more than a brief glance).

To summarize, developers should carefully consider their control schemes and how to integrate head-tracking when designing applications for VR. The OculusRoomTiny application provides a source code sample that shows how to integrate Oculus head tracking with the aforementioned standard FPS control scheme.

For more information about good and bad practices, refer to the *Oculus Best Practices Guide*.

Health and Safety Warning

All applications that use the Oculus Rift periodically display a health and safety warning.

This warning appears for a short amount of time when the user wears the Rift; it can be dismissed by pressing a key or gazing at the acknowledgement. After the screen is dismissed, it shouldn't display for at least 30 minutes.

Rendering to the Oculus Rift

The Oculus Rift requires split-screen stereo with distortion correction for each eye to cancel lens-related distortion.

Figure 3: OculusWorldDemo Stereo Rendering



Correcting for distortion can be challenging, with distortion parameters varying for different lens types and individual eye relief. To make development easier, Oculus SDK handles distortion correction automatically within the Oculus Compositor process; it also takes care of latency-reducing timewarp and presents frames to the headset.

With Oculus SDK doing a lot of the work, the main job of the application is to perform simulation and render stereo world based on the tracking pose. Stereo views can be rendered into either one or two individual textures and are submitted to the compositor by calling `ovr_WaitToBeginFrame`, `ovr_BeginFrame`, and `ovr_EndFrame`. We cover this process in detail in this section.

Rendering to the Oculus Rift

The Oculus Rift requires the scene to be rendered in split-screen stereo with half of the screen used for each eye.

When using the Rift, the left eye sees the left half of the screen, and the right eye sees the right half. Although varying from person-to-person, human eye pupils are approximately 65 mm apart. This is known as interpupillary distance (IPD). The in-application cameras should be configured with the same separation.

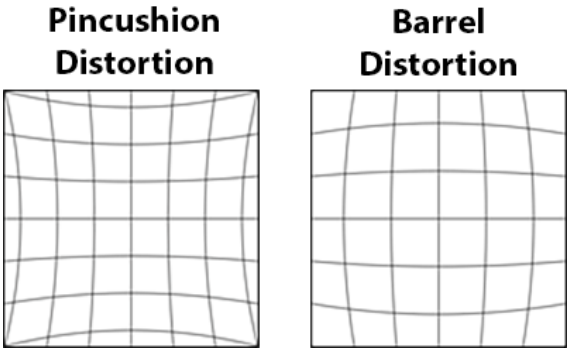
Note:

This is a translation of the camera, not a rotation, and it is this translation (and the parallax effect that goes with it) that causes the stereoscopic effect. This means that your application will need to render the entire scene twice, once with the left virtual camera, and once with the right.

The reprojection stereo rendering technique, which relies on left and right views being generated from a single fully rendered view, is usually not viable with an HMD because of significant artifacts at object edges.

The lenses in the Rift magnify the image to provide a very wide field of view (FOV) that enhances immersion. However, this process distorts the image significantly. If the engine were to display the original images on the Rift, then the user would observe them with pincushion distortion.

Figure 4: Pincushion and Barrel Distortion

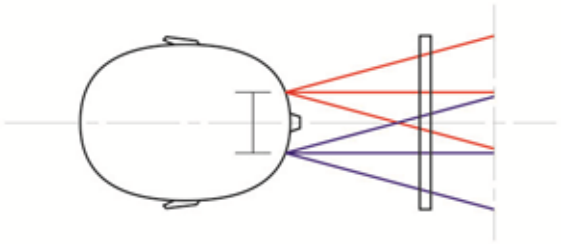


To counteract this distortion, the SDK applies post-processing to the rendered views with an equal and opposite barrel distortion so that the two cancel each other out, resulting in an undistorted view for each eye. Furthermore, the SDK also corrects chromatic aberration, which is a color separation effect at the edges caused by the lens. Although the exact distortion parameters depend on the lens characteristics and eye position relative to the lens, the Oculus SDK takes care of all necessary calculations when generating the distortion mesh.

When rendering for the Rift, projection axes should be parallel to each other as illustrated in the following figure, and the left and right views are completely independent of one another. This means that camera setup

is very similar to that used for normal non-stereo rendering, except that the cameras are shifted sideways to adjust for each eye location.

Figure 5: HMD Eye View Cones



In practice, the projections in the Rift are often slightly off-center because our noses get in the way! But the point remains, the left and right eye views in the Rift are entirely separate from each other, unlike stereo views generated by a television or a cinema screen. This means you should be very careful if trying to use methods developed for those media because they do not usually apply in VR.

The two virtual cameras in the scene should be positioned so that they are oriented in the same way as the eye poses, specified by `ovrEyeRenderDesc::HmdToEyePose`, and such that the distance between them is the same as the distance between the eyes, or the interpupillary distance (IPD).

Although the Rift's lenses are approximately the right distance apart for most users, they may not exactly match the user's IPD. However, because of the way the optics are designed, each eye will still see the correct view. It is important that the software makes the distance between the virtual cameras match the user's IPD as found in their profile (set in the configuration utility), and not the distance between the Rift's lenses.

Rendering Setup Outline

The Oculus SDK makes use of a compositor process to present frames and handle distortion.

To target the Rift, you render the scene into one or two render textures, passing these textures into the API. The Oculus runtime handles distortion rendering, GPU synchronization, frame timing, and frame presentation to the HMD.

The following are the steps for SDK rendering:

1. Initialize:
    a. Initialize Oculus SDK and create an `ovrSession` object for the headset as was described earlier.
    b. Compute the desired FOV and texture sizes based on `ovrHmdDesc` data.
    c. Allocate `ovrTextureSwapChain` objects, used to represent eye buffers, in an API-specific way: call `ovr_CreateTextureSwapChainDX` for either Direct3D 11 or 12, `ovr_CreateTextureSwapChainGL` for OpenGL, or `ovr_CreateTextureSwapChainVk` for Vulkan.
2. Set up frame handling:
    a. Use `ovr_GetTrackingState` and `ovr_CalcEyePoses` to compute eye poses needed for view rendering based on frame timing information.
    b. Perform rendering for each eye in an engine-specific way, rendering into the current texture within the texture set. Current texture is retrieved using `ovr_GetTextureSwapChainCurrentIndex` and `ovr_GetTextureSwapChainBufferDX`, `ovr_GetTextureSwapChainBufferGL`, or

`ovr_GetTextureSwapChainBufferVk`. After rendering to the texture is complete, the application must call `ovr_CommitTextureSwapChain`.

c. Call `ovr_WaitToBeginFrame`, and then when your application is ready to begin rendering the frame, call `ovr_BeginFrame`. When your application is ready to submit the frame, call `ovr_EndFrame`, passing swap texture set(s) from the previous step within an `ovrLayerEyeFov` structure. Although a single layer is required to submit a frame, you can use multiple layers and layer types for advanced rendering. `ovr_EndFrame` passes layer textures to the compositor which handles distortion, timewarp, and GPU synchronization before presenting it to the headset. Note that the combination of `ovr_WaitToBeginFrame`, `ovr_BeginFrame`, and `ovr_EndFrame` enables you to implement performance optimization techniques in multi-threaded environments, for example by splitting apart and overlapping the processing of multiple frames at the same time. In previous releases, these three functions were combined into `ovr_SubmitFrame`, but that call has now been deprecated; please use `ovr_WaitToBeginFrame`, `ovr_BeginFrame`, and `ovr_EndFrame` instead.

3. Shutdown:

a. Call `ovr_DestroyTextureSwapChain` to destroy swap texture buffers. Call `ovr_DestroyMirrorTexture` to destroy a mirror texture. To destroy the `ovrSession` object, call `ovr_Destroy`.

Texture Swap Chain Initialization

This section describes rendering initialization, including creation of texture swap chains.

Initially, you determine the rendering FOV and allocate the required ovrTextureSwapChain. The following code shows how the required texture size can be computed:

```
// Configure Stereo settings.
Sizei recommenedTex0Size = ovr_GetFovTextureSize(session, ovrEye_Left,
                                                 session->DefaultEyeFov[0], 1.0f);
Sizei recommenedTex1Size = ovr_GetFovTextureSize(session, ovrEye_Right,
                                                 session->DefaultEyeFov[1], 1.0f);
Sizei bufferSize;
bufferSize.w  = recommenedTex0Size.w + recommenedTex1Size.w;
bufferSize.h = max ( recommenedTex0Size.h, recommenedTex1Size.h );
```

Render texture size is determined based on the FOV and the desired pixel density at the center of the eye. Although both the FOV and pixel density values can be modified to improve performance, this example uses the recommended FOV (obtained from `session->DefaultEyeFov`). The function `ovr_GetFovTextureSize` computes the desired texture size for each eye based on these parameters.

The Oculus API allows the application to use either one shared texture or two separate textures for eye rendering. This example uses a single shared texture for simplicity, making it large enough to fit both eye renderings.

If you're using Vulkan, there are three steps that you need to add before you can create the texture swap chain.

- During initialization, call `ovr_GetSessionPhysicalDeviceVk` to get the current physical device matching the luid. Then, create a VkDevice associated with the returned physical device.
- AMD hardware uses different extensions on Vulkan. Add code similar to the following example to handle the AMD GPU extensions during app initalization. This code example comes from the Win32_VulkanAppUtil.h that comes with the OculusRoomTiny_Advanced sample app that ships with the Oculus SDK.

```
static const uint32_t AMDVendorId = 0x1002;
   isAMD = (gpuProps.vendorID == AMDVendorId);

   static const char* deviceExtensions[] =
   {
       VK_KHR_SWAPCHAIN_EXTENSION_NAME,
       VK_KHX_EXTERNAL_MEMORY_EXTENSION_NAME,
       #if defined(VK_USE_PLATFORM_WIN32_KHR)
```

```
            VK_KHX_EXTERNAL_MEMORY_WIN32_EXTENSION_NAME,
        #endif
    };

    static const char* deviceExtensionsAMD[] =
    {
       VK_KHR_SWAPCHAIN_EXTENSION_NAME
    };
```

- Then, after the game loop has been established, identify which queue to synchronize when rendering. Call `ovr_SetSynchonizationQueueVk` to identify the queue.

Create the Texture Swap Chain

Once texture size is known, the application can call `ovr_CreateTextureSwapChainGL`, `ovr_CreateTextureSwapChainDX`, or `ovr_CreateTextureSwapChainVk` to allocate the texture swap chains in an API-specific way.

Here's how a texture swap chain can be created and accessed under OpenGL:

```
ovrTextureSwapChain textureSwapChain = 0;

ovrTextureSwapChainDesc desc = {};
desc.Type = ovrTexture_2D;
desc.ArraySize = 1;
desc.Format = OVR_FORMAT_R8G8B8A8_UNORM_SRGB;
desc.Width = bufferSize.w;
desc.Height = bufferSize.h;
desc.MipLevels = 1;
desc.SampleCount = 1;
desc.StaticImage = ovrFalse;

if (ovr_CreateTextureSwapChainGL(session, &desc, &textureSwapChain) == ovrSuccess)
{
    // Sample texture access:
    int texId;
    ovr_GetTextureSwapChainBufferGL(session, textureSwapChain, 0, &texId);
    glBindTexture(GL_TEXTURE_2D, texId);
    ...
}
```

Here's a similar example of texture swap chain creation and access using Direct3D 11:

```
ovrTextureSwapChain textureSwapChain = 0;
std::vector<ID3D11RenderTargetView*> texRtv;

ovrTextureSwapChainDesc desc = {};
desc.Type = ovrTexture_2D;
desc.Format = OVR_FORMAT_R8G8B8A8_UNORM_SRGB;
desc.ArraySize = 1;
desc.Width = bufferSize.w;
desc.Height = bufferSize.h;
desc.MipLevels = 1;
desc.SampleCount = 1;
desc.StaticImage = ovrFalse;
desc.MiscFlags = ovrTextureMisc_None;
desc.BindFlags = ovrTextureBind_DX_RenderTarget;

 if (ovr_CreateTextureSwapChainDX(session, DIRECTX.Device, &desc, &textureSwapChain) == ovrSuccess)
 {
     int count = 0;
     ovr_GetTextureSwapChainLength(session, textureSwapChain, &count);
     texRtv.resize(textureCount);
     for (int i = 0; i < count; ++i)
     {
         ID3D11Texture2D* texture = nullptr;
         ovr_GetTextureSwapChainBufferDX(session, textureSwapChain, i, IID_PPV_ARGS(&texture));
         DIRECTX.Device->CreateRenderTargetView(texture, nullptr, &texRtv[i]);
         texture->Release();
     }
 }
```

Here's sample code from the provided OculusRoomTiny sample running in Direct3D 12:

```
ovrTextureSwapChain TexChain;
std::vector<D3D12_CPU_DESCRIPTOR_HANDLE> texRtv;
std::vector<ID3D12Resource*> TexResource;

ovrTextureSwapChainDesc desc = {};
desc.Type = ovrTexture_2D;
desc.ArraySize = 1;
desc.Format = OVR_FORMAT_R8G8B8A8_UNORM_SRGB;
desc.Width = sizeW;
desc.Height = sizeH;
desc.MipLevels = 1;
desc.SampleCount = 1;
desc.MiscFlags = ovrTextureMisc_DX_Typeless;
desc.StaticImage = ovrFalse;
desc.BindFlags = ovrTextureBind_DX_RenderTarget;

// DIRECTX.CommandQueue is the ID3D12CommandQueue used to render the eye textures by the app
ovrResult result = ovr_CreateTextureSwapChainDX(session, DIRECTX.CommandQueue, &desc, &TexChain);
if (!OVR_SUCCESS(result))
    return false;

int textureCount = 0;
ovr_GetTextureSwapChainLength(Session, TexChain, &textureCount);
texRtv.resize(textureCount);
TexResource.resize(textureCount);
for (int i = 0; i < textureCount; ++i)
{
    result = ovr_GetTextureSwapChainBufferDX(Session, TexChain, i, IID_PPV_ARGS(&TexResource[i]));
    if (!OVR_SUCCESS(result))
        return false;

    D3D12_RENDER_TARGET_VIEW_DESC rtvd = {};
    rtvd.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
    rtvd.ViewDimension = D3D12_RTV_DIMENSION_TEXTURE2D;
    texRtv[i] = DIRECTX.RtvHandleProvider.AllocCpuHandle(); // Gives new D3D12_CPU_DESCRIPTOR_HANDLE
    DIRECTX.Device->CreateRenderTargetView(TexResource[i], &rtvd, texRtv[i]);
}
```

Note: For Direct3D 12, when calling `ovr_CreateTextureSwapChainDX`, the caller provides a ID3D12CommandQueue instead of a ID3D12Device to the SDK. It is the caller's responsibility to make sure that this ID3D12CommandQueue instance is where all VR eye-texture rendering is executed. Or, it can be used as a "join-node" fence to wait for the command lists executed by other command queues rendering the VR eye textures.

Here's how a texture swap chain can be created and accessed using Vulkan:

```
bool Create(ovrSession aSession, VkExtent2D aSize, RenderPass& renderPass, DepthBuffer& depthBuffer)
{
    session = aSession;
    size = aSize;

    ovrTextureSwapChainDesc desc = {};
    desc.Type = ovrTexture_2D;
    desc.ArraySize = 1;
    desc.Format = OVR_FORMAT_R8G8B8A8_UNORM_SRGB;
    desc.Width = (int)size.width;
    desc.Height = (int)size.height;
    desc.MipLevels = 1;
    desc.SampleCount = 1;
    desc.MiscFlags = ovrTextureMisc_DX_Typeless;
    desc.BindFlags = ovrTextureBind_DX_RenderTarget;
    desc.StaticImage = ovrFalse;

    ovrResult result = ovr_CreateTextureSwapChainVk(session, Platform.device, &desc, &textureChain);
    if (!OVR_SUCCESS(result))
        return false;

    int textureCount = 0;
    ovr_GetTextureSwapChainLength(session, textureChain, &textureCount);
    texElements.reserve(textureCount);
    for (int i = 0; i < textureCount; ++i)
    {
        VkImage image;
```

```
        result = ovr_GetTextureSwapChainBufferVk(session, textureChain, i, &image);
        texElements.emplace_back(RenderTexture());
        CHECK(texElements.back().Create(image, VK_FORMAT_R8G8B8A8_SRGB, size, renderPass,
 depthBuffer.view));
   }

    return true;
}
```

Once these textures and render targets are successfully created, you can use them to perform eye-texture rendering. The Frame Rendering section describes viewport setup in more detail.

The Oculus compositor provides sRGB-correct rendering, which results in more photorealistic visuals, better MSAA, and energy-conserving texture sampling, which are very important for VR applications. As shown above, applications are expected to create sRGB texture swap chains. Proper treatment of sRGB rendering is a complex subject and, although this section provides an overview, extensive information is outside the scope of this document.

There are several steps to ensuring a real-time rendered application achieves sRGB-correct shading and different ways to achieve it. For example, most GPUs provide hardware acceleration to improve gamma-correct shading for sRGB-specific input and output surfaces, while some applications use GPU shader math for more customized control. For the Oculus SDK, when an application passes in sRGB-space texture swap chains, the compositor relies on the GPU's sampler to do the sRGB-to-linear conversion.

All color textures fed into a GPU shader should be marked appropriately with the sRGB-correct format, such as OVR_FORMAT_R8G8B8A8_UNORM_SRGB. This is also recommended for applications that provide static textures as quad-layer textures to the Oculus compositor. Failure to do so will cause the texture to look much brighter than expected.

For D3D 11 and 12, the texture format provided in `desc` for `ovr_CreateTextureSwapChainDX` is used by the distortion compositor for the ShaderResourceView when reading the contents of the texture. As a result, the application should request texture swap chain formats that are in sRGB-space (e.g. OVR_FORMAT_R8G8B8A8_UNORM_SRGB).

If your application is configured to render into a linear-format texture (e.g. OVR_FORMAT_R8G8B8A8_UNORM) and handles the linear-to-gamma conversion using HLSL code, or does not care about any gamma-correction, then:

- Request an sRGB format (e.g. OVR_FORMAT_R8G8B8A8_UNORM_SRGB) texture swap chain.
- Specify the `ovrTextureMisc_DX_Typeless` flag in the desc.
- Create a linear-format RenderTargetView (e.g. DXGI_FORMAT_R8G8B8A8_UNORM)



Note: The `ovrTextureMisc_DX_Typeless` flag for depth buffer formats (e.g. OVR_FORMAT_D32) is ignored as they are always converted to be typeless.

The provided code sample demonstrates how to use the provided `ovrTextureMisc_DX_Typeless` flag in D3D11:

```
    ovrTextureSwapChainDesc desc = {};
    desc.Type = ovrTexture_2D;
    desc.ArraySize = 1;
    desc.Format = OVR_FORMAT_R8G8B8A8_UNORM_SRGB;
    desc.Width = sizeW;
    desc.Height = sizeH;
    desc.MipLevels = 1;
    desc.SampleCount = 1;
    desc.MiscFlags = ovrTextureMisc_DX_Typeless;
    desc.BindFlags = ovrTextureBind_DX_RenderTarget;
    desc.StaticImage = ovrFalse;

    ovrResult result = ovr_CreateTextureSwapChainDX(session, DIRECTX.Device, &desc,
&textureSwapChain);

    if(!OVR_SUCCESS(result))
```

```
        return;

    int count = 0;
    ovr_GetTextureSwapChainLength(session, textureSwapChain, &count);
    for (int i = 0; i < count; ++i)
    {
        ID3D11Texture2D* texture = nullptr;
        ovr_GetTextureSwapChainBufferDX(session, textureSwapChain, i, IID_PPV_ARGS(&texture));
        D3D11_RENDER_TARGET_VIEW_DESC rtvd = {};
        rtvd.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
        rtvd.ViewDimension = D3D11_RTV_DIMENSION_TEXTURE2D;
        DIRECTX.Device->CreateRenderTargetView(texture, &rtvd, &texRtv[i]);
        texture->Release();
    }
```

For OpenGL, the `format` parameter of`ovr_CreateTextureSwapChainGL` is used by the distortion compositor when reading the contents of the texture. As a result, the application should request texture swap chain formats preferably in sRGB-space (e.g. OVR_FORMAT_R8G8B8A8_UNORM_SRGB). Furthermore, your application should call `glEnable(GL_FRAMEBUFFER_SRGB);` before rendering into these textures.

Even though it is not recommended, if your application is configured to treat the texture as a linear format (e.g. GL_RGBA) and performs linear-to-gamma conversion in GLSL or does not care about gamma-correction, then:

- Request an sRGB format (e.g. OVR_FORMAT_R8G8B8A8_UNORM_SRGB) texture swap chain.
- Do not call `glEnable(GL_FRAMEBUFFER_SRGB);` when rendering into the texture.

For Vulkan, the `format` parameter of `ovr_CreateTextureSwapChainVk` is used by the distortion compositor when reading the contents of the texture. Your application should request texture swap chain formats in the sRGB-space (e.g. OVR_FORMAT_R8G8B8A8_UNORM_SRGB) as the compositor does sRGB-correct rendering. The compositor will rely on the GPU's hardware sampler to perform the sRGB-to-linear conversion.

If your application prefers rendering to a linear format (e.g. OVR_FORMAT_R8G8B8A8_UNORM) while handling the linear-to-gamma conversion via SPIRV code, the application must still request the corresponding sRGB format and also use `ovrTextureMisc_DX_Typeless` in the `Flag` field of `ovrTextureSwapChainDesc`. This allows the application to create a `RenderTargetView` in linear format, while allowing the compositor to treat it as sRGB. Failure to do this will result in unexpected gamma-curve artifacts. The `ovrTextureMisc_DX_Typeless` flag for depth buffer formats (e.g. OVR_FORMAT_D32_FLOAT) is ignored as they are always converted to be typeless.

In addition to sRGB, these concepts also apply to the mirror texture creation. For more information, refer to the function documentation provided for `ovr_CreateMirrorTextureDX`, `ovr_CreateMirrorTextureGL`, and `ovr_CreateMirrorTextureWithOptionsVk` for D3D, OpenGL, and Vulkan, respectively.

Frame Rendering

Frame rendering typically involves several steps: obtaining predicted eye poses based on the headset tracking pose, rendering the view for each eye and, finally, submitting eye textures to the compositor through `ovr_WaitToBeginFrame`, `ovr_BeginFrame`, and `ovr_EndFrame`. After the frame is submitted, the Oculus compositor handles distortion, timewarp, and GPU synchronization before presenting it to the headset.

Before rendering frames, it is helpful to initialize some data structures that can be shared across frames. As an example, we query eye descriptors and initialize the layer structure outside of the rendering loop:

```
 // Initialize VR structures, filling out description.
ovrEyeRenderDesc eyeRenderDesc[2];
ovrPosef      hmdToEyeViewPose[2];
ovrHmdDesc hmdDesc = ovr_GetHmdDesc(session);
eyeRenderDesc[0] = ovr_GetRenderDesc(session, ovrEye_Left, hmdDesc.DefaultEyeFov[0]);
eyeRenderDesc[1] = ovr_GetRenderDesc(session, ovrEye_Right, hmdDesc.DefaultEyeFov[1]);
hmdToEyeViewPose[0] = eyeRenderDesc[0].HmdToEyePose;
hmdToEyeViewPose[1] = eyeRenderDesc[1].HmdToEyePose;
```

```
// Initialize our single full screen Fov layer.
ovrLayerEyeFov layer;
layer.Header.Type      = ovrLayerType_EyeFov;
layer.Header.Flags     = 0;
layer.ColorTexture[0]  = textureSwapChain;
layer.ColorTexture[1]  = textureSwapChain;
layer.Fov[0]           = eyeRenderDesc[0].Fov;
layer.Fov[1]           = eyeRenderDesc[1].Fov;
layer.Viewport[0]      = Recti(0, 0,                bufferSize.w / 2, bufferSize.h);
layer.Viewport[1]      = Recti(bufferSize.w / 2, 0, bufferSize.w / 2, bufferSize.h);
// ld.RenderPose and ld.SensorSampleTime are updated later per frame.
```

This code example first gets rendering descriptors for each eye, given the chosen FOV. The returned `ovrEyeRenderDesc` structure contains useful values for rendering, including the `HmdToEyePose` for each eye. Eye view offsets are used later to adjust for eye separation.

The code also initializes the `ovrLayerEyeFov` structure for a full screen layer. Starting with Oculus SDK 0.6, frame submission uses layers to composite multiple view images or texture quads on top of each other. This example uses a single layer to present a VR scene. For this purpose, we use `ovrLayerEyeFov`, which describes a dual-eye layer that covers the entire eye field of view. Since we are using the same texture set for both eyes, we initialize both eye color textures to `pTextureSet` and configure viewports to draw to the left and right sides of this shared texture, respectively.



Note: Although it is often enough to initialize viewports once in the beginning, specifying them as a part of the layer structure that is submitted every frame allows applications to change render target size dynamically, if desired. This is useful for optimizing rendering performance.

After setup completes, the application can run the rendering loop. First, we need to get the eye poses to render the left and right views.

```
// Get both eye poses simultaneously, with IPD offset already included.
double displayMidpointSeconds = GetPredictedDisplayTime(session, 0);
ovrTrackingState hmdState = ovr_GetTrackingState(session, displayMidpointSeconds, ovrTrue);
ovr_CalcEyePoses(hmdState.HeadPose.ThePose, hmdToEyeViewPose, layer.RenderPose);
```

In VR, rendered eye views depend on the headset position and orientation in the physical space, tracked with the help of internal IMU and external sensors. Prediction is used to compensate for the latency in the system, giving the best estimate for where the headset will be when the frame is displayed on the headset. In the Oculus SDK, this tracked, predicted pose is reported by `ovr_GetTrackingState`.

To do accurate prediction, `ovr_GetTrackingState` needs to know when the current frame will actually be displayed. The code above calls `GetPredictedDisplayTime` to obtain `displayMidpointSeconds` for the current frame, using it to compute the best predicted tracking state. The head pose from the tracking state is then passed to `ovr_CalcEyePoses` to calculate correct view poses for each eye. These poses are stored directly into the `layer.RenderPose[2]` array. With eye poses ready, we can proceed onto the actual frame rendering.

```
if (isVisible)
{
    // Get next available index of the texture swap chain
 int currentIndex = 0;
    ovr_GetTextureSwapChainCurrentIndex(session, textureSwapChain, &currentIndex);
    ovrResult       result = ovr_WaitToBeginFrame(session, 0);

    // Clear and set up render-target.
    DIRECTX.SetAndClearRenderTarget(pTexRtv[currentIndex], pEyeDepthBuffer);

    // Render Scene to Eye Buffers
    result = ovr_BeginFrame(session, 0);
    for (int eye = 0; eye < 2; eye++)
    {
        // Get view and projection matrices for the Rift camera
        Vector3f pos = originPos + originRot.Transform(layer.RenderPose[eye].Position);
        Matrix4f rot = originRot * Matrix4f(layer.RenderPose[eye].Orientation);
```

```
        Vector3f finalUp      = rot.Transform(Vector3f(0, 1, 0));
        Vector3f finalForward = rot.Transform(Vector3f(0, 0, -1));
        Matrix4f view         = Matrix4f::LookAtRH(pos, pos + finalForward, finalUp);

        Matrix4f proj = ovrMatrix4f_Projection(layer.Fov[eye], 0.2f, 1000.0f, 0);
        // Render the scene for this eye.
        DIRECTX.SetViewport(layer.Viewport[eye]);
        roomScene.Render(proj * view, 1, 1, 1, 1, true);
    }

 // Commit the changes to the texture swap chain
 ovr_CommitTextureSwapChain(session, textureSwapChain);
}

// Submit frame with one layer we have.
ovrLayerHeader* layers = &layer.Header;
result = ovr_EndFrame(session, 0, nullptr, &layers, 1);
isVisible = (result == ovrSuccess);
```

This code takes a number of steps to render the scene:

- It applies the texture as a render target and clears it for rendering. In this case, the same texture is used for both eyes.
- The code then computes view and projection matrices and sets viewport scene rendering for each eye. In this example, view calculation combines the original pose (`originPos` and `originRot` values) with the new pose computed based on the tracking state and stored in the layer. There original values can be modified by input to move the player within the 3D world.
- After texture rendering is complete, we call ovr_EndFrame to pass frame data to the compositor. From this point, the compositor takes over by accessing texture data through shared memory, distorting it, and presenting it on the Rift.

`ovr_EndFrame` returns once the submitted frame is queued up and the runtime is available to accept a new frame. When successful, its return value is either `ovrSuccess` or `ovrSuccess_NotVisible`.

`ovrSuccess_NotVisible` is returned if the frame wasn't actually displayed, which can happen when the VR application loses focus. Our sample code handles this case by updating the `isVisible` flag, checked by the rendering logic. While frames are not visible, rendering should be paused to eliminate unnecessary GPU load.

If you receive `ovrError_DisplayLost`, the device was removed and the session is invalid. Release the shared resources (ovr_DestroyTextureSwapChain), destroy the session (ovr_Destroy), recreate it (ovr_Create), and create new resources (ovr_CreateTextureSwapChainXXX). The application's existing private graphics resources do not need to be recreated unless the new ovr_Create call returns a different `GraphicsLuid`.

Frame Timing

The Oculus SDK reports frame timing information through the `ovr_GetPredictedDisplayTime` function, relying on the application-provided frame index to ensure correct timing is reported across different threads.

Accurate frame and sensor timing are required for accurate head motion prediction, which is essential for a good VR experience. Prediction requires knowing exactly when in the future the current frame will appear on the screen. If we know both sensor and display scanout times, we can predict the future head pose and improve image stability. Computing these values incorrectly can lead to under or over-prediction, degrading perceived latency, and potentially causing overshoot “wobbles”.

To ensure accurate timing, the Oculus SDK uses absolute system time, stored as a double, to represent sensor and frame timing values. The current absolute time is returned by `ovr_GetTimeInSeconds`. Current time should rarely be used, however, since simulation and motion prediction will produce better results when relying on the timing values returned by `ovr_GetPredictedDisplayTime`. This function has the following signature:

```
ovr_GetPredictedDisplayTime(ovrSession session, long long frameIndex);
```

The `frameIndex` argument specifies which application frame we are rendering. Applications that make use of multi-threaded rendering must keep an internal frame index and manually increment it, passing it across threads along with frame data to ensure correct timing and prediction. The same `frameIndex` that was used to obtain timing for the framevalue must be passed to `ovr_WaitToBeginFrame`, `ovr_BeginFrame`, and `ovr_EndFrame`. The details of multi-threaded timing are covered in the next section, *Rendering on Different Threads* on page 24.

A special `frameIndex` value of 0 can be used in both functions to request that the SDK keep track of frame indices automatically. However, this only works when all frame timing requests and render submission is done on the same thread.

Rendering on Different Threads

In some engines, render processing is distributed across more than one thread.

For example, one thread may perform culling and render setup for each object in the scene (we'll call this the "main" thread), while a second thread makes the actual D3D or OpenGL API calls (we'll call this the "render" thread). Both of these threads may need accurate estimates of frame display time, so as to compute best possible predictions of head pose.

The asynchronous nature of this approach makes this challenging: while the render thread is rendering a frame, the main thread might be processing the next frame. This parallel frame processing may be out of sync by exactly one frame or a fraction of a frame, depending on game engine design. If we used the default global state to access frame timing, the result of `GetPredictedDisplayTime` could either be off by one frame depending which thread the function is called from, or worse, could be randomly incorrect depending on how threads are scheduled. To address this issue, previous section introduced the concept of `frameIndex` that is tracked by the application and passed across threads along with frame data.

For multi-threaded rendering result to be correct, the following must be true: (a) pose prediction, computed based on frame timing, must be consistent for the same frame regardless of which thread it is accessed from; and (b) eye poses that were actually used for rendering must be passed into `ovr_EndFrame`, along with the frame index. (This must occur after calling `ovr_WaitToBeginFrame` and `ovr_BeginFrame`.)

Here is a summary of steps you can take to ensure this is the case:

1. The main thread needs to assign a frame index to the current frame being processed for rendering. It would increment this index each frame and pass it to `GetPredictedDisplayTime` to obtain the correct timing for pose prediction.
2. The main thread should call the thread safe function `ovr_GetTrackingState` with the predicted time value. It can also call `ovr_CalcEyePoses` if necessary for rendering setup.
3. Main thread needs to pass the current frame index and eye poses to the render thread, along with any rendering commands or frame data it needs.
4. When the rendering commands executed on the render thread, developers need to make sure these things hold:
    a. The actual poses used for frame rendering are stored into the `RenderPose` for the layer.
    b. The same value of `frameIndex` as was used on the main thead is passed into `ovr_BeginFrame` and `ovr_EndFrame`.

The following code illustrates this in more detail:

```
void MainThreadProcessing()
{
    frameIndex++;
    ovrResult result = ovr_WaitToBeginFrame(session, frameIndex);

    // Ask the API for the times when this frame is expected to be displayed.
    double frameTiming = GetPredictedDisplayTime(session, frameIndex);

    // Get the corresponding predicted pose state.
```

```
    ovrTrackingState state = ovr_GetTrackingState(session, frameTiming, ovrTrue);
    ovrPosef         eyePoses[2];
    ovr_CalcEyePoses(state.HeadPose.ThePose, hmdToEyeViewOffset, eyePoses);

    SetFrameHMDData(frameIndex, eyePoses);

    // Do render pre-processing for this frame.
    ...
}

void RenderThreadProcessing()
{
    int      frameIndex;
    ovrPosef eyePoses[2];
    ovrResult       result = ovr_BeginFrame(session, frameIndex);

    GetFrameHMDData(&frameIndex, eyePoses);
    layer.RenderPose[0] = eyePoses[0];
    layer.RenderPose[1] = eyePoses[1];

    // Execute actual rendering to eye textures.
    ...

   // Submit frame with the one layer we have.
   ovrLayerHeader* layers = &layer.Header;
   result = ovr_EndFrame(session, frameIndex, nullptr, &layers, 1);
}
```

The Oculus SDK also supports Direct3D 12, which allows submitting rendering work to the GPU from multiple CPU threads. When the application calls `ovr_CreateTextureSwapChainDX`, the Oculus SDK caches off the ID3D12CommandQueue provided by the caller for future usage. As the application calls `ovr_EndFrame`, the SDK drops a fence on the cached ID3D12CommandQueue to know exactly when a given set of eye-textures are ready for the SDK compositor.

For a given application, using a single ID3D12CommandQueue on a single thread is the easiest. But,it might also split the CPU rendering workload for each eye-texture pair or push non-eye-texture rendering work, such as shadows, reflection maps, and so on, onto different command queues. If the application populates and executes command lists from multiple threads, it will also have to make sure that the ID3D12CommandQueue provided to the SDK is the single join-node for the eye-texture rendering work executed through different command queues.

Layers

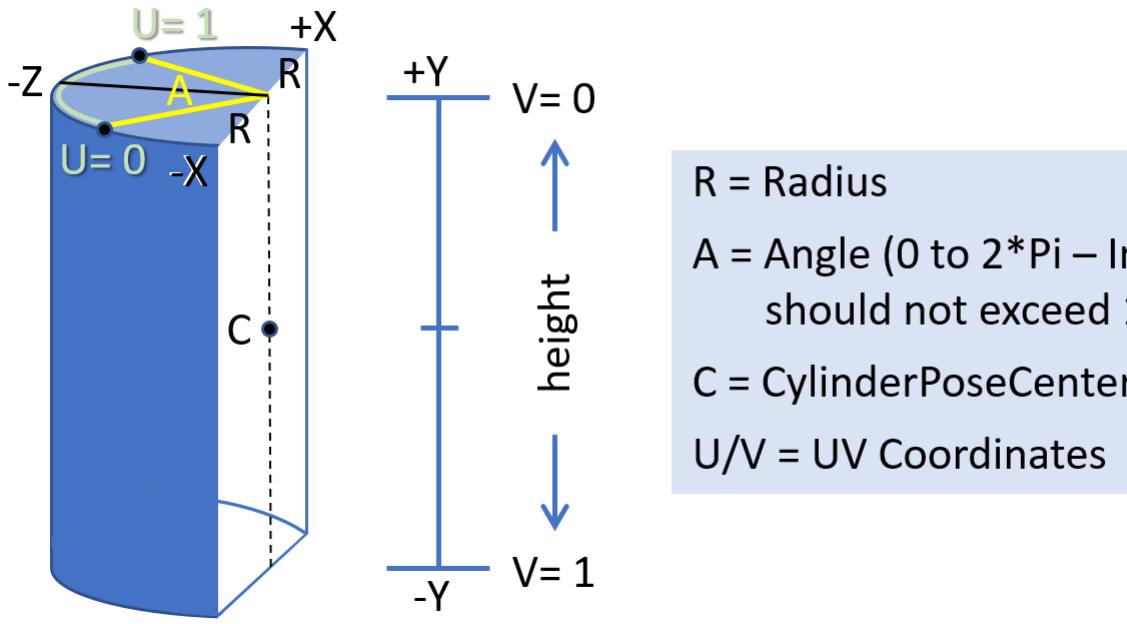
Similar to the way a monitor view can be composed of multiple windows, the display on the headset can be composed of multiple layers. Typically at least one of these layers will be a view rendered from the user's virtual eyeballs, but other layers may be HUD layers, cubemap layers, information panels, text labels attached to items in the world, aiming reticles, and so on.

Each layer can have a different resolution, can use a different texture format, can use a different field of view or size, and might be in mono or stereo. The application can also be configured to not update a layer's texture if the information in it has not changed. For example, it might not update if the text in an information panel has not changed since last frame or if the layer is a picture-in-picture view of a video stream with a low framerate. Applications can supply mipmapped textures to a layer and, together with a high-quality distortion mode, this is very effective at improving the readability of text panels.

Every frame, all active layers are composited from back to front using pre-multiplied alpha blending. Layer 0 is the furthest layer, layer 1 is on top of it, and so on; there is no depth-buffer intersection testing of layers, even if a depth-buffer is supplied.

A powerful feature of layers is that each can be a different resolution. This allows an application to scale to lower performance systems by dropping resolution on the main eye-buffer render that shows the virtual world, but keeping essential information, such as text or a map, in a different layer at a higher resolution.

There are several layer types available:

EyeFov	The standard "eye buffer" familiar from previous SDKs, which is typically a stereo view of a virtual scene rendered from the position of the user's eyes. Although eye buffers can be mono, this can cause discomfort. Previous SDKs had an implicit field of view (FOV) and viewport; these are now supplied explicitly and the application can change them every frame, if desired.
Quad	A monoscopic image that is displayed as a rectangle at a given pose and size in the virtual world. This is useful for heads-up-displays, text information, object labels and so on. By default the pose is specified relative to the user's real-world space and the quad will remain fixed in space rather than moving with the user's head or body motion. For head-locked quads, use the `ovrLayerFlag_HeadLocked` flag as described below.
Cubemap	A cubemap consists of six rectangles. These rectangles are placed around the user, as if the user is sitting inside of a room that is cube shaped. Each wall is a texture that your application submits. The cubemap appears to be at an infinite distance, and essentially is the background behind all other objects that your application renders. The cubemap does not look like a cube to the user. Rather, it simply appears to be the background, at an infinite distance. For example, you can use cubmaps to create the sky that will appear behind all the buildings in your VR experience. You don't need to handle occlusion by objects in the foreground. You can simply setup the cubemap, and it will appear in the background everywhere in your scene.
EyeMatrix	The EyeMatrix layer type is similar to the EyeFov layer type and is provided to assist compatibility with Gear VR applications. For more information, refer to the Mobile SDK documentation.
Cylindrical	You can use Cylinder layers to create curved quads, instead of flat quads. ovrLayerCylinder describes a layer of type `ovrLayerType_Cylinder` which is a single cylinder that is positioned relative to a recentered origin (represented by C in the illustration): **Figure 6: Cylindrical Layer Parameters**  This type of layer represents a single object placed in the world and not a stereo view of the world itself.

	Only the interior surface of the cylinder is visible. You should only use cylinder layers when the user cannot leave the extents of the cylinder. Artifacts may appear when viewing the cylinder's exterior surface. In addition, while the interface supports an Angle (A) that ranges from 0 to 2*Pi, the angle should always be less than 1.9*PI to avoid artifacts where the cylinder edges converge. For more information on using this feature, see `ovrLayerCylinder` in the reference documentation.
Depth Buffers	This layer specifies a monoscopic or stereoscopic view, with depth textures in addition to color textures. This layer is implemented by the `ovrLayerEyeFovDepth` struct. It is equivalent to `ovrLayerEyeFov`, but with the addition of `DepthTexture` and `ProjectionDesc`. Depth buffers are typically used to support positional time warp.
Disabled	Ignored by the compositor, disabled layers do not cost performance. We recommend that applications perform basic frustum-culling and disable layers that are out of view. However, there is no need for the application to repack the list of active layers tightly together when turning one layer off; disabling it and leaving it in the list is sufficient. Equivalently, the pointer to the layer in the list can be set to null.

Each layer style has a corresponding member of the `ovrLayerType` enum, and an associated structure holding the data required to display that layer. For example, the EyeFov layer is type number `ovrLayerType_EyeFov` and is described by the data in the structure `ovrLayerEyeFov`. These structures share a similar set of parameters, though not all layer types require all parameters:

Parameter	Type	Description
`Header.Type`	`enum ovrLayerType`	Must be set by all layers to specify what type they are.
`Header.Flags`	A bitfield of `ovrLayerFlags`	See below for more information.
`ColorTexture`	`TextureSwapChain`	Provides color and translucency data for the layer. Layers are blended over one another using premultiplied alpha. This allows them to express either lerp-style blending, additive blending, or a combination of the two. Layer textures must be RGBA or BGRA formats and might have mipmaps, but cannot be arrays, cubes, or have MSAA. If the application desires to do MSAA rendering, then it must resolve the intermediate MSAA color texture into the layer's non-MSAA ColorTexture.
`Viewport`	`ovrRecti`	The rectangle of the texture that is actually used, specified in 0-1 texture "UV" coordinate space (not pixels). In theory, texture data outside this region is not visible in the layer. However, the usual caveats about texture sampling apply, especially with mipmapped textures. It is good practice to leave a border of RGBA(0,0,0,0) pixels around the displayed region to avoid "bleeding," especially between two eye buffers packed side by side into the same texture. The size of the border depends on the exact usage case, but around 8 pixels seems to work well in most cases.
`Fov`	`ovrFovPort`	The field of view used to render the scene in an Eye layer type. Note this does not control the HMD's display, it simply tells the compositor what FOV was used to render the texture data in the layer - the compositor will then

Parameter	Type	Description
		adjust appropriately to whatever the actual user's FOV is. Applications may change FOV dynamically for special effects. Reducing FOV may also help with performance on slower machines, though typically it is more effective to reduce resolution before reducing FOV.
`RenderPose`	`ovrPosef`	The camera pose the application used to render the scene in an Eye layer type. This is typically predicted by the SDK and application using the `ovr_GetTrackingState` and `ovr_CalcEyePoses` functions. The difference between this pose and the actual pose of the eye at display time is used by the compositor to apply timewarp to the layer.
`SensorSampleTime`	`double`	The absolute time when the application sampled the tracking state. The typical way to acquire this value is to have an `ovr_GetTimeInSeconds` call right next to the `ovr_GetTrackingState` call. The SDK uses this value to report the application's motion-to-photon latency in the Performance HUD. If the application has more than one `ovrLayerType_EyeFov` layer submitted at any given frame, the SDK scrubs through those layers and selects the timing with the lowest latency. In a given frame, if no `ovrLayerType_EyeFov` layers are submitted, the SDK will use the point in time when `ovr_GetTrackingState` was called with the `latencyMarker`set to `ovrTrue` as the substitute application motion-to-photon latency time.
`QuadPoseCenter`	`ovrPosef`	Specifies the orientation and position of the center point of a Quad layer type. The supplied direction is the vector perpendicular to the quad. The position is in real-world meters (not the application's virtual world, the actual world the user is in) and is relative to the "zero" position set by `ovr_RecenterTrackingOrigin` or `ovr_SpecifyTrackingOrigin` unless the `ovrLayerFlag_HeadLocked` flag is used.
`QuadSize`	`ovrVector2f`	Specifies the width and height of a Quad layer type. As with position, this is in real-world meters.

Layers that take stereo information (all those except Quad layer types) take two sets of most parameters, and these can be used in three different ways:

- Stereo data, separate textures—the app supplies a different `ovrTextureSwapChain` for the left and right eyes, and a viewport for each.
- Stereo data, shared texture—the app supplies the same `ovrTextureSwapChain` for both left and right eyes, but a different viewport for each. This allows the application to render both left and right views to the same texture buffer. Remember to add a small buffer between the two views to prevent "bleeding", as discussed above.
- Mono data—the app supplies the same `ovrTextureSwapChain` for both left and right eyes, and the same viewport for each.

Texture and viewport sizes may be different for the left and right eyes, and each can even have different fields of view. However beware of causing stereo disparity and discomfort in your users.

The `Header.Flags` field available for all layers is a logical-or of the following:

- `ovrLayerFlag_HighQuality`—enables 4x anisotropic sampling in the compositor for this layer. This can provide a significant increase in legibility, especially when used with a texture containing mipmaps; this is recommended for high-frequency images such as text or diagrams and when used with the Quad layer types. For Eye layer types, it will also increase visual fidelity towards the periphery, or when feeding in textures that have more than the 1:1 recommended pixel density. For best results, when creating mipmaps for the textures associated to the particular layer, make sure the texture sizes are a power of 2. However, the application does not need to render to the whole texture; a viewport that renders to the recommended size in the texture will provide the best performance-to-quality ratios.
- `ovrLayerFlag_TextureOriginAtBottomLeft`—the origin of a layer's texture is assumed to be at the top-left corner. However, some engines (particularly those using OpenGL) prefer to use the bottom-left corner as the origin, and they should use this flag.
- `ovrLayerFlag_HeadLocked`—Most layer types have their pose orientation and position specified relative to the "zero position" defined by calling `ovr_RecenterTrackingOrigin`. However the app may wish to specify a layer's pose relative to the user's face. When the user moves their head, the layer follows. This is useful for reticles used in gaze-based aiming or selection. This flag may be used for all layer types, though it has no effect when used on the Direct type.

At the end of each frame, after rendering to whichever `ovrTextureSwapChain` the application wants to update and calling `ovr_CommitTextureSwapChain`, the data for each layer is put into the relevant `ovrLayerEyeFov` / `ovrLayerQuad` / `ovrLayerDirect` structure. The application then creates a list of pointers to those layer structures, specifically to the `Header` field which is guaranteed to be the first member of each structure. Then the application builds a `ovrViewScaleDesc` struct with the required data, and calls the `ovr_WaitToBeginFrame`, `ovr_BeginFrame`, and `ovr_EndFrame` functions.

```
ovrResult result = ovr_WaitToBeginFrame(Session, 0);
result = ovr_BeginFrame(Session, 0);
// Create eye layer.
ovrLayerEyeFov eyeLayer;
eyeLayer.Header.Type    = ovrLayerType_EyeFov;
eyeLayer.Header.Flags   = 0;
for ( int eye = 0; eye < 2; eye++ )
{
 eyeLayer.ColorTexture[eye] = EyeBufferSet[eye];
 eyeLayer.Viewport[eye]     = EyeViewport[eye];
 eyeLayer.Fov[eye]          = EyeFov[eye];
 eyeLayer.RenderPose[eye]   = EyePose[eye];
}

// Create HUD layer, fixed to the player's torso
ovrLayerQuad hudLayer;
hudLayer.Header.Type    = ovrLayerType_Quad;
hudLayer.Header.Flags   = ovrLayerFlag_HighQuality;
hudLayer.ColorTexture   = TheHudTextureSwapChain;
// 50cm in front and 20cm down from the player's nose,
// fixed relative to their torso.
hudLayer.QuadPoseCenter.Position.x =  0.00f;
hudLayer.QuadPoseCenter.Position.y = -0.20f;
hudLayer.QuadPoseCenter.Position.z = -0.50f;
hudLayer.QuadPoseCenter.Orientation.x = 0;
hudLayer.QuadPoseCenter.Orientation.y = 0;
hudLayer.QuadPoseCenter.Orientation.z = 0;
hudLayer.QuadPoseCenter.Orientation.w = 1;
// HUD is 50cm wide, 30cm tall.
hudLayer.QuadSize.x = 0.50f;
hudLayer.QuadSize.y = 0.30f;
// Display all of the HUD texture.
hudLayer.Viewport.Pos.x = 0.0f;
hudLayer.Viewport.Pos.y = 0.0f;
hudLayer.Viewport.Size.w = 1.0f;
hudLayer.Viewport.Size.h = 1.0f;

// The list of layers.
ovrLayerHeader *layerList[2];
layerList[0] = &eyeLayer.Header;
layerList[1] = &hudLayer.Header;

// Set up positional data.
ovrViewScaleDesc viewScaleDesc;
viewScaleDesc.HmdSpaceToWorldScaleInMeters = 1.0f;
```

```
viewScaleDesc.HmdToEyeViewOffset[0] = HmdToEyePose[0];
viewScaleDesc.HmdToEyeViewOffset[1] = HmdToEyePose[1];

result = ovr_EndFrame(Session, 0, &viewScaleDesc, layerList, 2);
```

The compositor performs timewarp, distortion, and chromatic aberration correction on each layer separately before blending them together. The traditional method of rendering a quad to the eye buffer involves two filtering steps (once to the eye buffer, then once during distortion). Using layers, there is only a single filtering step between the layer image and the final framebuffer. This can provide a substantial improvement in text quality, especially when combined with mipmaps and the `ovrLayerFlag_HighQuality` flag.

One current disadvantage of layers is that no post-processing can be performed on the final composited image, such as soft-focus effects, light-bloom effects, or the Z intersection of layer data. Some of these effects can be performed on the contents of the layer with similar visual results.

Calling `ovr_EndFrame` queues the layers for display, and transfers control of the committed textures inside the `ovrTextureSwapChains` to the compositor. It is important to understand that these textures are being shared (rather than copied) between the application and the compositor threads, and that composition does not necessarily happen at the time `ovr_EndFrame` is called, so care must be taken. To continue rendering into a texture swap chain the application should always get the next available index with `ovr_GetTextureSwapChainCurrentIndex` before rendering into it. For example:

```
ovrResult result = ovr_WaitToBeginFrame(Hmd, 0);
result = ovr_BeginFrame(Hmd, 0);
// Create two TextureSwapChains to illustrate.
ovrTextureSwapChain eyeTextureSwapChain;
ovr_CreateTextureSwapChainDX ( ... &eyeTextureSwapChain );
ovrTextureSwapChain hudTextureSwapChain;
ovr_CreateTextureSwapChainDX ( ... &hudTextureSwapChain );

// Set up two layers.
ovrLayerEyeFov eyeLayer;
ovrLayerEyeFov hudLayer;
eyeLayer.Header.Type = ovrLayerType_EyeFov;
eyeLayer...etc... // set up the rest of the data.
hudLayer.Header.Type = ovrLayerType_Quad;
hudLayer...etc... // set up the rest of the data.

// the list of layers
ovrLayerHeader *layerList[2];
layerList[0] = &eyeLayer.Header;
layerList[1] = &hudLayer.Header;

// Each frame...
int currentIndex = 0;
ovr_GetTextureSwapChainCurrentIndex(... eyeTextureSwapChain, &currentIndex);
// Render into it. It is recommended the app use ovr_GetTextureSwapChainBufferDX for each index on
 texture chain creation to cache
// textures or create matching render target views. Each frame, the currentIndex value returned can
 be used to index directly into that.
ovr_CommitTextureSwapChain(... eyeTextureSwapChain);

ovr_GetTextureSwapChainCurrentIndex(... hudTextureSwapChain, &currentIndex);
// Render into it. It is recommended the app use ovr_GetTextureSwapChainBufferDX for each index on
 texture chain creation to cache
// textures or create matching render target views. Each frame, the currentIndex value returned can
 be used to index directly into that.
ovr_CommitTextureSwapChain(... hudTextureSwapChain);

eyeLayer.ColorTexture[0] = eyeTextureSwapChain;
eyeLayer.ColorTexture[1] = eyeTextureSwapChain;
hudLayer.ColorTexture = hudTextureSwapChain;

result = ovr_EndFrame(Hmd, 0, nullptr, layerList, 2);
```

Rectilinear Capture

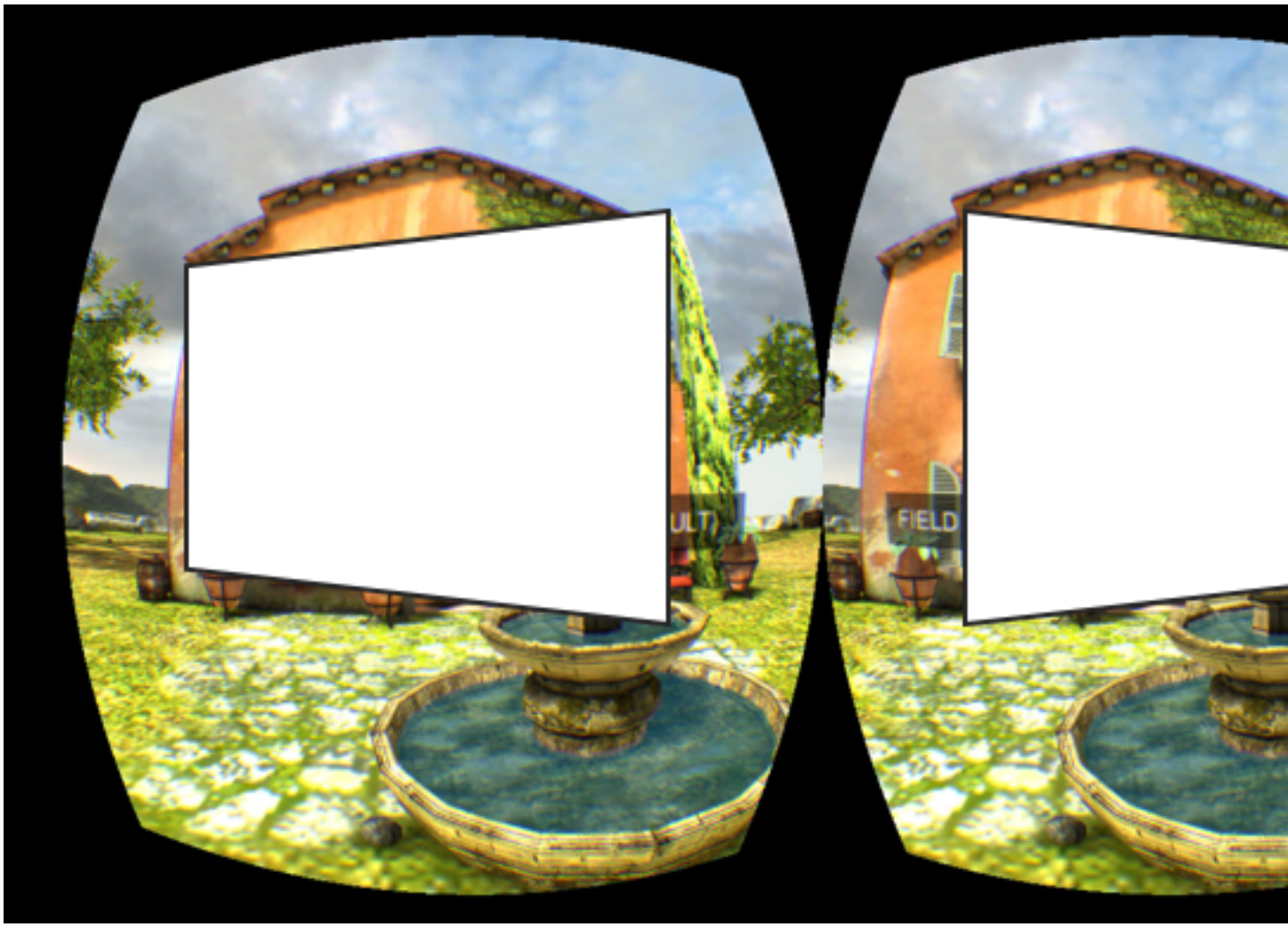
This feature enables your applications to obtain screenshots (at 90 frames per second) of a single non-distorted image that corresponds to what the user is seeing in the VR experience. You might use this feature to mirror the VR experience to an external monitor.

Working with HMD Eye Poses

In the Oculus PC SDK, prior to version 1.17, eye poses only had three degrees-of-freedom (DOF), i.e. only translation. Eye poses were specified in the `HmdToEyeOffset`vector provided by the `ovr_GetRenderDesc` function. Starting with version 1.17, `HmdToEyeOffset` has been renamed to `HmdToEyePose` using the type `ovrPosef` which contains a `Position` and `Orientation`, effectively giving eye poses six degrees-of-freedom. This means that each eye's render frustum can now be rotated away from the HMD's orientation, in addition to being translated by the SDK. Because of this, the eye frustums' axes are no longer guaranteed to be parallel to each other or to the HMD's orientation axes. This generalization provides greater freedom to the SDK in defining the HMD geometry. But it also means that, as a VR app developer, you need to be more careful about your previous assumptions, especially when it comes to rendering.

Here are some pointers to make sure your VR app is correctly using `HmdToEyePose`:

- If your VR app needs the vector translation value of the (pre-version 1.17) `HmdToEyeOffset`, you can use `HmdToEyePose.Position` instead. However, unless you are absolutely sure about what you are doing, there is a good chance you actually want to treat `HmdToEyePose` as a whole transform, rather than separate out `Orientation` from `Position`.
- Prior to PC-SDK version 1.17, rendering a 2D quad flat across the screen (e.g. a rectangle) would have always been acceptable. But with the possibility of rotating each eye frustum independently, your VR app will need to incorporate each eye's orientation into the transformation of the quad so that the quad is rendered with the correct perspective in each eye. Here is a (somewhat exaggerated) example:



The idea is to orient the quad such that it appears to use either the "center-eye" orientation, or the HMD orientation. This also applies to other screen-aligned quads, such as 3D splash screens, or particle effects such as large flip-book smoke quads. Except for particles, avoid rendering such quads natively; prefer using `ovrLayerQuad` instead.

- Some VR apps generate a single monoscopic camera frustum from the `ovrFovPort` structures of both eyes, in order to take advantage of various rendering optimizations. This is normally done by using an `ovrFovPort`, which takes the maximum of the `ovrFovPort` values for both eyes, on all four sides of the frustum. Before generating the monoscopic frustum this way however, be sure to remove any potential rotation from the `ovrFovPort` values by calling `FovPort::Uncant`, which is located in the `ovr_math.h` header. See the `OculusWorldDemo` sample code, to see how to use `FovPort::Uncant`.

Asynchronous TimeWarp

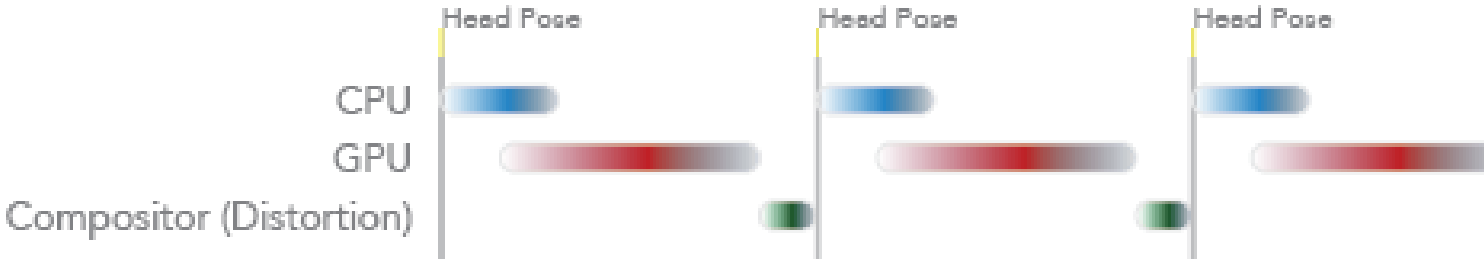
Asynchronous TimeWarp (ATW) is a technique for reducing latency and judder in VR applications and experiences.

In a basic VR game loop, the following occurs:

1. The software requests your head position.
2. The CPU processes the scene for each eye.
3. The GPU renders the scenes.
4. The Oculus Compositor applies distortion and displays the scenes on the headset.

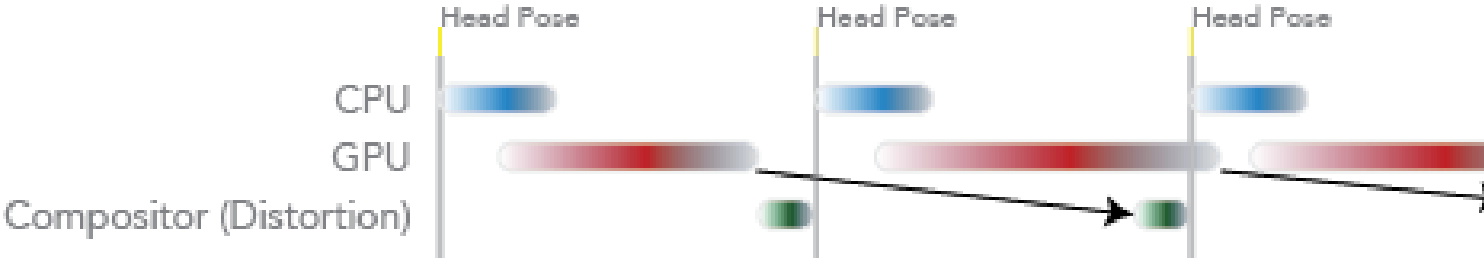
The following shows a basic example of a game loop:

Figure 7: Basic Game Loop



When frame rate is maintained, the experience feels real and is enjoyable. When it doesn't happen in time, the previous frame is shown which can be disorienting. The following graphic shows an example of judder during the basic game loop:

Figure 8: Basic Game Loop with Judder



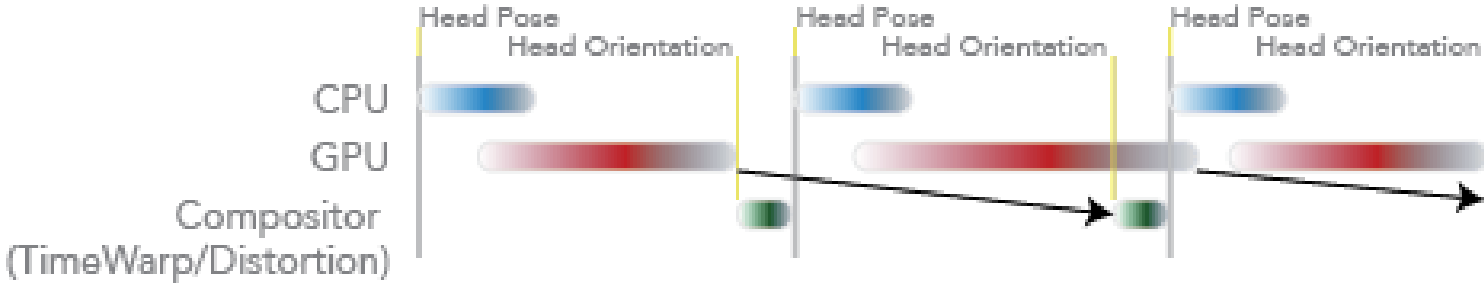
When you move your head and the world doesn't keep up, this can be jarring and break immersion.

ATW is a technique that shifts the rendered image slightly to adjust for changes in head movement. Although the image is modified, your head does not move much, so the change is slight.

Additionally, to smooth issues with the user's computer, game design or the operating system, ATW can help fix "potholes" or moments when the frame rate unexpectedly drops.

The following graphic shows an example of frame drops when ATW is applied:

Figure 9: Game Loop with ATW



At the refresh interval, the Compositor applies TimeWarp to the last rendered frame. As a result, a TimeWarped frame will always be shown to the user, regardless of frame rate. If the frame rate is very bad, flicker will be noticeable at the periphery of the display. But, the image will still be stable.

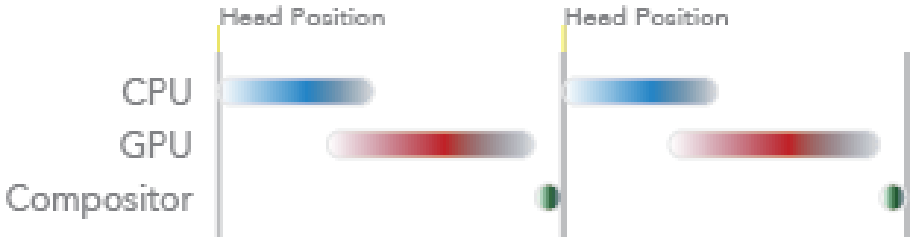
ATW is automatically applied by the Oculus Compositor; you do not need to enable or tune it. However, although ATW reduces latency, make sure that your application or experience makes frame rate.

Adaptive Queue Ahead

To improve CPU and GPU parallelism and increase the amount of time that the GPU has to process a frame, the SDK automatically applies queue ahead up to 1 frame.

Without queue ahead, the CPU begins processing the next frame immediately after the previous frame displays. After the CPU finishes, the GPU processes the frame, the compositor applies distortion, and the frame is displayed to the user. The following graphic shows CPU and GPU utilization without queue ahead:

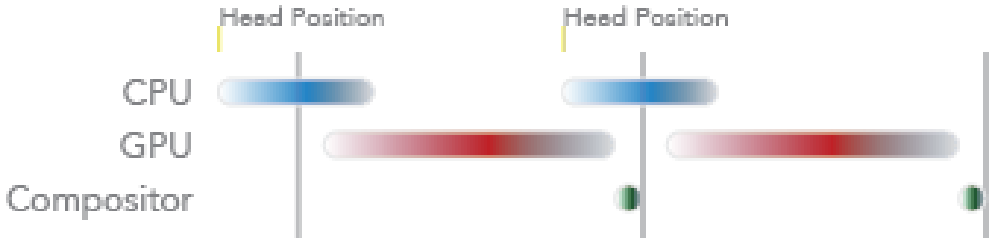
Figure 10: CPU and GPU Utilization without Queue Ahead



If the GPU cannot process the frame in time for display, the previous frame displays. This results in judder.

With queue ahead, the CPU can start earlier; this provides the GPU more time to process the frame. The following graphic shows CPU and GPU utilization with queue ahead:

Figure 11: CPU and GPU Utilization with Queue Ahead



Advanced Rendering Configuration

By default, the SDK generates configuration values that optimize for rendering quality.

It also provides a degree of flexibility. For example, you can make changes when creating render target textures.

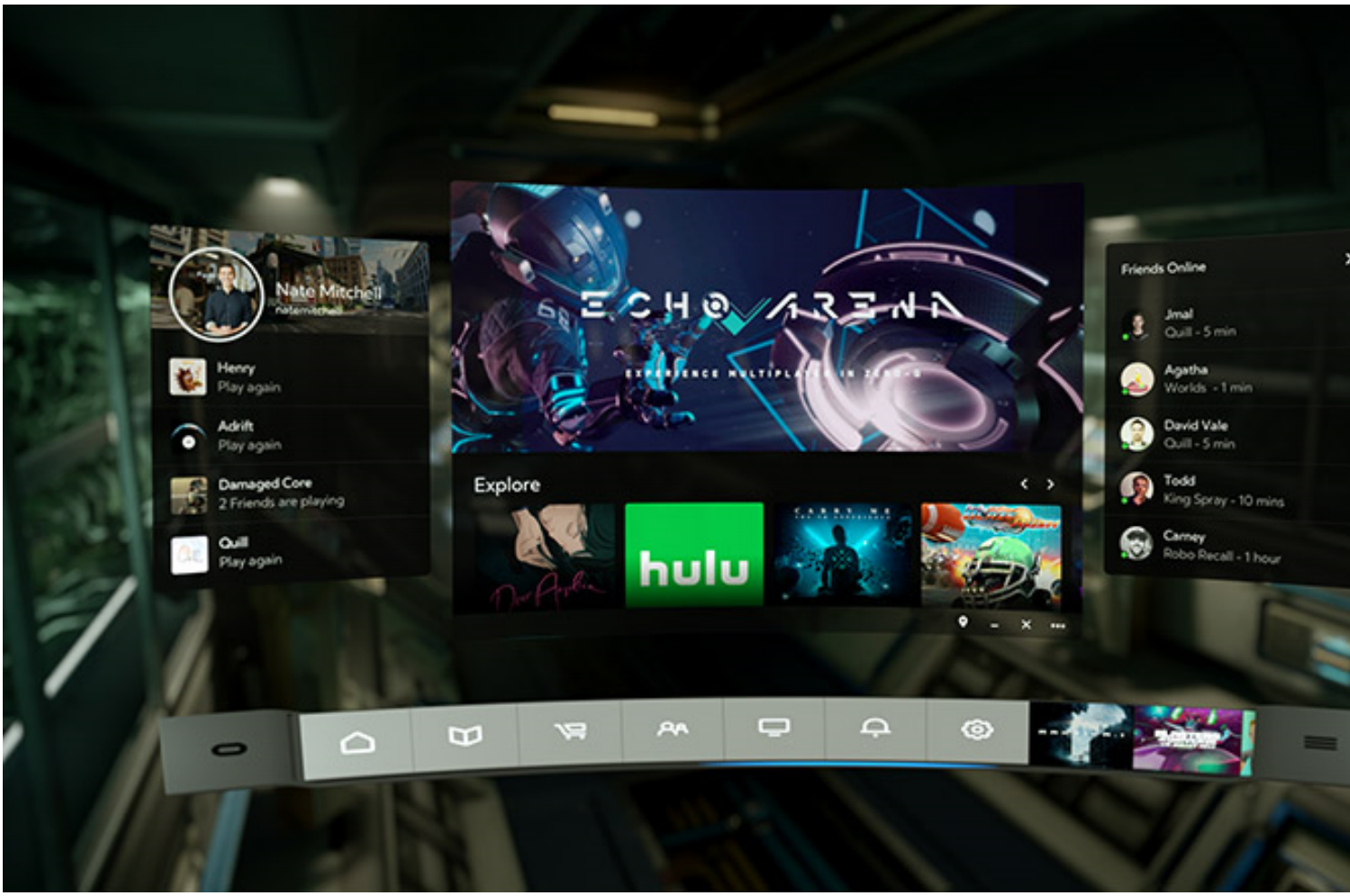
This section discusses changes you can make when choosing between rendering quality and performance, or if the engine you are using imposes constraints.

Using Oculus Dash

This section introduces Oculus Dash for the PC-SDK.

Rift Core 2.0 introduces substantial changes to Oculus Home and replaces the Universal Menu with Oculus Dash. We plan to roll it out to Rift users with the 1.22 runtime in early 2018.

Figure 12: Oculus Dash User Interface



Dash re-implements the Universal Menu as a VR compositor layer. Please see the "Introducing Oculus Dash" video in our *Welcome to Rift Core 2.0* blog post to get a sense of how it works.

Beginning with runtime 1.22, when users pause an application, instead of rendering the Universal Menu in an empty room, one of two things will happen:

- If the application includes Dash support, the application will pause and the Dash menu UI will be drawn over the paused application.
- If the application does not include Dash support, the application will be paused by the runtime and the user will be presented with the Dash menu UI in an empty room, similar to the way the Universal Menu is displayed in earlier runtimes.

When the Dash UI is active, the runtime will render tracked controllers in the scene to interact with the menu. Your application should pause, mute, and hide any tracked controllers it renders in the scene, so there will not be a duplicate pair of hands.

We recommend adding Dash support to your application to provide the best possible user experience. Developers who would like to add Dash support before it rolls out in early 2018 can test it using Oculus runtime 1.21, which includes preview support. Oculus runtime 1.21 is now available through our opt-in Public Test Channel (PTC). For information on how to access PTC, see this support article: *https://support.oculus.com/200468603765391/*

When you implement support for Dash, you need to consider three areas:

- Input Focus Handling
- Depth Buffer Support
- Declaring an Application is Input Focus Aware

Input Focus Handling

When the Dash UI is active, the running application loses input focus and the `ovrSessionStatus::hasInputFocus` flag will return false. In this state, the runtime renders tracked controllers in the scene to interact with the menu.

When `ovrSessionStatus::hasInputFocus` is false, your application should pause all activity, mute audio playback, hide any tracked controllers in the scene so there will not be a duplicate pair of hands, and hide any near field objects (within about one meter of the user). Depending on the application, additional action may also be warranted when input focus is lost. For example, during a multiplayer combat game, you may wish to indicate that the player is unavailable and take any other appropriate action.

The VR Compositor may use up to 3 ms of additional rendering time during each frame cycle while `HasInputFocus` is False. Because of this, you may wish to switch your application into a lower performance mode, if possible, as long as `HasInputFocusis` False. You can then wait until `HasInputFocus` becomes True again before reverting the actions described above. This is not required, and Oculus does not enforce performance VRC requirements during the time that the Dash UI is active.

Note that `HasInputFocus` returns false under any other conditions in which the application loses input focus, such as when the HMD is removed from the head.

You can check the `HasInputFocus` flag with the following code:

```
ovrSessionStatus sessionStatus = {};
ovr_GetSessionStatus(Session, &sessionStatus);
if (!sessionStatus.HasInputFocus) {/*Handle situation where your app has lost input focus*/}
```

This code should be executed once during every frame render cycle.

Depth Buffer Support

If you are rendering of lot of geometry near the user, it may cause uncomfortable visual disparities when a Dash panel renders on top of geometry that is closer to the player than the Dash panel. To avoid that disparity, you can submit depth with your eye buffers. This will allow Dash to draw an x-ray effect that prevents this discomfort. For this reason, and for future improvements, we recommend submitting depth data with your eye

buffers. However, if you are unable to do this, it is still better to support Dash (and live with the disparity) than to not support Dash.

To do this, use `ovrLayerType_EyeFovDepth`, as shown in the Oculus World Demo (available in TeamCity).

Declaring an Application is Input Focus Aware

Your application should indicate whether or not it is it is prepared to respond to `ovrSessionStatus` focus states, including `HasInputFocus`. If your application is prepared to handle the loss of focus, as described under **Input Focus Handling** (above), then set the `ovrInit_FocusAware` flag to True. Otherwise, set `ovrInit_FocusAware` to False.

Coping with Graphics API or Hardware Render Target Granularity

The SDK is designed with the assumption that you want to use your video memory as carefully as possible and that you can create exactly the right render target size for your needs.

However, real video cards and real graphics APIs have size limitations (all have a maximum size; some also have a minimum size). They might also have granularity restrictions, for example, only being able to create render targets that are a multiple of 32 pixels in size or having a limit on possible aspect ratios. As an application developer, you can also impose extra restrictions to avoid using too much graphics memory.

In addition to the above, the size of the actual render target surface in memory might not necessarily be the same size as the portion that is rendered to. The latter may be slightly smaller. However, since it is specified as a viewport, it typically does not have any granularity restrictions. When you bind the render target as a texture, however, it is the full surface that is used, and so the UV coordinates must be corrected for the difference between the size of the rendering and the size of the surface it is on. The API will do this for you, but you need to tell it the relevant information.

The following code shows a two-stage approach for settings render target resolution. The code first calls `ovr_GetFovTextureSize` to compute the ideal size of the render target. Next, the graphics library is called to create a render target of the desired resolution. In general, due to idiosyncrasies of the platform and hardware, the resulting texture size might be different from that requested.

```
// Get recommended left and right eye render target sizes.
Sizei recommenedTex0Size = ovr_GetFovTextureSize(session, ovrEye_Left,
                           session->DefaultEyeFov[0], pixelsPerDisplayPixel);
Sizei recommenedTex1Size = ovr_GetFovTextureSize(session, ovrEye_Right,
                           session->DefaultEyeFov[1], pixelsPerDisplayPixel);

// Determine dimensions to fit into a single render target.
Sizei renderTargetSize;
renderTargetSize.w = recommenedTex0Size.w + recommenedTex1Size.w;
renderTargetSize.h = max ( recommenedTex0Size.h, recommenedTex1Size.h );

// Create texture.
pRendertargetTexture = pRender->CreateTexture(renderTargetSize.w, renderTargetSize.h);

// The actual RT size may be different due to HW limits.
renderTargetSize.w = pRendertargetTexture->GetWidth();
renderTargetSize.h = pRendertargetTexture->GetHeight();

// Initialize eye rendering information.
// The viewport sizes are re-computed in case RenderTargetSize changed due to HW limitations.
ovrFovPort eyeFov[2] = { session->DefaultEyeFov[0], session->DefaultEyeFov[1] };

EyeRenderViewport[0].Pos  = Vector2i(0,0);
EyeRenderViewport[0].Size = Sizei(renderTargetSize.w / 2, renderTargetSize.h);
EyeRenderViewport[1].Pos  = Vector2i((renderTargetSize.w + 1) / 2, 0);
EyeRenderViewport[1].Size = EyeRenderViewport[0].Size;
```

This data is passed into `ovr_EndFrame` as part of the layer description.

You are free to choose the render target texture size and left and right eye viewports as you like, provided that you specify these values when calling `ovr_EndFrame` using the `ovrTexture`. However, using `ovr_GetFovTextureSize` will ensure that you allocate the optimum size for the particular HMD in use. The following sections describe how to modify the default configurations to make quality and performance trade-offs. You should also note that the API supports using different render targets for each eye if that is required by your engine (although using a single render target is likely to perform better since it will reduce context switches). OculusWorldDemo allows you to toggle between using a single combined render target versus separate ones for each eye, by navigating to the settings menu (press the Tab key) and selecting the Share RenderTarget option.

Forcing a Symmetrical Field of View

Typically the API will return an FOV for each eye that is not symmetrical, meaning the left edge is not the same distance from the center as the right edge.

This is because humans, as well as the Rift, have a wider FOV when looking outwards. When you look inwards, your nose is in the way. We are also better at looking down than we are at looking up. For similar reasons, the Rift's view is not symmetrical. It is controlled by the shape of the lens, various bits of plastic, and the edges of the screen. The exact details depend on the shape of your face, your IPD, and where precisely you place the Rift on your face; all of this is set up in the configuration tool and stored in the user profile. All of this means that almost nobody has all four edges of their FOV set to the same angle, so the frustum produced will be off-center. In addition, most people will not have the same fields of view for both their eyes. They will be close, but rarely identical.

As an example, on our first generation DK1 headset, the author's left eye has the following FOV:

- 53.6 degrees up
- 58.9 degrees down
- 50.3 degrees inwards (towards the nose)
- 58.7 degrees outwards (away from the nose)

In the code and documentation, these are referred to as 'half angles' because traditionally a FOV is expressed as the total edge-to-edge angle. In this example, the total horizontal FOV is 50.3+58.7 = 109.0 degrees, and the total vertical FOV is 53.6+58.9 = 112.5 degrees.

The recommended and maximum fields of view can be accessed from the HMD as shown below:

```
ovrFovPort defaultLeftFOV = session->DefaultEyeFov[ovrEye_Left];

ovrFovPort maxLeftFOV = session->MaxEyeFov[ovrEye_Left];
```

`DefaultEyeFov` refers to the recommended FOV values based on the current user's profile settings (IPD, eye relief etc). `MaxEyeFov` refers to the maximum FOV that the headset can possibly display, regardless of profile settings.

The default values provide a good user experience with no unnecessary additional GPU load. If your application does not consume significant GPU resources, you might want to use the maximum FOV settings to reduce reliance on the accuracy of the profile settings. You might provide a slider in the application control panel that enables users to choose interpolated FOV settings between the default and the maximum. But, if your application is heavy on GPU usage, you might want to reduce the FOV below the default values as described in *Improving Performance by Decreasing Field of View* on page 41.

The FOV angles for up, down, left, and right (expressed as the tangents of the half-angles), is the most convenient form to set up culling or portal boundaries in your graphics engine. The FOV values are also used to determine the projection matrix used during left and right eye scene rendering. We provide an API utility function ovrMatrix4f_Projection for this purpose:

```
ovrFovPort fov;

// Determine fov.
...

ovrMatrix4f projMatrix = ovrMatrix4f_Projection(fov, znear, zfar, 0);
```

It is common for the top and bottom edges of the FOV to not be the same as the left and right edges when viewing a PC monitor. This is commonly called the 'aspect ratio' of the display, and very few displays are square. However, some graphics engines do not support off-center frustums. To be compatible with these engines, you will need to modify the FOV values reported by the `ovrHmdDesc` struct. In general, it is better to grow the edges than to shrink them. This will put a little more strain on the graphics engine, but will give the user the full immersive experience, even if they won't be able to see some of the pixels being rendered.

Some graphics engines require that you express symmetrical horizontal and vertical fields of view, and some need an even less direct method such as a horizontal FOV and an aspect ratio. Some also object to having frequent changes of FOV, and may insist that both eyes be set to the same. The following is a an example of code for handling this restrictive case:

```
ovrFovPort fovLeft = session->DefaultEyeFov[ovrEye_Left];
ovrFovPort fovRight = session->DefaultEyeFov[ovrEye_Right];

ovrFovPort fovMax = FovPort::Max(fovLeft, fovRight);
float combinedTanHalfFovHorizontal = max ( fovMax.LeftTan, fovMax.RightTan );
float combinedTanHalfFovVertical = max ( fovMax.UpTan, fovMax.DownTan );

ovrFovPort fovBoth;
fovBoth.LeftTan = fovBoth.RightTan = combinedTanHalfFovHorizontal;
fovBoth.UpTan = fovBoth.DownTan = combinedTanHalfFovVertical;


// Create render target.
Sizei recommenedTex0Size = ovr_GetFovTextureSize(session, ovrEye_Left,
                                                 fovBoth, pixelsPerDisplayPixel);
Sizei recommenedTex1Size = ovr_GetFovTextureSize(session, ovrEye_Right,
                                                 fovBoth, pixelsPerDisplayPixel);

...


// Initialize rendering info.
ovrFovPort eyeFov[2];
eyeFov[0]                       = fovBoth;
eyeFov[1]                       = fovBoth;

...


// Compute the parameters to feed to the rendering engine.
// In this case we are assuming it wants a horizontal FOV and an aspect ratio.
float horizontalFullFovInRadians = 2.0f * atanf ( combinedTanHalfFovHorizontal );
float aspectRatio = combinedTanHalfFovHorizontal / combinedTanHalfFovVertical;

GraphicsEngineSetFovAndAspect ( horizontalFullFovInRadians, aspectRatio );
...
```

Note: You will need to determine FOV before creating the render targets, since FOV affects the size of the recommended render target required for a given quality.

Improving Performance by Decreasing Pixel Density

The DK1 has a resolution of 1280x800 pixels, split between the two eyes. However, because of the wide FOV of the Rift and the way perspective projection works, the size of the intermediate render target required to match the native resolution in the center of the display is significantly higher.

For example, to achieve a 1:1 pixel mapping in the center of the screen for the author's field-of-view settings on a DK1 requires a much larger render target that is 2000x1056 pixels in size.

Even if modern graphics cards can render this resolution at the required 60Hz, future HMDs might have significantly higher resolutions. For virtual reality, dropping below 60Hz provides a terrible user experience; it is always better to decrease the resolution to maintain framerate. This is similar to a user having a high resolution 2560x1600 monitor. Very few 3D applications can run at this native resolution at full speed, so most allow the user to select a lower resolution to which the monitor upscales to the fill the screen.

You can use the same strategy on the HMD. That is, run it at a lower video resolution and let the hardware upscale for you. However, this introduces two steps of filtering: one by the distortion processing and one by the video upscaler. Unfortunately, this double filtering introduces significant artifacts. It is usually more effective to leave the video mode at the native resolution, but limit the size of the intermediate render target. This gives a similar increase in performance, but preserves more detail.

One way to resolve this is to allow the user to adjust the resolution through a resolution selector. However, the actual resolution of the render target depends on the user's configuration, rather than a standard hardware setting This means that the 'native' resolution is different for different people. Additionally, presenting resolutions higher than the physical hardware resolution might confuse some users. They might not understand that selecting 1280x800 is a significant drop in quality, even though this is the resolution reported by the hardware.

A better option is to modify the `pixelsPerDisplayPixel` value that is passed to the `ovr_GetFovTextureSize` function. This could also be based on a slider presented in the applications render settings. This determines the relative size of render target pixels as they map to pixels at the center of the display surface. For example, a value of 0.5 would reduce the render target size from 2000x1056 to 1000x528 pixels, which might allow mid-range PC graphics cards to maintain 60Hz.

```
float pixelsPerDisplayPixel = GetPixelsPerDisplayFromApplicationSettings();

Sizei recommenedTexSize = ovr_GetFovTextureSize(session, ovrEye_Left,  fovLeft,
                                                pixelsPerDisplayPixel);
```

Although you can set the parameter to a value larger than 1.0 to produce a higher-resolution intermediate render target, Oculus hasn't observed any useful increase in quality and it has a high performance cost.

OculusWorldDemo allows you to experiment with changing the render target pixel density. Navigate to the settings menu (press the Tab key) and select Pixel Density. Press the up and down arrow keys to adjust the pixel density at the center of the eye projection. A value of 1.0 sets the render target pixel density to the display surface 1:1 at this point on the display. A value of 0.5 means sets the density of the render target pixels to half of the display surface. Additionally, you can select Dynamic Res Scaling which will cause the pixel density to automatically adjust between 0 to 1.

Improving Performance by Decreasing Field of View

In addition to reducing the number of pixels in the intermediate render target, you can increase performance by decreasing the FOV that the pixels are stretched across.

Depending on the reduction, this can result in tunnel vision which decreases the sense of immersion. Nevertheless, reducing the FOV increases performance in two ways. The most obvious is fillrate. For a fixed pixel density on the retina, a lower FOV has fewer pixels. Because of the properties of projective math, the outermost edges of the FOV are the most expensive in terms of numbers of pixels. The second reason is that there are fewer objects visible in each frame which implies less animation, fewer state changes, and fewer draw calls.

Reducing the FOV set by the player is a very painful choice to make. One of the key experiences of virtual reality is being immersed in the simulated world, and a large part of that is the wide FOV. Losing that aspect is not a thing we would ever recommend happily. However, if you have already sacrificed as much resolution as you can, and the application is still not running at 60Hz on the user's machine, this is an option of last resort.

We recommend giving players a Maximum FOV slider that defines the four edges of each eye's FOV.

```
ovrFovPort defaultFovLeft = session->DefaultEyeFov[ovrEye_Left];
ovrFovPort defaultFovRight = session->DefaultEyeFov[ovrEye_Right];

float maxFovAngle = ...get value from game settings panel...;
float maxTanHalfFovAngle = tanf ( DegreeToRad ( 0.5f * maxFovAngle ) );

ovrFovPort newFovLeft  = FovPort::Min(defaultFovLeft, FovPort(maxTanHalfFovAngle));
ovrFovPort newFovRight = FovPort::Min(defaultFovRight, FovPort(maxTanHalfFovAngle));


// Create render target.
Sizei recommenedTex0Size = ovr_GetFovTextureSize(session, ovrEye_Left  newFovLeft,
 pixelsPerDisplayPixel);
Sizei recommenedTex1Size = ovr_GetFovTextureSize(session, ovrEye_Right, newFovRight,
 pixelsPerDisplayPixel);

...


// Initialize rendering info.
ovrFovPort eyeFov[2];
eyeFov[0]                       = newFovLeft;
eyeFov[1]                       = newFovRight;

...


// Determine projection matrices.
ovrMatrix4f projMatrixLeft = ovrMatrix4f_Projection(newFovLeft, znear, zfar, 0);
ovrMatrix4f projMatrixRight = ovrMatrix4f_Projection(newFovRight, znear, zfar, 0);
```

It might be interesting to experiment with non-square fields of view. For example, clamping the up and down ranges significantly (e.g. 70 degrees FOV) while retaining the full horizontal FOV for a 'Cinemascope' feel.

OculusWorldDemo allows you to experiment with reducing the FOV below the defaults. Navigate to the settings menu (press the Tab key) and select the "Max FOV" value. Pressing the up and down arrows to change the maximum angle in degrees.

Improving Performance by Rendering in Mono

A significant cost of stereo rendering is rendering two views, one for each eye.

For some applications, the stereoscopic aspect may not be particularly important and a monocular view might be acceptable in return for some performance. In other cases, some users may get eye strain from a stereo view and wish to switch to a monocular one. However, they still wish to wear the HMD as it gives them a high FOV and head-tracking.

OculusWorldDemo allows the user to toggle mono render mode by pressing the F7 key.

To render in mono, your code should have the following changes:

- Set the FOV to the maximum symmetrical FOV based on both eyes.
- Call `ovhHmd_GetFovTextureSize` with this FOV to determine the recommended render target size.
- Configure both eyes to use the same render target and the same viewport when calling `ovr_EndFrame`.
- Render the scene once to the shared render target.

This merges the FOV of the left and right eyes into a single intermediate render. This render is still distorted twice, once per eye, because the lenses are not exactly in front of the user's eyes. However, this is still a significant performance increase.

Setting a virtual IPD to zero means that everything will seem gigantic and infinitely far away, and of course the user will lose much of the sense of depth in the scene.



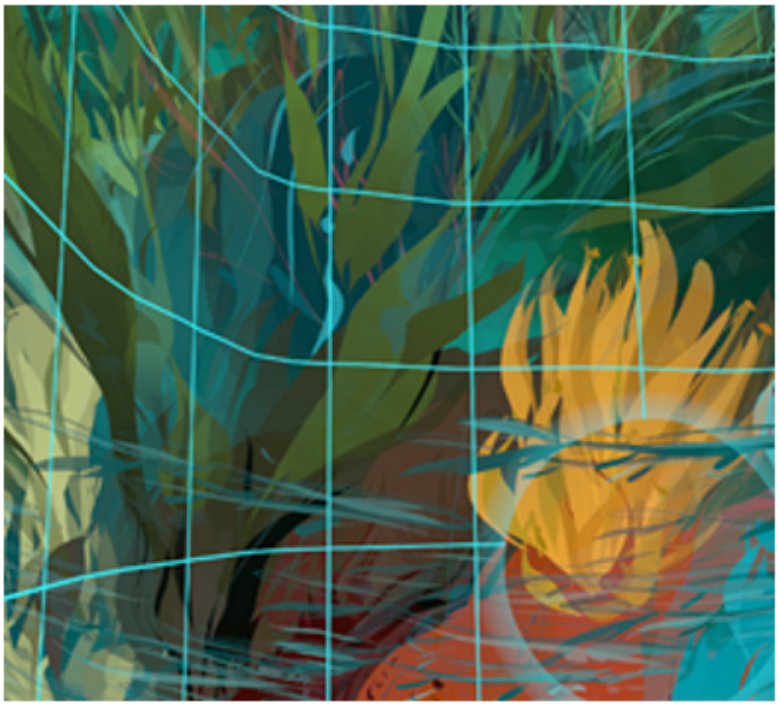
Note: It is important to scale virtual IPD and virtual head motion together so, if the virtual IPD is set to zero, all virtual head motion due to neck movement is also be eliminated. Sadly, this loses much of the depth cues due to parallax. But, if the head motion and IPD do not agree, it can cause significant disorientation and discomfort. Experiment with caution!

Using Octilinear Rendering for NVIDIA Lens Matched Shading

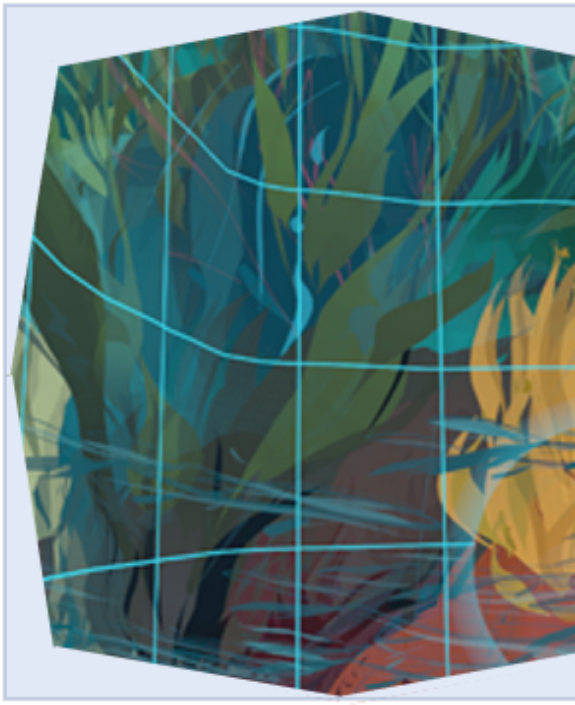
Octilinear rendering implements NVIDIA Lens Matched Shading. You can use octilinear rendering to improve the performance of your application when it executes on an NVIDIA GPU.

Octilinear rendering avoids the need to render a significant number of pixels that would otherwise be discarded before the image is output to the headset. The application has to draw a larger image than is displayed on the Rift. It requires extra time to draw these larger images. So Lens Matched Shading allows the system to draw a smaller image, because it draws a pre-compressed image. An octilinear image appears as follows:

Original Image



Octilinear Image



Using octilinear rendering reduces the size of the drawing area by about 20%. The bounding box remains the same as for the non-octilinear image.

The requirements for using octilinear rendering are as follows:

1. You must run your application on an NVIDIA GPU that supports lens mapped shading. (See the NVIDIA documentation for more information.)
2. You must use an NVIDIA SDK that supports the lens mapped shading feature of the GPU. (See the NVIDIA documentation for more information.)
3. You must generate textures in the right format with the NVIDIA SDK.
4. Your application must call `ovr_EnableExtension` to enable the octilinear extension from the Oculus side.
5. Then, the Oculus runtime will automatically consume your specially-format images at runtime.

Protecting Content

There are some cases where you only want the content to display on the headset. The protected content feature is designed to prevent any mirroring of the compositor.

To use the protected content feature, configure your application to create one or more ovrTextureSwapChain objects with the ovrTextureMisc_ProtectedContent flag specified. Any submission which references a protected swap chain is considered a protected frame; any protected frames are composited and displayed in the HMD, but are not replicated to mirrors or available to the capture buffer API.

If the HMD is not HDCP compliant, the texture swap chain creation API will fail with ovrError_ContentProtectionNotAvailable. If the textures can be created (HDCP-compliant HMD), but the link is broken later, the next `ovr_EndFrame` call that references protected texture swap chains will fail with the ovrError_ContentProtectionNotAvailable error. Configure your application to respond according to your

requirements. For example, you might submit that next frame without protected swap chains, but at a lower quality that doesn't require protection. Or, you might stop playback and display an error or warning to the user.

Note: Because the mirror window is in the control of the application, if your application does not use the mirror texture, it is up to your application to render something to the preview window. Make sure your application does not display protected content. Since the surfaces are not known to be protected by the OS, they will be displayed normally inside the application that created them.

To enable protected content, specify the

```
ovrTextureMisc_ProtectedContent
```

flag similarly to the following:

```
ovrTextureSwapChainDesc desc = {};
desc.Type = ovrTexture_2D;
desc.ArraySize = 1;
desc.Format = OVR_FORMAT_R8G8B8A8_UNORM_SRGB;
desc.Width = sizeW;
desc.Height = sizeH;
desc.MipLevels = 1;
desc.SampleCount = 1;
desc.MiscFlags = ovrTextureMisc_DX_Typeless | ovrTextureMisc_ProtectedContent;
desc.BindFlags = ovrTextureBind_DX_RenderTarget;
desc.StaticImage = ovrFalse;

ovrResult result = ovr_CreateTextureSwapChainDX(session, Device, &desc, &TextureChain);
```

VR Focus Management

When you submit your application to Oculus, you provide the application and metadata necessary to list it in the Oculus Store and launch it from Oculus Home.

Once launched from Oculus Home, you need to write a loop that polls for session status. ovr_GetSessionStatus returns a struct with the following booleans:

- `ShouldQuit`—True if the application should initiate shutdown.
- `HmdPresent`—True if an HMD is present.
- `DisplayLost`—True if the HMD was unplugged or the display driver was manually disabled or encountered a TDR.
- `HmdMounted`—True if the HMD is on the user's head.
- `IsVisible`—True if the game or experience has VR focus and is visible in the HMD.
- `ShouldRecenter`—True if the application should call `ovr_RecenterTrackingOrigin`. This is triggered when the user initiates recentering through the Universal Menu.

Managing When a User Quits

If `ShouldQuit` is true, save the application state and shut down or shut down without saving the application state. The user will automatically return to Oculus Home.

Depending on the type of application, you can prompt the user to start where he or she left off the next time it is opened (e.g., a multi-level game) or you can just start from the beginning of the experience (e.g., a passive video). If this is a multiplayer game, you might want to quit locally without ending the game.

Managing When a User Requests Recentering

If `ShouldRecenter` is true, the application should call `ovr_RecenterTrackingOrigin` or `ovr_SpecifyTrackingOrigin` and be prepared for future tracking positions to be based on a different origin.

Some applications may have reason to ignore the request or to implement it via an internal mechanism other than via `ovr_RecenterTrackingOrigin`. In such cases the application can call `ovr_ClearShouldRecenterFlag` to cause the recenter request to be cleared.

Managing an Unplugged Headset

If `DisplayLost` is true:

1. Pause the application, including audio.
2. Display a prompt on the monitor that says that the headset was unplugged.
3. Destroy any TextureSwapChains or mirror textures.
4. Call `ovrDestroy`.
5. Poll `ovrSessionStatus::HmdPresent` until true.
6. Call `ovrCreate` to recreate the session.
7. Recreate any TextureSwapChains or mirror textures.
8. Resume the application.

If `ovrDetect` doesn’t isn’t returned as true after a specified amount of time, act as though `ShouldQuit` returned true. If the user takes no action after a specified amount of time, choose a default action (save the session or close without saving) and close the application.

Note: For multiplayer games, you might want to follow the same process without pausing the game.

Managing an Unavailable Headset

When a user removes the headset or if your application does not have VR focus, `HmdMounted` or `IsVisible` returns false. Pause the application until they return true.

When your application loses VR focus, it automatically stops receiving input. If your application does not use the Oculus input API, it will need to ignore any received input.

Note: For multiplayer games, you might want the game to continue without pausing.

Managing Loss of Windows Focus

When your application loses Windows focus, the Oculus Remote, Xbox controller, and Touch controllers will continue to work normally. However, the application will lose control of the mouse and keyboard.

If your application loses Windows focus and maintains VR focus (`IsVisible`), continue to process input and allow the application to run normally. If the keyboard or mouse is needed to continue, prompt the user to remove the headset and use Alt-Tab to regain Windows focus.

Managing Dashboards and Application Focus

With the introduction of Dash, your application should now indicate whether or not it is it is prepared to respond to `ovrSessionStatus` focus states, including `ovrSessionStatus::HasInputFocus`. For more information, please see *Using Oculus Dash* on page 35

Code Sample

```
bool shouldQuit = false;

void RunApplication()
{
    ovrResult result = ovr_Initialize();

    if (OVR_SUCCESS(result))
    {
        ovrSession session;
        ovrGraphicsLuid luid;
        result = ovr_Create(&session, &luid);
        result = ovr_WaitToBeginFrame(session, 0);
        result = ovr_BeginFrame(session, 0);

        if (OVR_SUCCESS(result))
        {
            ovrSessionStatus ss;

            <create graphics device with luid>
            <create render target via ovr_CreateTextureSwapChain>

            while (!shouldQuit)
            {
                <get next frame pose, e.g. via ovr_GetEyePoses>
                <render frame>

                result = ovr_EndFrame(...);

                if (result == ovrSuccess_NotVisible)
                {
                    <turn off audio output>
                    do { // Wait until we regain visibility or should quit
                        <sleep>
                        result = ovr_GetSessionStatus(session, &ss);
```

```
                        if (ss.ShouldQuit)
                            shouldQuit = true;
                    } while (OVR_SUCCESS(result) && !ss.IsVisible && !shouldQuit);
                    <possibly re-enable audio>
                }
                else if (result == ovrError_DisplayLost)
                {
                    // We can either immediately quit or do the following:
                    <destroy render target and graphics device>
                    ovr_Destroy(session);

                    do { // Spin while trying to recreate session.
                        result = ovr_Create(&session, &luid);
                    } while (OVR_FAILURE(result) && !shouldQuit);

                    if (OVR_SUCCESS(result))
                    {
                        <recreate graphics device with luid>
                        <recreate render target via ovr_CreateTextureSwapChain>
                    }
                }
                else if (OVR_FAILURE(result))
                {
                    shouldQuit = true;
                }

                ovr_GetSessionStatus(session, &ss);
                if (ss.ShouldQuit)
                    shouldQuit = true;
                if (ss.ShouldRecenter)
                {
                    ovr_RecenterTrackingOrigin(session); // or ovr_ClearShouldRecenterFlag(session)
 to ignore the request.
                    <do anything else needed to handle this>
                }
            }

            <destroy render target via ovr_DestroyTextureSwapChain>
            <destroy graphics device>

            ovr_Destroy(session);
        }

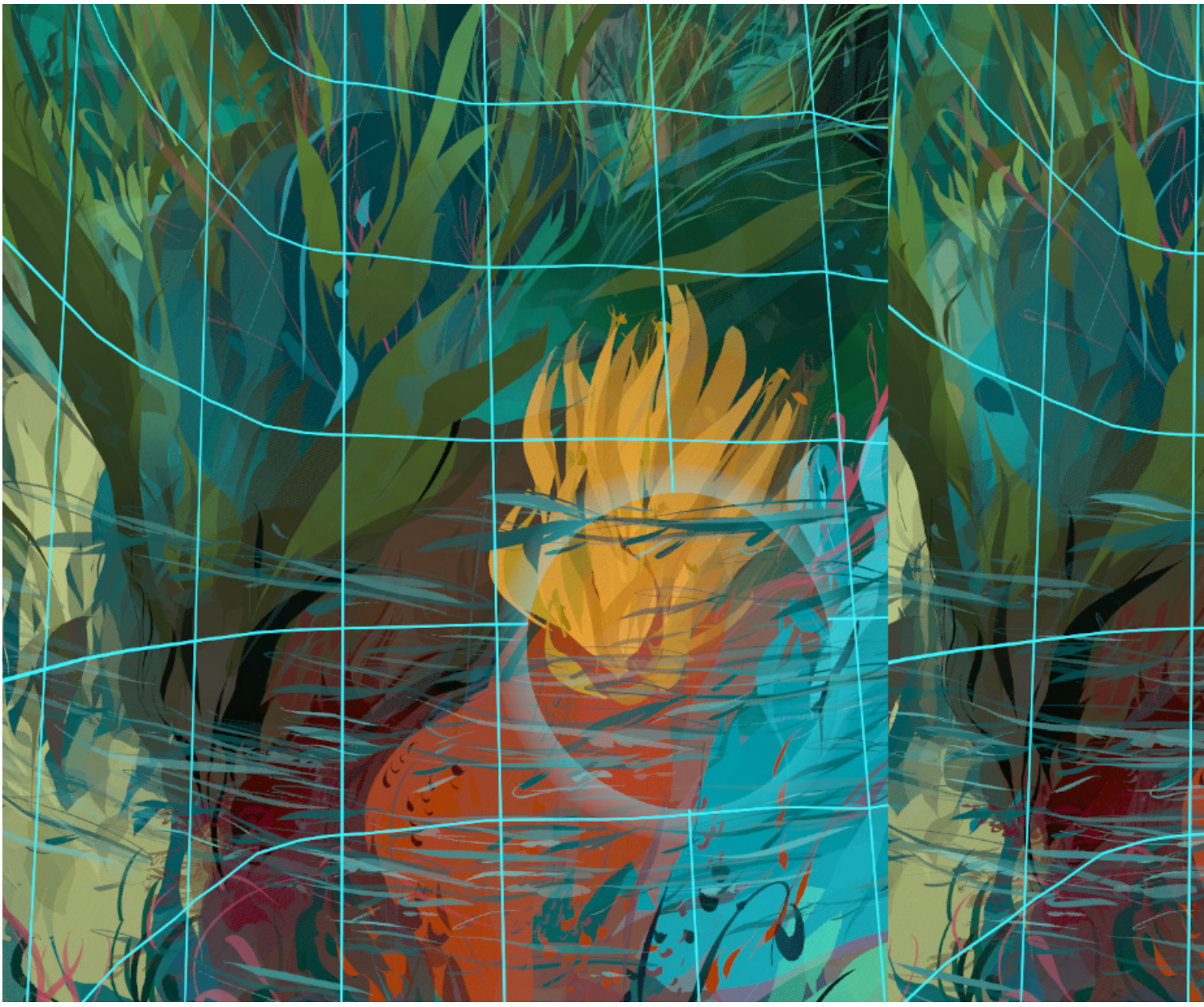
        ovr_Shutdown();
    }
}
```

Oculus Guardian System

The Oculus Guardian System is designed to display in-application wall and floor markers when users get near boundaries they defined. When the user gets too close to the edge of a boundary, translucent boundary markers are displayed in a layer that is superimposed over the game or experience.

The following image shows the Guardian System activated. In this example, the user's hands (which are holding Touch controllers) protrude through the guardian.

Figure 13: Oculus Boundary System



Setting Up the Guardian System on Your Rift

After the user sets up the boundaries, they will show up in a layer over any application whenever the user gets too close.

Note: Users can set the boundaries to Walls and Floor, set them to Floor Only, or disable them entirely.

To set up the boundaries:

1. Open the Oculus app.
2. Select Settings -> Devices -> Run Full Setup.
3. Select Rift and Touch.
4. Follow the on-screen instructions to confirm sensor tracking.
5. Continue until the Mark Your Boundaries page appears.
6. Follow the on-screen instructions, using the INDEX trigger button to draw the outer bounds of your play area. Currently, there is no minimum width and depth.

   When you are finished, click Next. Your boundaries are saved.
7. If you need to disable the Guardian System, toggle Guardian System Enabled/Disabled on the Universal Menu.

Game Configuration

During initialization, your application can make an API request to get the outer boundary and play area. The outer boundary is the space that the user defined during configuration. The play area is a rectangular space within the outer boundary. With this information, your application can set up a virtual world with "barriers" that align with the real world. For example, you can adjust the size of a cockpit based on the user-defined play area.

The following functions return information about the outer boundary and play area:

Function	Description
ovr_GetBoundaryDimensions	Returns the width, height, and depth of the play area or outer boundary in meters.
ovr_GetBoundaryGeometry	Returns the points that define the play area or outer boundary. For the play area, it returns the four points that define the rectangular area. For the outer boundary, it returns all of the points that define the outer boundary.

During runtime, your application can request whether the boundaries are visible using `ovr_GetBoundaryVisible`. When visible, you can choose how the application will respond. For example, you might choose to pause the application, slow the application, or simply display a message.

The boundary status information is returned in a struct which contains the following:

Member	Type	Description
IsTriggering	ovrBool	Returns whether the boundaries are currently visible.
ClosestDistance	float	Distance to the closest play area or outer boundary surface.
ClosestPoint	ovrVector3f	Closest point on the boundary surface.
ClosestPointNormal	ovrVector3f	Unit surface normal of the closes boundary surface.

Additionally, you can set the bounds to be visible to orient the user or explain how you will use the space by setting `ovr_RequestBoundaryVisible()` to `ovrTrue`. When you are finished, simply pass `ovrFalse`.

Note: You can't force the boundaries off if they were triggered by user.

The default boundary color is cyan. To change the color, use `ovr_SetBoundaryLookAndFeel()`.

Code Sample

To help you get started, we provide a code sample at `Samples/GuardianSystemDemo` that shows usage of the following APIs:

- ovr_TestBoundary
- ovr_TestBoundaryPoint
- ovr_SetBoundaryLookAndFeel
- ovr_RequestBoundaryVisible
- ovr_ResetBoundaryLookAndFee

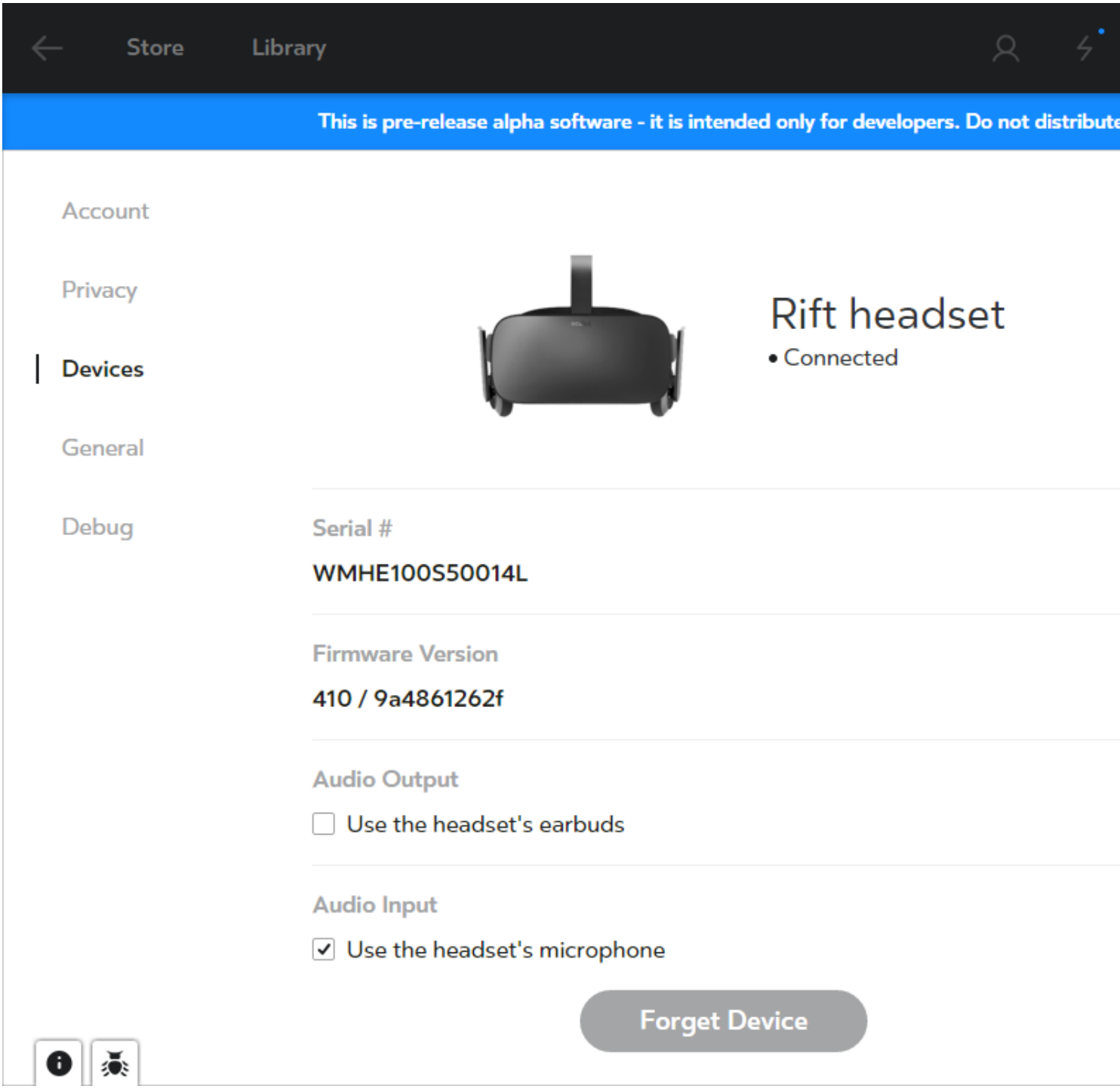
Boxes collide with the boundary data using the test API, the boundary visibility and color changes every second, and the simulation slows (and then stops) when the HMD or Touch controllers get too close to the boundary.

Rift Audio

When setting up audio for the Rift, you need to determine whether the Rift headphones are active and pause the audio when your app doesn't have focus.

The user can enable the Rift headphones and microphone in the Oculus app or use the default Windows audio devices. Configuration is made the Oculus app in Settings -> Devices -> Headset. The following screenshot shows the Rift headphones disabled and the microphone enabled:

Figure 14: Audio Configuration



The headphone setting is handled automatically by the function `ovr_GetAudioDeviceOutGuid` (located in OVR_CAPI_Audio.h), which returns the GUID for the device to target when playing audio. Similarly, use `ovr_GetAudioDeviceInGuid` to identify the microphone device used for input.

FMOD D for Native Rift Development

If you detect that the Rift headphones are enabled, use code similar to the following for FMOD:

```
ERRCHECK(FMOD::System_Create(&sys));
GUIDguid;
ovr_GetAudioDeviceOutGuid(&guid);

intdriverCount=0;
sys->getNumDrivers(&driverCount);

intdriver=0;
while(driver<driverCount)
{
        charname[256]={0};
        FMOD_GUIDfmodGuid={0};
        sys->getDriverInfo(driver,name,256,&fmodGuid,nullptr,nullptr,nullptr);

        if(guid.Data1==fmodGuid.Data1&&
                guid.Data2==fmodGuid.Data2&&
                guid.Data3==fmodGuid.Data3&&
                memcmp(guid.Data4,fmodGuid.Data4,8)==0)
        {
            break;
        }

       ++driver;
}

if(driver<driverCount)
{
    sys->setDriver(driver);
}
else
{
// error rift not connected
}
```

For instructions on using FMOD with Unreal Engine, see “Unreal and FMOD” below.

Wwise for Native Rift Development

If you detect that the Rift headphones are enabled, use code similar to the following for Wwise:

```
AkInitSettings initSettings;
AkPlatformInitSettings platformInitSettings;
AK::SoundEngine::GetDefaultInitSettings( initSettings );
AK::SoundEngine::GetDefaultPlatformInitSettings( platformInitSettings );

// Configure initSettings and platformInitSettings...

WCHAR outStr[128];
if (OVR_SUCCESS(ovr_GetAudioDeviceOutGuidStr(outStr)))
{
    initSettings.eMainOutputType = AkAudioAPI::AkAPI_Wasapi;
    platformInitSettings.idAudioDevice = AK::GetDeviceIDFromName(outStr);
}
```

For instructions on using Wwise with Unity 5, see “Unity 5 and Wwise” below. For instructions on using Wwise with Unreal Engine, see “Unreal and FMOD” below.

Unity 5

Audio input and output automatically use the Rift microphone and headphones unless configured to use the Windows default audio device by the user in the Oculus app. Events `OVRManager.AudioOutChanged` and `AudioInChanged` occur when audio devices change, making audio playback impossible without a restart.

Unity 5 and Wwise

To configure Wwise to use to configured audio device in Unity 5, pass the user-configured audio device name/GUID (set in the Oculus app) into the function `AkSoundEngine.GetDeviceIDFromName()`, located in `AkInitializer.cs`.

To get the audio device GUID from libOVR, you must include the Oculus Utilities unitypackage, which exposes that string through the class OVRManager.

The following function should be called before `AkSoundEngine.Init(...)`:

```
void SetRiftAudioDevice(AkPlatformInitSettings settings)
{
string audioDevice = OVRManager.audioOutId;
uint audioOutId = AkSoundEngine.GetDeviceIDFromName (audioDevice);
settings.idAudioDevice = audioOutId;
}
```

Pass `AkPlatformInitSettings` into the function above and use it to initialize the Ak sound engine.

Note: `OVRManager.audioOutId` will be deprecated in the future. This minor change should not impact the integration.

VR Audio Output in Oculus Store > Settings > Devices > Rift Headset may be used to configure which input mic to use within the Ak sound engine. The GUID for this is exposed through `OVRManager.audioInId`.

Unity 4

Audio input and output automatically use the Rift microphone and headphones, unless configured to use the Windows default audio device by the user in the Oculus app.

The Rift's microphone is not used by default when calling `Microphone.Start(null,..)`. Find the entry in `Microphone.devices` that contains the word Rift and use it.

Unreal

When Unreal PC applications are launched, if the OculusRift plugin is enabled and the Oculus VR Runtime Service is installed, then the application will automatically override the default Windows graphics and audio devices and target the Rift. The Oculus VR Runtime Service is installed with the Oculus app.

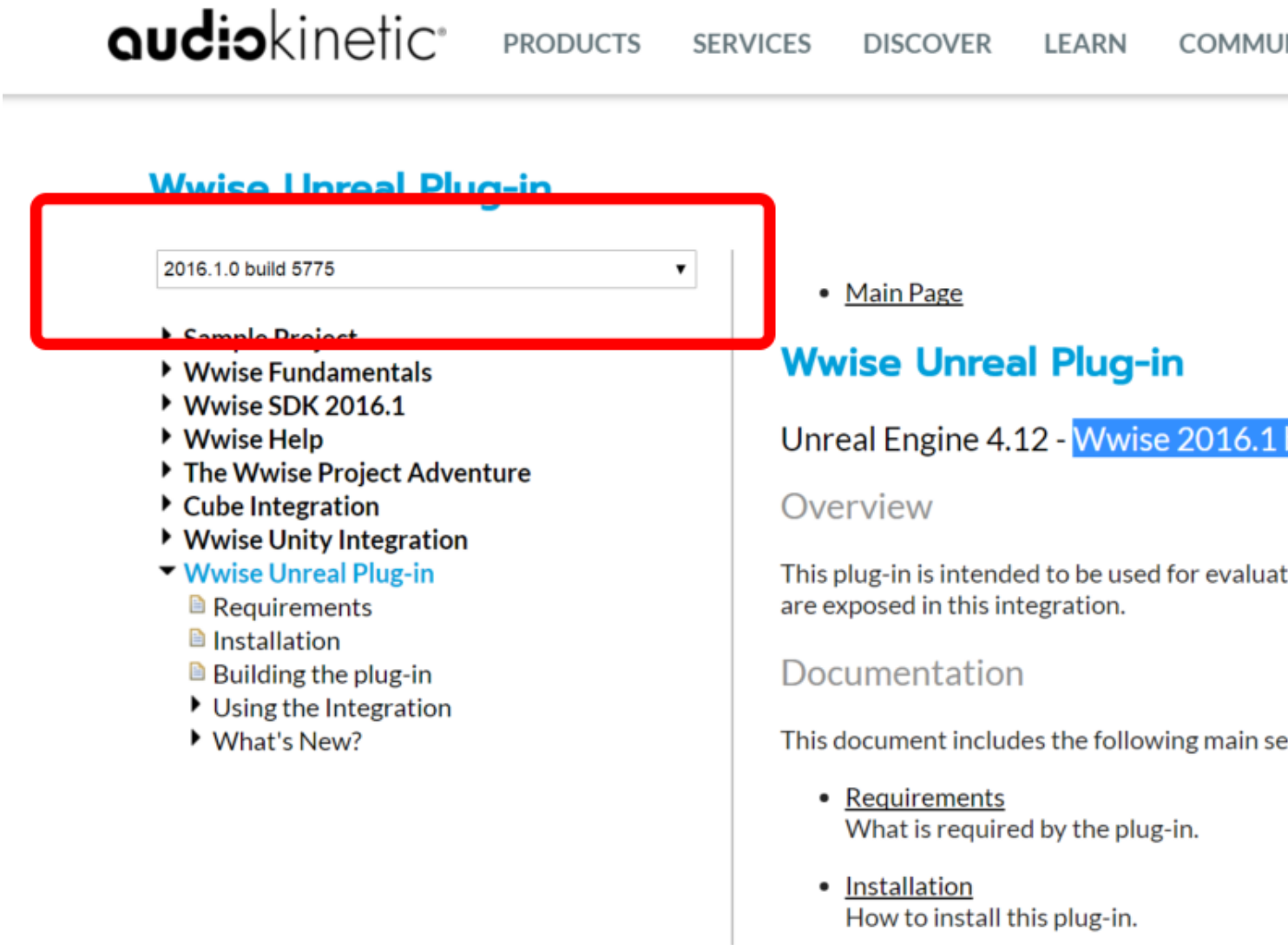
Unless your application is intended to run in VR, do not enable the OculusRift plugin. Otherwise, it is possible that audio and/or video will be incorrectly targeted to the Oculus Rift when the application is run.

Alternatively, users can disable loading all HMD plugins by specifying "-nohmd" on the command line.

Unreal and Wwise

To use Wwise with any Unreal version, use the Wwise Unreal Plug-in provided by Audiokinetic here: *https://www.audiokinetic.com/library/2016.1.0_5775/?source=UE4&id=index.html*.

Note: The link above links to build Wwise 2016.1 build 5775 for use with Unity 4.12. Be sure to select the appropriate build for your Unity version in the upper-left.



Unreal and FMOD

For an illustration of how to target the Oculus Rift headphones using FMOD in UE4, see the FMOD section above.

Oculus Audio Spatialization

The Oculus Audio SDK, available from our Downloads page, provides free, easy-to-use spatializer plugins for engines and middleware. Our spatialization features support both Rift and Gear VR development.

For a detailed discussion of audio spatialization and virtual reality audio, we recommend beginning with our Introduction to Virtual Reality Audio and Oculus Audio SDK guides.

For additional information on using Oculus spatializer plugins with Unity and Unreal, see Unity Audio and Unreal VR Audio.

VR Sound Level Best Practices

Audio is an important part of the virtual reality experience, and it is important that all VR apps provide a comfortable listening level for users. Developers should target a reasonable sound level that is consistent between different experiences. To achieve this, Oculus recommends the following best practices.

First, target -18 LUFS during final mix, using tools such as Avid's Pro Limited Plugin, Nugen's VisLM, Klangfreund LUFS Meter, Audacity VuMeter, or similar loudness measurement tools.

Second, measure your experience against the audio levels in published Oculus experiences, especially the ambient audio in Home and the Dreamdeck experience. Set overall system volume so that Dreamdeck and Home sound comfortable, and then adjust your experience's mix to this volume.

Finally, mix your application using the Rift headphones. This ensures that the sounds you're creating and mixing have a frequency content appropriate for the headphones most Oculus users will use.

By adhering to these guidelines, we can guarantee that our Oculus VR users will have a pleasant audio experience.

Oculus Touch Controllers

This section describes Oculus Touch best practices gathered from developing and reviewing large numbers of games and experiences. These are not requirements and we expect them to evolve over time.

Note: To view application requirements, go to *https://developer.oculus.com/distribute/latest/concepts/publish-rift-app-submission/*

Input, Hands, and Controller Basics

When developing applications that use Oculus Touch, consider the following:

- If your application supports multiple input methods (such as Oculus Touch, Xbox 360 Controller, and Oculus Remote), use `ovrControllerType_Active` to determine which one is in use. You can also use this to determine if one or both Touch controllers are in use. Some applications render the Touch controllers differently (e.g. as hands vs. controllers) depending on their in-use state. For more information, see this section and *https://developer.oculus.com/doc/1.19-libovr/_o_v_r___c_a_p_i_8h.html*.
- Unless you have an uncommon use case, use the Avatar SDK to represent high quality hands and/or controllers in your app.
- Map the grip button to grab actions. Although there are some exceptions to the rule (especially involving throwing), the new user experience and most applications condition users to use it this way. If you break expectations, make sure to educate your users.
- Prefer hands over controller models, especially for any application that involves social interaction or picking things up.
- In-application hands and controllers should line up with the user's real-world counterparts in position and orientation as closely as possible. We call this "registration" and recommend the Avatar SDK as an example. When testing your application, one technique is to hold your hands in front of your face and raise your headset slightly so you can compare your real-world hands and your virtual hands.
- Fully animate all hands and controllers in the game, to reflect what the user is doing. Users will expect to be able to grab, point, and thumbs-up with their hands. They will expect controller joysticks, buttons, and triggers to animate on use. We have demos with full code and blueprints for doing this in UE4 and Unity.
- To support left- and right-handed players, we recommend designing objects for use with either hand. For example, if the player is going to use a double rotating-wheel can opener to open a can in VR, make sure the cutting wheel handle faces the other hand.
- When controllers are mirrored, make controls identical. When controllers have different functions, make sure to establish a dominant and non-dominant hand (instead of favoring the right hand). For example, in the Toy Box demo, players usually use their non-dominant to hold the slingshot and their dominant hand to pull the ammunition holder.
- Unless important to the experience, do not render additional body parts besides the hands in single player games. Although inverse kinematics (i.e., extrapolating body part positions based on how the body can and cannot move) are a tempting solution to represent more the player's body, they become less accurate as you try to simulate more of the body. The mismatch can end up being distracting and ironically less immersive.

  For multiplayer games, you can render the hands of the first person and the up to the full bodies of the other players.

Tracking

The following are basic tracking tips:

- For front-facing apps, keep the action in front of the user. Don't encourage users to do things that will break the line of sight between the controllers and sensors, which will lead to poor tracking.

- Don't require the user to interact with VR elements near the floor or far above their heads. Remove those objects, or allow a distance grab. If you decide to allow interactions at a distance, make sure to indicate that the object is available through a highlight, glow, shake, haptics, or another mechanism.
- Avoid interactions that encourage users to get too close or too far from one or more position trackers.

Large Play Areas

For applications that use a large amount of the tracking space, either horizontally or vertically, you'll want to put a little more effort into making the best use of the available space:

- Our user-facing description of "Standing" apps is "may require a step in any direction." Even if your application uses a lot of space, if it is going to be submitted as a standing app, it needs to meet that description. Additionally, Standing apps must be fully usable in a 2m x 1.5 tracking area.
- If your application requires lots of space, use the user's Guardian boundaries to make sure you place game objects within reachable areas. For more information, see *Oculus Guardian System* on page 48.
- Applications can modify recenter behavior in a way that makes sense for the app. For example, an app might clamp the recenter origin inside of the play area by some amount of padding, so that all gameplay elements are reachable. Or it might choose to ignore the yaw component, to maintain an axis-aligned play area. If an app substantially violates user expectations for recenter, it should inform the user about what's happening. For more information, see *VR Focus Management* on page 45.
- Applications can also use the above recenter functionality to set an optimal center position (overriding the user's home position) on launch. You can use the default recenter behavior on subsequent user-initiated recenters, or continue to use modified recenter logic.
- If the user's play space is undefined, create your own play space around the user's recenter/home point. This assumed play space should not exceed 2m x 1.5m (the recommended play space size). If your application can use a smaller space, default to that.
- If your app requires high or low tracking, keep the shape of the sensor frustums in mind. For example, a front-facing app that involves shooting basketballs will want to keep the user towards the rear of the play area, where the tracking frustums are taller. For more information on sensor field of view and tracking ranges, see: *Initialization and Sensor Enumeration* on page 6.
- Some users might not have the full 2m x 1.5m recommended play space, but will play your game despite warnings. If your application requires this full amount of space, choose the best fallback. Possible solutions include:
  - It is generally better to overflow away from the sensors, not towards, since tracking is better further back (and closer might put you past the sensor position).
  - Users can attempt to use recenter to reach playable areas they would otherwise be unable to reach.

Controller Data

The Oculus SDK provides APIs that return the position and state for each Oculus Touch controller.

This data is exposed through two locations:

- ovrTrackingState::HandPoses[2]—returns the pose and tracking state for each Oculus Touch controller.
- ovrInputState—structure returned by ovr_GetInputState that contains the Oculus Touch button, joystick, trigger, and capacitive touch sensor state.

The controller hand pose data is separated from the input state because it comes from a different system and is reported at separate points in time. Controller poses are returned by the constellation tracking system and are predicted simultaneously with the headset, based on the absolute time passed into GetTrackingState. Having both hand and headset data reported together provides a consistent snapshot of the system state.

Hand Tracking

The constellation sensor used to track the head position of the Oculus Rift also tracks the hand poses of the Oculus Touch controllers.

For installations that have the Oculus Rift and Oculus Touch controllers, there will be at least two constellation sensors to improve tracking accuracy and help with occlusion issues.

The SDK uses the same `ovrPoseStatef` struct as the headset, which includes six degrees of freedom (6DoF) and tracking data (orientation, position, and their first and second derivatives).

Here's an example of how to get tracking input:

```
ovrTrackingState trackState = ovr_GetTrackingState(session, displayMidpointSeconds, ovrTrue);
ovrPosef         handPoses[2];
ovrInputState    inputState;
```

In this code sample, we call ovr_GetTrackingState to get predicted poses. Hand controller poses are reported in the same coordinate frame as the headset and can be used for rendering hands or objects in the 3D world. An example of this is provided in the Oculus World Demo.

Button State

The input button state is reported based on the HID interrupts arriving to the computer and can be polled by calling ovr_GetInputState.

The following example shows how input can be used in addition to hand poses:

```
double displayMidpointSeconds = ovr_GetPredictedDisplayTime(session, frameIndex);
ovrTrackingState trackState = ovr_GetTrackingState(session, displayMidpointSeconds, ovrTrue);
ovrPosef         handPoses[2];
ovrInputState    inputState;

// Grab hand poses useful for rendering hand or controller representation
handPoses[ovrHand_Left]  = trackState.HandPoses[ovrHand_Left].ThePose;
handPoses[ovrHand_Right] = trackState.HandPoses[ovrHand_Right].ThePose;

if (OVR_SUCCESS(ovr_GetInputState(session, ovrControllerType_Touch, &inputState)))
{
    if (inputState.Buttons & ovrButton_A)
    {
        // Handle A button being pressed
    }
    if (inputState.HandTrigger[ovrHand_Left] > 0.5f)
    {
        // Handle hand grip...
    }
}
```

The ovrInputState struct includes the following fields:

Field	Type	Description
TimeInSeconds	double	System time when the controller state was last updated.
ControllerType	unsigned int	Described by ovrControllerType. Indicates which controller types

Field	Type	Description
		are present; you can check the ovrControllerType_LTouch bit, for example, to verify that the left touch controller is connected. Options include: • ovrControllerType_None (0x0000) • ovrControllerType_LTouch (0x0001) • ovrControllerType_RTouch (0x0002) • ovrControllerType_Touch (0x0003) • ovrControllerType_Remote (0x0004) • ovrControllerType_XBox (0x0010)
Buttons	unsigned int	Button state described by ovrButtons. A corresponding bit is set if the button is pressed.
Touches	unsigned int	Touch values for buttons and sensors as indexed by ovrTouch. A corresponding bit is set if users finger is touching the button or is in a gesture state detectable by the controller.
IndexTrigger[2]	float	Left and right finger trigger values (ovrHand_Left and ovrHand_Right), in the range 0.0 to 1.0f. A value of 1.0 means that the trigger is fully pressed.
HandTrigger[2]	float	Left and right grip button values (ovrHand_Left and ovrHand_Right), in the range 0.0 to 1.0f. Hand trigger is often used to grab items. A value of 1.0 means that the trigger is fully pressed.
Thumbstick[2]	ovrVector2f	Horizontal and vertical thumbstick axis values (ovrHand_Left and ovrHand_Right), in the range -1.0f to 1.0f. The API automatically applies the dead zone, so developers don't need to handle it explicitly.
IndexTriggerNoDeadzone[2]	float	Left and right finger trigger values (ovrHand_Left and ovrHand_Right), in the range 0.0 to 1.0f, without a deadzone. A value of 1.0 means that the trigger is fully pressed.

Field	Type	Description
HandTriggerNoDeadzone[2]	float	Left and right grip button values (ovrHand_Left and ovrHand_Right), in the range 0.0 to 1.0f, without a deadzone. The grip button, formerly known as the hand trigger, is often used to grab items. A value of 1.0 means that the button is fully pressed.
ThumbstickNoDeadzone[2]	ovrVector2f	Horizontal and vertical thumbstick axis values (ovrHand_Left and ovrHand_Right), in the range -1.0f to 1.0f, without a deadzone.
IndexTriggerRaw[2]	float	Raw left and right grip button values (ovrHand_Left and ovrHand_Right), in the range 0.0 to 1.0f, without a deadzone or filter. A value of 1.0 means that the trigger is fully pressed.
HandTriggerRaw[2]	float	Left and right grip button values (ovrHand_Left and ovrHand_Right), in the range 0.0 to 1.0f, without a deadzone or filter. The grip button, formerly known as the hand trigger, is often used to grab items. A value of 1.0 means that the button is fully pressed.
ThumbstickRaw[2]	ovrVector2f	Horizontal and vertical thumbstick axis values (ovrHand_Left and ovrHand_Right), in the range -1.0f to 1.0f, without a deadzone or filter.

The ovrInputState structure includes the current state of buttons, thumb sticks, triggers and touch sensors on the controller. You can check whether a button is pressed by checking against one of the button constants, as was done for ovrButton_A in the above example. The following is a list of binary buttons available on touch controllers:

Button Constant	**Description**
ovrButton_A	A button on the right Touch controller.
ovrButton_B	B button on the right Touch controller.
ovrButton_RThumb	Thumb stick button on the right Touch controller.
ovrButton_X	X button on the left Touch controller.
ovrButton_Y	Y button on the left Touch controller.
ovrButton_LThumb	Thumb stick button on the left Touch controller.
ovrButton_Enter	Enter button on the left Touch controller. This is equivalent to the Start button on the Xbox controller.

Button Touch State

In addition to buttons, Touch controllers can detect whether user fingers are touching some buttons or are in certain positions.

These states are reported as bits in the Touches field, and can be checked through one of the following constants:

ovrTouch_A	User in touching A button on the right controller.
ovrTouch_B	User in touching B button on the right controller.
ovrTouch_RThumb	User has a finger on the thumb stick of the right controller.
ovrTouch_RThumbRest	User has a finger on the textured thumb rest of the right controller.
ovrTouch_RIndexTrigger	User in touching the index finger trigger on the right controller.
ovrTouch_X	User in touching X button on the left controller.
ovrTouch_Y	User in touching Y button on the left controller.
ovrTouch_LThumb	User has a finger on the thumb stick of the left controller.
ovrTouch_LThumbRest	User has a finger on the textured thumb rest of the left controller.
ovrTouch_LIndexTrigger	User in touching the index finger trigger on the left controller.
ovrTouch_RIndexPointing	Users right index finger is pointing forward past the trigger.
ovrTouch_RThumbUp	Users right thumb is up and away from buttons on the controller, a gesture that can be interpreted as right thumbs up.
ovrTouch_LIndexPointing	Users left index finger is pointing forward past the trigger.
ovrTouch_LThumbUp	Users left thumb is up and away from buttons on the controller, a gesture that can be interpreted as left thumbs up.

Haptic Feedback

In addition to reporting input state, Oculus touch controllers can provide haptic feedback through vibration.

The SDK supports two types of haptics:

- **Non-Buffered Haptics** – With this approach, haptics are controlled by simply turning vibrations on and off, while specifying a frequency (160Hz or 320Hz) and an amplitude (0 to 255). Non-buffered haptics are designed for simple effects that don't have tight latency requirements, since the controller requires 33ms to respond to the API call that modifies the haptics settings.

- **Buffered Haptics** – This approach enables a much wider variety of effects, such as patterning vibrational amplitudes around sine wave or tangent functions, panning the vibrations across controllers, generating a variety of low-frequency carrier waves, and more. With buffered haptics, your application sends a buffer of bytes to one or both controllers, where the bytes are interpreted as a series of amplitude values, from 0 (no vibrational amplitude) to 255 (maximum vibrational amplitude). The controller then "plays" the buffered amplitude values at 320Hz (or one byte every 3.125ms). Thus, you can control the vibrational amplitudes at a much finer degree of granularity when compared to non-buffered haptics (although it requires about 10ms from the time of the API call until the buffer is available to be played within the controller).



Note: The buffered and non-buffered functions should not be used together, as they will result in unpredictable haptic feedback.

Using Non-Buffered Haptics

Vibration can be enabled by calling `ovr_SetControllerVibration`:

```
ovr_SetControllerVibration( Hmd, ovrControllerType_LTouch, freq, trigger);
```

Vibration is enabled by specifying the frequency. Specifying 0.0f will vibrate at 160Hz. Specifying 1.0f will vibrate at 320Hz.

Buffered Haptics Overview

Buffered haptics enable you to create many cool effects beyond what is possible with non-buffered haptics. For example, the buffered haptics sample app that is provided with the PC-SDK (and which is described later in this section) produces the following effects:

- Smooth sine wave vibration with a "buzz down" effect at the end of each wave cycle
- Vibrational panning across the left and right controllers, again with a "buzz down" effect at the end of the panning cycle
- Ultra low-frequency buzz, essentially a series of ticks at 64Hz
- A "messed up" low frequency vibration based on a chaotic formula that utilizes a trigonometric tangent wave function

A buffer consists of a series of bytes with values from 0 to 255, where 0 represents no amplitude (i.e no vibration), and 255 represents the maximum amplitude (or intensity) of vibration that is allowed by the SDK. After your code fills in the values within a buffer, you send the buffer to one or both Touch controllers via `ovr_SubmitControllerVibration`. Each byte in the buffer is then "played" in sequence at a rate of 320Hz. The maximum buffer size (i.e. the maximum number of bytes that can be sent to a controller at one time, and also the maximum size of the controller's internal buffer) is 256 bytes. The length of time that it takes to "play" a single 256 byte buffer is 0.8 seconds (256 bytes played at a rate of 320Hz). So, you have full control over the amplitude of the vibrational effects down to a resolution of 3.125ms (which equates to 320Hz). However, the frequency can only be 320Hz or some integral quotient of 320Hz, such as 320/2=160Hz, 320/3=106.7Hz, 320/4=80Hz, 320/5=64Hz, etc. You can achieve these lower frequencies by sending bytes that are zero filled, interspersed with bytes that have amplitude values that are greater than zero. Here are some examples:

- 320Hz, full amplitude - [255, 255, 255, 255, ...]
- 160Hz, full amplitude - [255, 0, 255, 0, 255, 0, 255, 0, ...]
- 320Hz, half amplitude - [127, 127, 127, ..., 127, ...]
- 160Hz, half amplitude - [127, 0, 127, 0, 127, 0, …, 127, 0, ...]
- Single sharp tick (320Hz) - [0, 0, 255, 255, 255, 0, 0] [delay x ms] [0, 0, 255, 255, 255, 0, 0]
- Single blunt tick (160Hz) - [0, 255, 0, 255, 0, 255, 0] [delay x ms] [0, 255, 0, 255, 0, 255, 0]

In general, use the 320Hz resonant mode for lighter, sharper actions and the 160Hz mode for heavier, blunter actions.

In the above "sharp tick" and "blunt tick" examples, you can try varying the pulse count anywhere from 1 to 50 to obtain different lengths of effect. You can also try varying the amplitude to obtain different vibrational intensities. In general, you can vary both the amplitude and the frequencies in a more random or chaotic way. You can also vary the vibrational effects based on input streams, such as controller movement or position, or any other conditions or events within the VR experience.

In addition to the types of effects described above, there are many other possibilities, including:

- Hybrid Ticks and Modulation

  You can combine a repeating tick with an amplitude modulated segment.
- Mixing Multiple Input Streams

  You can pre-mix multiple input streams prior to passing to the haptics buffer API.

Note: Because of the relatively slow resonant decay of the haptics hardware within the controller, there is going to be a non-linear result based upon previous samples in a buffer. For example, if you want a sample to resonate at 50% amplitude but the previous sample was at 100% amplitude, you might need to drive the amplitude in a different way than if the previous sample was at 0% amplitude. An overdrive algorithm may improve the response. Positive (attack) and negative (decay) likely require different constants. A basic algorithm could simply factor in the sample-to-sample delta. A more advanced algorithm may need to use a weighted average of last several samples.

Note: Once a buffer is sent to a controller (via `ovr_SubmitControllerVibration`), that entire buffer will be "played". There is no way to halt the playing of a buffer after it has been sent to a controller. So, you should take care to only buffer up content that you know you wish to play within your VR application.

It is important to keep your sample pipeline at around the right size. Assuming a haptic frequency of 320 Hz and an application frame rate of 90 Hz, we recommend targeting a buffer size of around 10 samples per frame. This allows you to play 3-4 haptics bytes per frame, while preserving a buffer zone to account for any asynchronous interruptions. The more bytes you queue, the safer you are from interruptions, but you add additional latency before newly queued vibrations will be played.

Take care to not overflow or underflow the 256 byte internal buffer within the controller. If you call `ovr_SubmitControllerVibration` with a larger buffer than the number of bytes that are currently available within the (circular) internal buffer, the entire submitted buffer is discarded. On the other hand, if you submit bytes at a rate that fails to keep up with the consumption of buffered bytes within the controller, there will be gaps in the vibrational playback.

Using Buffered Haptics

To check the status of the buffer, call `ovr_GetControllerVibrationState`:

```
ovr_GetControllerVibrationState(ovrSession session, ovrControllerType controllerType,
 ovrHapticsPlaybackState* outState);
```

To submit to the buffer, call `ovr_SubmitControllerVibration`:

```
ovr_SubmitControllerVibration(ovrSession session, ovrControllerType controllerType, const
 ovrHapticsBuffer* buffer);
```

The following code sample shows how to produce some interesting and cool effects. This code extends the basic sample application contained in the Oculus PC-SDK distribution, in the following Visual Studio project `<install_folder>\Samples\OculusRoomTiny_Advance\ORT (Buffered Haptics)`. You can copy/paste this code into the project, overwriting main.cpp, if desired.

```
/// A sample to show vibration generation, using buffered input.
/// Press A to generate a sine wave vibration in the right controller.
```

```
/// Press B to generate a 320 Hz vibration that pans from the right controller to the left
 controller.
/// Press X to generate a low-frequency buzz at 64 hz (320 Hz / 5) in the left controller.
/// Press Y to generate a "messed up" low frequency vibration in the left controller, based on a
 tangent function.
/// Hold any of the buttons down in order to repeat the pattern continuously.
/// Note: In order to keep the sample minimal, the Touch controller is not graphically displayed
 within the VR scene.

#include "../Common/Win32_DirectXAppUtil.h" // DirectX
#include "../Common/Win32_BasicVR.h"        // Basic VR

struct BufferedHaptics : BasicVR
{
       BufferedHaptics(HINSTANCE hinst) : BasicVR(hinst, L"BufferedHaptics") {}

       void MainLoop()
       {
             Layer[0] = new VRLayer(Session);

             // We create haptic buffers that will be associated with the A, B, X, and Y buttons.
             int bufferSize = 256;
             unsigned char * dataBufferA = (unsigned char *)malloc(bufferSize);
             unsigned char * dataBufferBRight = (unsigned char *)malloc(bufferSize);
             unsigned char * dataBufferBLeft = (unsigned char *)malloc(bufferSize);
             unsigned char * dataBufferX = (unsigned char *)malloc(bufferSize);
             unsigned char * dataBufferY = (unsigned char *)malloc(bufferSize);

             // Verify that the buffer format is what we expect.
             ovrTouchHapticsDesc desc = ovr_GetTouchHapticsDesc(Session, ovrControllerType_LTouch);
             if (desc.SampleSizeInBytes != 1)        FATALERROR("Our assumption of 1 byte per
 element, is no longer valid");
             if (desc.SubmitMaxSamples < bufferSize) FATALERROR("Can't handle this many samples");
             desc = ovr_GetTouchHapticsDesc(Session, ovrControllerType_RTouch);
             if (desc.SampleSizeInBytes != 1)        FATALERROR("Our assumption of 1 byte per
 element, is no longer valid");
             if (desc.SubmitMaxSamples < bufferSize) FATALERROR("Can't handle this many samples");

             // Fill dataBufferA with a sine wave amplitude pattern that will smoothly
             // rise and fall over a cycle period of 0.8 seconds. The 0.8 value is
             // equal to 256/320, where 256 is the number of bytes that we will use
             // to define a single sine wave cycle, and the bytes are "played"
             // on the controller at 320 Hz. An interesting effect is added by lowering
             // the effective frequency in latter half of the cycle by setting
             // alternate intensities (amplitudes) to zero. The overall effect is a
             // smoothly repeating vibrational pattern that "buzzes down" at the end of
             // the wave cycle.
             for (int i = 0; i < bufferSize; i++)
             {
                    dataBufferA[i] = (unsigned char)(255.0f*(sin(((3.14159265359f*i) /
 ((float)bufferSize)))));
                    if ((i > bufferSize / 2) && (i % 2)) dataBufferA[i] = 0;
             }

             // Fill dataBufferBRight with values that decrement from 255 down to 0.
             // Similarly, fill dataBufferBLeft with values that increment from 0 up to 255.
             // An interesting effect is added by lowering the effective frequency in
             // latter half of dataBufferBLeft by setting intensities (amplitudes) to zero
             // for odd numbered bytes that are not divisible by 3. The overall effect
             // is a right-to-left vibrational pan, with a distinct "buzzing down" feeling
             // at the end of the panning cycle.
             for (int i = 0; i < bufferSize; i++)
             {
                    dataBufferBRight[i] = (unsigned char)(255.0f*(255 - i / ((float)bufferSize)));
                    dataBufferBLeft[i] = (unsigned char)(255.0f*(i / ((float)bufferSize)));
                    if ((i > bufferSize / 2) && ((i % 2) || (i % 3))) dataBufferBLeft[i] = 0;
             }

             // Fill dataBufferX with zeros, and set every fifth byte to 255. This creates
             // maximum amplitude ticks at 64 Hz. (This is calculated by dividing
             // the number of ticks in the buffer, 256/5 = 51.2, and dividing that
             // by the time period over which the buffer is played, 256/320 of a second,
             // which is 0.8 seconds.)
             for (int i = 0; i < bufferSize; i++)
             {
                    dataBufferX[i] = (unsigned char)0;
                    if (i % 5 == 0) dataBufferX[i] = 255;
             }

             // Fill dataBufferY with a tangent wave function that varies the intensity
```

```
            // amplitude) between 0 and 255, but set every other byte to 0. This produces a
            // "messed up" low frequency vibration.
            for (int i = 0; i < bufferSize; i++)
            {
                  dataBufferY[i] = (unsigned char)0;
                  if (i % 2 == 0) dataBufferY[i] = (unsigned char)
(255.0f*(tan(((3.14159265359f*i) / ((float)bufferSize)))));
            }

            // Create the SDK structures that contain the buffers, and prepare
            // them to be submitted to the controllers.
            ovrHapticsBuffer bufferX;
            bufferX.SubmitMode = ovrHapticsBufferSubmit_Enqueue;
            bufferX.SamplesCount = bufferSize;
            bufferX.Samples = (void *)dataBufferX;

            ovrHapticsBuffer bufferY;
            bufferY.SubmitMode = ovrHapticsBufferSubmit_Enqueue;
            bufferY.SamplesCount = bufferSize;
            bufferY.Samples = (void *)dataBufferY;

            ovrHapticsBuffer bufferA;
            bufferA.SubmitMode = ovrHapticsBufferSubmit_Enqueue;
            bufferA.SamplesCount = bufferSize;
            bufferA.Samples = (void *)dataBufferA;

            ovrHapticsBuffer bufferBRight;
            ovrHapticsBuffer bufferBLeft;
            bufferBRight.SubmitMode = ovrHapticsBufferSubmit_Enqueue;
            bufferBRight.SamplesCount = bufferSize;
            bufferBRight.Samples = (void *)dataBufferBRight;
            bufferBLeft.SubmitMode = ovrHapticsBufferSubmit_Enqueue;
            bufferBLeft.SamplesCount = bufferSize;
            bufferBLeft.Samples = (void *)dataBufferBLeft;

            // Main Loop
            while (HandleMessages())
            {
                  Layer[0]->GetEyePoses();

                  // Submit the haptic buffers to "play" upon pressing A, B, X or Y buttons.
                  ovrInputState inputState;
                  ovr_GetInputState(Session, ovrControllerType_Touch, &inputState);
                  if (inputState.Buttons & ovrTouch_A)
                  {
                        // Only submit the buffer if there is enough space available.
                        ovrHapticsPlaybackState playbackState;
                        ovrResult result = ovr_GetControllerVibrationState(Session,
 ovrControllerType_RTouch, &playbackState);
                        if (playbackState.RemainingQueueSpace >= bufferSize)
                        {
                              ovr_SubmitControllerVibration(Session, ovrControllerType_RTouch,
 &bufferA);
                        }
                  }
                  if (inputState.Buttons & ovrTouch_B)
                  {
                        // Only submit the buffers if there is enough space available.
                        ovrHapticsPlaybackState playbackState;
                        ovrResult result = ovr_GetControllerVibrationState(Session,
 ovrControllerType_LTouch, &playbackState);
                        if (playbackState.RemainingQueueSpace >= bufferSize)
                        {
                              ovr_SubmitControllerVibration(Session, ovrControllerType_LTouch,
 &bufferBLeft);
                        }
                        result = ovr_GetControllerVibrationState(Session,
 ovrControllerType_RTouch, &playbackState);
                        if (playbackState.RemainingQueueSpace >= bufferSize)
                        {
                              ovr_SubmitControllerVibration(Session, ovrControllerType_RTouch,
 &bufferBRight);
                        }
                  }
                  if (inputState.Buttons & ovrTouch_X)
                  {
                        // Only submit the buffer if there is enough space available.
                        ovrHapticsPlaybackState playbackState;
                        ovrResult result = ovr_GetControllerVibrationState(Session,
 ovrControllerType_LTouch, &playbackState);
```

```
                              if (playbackState.RemainingQueueSpace >= bufferSize)
                              {
                                    ovr_SubmitControllerVibration(Session, ovrControllerType_LTouch,
 &bufferX);
                              }
                        }
                        if (inputState.Buttons & ovrTouch_Y)
                        {
                              // Only submit the buffer if there is enough space available.
                              ovrHapticsPlaybackState playbackState;
                              ovrResult result = ovr_GetControllerVibrationState(Session,
 ovrControllerType_LTouch, &playbackState);
                              if (playbackState.RemainingQueueSpace >= bufferSize)
                              {
                                    ovr_SubmitControllerVibration(Session, ovrControllerType_LTouch,
 &bufferY);
                              }
                        }

                        // Just render a standard scene in the HMD, to keep the code as simple as
 possible.
                        // The controllers are not rendered in the scene.
                        for (int eye = 0; eye < 2; ++eye)
                        {
                              XMMATRIX viewProj = Layer[0]->RenderSceneToEyeBuffer(MainCam, RoomScene,
 eye);
                        }

                        Layer[0]->PrepareLayerHeader();
                        DistortAndPresent(1);
                  }

                  free(dataBufferX);
                  free(dataBufferY);
                  free(dataBufferA);
                  free(dataBufferBLeft);
                  free(dataBufferBRight);
            }
};
};
```

Emulating Gamepad Input with Touch

Touch controllers can partially emulate Microsoft XInput API gamepad input without any code changes. However, you must account for the missing logical and ergonomic equivalences between the two types of controllers.

How it Works

The Oculus runtime has a gamepad emulation mode to map Touch controller input to Microsoft XInput API input. This lets you allow users to use their Touch controllers in games designed for gamepad controllers without having to modify your code.

The Oculus runtime enables gamepad via Touch when all the following conditions are met:

- VR must have focus.
- The app must have the Gamepad Emulation Via Touch option enabled.
- The Touch controllers must be detected.
- A gamepad must not be detected. You cannot use Touch to emulate a second gamepad.

Touch controllers revert back to being Touch controllers when the app closes or when the user presses the Oculus button and exits to Home.

Activating and Deactivating Gamepad Emulation

Gamepad emulation is an Oculus Store attribute attached to the build binary. It can only be enabled or disabled by uploading a new build.

To set a gamepad emulation mode:

1. Upload a new Rift build from the Oculus Developer Dashboard.
2. On the build upload information page, select the appropriate **Gamepad Emulation Via Touch** option:
    - Off
    - Twin Stick
    - Left D-pad
    - Right D-pad

Gamepad Emulator Compatibility Guidelines

If your gamepad control mechanism is affected by any of the following limitations, your game is probably a poor candidate for using gamepad emulation. Instead of emulation, consider creating a new control scheme for Touch controllers to provide a better experience for your users.

Haptics

The current implementation does not include haptics. If vibration and haptics are critical to your experience, such as in a rhythm game, your app might not work well with emulation.

D-pad

The Touch controller does not have a D-pad. If your control scheme uses the D-pad and both sticks, your app might not work well with emulation.

If your control scheme only needs one stick, you can use our emulation modes to map the D-pad to either the right or left Touch.

View button

Twin stick emulation does not have a view button (#). If this button is critical to your control scheme, your app might not work well with twin-stick emulation mode.

The D-pad emulation modes do feature a mapping for this button.

Left Touch thumbstick can't be used with the X or Y buttons

Any action that combines left Touch thumbstick movement with X or Y button presses will be difficult or impossible for users because these actions are controlled by the same thumb. If your control scheme requires such an action, your app might not work well with emulation.

Testing Gamepad Emulation

You can force a gamepad emulation mode in the Oculus Runtime for testing purposes.

To force gamepad emulation:

1. Add or modify the following DWORD value in the Windows registry:

    ```
    [HKEY_CURRENT_USER\SOFTWARE\Oculus]
    ```

    ```
    "GamepadEmulationMode"=dword:mode_enum
    ```

    where *mode_enum* is one of the following:

    - `00000002` - Twin stick mode.

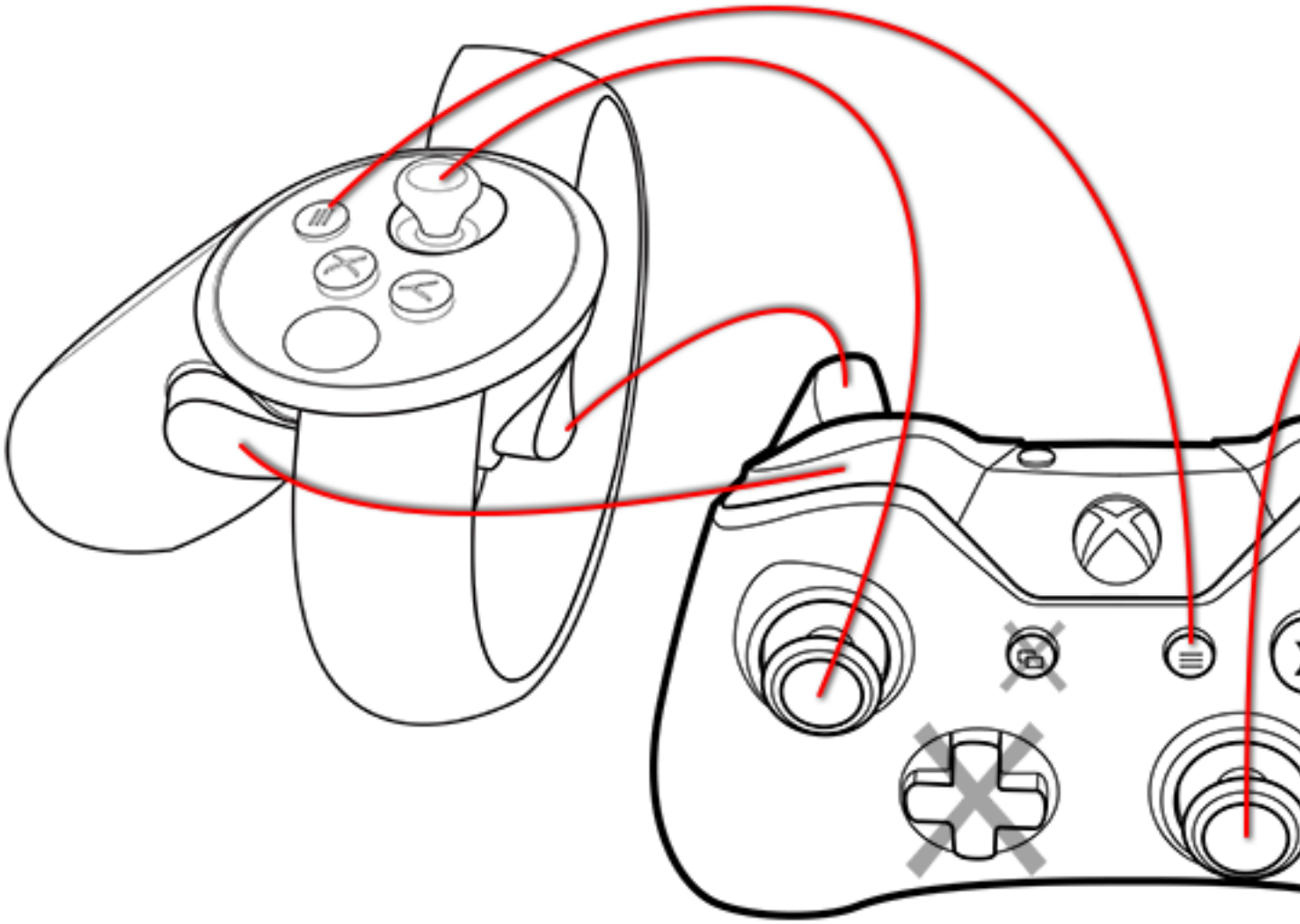
- `00000003` - Right D-pad mode.
- `00000004` - Left D-pad mode.

2. Restart the Oculus runtime service. The service name is "OVRService" and the display name is "Oculus VR Runtime Service".

To revert to normal behavior, delete the `GamepadEmulationMode` value from the Windows registry and then restart OVRService.

Twin Stick Gamepad Emulation Mode

Maps the two gamepad sticks to the Touch thumbsticks. The D-pad and view button (#) are not mapped.

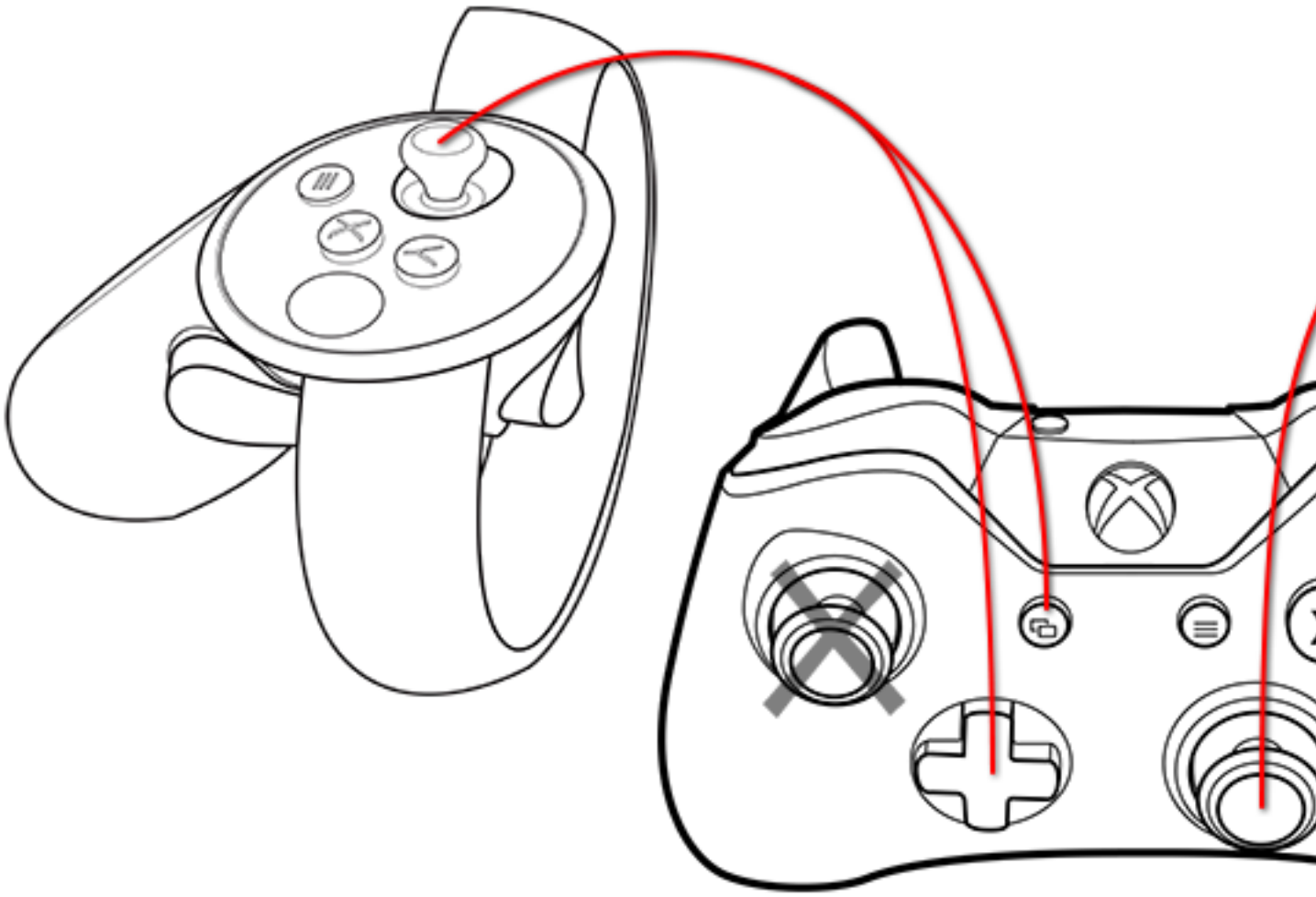


Gamepad Input	Touch Equivalent
vibration	NONE
D-pad	NONE
left stick	left thumbstick
left bumper	left grip
left trigger	left trigger

Gamepad Input	Touch Equivalent
view button (#)	NONE
Xbox button	Oculus button
menu button (☰)	menu button (☰)
right stick	right thumbstick
right bumper	right grip
right trigger	right trigger
A button	A button
B button	B button
X button	X button
Y button	Y button

Left D-Pad Gamepad Emulation Mode

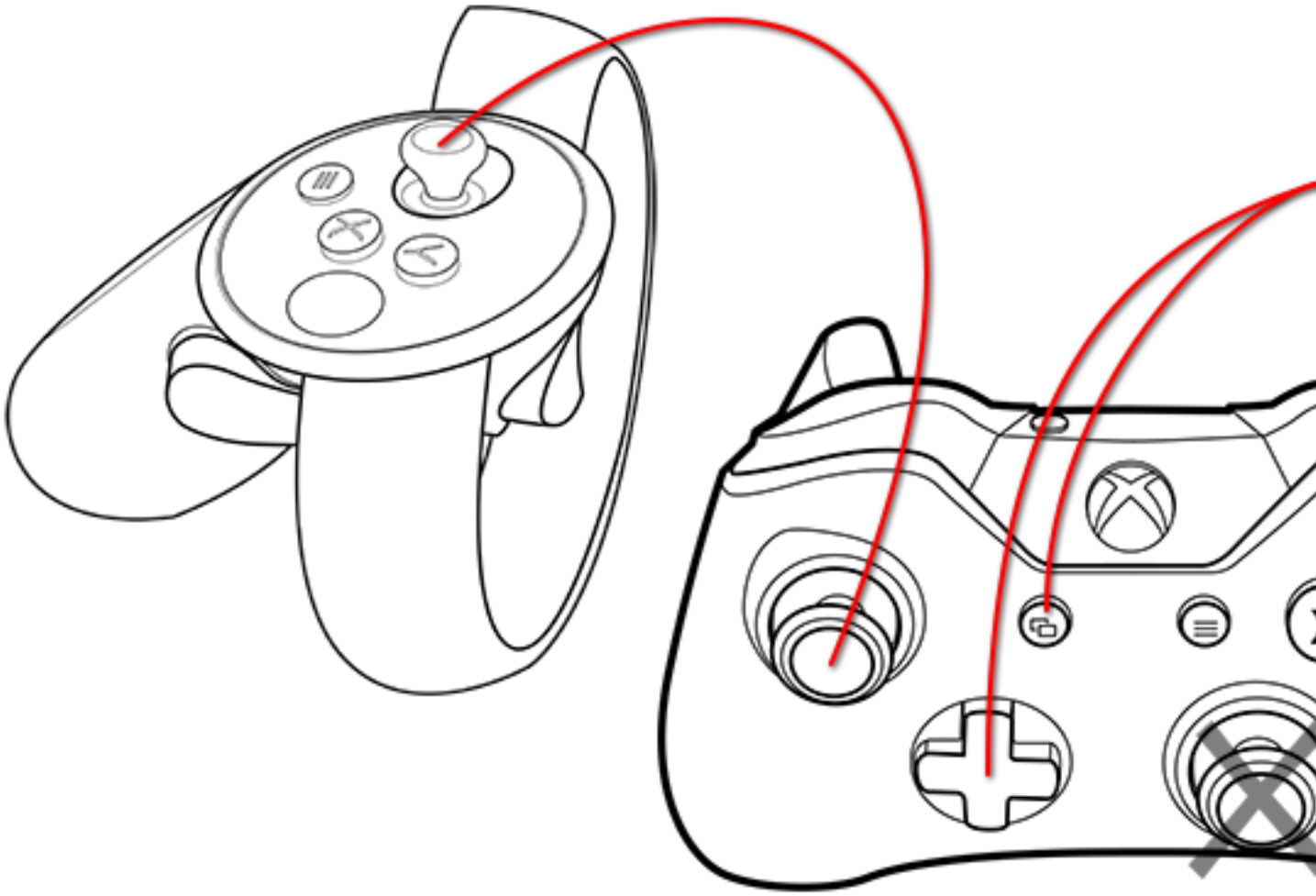
Same as twin stick gamepad emulation mode except for the following:



Gamepad Input	Touch Equivalent
D-pad	left thumbstick
left stick	NONE
view button (#)	left thumbstick press

Right D-Pad Gamepad Emulation Mode

Same as twin stick gamepad emulation mode except for the following:



Gamepad Input	Touch Equivalent
D-pad	right thumbstick
right stick	NONE
view button (#)	right thumbstick press

SDK Samples and Gamepad Usage

Some of the Oculus SDK samples use gamepad controllers to enable movement around the virtual world.

This section describes the devices that are currently supported and setup instructions.

Xbox 360 Wired Controller for Windows

To set up the controller:

- Plug the device into a USB port. Windows should recognize the controller and install any necessary drivers automatically.

Logitech F710 Wireless Gamepad

To set up the gamepad for Windows:

1. Put the controller into 'XInput' mode by moving the switch on the front of the controller to the 'X' position.
2. Press a button on the controller so that the green LED next to the 'Mode' button begins to flash.
3. Plug the USB receiver into the PC while the LED is flashing.
4. Windows should recognize the controller and install any necessary drivers automatically.

Optimizing Your Application

To provide the best user experience, your application must meet or exceed the minimum requirements to be considered for publication on the Oculus Store.

For more information about publishing requirements, see our *Publishing* documentation.

This section describes how to use tools we provide to optimize the performance of your application.

VR Performance Optimization Guide

This guide provides actionable guidance for tracking down and solving VR performance issues.

Why Performance Optimization?

In order to create the best VR experience for your users, it is important to optimize your applications for peak performance on Oculus recommended spec hardware. Otherwise, you may find that your applications exhibit judder, flickering black areas on the peripheries, or other performance-related problems.

In general, performance issues arise because it can be difficult for applications and game engines to keep up with Rift's refresh rate of 90 Hz. By carefully and systematically tracking down the issues that are causing performance problems and then implementing the necessary optimizations, you can create a significantly better overall user experience.

Scope of this Guide

This guide targets both application developers and engine developers.

This guide addresses a range of tools and methods. We start by covering high-level approaches that help you to narrow down the basic causes for the performance issues that your application may be exhibiting. Then we delve deeper into lower-level tools and analytical approaches, including Event Tracing for Windows (ETW), Windows Performance Analyzer (WPA), and GPUView. These lower-level approaches can be very helpful for solving complex and subtle VR performance issues, especially for game engine developers.

At the end of this guide, we provide a step-by-step tutorial that utilizes all of the tools discussed in this guide, and includes code samples and screenshots.

Guidelines for VR Performance Optimization

This section covers the general principles that you should follow in order to effectively optimize your VR applications.

Overview

Optimizing VR applications can be challenging. It is easy to go down the wrong path, and end up optimizing code that doesn't improve the overall performance of your application. It is important to identify and focus on where the bottlenecks really are, and optimize those sections first.

VR performance issues are generally of two types: CPU issues and GPU issues. The CPU tends to be involved with simulation logic, state management, and generating the scene to be rendered. The GPU tends to be involved with sampling textures and shading for the meshes in your scenes. It is important to determine whether a performance problem is due to CPU load or GPU load, and to optimize your code accordingly.

In general, you follow Amdahl's Law for parallel programming: Optimize the sections that are utilizing the system the most. Focus on the big expensive code paths. This document will provide guidance on how to identify these. Don't focus on issues which, even if you reduce the costs to near zero, you would only achieve minor reductions to the overall performance costs.

It is not uncommon for one area within your application to utilize a large percentage of the system's processing time, while the remaining areas consume much smaller percentages. You should aim to optimize the larger problem area first.

As you optimize your application, strive to change one thing at a time. Keep in mind that there can be non-trivial interactions between the changes you make, especially with complex VR applications. If you are tracking down a performance regression in your application, try to locate in your version control software history the single change that caused the performance problem you are experiencing. Then, look for the root cause of the performance issue there. Don't assume that multiple changes work together to cause a single performance problem.

For many people, it is easy to get a bit pedantic or obsessive about things that don't matter in terms of bottom line performance. It is usually best to simply consider timing issues: Is the application hitting frame rate? If so, you may not need to further optimize your application, even if your code is not designed as well as would be ideal. Focus on what really counts, in terms of performance issues that impact the user experience.

Techniques for Hitting Frame Rate

With VR, every frame must be typically drawn twice, once for each eye. That typically means that every draw call is issued twice, every mesh is drawn twice, and every texture is bound twice. There is also a small amount of overhead that is required to apply distortion and TimeWarp to the final output frame (approximately 2 ms per frame). Since the Rift refreshes frames at 90 Hz, it can be challenging to hit fame rate consistently.

The following general guidelines can help you to meet frame rate:

- Limit each frame to a maximum of 500-1,000 draw calls
- Limit each frame to a maximum of 1-2 million triangles or vertices
- Use as few textures as possible, although they can be large. Smaller working sets, texture compression, and mipmapping will minimize texture bandwidth consumption.
- Limit the time spent in script (or other logic) execution to 1 ~ 3 ms, for example when running Unity Update()
- Systems are full of surprises, so always run a profiler to understand the way your application is using resources.
- Don't optimize too early in the development process. Simplify the code first. Conversely, don't ignore obvious performance issues when you identify them.
- Don't rely on unproven technology or techniques that are not known to be performant.
- Everything is relative. Compare apples to apples.
- Change one thing at a time: resolution, hardware resources, image quality, etc.
- Some artifacts are worse than others. Dropped frames that cause discomfort are not worth better quality graphics.
- Don't rely on Asynchronous SpaceWarp (ASW) to hit rendering frame rate. ASW generates intermediate frames based on very recent head pose information, if your application begins to drop frames. It works by distorting the previous frame to match the more recent head pose. While ASW will help smooth out a few dropped frames now and then, applications must meet a consistent 90 frames per second (FPS) on a recommended spec machine and maintain 45 frames per second on a minimum spec machine to qualify for the Oculus Store.
- Due to higher resolution and GPU load, the CPU tends to be less of a bottleneck on the Rift, when compared with mobile VR devices.

- Graphical styles with simple shaders and relatively few polygons can often provide just as good of a VR experience as photorealistic graphics, which typically require significantly more processing in order to render each frame.
- Use techniques such as Level of Detail (LOD), culling, and batching.
- Cut the shading rate by scaling eye buffers, and using Oculus octilinear rendering (which leverages NVIDIA Lens Matched Shading).
- Use projector shadows to save bandwidth.
- When you are rendering to the cascaded shadow map, which can cost a lot in terms of bandwidth, consider the resolution and the number of cascades you are using. Try not to use expensive filtering. However, this approach pretty quickly ends up producing lower quality graphics. So you might use projector shadows.
- Use simplified shader math and baked shading if necessary.

Common Causes of Performance Problems

Performance problems are most commonly caused by the following issues (in order of severity):

Performance Problem	Resource Costs
Scenes that require dependent renders, including shadows and reflections	CPU, GPU
Binding of Vertex Buffer Objects (VBOs) in order to issue draw calls	CPU, Graphics Driver
Transparency, multi-pass shaders, per-pixel lighting, and other effects that fill large numbers of pixels	GPU
Large texture loads, blits, and other forms of memcpy	GPU, Memory Controller
Skinned animation	CPU, GPU
Unity garbage collection overhead	CPU

Workflows: The process flows you should follow

This section covers the workflows that you should use when tracking down performance problems.

Overview

This section provides an overview of the workflows that you should follow when isolating and analyzing VR performance issues. Also see these slides from the 2015 Unity Unite conference: *https://imgur.com/a/SVs3I*.

Follow these workflows:

Begin

1. Choose a workload, such as a scene with judder in your application.
2. View the scene while displaying the Oculus Performance HUD, or capture the scene in the Lost Frames Capture utility. Both of these tools are available within the Oculus Debug Tool. You can also use NVIDIA Frame Capture Analysis Tool for VR Games (FCAT VR).
3. IF dropped_frame_rate > 1 frame per 5 seconds THEN
    - IF render_time > 8 ms THEN *Analyze Rendering* ELSE *Analyze Call Hierarchy*.

Analyze Rendering

1. IF rendering time is excessive THEN
    - Check if batches > 1000
    - Check if triangles/vertices > 1,000,000

- Check if SetPass calls > 1000

2. IF the hierarchy view uses excessive time THEN

- Look for a single object that takes 8 ms or more to render
- Check to see if the CPU is stalled waiting on the GPU to complete tasks
- Check if mesh rendering is too complex
- Check to see if shadows are too complex
- Check to see if VSync points arrive before the frames are fully rendered
- Check the timeline view to see if there are scheduling bubbles

Analyze Call Hierarchy

1. Capture an ETW trace file, using `ovrlog` or `ovrlog_win10`.
2. Load the trace file into Windows Performance Analyzer (WPA).
3. Expand to see which objects/calls take the most time.
4. IF there are garbage collection spikes THEN don't allocate memory for each frame. Note: If your scripts allocate and free memory during each frame cycle, this may result in fragmentation that eventually forces the garbage collection process to defragment the heap.
5. Once you have identified the objects/calls that take the most time, analyze the corresponding source code and optimize it.

Analyze Queuing and System-Level Contention

1. Capture the Event Tracing for Windows (ETW) output by running `ovrlog` or `ovrlog_win10`
2. Analyze the resulting `merged.etl` file in GPUVIew.
3. Highlight processes within your application.
4. Show the VSyncs.
5. Zoom in on the problem area.
6. Examine the GPU packets, command packets, and fences.
7. Analyze the dependencies between these packets and the processes they are implementing. You can bring up the Event Viewer, which displays detailed information about each packet.

Note: GPU packets represent indivisible units of GPU work. Command packets represent indivisible packets of CPU work. And fences represent barriers that cannot be passed until the previous work is completed.

The tutorial at the end of this guide provides detailed guidance for using WPA and GPUView.

Performance Optimization Tools

This section covers the tools that you should use when tracking down performance problems.

The reason for discussing a number of different performance optimization tools in this guide is to provide guidance regarding the workflow, or the decision tree, that you need to follow when triaging and resolving issues. Typically, when you are working on a graphics problem, your application may be crashing, or you may see a black screen, or you may experience slow performance, perhaps due to high latency issues. However, there are many conditions that can lead to the same types of problems. So you are often interested in finding the root causes of the symptoms that you are seeing. The tools that are introduced in this section allow you to quickly rule out large classes of problems, and then to drill down to the underlying issues.

The following optimization tools help you to isolate and resolve performance problems:

Lost Frame Capture

This tool captures eye displays for any frames that your application drops, and lets you replay just those frames while viewing performance statistics and graphs. This helps you to quickly get a sense where your application is failing to maintain frame rate. In many cases, this may be sufficient for you to discern what the problem is. Lost Frame Capture is an offline analytical tool: you place the tool into capture mode, run your VR application, stop the capture mode, and then step through the dropped frame content offline, as desired. You can also export the captured content by saving it to an Oculus Debug Archive (ODA) file. This makes it possible to easily share the lost frame data with others who can then reproduce the scenarios where frames are dropped, and help to diagnose the underlying issues. For more information, see *Lost Frame Capture*.

NVIDIA Frame Capture Analysis Tool for VR Games (FCAT VR)

FCAT VR is a tool that is provided by NVIDIA. It can be used with any GPU hardware, however. One component of the FCAT VR Capture tool executes on your PC. It uses event tracing data that is generated by Event Tracing for Windows (ETW). (The process of capturing ETW event tracing data is described in *Tutorial: Optimizing a Sample Application* in this guide.) A second component of FCAT VR imports the event tracing data that is captured by ETW and displays charts that help you to analyze frame timing, dropped frames, warped frames, synthesized frames (Asynchronous SpaceWarp), reprojection, and other issues. For more information, see *Frame Capture Analysis Tool for VR Games*.

Oculus Debug Tool

This tool provides a live heads-up display, called the Oculus Performance HUD. This information display can help you to associate performance issues with specific contexts within your VR experience, and to experiment with those situations in real time. You can view statistics such as how many compositor frames are being dropped, how much time the application is taking to render frames, how much time the compositor is taking to render frames, and so forth.

The Oculus Performance HUD enables you to quickly rule out large classes of possible issues, since you don't want to spend effort enhancing the performance of a component that doesn't impact the overall performance of the application. For example, if your application is dropping frames, and the CPU profile shows a large burst of usage at some point during a frame cycle, then you don't even want to consider GPU issues (unless the CPU is waiting for the GPU).

Once you have made a determination such as this, you can do important lower-level analysis by taking an ETW trace and analyzing that trace with Windows Performance Analyzer (WPA) or GPUView. For example, suppose your application is CPU bound, and is stalling around physics processing and draw call submissions. Even though the application is CPU bound, it can have a stack in the draw call generation, in which case you may want to use more batching. A tool such as GPUView can help you to narrow down this type of issue very precisely, so that you know exactly what needs to change in your code. For more information, see *Oculus Debug Tool* and *Performance Head-Up Display*.

SDK Statistics

You can also call the SDK directly to obtain the same statistics displayed in the Oculus Performance HUD, and then utilize those statistics within your code as desired. For example, you could write specific statistics to a console output, or invoke a debugger when a certain condition arises. For more information, see *SDK Performance Statistics*.

Performance Profiler

This analytical tool produces graphs based on the same types of statistical data that are available directly from the SDK and within the Oculus Performance HUD. You might use these graphs if you need to analyze patterns that play out over a given time period. For more information, see *Performance Profiler*.

Event Tracing for Windows (ETW)

ETW is a trace utility for performance analysis. It collects event data while the VR application is running, and then saves that data to event trace log (.etl) files. Performance analysis using ETW is centered on the events generated by the Windows kernel, which provides extensive details about the operation of the system.

ETW profiles the entire system, not just the GPU. The full ETW documentation is located here: *https://msdn.microsoft.com/en-us/library/windows/desktop/bb968803*.

In order to analyze the ETW traces, you will typically use either Windows Performance Analyzer (WPA), or GPUView, or both.

Windows Performance Analyzer (WPA)

WPA provides a top-down view of the trace output, including contextual information that helps you to understand the system load.

When WPA loads a trace, the view it shows of that trace is a hierarchy of events that led to each other. For example, you might find that 90% of the time, your application is rendering a camera. And, 90% of that time was spent drawing bushes. Within that, 80% of the time is involved with rendering work for the foliage, and 30% of the time is spent on alpha blending. In this example, you may decide that you need to change the shading so that alpha blending is computationally cheaper. You would expect that this approach will improve the performance of the application as a whole.

GPUView

GPUView provides a lower-level view of the trace output, and lacks the contextual information that WPA provides. However, GPUView provides insights into the interaction between the CPU and the GPU which cannot be obtained by using WPA. Since GPUView lacks contextual information, it can be difficult to locate specific points in your application's life cycle. However, you can use the two tools in a complementary way. For example, you can zoom in with WPA and locate the exact VR frame index that you are interested in, and then use this information to locate the same frame within GPUView.

When you are using Unity or Unreal, you can take advantage of their built-in profilers in order to debug most bottlenecks. For example, if your shaders are too complex, or your scripts are running for too long, those profilers will help you to track down the issue. But if your application is exhibiting a lot of contention and synchronization issues, or if there is judder that isn't explained by application utilization, GPUView can allow you to look more closely at the queuing in the system. This enables you to find inefficient policies or bad synchronization mechanisms. For example, if you are using a vertex buffer that is too large, it may cause page faults. So the GPU might be fetching from system memory, and that might tell you that your meshes are too complex.

GPUView helps you to resolve the following types of issues:

- Why is the application missing VSync intervals?
- Are new surface allocations stalling the GPU and causing the frame stuttering problem we are observing?
- Will optimizing the CPU code improve performance, or do we need to reduce the amount of work we send to the GPU?
- Are we sending graphics tasks to the GPU early enough in the frame, or is the GPU idle while waiting on our CPU code?

ovrlog_win10

This script is used to start and stop ETW tracing sessions on Windows 10. (On previous versions of Windows, use `ovrlog`.) The ETW trace output is used as the input to WPA and GPUView. The `ovrlog_win10` and `ovrlog` scripts call the `xperf` tool which initiates the event capture process, sets up file paths, injects all the events that are relevant to Oculus applications, and then turns off event tracing when done.

`ovrlog_win10` and `ovrlog` are designed to capture kernel-level and application-level events, including:

- Application start times
- Interrupt activity
- System responsiveness issues
- Application resource utilization
- Interrupts

- and more

`ovrlog_win10` and `ovrlog` are located here:

```
%PROGRAMFILES%\Oculus\Support\oculus-diagnostics\ETW> ovrlog_win10.cmd
```

Both `ovrlog` and `ovrlog_win10` are shipped with the Oculus runtime.

A VSync (vertical synchronization) is a timing point that prevents any changes to the display memory until after the display finishes its current refresh cycle. VSyncs are generally your most important navigational unit when optimizing VR applications.

The typical `ovrlog` workflow is:

1. Generate an ETW trace while running the Oculus application.
2. Load the trace file into WPA, and locate the frame index where the problem is occurring.
3. Load the trace file into GPUView, and locate the VSync for the desired frame index
4. Zoom in around the frame index, perhaps showing a surrounding 10-frame interval.
5. Compare the problem frame to a nearby healthy frame, and in particular examine how the frames are scheduling CPU and GPU resources.
6. Match the CPU/GPU work against the functions that they are performing within your application (by clicking on the packets, and thereby highlighting the corresponding application-level work that they are performing).
7. Infer from this what is causing the performance issue, such as a vertex buffer that is too large.

For a detailed walkthrough of this procedure, see the tutorial, below.

Tutorial: Optimizing a Sample Application

This section is a tutorial that provides a detailed hands-on guide to VR performance optimization.

This tutorial leads you through the process of optimizing a sample application by using:

- Oculus Debug Tool
- Oculus Performance HUD
- Lost Frame Capture Tool
- Event Tracing for Windows (ETW)
- `ovrlog`/`ovrlog_win10` and `xperf`
- Windows Performance Analyzer (WPA)
- GPUView

Install Components

1. Install the Oculus PC-SDK:

https://developer.oculus.com/documentation/pcsdk/latest/concepts/book-gsg/

2. Install Visual Studio 2015:

https://msdn.microsoft.com/en-us/library/mt613162.aspx

Note: The Oculus ORT demo applications which will be used in this tutorial and the Windows Driver Kit (WDK) are not yet compatible with Visual Studio 2017.

3. Install GPUView.

GPUView is included in the Windows Performance Toolkit (WPT), which is included in the Windows 8.1 SDK:

https://developer.microsoft.com/en-us/windows/downloads/windows-8-1-sdk

4. Install Event Tracing for Windows (ETW).

ETW is included in the Windows Driver Kit 8.1: *https://developer.microsoft.com/en-us/windows/hardware/windows-driver-kit*.

5. Install both of the following:

- Windows SDK for Windows 10, version 1703 (or later)
- WDK for Windows 10, version 1703 (or later)

6. Install the Windows Performance Analyzer (WPA).

WPA is included in the Windows Assessment and Deployment Kit (Windows ADK):

https://developer.microsoft.com/en-us/windows/hardware/windows-assessment-deployment-kit

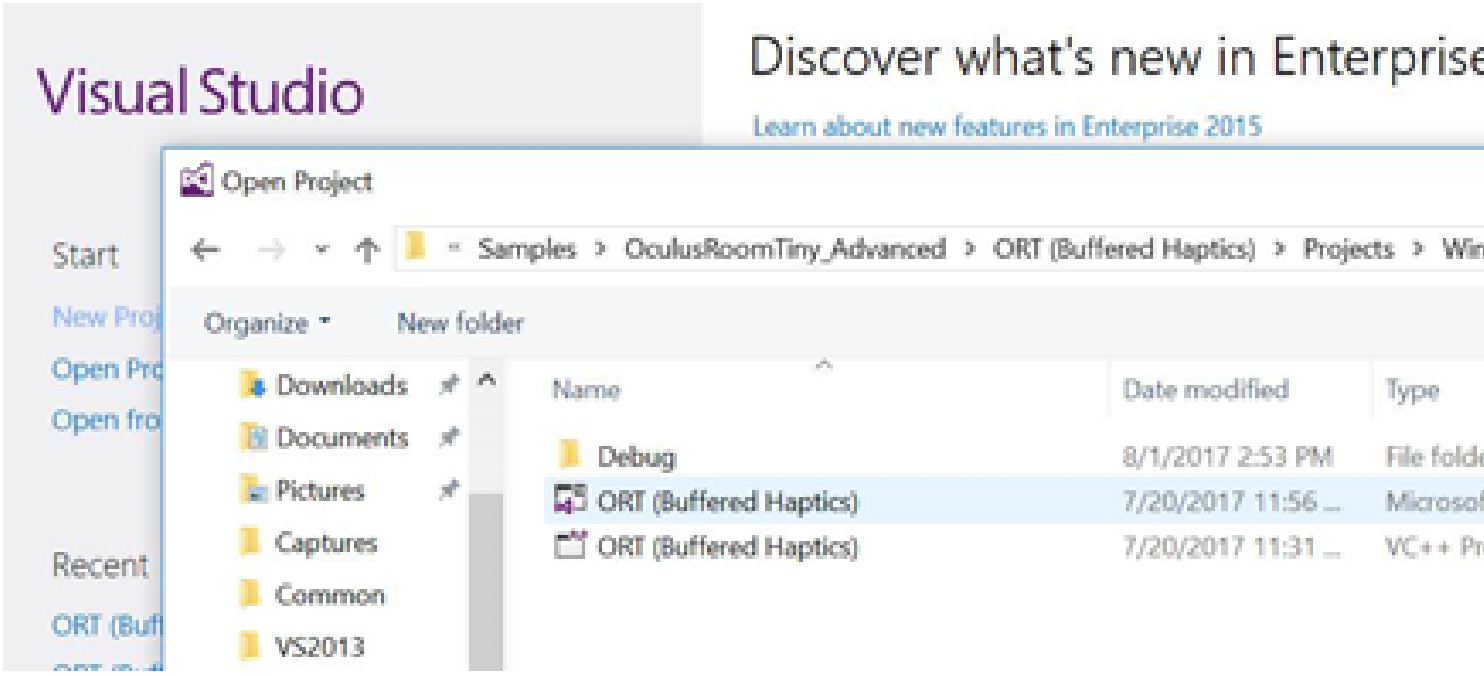
Create an Application with a Performance Issue

We will work with the ORT Buffered Haptics sample project (which is located under the VS2013 folder at this writing):

```
<SDK Folder>\Samples\OculusRoomTiny_Advanced\ORT (Buffered Haptics)\Projects\Win
\VS2013
```

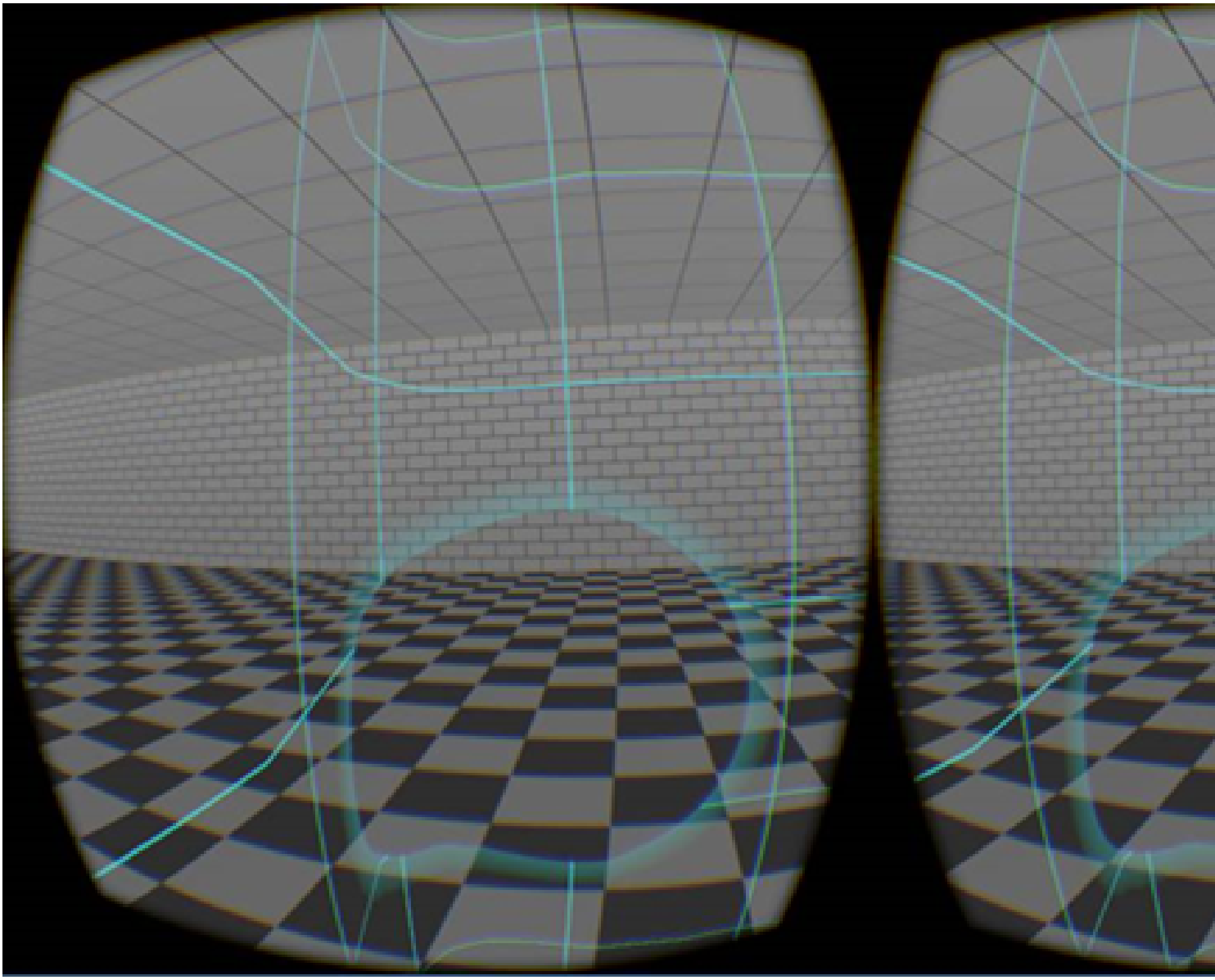
1. Run the Oculus application, and assure that your Rift is set up and working properly.

2. Open the following *solution* (not the project) in Visual Studio 2015:



3. Run this application in the Debugger (press F5), and view the scene in the Rift headset.

As you turn your head, the scene behaves normally. In the Oculus Mirror, the eye displays remain centered and retain their usual shape. (In this example, a penetrated Oculus Guardian System boundary is shown.)



4. Stop the Debugger.

5. Create a GPU performance issue.

We will do this by editing the default texture shader routine that is used by this application, and adding a For loop that artificially overloads the GPU with an excessive amount of work, mimicking an expensive shader routine.

In this sample, the shader routine is contained in:

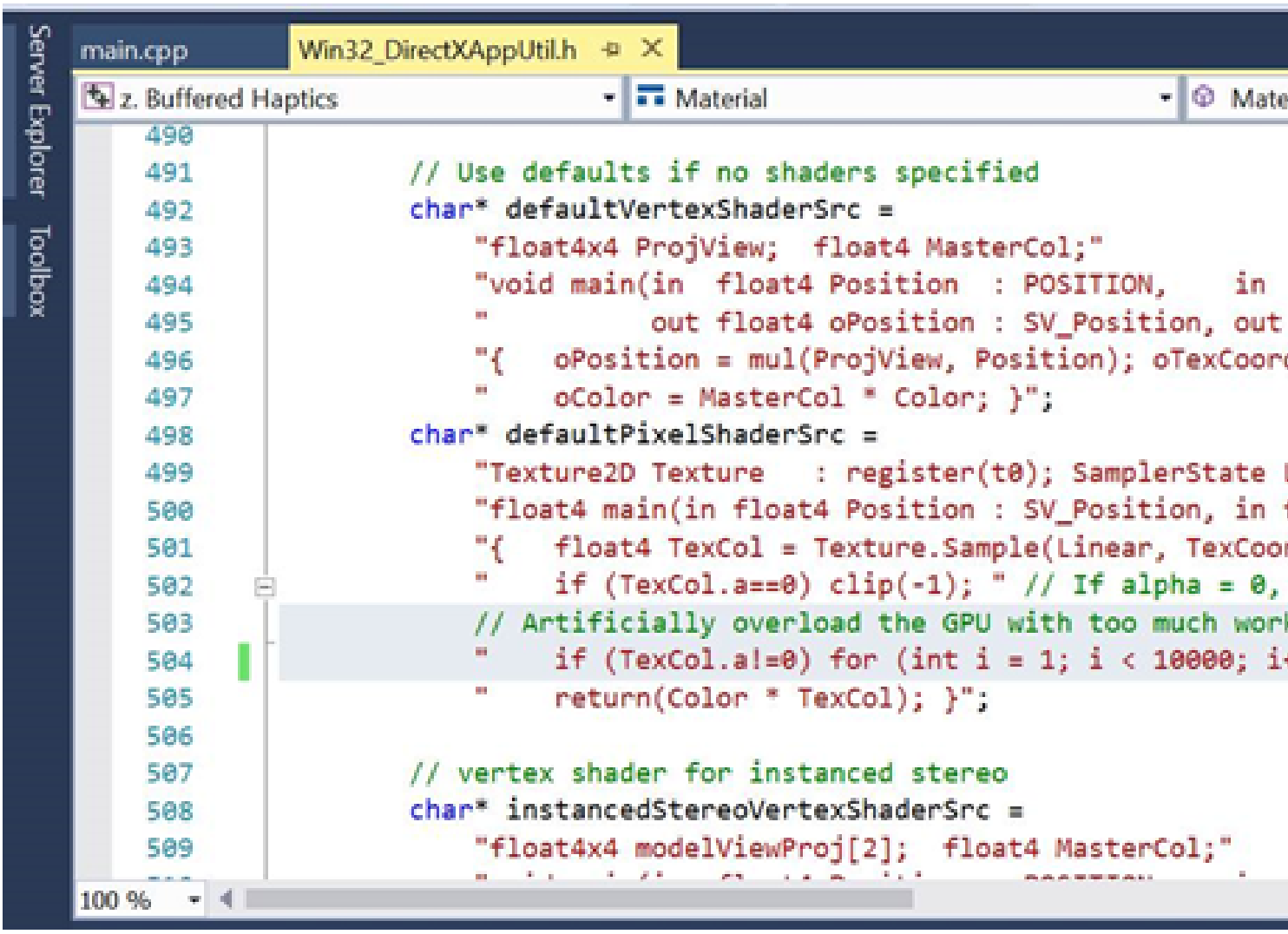
```
<Oculus SDK Folder>\Samples\OculusRoomTiny_Advanced\Common\Win32_DirectXAppUtil.h
```

Open this header file in Visual Studio, and add the following code to defaultPixelShaderSrc:

```
// Artificially overload the GPU with too much work, mimicking an expensive shader
routine

" if (TexCol.a!=0) for (int i = 1; i < 10000; i++) { Color = Color * i * 2; Color =
Color/i; Color = Color/2; } "
```

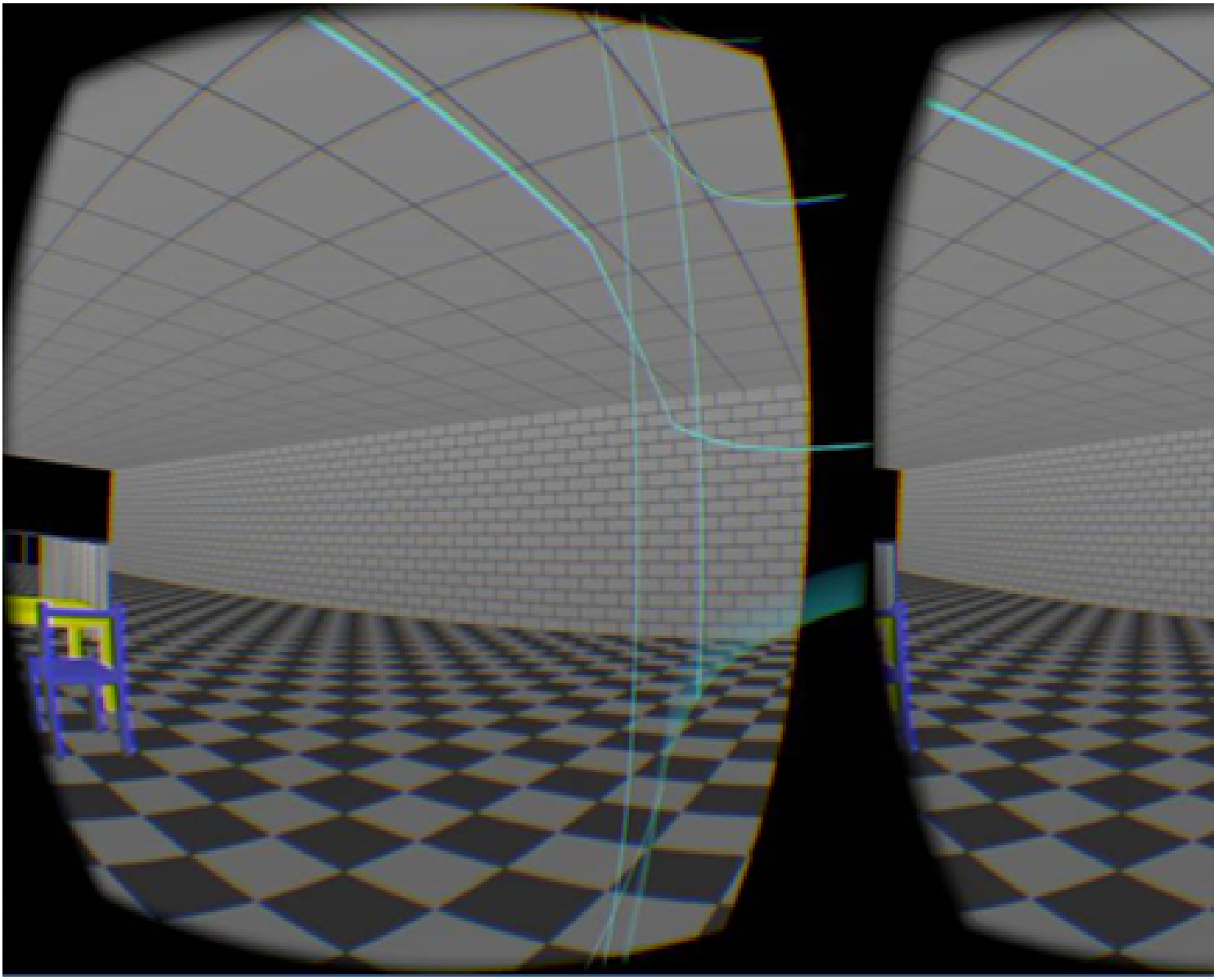
The location for this code is highlighted below:



Note: You may wish to try different values for the loop variable. The 10,000 value shown above may be too extreme, depending upon the characteristics of your hardware. You might find that the tutorial works better with a value around 1500. Later in the tutorial, results will be shown with different values for that loop variable. This mimics different levels of shader complexity.

6. Run the application again.

The result is a delay in eye rendering that is noticeable when you turn your head. In the headset, black vertical rectangular areas appear on the sides of the displays during head movement. In the Oculus Mirror, the eye displays appear flattened and off center as you turn your head.



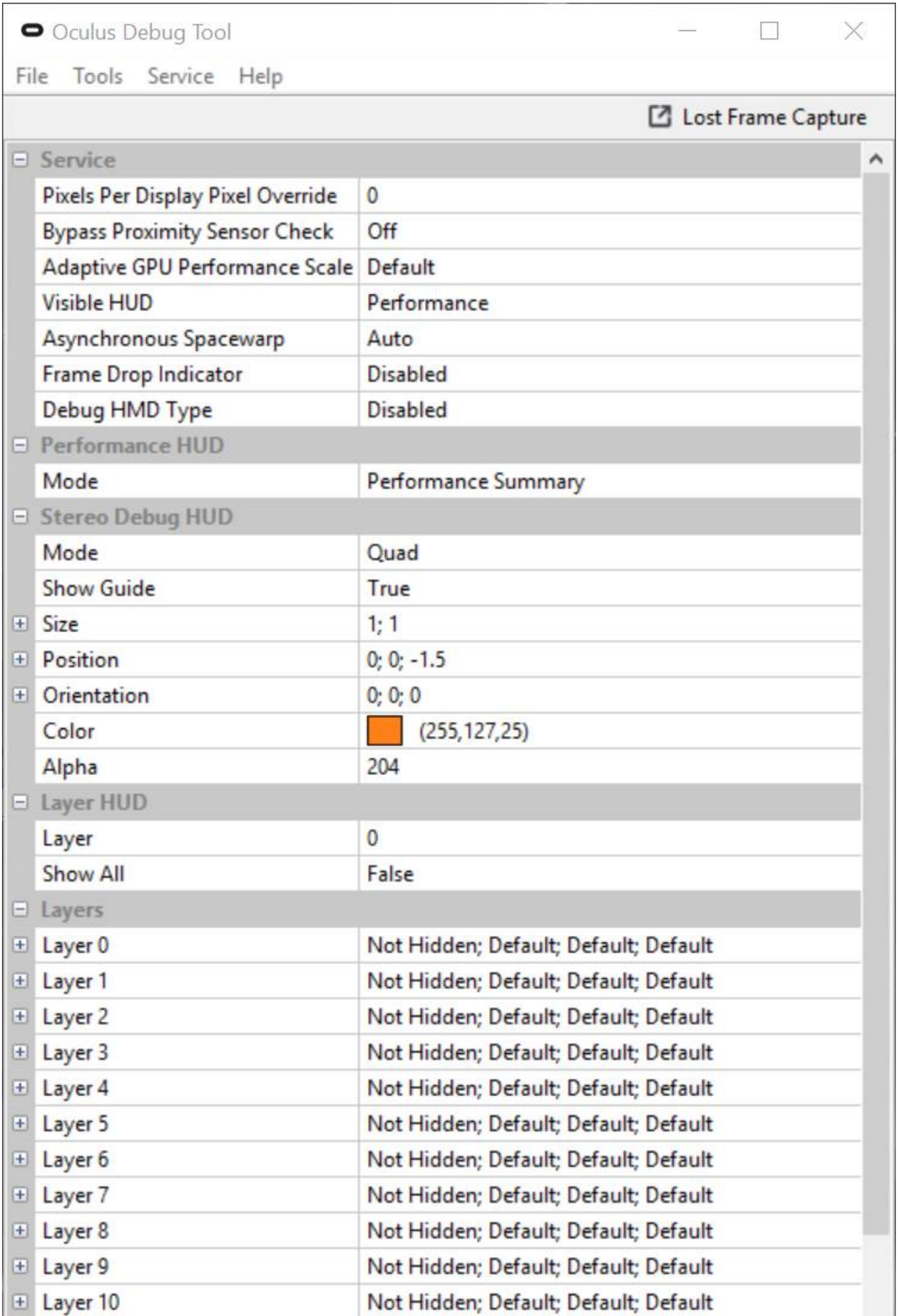
In this example, we know the problem is due to the GPU issue which we purposefully created. However, in general, this symptom could be due to a number of issues, including CPU or GPU overload. We will begin to narrow down the problem using the Oculus Debug Tool.

Using Oculus Debug Tool

1. Run the Oculus Debug Tool:

```
%PROGRAMFILES%\Oculus\Support\oculus-diagnostics\OculusDebugTool
```

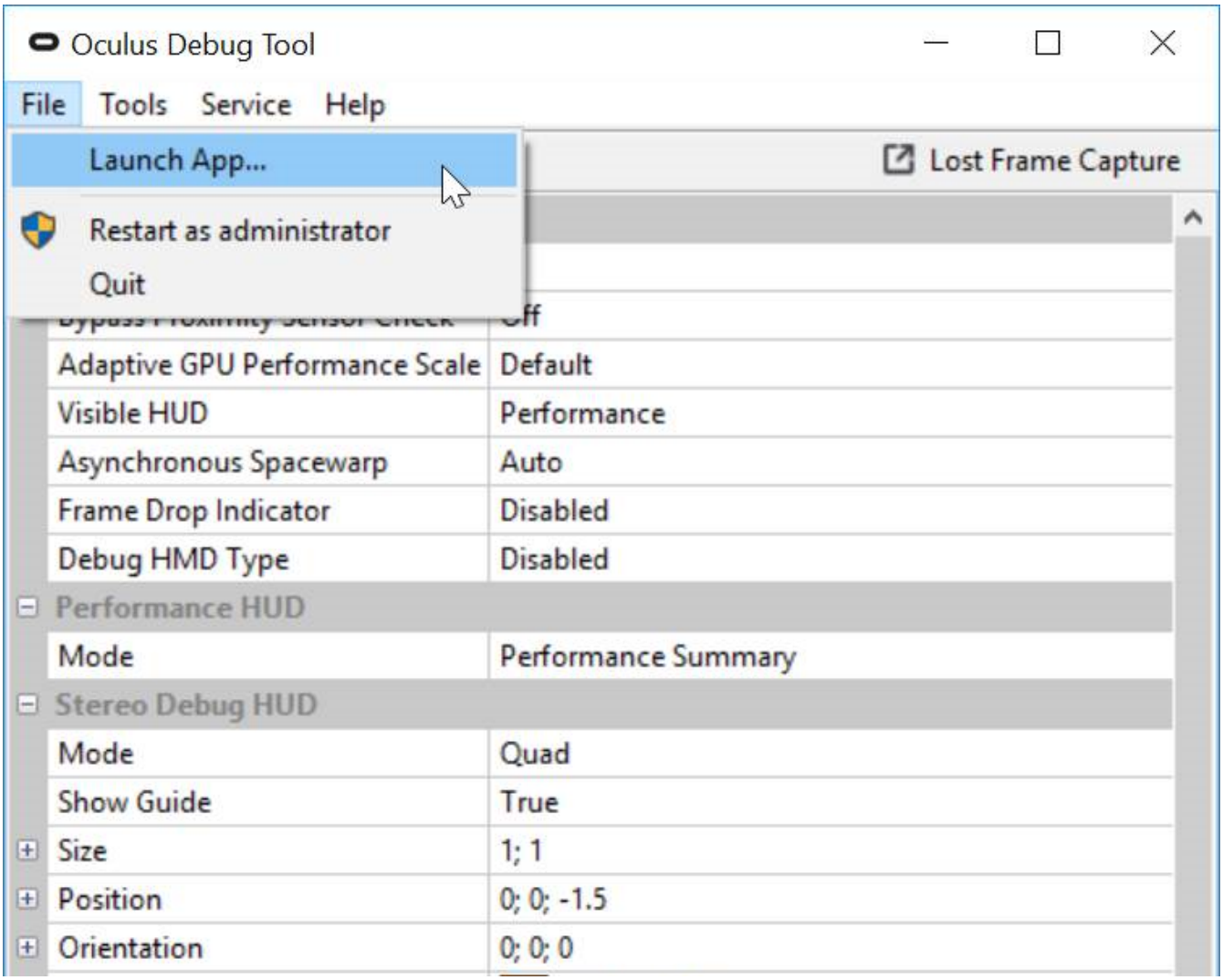
2. The Oculus Debug Tool window appears:


Oculus Debug Tool
File Tools Service Help
Lost Frame Capture
Service
Pixels Per Display Pixel Override 0
Bypass Proximity Sensor Check Off
Adaptive GPU Performance Scale Default
Visible HUD Performance
Asynchronous Spacewarp Auto
Frame Drop Indicator Disabled
Debug HMD Type Disabled
Performance HUD
Mode Performance Summary
Stereo Debug HUD
Mode Quad
Show Guide True
Size 1; 1
Position 0; 0; -1.5
Orientation 0; 0; 0
Color (255,127,25)
Alpha 204
Layer HUD
Layer 0
Show All False
Layers
Layer 0 Not Hidden; Default; Default; Default
Layer 1 Not Hidden; Default; Default; Default
Layer 2 Not Hidden; Default; Default; Default
Layer 3 Not Hidden; Default; Default; Default
Layer 4 Not Hidden; Default; Default; Default
Layer 5 Not Hidden; Default; Default; Default
Layer 6 Not Hidden; Default; Default; Default
Layer 7 Not Hidden; Default; Default; Default
Layer 8 Not Hidden; Default; Default; Default
Layer 9 Not Hidden; Default; Default; Default
Layer 10 Not Hidden; Default; Default; Default

Make sure:

- **Visible HUD** is set to **Performance**
- Under **Performance HUD**, the **Mode** is set to **Performance Summary**

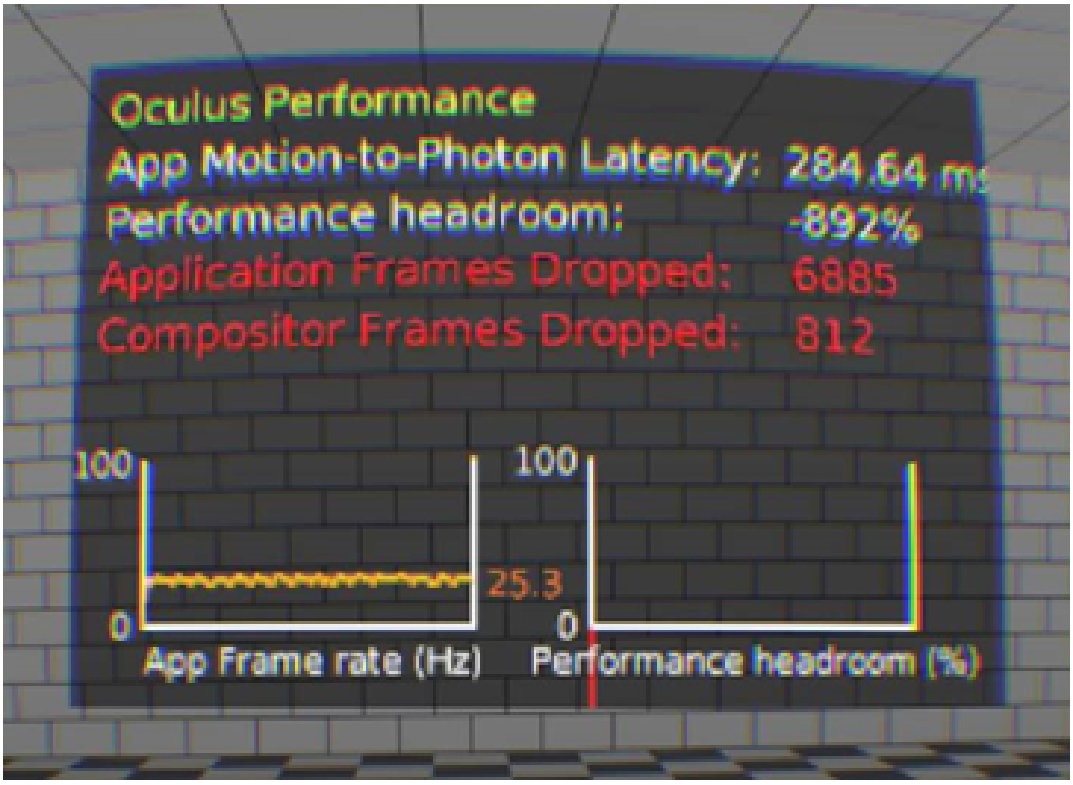
3. As a convenience, you may launch the application by selecting File > Launch App...



4. Launch the application binary, which is located here:

```
Oculus_SDK>\Samples\OculusRoomTiny_Advanced\ORT (Buffered Haptics)\Bin\Windows
\Win32\Debug\VS2013\z.Buffered Haptics.exe
```

5. In the headset, you should now see the Oculus Performance HUD:

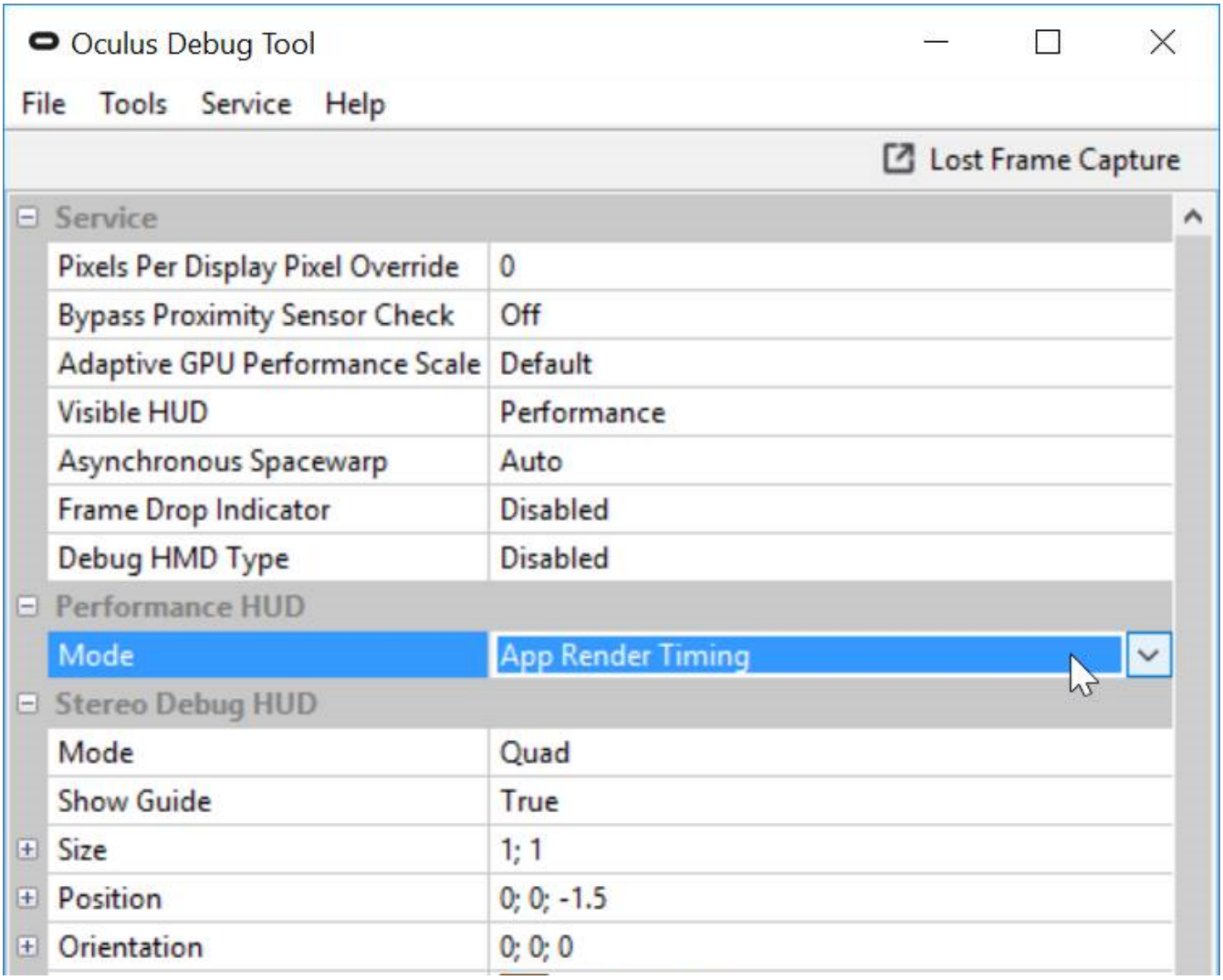


The HUD displays a performance summary showing four metrics:

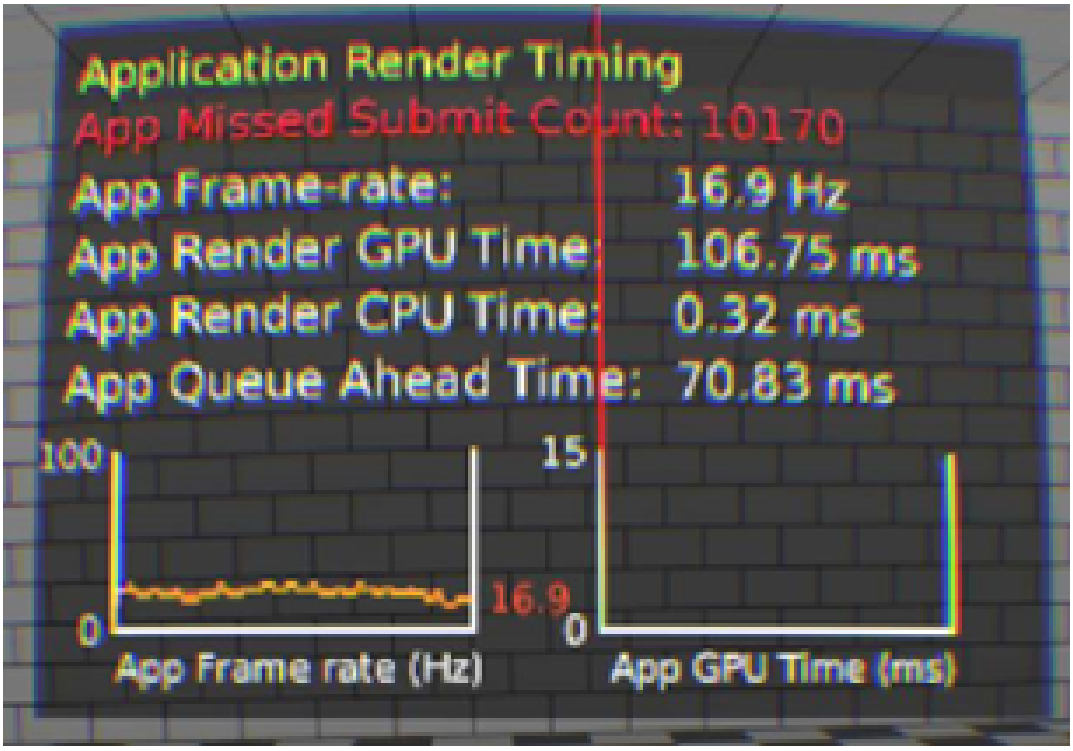
- App Motion-to-Photon Latency — This indicates how long it takes from the time the application was given control until the time the frame completed. Most of this is the application rendering time. The value shown above is about 284 ms, which is an extremely high value, and clearly represents a problem. Based on this, the HUD provides a recommendation for how much the application should change the level of CPU utilization, in this case about a 15x reduction in utilization.
- Performance Headroom — This is the percentage of the rendering time that is still available to be used, before the application will begin to exhibit performance problems. (It is calculated as follows: 1 - (FrameRenderTime / FrameVSyncToNextVsync). For example, if the frame took 5 ms to render, then the headroom will be: 1 - (5/11.11) = 0.54 = 54%) In this example, the Performance Headroom is actually a large negative number, which indicates that the application is dropping frames.
- Application Frames Dropped — This is the number of frames that have been dropped by the application. In this example, a very large number of frames are being dropped, and therefore the line has turned red.
- Compositor Frames Dropped — This is the number of frames that have been dropped by the Oculus Compositor.

Clearly this application is spending too much time in the App Motion-to-Photon Latency category. The problem is actually highly exaggerated in this example. Spending 18 ms per frame would be a more typical example of a performance problem.

6. Under Performance HUD, set Mode to App Render Timing:



The content of the Oculus Performance HUD now displays application render timing data. Here you can easily compare the computational loads on the CPU and GPU:

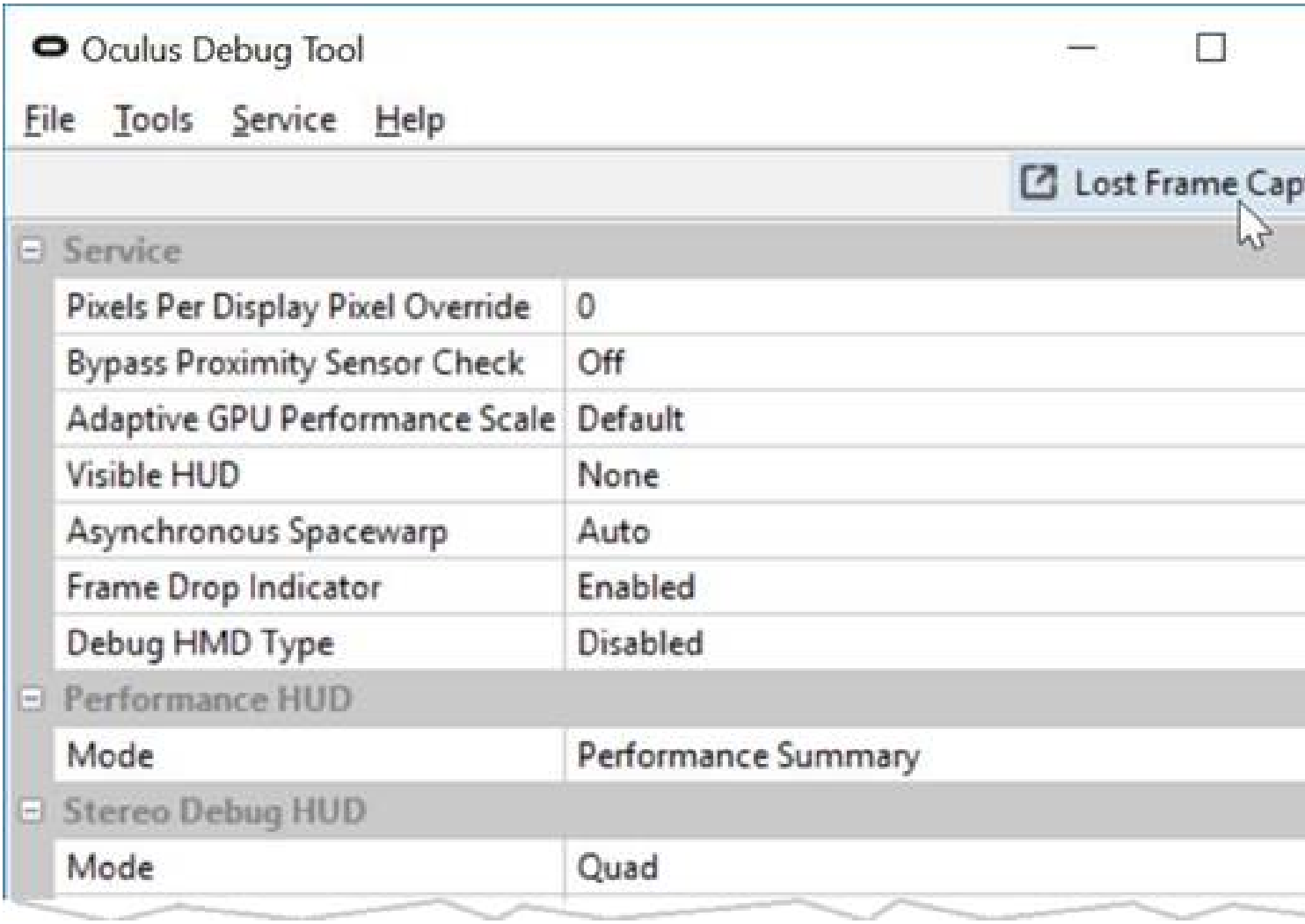


It is easy to see that the problem is with the GPU load. Again, this is a very exaggerated example, but it serves to illustrate the kind of insights you can gain by using these analysis tools.

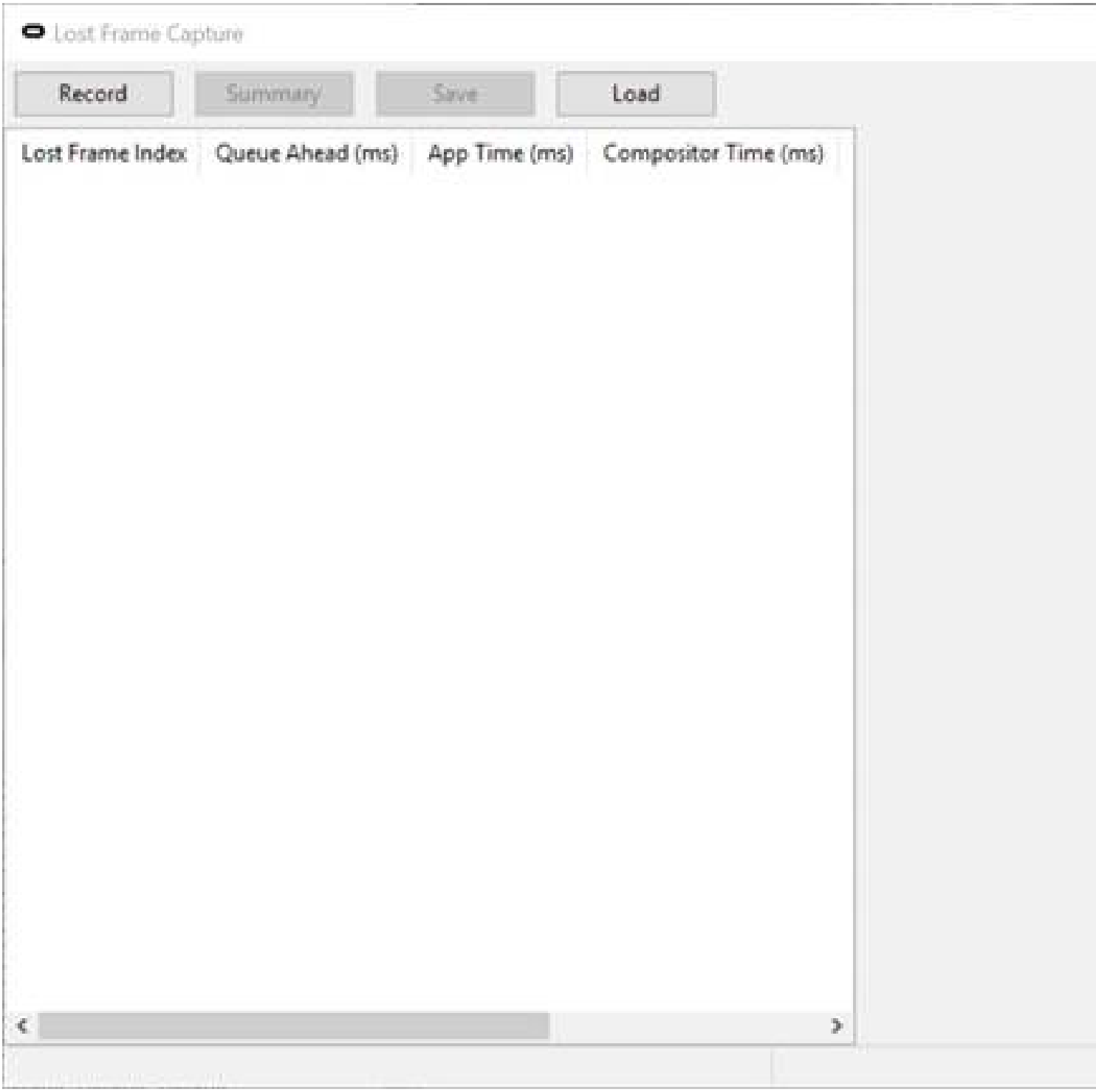
Using Lost Frame Capture Tool

Next, we will look at the frames that are dropped by this application, using the Lost Frame Capture tool.

1. Click the Lost Frame Capture button in the Oculus Debug Tool:



2. The Lost Frame Capture tool appears:



3. Make sure the application is displayed in the Oculus headset.

(The Lost Frame Capture tool is specific to Oculus, and knows to capture frames for the currently running Oculus application.)

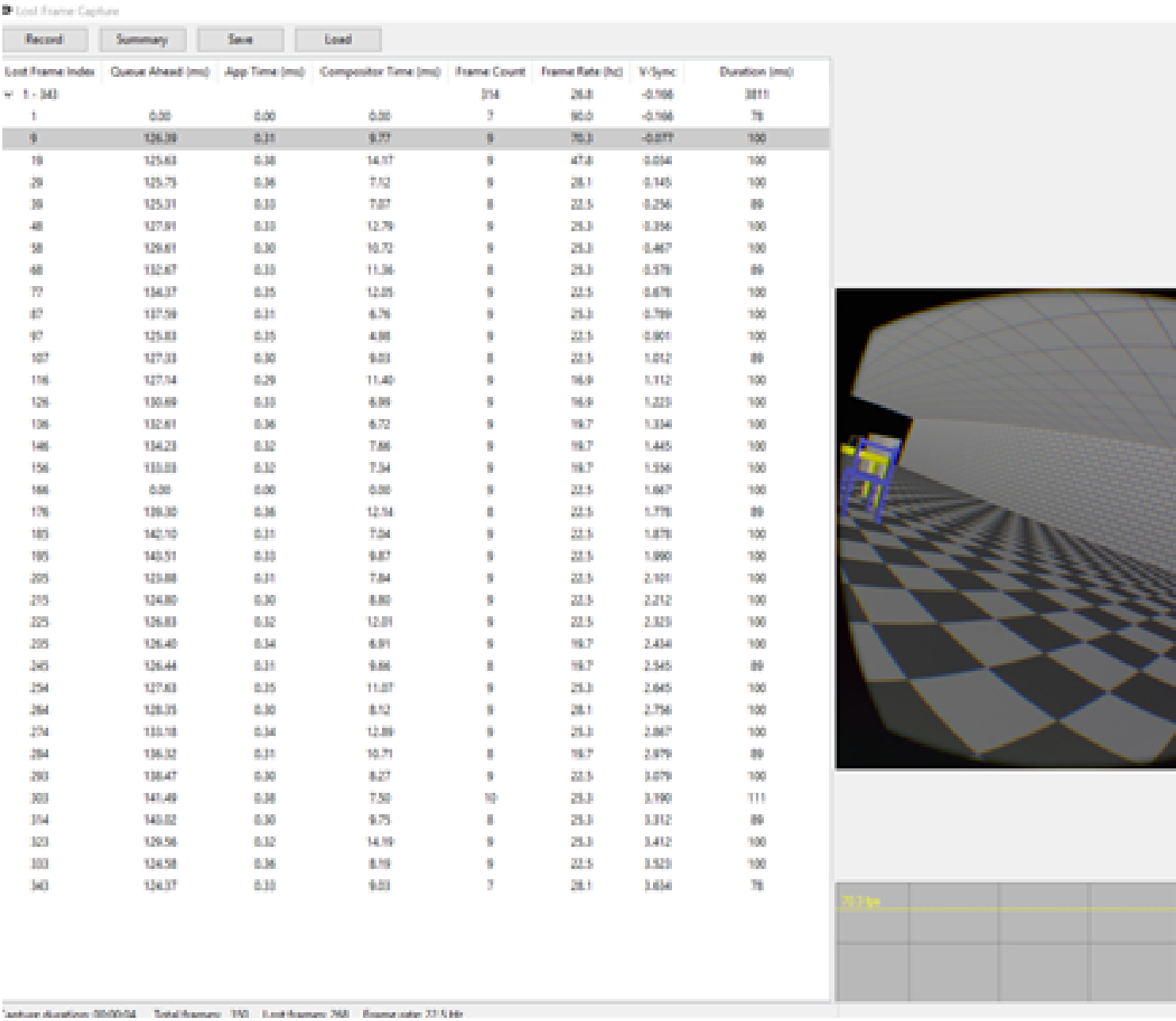
4. Click Record in the Lost Frame Capture tool.

5. Perform a few actions with the application, such as moving the headset in different directions.

6. Take off the headset, and click the Stop button in the Lost Frame Capture tool window.

7. Optionally, click Save to save the captured data to an Oculus Debug Archive (ODA) file. Name it appropriately, for example `app_1000.oda` (if the loop variable in the shader routine was set to 1000).

You can now see the lost frames, and even step through them by moving the cursor up and down the list of frames:



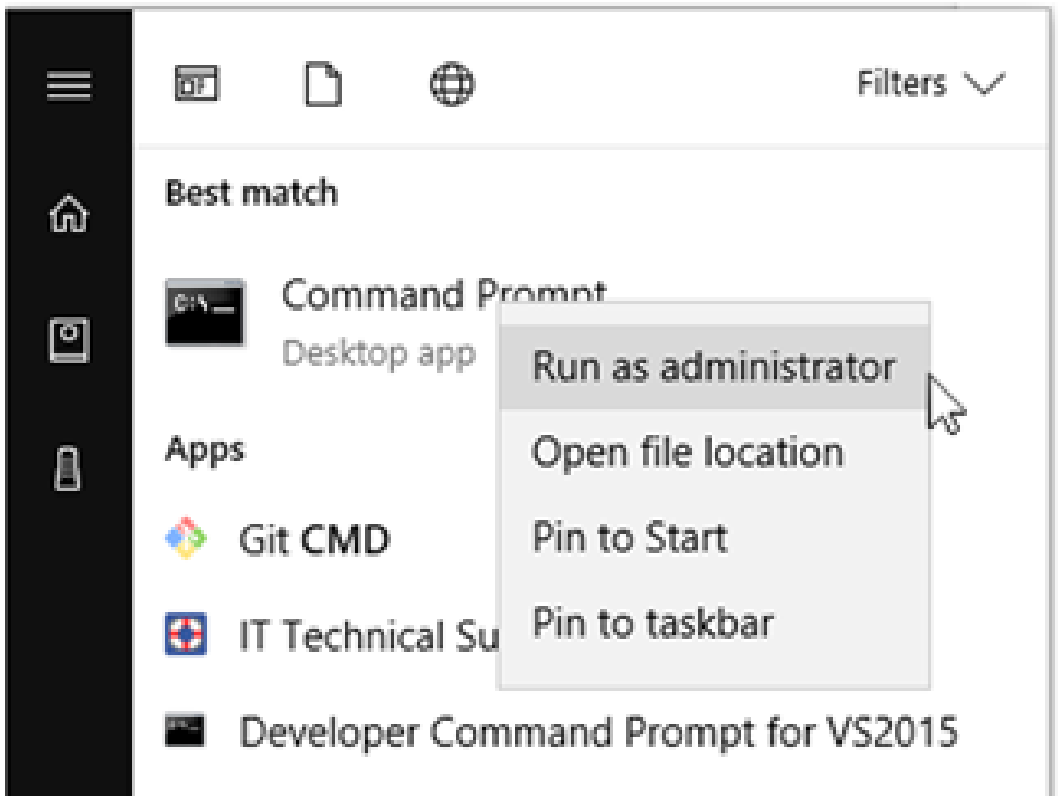
If you see that certain objects tend to come into view when frames are lost, you may need to optimize the rendering for those objects by reducing the number of polygons they contain, simplifying the shaders they use, simplifying their alpha blending routines, and so forth.

Using `ovrlog`

We are now going to capture low-level events using Event Tracing for Windows (ETW). This will make it possible to analyze the behavior of the application at a much finer level of detail. We will do this by running the `ovrlog` utility (which provides a convenient way to start and stop ETW sessions). The events are captured into an Event Trace Log (.etl) file. We will analyze the event stream using two tools: Windows Performance Analyzer

(WPA) and GPUView. You can think of WPA as providing a higher-level view of the event stream. Essentially WPA provides charts and graphs that *summarize* the performance-related characteristics of the event stream. GPUView, on the other hand, displays a highly-granular view of the events, themselves. So, with GPUView, you can drill down into the fine details of the interactions between the CPU and GPU workloads, and fine tune your application in order to optimize the timing and content of those workloads.

1. Start a Windows CMD window *with admin privileges*.



During the rest of this tutorial, you will run the `ovrlog_win10` script or the `ovrlog` script – depending on whether you are running Windows 10 or an earlier version of Windows, respectively. This script calls the xperf utility which starts an ETW trace session that captures the events that occur while the VR application is running.

You must run the ovrlog_win10 (or ovrlog) script with admin privileges. Elevated privileges are required in order to perform kernel-level event tracing, which is necessary in order to capture all the events that are relevant to Oculus applications.

The first time you run the `ovrlog_win10` or `ovrlog` script, it performs the following actions:

- Locates the xperf utility
- Installs the Oculus manifest of event definitions that provide insight regarding Oculus application performance
- Runs the xperf utility in order to start an ETW session
- Begins capturing events

The second time you run the `ovrlog_win10` or `ovrlog` script, it performs the following actions:

- Halts the ETW session
- Uninstalls the Oculus manifest
- Aggregates results into a trace file (<xperf_folder>\trace\merged.etl) that you can subsequently load into WPA or GPUView in order to analyze the flow of events that occurred

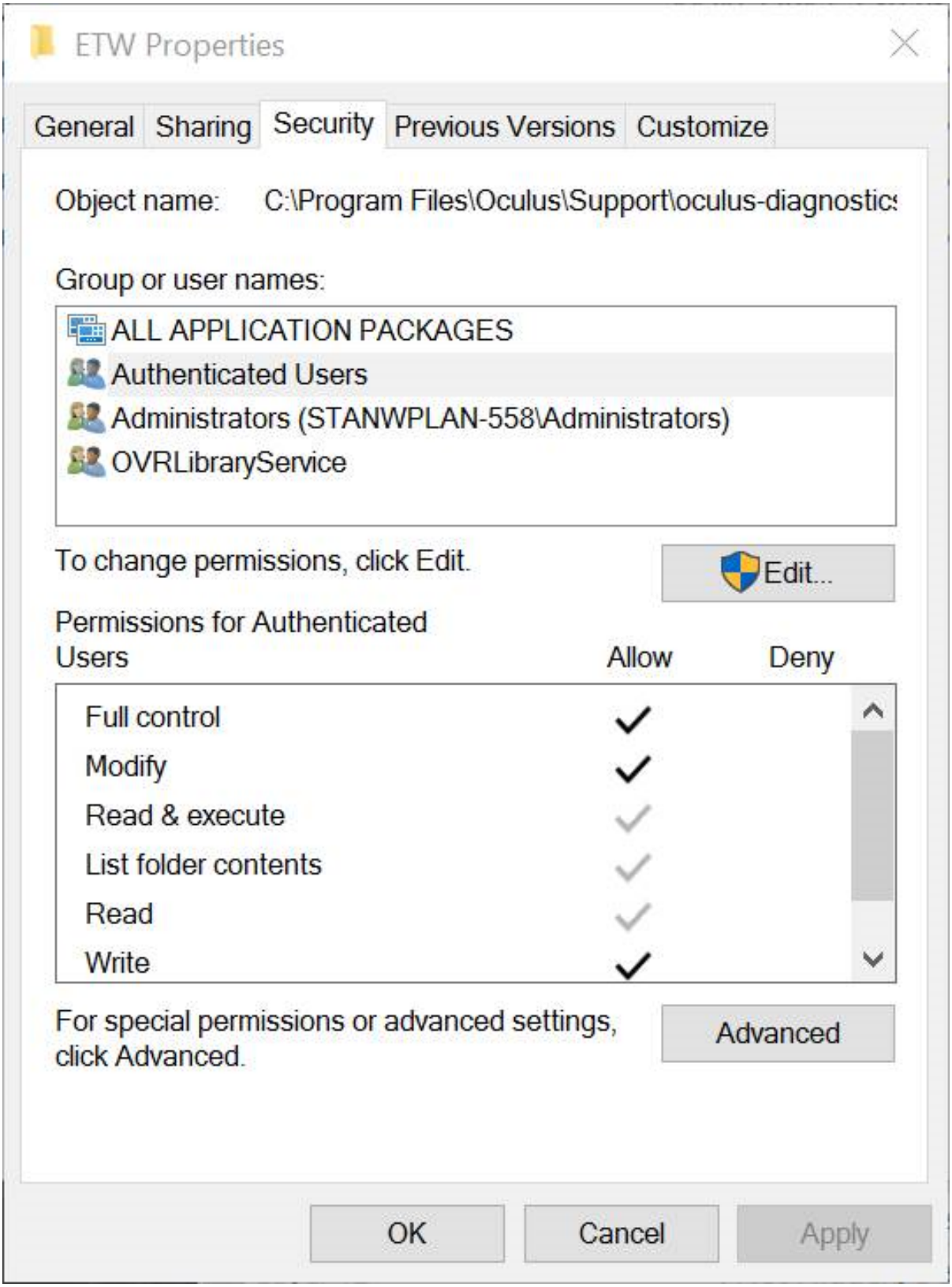
2. Make sure the Xperf folder provides permissions to Authenticated Users for:

- Full control
- Modify

The Xperf folder is located here:

```
%PROGRAMFILES%\Oculus\Support\oculus-diagnostics\ETW
```

So, the permissions for the Xperf folder should appear as follows:



3. Prepare to run through the following four steps cleanly, in reasonably quick succession.

After you complete these four steps, if you see error messages in the console output (which appear in red), then try deleting the `%PROGRAMFILES%\Oculus\Support\oculus-diagnostics\ETW\trace\` folder.

This is necessary so that the intermediate trace files don't conflict when you run `ovrlog_win10` or `ovrlog`. After you delete this folder, run through the following four steps again.

4. Run the Buffered Haptics application, and make sure it is currently being viewed in the headset. (You can run it directly within the Visual Studio debugger, if desired.)

5. Run the following script from a Command Prompt *with admin privileges*:

```
%PROGRAMFILES%\Oculus\Support\oculus-diagnostics\ETW\overlog_win10.cmd
```

or if you are using a Windows version prior to Windows 10:

```
%PROGRAMFILES%\Oculus\Support\oculus-diagnostics\ETW\overlog.cmd
```

6. Perform any desired action within the VR application, while continuing to view it within the headset.

7. Run `ovrlog_win10` or `ovrlog` again to stop capturing events.

Alternatively, you can run the `ovrlog_win10` or `ovrlog` command with the argument "stop" to stop tracing. Running the command again with no arguments from the same command prompt will stop tracing, and is the way most people typically use the command. However, this method is more failure-prone due to its reliance on environment variable settings.

The following file should be generated:

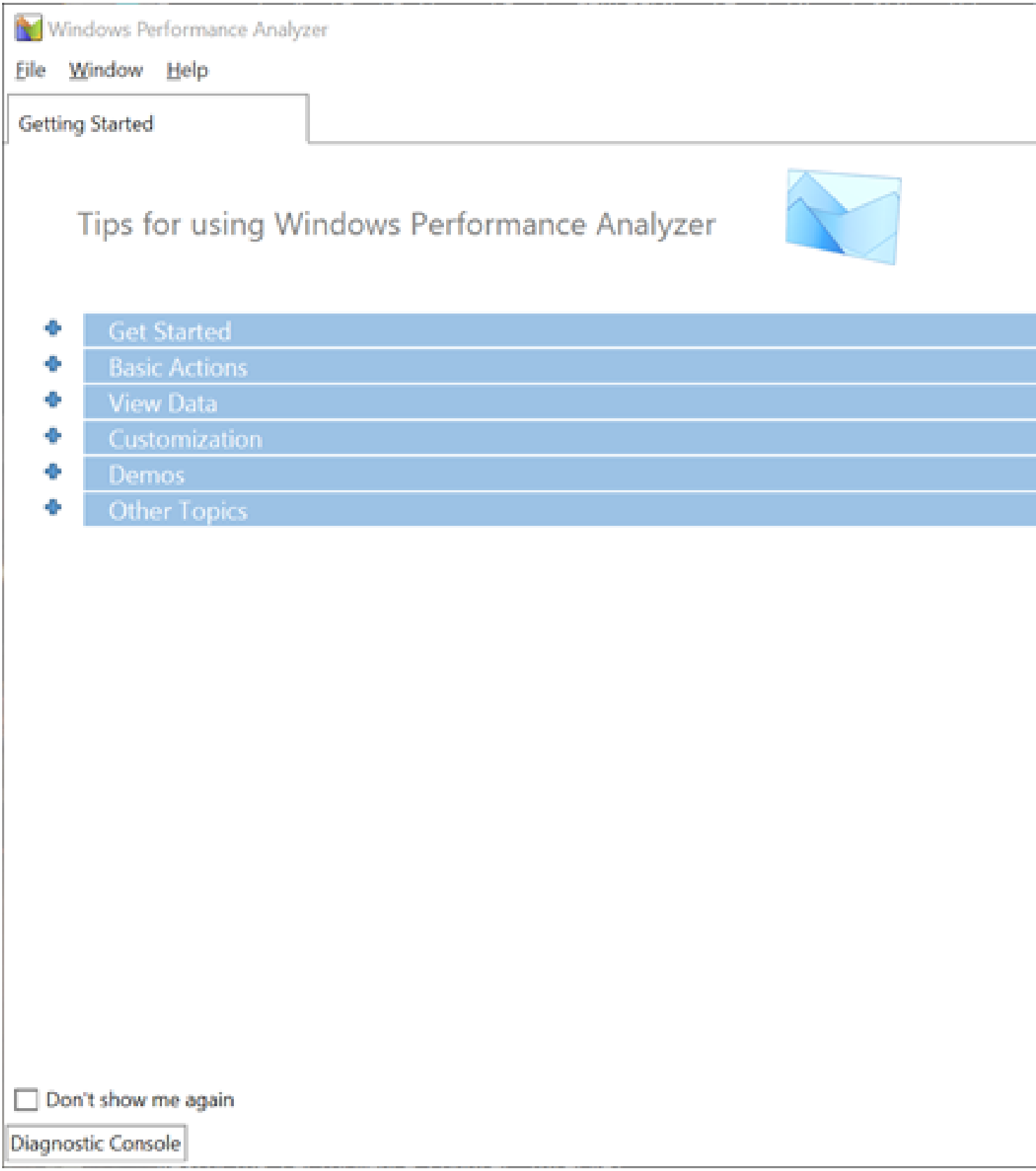
```
%PROGRAMFILES%\Oculus\Support\oculus-diagnostics\ETW\trace\merged.etl
```

Using Windows Performance Analyzer (WPA)

1. Start Windows Performance Analyzer (WPA), which is located here:

```
C:\Program Files (x86)\Windows Kits\10\Windows Performance Toolkit
```

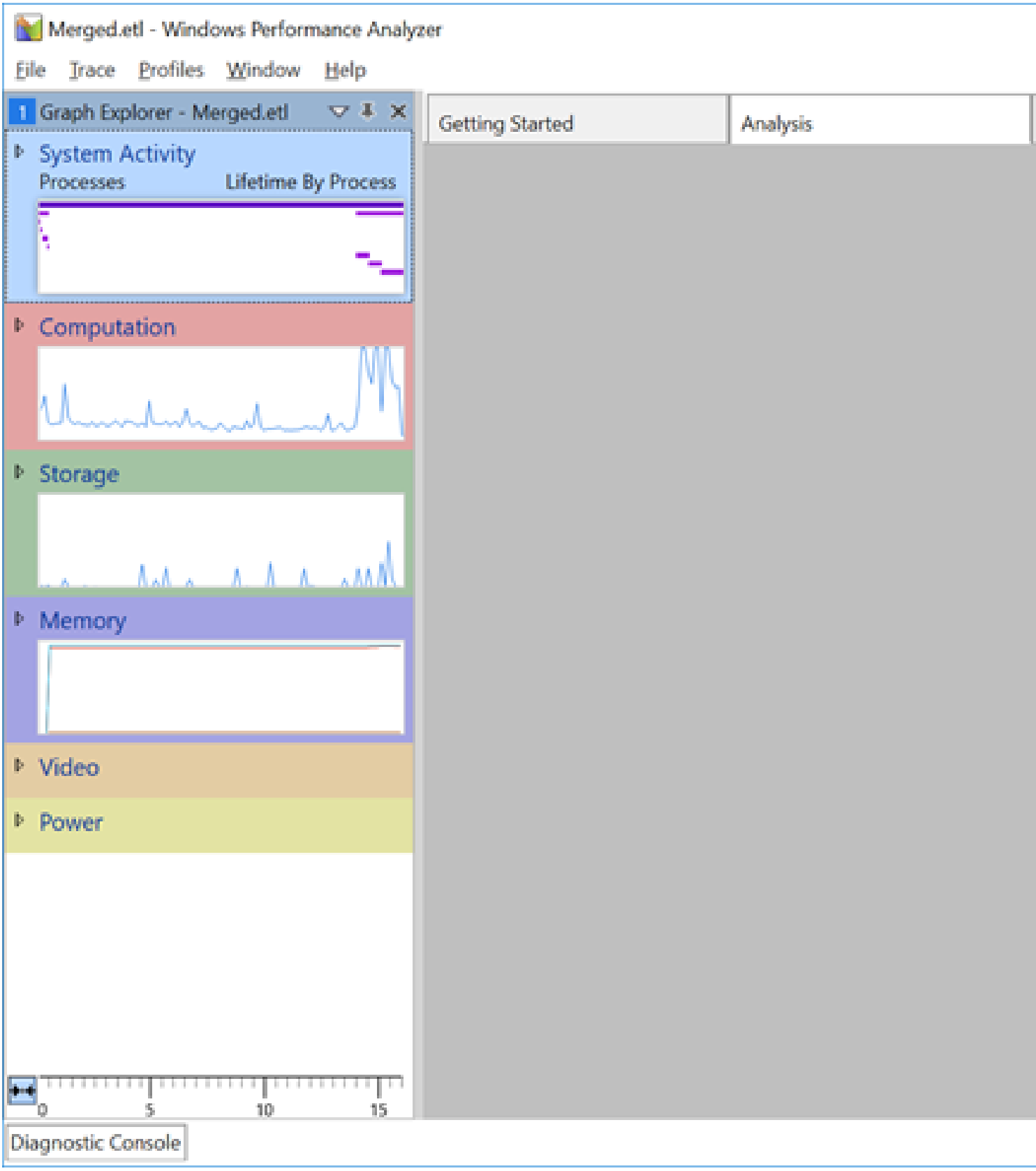
2. The WPA window appears:



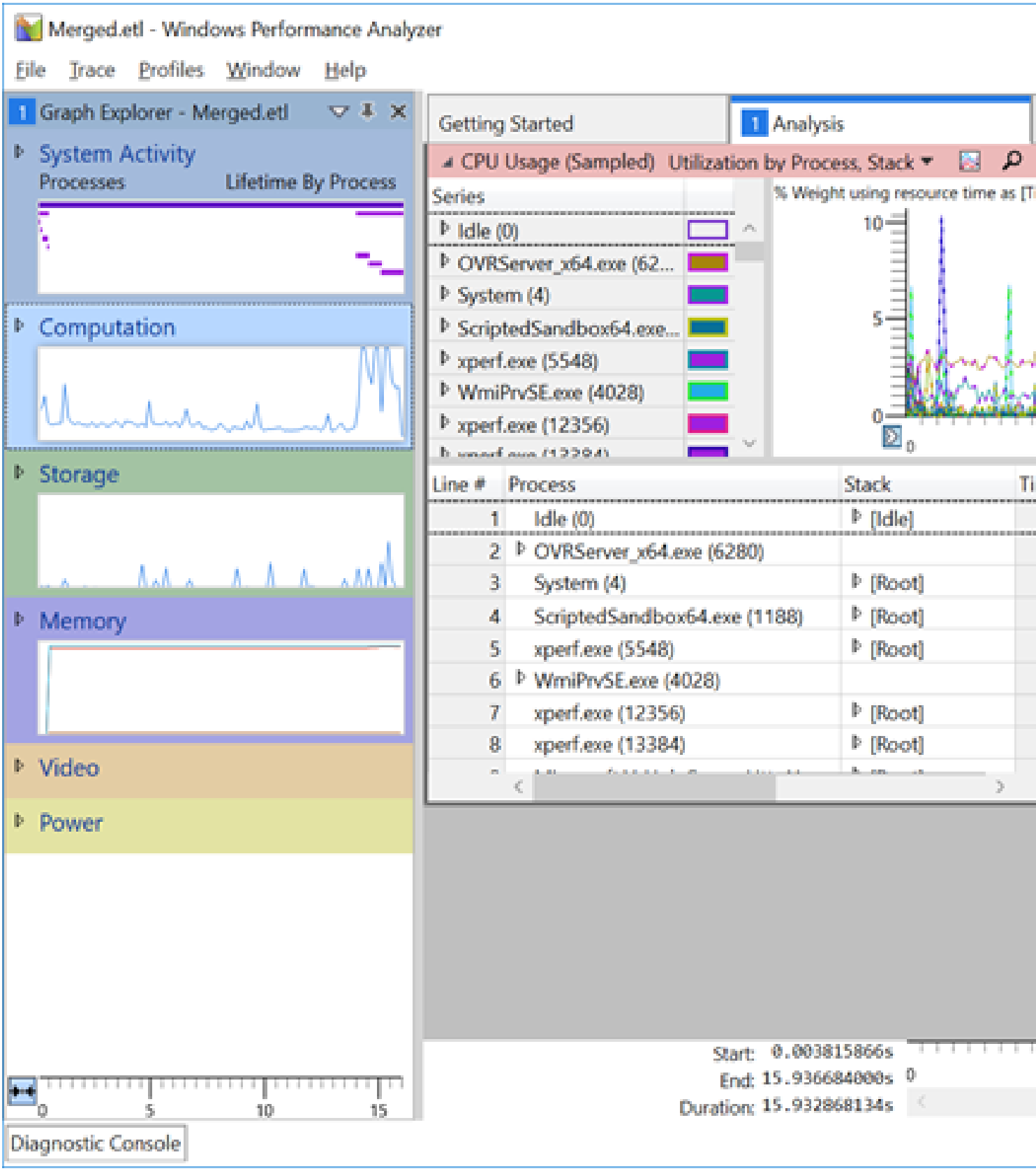
3. Select File > Open, and open the `merged.etl` file that was produced above:

`%PROGRAMFILES%\Oculus\Support\oculus-diagnostics\ETW\trace\merged.etl`

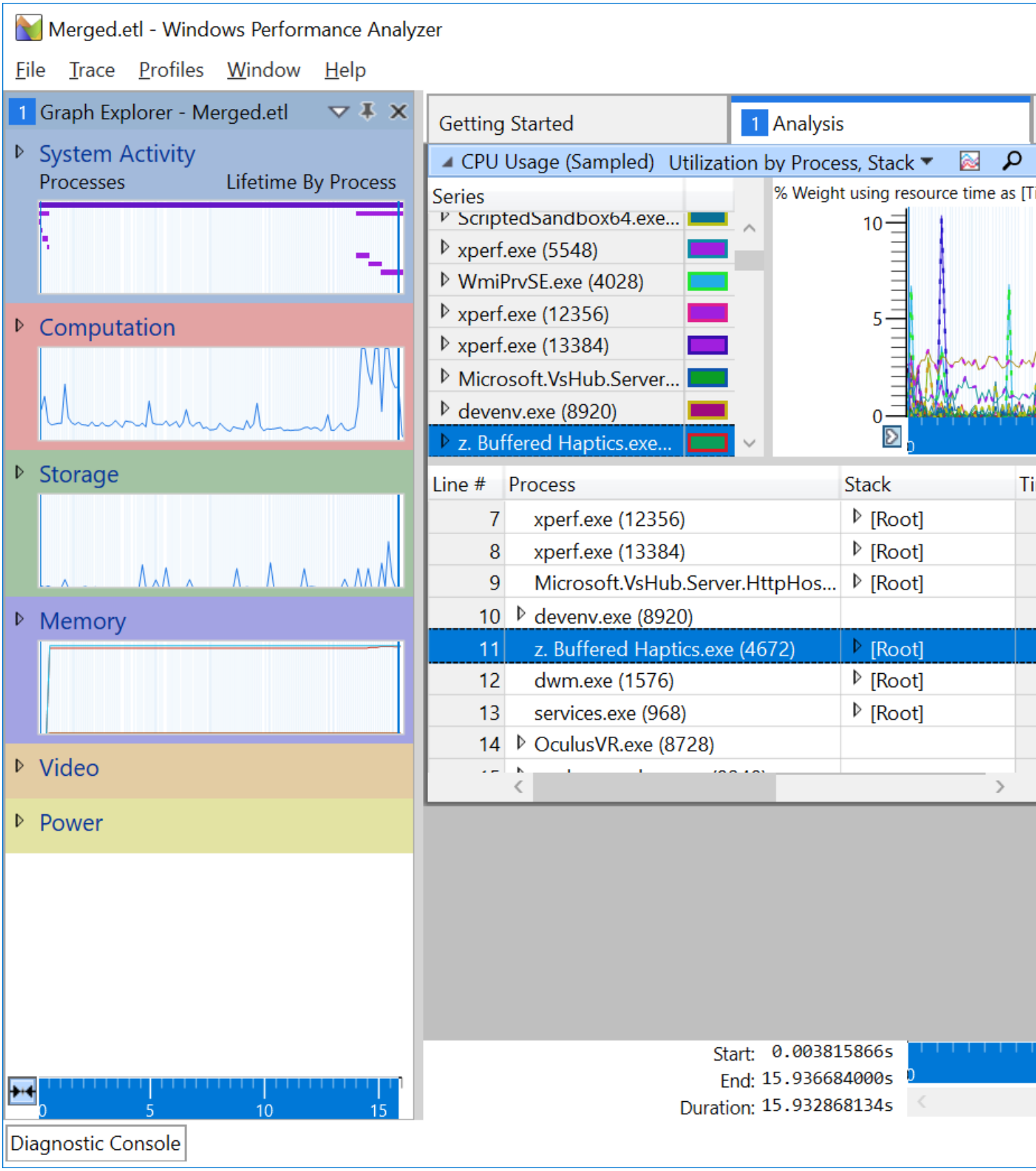
4. You should see the following:



5. To check the CPU usage, double click on the small Computation window in the left panel. The full CPU usage graph is displayed in the work area:

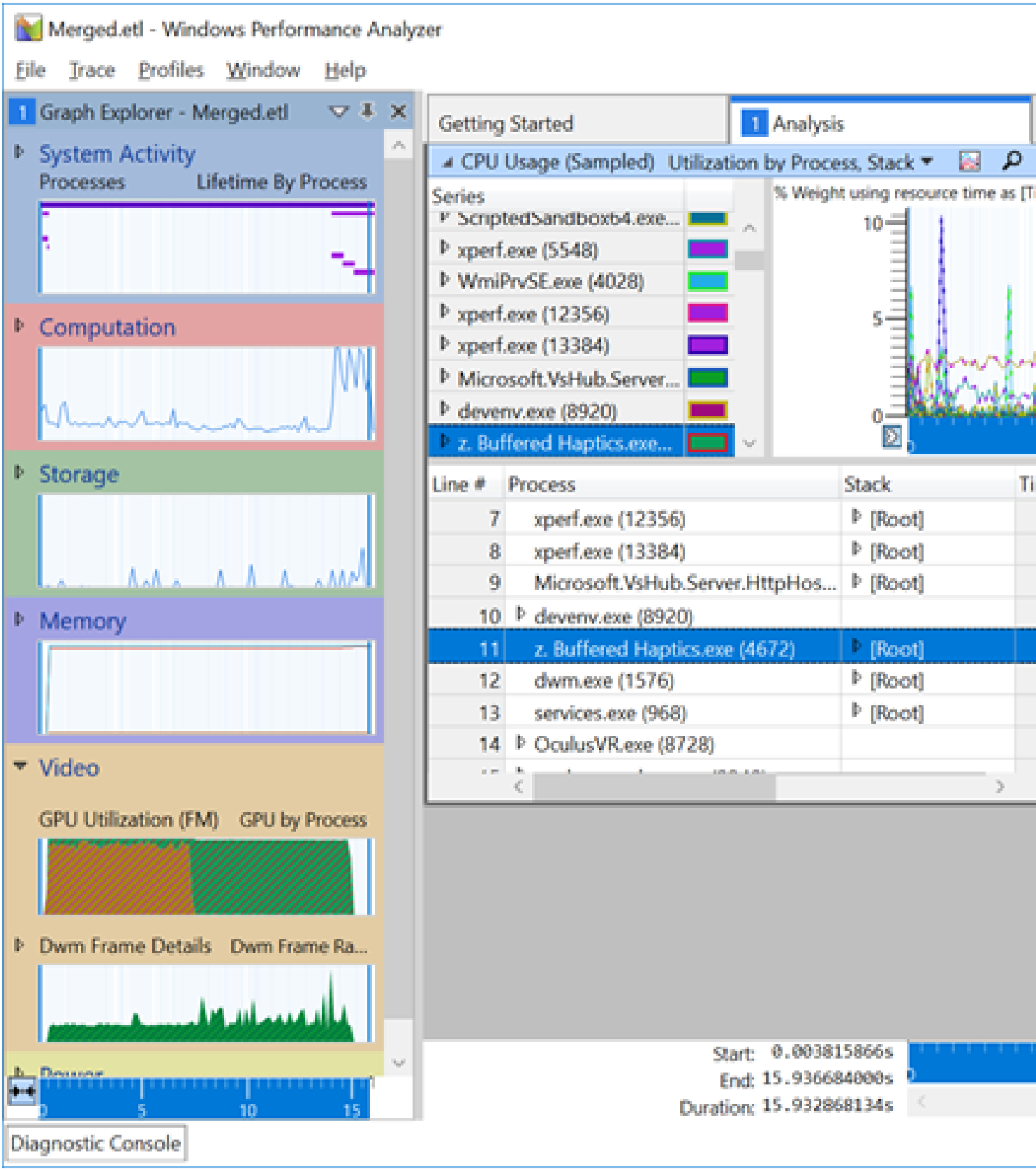


6. Scroll down to see the z.Buffered.Haptics.exe process. You can see it is using a very small percentage of the CPU resources:



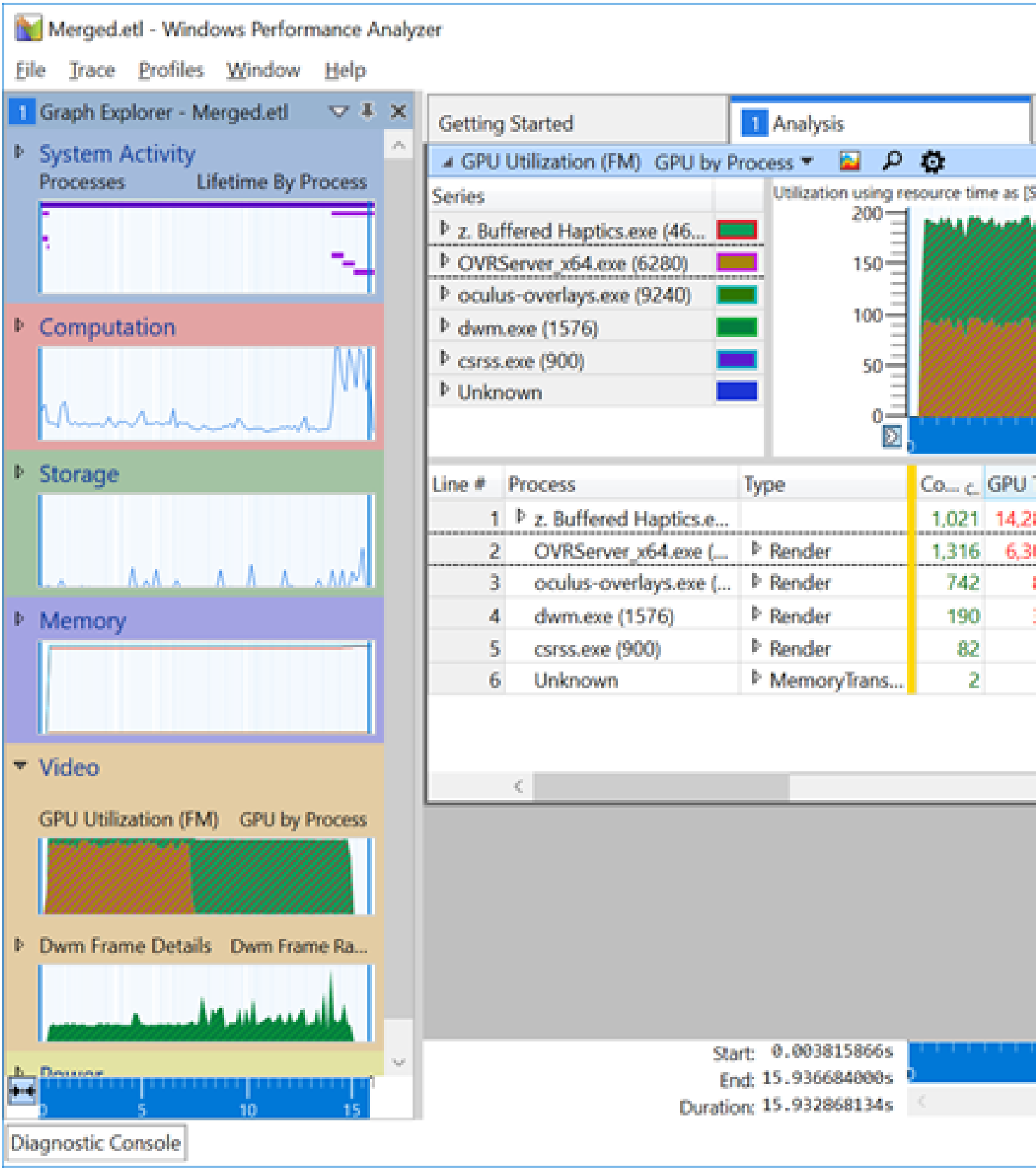
It is clear from the above that the performance problem is not due to over utilization of the CPU resource in this application.

7. To view the GPU usage, expand the Video tab:



It is immediately clear by looking at the small Video window that the GPU is overloaded. (Actually, this example is rather extreme. A typical GPU usage issue would be less pronounced than what we are seeing here.)

8. Remove the Computation graph from the work area, and double click on the Video window to display the GPU usage graph in the work area:



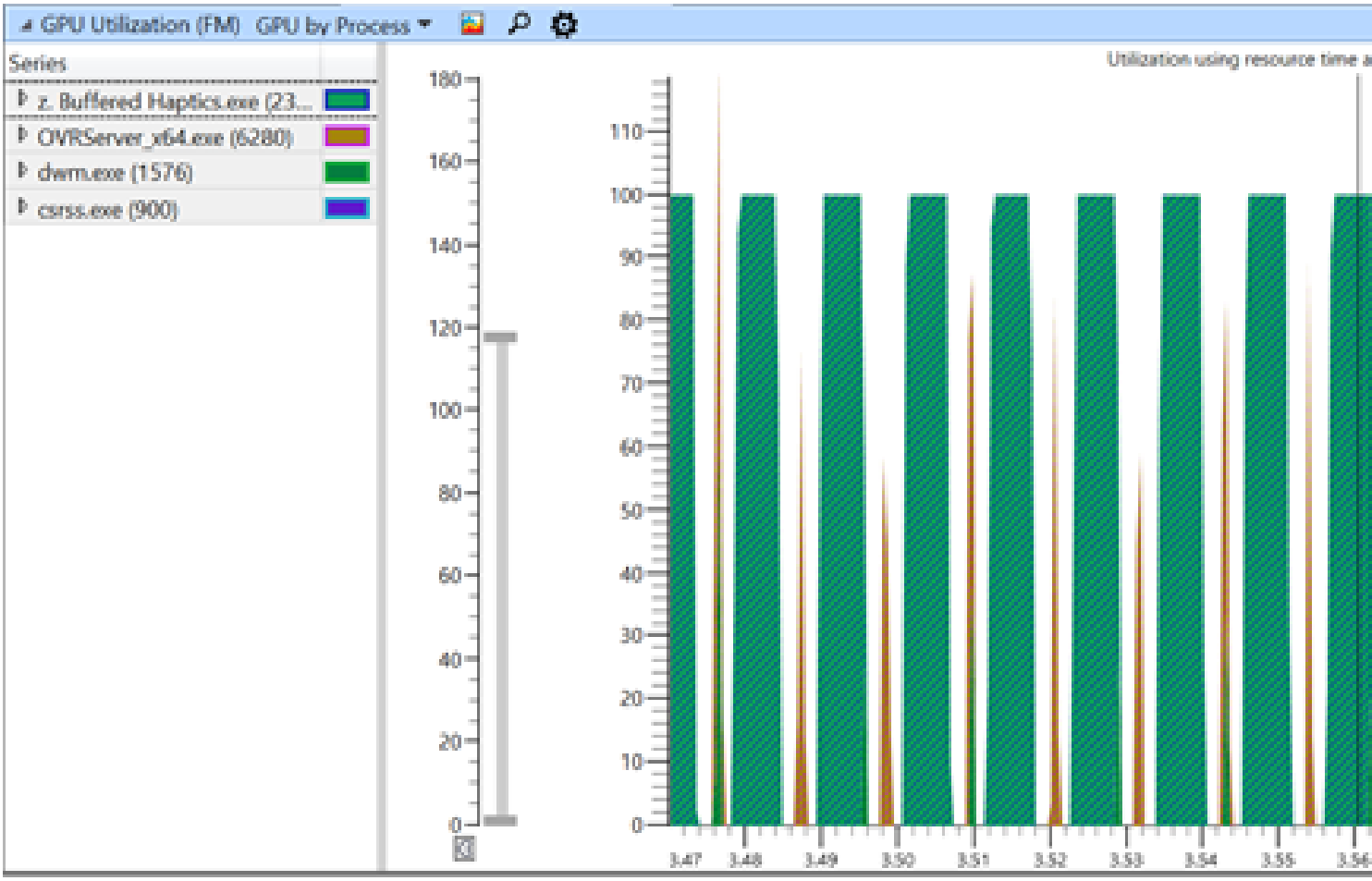
In this example, the application was actively running at the time `ovrlog_win10` or `ovrlog` was run the first time. About half way through the event capture period, the headset was set down. At that point, the VR application was frozen, re-rendering a single frame. The Compositor is not called during that time period, so the GPU usage of OVRServer_v64.exe goes to zero. Subsequently, after a few seconds, `ovrlog_win10` or `ovrlog` was run for the second time, and the event capture period ended.

9. Select a small portion of the graph where both the application and the Compositor are executing, by dragging the mouse horizontally over the area. Then, use Ctrl-ScrollWheel (on the mouse) to zoom in so that you can see 100ths of a second.

Note: This time scale is close to the frame rate, which is 90 frames per second.

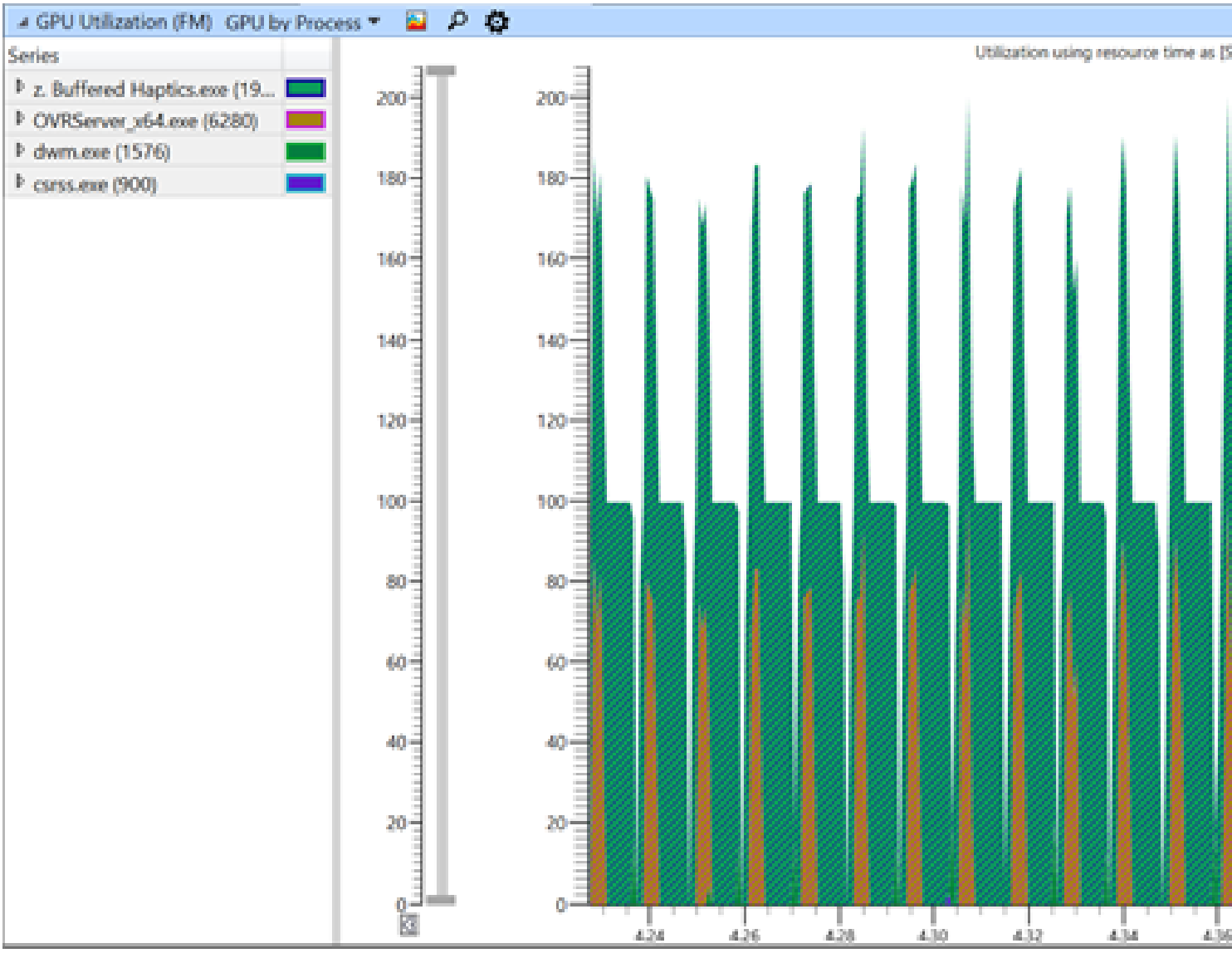
In the loop that we created within the shader routine, you can experiment by setting the loop variable so that it loops through 100, 500, 1000, or more cycles per pixel. The exact behavior that the application will exhibit -- given different values for the loop variable -- depends on the characteristics of the computer that you are using, and how heavily loaded it is during the event capture period.

When the loop variable is set to a relatively low value, and you zoom in to 100ths of a second, you should see a result similar to the following:



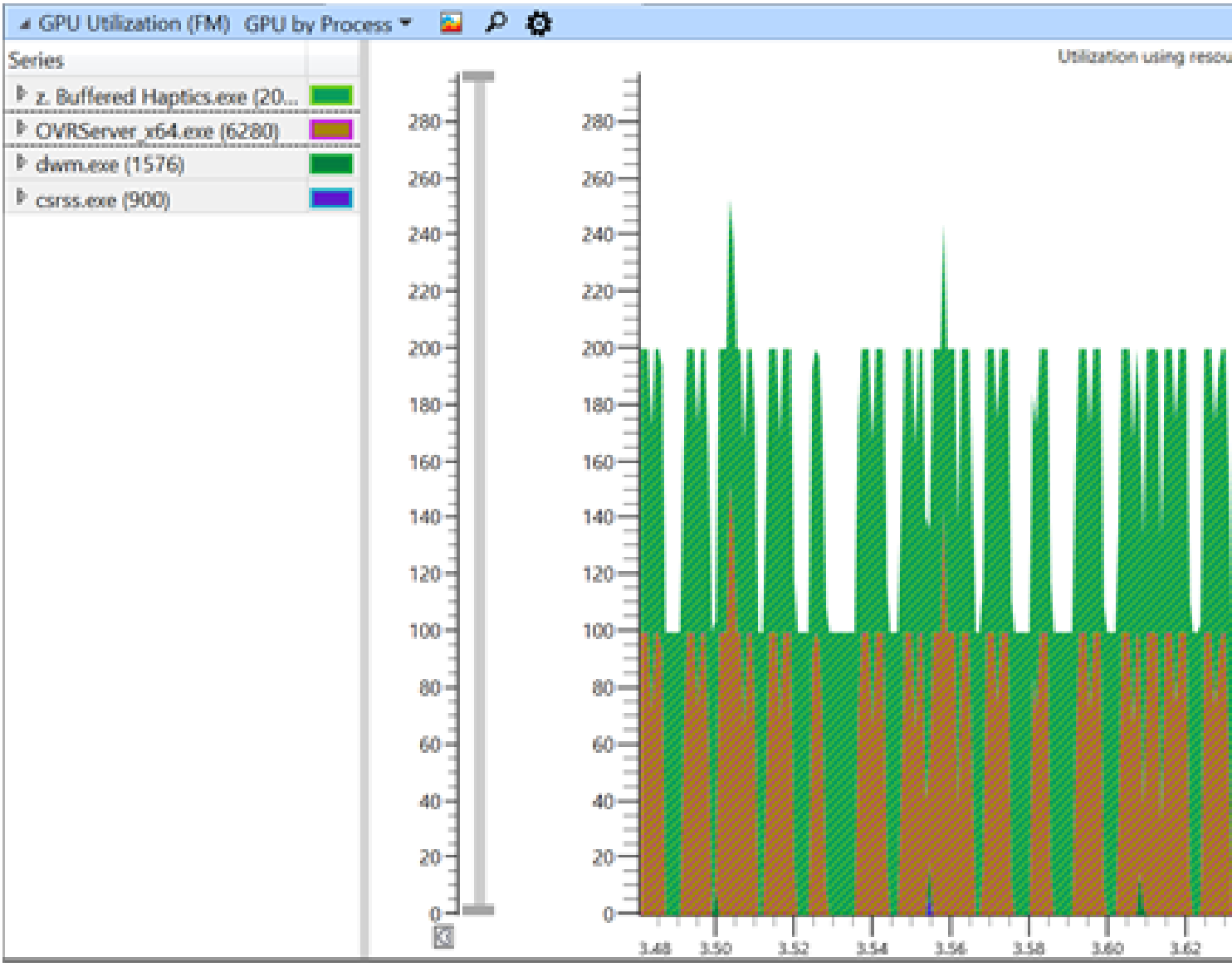
Here you can see clearly see the frame cycles. The application (shown in green) consumes a fairly large amount of the GPU resource during each frame, and the Compositor (shown in light brown) consumes a smaller amount of the GPU resource for the same frame.

When the loop in the shader routine is run more times per pixel, the situation becomes tighter:



Here there is very little time, if any, between the time the application finishes processing the frame, and the time that the Compositor outputs the frame. The Compositor is also using GPU processing *at the same time* as the application (as you can see by the fact that the two usage graphs are stacked). In this example, the shader routine is at or above the limit of complexity that is acceptable for a good user experience.

When the loop variable in the shader routine is increased even further, the situation becomes significantly worse:

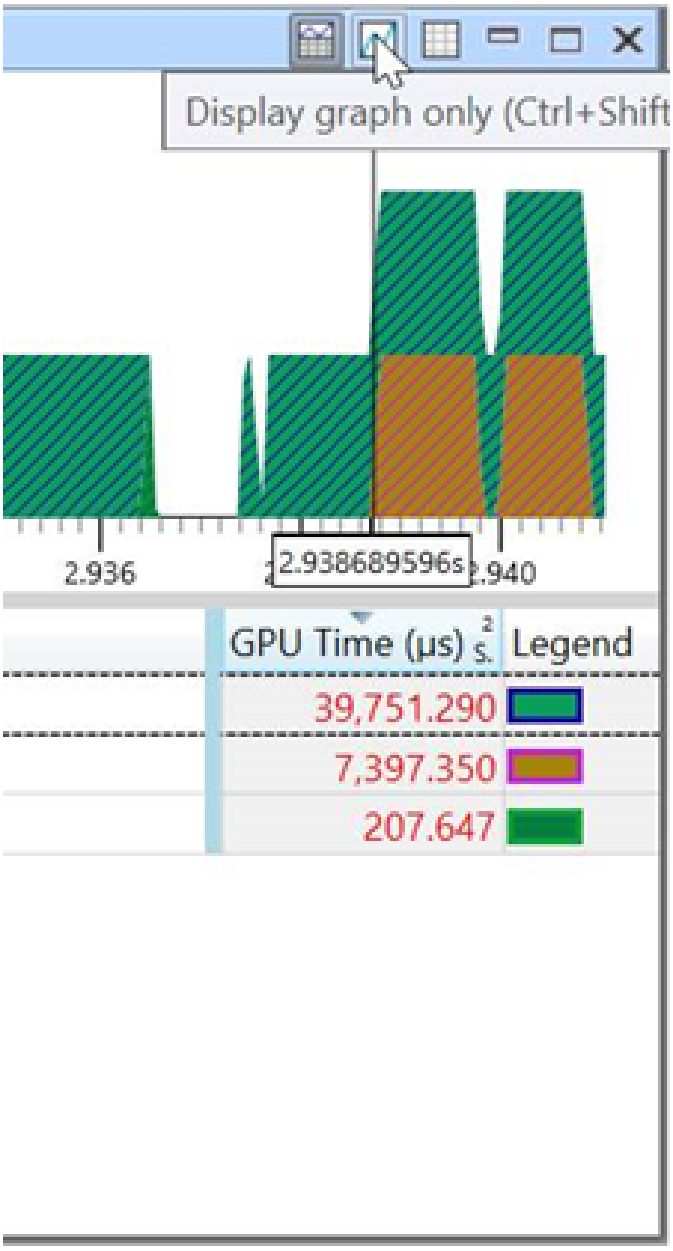


Here is it no longer possible to even see the frame cycles clearly.

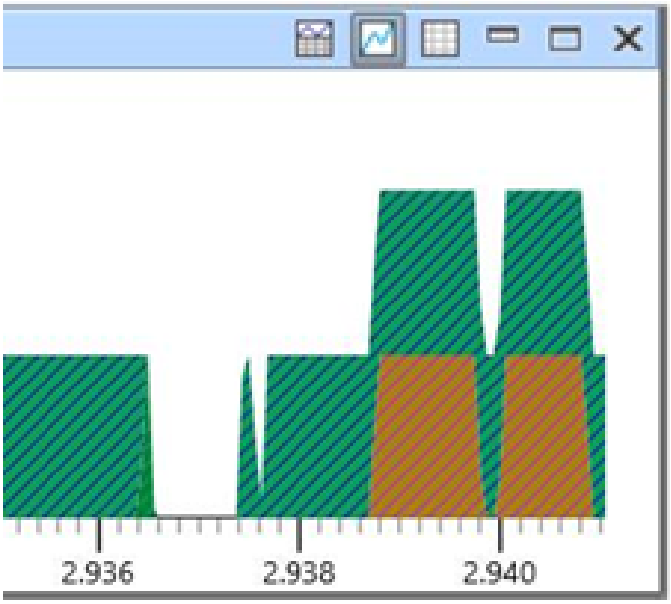
Essentially, the GPU never finishes what it is doing. You can see that z.Buffered Haptics.exe is the process that is heavily utilizing the GPU, by highlighting that process, and viewing the corresponding activity in the graph. None of the other processes are using the GPU very much at all.

We will now look a little closer at the trace data within WPA.

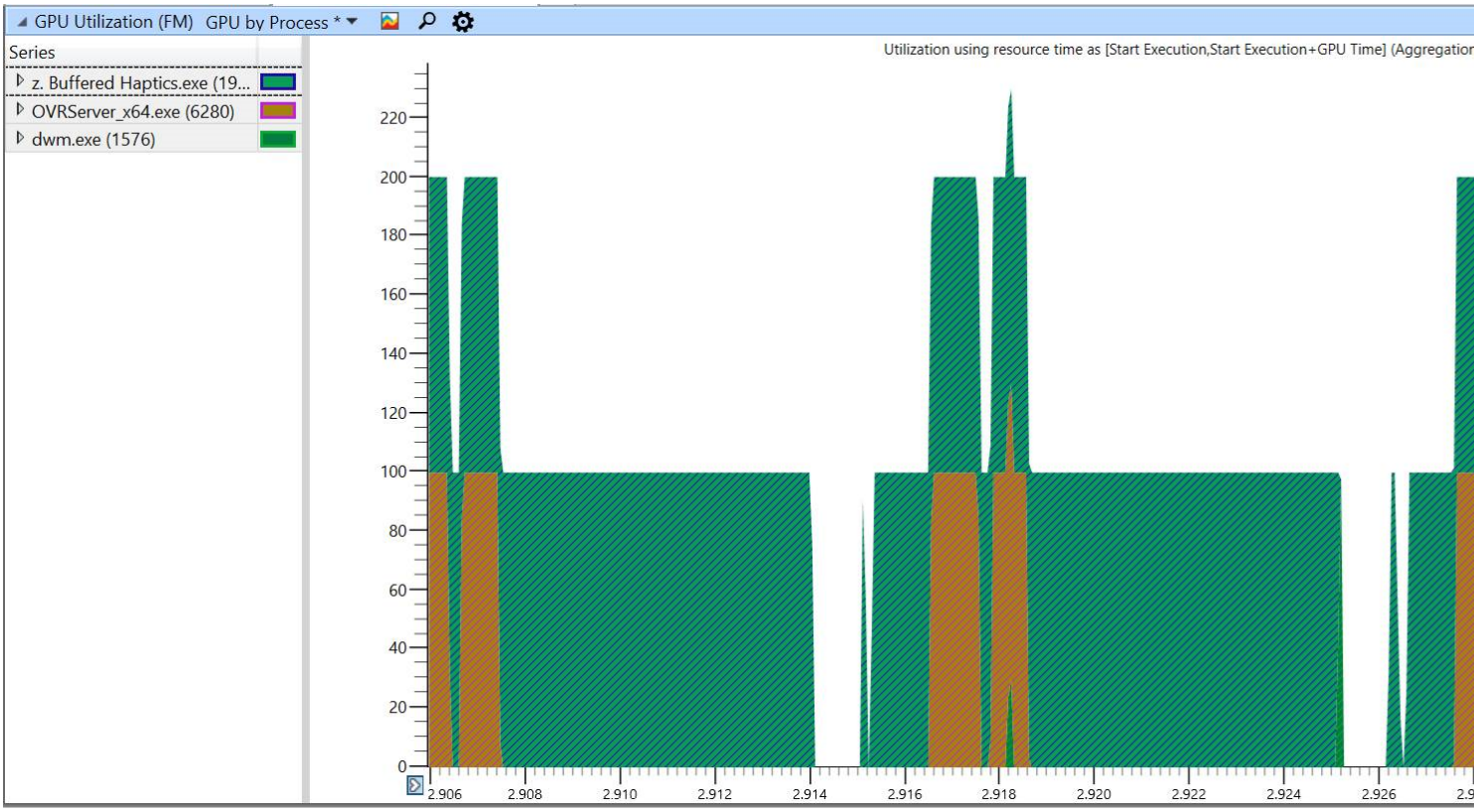
10. Zoom in further, and click on the icon to display the graph only:



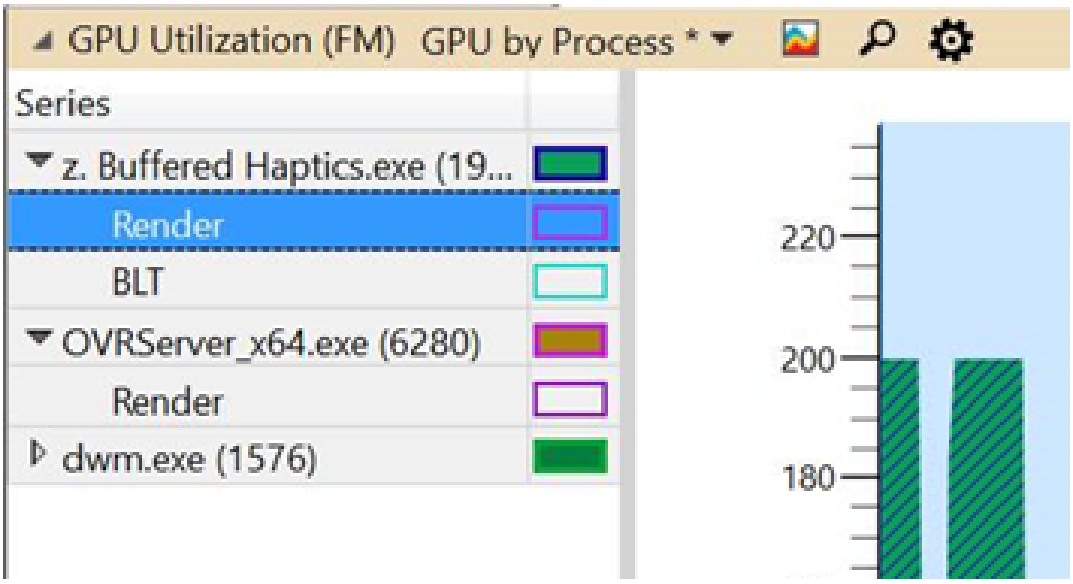
The table disappears and only the graph is visible::



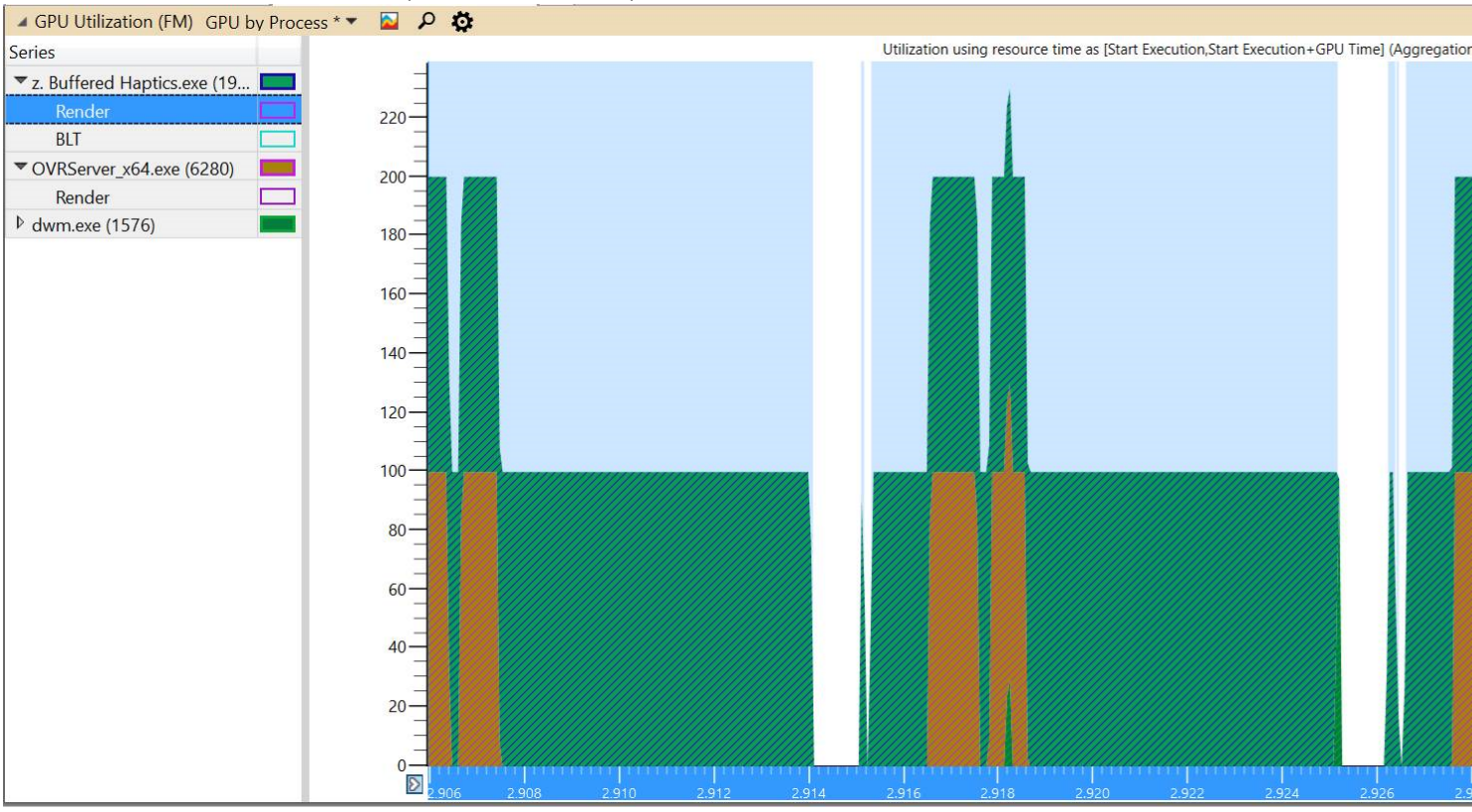
11. Drag down the bottom of the chart, in order to enlarge the view:



In the Series box on the left, open the hierarchies under z.Buffered.Haptics.exe and OVRServer_x64.exe, and select Render under z.Buffered.Haptics.exe:

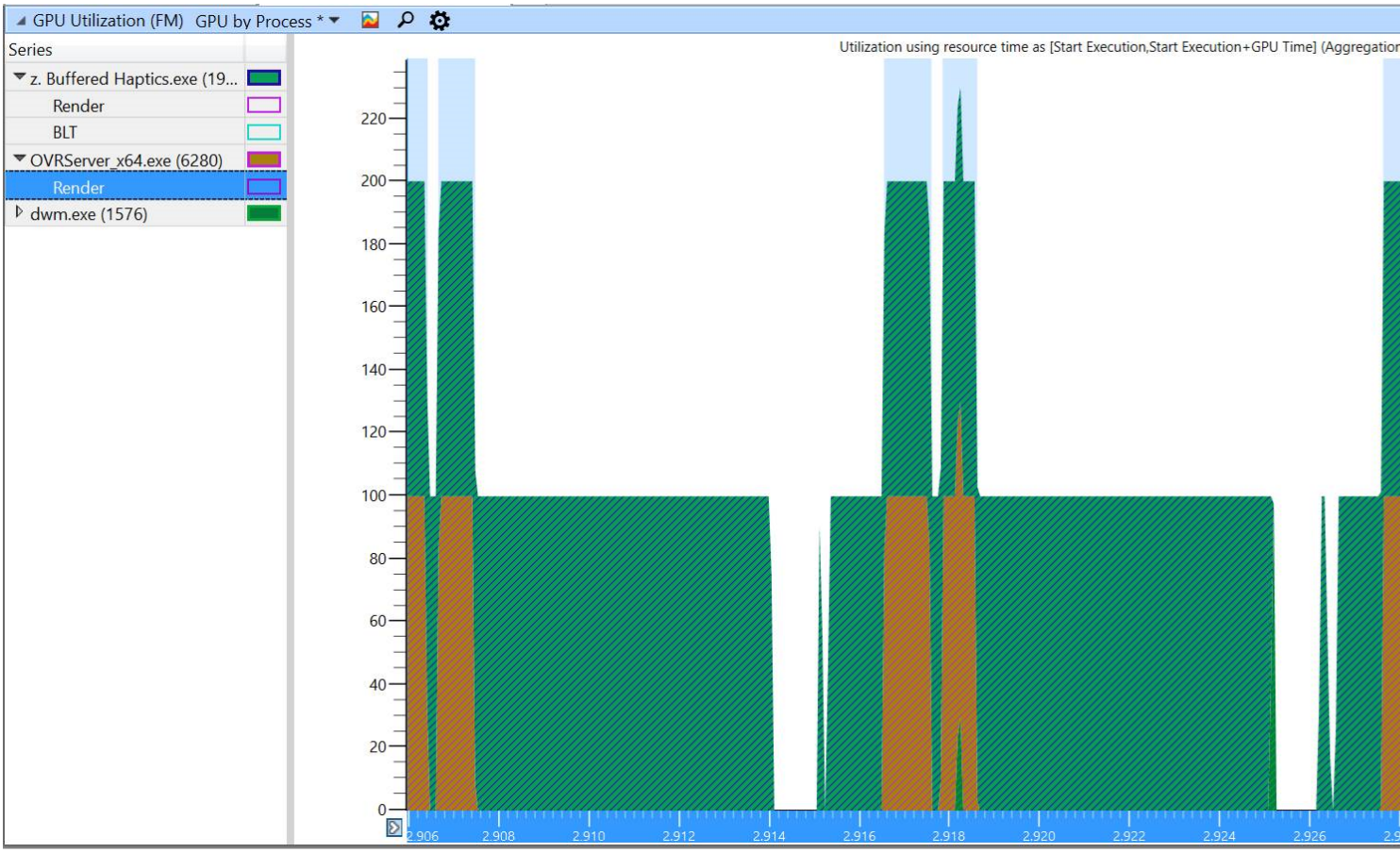


The space above the z.Buffered Haptics.exe process that is involved in rendering is filled in. As you can see, by far most of the time is being used for rendering within z.Buffered Haptics.exe. (This is, of course, the result we expect, given the loop the we inserted into the shader routine.)

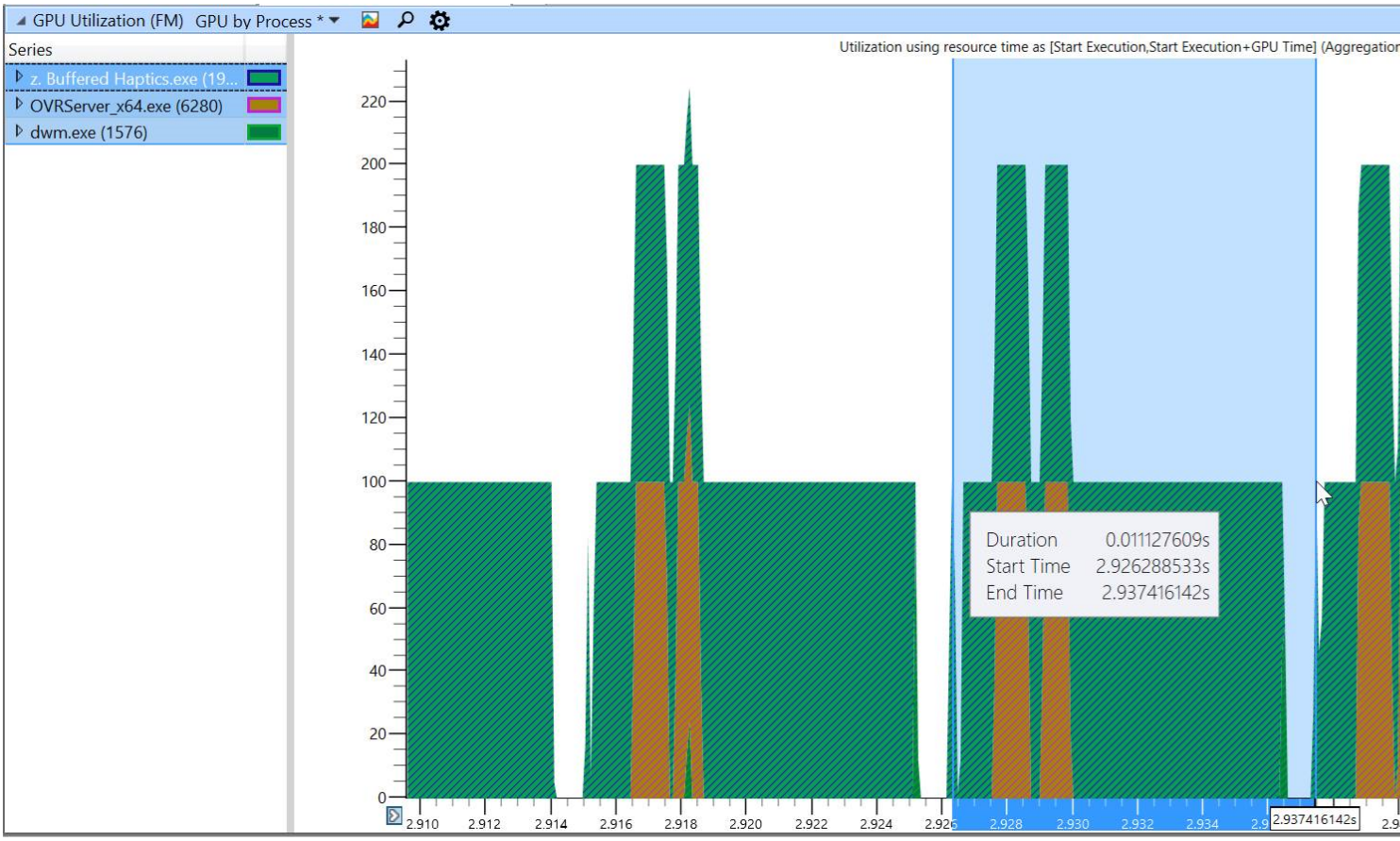


WPA does not provide a detailed view of what the application is actually doing with the GPU, but we know there are two major phases to the GPU activity: a rendering phase and a BLT phase (a buffer copy at the end of the cycle). If you select BLT, you will find that very little time is spent on that phase.

12. In the Series box on the left, select Render under OVRServer_x64.exe. You can see that all of the GPU time used by the server is applied toward rendering:



If you hold down the mouse and drag from one point in the cycle to the identical point in the next cycle, you can see that the duration is about 11.1 microseconds, which is a 90th of a second. This is the frame rate for the Rift. So, we can see in this example that frame rate is being met.



It is not possible to actually display the VSync events in WPA. But, you can usually identify the frame cycles, as in the example above.

Using GPUView

In the following steps, we will view the ETW trace file(s) within GPUView. WPA doesn't show the specific queuing of packets (indivisible collections of computational work) on each hardware resource. GPUView provides that level of analysis.

1. Start GPUView:

```
c:\Program Files (x86)\Windows Kits\10\Windows Performance Toolkit\gpuview
\GPUView.exe
```

2. When GPUView launches, it first displays a File Manager dialog. Locate the merged.etl file that you wish to analyze and load it into GPUVIew:

```
%PROGRAMFILES%\Oculus\Support\oculus-diagnostics\ETW\trace\merged.etl
```

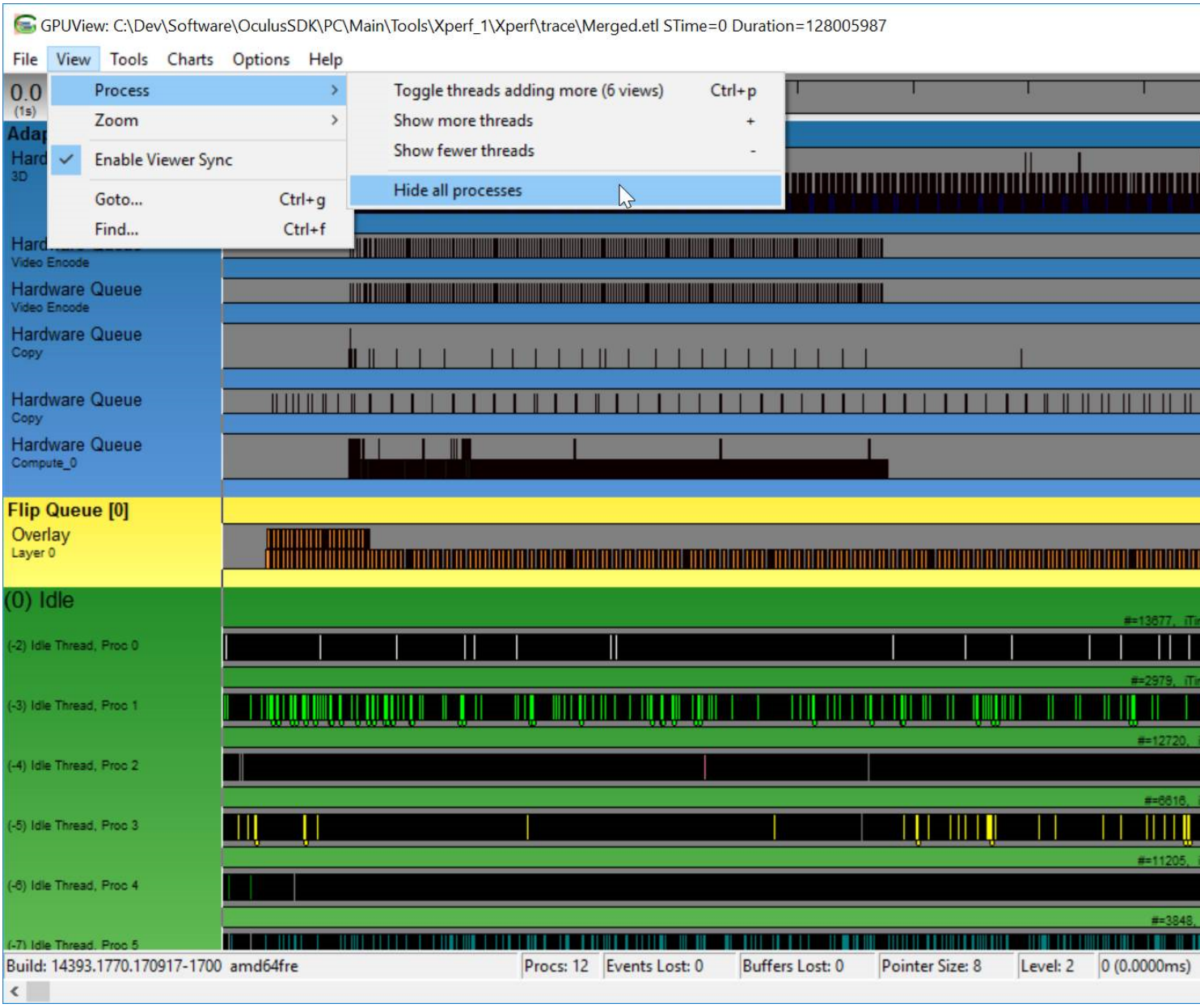
The GPUView window appears and presents a high-level view of the ETW trace that was captured in the merged.etl file during the time that the application was running:



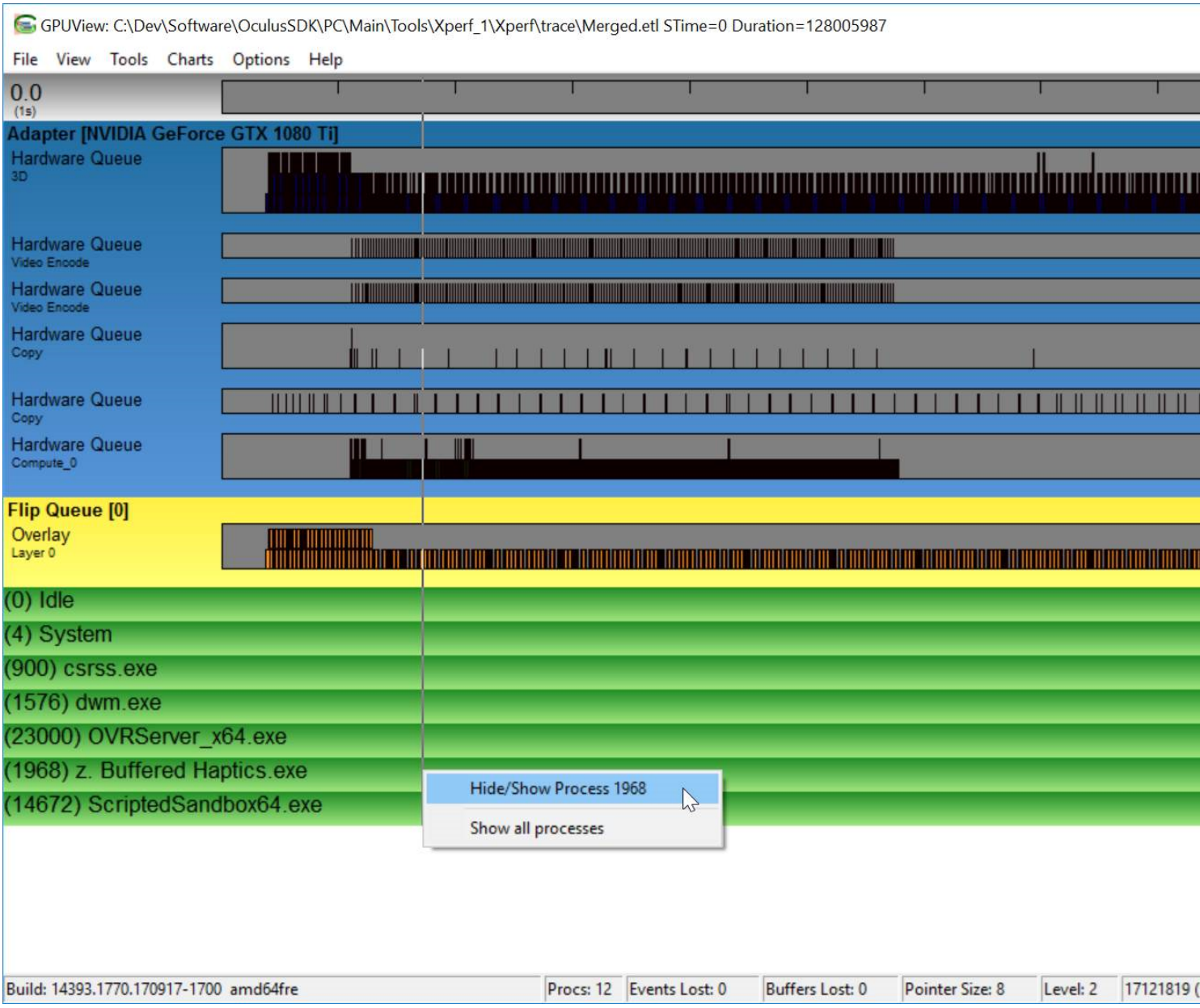
In this example, the `merged.etl` file represents the behavior of the application when the loop variable was set to 1500. This behavior may vary, however, depending on the characteristics of your hardware.

3. Many processes are captured in the trace, including kernel-level processes and processes related to other applications. These processes are not of interest in this tutorial. So, in order to simplify the display, collapse all processes with:

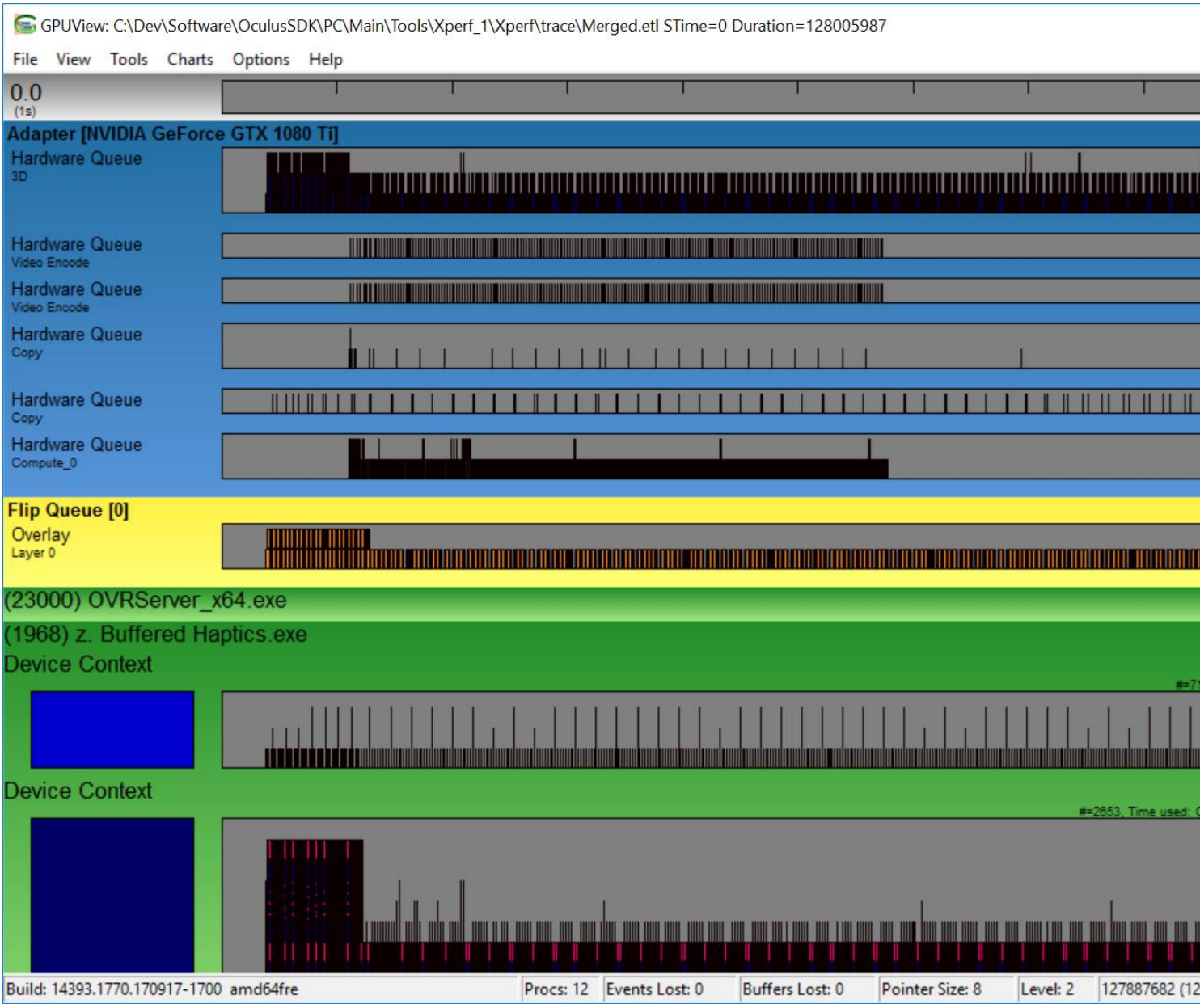
View > Process > Hide All Processes



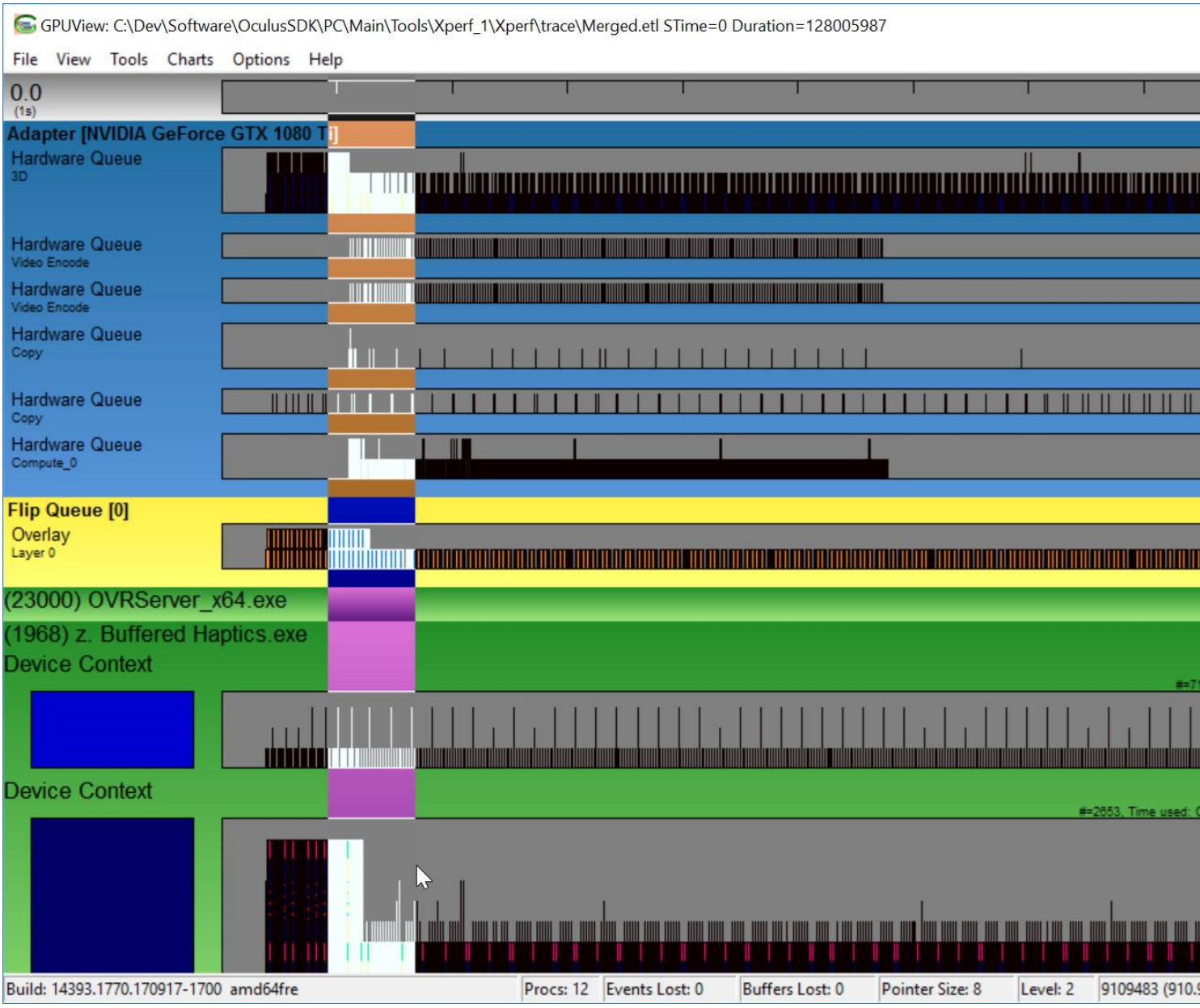
4. We only want to show the z.Buffered Haptics.exe process. So, right click on that process and select Hide/Show Process:



At the top of the screen, the hardware resources are shown. At the bottom of the screen the corresponding activities in the application are shown:

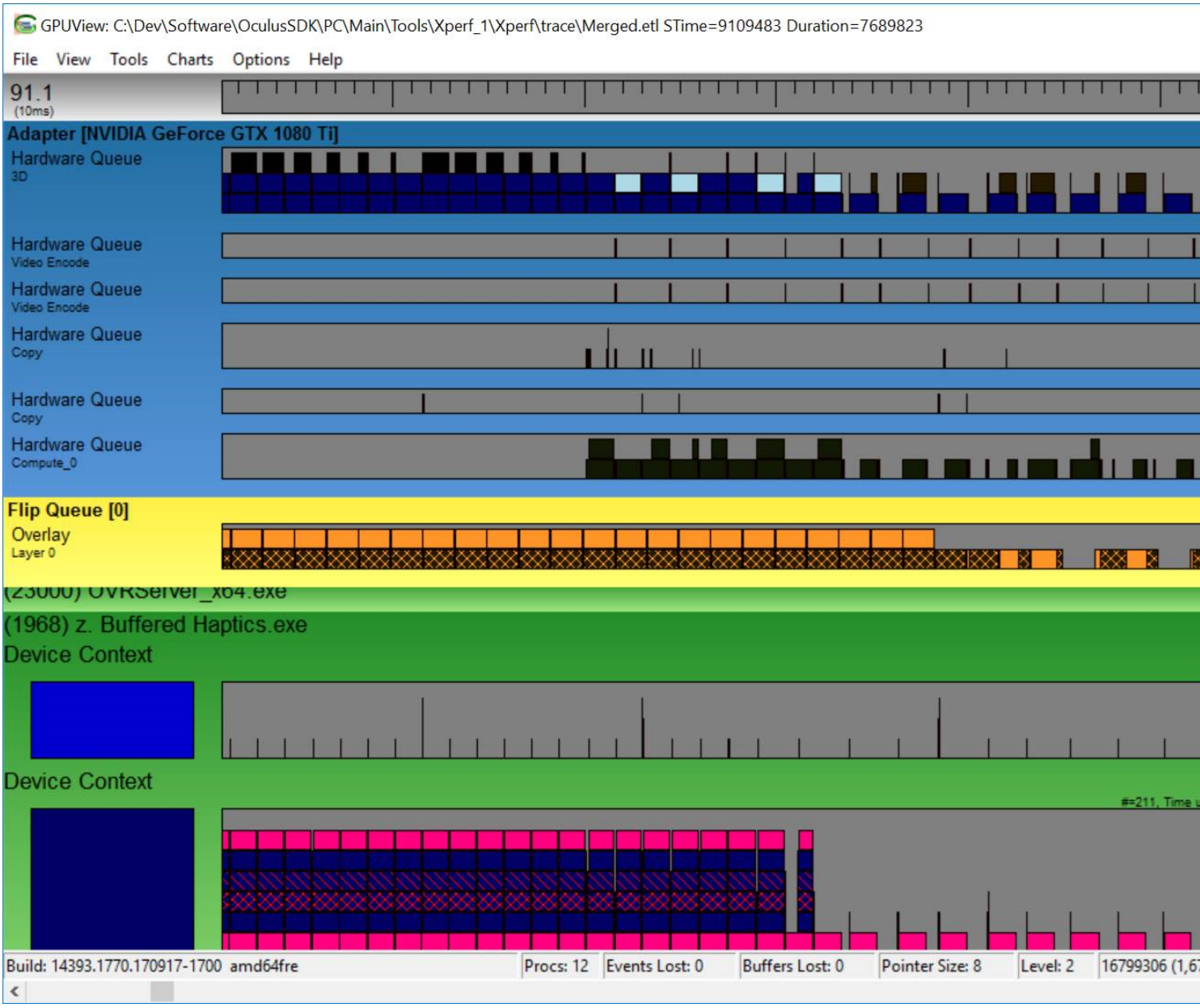


5. Zoom in by selecting a relevant portion of the event flow, and typing Ctrl-Z:



To zoom incrementally outward, type Z. To zoom all the way out to the original view, type Ctrl-H.

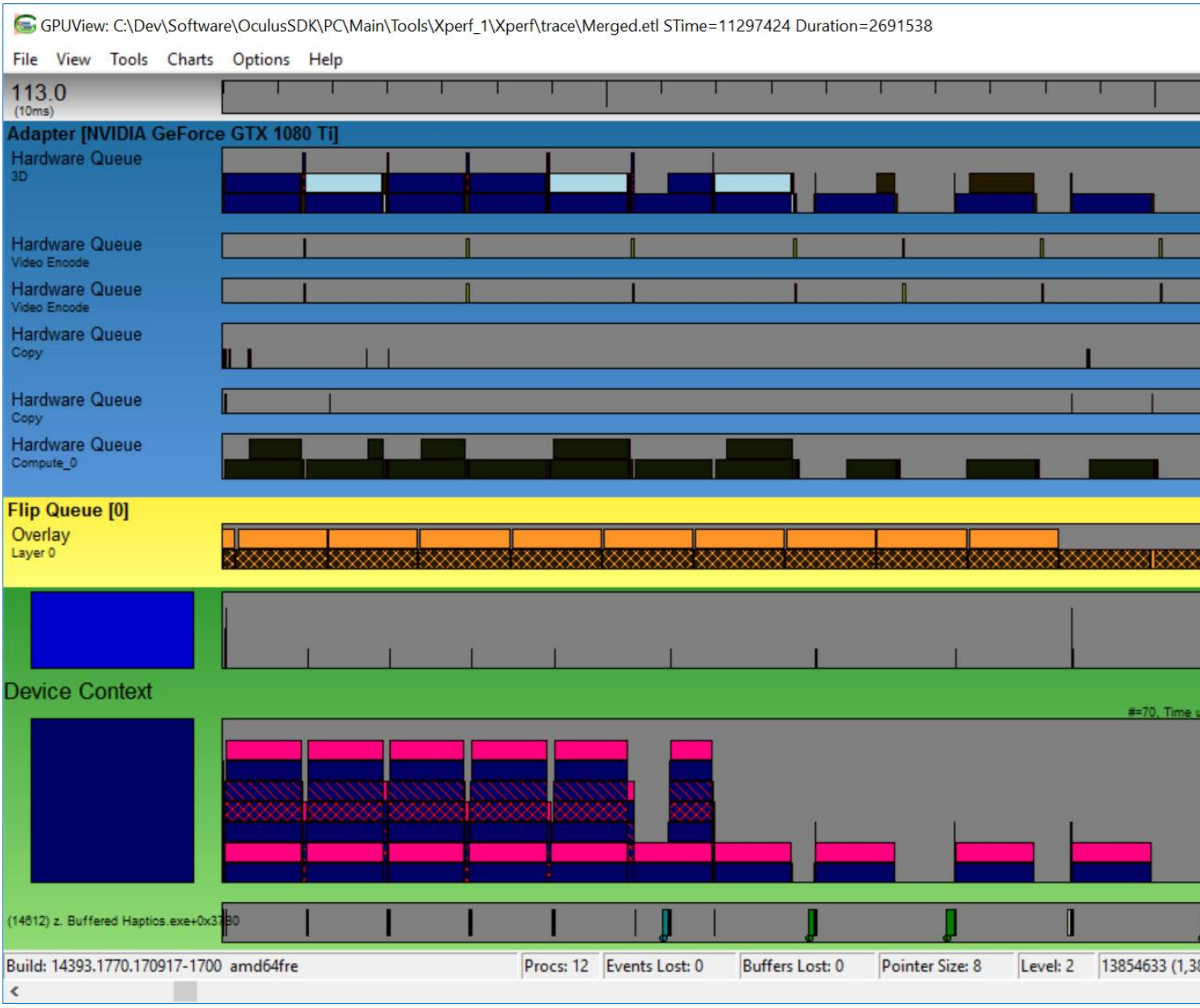
An expanded view is shown (below). In this example, the headset was stationary as the trace began, and was then moved actively around the scene. You can see that the frames are separated by space on the right hand side of the display. That may or may not indicate that frames are being dropped. This depends on the timing.



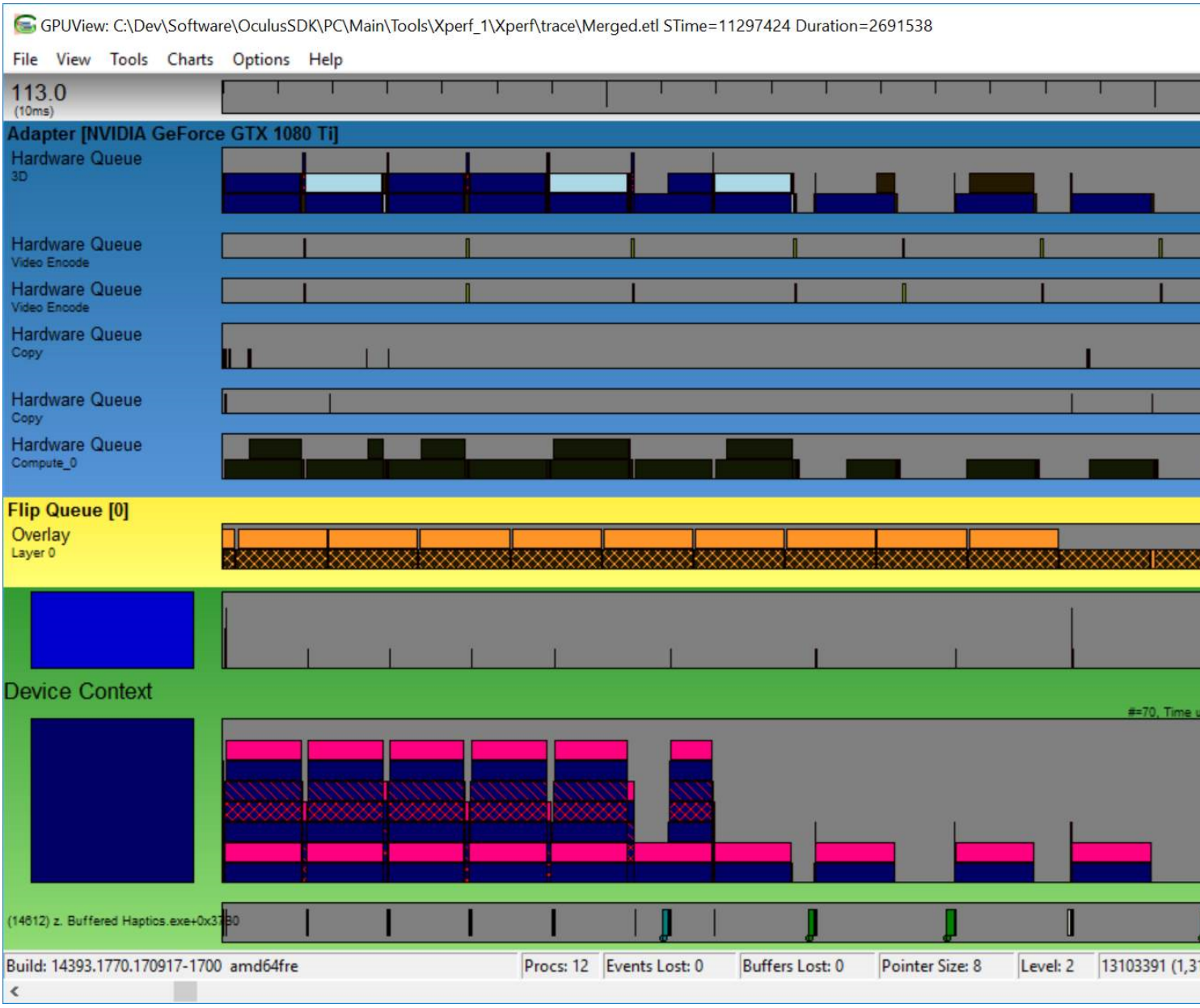
6. To examine the timing more carefully, zoom in further by selecting a section and typing Ctrl-Z:



7. You can now see the packets in greater detail:



Select a frame and look at the time scale shown at the top of the screen. The time scale increments are 10 ms. To hit the required frame rate of 90 frames per second, the frame cycle must be completed within about 11.1 ms. You can see that the frame is taking longer than that, and in fact the subsequent frame is delayed, as indicated by the blank space between the frames. So, in this example, every other frame is being dropped on the right-hand side of the timeline:

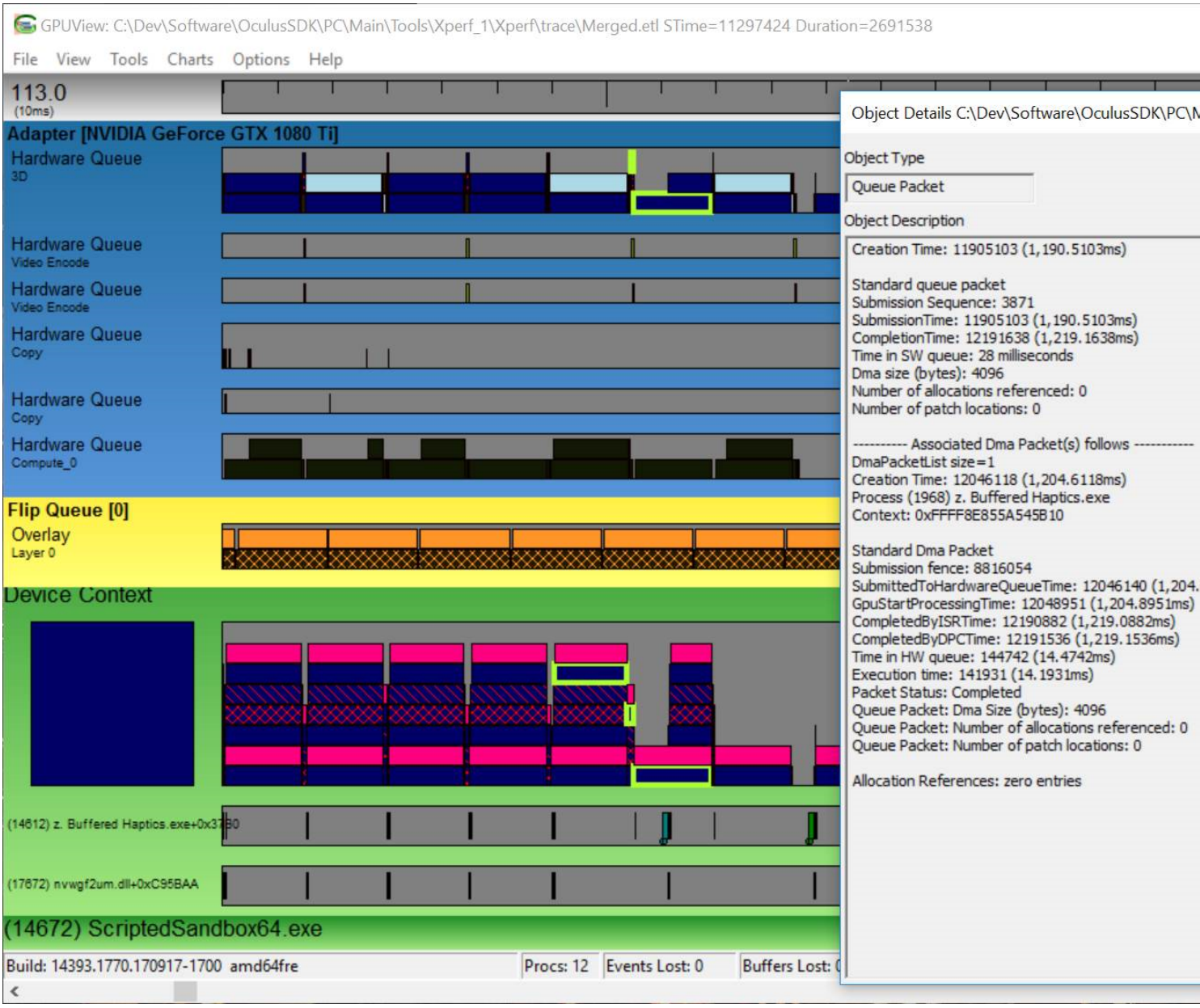


At this zoom level, you can see stacks of colored rectangles, which are called packets. A packet is the minimum indivisible unit of work that the application schedules to run on the CPU or GPU. A *fence* is a synchronization primitive that assures that one operation completes before the next operation begins.

The fences appear as the pink boxes above.

The dark blue packets are collections of graphics commands that resolve to GPU work.

8. Click on a packet. You can now see the exact life cycle for that packet, as well as detailed information about it:



In GPUView, the packets are arranged in stacks, with time progressing along the horizontal axis. The packet at the bottom is the item that is currently being worked on. The further up the stack you go, the further back in line the packet is. The height of the stack shrinks as work is finished, and grows as more work is added to the queue. By clicking on a packet, you can see its life cycle, as well as the dependencies between packets.

In WPA, we could have zoomed in and determined the exact frame index for a frame that we are interested in examining. We could then use that frame index to locate the same frame within GPUView. This is helpful because GPUView does not provide a contextual top-down view.

In the above example, the application called the DirectX driver, and requested that it render a primitive. That driver call created GPU work. So, the 3D pipe is processing that work in the highlighted selection.

The typical flow is: initialize, submit CPU work, submit related graphics work, install the fences that assure work is processed in the correct order, wait until the CPU work (and therefore the fence) finishes, then move on to the next frame.

The same GPU packet is highlighted across time, above. But at a later time, it is at the front of the queue (at the bottom), and is being processed. The width of the packet indicates how long it took to process it, in terms of the time scale shown along the top.

You can also obtain traces for the packets. If you see that your application is using the system heavily, and you don't know why, you can see the exact sequence of calls that caused the problem.

It is very useful to be able to view the VSyncs within GPUView. But, the menu item **Options > Toggle VSync** does *not* display Oculus VSyncs. This menu item displays the VSyncs for the Windows monitor.

`ovrlog_win10` and `ovrlog` generate traces that include NVIDIA or AMD vsync events, depending on the hardware that your system is using. In order to display vsyncs within GPUView, you must run the following batch file before generating the ETW trace:

```
%PROGRAMFILES%\Oculus\Support\oculus-diagnostics\ETW\IHVETW\setup.bat
```

Once the manifests are installed, the event that captures VSyncs for NVIDIA is:

```
NVIDIA-VR-DirectMode VsyncDPC
```

There are similar events for AMD.

Open the trace in GPUView, click `Tools > Event Viewer`, and then enable `NVIDIA-VR-DirectMode`, and vertical red lines should appear in your trace diagram.

Additional Resources

This section provides links to additional resources that you can consult for more information about VR application optimization issues.

For more information about creating high performance VR applications, please see the following:

- All sections under *Optimizing Your Application*.
- The Rift documentation for *Asynchronous SpaceWarp*.
- Squeezing Performance out of your Unity Gear VR Game (Oculus Developer Blog article, posted by Chris Pruett): *https://developer.oculus.com/blog/squeezing-performance-out-of-your-unity-gear-vr-game/*
- Squeezing Performance out of your Unity Gear VR Game Continued (Oculus Developer Blog article, posted by Chris Pruett): *https://developer.oculus.com/blog/squeezing-performance-out-of-your-unity-gear-vr-game-continued/*
- For an in-depth look at VR graphics optimization issues, and a close look at how to use GPUView to analyze ETW traces, see Chapter 2. Understanding, Measuring, and Analyzing VR Graphics Performance (by James Hughes, Reza Nourai, and Ed Hutchins) in the book *GPU Zen: Advanced Rendering Techniques (Wolfgang Engel*, ed): *https://www.amazon.com/GPU-Zen-Advanced-Rendering-Techniques-ebook/dp/B0711SD1DW*
- For a description of anti-aliasing techniques from John Carmack, see this post: *https://www.facebook.com/permalink.php?story_fbid=1818885715012604&id=100006735798590*
- Also see the video *Understanding VR Performance* (David Borel, speaking at Unite 2016):*https://www.youtube.com/watch?v=kGCS8LqXJ4o*

Lost Frame Capture Tool

The Lost Frame Capture tool collects information about dropped frames while your VR application is running. You can then replay the dropped frames while viewing statistical data, in order to help track down performance problems within your application.

Overview

When you submit a VR application to the Oculus Store, it is examined by Oculus engineers to ensure that it adheres to the Rift Virtual Reality Checklist (VRC), which specifies a number of requirements that your application must satisfy, including performance targets. For information about the VRC, see: *https://developer.oculus.com/distribute/latest/concepts/publish-rift-app-submission/*. If your application fails due to performance issues, it is generally because it failed to maintain the required frame rate of 90 frames per second at some point during its execution. (Large clusters of dropped frames are not allowed outside of scene transitions. Applications are allowed to drop frames while loading code into memory, or transitioning from scene to scene.)

If your application is rejected for performance reasons, you will need to address the those issues, and re-submit the application. However, in the past it has been difficult for developers to pinpoint the exact problem areas within their application. Oculus will soon begin providing an Oculus Debug Archive (ODA file) to developers, whenever their application is rejected because of performance issues. You can use the Lost Frame Capture tool to load the ODA file, and then analyze the performance of your application. With this tool, you can see exactly where the performance problems occurred, and even view the lost frames that are causing performance issues.

You can also use the Lost Frame Capture tool to analyze your applications before you submit them to Oculus. This helps you to discover and address any performance issues prior to submission.

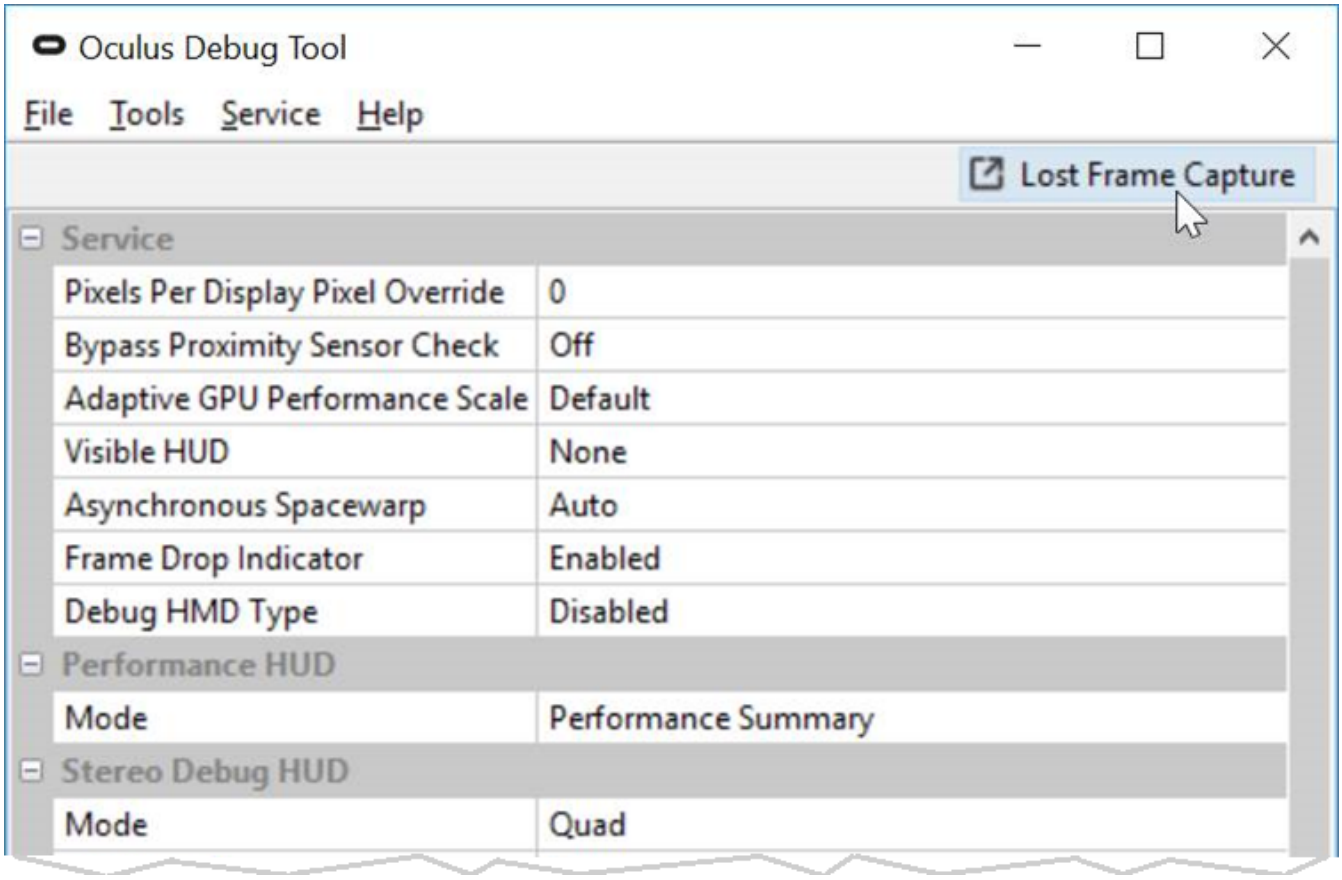
The rest of this document describes how to use the Lost Frame Capture tool.

Launching the Lost Frame Capture Tool

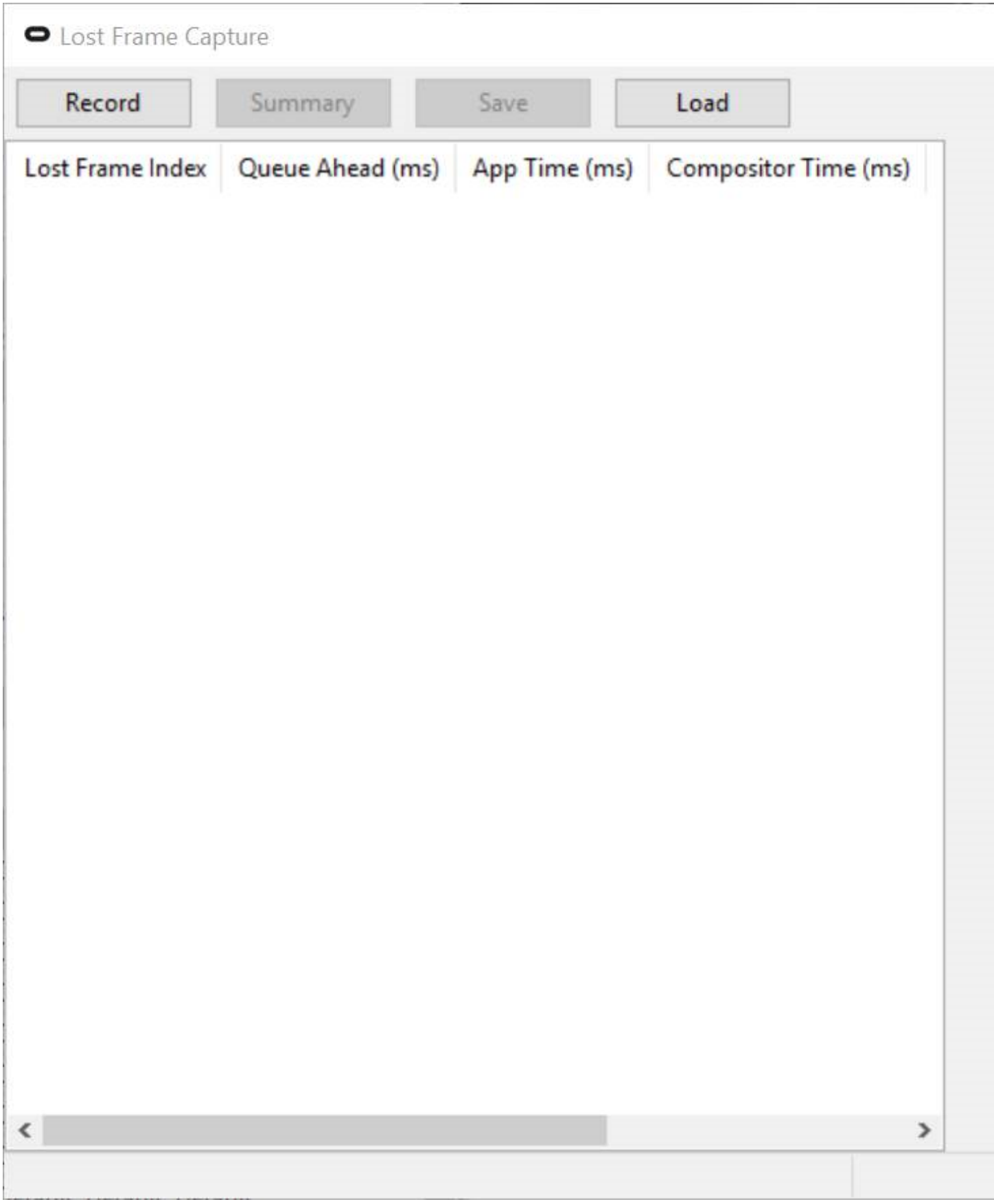
1. Run the Oculus Debug Tool, located here:

```
C:\Program Files\Oculus\Support\oculus-diagnostics\OculusDebugTool.exe
```

2. The Oculus Debug Tool appears. Click the Lost Frame Capture button:



3. The Lost Frame Capture tool appears:



4. The following actions are available when appropriate:

- Click `Record` to begin capturing lost frames from the currently running Oculus session.
- Click `Save` to save the currently captured content to an Oculus Debug Archive (ODA) file.
- Click `Load` to load an existing Oculus Debug Archive (ODA) file.
- Click `Summary` to view a statistical summary based on the currently loaded ODA file or the current capture session (whichever applies).

Analyzing the Lost Frame Capture Output

You can load and analyze any ODA file, including:

- An ODA file that you previously created yourself.
- An ODA file that you received from Oculus, if your submitted application did not pass the VRC.

You can also analyze the Lost Frame Capture output that you captured during the currently running session, without necessarily saving that output to an ODA file.

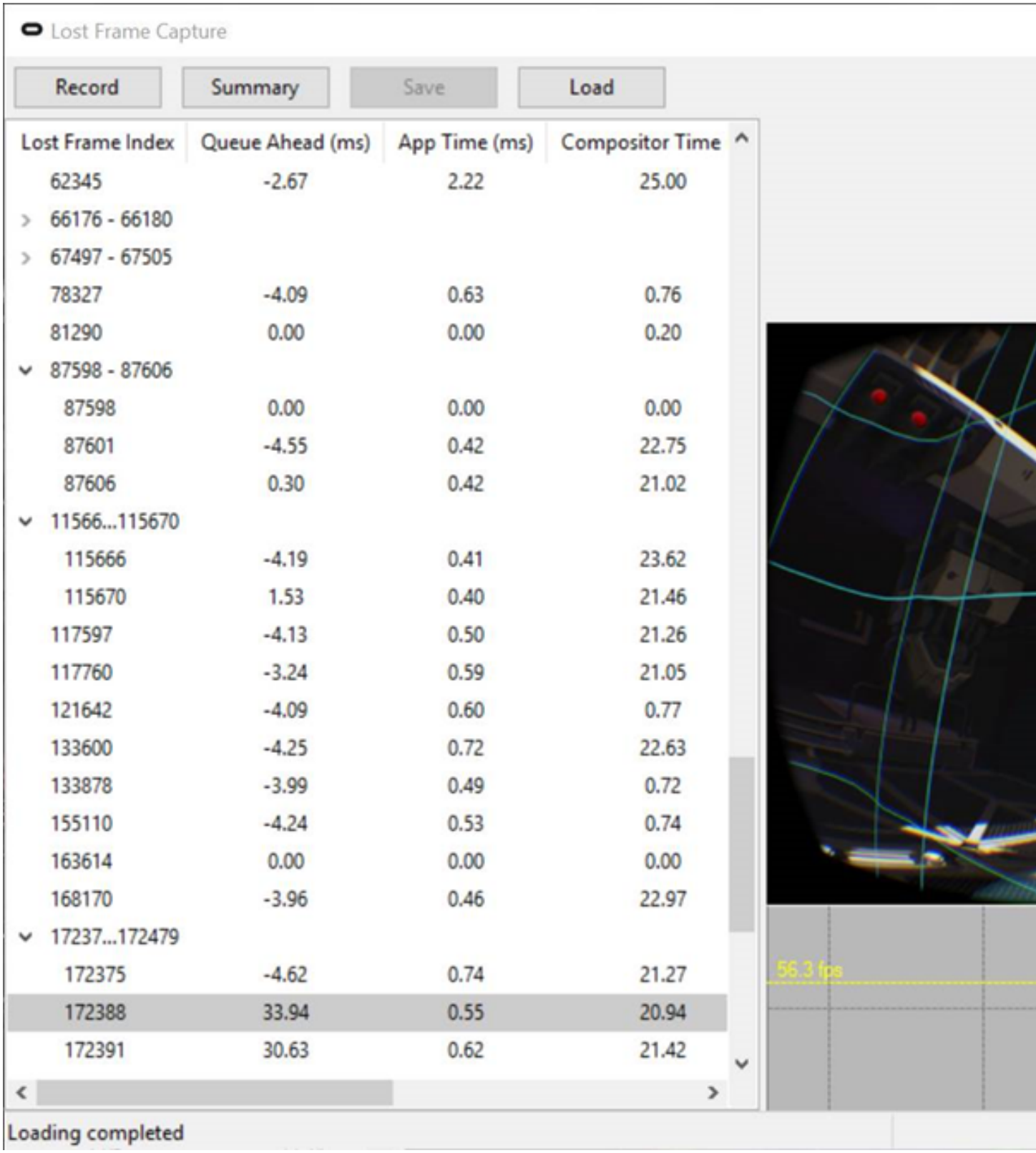
Before you submit your application to Oculus, it is a good idea to run it, and collect the Lost Frame Capture output. While running your application, you should go through all the scenarios that end users may encounter. Then, analyze the resulting output to determine if there were any large clusters of dropped frames outside of scene transitions. If you discover any performance problems, fix them before submitting your application to Oculus.

To load and analyze an ODA file, follow these steps:

1. Launch the Lost Frame Capture tool (as described above).
2. Click the Load button:



3. Select the .oda file to load.
4. The Lost Frame Capture tool populates the main window:



This window lists the frames that were lost during the time that the ODA content was being captured. The frame indexes are listed in the first column. You will notice that the lost frames are sometimes grouped into folder-like structures. This is intended to help you recognize when multiple frames are lost in close proximity to each other. Whenever a sequence of frames is lost, where there is never a gap of more than one second

between any two lost frames, those lost frames are grouped together into a folder structure—called a *lost frame cluster*.

You can select a frame at the top level of the list, or a frame from within a lost frame cluster. The rendered left and right eye views for the frame are displayed on the right. A graph appears below, which indicates the frame rate for a set of lost frames, where 90 frames per second is the top-most value in the graph. When the graph dips down, it indicates that frames are being generated at a lower frame rate (which, if it is sustained, is not acceptable since it results in a poor user experience).

In the example above, the currently selected frame was generated at a rate of 56.3 frames per second, as indicated by the horizontal line. You can step through the lost frames using the arrow keys, and see how the line moves over time. The point at which the horizontal line intersects the vertical dashed line in the grid, and which also intersects the graph, indicates where the currently selected frame is in the timeline.

This graph view helps you to visualize how the frame rate for the currently selected frame compares with other nearby frames (if you are stepping through frames within a cluster). For example, you can compare the current level of the horizontal line to the overall graph. This can make it much easier to spot where a problem is beginning, and where it is at its worst. In many cases, you should be able to get an intuitive grasp of what the problem is by simply looking at the content of the frames. For example, an object may come into view when the frame rate drops. Perhaps the shader that is used by that object is too complex, or some other aspect of the rendering process for that object is using up too much of the CPU or GPU resources.

Columns in the Display

The columns displayed by the Lost Frame Capture tool are listed in the following table. You can click on any column header in order to sort the rows by that column. If you sort by a column other than Lost Frame Index, the clusters are sorted as a group at the top of the list, and the non-clustered individual frames are sorted as a group at the bottom of the list.

The columns include:

Lost Frame Index	The frame index for the lost frame in this row.
Queue Ahead (ms)	The queue ahead time in milliseconds for the frame. (For information about queue ahead time, see *https://developer.oculus.com/documentation/pcsdk/latest/concepts/dg-render/#dg-queue-ahead*.)
App Time (ms)	The amount of time that the application spent processing the frame (in milliseconds).
Compositor Time (ms)	The amount of time that the Oculus compositor spent processing the frame (in milliseconds).
Frame Count	The number of frame cycles represented by the row. For a single frame, this number indicates the number of frame cycles that this frame required. Often this number is 2, since the frame was lost, and was not able to be rendered within a single frame cycle. For lost frame clusters, this number can be much higher, since it represents the number of frame cycles required by all of the frames within the cluster. The Frame Count column can be very helpful, since you can see by simply scanning down the column where a large number of frames were all dropped in close proximity to each other. When this happens, it is likely that you have a performance issue that occurred at that location.
Frame Rate (hz)	The frame rate for the currently selected row.

	For a single frame, this is the frame rate that would result if frames were processed continually at the rate of the currently selected frame, expressed in frames per second (hz). For a cluster, this is the frame rate that would result if the *average frame rate* of the frames within the cluster proceeded continuously, expressed in frames per second (hz). Note that this is the average frame rate for the *lost frames* within the time period represented by the cluster. It is entirely possible (and would, in fact, usually be that case) that many other frames were processed within that time frame, but they were not lost frames and are not shown in the Lost Frame Capture tool.
V-Sinc	The vsinc (vertical synchronization) for the row. When a single frame is selected, this is the location in the timeline where the vsync occurred for that frame (in seconds from the start of the capture period). When a cluster is selected, this is the vsync for the first frame in the cluster.
Duration (ms)	The length of time (in milliseconds) that frames were persistently dropped, for the currently selected row. When a single frame is selected, this is the length of time that was spent processing that frame. When a cluster is selected: • The period begins when the system starts processing the first dropped frame in the cluster. • The period ends when the system finishes processing the last dropped frame in the cluster. • The Duration value includes the processing time for all frames that were processed between the start time and the end time, including dropped frames and non-dropped frames.

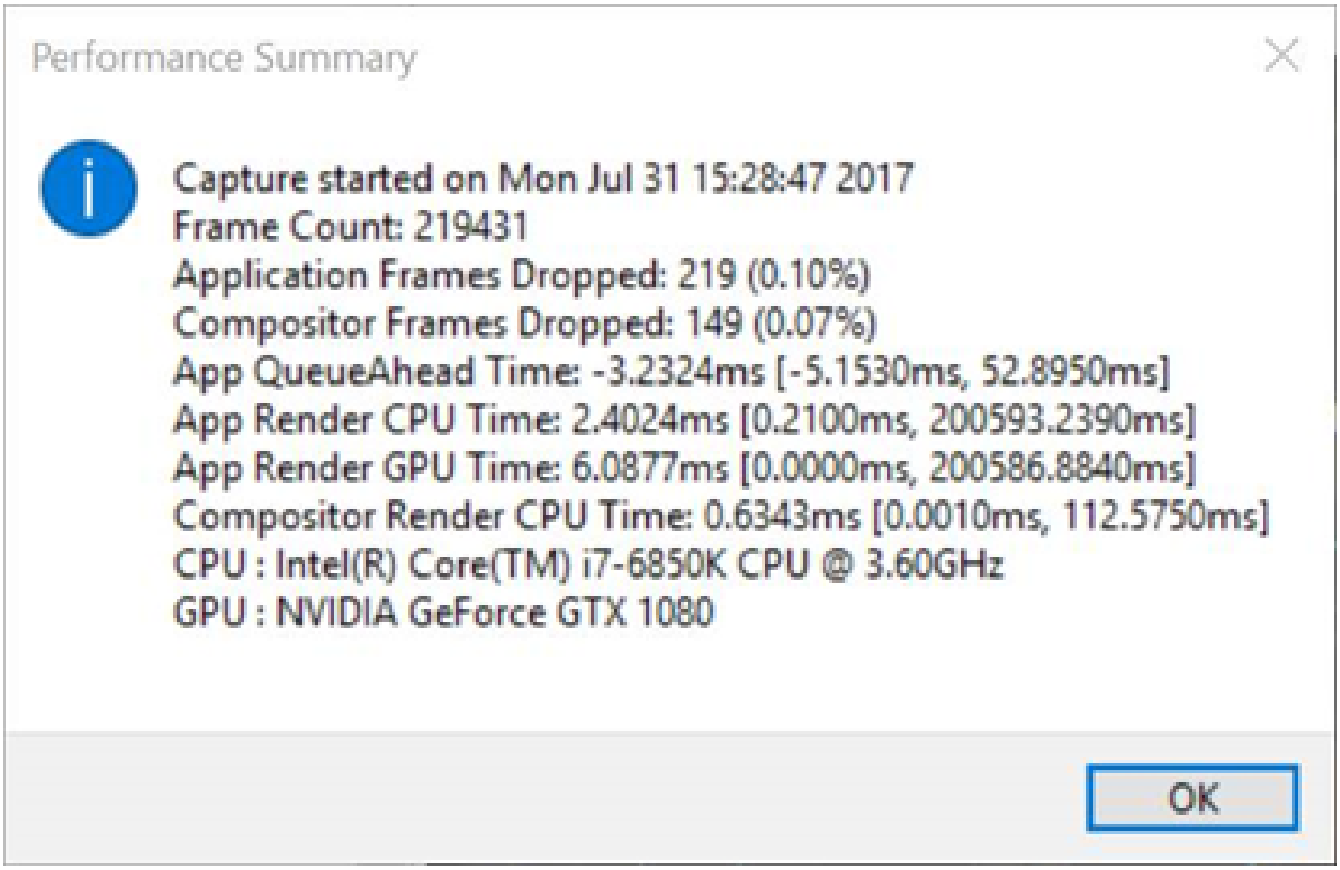
Recording an ODA File

To capture any lost frames produced by your application, and then save that content as an ODA file:

1. Click `Record`.
2. Perform whatever actions are of interest in the VR application.
3. Take off headset, and stop the recording. (A `Stop` button is available in the Lost Frame Capture tool window.)
4. Click `Save` to save the captured data to an ODA file.

View Performance Summary

To view a set of overall statistics for the currently open ODA file, or the current recording session, click the `Summary` button. The Performance Summary popup window is displayed:



SDK Performance Statistics

The SDK performance statistics provide information about application and compositor performance on the system.

Stats are populated after each call to `ovr_EndFrame`. To get performance stats, use `ovr_GetPerfStats()`.

To reset the stats, use `ovr_ResetPerfStats`.

Note: If your performance tool runs separately from your application, make sure to read the VisibleProcessId in case your application loses focus.

The following table describes performance statistics:

Table 1: Statistics

Statistics	
Description	
ovrPerfStatsPerCompositorFrame	The per-compositor frame statistics in the following tables.
AnyFrameStatsDropped	If the app calls `ovr_EndFrame` at a rate less than 18 fps, then when calling ovr_GetPerfStats, expect AnyFrameStatsDropped to become ovrTrue while FrameStatsCount is equal to ovrMaxProvidedFrameStats.

Statistics Description	
AdaptiveGpuPerformanceScale	An edge-filtered value that you can use to adjust the graphics quality of the application to keep the GPU utilization in check. The value is calculated as: (desired_GPU_utilization / current_GPU_utilization). When this value is 1.0, the GPU is doing the right amount of work for the app. Lower values mean the application needs to reduce the GPU utilization. Note: If the app directly drives render-target resolution using this value, make sure to take the square-root of the value before scaling the resolution with it. Changing the render target resolution is only one of the many things your application can do increase or decrease the amount of GPU utilization. Since AdaptiveGpuPerformanceScale is edge-filtered and does not change rapidly (i.e., it reports non-1.0 values once every couple of seconds), your application can make the necessary adjustments and continue watching the value to see if it has been satisfied.
AswIsAvailable	Returns true if ASW is available for this system, based on the user's GPU, operating system, and debug override settings.

The following table describes statistics specific to your application's performance:

Table 2: Application Statistics

Statistic Description	
AppFrameIndex	Index that increments with each `ovr_EndFrame` call.
AppDroppedFrameCount	Increments each time the application fails to submit a new set of layers using `ovr_EndFrame` before the compositor is executed before each V-Sync (Vertical Synchronization).
AppMotionToPhotonLatency	Latency from when the last predicted tracking information was queried by the application using ovr_GetTrackingState() to when the middle scanline of the target frame is illuminated on the HMD display. This is the same information provided by the Latency Timing HUD.
AppQueueAheadTime	To improve CPU and GPU parallelism and increase the amount of time that the GPU has to process a frame, the SDK automatically applies queue ahead up to 1 frame. This value displays the amount of queue ahead time being applied (in milliseconds). For more information about adaptive queue ahead, see *Adaptive Queue Ahead* on page 33.

Statistic Description	
AppCpuElapsedTime	The time difference from when the application continued execution on CPU after `ovr_EndFrame` returned the subsequent call to `ovr_EndFrame`. This will show "N/A" if the latency tester is not functioning as expected (e.g., HMD display is sleeping due to prolonged inactivity). This includes the IPC call overhead to the compositor after `ovr_EndFrame` is called by the client application.
AppGpuElapsedTime	The total GPU time spent on rendering by the client application. This includes the work done by the application after returning from `ovr_EndFrame`, using the mirror texture if applicable. It can also includes GPU command-buffer "bubbles" if the application's CPU thread doesn't push data to the GPU fast enough to keep it occupied. Similarly, if the app pushes the GPU close to full-utilization, the work on next frame (N+1) might be preempted by the compositor's render work on the current frame (N). Because of how the application GPU timing query operates, this can lead to artificially inflated application GPU times as they will start to include the compositor GPU usage times.

The compositor operates asynchronously and will increment for each vsync, regardless of whether the application calls `ovr_EndFrame`.

The following table describes compositor statistics:

Table 3: Compositor Statistics

Statistic Description	
CompositorFrameIndex	Index that increments each time the SDK compositor completes a distortion/TimeWarp pass.
CompositorDroppedFrameCount	Increments each time the compositor fails to present a new rendered frame at V-Sync (Vertical Synchronization).
CompositorLatency	Specifies the TimeWarp latency, which corrects app latency and dropped frames.
CompositorCpuElapsedTime	The amount of time in seconds spent on the GPU by the SDK compositor. Any time spent on the compositor takes available GPU time away from the application.
CompositorGpuElapsedTime	The amount of time the GPU spends executing the compositor renderer. This includes TimeWarp and distortion of all layers submitted by the application. The amount of active layers, their resolutions, and

Statistic Description	
	the requested sampling quality can all affect the GPU times.
CompositorCpuStartToGpuEndElapsedTime	The amount of time from when the CPU kicks off the compositor to when the compositor completes distortion and TimeWarp on the GPU. If the time is not available, it returns -1.0f.
CompositorGpuEndToVsyncElapsedTime	The amount of time between when the GPU completes the compositor rendering to the point in time when V-Sync is hit and that buffer starts scanning out on the HMD.

The Asynchronous SpaceWarp (ASW) HUD displays activity and tracking statistics for ASW, which runs as part of the Oculus Runtime Compositor. ASW automatically activates when an application fails to meet the required native frame rate for the connected HMD. Once active, ASW forces the application to run at half the normal frame rate while extrapolating every other frame. This gives the application more time to complete its work.

The following table describes Asynchronous SpaceWarp (ASW) statistics:

Table 4: ASW Statistics

Statistic Description	
AswIsActive	Shows the availability and current status of ASW. "Not Available" can be due to the OS and/or GPU type used on the PC. "Available - Not Active" will mean the application is successfully maintaining the required native refresh rate, so ASW is not activated.
AswActivatedToggleCount	Tracks the number of times ASW has been activated for the lifetime of the HMD.
AswPresentedFrameCount	Tracks the number of frames extrapolated by ASW that were displayed. When ASW is active, since the app is forced to run at half-rate, expect this value to increase by 45 fps on a 90 Hz refresh rate HMD.
AswFailedFrameCount	Tracks the number of extrapolated frames ASW needed to display, but failed to prepare in time. This can occur for different reasons, but might be caused by contention for OS resources or when the capabilities of the system are exceeded.

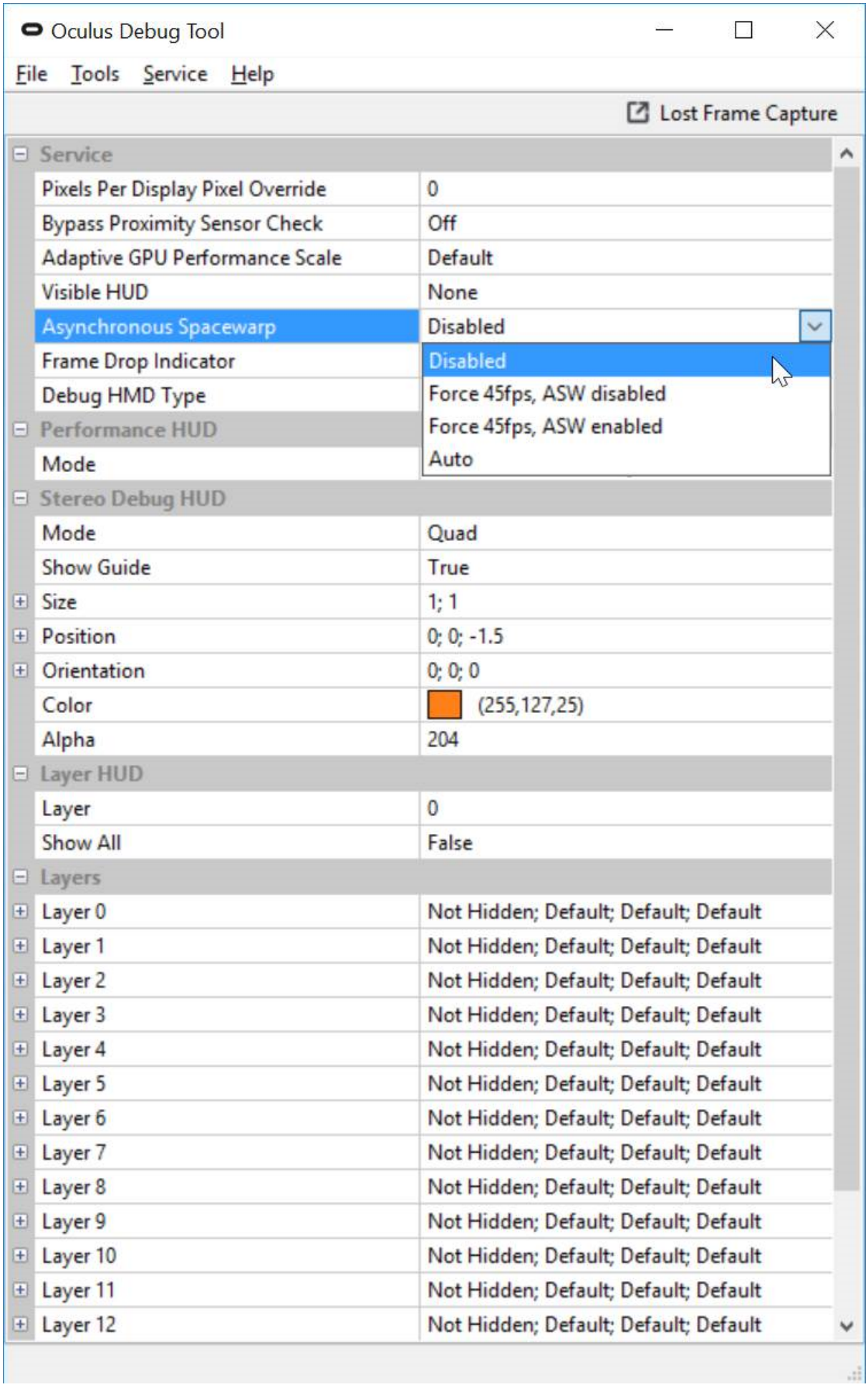
Oculus Debug Tool

The Oculus Debug Tool enables you to view performance or debugging information within your game or experience.

To use the tool:

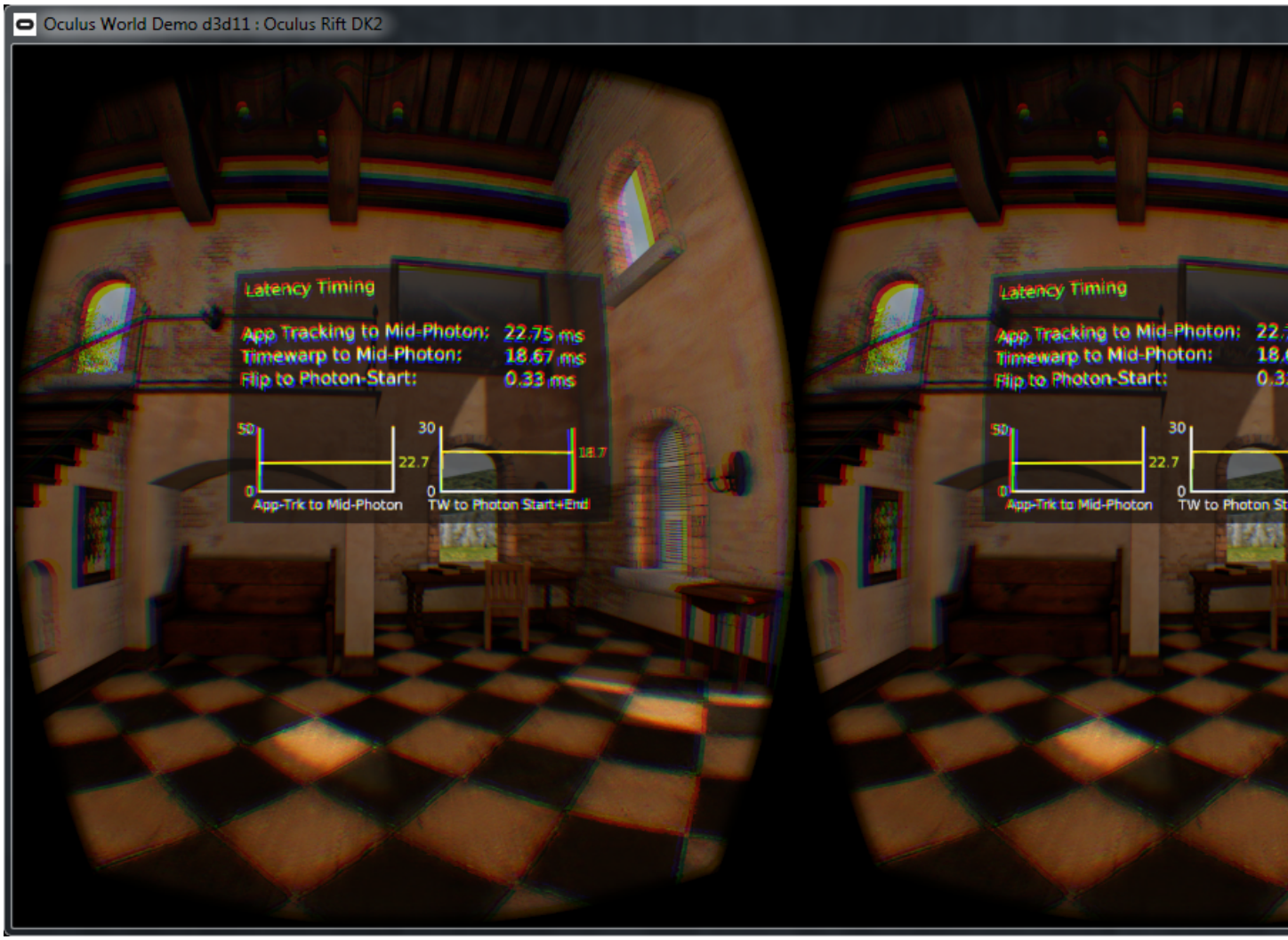
1. Make sure that you have admin privilege. This is required in order to run the Oculus Debug Tool.

2. Go to `Program Files\Oculus\Support\oculus-diagnostics\`. Note that the Oculus Debug Tool should always be run directly from this location, in order to ensure a version match with the Oculus distribution. If you copy the Oculus Debug Tool to another location, it might not work after subsequent Oculus updates.
3. Double-click `OculusDebugTool.exe`. The Oculus Debug Tool opens.
4. It is a good idea to turn off Asynchronous Spacewarp (ASW), so that you can get a true sense of how your application is performing (without the assistance of ASW). To do this, set the Asynchronous Spacewarp option to Disabled:


Oculus Debug Tool
File
Tools
Service
Help
Lost Frame Capture
Service
Pixels Per Display Pixel Override
0
Bypass Proximity Sensor Check
Off
Adaptive GPU Performance Scale
Default
Visible HUD
None
Asynchronous Spacewarp
Disabled
Frame Drop Indicator
Debug HMD Type
Performance HUD
Mode
Disabled
Force 45fps, ASW disabled
Force 45fps, ASW enabled
Auto
Stereo Debug HUD
Mode
Quad
Show Guide
True
Size
1; 1
Position
0; 0; -1.5
Orientation
0; 0; 0
Color
(255,127,25)
Alpha
204
Layer HUD
Layer
0
Show All
False
Layers
Layer 0
Not Hidden; Default; Default; Default
Layer 1
Not Hidden; Default; Default; Default
Layer 2
Not Hidden; Default; Default; Default
Layer 3
Not Hidden; Default; Default; Default
Layer 4
Not Hidden; Default; Default; Default
Layer 5
Not Hidden; Default; Default; Default
Layer 6
Not Hidden; Default; Default; Default
Layer 7
Not Hidden; Default; Default; Default
Layer 8
Not Hidden; Default; Default; Default
Layer 9
Not Hidden; Default; Default; Default
Layer 10
Not Hidden; Default; Default; Default
Layer 11
Not Hidden; Default; Default; Default
Layer 12
Not Hidden; Default; Default; Default

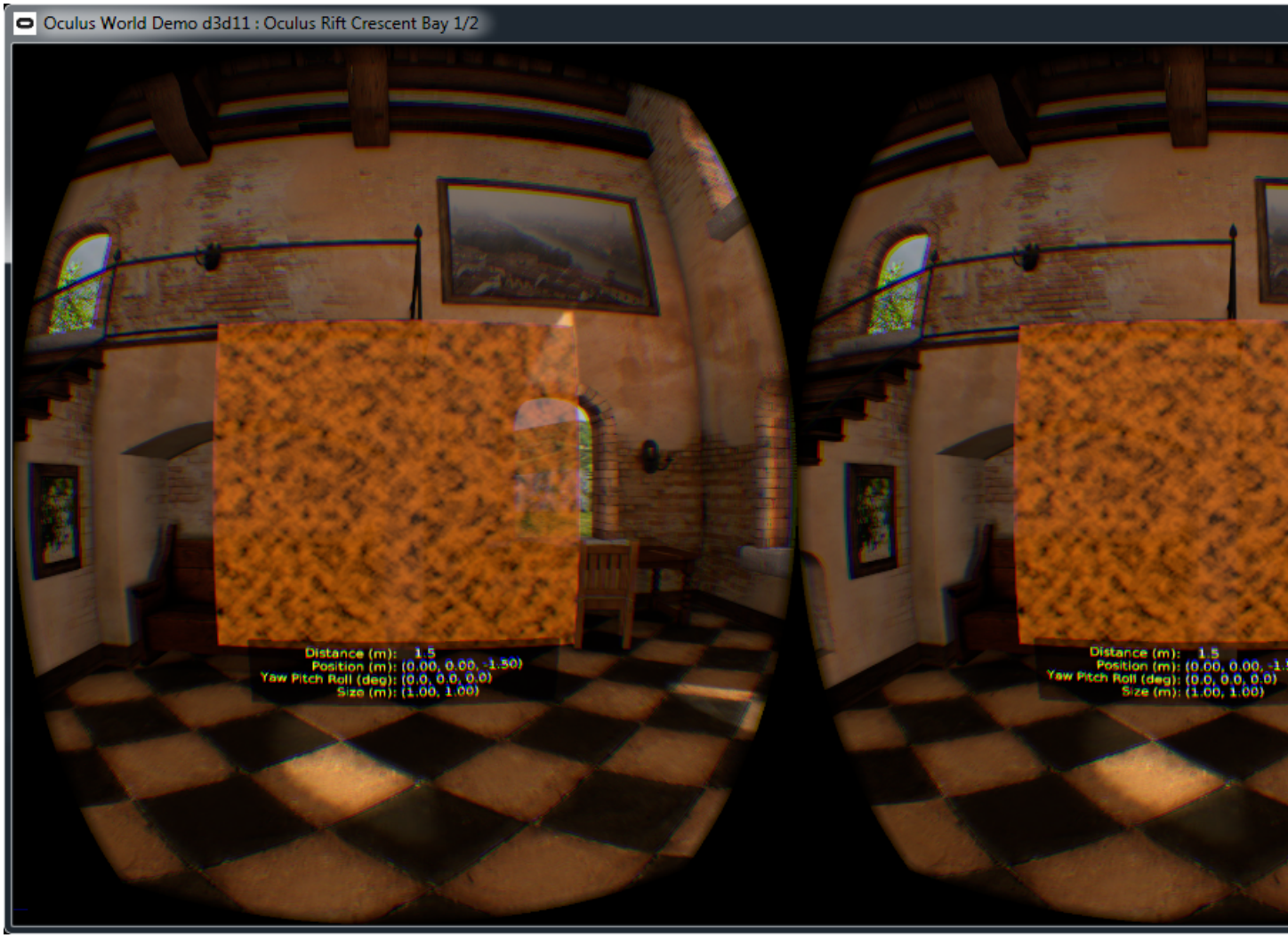
5. Select the Visible HUD display that you wish to view. Options include: None (no HUD is displayed), Performance HUD, Stereo Debug HUD, or Layer HUD.
6. If you selected Performance HUD, select which Performance HUD you want to view. Options include: Latency Timing, Render Timing, Performance Headroom, and Version Information. For more information, see *Performance Head-Up Display* on page 139.

   The following is an example of the Performance HUD:



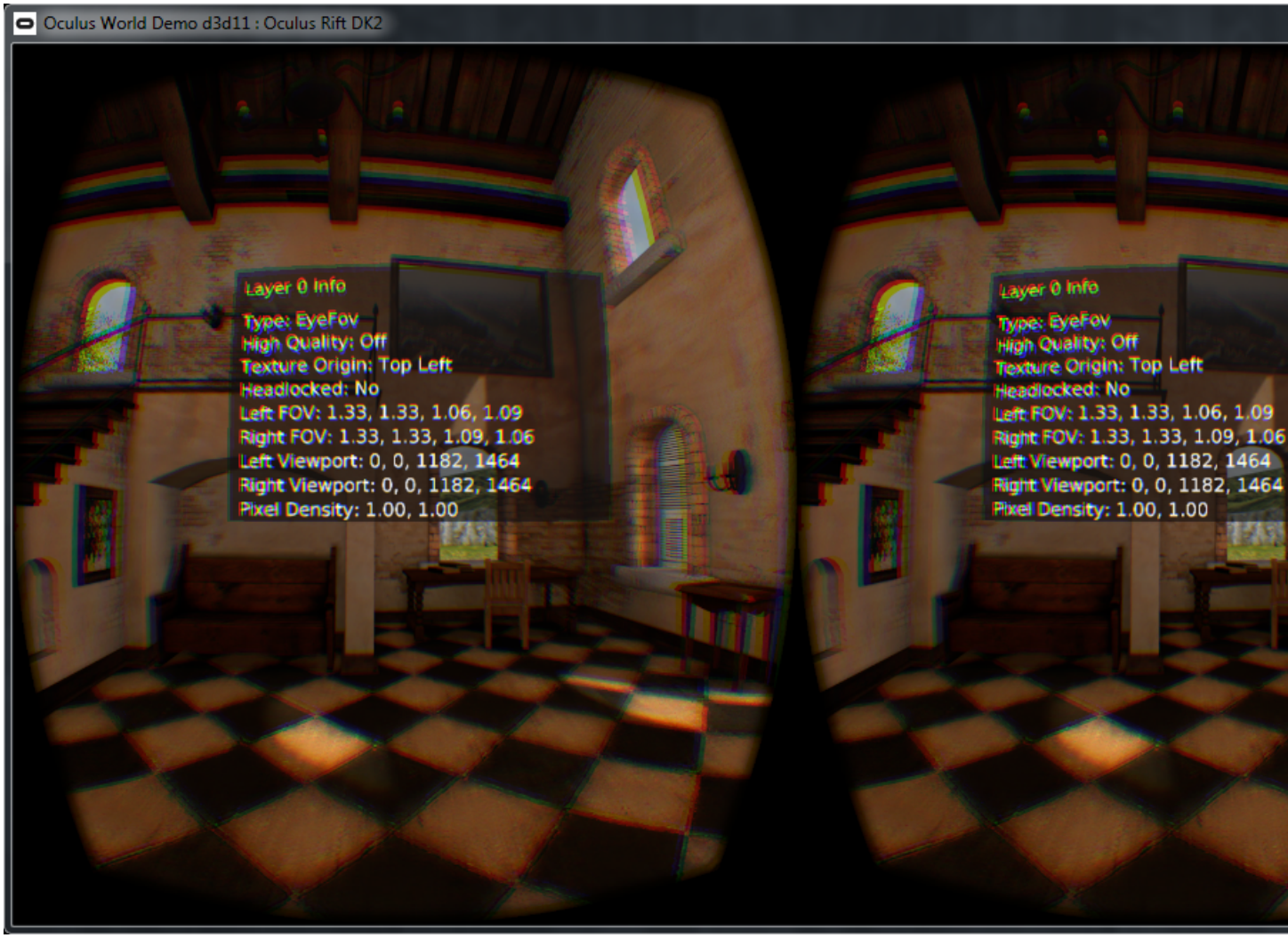
7. If you selected Stereo Debug HUD, configure the mode, size, position, and color from the Stereo Debug HUD options.

   The following is an example of the Stereo Debug HUD:



8. If you selected Layer HUD. select the layer for which to show information or select the Show All check box.

   The following is an example of the Layer HUD:



9. Select Launch App from the File menu and select the executable of the application.
10. Put on the headset and view the results.

Performance Profiler

The Oculus Performance Profiler displays a graph that shows statistics on the performance of your application.

You can view the statistics in real time or export them to a comma-separated value (CSV) file that you can import into a spreadsheet or database.

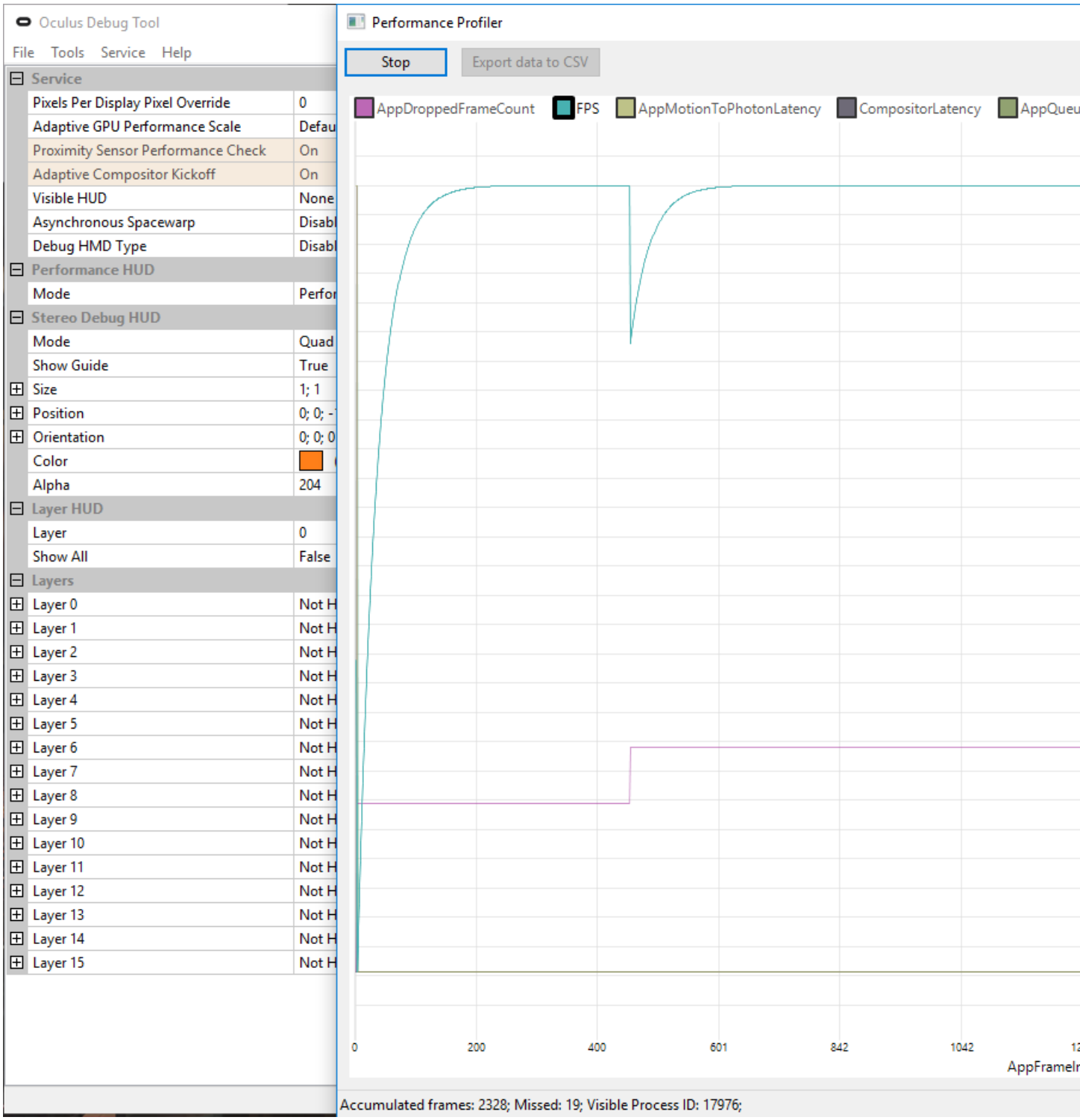
Note: For more information about the statistics collected, see *Performance Head-Up Display* on page 139.

To use the tool:

1. Go to Program Files\Oculus\Support\oculus-diagnostics\OculusDebugTool.exe.
2. Double-click `OculusDebugTool.exe`. The Oculus Debug Tool opens.
3. Select Performance Profiler from the Mode list box.
4. Select Launch App from the File menu and select the application to profile.
5. Click Start.

6. Have someone put on the headset and start using with the application.
7. View the results on screen.

Figure 15: Peformance Profiler



8. When you are finished, click Stop.
9. To export the results to CSV, click Export data to CSV and save the file.

Performance Head-Up Display

The Performance Head-Up Display (HUD) enables you or your users to view performance information for applications built with the SDK.

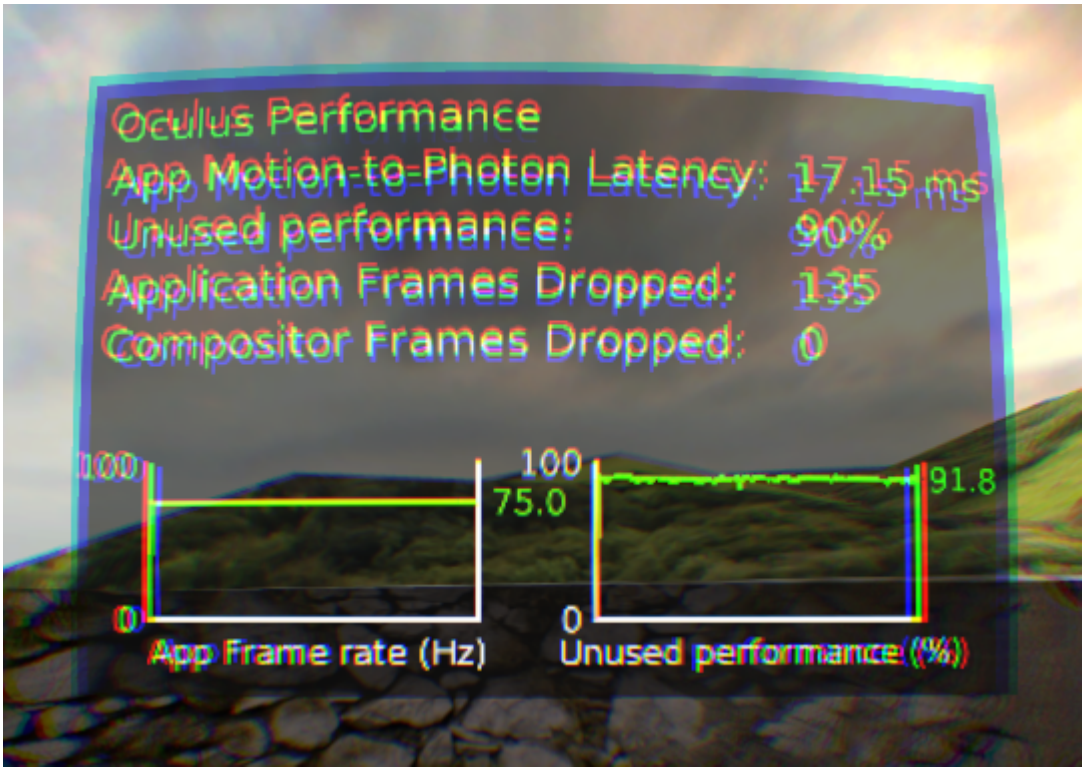
The Performance HUD screens are rendered by the compositor, which enables them to be displayed with a single SDK call to `ovr_EndFrame`. In the Oculus Debug Tool or OculusWorldDemo, you can toggle through the Performance HUD screens by pressing F11.

Performance Summary

The Performance Summary HUD displays the frame rate of the application and the unused hardware performance available. This HUD can be used by you or the user to tune an application's simulation and graphics fidelity. Because the user cannot disable V-Sync, it can be used to gauge performance instead of a frame rate counter. It is also useful for troubleshooting whether issues are related to the application or the hardware setup.

The following image shows the Performance Summary HUD:

Figure 16: Performance Summary HUD



The following table describes each metric:

Metric	Description
App Motion-to-Photon Latency	Latency from when the last predicted tracking information was queried by the application using ovr_GetTrackingState() to when the middle scanline of the target frame is illuminated on the HMD display. This is the same information provided by the Latency Timing HUD.

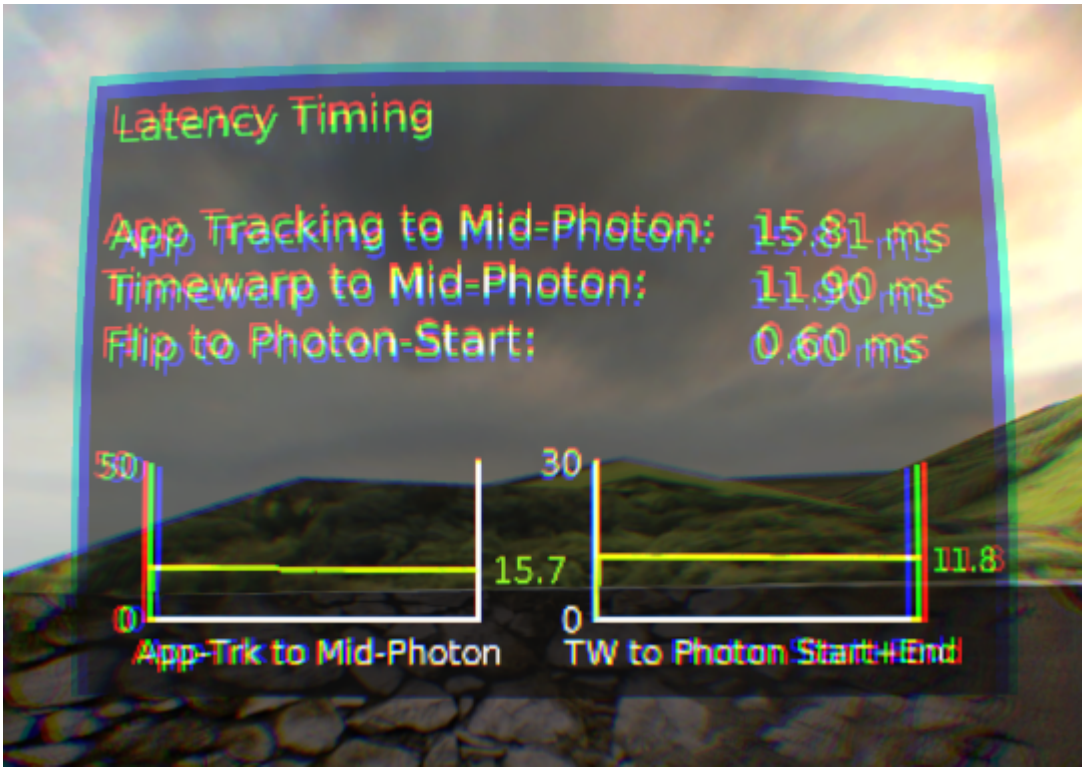
Metric	Description
Unused performance	Designed to help the user verify that the PC is powerful enough to avoid dropping frames, this displays the percentage of available PC performance not used by the application and compositor. This is calculated using the CPU and GPU time tracked by the Application Render Timing HUD divided by the native frame time (inverse of refresh rate) of the HMD. Note: As GPU utilization approaches 100%, adaptive queue ahead will choose an earlier render start point. If this start point overlaps with the compositor process in the previous frame, the performance will appear spiky. If you start to lower utilization, the graph will show an initial drop before becoming more linear.
Application Frames Dropped	Increments each time the application fails to submit a new set of layers using `ovr_EndFrame` before the compositor is executed before each V-Sync (Vertical Synchronization). This is identical to App Missed Submit Count in the Application Render Timing HUD.
Compositor Frames Dropped	Increments each time the compositor fails to present a new rendered frame at V-Sync (Vertical Synchronization). This is identical to Compositor Missed V-Sync Count in the Compositor Render Timing HUD.

Latency Timing

The Latency Timing HUD displays the App to Mid - Photon, Timewarp to Photon - Start, and Timewarp to Photon - Start graphs.

The following screenshot shows the Latency Timing HUD:

Figure 17: Latency Timing



The following table describes each metric:

Table 5: Latency Timing HUD

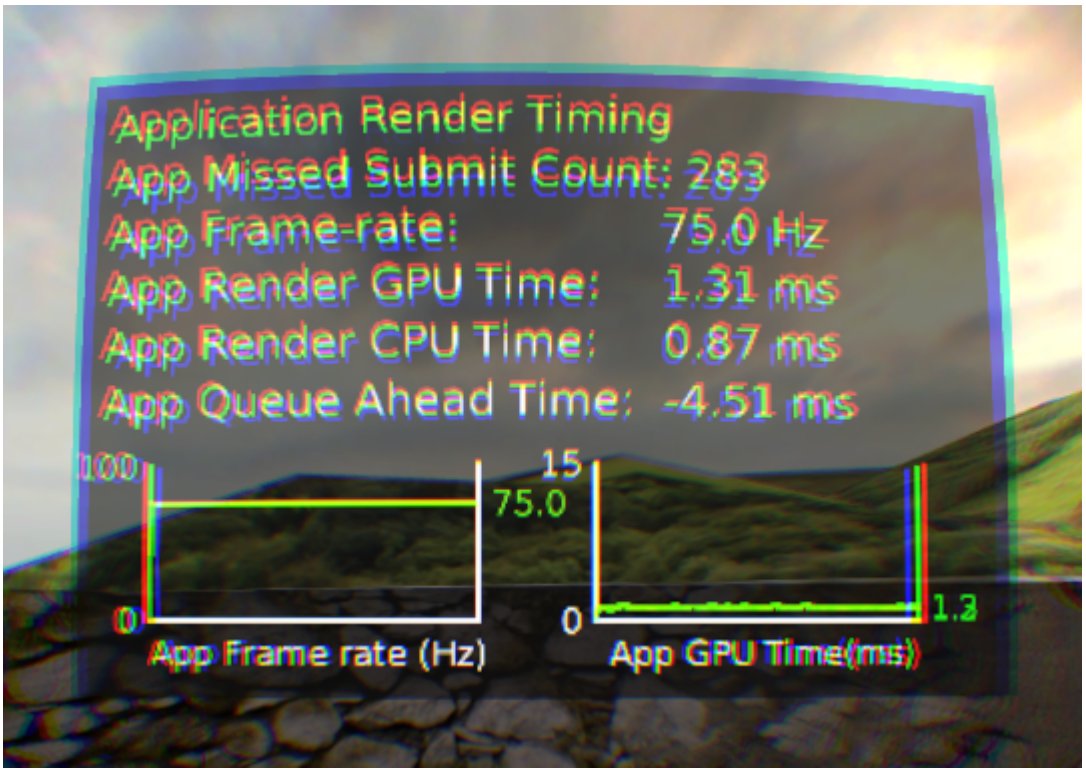
Metric	Description
App Tracking to Mid-Photon	Latency from when the app called ovr_GetTrackingState() to when the target frame eventually was shown (i.e.illuminated) on the HMD display - averaged mid - point illumination
Timewarp to Mid-Photon	Latency from when the last predictied tracking info is fed to the GPU for timewarp execution to the point when the middle scanline of the target frame is illuminated on the HMD display
Flip to Photon - Start	Time difference from the point the back buffer is presented to the HMD to the point the target frame's first scanline is illuminated on the HMD display

Application Render Timing

The Application Render Timing HUD displays application-specific render timing information.

The following screenshot shows the Application Render Timing HUD:

Figure 18: Application Render Timing



The following table describes each metric:

Table 6: Application Render Timing HUD

Metric	Description
App Missed Submit Count	Increments each time the application fails to submit a new set of layers using `ovr_EndFrame` before the compositor is executed and before each V-Sync (Vertical Synchronization).

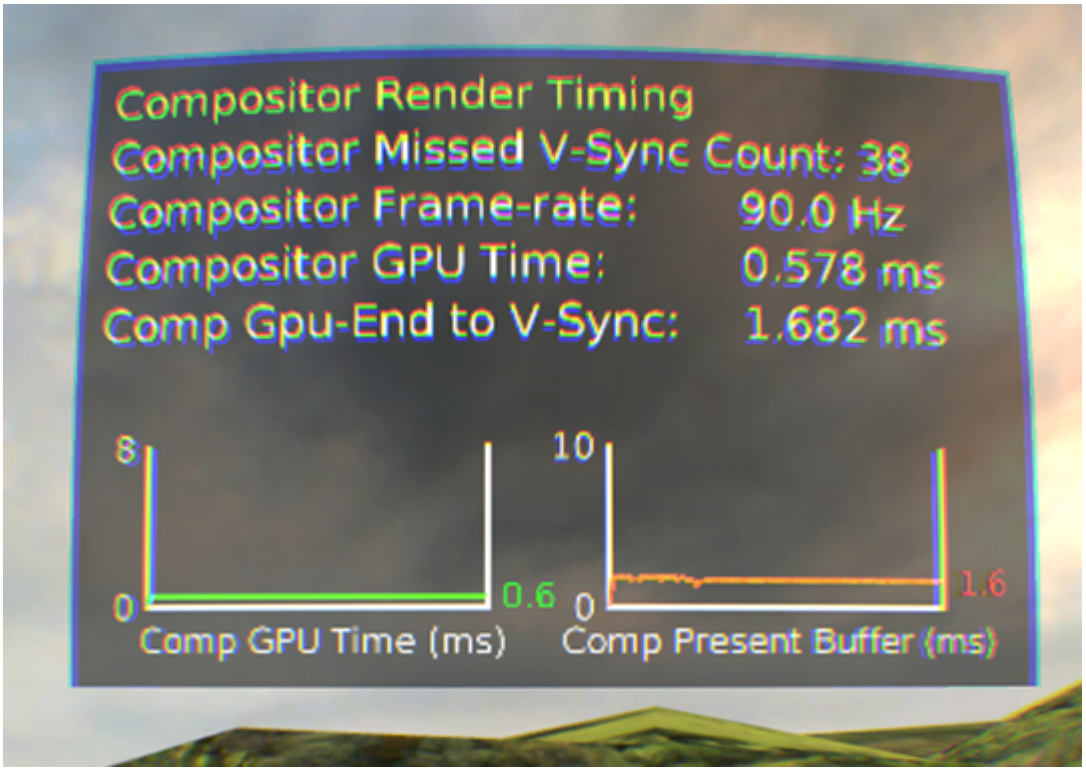
Metric	Description
App Frame-rate	The rate at which application rendering calls `ovr_EndFrame`. It will never exceed the native refresh rate of the HMD as the call to `ovr_EndFrame` throttles the application's CPU execution as needed.
App Render GPU Time	The total GPU time spent on rendering by the client application. This includes the work done by the application after returning from `ovr_EndFrame`, using the mirror texture if applicable. It can also includes GPU command-buffer "bubbles" if the application's CPU thread doesn't push data to the GPU fast enough to keep it occupied. Similarly, if the app pushes the GPU close to full-utilization, the work on next frame (N+1) might be preempted by the compositor's render work on the current frame (N). Because of how the application GPU timing query operates, this can lead to artificially inflated application GPU times as they will start to include the compositor GPU usage times.
App Render CPU Time	The time difference from when the application continued execution on CPU after `ovr_EndFrame` returned the subsequent call to `ovr_EndFrame`. This will show "N/A" if the latency tester is not functioning as expected (e.g., HMD display is sleeping due to prolonged inactivity). This includes the IPC call overhead to the compositor after `ovr_EndFrame` is called by the client application.
App Queue Ahead Time	To improve CPU and GPU parallelism and increase the amount of time that the GPU has to process a frame, the SDK automatically applies queue ahead up to 1 frame. This value displays the amount of queue ahead time being applied (in milliseconds). For more information about adaptive queue ahead, see *Adaptive Queue Ahead* on page 33.

Compositor Render Timing

The Compositor Render Timing HUD displays render timing information for the Oculus Runtime Compositor. The Oculus Compositor applies distortion and TimeWarp to the layered eye textures provided by the VR application.

The following screenshot shows the Compositor Render Timing HUD:

Figure 19: Render Timing



The following table describes each metric:

Table 7: Compositor Render Timing HUD

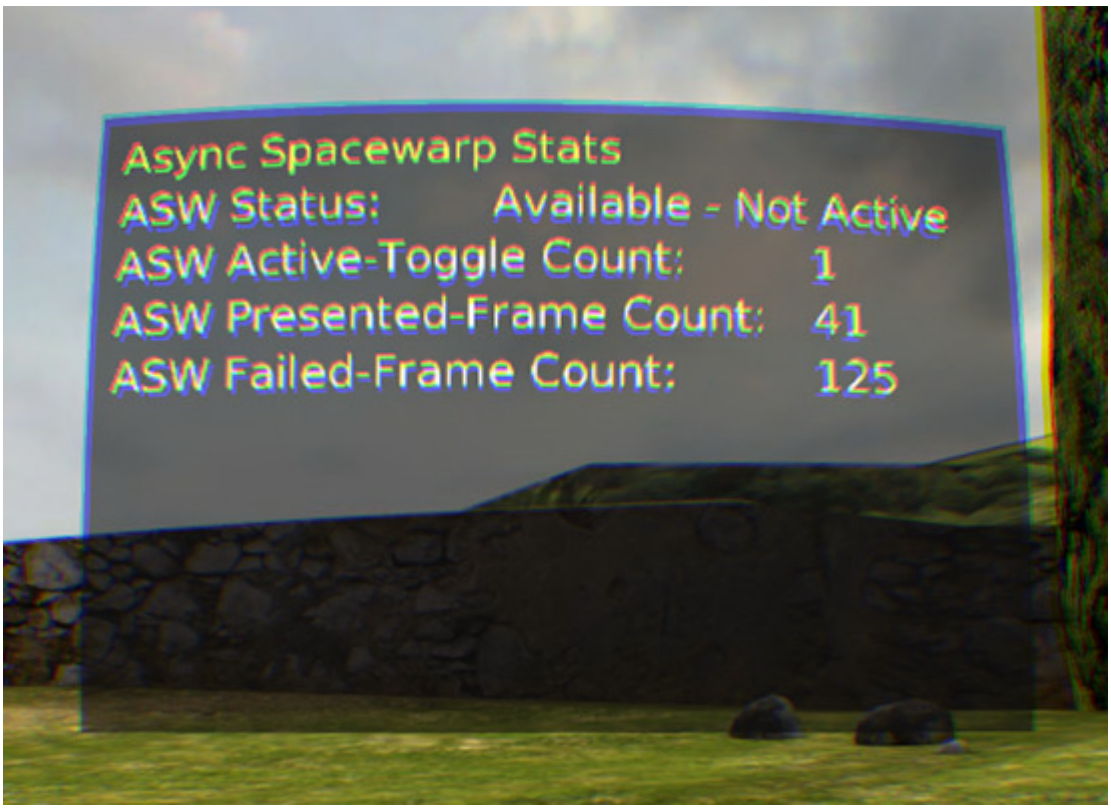
Metric	Description
Compositor Missed V-Sync Count	Increments each time the compositor fails to present a new rendered frame at V-Sync (Vertical Synchronization).
Compositor Frame-rate	The rate of the final composition; this is independent of the client application rendering rate. Because the compositor is always locked to V-Sync, this value will never exceed the native HMD refresh rate. But, if the compositor fails to finish new frames on time, it can drop below the native refresh rate.
Compositor GPU Time	The amount of time the GPU spends executing the compositor renderer. This includes TimeWarp and distortion of all layers submitted by the application. The amount of active layers, their resolutions, and the requested sampling quality can all affect the GPU times.
Comp Gpu-End to V-Sync	The amount of time between when the GPU completes the compositor rendering to the point in time when V-Sync is hit and that buffer starts scanning out on the HMD.

Asynchronous SpaceWarp Stats

The Asynchronous SpaceWarp (ASW) HUD displays activity and tracking statistics for ASW, which runs as part of the Oculus Runtime Compositor. ASW automatically activates when an application fails to meet the required native frame rate for the connected HMD. Once active, ASW forces the application to run at half the normal frame rate while extrapolating every other frame. This gives the application more time to complete its work.

The following screenshot shows the ASW HUD:

Figure 20: ASW Stats



The following table describes each metric:

Table 8: ASW Stats HUD

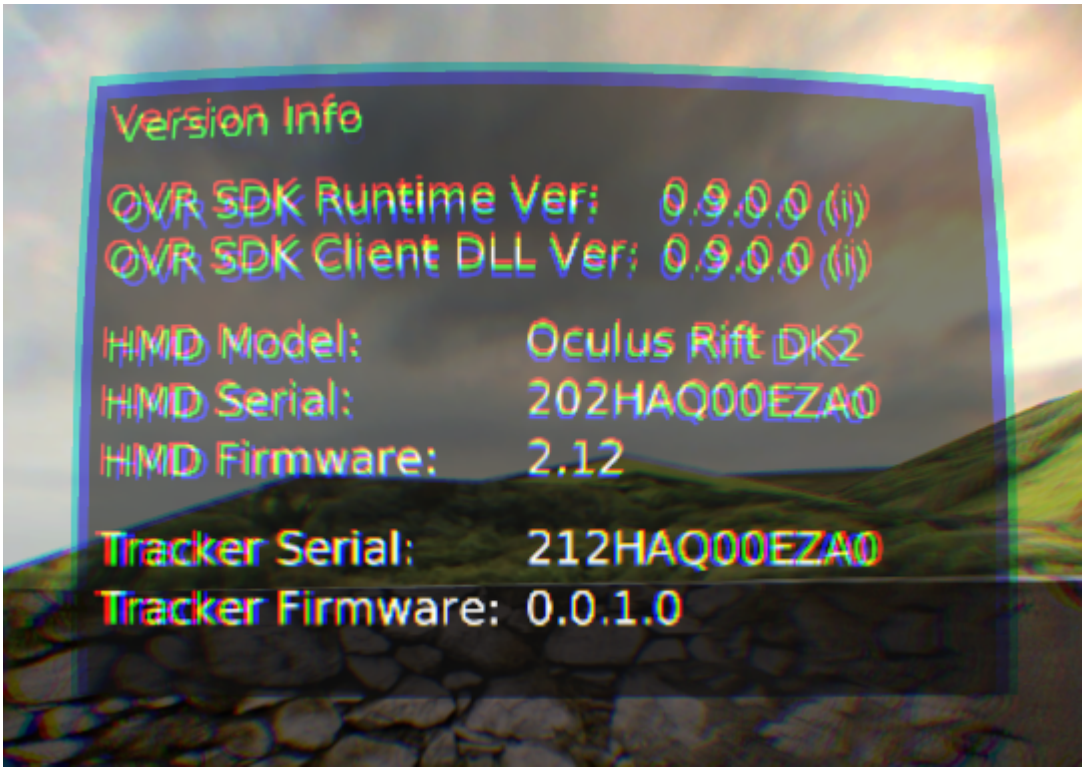
Metric	Description
ASW Status	Shows the availability and current status of ASW. "Not Available" can be due to the OS and/or GPU type used on the PC. "Available - Not Active" will mean the application is successfully maintaining the required native refresh rate, so ASW is not activated.
ASW Active-Toggle Count	Tracks the number of times ASW has been activated for the lifetime of the HMD.
ASW Presented-Frame Count	Tracks the number of frames extrapolated by ASW that were displayed. When ASW is active, since the app is forced to run at half-rate, expect this value to increase by 45 fps on a 90 Hz refresh rate HMD.
ASW Failed-Frame Count	Tracks the number of extrapolated frames ASW needed to display, but failed to prepare in time. This can occur for different reasons, but might be caused by contention for OS resources or when the capabilities of the system are exceeded.

Version Information

The Version Information HUD displays information about the HMD and the version of the SDK used to create the app.

The following screenshot shows the Version Information HUD:

Figure 21: Version Info HUD



The following table describes each piece of information:

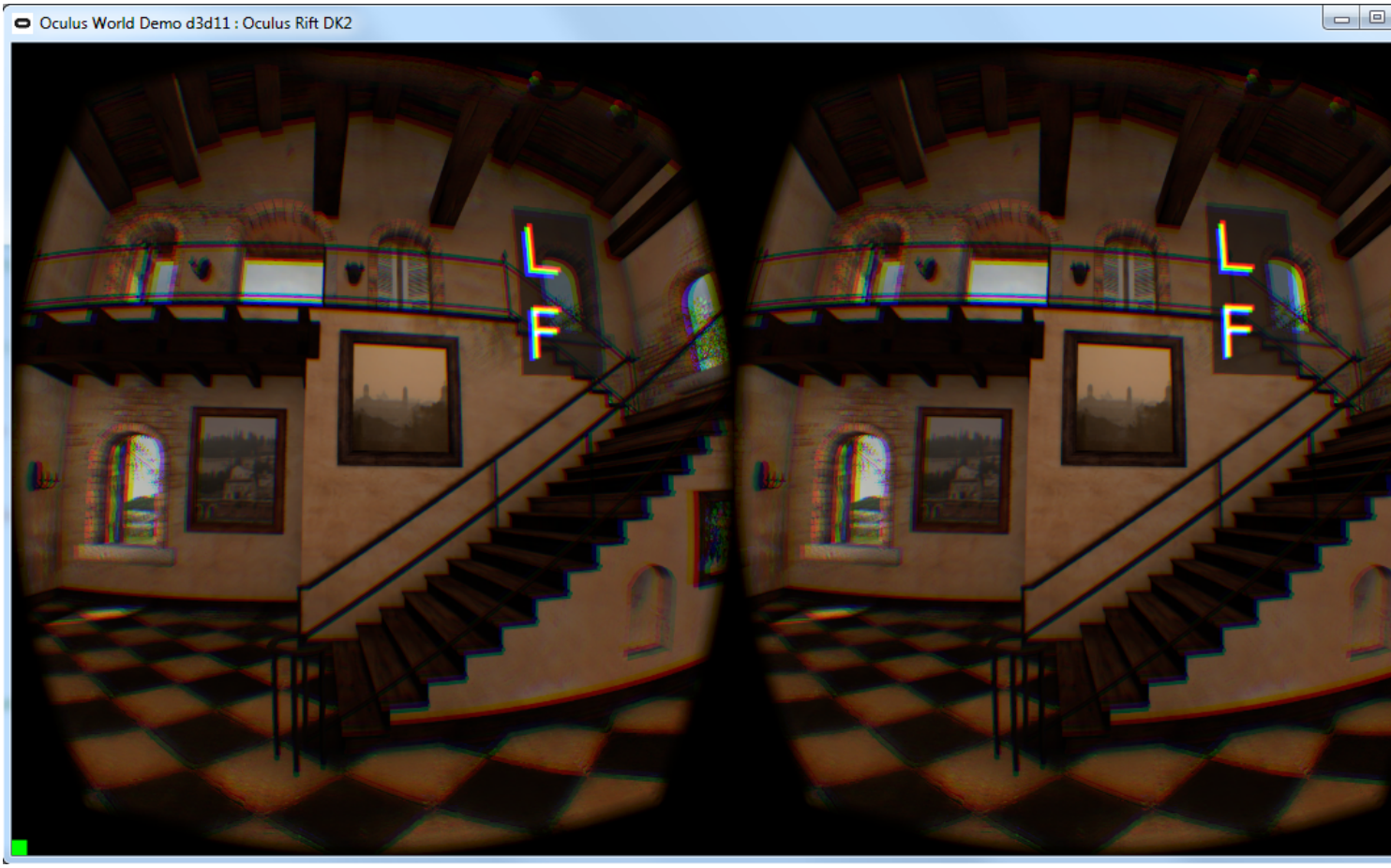
Name	Description
OVR SDK Runtime Ver	Version of the currently installed runtime. Every VR application that uses the OVR SDK since 0.5.0 uses this runtime.
OVR SDK Client DLL Ver	The SDK version that the client app was compiled against.
HMD Type	The type of HMD.
HMD Serial	The serial number of the HMD.
HMD Firmware	The version of the installed HMD firmware.
Sensor Serial	The serial number of the positional sensor.
Sensor Firmware	The version of the installed positional sensor firmware.

Performance Indicator

Asynchornous TimeWarp (ATW) can mask latency and judder issues that would normally be apparent. To help you identify when your application or experience isn't performing and to test your game or experience before submitting it, Oculus provides performance indicators.

When enabled, a letter code appears in the upper right of the headset whenever the application is experiencing a performance issue. The following figure shows an example of a performance indicator with L and F displayed:

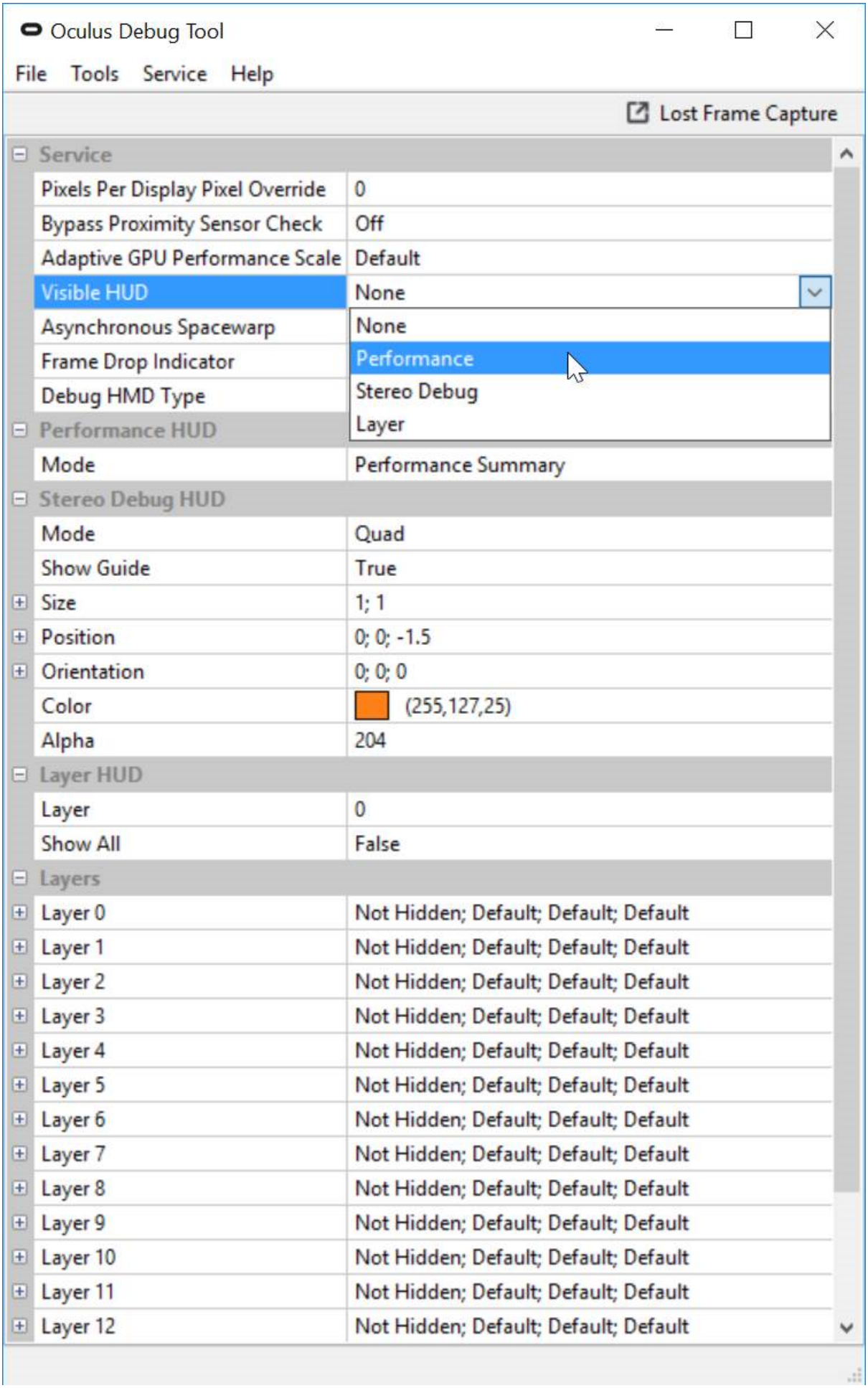
Figure 22: Performance Indicator



To enable the Performance Indicator, start the Oculus Debug Tool, which is located here:

```
C:\Program Files\Oculus\Support\oculus-diagnostics\OculusDebugTool
```

Set the Visible HUD option to Performance:


Oculus Debug Tool
File
Tools
Service
Help
Lost Frame Capture
Service
Pixels Per Display Pixel Override
0
Bypass Proximity Sensor Check
Off
Adaptive GPU Performance Scale
Default
Visible HUD
None
Asynchronous Spacewarp
Frame Drop Indicator
Debug HMD Type
None
Performance
Stereo Debug
Layer
Performance HUD
Mode
Performance Summary
Stereo Debug HUD
Mode
Quad
Show Guide
True
Size
1; 1
Position
0; 0; -1.5
Orientation
0; 0; 0
Color
(255,127,25)
Alpha
204
Layer HUD
Layer
0
Show All
False
Layers
Layer 0
Not Hidden; Default; Default; Default
Layer 1
Not Hidden; Default; Default; Default
Layer 2
Not Hidden; Default; Default; Default
Layer 3
Not Hidden; Default; Default; Default
Layer 4
Not Hidden; Default; Default; Default
Layer 5
Not Hidden; Default; Default; Default
Layer 6
Not Hidden; Default; Default; Default
Layer 7
Not Hidden; Default; Default; Default
Layer 8
Not Hidden; Default; Default; Default
Layer 9
Not Hidden; Default; Default; Default
Layer 10
Not Hidden; Default; Default; Default
Layer 11
Not Hidden; Default; Default; Default
Layer 12
Not Hidden; Default; Default; Default

The performance indicator can return the following codes:

- L - A latency issue is occurring; more than one frame of queue ahead is being applied.
- F - The application is not maintaining frame rate.
- C - The compositor is not maintaining frame rate. Possible causes include:
  - Programs, such as anti-virus software, are overloading the CPU.
  - The CPU cannot handle the number of threads.
  - The CPU or GPU does not meet the recommended specification.
  - Certain patterns of GPU usage, such as heavy usage of shaders and tessellation, are affecting frame rate.
  - There is an issue with the GPU driver.
  - There is an unknown hardware issue.
- U - An unknown error occurred.

Each warning lasts one frame. So, if L stays visible, the application is having continuous latency issues.

Compositor Mirror

The Compositor Mirror tool displays the content that appears within the Rift headset on your computer monitor. It has several display options that are useful for development, troubleshooting, and presentations.

Anything that appears in the Rift headset can be shown in the Compositor Mirror tool, including Oculus Home, Guardian System boundaries, in-game notifications, and transition fades. It is compatible with all games and experiences regardless whether they are developed using the native PC SDK or a game engine.

Finding the Tool

The Compositor Mirror tool is located in C:\Program Files\Oculus\Support\oculus-diagnostics\OculusMirror.exe

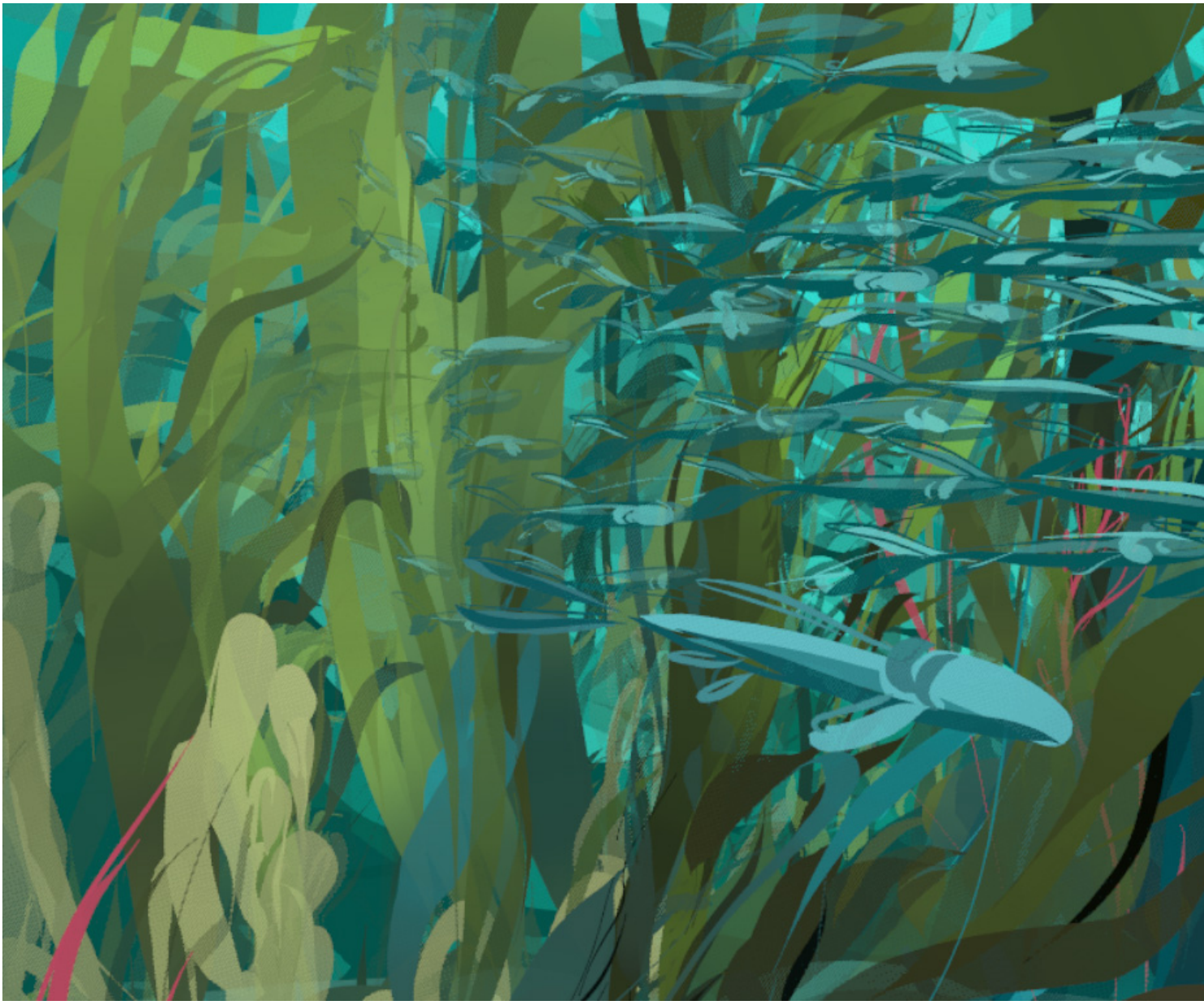
Image Display Options

The Compositor Mirror tool provides several display options.

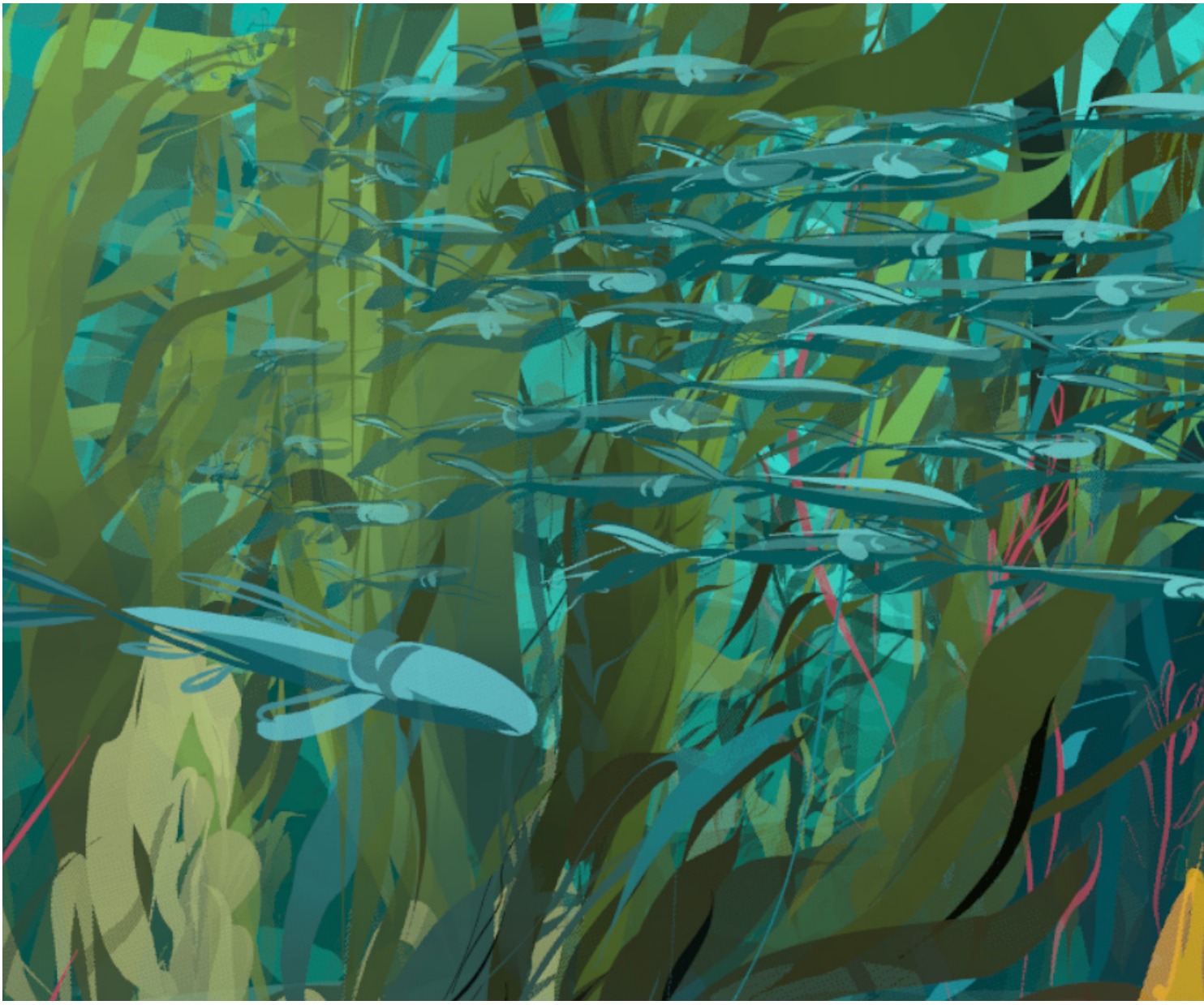
The default mode is recommended for live presentations and demos. If you double-click OculusMirror from Windows Explorer or run from the command line without specifying any options, the tool displays a 1366x768 (pixel) window showing a rectilinear view of the right eye image, along with the Guardian System boundary layer and the Oculus notification layer.

The other display modes are as follows:

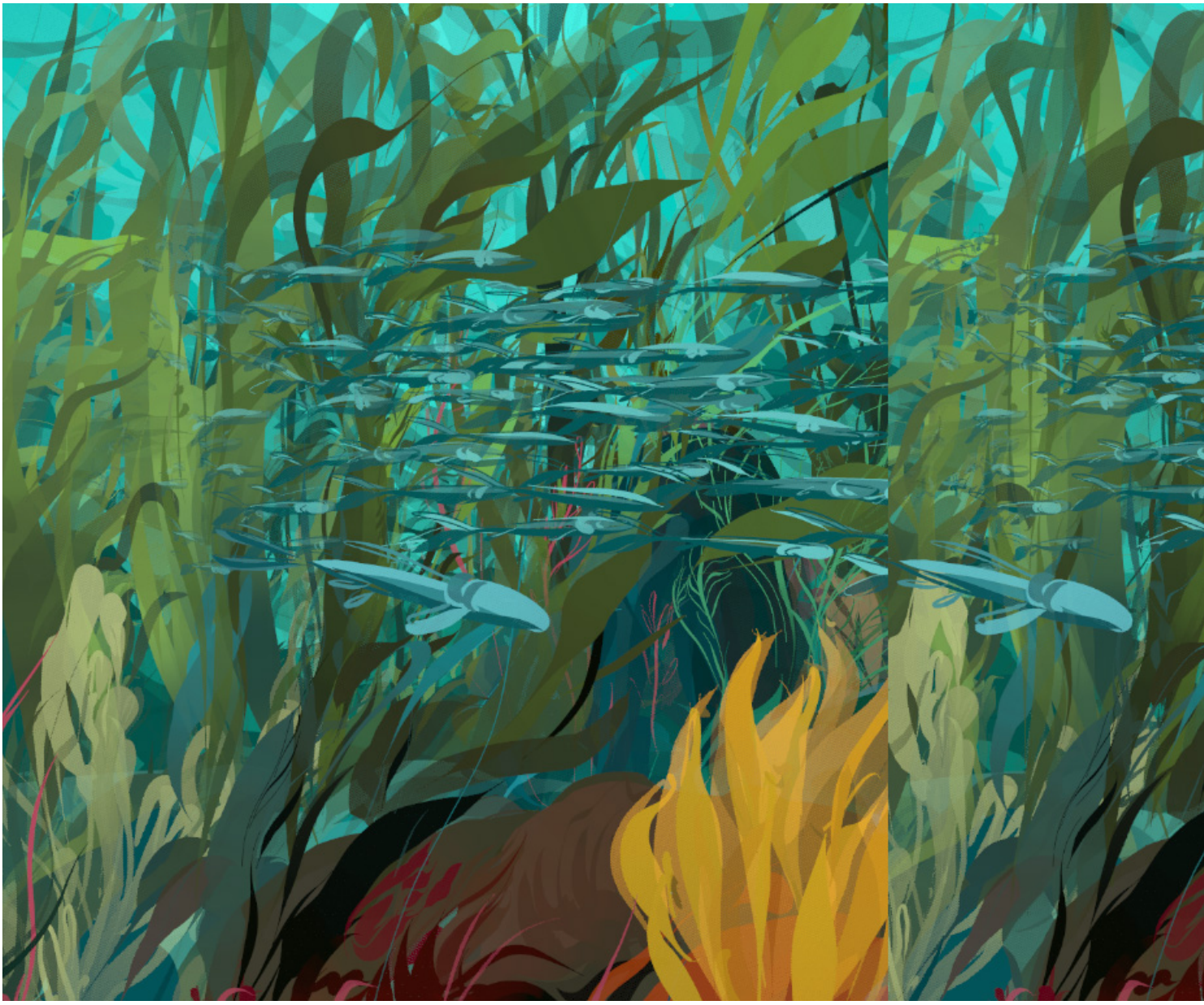
`OculusMirror.exe --LeftEyeOnly` Display the left eye image in rectilinear view:



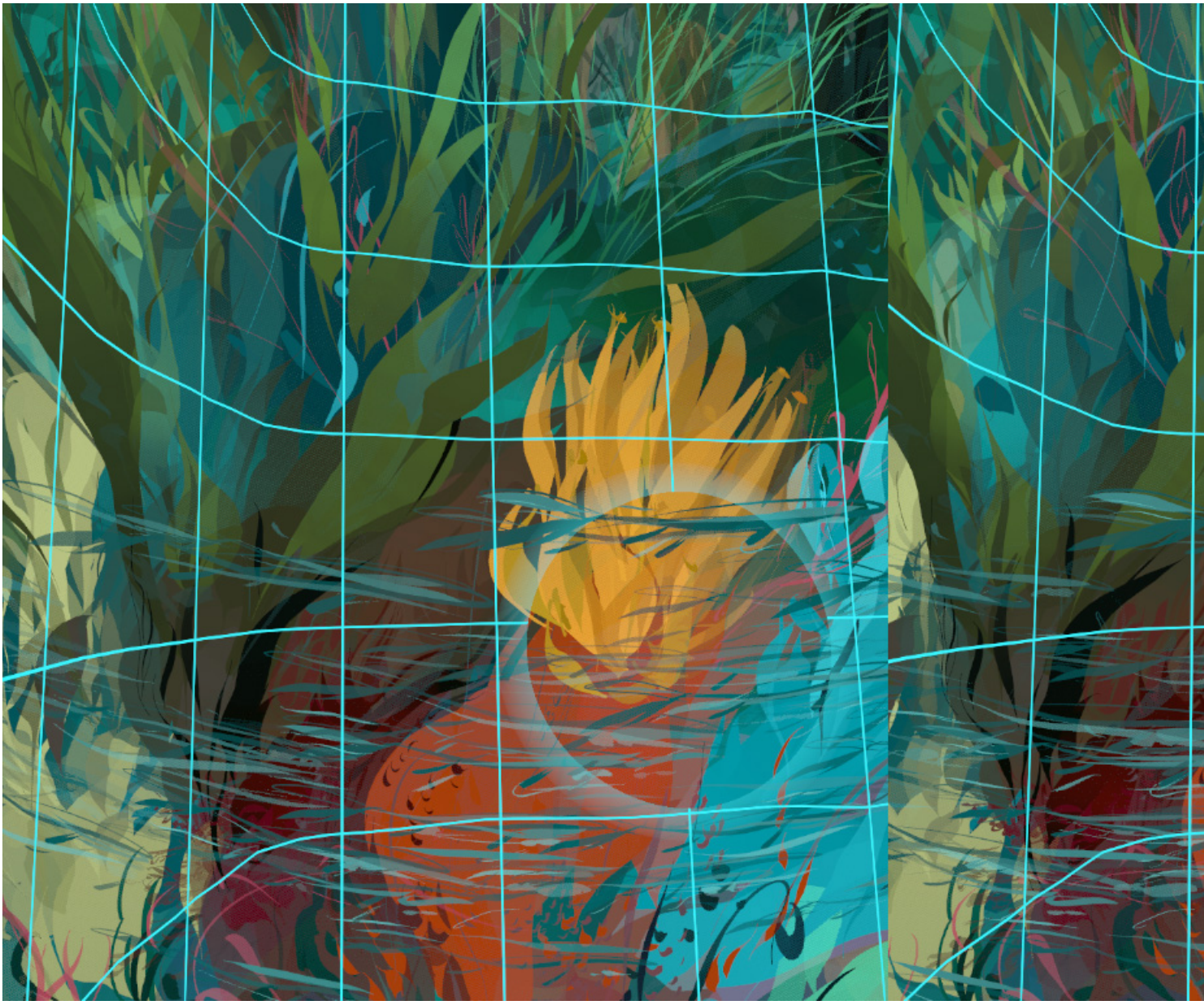
`OculusMirror.exe --RightEyeOnly` Display the right eye image in rectilinear view:



`OculusMirror.exe --RectilinearBothEyes` Display both eye images in rectilinear view:

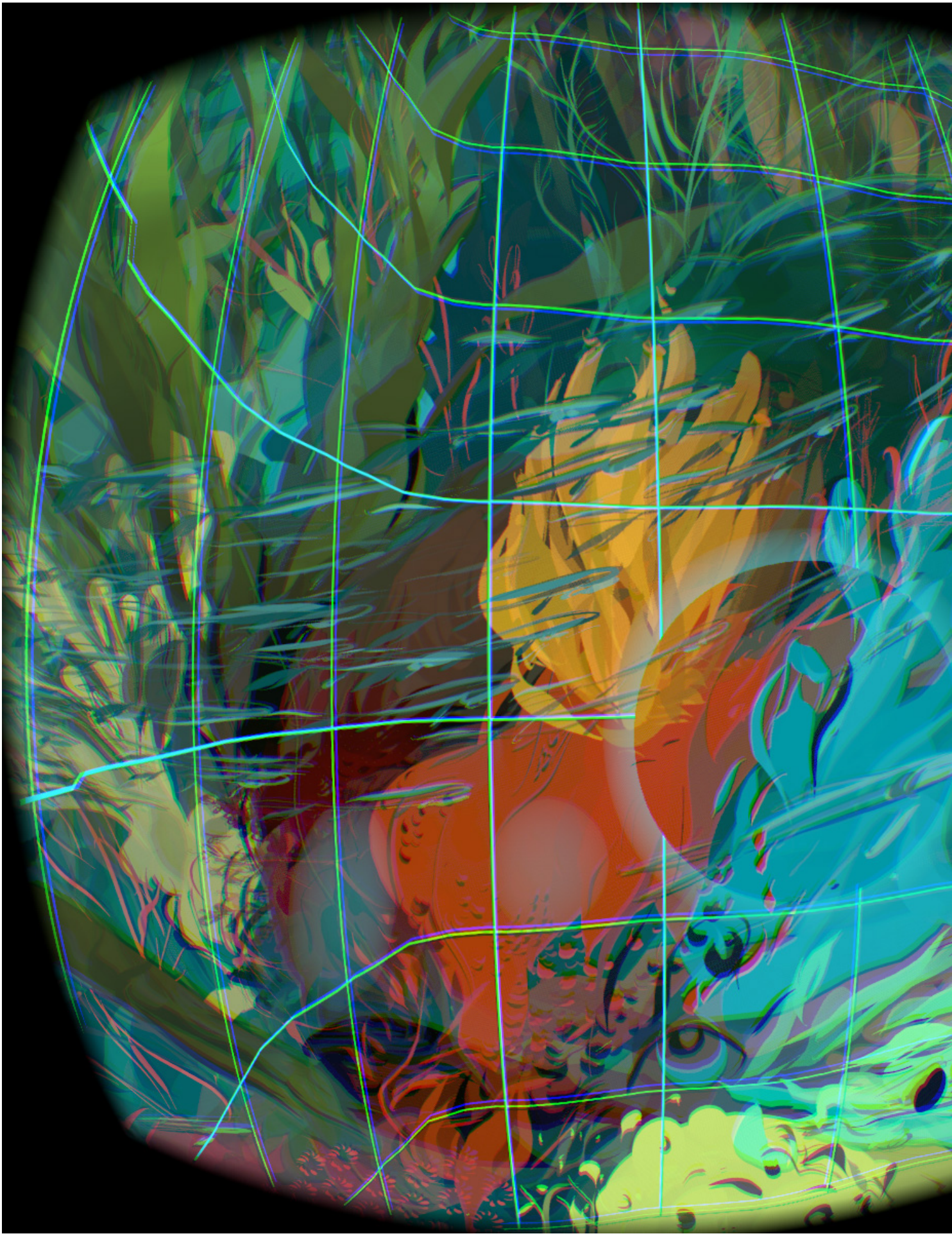


`OculusMirror.exe --RectilinearBothEyes --IncludeGuardian` Display both eye images in rectilinear view along with the Guardian System boundary. In the following example, the user's hands have protruded through the Guardian System boundary in two locations, as indicated by the circular holes in the boundary:



`OculusMirror.exe --PostDistortion`

Display both eye images corrected for lens distortion and chromatic aberration. This will also display any other options that appear within the headset. For example, the following output is produced with --PostDistortion, but also shows the Guardian system boundary, since it is visible within the headset. No additional options are allowed in conjunction with --PostDistortion. For example, you cannot explicitly exclude notifications or the Guardian system boundary, if they are visible within the headset.



Changing the Window Size

You can change the size of the window by dragging the window borders on the desktop. You can also use the `--Size width height` command to set the size of the window (in pixels) from the command line. If you exceed the resolution of your main display, the window size is reduced to fit. For example:

```
OculusMirror.exe --Size 2160 2160 --RightEyeOnly
```

Displaying Notifications

You can display the Oculus notifications layer by using `--IncludeNotifications`. For example:

```
OculusMirror.exe --RectilinearBothEyes --IncludeNotifications
```

Flash on Frame Drops

It is possible that the CompositorMirror tool may drop frames that are displayed within the Rift headset. If you wish to clearly see when this happens, use the `--FlashFrameDrops` command. This will cause the display in the CompositorMirror to flash whenever a frame is dropped. For example:

```
OculusMirror.exe --FlashFrameDrops --RectilinearBothEyes
```

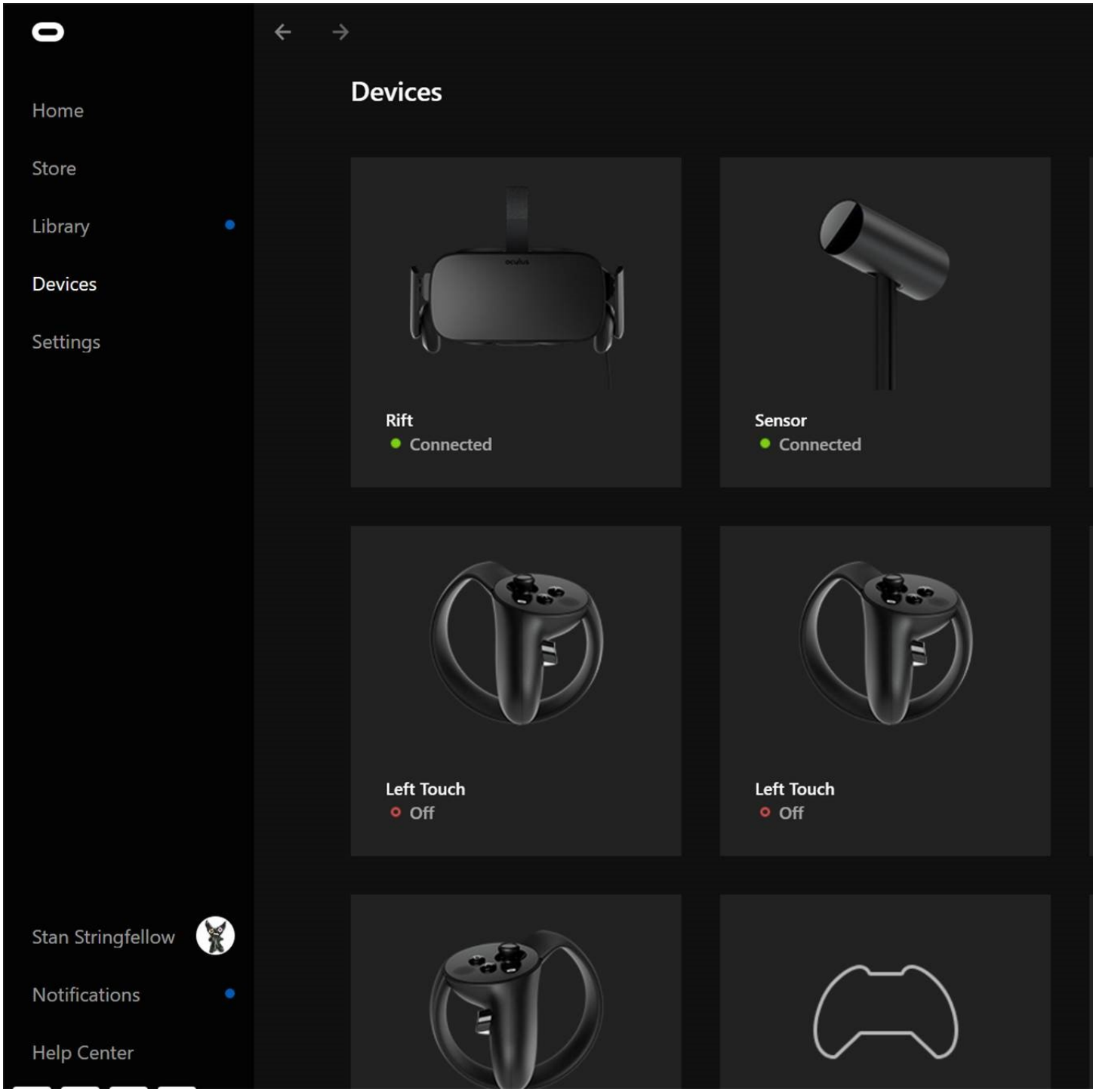
Pairing the Oculus Touch Controllers

After you receive your Touch Controllers, you need to pair them with the headset.

You need to pair the Touch controllers in order to establish a connection between the controllers and the headset. To pair your Touch controllers, follow these steps:

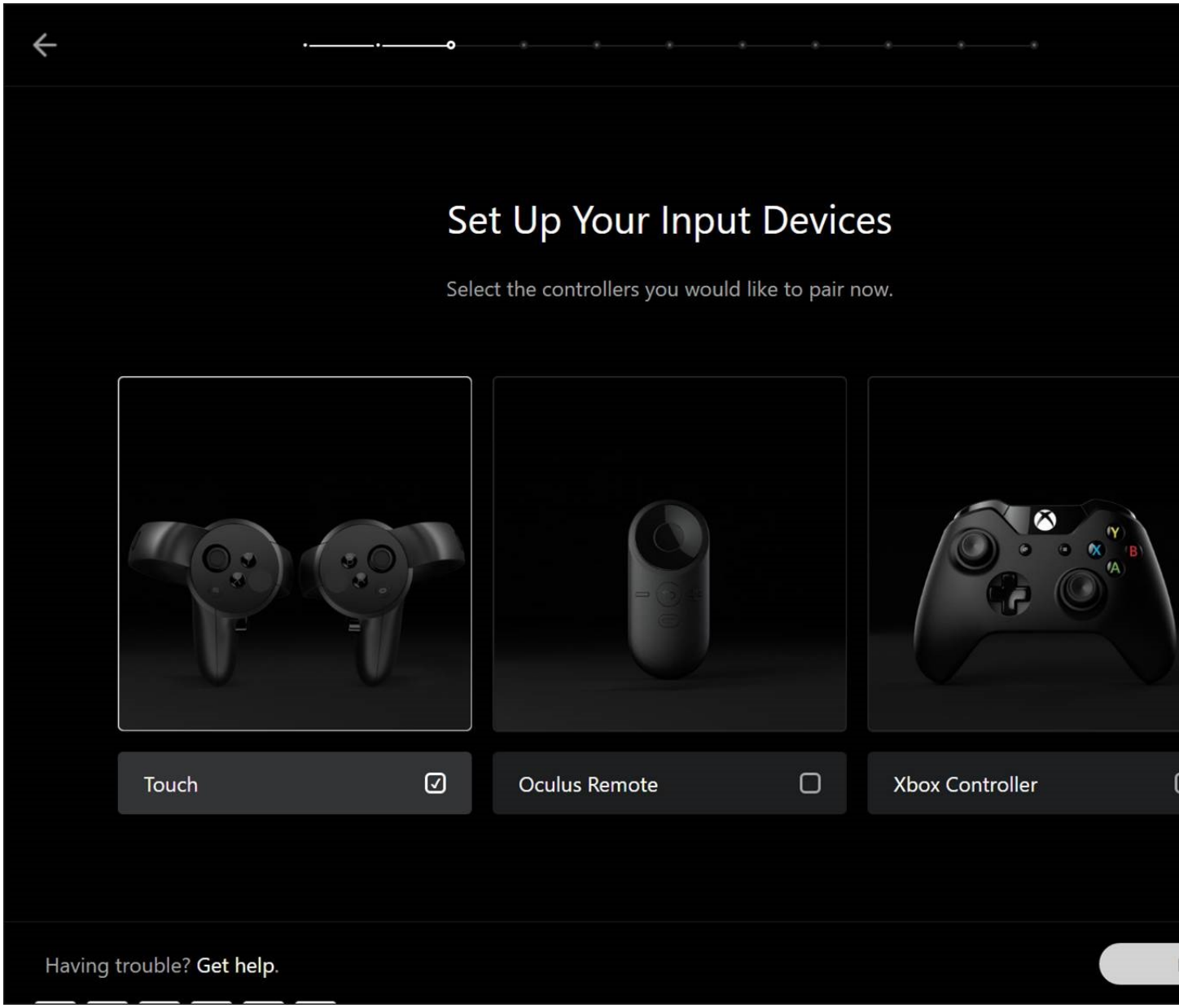
1. Make sure your headset is connected.
2. Launch the Oculus app.

3. In the left panel, select Devices, and then select Configure Rift > Full Setup.



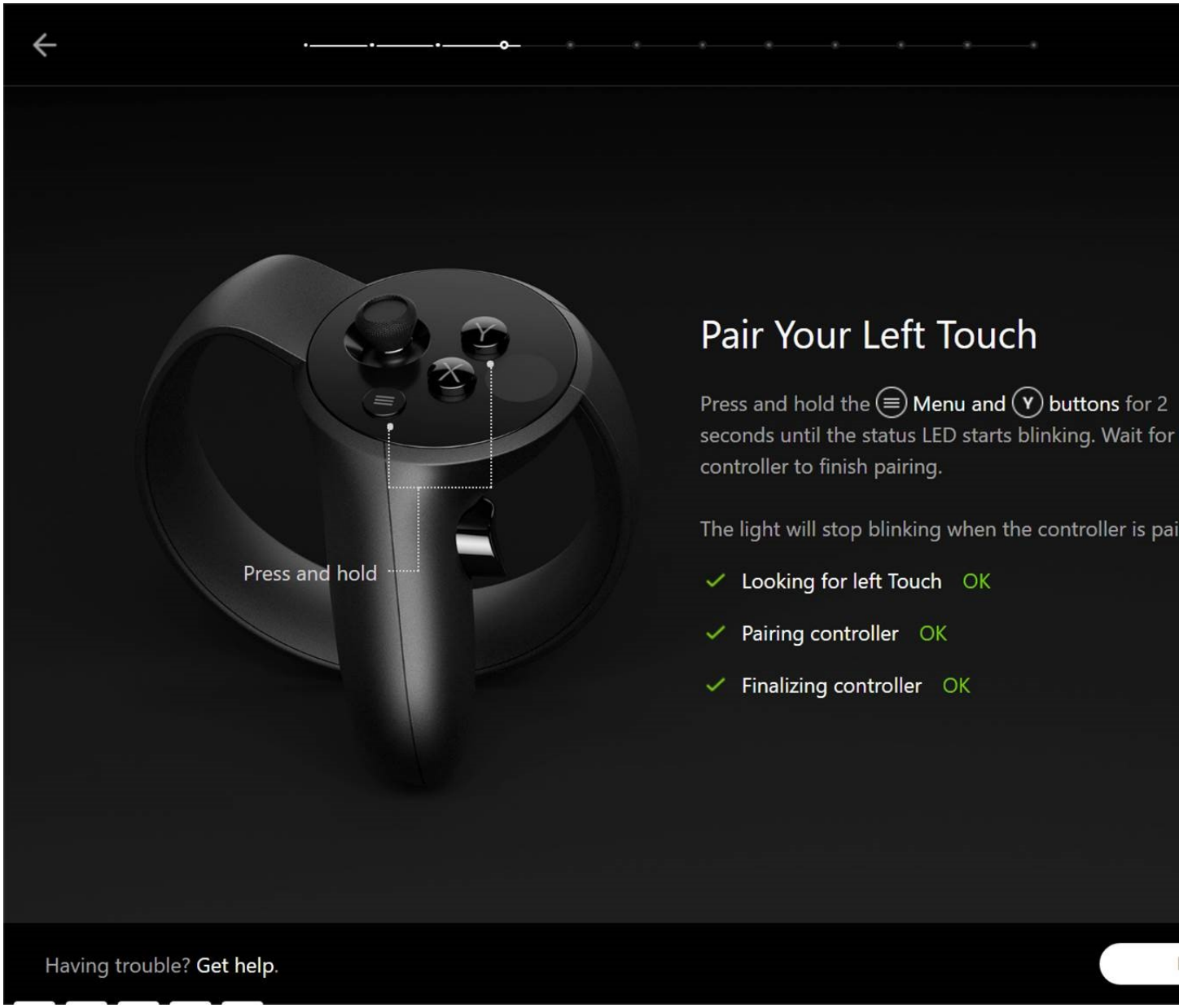
4. Click through the informational screens that follow.

5. When the Set Up Your Input Devices screen appears, select Touch, and click Next.



6. After another informational screen, the Pair Your Left Touch screen appears. Hold down the Menu and Y buttons, as indicated on the screen. After a

few moments, the Left Touch controller should be successfully paired:



7. Continue and follow the same procedure for the Right Touch controller.

Asynchronous SpaceWarp

Asynchronous Spacewarp (ASW) is a frame-rate smoothing technique that almost halves the CPU/GPU time required to produce nearly the same output from the same content.

Overview

ASW applies animation detection, camera translation, and head translation to previous frames in order to predict the next frame. As a result, motion is smoothed and applications can run on lower performance hardware. For a great introduction to ASW, please see *Asynchronous Spacewarp*.

The Rift operates at 90Hz. With ASW, when an application fails to submit frames at 90Hz, the Rift runtime drops the application down to 45Hz with ASW providing each intermediate frame.

By default, ASW is enabled for all supported Rift versions.

ASW tends to predict linear motion better than non-linear motion. If your application is dropping frames, you can either adjust the resolution or simply allow ASW to take over.

Requirements

ASW requires the following:

- Oculus Runtime 1.9 or later
- Windows 8 or later
- For NVIDIA, driver 373.06 or later
- For AMD, driver 16.40.2311 or later

Until the minimum specification is released, we recommend the following GPU versions for ASW testing:

Manufacturer	Series	Minimum RAM	Minimum Model
NVIDIA	Pascal	3GB	1060
NVIDIA	Maxwell	4GB	960
AMD	Polaris	4GB	470

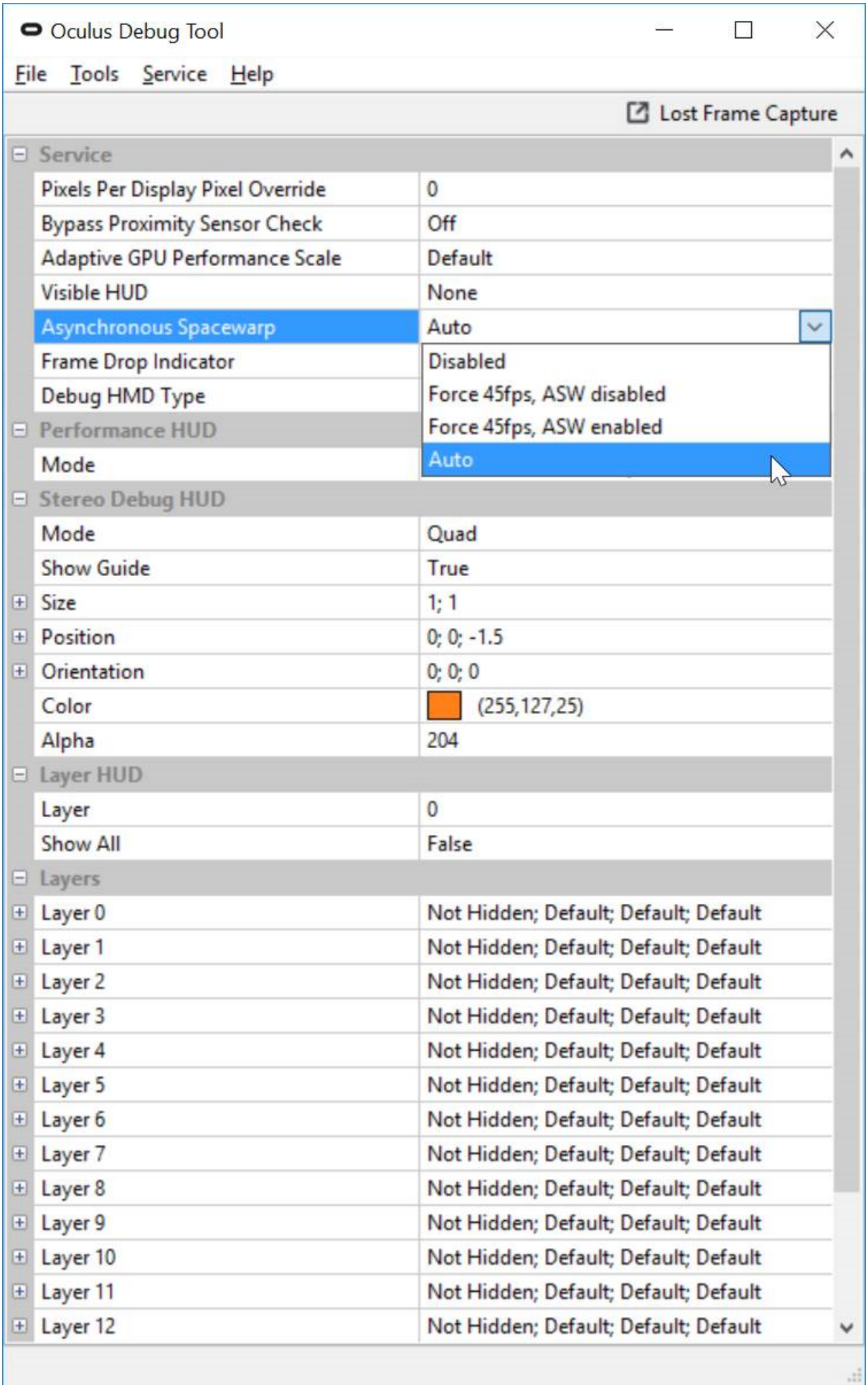
Testing ASW

To test ASW, start the Oculus Debug Tool, which is located here:

```
C:\Program Files\Oculus\Support\oculus-diagnostics\OculusDebugTool
```

Set the Asynchronous Spacewarp option to one of the following values (or run tests using various different options):

- Auto enables ASW, so that it is applied automatically when needed.
- Disabled disables ASW.
- Force 45fps, ASW disabled causes the refresh rate to be 45hz with ASW disabled.
- Force 45fps, ASW enabled causes the refresh rate to be 45hz with ASW enabled


Oculus Debug Tool
File
Tools
Service
Help
Lost Frame Capture
Service
Pixels Per Display Pixel Override
0
Bypass Proximity Sensor Check
Off
Adaptive GPU Performance Scale
Default
Visible HUD
None
Asynchronous Spacewarp
Auto
Frame Drop Indicator
Debug HMD Type
Performance HUD
Mode
Disabled
Force 45fps, ASW disabled
Force 45fps, ASW enabled
Auto
Stereo Debug HUD
Mode
Quad
Show Guide
True
Size
1; 1
Position
0; 0; -1.5
Orientation
0; 0; 0
Color
(255,127,25)
Alpha
204
Layer HUD
Layer
0
Show All
False
Layers
Layer 0
Not Hidden; Default; Default; Default
Layer 1
Not Hidden; Default; Default; Default
Layer 2
Not Hidden; Default; Default; Default
Layer 3
Not Hidden; Default; Default; Default
Layer 4
Not Hidden; Default; Default; Default
Layer 5
Not Hidden; Default; Default; Default
Layer 6
Not Hidden; Default; Default; Default
Layer 7
Not Hidden; Default; Default; Default
Layer 8
Not Hidden; Default; Default; Default
Layer 9
Not Hidden; Default; Default; Default
Layer 10
Not Hidden; Default; Default; Default
Layer 11
Not Hidden; Default; Default; Default
Layer 12
Not Hidden; Default; Default; Default

Mixed Reality Capture

Mixed reality capture places real-world people and objects in VR. This guide will review how to add support for mixed reality capture in your native Rift app.

Retrieving the Camera Details

When running a mixed reality capture supported app there are a couple of inputs that your app need to use. Prior to launching your app, each user will run the CameraTool introduced on the *Camera Calibration* page in the user guide. This will load details about the camera into the system that your app can retrieve. Users won't have to perform the calibration each time, as they will be able to use this tool to load a previously saved file to the system.

1. Camera intrinsics are the attributes of the camera. Resolution, field of view, exposure frequency, etc… Details of the camera intrinsics are sent to, and retrieved from, the SDK as part of `OvrCameraIntrisics`. We do not expect the camera intrinsics to change frequently.
2. Camera extrinsics are everything external to the camera. Relative pose, latency to system, attached to (VR Object), etc.. We expect the camera extrinsics to change frequently.

Implementing Mixed Reality Capture

This guide assumes that you have previously implemented the Oculus SDK and have a functioning Rift application.

1. The first step is to define an additional camera perspective, from a 3rd person view, that is capable of being moved around the scene.
2. In your app, pair that perspective with the external camera configured in the camera tool. Call `ovr_GetExternalCameras` to retrieve a list of external cameras.
3. Retrieve the properties of that camera from the pointer provided by the `ovr_GetExternalCameras` response. This gives you the following information.

    a. Intrinsics

    - `LastChangedTime` - Time in seconds from last change to the parameters.
    - `FOVPort` - Angles of all 4 sides of viewport.
    - `VirtualNearPlaneDistanceMeters` - Distance, in virtual meters, of the near plane clipping distance. Your app will determine this distance.
    - `VirtualFarPlaneDistanceMeters` - Distance, in virtual meters, of the far plane clipping distance. Your app will determine this distance.
    - `ImageSensorPixelResolution` - Height and width, in pixels, of image sensor.
    - `LensDistortionMatrix` - The lens distortion of the camera.
    - `ExposurePeriodSeconds` - How frequently, in seconds, the exposure is taken. This value is not provided by the CameraTool. You should request this information in your app from the user.
    - `ExposureDurationSeconds` - Length of the exposure time. This value is not provided by the CameraTool. You should request this information in your app from the user.

    b. Extrinsics

    - `CameraStatusFlags` - Current Status of the camera, a mix of bits from `ovrCameraStatusFlags`.
    - `AttachedToDevice` - Which Tracked device, if any, is the camera rigidly attached to. If set to `ovrTrackedDevice_None`, then the camera is not attached to a tracked object. If the external camera moves while unattached (i.e. set to `ovrTrackedDevice_None`), its Pose won't be updated.
    - `RelativePose` - The relative Pose of the External Camera. If `AttachedToDevice` is `ovrTrackedDevice_None`, then this is a absolute pose in tracking space.

- `LastExposureTimeSeconds` - The time, in seconds, when the last successful exposure was taken.
- `ExposureLatencySeconds` - Estimated exposure latency to get from the exposure time to the system.
- `AdditionalLatencySeconds` - Additional latency to get from the exposure time of the real camera to match the render time of the virtual camera.

4. Now you have the external camera with the initial pose, the tracked VR Object that will give you the transform and pose in the scene, and the details about the camera that is capturing the real-world portion of the scene. The external camera has also been associated with the in-app camera.

   The following example from our Oculus World Demo demonstrates the process of retrieving an external camera and using the camera intrinsics to set the window/mirror size.

```
bool OculusWorldDemoApp::SetupMixedReality()
{
    ovrResult error = ovr_GetExternalCameras(&ExternalCameras[0], &NumberOfCameras);

    if (!OVR_SUCCESS(error))
    {
        DisplayLastErrorMessageBox("ovr_GetExternalCameras failure.");
        return false;
    }
    CurrentCameraID = 0;
    // We use 0 as the default camera ID. If more than one camera is connected,
    // you'll need to find the camera ID based on the camera name string provided during
 calibrating in the CameraTool.

    Sizei tempWindowSize;
    tempWindowSize.w = WindowSize.w + 2 *
 ExternalCameras[CurrentCameraID].Intrinsics.ImageSensorPixelResolution.w;
    tempWindowSize.h = std::max(WindowSize.h,
 ExternalCameras[CurrentCameraID].Intrinsics.ImageSensorPixelResolution.h);

    RenderParams.Resolution = tempWindowSize;
    if (pRender != nullptr)
    {
        pRender->SetWindowSize(tempWindowSize.w, tempWindowSize.h);
        pRender->SetParams(RenderParams);
    }
    pPlatform->SetWindowSize(tempWindowSize.w, tempWindowSize.h);   // resize the window

    NearRenderViewport = Recti(Vector2i(WindowSize.w, 0),
 ExternalCameras[CurrentCameraID].Intrinsics.ImageSensorPixelResolution);
    FarRenderViewport =
 Recti(Vector2i(ExternalCameras[CurrentCameraID].Intrinsics.ImageSensorPixelResolution.w +
 WindowSize.w, 0),
        ExternalCameras[CurrentCameraID].Intrinsics.ImageSensorPixelResolution);

     return true;
}
```

5. While rendering the scene you'll want to render every frame or match the frame-rate of the external camera. Simultaneously, retrieve the location pose of the camera by calling a) `ovr_GetDevicePoses` which gives you the transform from the original pose. If it's attached to `ovrtrackeddevice_none` then the relative pose is actually absolute.
6. Save the rendered images and transform data with a timestamp to a temporary buffer. You'll use that timestamp to pair the images from rendered scene to the external camera images. Images from the in-app scene will be later retrieved from the temporary buffer after the system latency, gathered from the external camera extrinsics, has passed.

   The following example from Oculus World Demo App demonstrates rendering the MR scene.

```
void OculusWorldDemoApp::RenderCamNearFarView()
{
    Posef tempPose;
    tempPose.SetIdentity();     // fixed camera pose
    if (ExternalCameras[CurrentCameraID].Extrinsics.AttachedToDevice ==  ovrTrackedDevice_LTouch)
        tempPose = HandPoses[0];
    else if (ExternalCameras[CurrentCameraID].Extrinsics.AttachedToDevice ==
 ovrTrackedDevice_LTouch)
```

```
        tempPose = HandPoses[1];
    else if (ExternalCameras[CurrentCameraID].Extrinsics.AttachedToDevice ==
ovrTrackedDevice_Object0)
        tempPose = TrackedObjectPose;

    Posef CamPose = tempPose * Posef(ExternalCameras[CurrentCameraID].Extrinsics.RelativePose);

    Posef CamPosePlayer = ThePlayer.VirtualWorldTransformfromRealPose(CamPose,
TrackingOriginType);
    Vector3f up = CamPosePlayer.Rotation.Rotate(UpVector);
    Vector3f forward = CamPosePlayer.Rotation.Rotate(ForwardVector);
    Vector3f dif = ThePlayer.GetHeadPosition(TrackingOriginType) - CamPosePlayer.Translation;
    float bodyDistance = forward.Dot(dif);

    bool flipZ = DepthModifier != NearLessThanFar;
    bool farAtInfinity = DepthModifier == FarLessThanNearAndInfiniteFarClip;
    unsigned int projectionModifier = ovrProjection_None;
    projectionModifier |= (RenderParams.RenderAPI == RenderAPI_OpenGL) ?
ovrProjection_ClipRangeOpenGL : 0;
    projectionModifier |= flipZ ? ovrProjection_FarLessThanNear : 0;
    projectionModifier |= farAtInfinity ? ovrProjection_FarClipAtInfinity : 0;


    ViewFromWorld[2] = Matrix4f::LookAtRH(CamPosePlayer.Translation, CamPosePlayer.Translation +
forward, up);

    // near view
    CamProjection = ovrMatrix4f_Projection(ExternalCameras[CurrentCameraID].Intrinsics.FOVPort,
        ExternalCameras[CurrentCameraID].Intrinsics.VirtualNearPlaneDistanceMeters,
        bodyDistance,
        projectionModifier);
    pRender->ApplyStereoParams(NearRenderViewport, CamProjection);
    pRender->SetDepthMode(true, true, (DepthModifier == NearLessThanFar ?
        RenderDevice::Compare_Less :
        RenderDevice::Compare_Greater));

    if ((GridDisplayMode != GridDisplay_GridOnly) && (GridDisplayMode != GridDisplay_GridDirect))
    {
        if (SceneMode != Scene_OculusCubes && SceneMode != Scene_DistortTune)
        {
            MainScene.Render(pRender, ViewFromWorld[2]);
            RenderControllers(ovrEye_Count);   // 2 : from the external camera
        }
    }

    // far view
    CamProjection = ovrMatrix4f_Projection(ExternalCameras[CurrentCameraID].Intrinsics.FOVPort,
        bodyDistance,
        ExternalCameras[CurrentCameraID].Intrinsics.VirtualFarPlaneDistanceMeters,
        projectionModifier);
    pRender->ApplyStereoParams(FarRenderViewport, CamProjection);
    pRender->SetDepthMode(true, true, (DepthModifier == NearLessThanFar ?
        RenderDevice::Compare_Less :
        RenderDevice::Compare_Greater));

    if ((GridDisplayMode != GridDisplay_GridOnly) && (GridDisplayMode != GridDisplay_GridDirect))
    {
        if (SceneMode != Scene_OculusCubes && SceneMode != Scene_DistortTune)
        {
            MainScene.Render(pRender, ViewFromWorld[2]);
            RenderControllers(ovrEye_Count);   // 2 : from the external camera
        }
    }
}
```

7. The final step is to combine the images. This process will vary depending on how your app is going to handle composition of the final scene.

    a. Direct Composition - If you're handling the composition in your app, your app will need to be able to capture the external camera images, apply the chroma key to clip the greenscreen, and match the rendered scene with the clipped external camera image (to match take the timestamp of the rendered scene in the buffer, add the `AdditionalLatencySeconds` retrieved from the camera extrinsics, and apply to the external camera image that most closely matched the calculated timestamp).

    b. External Composition - If you're not doing your own composition and are planning to have users run an external program, you'll want to delay passing the rendered scene, from the buffer, by the

`AdditionalLatencySeconds` so the images are properly aligned. The external program will handle the capture of the external camera, greenscreen clipping, and producing the final MR scene. Details about this process can be found in the *Mixed Reality Capture Setup Guide*.

All sample code provided on this page is covered under the Oculus Examples License.

VRC Validator

The Virtual Reality Check (VRC) Validator utility runs automated tests to determine if your Rift app is ready for Oculus Store technical review. The VRC Validator can reveal shortcomings that need to be addressed before your app can pass the Oculus Store review process. The VRC Validator has a commandline interface as well as a GUI interface.

Running All Default Tests

VRC Validator is included in the Oculus Runtime. You do not need to wear the Rift during the tests.

To run the VRC Validator:

1. Start a Windows Command Prompt window as an administrator.
2. Enter: `cd "C:\Program Files\Oculus\Support\oculus-diagnostics\"`
3. Run `OculusVRCValidator.exe` with the `--path` parameter set to the executable file of your Rift app. For example:

   ```
   OculusVRCValidator --path "C:\Program Files\Oculus\Software\Software\oculus-first-contact\TouchNUX.exe"
   ```

The tests results are written to the command prompt window. You can also log the test results to a .txt file.

Troubleshooting: If all the tests are failing, make sure you selected Run as Administrator when starting the Windows command prompt window. We require Administrator privileges to activate the HMD when it isn't worn.

Lost events: The VRC Validator will retry the tests if any events were lost while the application was being evaluated.

Logging Test Results to a File

Append the `-l` option to your OculusVRCValidator command line to write the test results to the text file. For example: `C:\Program Files\Oculus\Support\oculus-diagnostics\OculusVrcValidator_Log.txt`

If you want to write the test results to a different file, use `--log_file` *`fullpath`*.

Example:

```
OculusVRCValidator --path "C:\Program Files\Oculus\Software\Software\oculus-oculus-video\Cinema.exe" --log_file "C:\temp\log.txt"
```

Delaying the Start Time of Tests with the --load_time_ms Option

If you need to add a delay before each test begins to navigate to a specific part of your app you wish to test, append the `--load_time_ms` *`duration`* option to your OculusVRCValidator command line.

You can specify the wait duration in milliseconds. This example delays for 30 seconds:

```
OculusVRCValidator --path "C:\Program Files\Oculus\Software\Software\oculus-first-contact\TouchNUX.exe" --load_time_ms 30000
```

To make the tests wait until you press ENTER in the command prompt window, specify a duration of 0. The command prompt window must have focus. For example:

```
OculusVRCValidator --path "C:\Program Files\Oculus\Software\Software\oculus-first-contact\TouchNUX.exe" --load_time_ms 0
```

Running Selected Tests

To retrieve the complete list of tests, run `OculusVRCValidator --list_tests`.

Append the `--test testname` option to your OculusVRCValidator command line to specify that you want to run a particular test. If you want to run several selected tests, specify more than one `--test testname` option.

For example, to run the TestFrameRate and TestAppShouldQuit tests:

```
OculusVRCValidator --path "C:\Program Files\Oculus\Software\Software\oculus-first-contact
\TouchNUX.exe" --test TestFrameRate --test TestAppShouldQuit
```

Tests and Pass Criteria

The default tests of the OculusVRCValidator test your app against specific VRC criteria. If your app fails any of these default tests, it is likely to fail its official Oculus Store technical review.

There are also a number of optional tests. We do not run these tests during Oculus Store technical review, but ensuring that your app can pass them adds quality to your app.

Default Tests (in order they will be tested)

Test Name	Pass Criteria
TestSdkVersion	Your app must use the correct versions of Oculus PC SDK, Unity, or Unreal Engine.
TestEntitlementCheck	Your app must perform an Oculus Platform entitlement check within 10 seconds of launch.
TestOculusDLLIncludes	Your app must not distribute its own copies of Oculus DLLs.
TestLaunchIntoVR	Your app must launch into VR within 4 seconds and display a non-headlocked layer.
TestFrameRate	Your app must maintain 90 frames per second.
TestSubmitFramesWhenVisible	Your app must submit frames when visible.
TestSubmitFramesWhenNotVisible	Your app must stop submitting frames when the Universal Menu is open.
TestResponseToRecenterRequest	Your app must respond to requests to reset the view.
TestAppShouldQuit	Your app must quit gracefully.
CheckForExtraneousFiles	Your app must not contain DLLs from other platforms.
TestAudioOutput	Your app must target the audio device specified in the Oculus app.

Optional Tests (in order they will be tested)

Test Name	Pass Criteria
TestMismatchedAdapters	Your app must run correctly when the HMD is connected to a different GPU than the main display.
TestResponseToDisplayLost	Your app must either quit properly or stop submitting frames when the HMD is unplugged from the GPU.
TestPropertyAccess	Your app must not use any deprecated properties.

Test Name	Pass Criteria
RunErrorCapture	Your app must not generate runtime errors.
TestResponseToIadChanges	Your app must respond to changes in the lens slider.

Individual Test Details

The following are details of the tests as they will be run by the validator.

TestSdkVersion

Tests if your app is built with the supported versions of Oculus PC SDK, Unity, or Unreal Engine.

TestEntitlementCheck

Tests if your app performs an Oculus Platform entitlement check within 10 seconds of launch.

TestOculusDLLIncludes

Tests if your app is distributing copies of the following Oculus DLLs instead of loading the DLLs from the Oculus runtime directory:

- LibOVRRT32_1.dll
- LibOVRRT64_1.dll
- LibOVRPlatform32_1.dll
- LibOVRPlatform64_1.dll
- LibOVRP2P32_1.dll
- LibOVRP2P64_1.dll
- LibOVRAvatar32_1.dll
- LibOVRAvatar64_1.dll

TestLaunchIntoVR

Tests if your app displays a non-headlocked graphic in VR and responds to head tracking within 4 seconds of launch. If you want to test a different duration, append the `--max_time_to_frame` *duration* option to your command line. Specify the duration in milliseconds.

TestFrameRate

Tests if your app displays graphics in the headset at 90 frames per second.

Note: If desired, you may add `--output_fps` to print the fps at 500ms intervals as the test runs.

TestSubmitFramesWhenVisible

Tests if your app renders when it is visible. The test counts the number of texture swap chains committed and reports it at the end of the test.

TestSubmitFramesWhenNotVisible

Tests if your app stops submitting frames when the Universal Menu is open.

TestResponseToRecenterRequest

Tests if your app resets the user's position and orientation when a user selects Reset View in the Universal Menu.

TestAppShouldQuit

Tests if your app quits properly from the Universal Menu with ovr_Destroy when it receives a quit request.

CheckforExtraneousFiles

Tests if your app folder contains extraneous .pdb symbol files.

TestAudioOutput

Tests if your app targets the audio device selected in the "Audio Output in VR" setting in the Oculus app.

TestMismatchedAdapters

Tests if the application supports plugging the HMD and the primary display into different display adapters. To run this test, the system must have at least two separate display adapters, with the HMD and primary display connected to different display adapters.

TestResponseToDisplayLost

Tests if the application responds gracefully if the HMD cable is unplugged from the display adapter. Once the HMD display is lost, the application should either quit with ovr_Destroy, or pause and stop submitting frames to the HMD. Example:

```
OculusVRCValidator --path "C:\Program Files\Oculus\Software\Software\oculus-first-contact\TouchNUX.exe" --test TestResponseToDisplayLost
```

During this test, you will be asked to unplug the HMD from the display adapter, allow the test to run, and then plug the HMD back in.

TestPropertyAccess

Tests if your app calls any internal, deprecated, or otherwise unsupported API functions that are carryovers from DK2 development. Deprecated property functions include:

- User
- Name
- Gender
- PlayerHeight
- EyeHeight
- NeckEyeDistance
- EyeNoseDist

RunErrorCapture

Tests if your app generates runtime errors. Some common errors include:

- ovrError_InvalidParameter - invalid parameter provided. More information is output about the function which is called with invalid parameter.
- ovrError_MismatchedAdapters - occurs when the HMD is not plugged in the primary display adapter and the application is not handling this.
- ovrError_LeakingResources - calling application has leaked resources
- ovrError_TextureSwapChainFull - ovr_CommitTextureSwapChain was called too many times on a texture swapchain without calling submit to use the chain

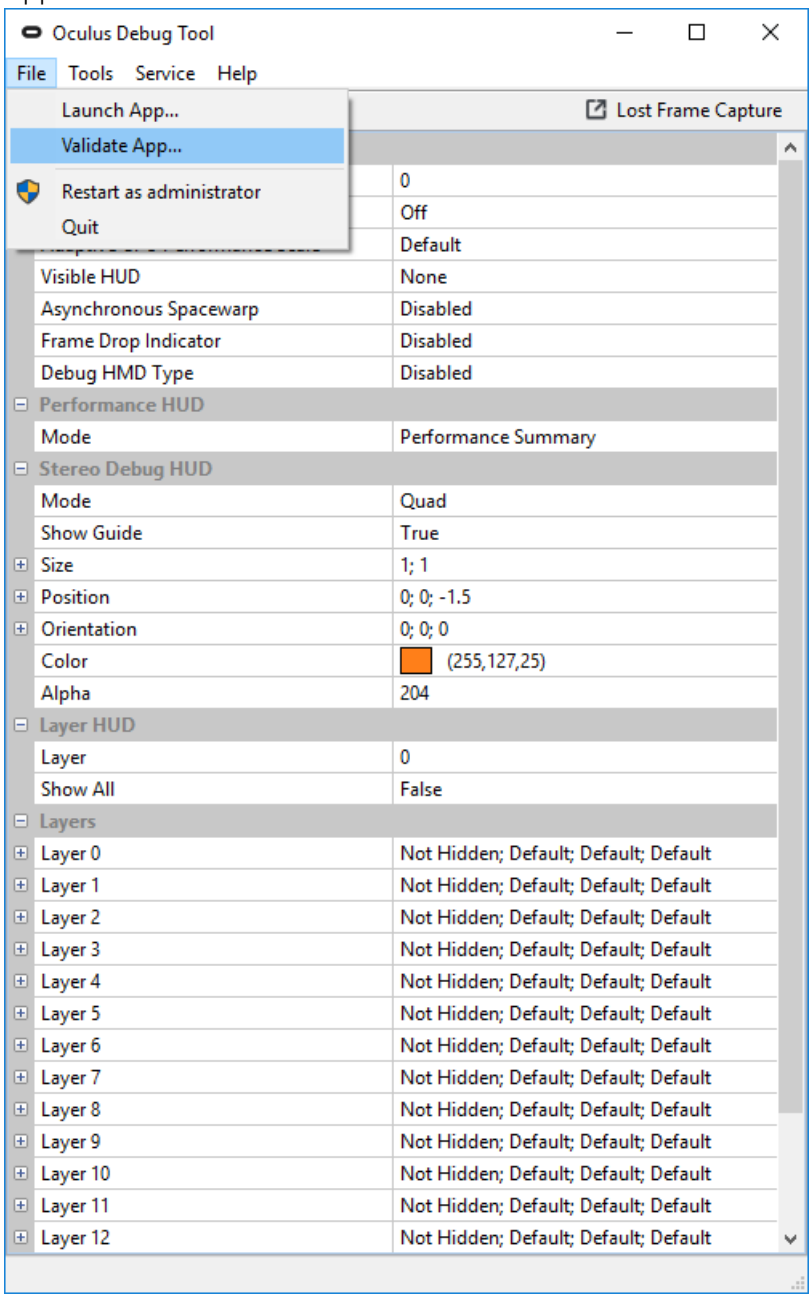
TestResponseToIadChanges

Tests if your app correctly adjusts for Inter Axial Distance (IAD) changes by getting updated HmdToEyeOffset values from ovr_GetRenderDesc at least once every 500ms. It also tests that your app actually responds to IAD changes.

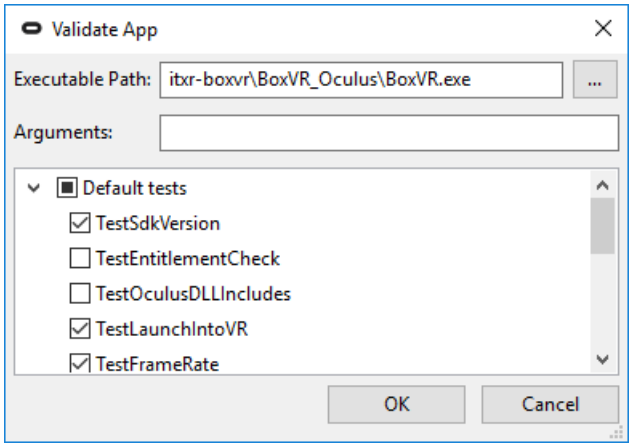
GUI Interface to VRC Validator

The Oculus Debug Tool provides a VRC Validator dialog. To use this GUI interface, follow these steps:

1. Make sure the Oculus application is running, and the headset is connected.
2. Start the Oculus Debug Tool as described in *Oculus Debug Tool* on page 132. Then, select File > Validate App...:



3. The Validate App dialoag appears (as shown below). For Executable Path, browse to an executable VR application that you wish to test. For Arguments, enter any arguments that you wish to specify for the VR application. Then, select the VRC tests that you wish to run. In this example, three VRC tests are selected:



4. Click OK to run the tests. The Validating App dialog appears (as shown below). In the left panel, a check mark indicates a successful test, an x indicates a failed test, and an animated clock icon indicates a test that is currently underway. The detailed results are shown in the right panel:

