Oculus Rift Developer Guide

Version 0.8.0.0

Copyrights and Trademarks

Contents

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_CreateSwapTextureSetD3D11` and`ovr_CreateSwapTextureSetGL`.
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 `ovr_SubmitFrame`.
6. Customize UI screens to work well inside of the headset.
7. Destroy the created resources during shutdown.
    - See: `ovr_DestroySwapTextureSet`, `ovr_Destroy`, and `ovr_Shutdown`.

A more complete summary of rendering details is provided in the *Rendering Setup Outline* on page 14 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 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_DK1` 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.
`CameraFrustumHFovInRadians`	float	The horizontal FOV of the position tracker frustum.
`CameraFrustumVFovInRadians`	float	The vertical FOV of the position tracker frustum.
`CameraFrustumNearZInMeters`	float	The distance from the position tracker to the near frustum bounds.
`CameraFrustumNearZInMeters`	float	The distance from the position tracker to the far frustum bounds.
`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.

Head Tracking and Sensors

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

Starting with DK2, there is also a tracker 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.

By default the SDK will enable all of the available tracking features for the attached HMD. If you'd like to toggle the tracking features, you can do so by calling `ovr_ConfigureTracking`. This function has the following signature:

```
ovrResult ovr_ConfigureTracking(ovrSession session, unsigned int requestedTrackingCaps,
                  unsigned int requiredTrackingCaps);
```

`ovr_ConfigureTracking` takes two sets of capability flags as input. These both use flags declared in `ovrTrackingCaps`. `supportedTrackingCaps` describes the HMD tracking capabilities that should be used when available. `requiredTrackingCaps` specifies capabilities that must be supported by the HMD at the time of the call for the application to operate correctly. If the required capabilities are not present, `ovr_ConfigureTracking` will fail.

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 the current tracking state.
ovrTrackingState ts  = ovr_GetTrackingState(session, ovr_GetTimeInSeconds());

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

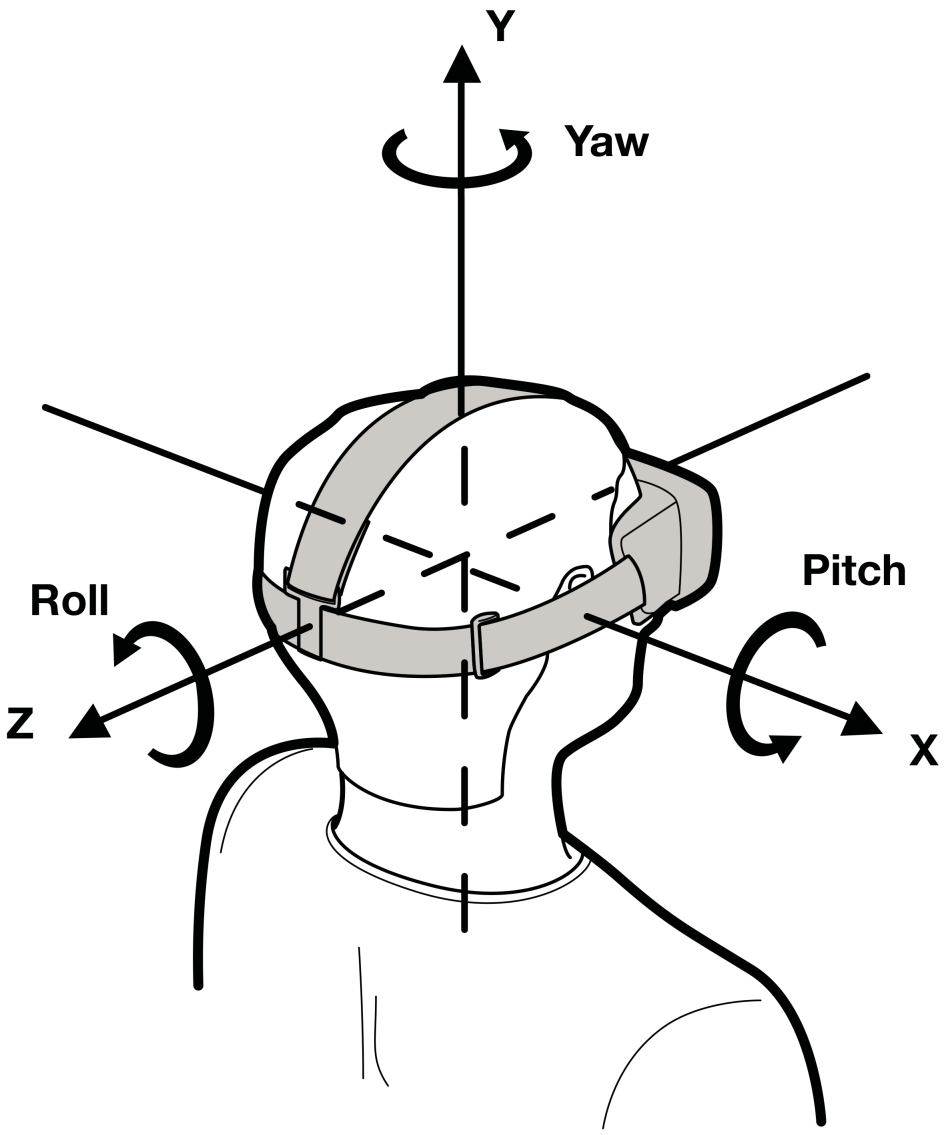
This example initializes the sensors with orientation, yaw correction, and position tracking capabilities if available, while only requiring basic orientation tracking. This means that the code will work for DK1, but will automatically use DK2 tracker-based position tracking. If you are using a DK2 headset and the DK2 tracker is not available during the time of the call, but is plugged in later, the tracker 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 with DK2, the `ovrStatus_PositionTracked` flag is only reported when `HeadPose` includes the absolute positional tracking data from the tracker.

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.

Position Tracking

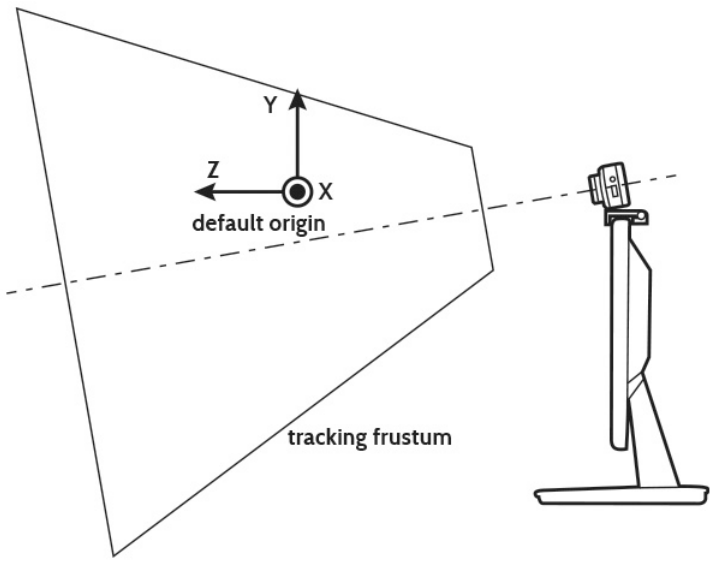
The frustum is defined by the horizontal and vertical FOV, and the distance to the front and back frustum planes.

Approximate values for these parameters can be accessed through the `ovrHmdDesc` 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 DK2 position tracker mounted on a PC monitor and a representation of the resulting tracking frustum.

Figure 2: Position Tracking Camera and Tracking Frustum



The relevant parameters and typical values are list below:

Field	Type	Typical Value
`CameraFrustumHFovInRadians`	float	1.292 radians (74 degrees)
`CameraFrustumVFovInRadians`	float	0.942 radians (54 degrees)
`CameraFrustumNearZInMeters`	float	0.4m
`CameraFrustumFarZInMeters`	float	2.5m

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

Note: Although the tracker 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 tracker in the direction of the optical axis but with the same height as the tracker. The default origin orientation is level with the ground with the negative axis pointing towards the tracker. In other words, a headset yaw angle of zero corresponds to the user looking towards the tracker.

Note: This can be modified using the API call `ovr_RecenterPose` which resets the tracking origin to the headset's current location, and sets the yaw origin to the current headset yaw value.

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.
- `CameraPose`—the pose of the tracker relative to the tracking origin.
- `LeveledCameraPose`— the pose of the tracker 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.

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

- `ovrStatus_PositionConnected`—set when the position tracker is connected and functioning properly.
- `ovrStatus_PositionTracked`—flag that is set only when the headset is being actively tracked.
- `ovrStatus_CameraPoseTracked`—set after the initial tracker calibration has taken place. Typically this requires the headset to be reasonably stationary within the view frustum for a second or so at the start of tracking. It may be necessary to communicate this to the user if the `ovrStatus_CameraPoseTracked` flag doesn't become set quickly after entering VR.

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 tracker).
- The exterior of the headset is partially or fully occluded from the tracker'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.

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 display a health and safety warning when the device is used.

This warning appears for a short amount of time when the Rift first displays a VR scene; it can be dismissed by pressing a key or tapping on the headset. Currently, the warning displays for at least 15 seconds the first time a new profile user puts on the headset and 3 seconds afterwards.

The warning displays automatically as a layer.

The Health and Safety Warning can be disabled through the Oculus Configuration Utility. Before suppressing the Health and Safety Warning, please note that by disabling the Health and Safety warning screen, you agree that you have read the warning, and that no other person will use the headset without reading this warning screen.

To use the Oculus Configuration Utility to suppress the Health and Safety Warning, a registry key setting must be added for Windows builds, while an environment variable must be added for non-Windows builds.

For Windows, the following key must be added:

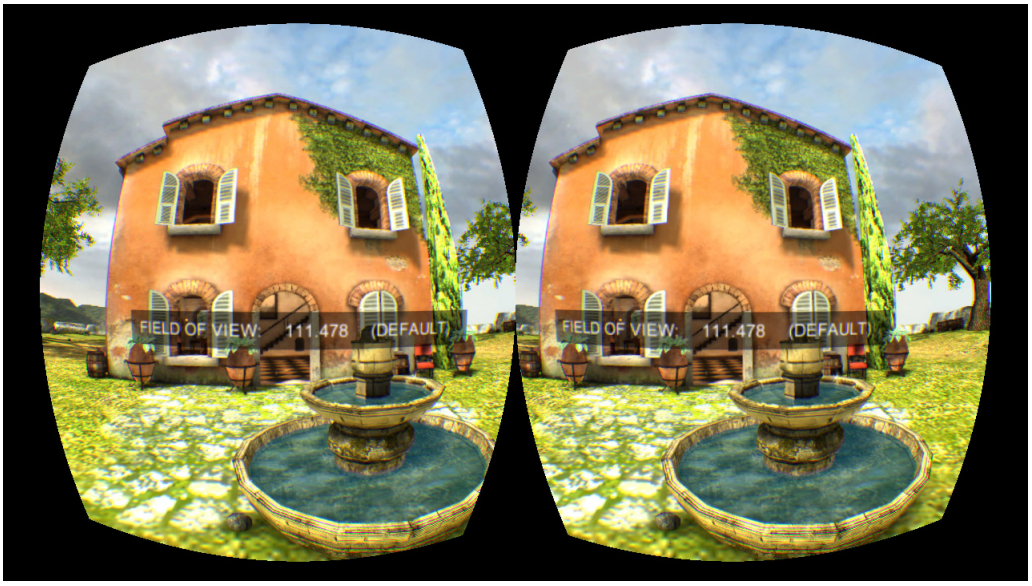
```
HKEY LOCAL MACHINE\Software\Wow6432Node\Oculus VR, LLC\LibOVR\HSWToggleEnabled
```

Setting the value of `HSWToggleEnabled` to 1 enables the Disable Health and Safety Warning check box in the Advanced Configuration panel of the Oculus Configuration Utility. For non-Windows builds, you must create an environment variable named `Oculus LibOVR HSWToggleEnabled` with the value of "1".

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_SubmitFrame`. 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.

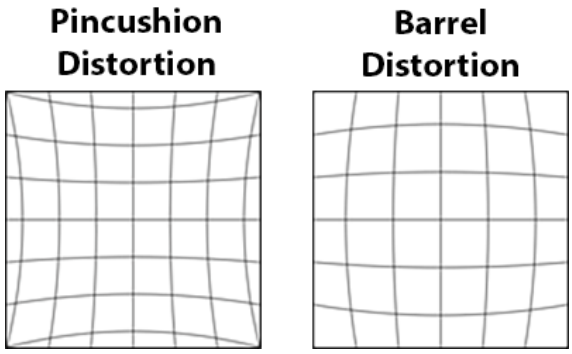
Note:

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.

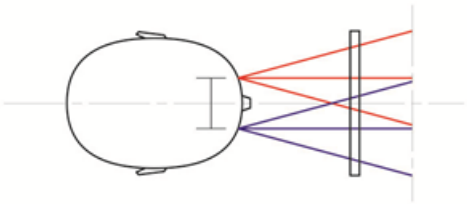
Figure 4: Pincushion and Barrel Distortion



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 pointing in the same direction (determined by the orientation of the HMD in the real world), and such that the distance between them is the same as the distance between the eyes, or interpupillary distance (IPD). This is typically done by adding the `ovrEyeRenderDesc::HmdToEyeViewOffset` translation vector to the translation component of the view matrix.

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 `ovrSwapTextureSet` objects, used to represent eye buffers, in an API-specific way: call `ovr_CreateSwapTextureSetD3D11` for Direct3D or `ovr_CreateSwapTextureSetGL` for OpenGL.
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 identified by the `ovrSwapTextureSet::CurrentIndex` variable.
    c. Call `ovr_SubmitFrame`, passing swap texture set(s) from the previous step within a `ovrLayerEyeFov` structure. Although a single layer is required to submit a frame, you can use multiple layers and layer types for advanced rendering. `ovr_SubmitFrame` passes layer textures to the compositor which handles distortion, timewarp, and GPU synchronization before presenting it to the headset.
    d. Advance `CurrentIndex` within each used texture set to target the next consecutive texture buffer for the following frame.
3. Shutdown:
    a. Call `ovr_DestroySwapTextureSet` to destroy swap texture buffers. Call `ovr_DestroyMirrorTexture` to destroy a mirror texture. To destroy the `ovrSession` object, call `ovr_Destroy`.

Swap Texture Set Initialization

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

Initially, you determine the rendering FOV and allocate the required ovrSwapTextureSet. 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. Once texture size is known, the application can call `ovr_CreateSwapTextureSetGL` or `ovr_CreateSwapTextureSetD3D11` to allocate the texture sets in an API-specific way. Here's how a texture set can be created and accessed under OpenGL:

```
ovrSwapTextureSet * pTextureSet = 0;

if (ovr_CreateSwapTextureSetGL(session, GL_SRGB8_ALPHA8, bufferSize.w, bufferSize.h,
                                       &pTextureSet) == ovrSuccess)
{
    // Sample texture access:
    ovrGLTexture* tex = (ovrGLTexture*)&pTextureSet->Textures[i];
    glBindTexture(GL_TEXTURE_2D, tex->OGL.TexId);
    ...
}
```

As can be seen from this example, `ovrSwapTextureSet` contains an array of `ovrTexture` objects, each wrapping either a D3D texture handle or OpenGL texture ID that can be used for rendering. Here's a similar example of texture set creation and access using Direct3D:

```
ovrSwapTextureSet *      pTextureSet = 0;
ID3D11RenderTargetView * pTexRtv[3];

D3D11_TEXTURE2D_DESC dsDesc;
dsDesc.Width            = bufferSize.w;
dsDesc.Height           = bufferSize.h;
dsDesc.MipLevels        = 1;
dsDesc.ArraySize        = 1;
dsDesc.Format           = DXGI_FORMAT_B8G8R8A8_UNORM_SRGB;
dsDesc.SampleDesc.Count   = 1;
dsDesc.SampleDesc.Quality = 0;
dsDesc.Usage            = D3D11_USAGE_DEFAULT;
dsDesc.CPUAccessFlags   = 0;
dsDesc.MiscFlags        = 0;
dsDesc.BindFlags        = D3D11_BIND_SHADER_RESOURCE | D3D11_BIND_RENDER_TARGET;

if (ovr_CreateSwapTextureSetD3D11(session, DIRECTX.Device, &dsDesc, 0, &pTextureSet) == ovrSuccess)
{
    for (int i = 0; i < pTextureSet->TextureCount; ++i)
    {
        ovrD3D11Texture* tex = (ovrD3D11Texture*)&pTextureSet->Textures[i];
        DIRECTX.Device->CreateRenderTargetView(tex->D3D11.pTexture, NULL, &pTexRtv[i]);
    }
}
```

In this case, you can use the newly created render target views 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 swap texture sets. 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 swap-texture-sets, 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 DXGI_FORMAT_BC1_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 D3D11, the texture format provided in `desc` for `ovr_CreateSwapTextureSetD3D11` is used by the distortion compositor for the ShaderResourceView when reading the contents of the texture. As a result, the application should request swap-texture-set formats that are in sRGB-space (e.g. DXGI_FORMAT_R8G8B8A8_UNORM_SRGB).

If your application is configured to render into a linear-format texture (e.g. DXGI_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. DXGI_FORMAT_R8G8B8A8_UNORM_SRGB) swap-texture-set.
- Use the `ovrSwapTextureSetD3D11_Typeless` flag.
- Create a linear-format RenderTargetView (e.g. DXGI_FORMAT_R8G8B8A8_UNORM)

Note: The `ovrSwapTextureSetD3D11_Typeless` flag for depth buffer formats (e.g. DXGI_FORMAT_D32) is ignored as they are always converted to be typeless.

For OpenGL, the `format` parameter of`ovr_CreateSwapTextureSetGL` is used by the distortion compositor when reading the contents of the texture. As a result, the application should request swap-texture-set formats preferably in sRGB-space (e.g. GL_SRGB8_ALPHA8). 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. GL_SRGB8_ALPHA8) swap-texture-set.
- Do not call `glEnable(GL_FRAMEBUFFER_SRGB);` when rendering into the swap texture.

The provided code sample demonstrates how to use the provided `ovrSwapTextureSetD3D11_Typeless` flag in D3D11:

```
D3D11_TEXTURE2D_DESC dsDesc;
dsDesc.Width = sizeW;
dsDesc.Height = sizeH;
dsDesc.MipLevels = 1;
dsDesc.ArraySize = 1;
dsDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM_SRGB;
dsDesc.SampleDesc.Count = 1;   // No multi-sampling allowed
dsDesc.SampleDesc.Quality = 0;
dsDesc.Usage = D3D11_USAGE_DEFAULT;
dsDesc.CPUAccessFlags = 0;
dsDesc.MiscFlags = 0;
dsDesc.BindFlags = D3D11_BIND_SHADER_RESOURCE | D3D11_BIND_RENDER_TARGET;

ovrResult result = ovr_CreateSwapTextureSetD3D11(session, DIRECTX.Device, &dsDesc,
 ovrSwapTextureSetD3D11_Typeless, &TextureSet);

if(!OVR_SUCCESS(result))
    return;

for (int i = 0; i < TextureSet->TextureCount; ++i)
{
    ovrD3D11Texture* tex = (ovrD3D11Texture*)&TextureSet->Textures[i];
    D3D11_RENDER_TARGET_VIEW_DESC rtvd = {};
    rtvd.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
    rtvd.ViewDimension = D3D11_RTV_DIMENSION_TEXTURE2D;
    DIRECTX.Device->CreateRenderTargetView(tex->D3D11.pTexture, &rtvd, &TexRtv[i]);
}
```

In addition to sRGB, these concepts also apply to the mirror texture creation. For more information, refer to the function documentation provided for `ovr_CreateMirrorTextureD3D11` and `ovr_CreateMirrorTextureGL` for D3D11 and OpenGL, 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_SubmitFrame`. After the frame is submitted, the Oculus compositor handles distortion and presents it on the Rift.

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];
ovrVector3f      hmdToEyeViewOffset[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]);
hmdToEyeViewOffset[0] = eyeRenderDesc[0].HmdToEyeViewOffset;
hmdToEyeViewOffset[1] = eyeRenderDesc[1].HmdToEyeViewOffset;

// Initialize our single full screen Fov layer.
ovrLayerEyeFov layer;
layer.Header.Type      = ovrLayerType_EyeFov;
layer.Header.Flags     = 0;
layer.ColorTexture[0]  = pTextureSet;
layer.ColorTexture[1]  = pTextureSet;
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 `HmdToEyeViewOffset` 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, hmdToEyeViewOffset, 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 trackers. 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)
{
    // Increment to use next texture, just before writing
    pTextureSet->CurrentIndex = (pTextureSet->CurrentIndex + 1) % pTextureSet->TextureCount;

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

    // Render Scene to Eye Buffers
    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,
                                               ovrProjection_RightHanded);
        // Render the scene for this eye.
        DIRECTX.SetViewport(layer.Viewport[eye]);
        roomScene.Render(proj * view, 1, 1, 1, 1, true);
    }
}

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

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

- First it increments the `CurrentIndex` to point to the next texture within the output texture set. `CurrentIndex` must be advanced round-robin fashion every time we draw a new frame.
- 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_SubmitFrame 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_SubmitFrame` returns once frame present is queued up and the next texture slot in the `ovrSwapTextureSet` is available for the next 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 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 is 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_DestroySwapTextureSet), destroy the session (ovr_Destory), recreate it (ovr_Create), and create new resources (ovr_CreateSwapTextureSetXXX). 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` value must be passed to `ovr_SubmitFrame` as was used to obtain timing for the frame. The details of multi-threaded timing are covered in the next section, *Rendering on Different Threads* on page 19.

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_SubmitFrame`, along with the frame index.

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_SubmitFrame`.

The following code illustrates this in more detail:

```
void MainThreadProcessing()
{
    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];

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

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

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

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, 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.
EyeFovDepth	An eye buffer render with depth buffer information. Currently, only layer #0 can be of this type.  Note: The depth buffer is not currently used for occlusion (Z testing) between layer types.
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.
Direct	Displayed directly on the framebuffer, this is intended primarily for debugging. No timewarp, distortion or chromatic aberration is applied to this layer; images from this layer type will usually not look correct or comfortable while wearing the HMD.
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`	`ovrSwapTextureSet`	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.
`DepthTexture`	`ovrSwapTextureSet`	Provides depth data for the EyeFovDepth layer type, and is used by positional timewarp to try to apply the correct parallax for the layer. This data is not used for occlusion or

Parameter	Type	Description
		intersection with other layers. It does not have to match the ColorTexture resolution, and 2x or 4x MSAA is allowed.
`ProjectionDesc`	`ovrTimewarp ProjectionDesc`	Supplies information about how to interpret the data held in `DepthTexture` for the EyeFovDepth layer type. This should be extracted from the application's projection matrix using the `ovrTimewarpProjectionDesc _FromProjection` utility function.
`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 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

Parameter	Type	Description
		"zero" position set by `ovr_RecenterPose` 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 `ovrSwapTextureSet` for the left and right eyes, and a viewport for each.
- Stereo data, shared texture—the app supplies the same `ovrSwapTextureSet` 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 `ovrSwapTextureSet` 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 a slightly more expensive but higher-quality path in the compositor for this layer. This can provide a significant increase in legibility, especially when used with a texture with mipmaps; this is recommended for high-frequency images such as text or diagrams and when used with the Quad layer types. It has relatively little visual effect on the Eye layer types with typical virtual world images.
- `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_RecenterPose`. 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 `ovrSwapTextureSet` the application wants to update, the data for each layer is put into the relevant `ovrLayerEyeFov` / `ovrLayerEyeFovDepth` / `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_SubmitFrame` function.

```
// 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   = TheHudTextureSet;
// 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] = hmdToEyeViewOffset[0];
viewScaleDesc.HmdToEyeViewOffset[1] = hmdToEyeViewOffset[1];

ovrResult result = ovr_SubmitFrame(Hmd, 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_SubmitFrame` queues the layers for display, and transfers control of the `CurrentIndex` texture inside the `ovrSwapTextureSet` 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_SubmitFrame` is called, so care must be taken. Oculus strongly recommends that the application should not try to use or render to any of the textures and indices that were submitted in the most recent `ovr_SubmitFrame` call. For example:

```
// Create two SwapTextureSets to illustrate. Each will have two textures, [0] and [1].
ovrSwapTextureSet *eyeSwapTextureSet;
ovr_CreateSwapTextureSetD3D11 ( ... &eyeSwapTextureSet );
ovrSwapTextureSet *hudSwapTextureSet;
ovr_CreateSwapTextureSetD3D11 ( ... &hudSwapTextureSet );

// 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;

// Right now (no calls to ovr_SubmitFrame done yet)
// eyeSwapTextureSet->Textures[0]: available
// eyeSwapTextureSet->Textures[1]: available
// hudSwapTextureSet->Textures[0]: available
```

```
// hudSwapTextureSet->Textures[1]: available

// Frame 1.
eyeSwapTextureSet->CurrentIndex = 0;
hudSwapTextureSet->CurrentIndex = 0;
eyeLayer.ColorTexture[0] = eyeSwapTextureSet;
eyeLayer.ColorTexture[1] = eyeSwapTextureSet;
hudLayer.ColorTexture = hudSwapTextureSet;
ovr_SubmitFrame(Hmd, 0, nullptr, layerList, 2);

// Now,
// eyeSwapTextureSet->Textures[0]: in use by compositor
// eyeSwapTextureSet->Textures[1]: available
// hudSwapTextureSet->Textures[0]: in use by compositor
// hudSwapTextureSet->Textures[1]: available

// Frame 2.
eyeSwapTextureSet->CurrentIndex = 1;
AppRenderScene ( eyeSwapTextureSet->Textures[1] );
// App does not render to the HUD, does not change the layer setup.
ovr_SubmitFrame(Hmd, 0, nullptr, layerList, 2);

// Now,
// eyeSwapTextureSet->Textures[0]: available
// eyeSwapTextureSet->Textures[1]: in use by compositor
// hudSwapTextureSet->Textures[0]: in use by compositor
// hudSwapTextureSet->Textures[1]: available

// Frame 3.
eyeSwapTextureSet->CurrentIndex = 0;
AppRenderScene ( eyeSwapTextureSet->Textures[0] );
// App hides the HUD
hudLayer.Header.Type = ovrLayerType_Disabled;
ovr_SubmitFrame(Hmd, 0, nullptr, layerList, 2);

// Now,
// eyeSwapTextureSet->Textures[0]: in use by compositor
// eyeSwapTextureSet->Textures[1]: available
// hudSwapTextureSet->Textures[0]: available
// hudSwapTextureSet->Textures[1]: available
```

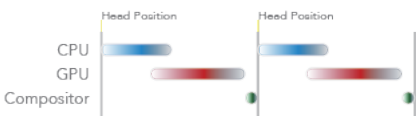
In other words, if the texture was used by the last `ovr_SubmitFrame` call, don't try to render to it. If it wasn't, you can.

Queue Ahead

To improve CPU and GPU parallelism and increase the amount of time that the GPU has to process a frame, the SDK now provides 2.8 milliseconds of queue ahead time by default.

When queue ahead is disabled, 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:

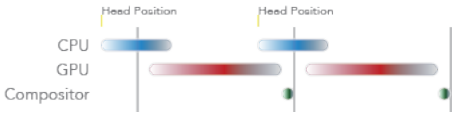
Figure 6: CPU and GPU Utilization without Queue Ahead



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

When queue ahead is enabled, 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 enabled:

Figure 7: CPU and GPU Utilization with Queue Ahead



If you need to disable queue ahead, call:

```
ovr_SetBool(Hmd, "QueueAheadEnabled", ovrFalse);
```

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.

Coping with Graphics API or Hardware Rendertarget 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_SubmitFrame` 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_SubmitFrame` 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 the DK1, 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 30.

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, isRightHanded);
```

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, isRightHanded);
ovrMatrix4f projMatrixRight = ovrMatrix4f_Projection(newFovRight, znear, zfar, isRightHanded);
```

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_SubmitFrame` or `ovr_GetRenderScaleAndOffset`.
- 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!

Chromatic Aberration

Chromatic aberration is a visual artifact seen when viewing images through lenses.

The phenomenon causes colored fringes to be visible around objects, and is increasingly more apparent as our view shifts away from the center of the lens. The effect is due to the refractive index of the lens varying for different wavelengths of light (shorter wavelengths towards the blue end of the spectrum are refracted less than longer wavelengths towards the red end). Since the image displayed on the Rift is composed of individual red, green, and blue pixels, it is susceptible to the unwanted effects of chromatic aberration. The manifestation, when looking through the Rift, is that the red, green, and blue components of the image appear to be scaled out radially, and by differing amounts. Exactly how apparent the effect is depends on the image content and to what degree users are concentrating on the periphery of the image versus the center.

Chromatic Aberration

Fortunately, programmable GPUs enable you to significantly reduce the degree of visible chromatic aberration, albeit at some additional GPU expense.

To do this, pre-transform the image so that the chromatic aberration of the lens will result in a more normal looking image. This is analogous to the way in which we pre-distort the image to cancel out the distortion effects generated by the lens.

Sub-Channel Aberration

Although we can reduce the artifacts through the use of distortion correction, we cannot completely remove them for an LCD display panel.

This is due to the fact that each color channel is actually comprised of a range of visible wavelengths, each of which is refracted by a different amount when viewed through the lens. As a result, although we are able to distort the image for each channel to bring the peak frequencies back into spatial alignment, it is not possible to compensate for the aberration that occurs within a color channel. Typically, when designing optical systems, chromatic aberration across a wide range of wavelengths is managed by carefully combining specific optical elements (in other texts, for example, look for “achromatic doublets”).

Oculus Touch Controllers

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 tracker 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 trackers 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);
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:

```
ovrTrackingState trackState = ovr_GetTrackingState(session, displayMidpointSeconds);
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.
ConnectionState	unsigned int	Described by ovrControllerType. Indicates which controller types are present; you can check the ovrControllerType_LTouch bit, for example, to verify that the left touch controller is connected. Options include: • ovrControllerType_LTouch (0x01) • ovrControllerType_RTouch (0x02) • ovrControllerType_Touch (0x03)
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 depressed.
HandTrigger[2]	float	Left and right hand trigger 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 depressed.
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

Field	Type	Description
		developers don't need to handle it explicitly.

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.

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 finder on the thumb stick 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 finder on the thumb stick 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.

Vibration is enabled by calling ovr_SetControllerVibration as follows:

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

Vibration is enabled by specifying frequency and amplitude. Frequency can take on values of 0.0f, 0.5f, and 1.0f, while amplitude ranges from 0.0f to 1.0f. Set amplitude and frequency to 0.0f to disable vibration.

Note: Prolonged high levels of vibration may reduce positional tracking quality. Right now, we recommend turning on vibration only for short periods of time.

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.

Low-Level Sensor Details

In normal use, applications use the API functions which handle sensor fusion, correction, and prediction for them.

Note: This section is left for reference; parts of it may be out of date after the introduction of the external position tracker with DK2.

In normal use, applications will use the API functions which handle sensor fusion, correction, and prediction for them. This section is provided purely for interest.

Developers can read the raw sensor data directly from ovrTrackingState::RawSensorData. This contains the following data:

```
ovrVector3f Accelerometer;  // Acceleration reading in m/s^2.
ovrVector3f Gyro;           // Rotation rate in rad/s.
ovrVector3f Magnetometer;   // Magnetic field in Gauss.
float       Temperature;    // Temperature of the sensor in degrees Celsius.
float       TimeInSeconds;  // Time when the reported IMU reading took place, in seconds.
```

Over long periods of time, a discrepancy will develop between Q (the current head pose estimate) and the true orientation of the Rift. This problem is called drift error, which described more in the following section. Errors in pitch and roll are automatically reduced by using accelerometer data to estimate the gravity vector.

Errors in yaw are reduced by magnetometer data. For many games, such as a standard first person shooter (FPS), the yaw direction is frequently modified by the game controller and there is no problem. However, in many other games or applications, the yaw error will need to be corrected. For example, if you want to maintain a cockpit directly in front of the player. It should not unintentionally drift to the side over time. Yaw error correction is enabled by default.

Performance Head-Up Display

The Performance Head-Up Display (HUD) enables you or your users to view performance information for any application built with SDK 0.6 or later.

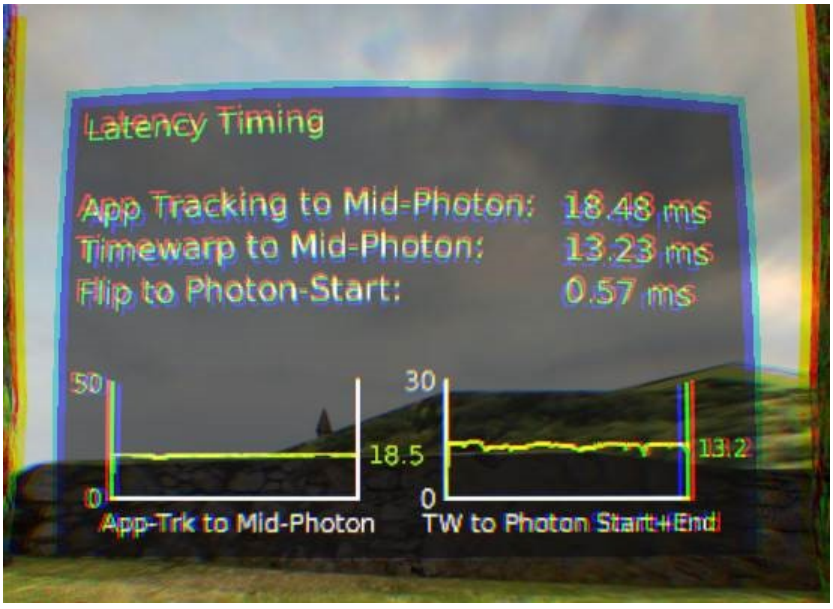
The Performance HUD screens are rendered by the compositor, which enables them to be displayed with a single SDK call. In OculusWorldDemo, you can toggle through the Performance HUD screens by pressing F11.

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 8: Latency Timing



The following table describes each metric:

Table 1: Latency Timing HUD

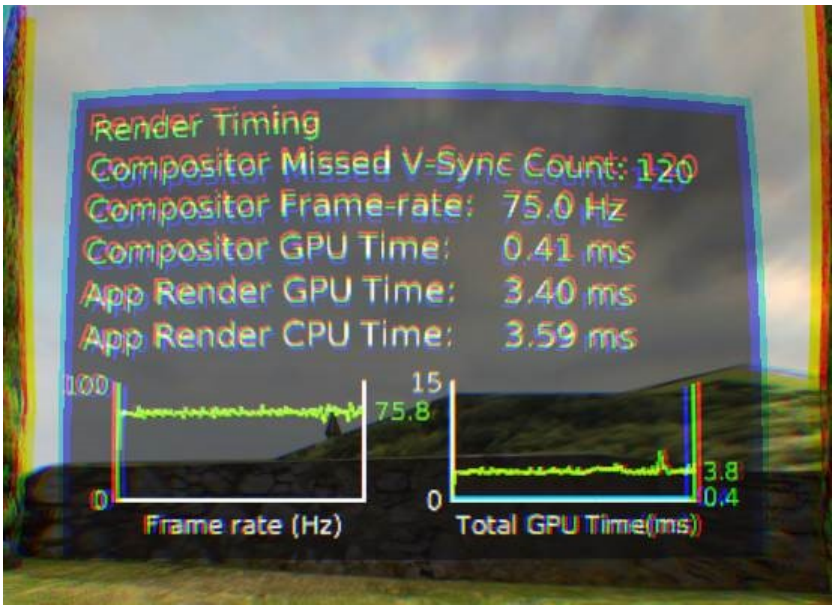
Metric	Description
App Tracking to Mid-Photon	Latency from when the app called ovr_GetTrackingState() to when that frame eventually was shown (i.e.illuminated) on the HMD display - averaged mid - point illumination
Timewarp to Mid-Photon	Latency from when the last predictied tracking info is fed to the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

Render Timing

The Render Timing HUD displays the frame rate of the compositor and the total time spent by the GPU rendering for both the client app and the compositor.

The following screenshot shows the Render Timing HUD:

Figure 9: Render Timing



The following table describes each metric:

Table 2: 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 at which final composition is happening outside of client app rendering. Because the compositor is locked to vsync, it will never exceed the native HMD refresh rate. But, if the compositor fails to finish a frame on time, it can drop below the HMD 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 client application.
App Render GPU Time	The total GPU time spent on rendering by the client application. This includes the work done after ovr_SubmitFrame() using the mirror texture if applicable. It also includes GPU command-buffer "bubbles" that might be injected due to the client application's CPU thread not pushing data fast enough to the GPU command buffer to keep it occupied.

Metric	Description
App Render CPU Time	The time difference from when the app starting executing on CPU after ovr_SubmitFrame() returned to when timewarp draw call was executed on the CPU. Will show "N/A" if latency tester is not functioning as expected. Includes IPC call overhead to compositor after ovr_SubmitFrame() is called by client application.
App - Tracking to TW GPU	The time from when the app called ovr_GetTrackingState to when the Timewarp draw call was executed on the GPU. The HUD displays "N/A" if the latency tester is not running as expected.

Performance Headroom

The Performance Headroom HUD displays the frame rate of the compositor and the unused hardware performance available. This HUD can be utilized by the developer or consumer when tuning their applications' simulation and graphics fidelity. Since the user cannot disable V-Sync, it can be thought of as a replacement for a frame rate counter to judge available performance. It can also help debug the setup to make sure it is providing a consistent experience in VR and not dropping frames due to issues unrelated to hardware performance.

The following screenshot shows the Performance Headroom HUD:

Figure 10: Performance Headroom HUD

The following table describes each metric:

Metric	Description
Motion-to-Photon Latency	Latency from when the last predictied tracking info is fed to the GPU for timewarp execution to the point when the middle scanline of that frame is illuminated on the HMD display. This is the same info presented in "Latency Timing" section, presented here for consumer - friendliness.

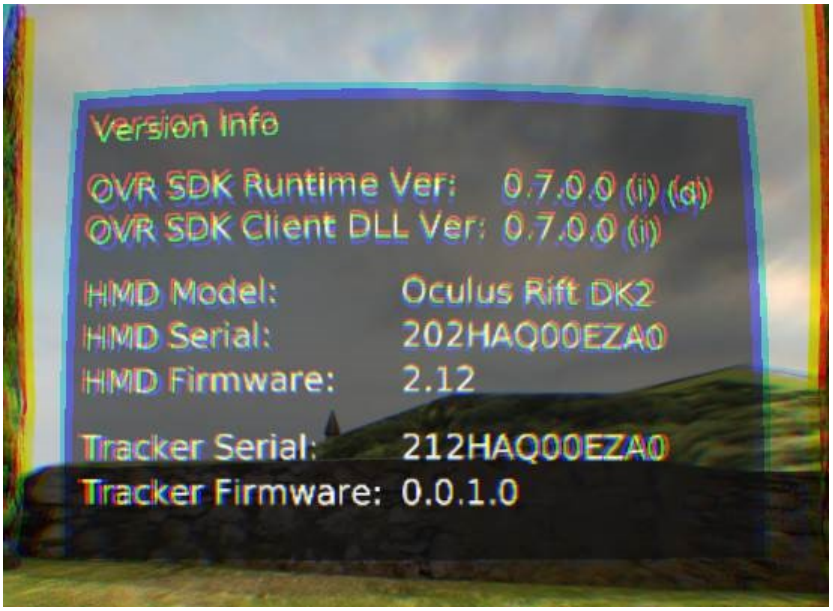
Metric	Description
Unused GPU performance	The percentage of GPU performance not used by the client application and compositor. This is essentially the total GPU time tracked in the Render Timing HUD divided by the native frame time (inverse of refresh rate) of the HMD. It is designed to help the user verify that his or her PC has enough GPU buffer to avoid dropping frames and to help the user avoid judder.
Total Frames Dropped	This is the same value provided in the Render Timing HUD to help the user determine whether he or she is encountering performance issues.

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 11: 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.

Name	Description
Tracker Serial	The serial number of the positional tracker.
Tracker Firmware	The version of the installed positional tracker firmware.

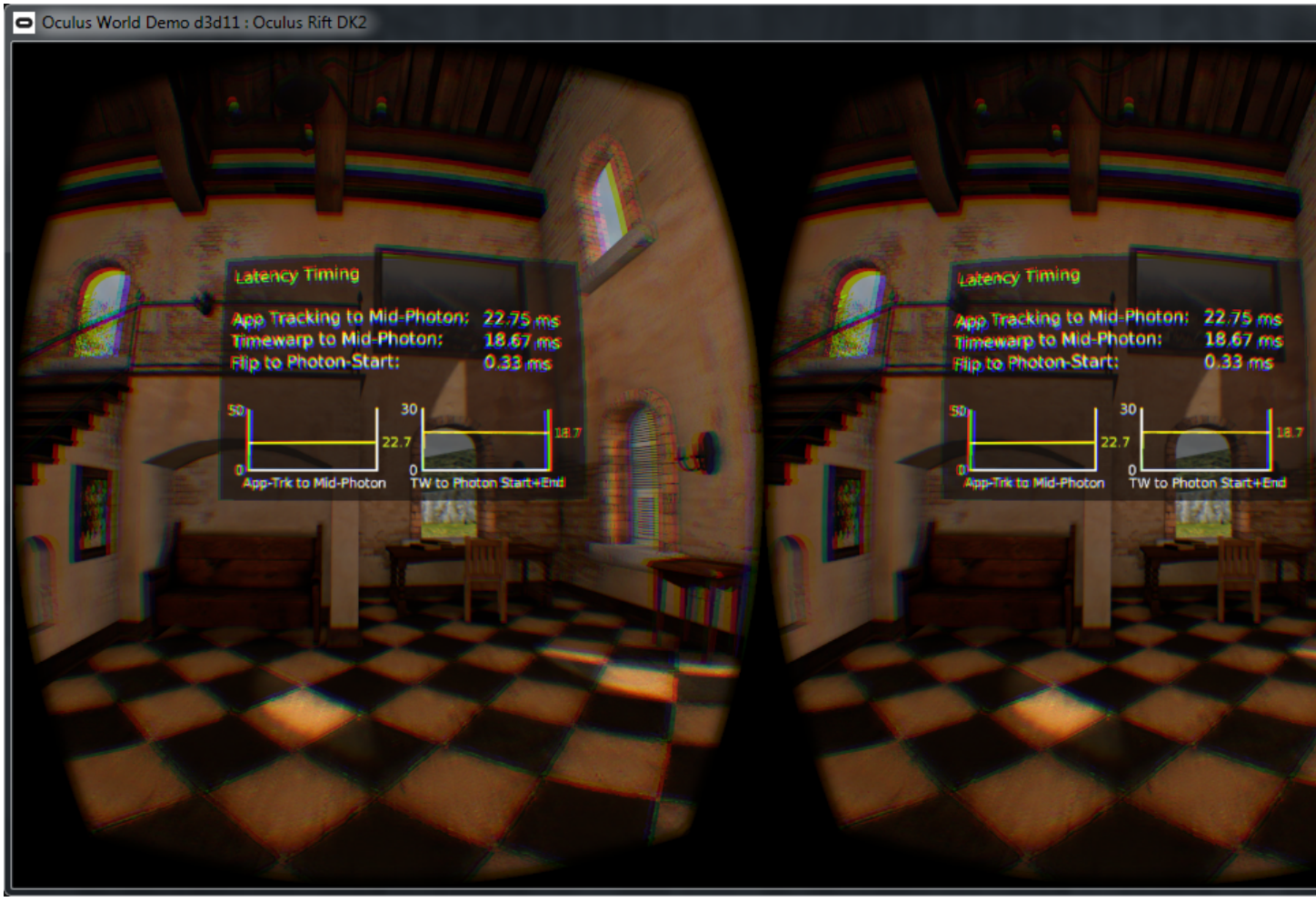
Oculus Debug Tool

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

To use the tool:

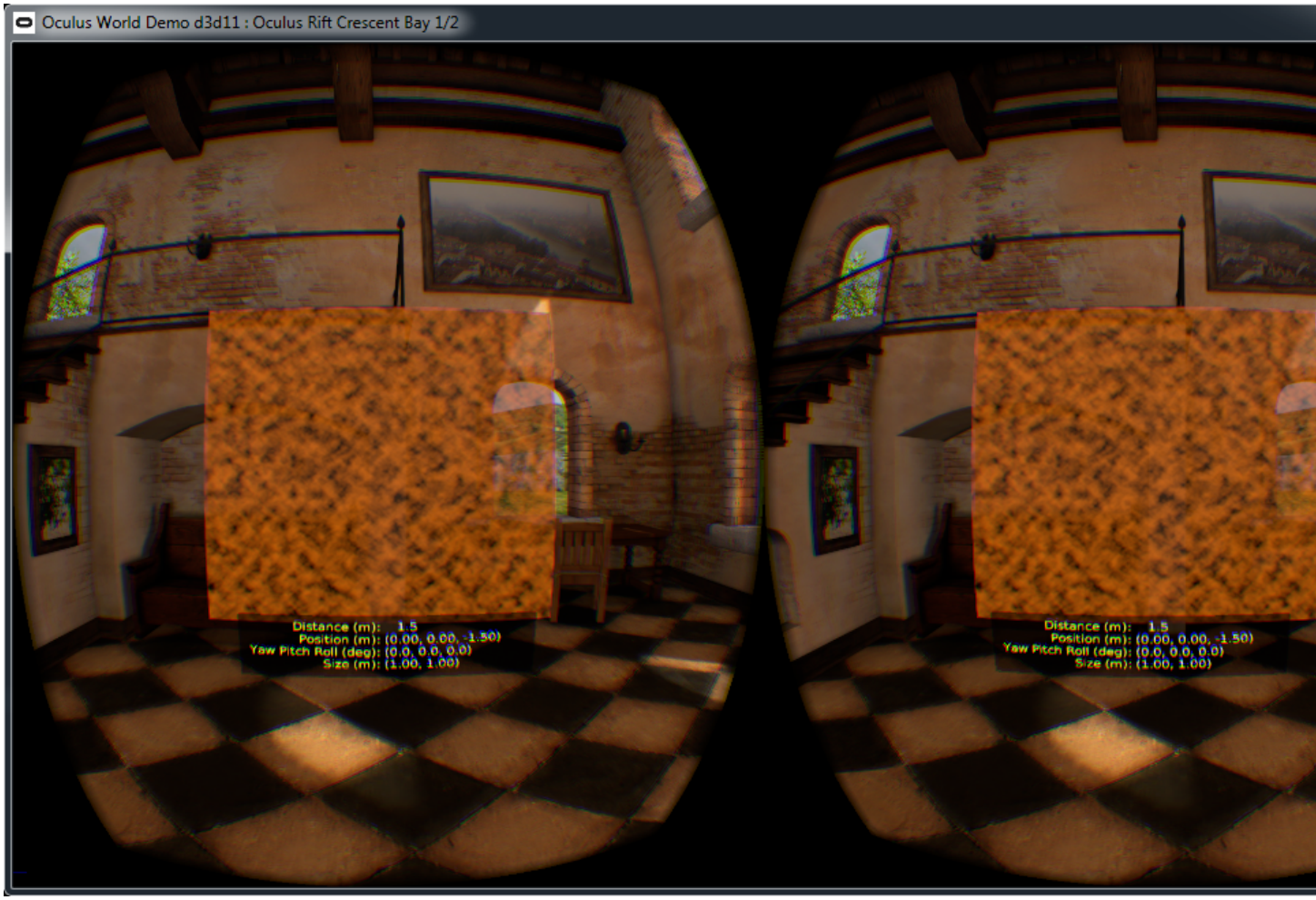
1. Go to `Tools` directory of the Oculus SDK.
2. Double-click `OculusDebugTool.exe`. The Oculus Debug Tool opens.
3. Select the Visible HUD to view. Options include: None (no HUD is displayed), Performance HUD, Stereo Debug HUD, or Layer HUD.
4. If you selected Performance HUD, select which Performance 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 40.

   The following is an example of the Performance HUD:



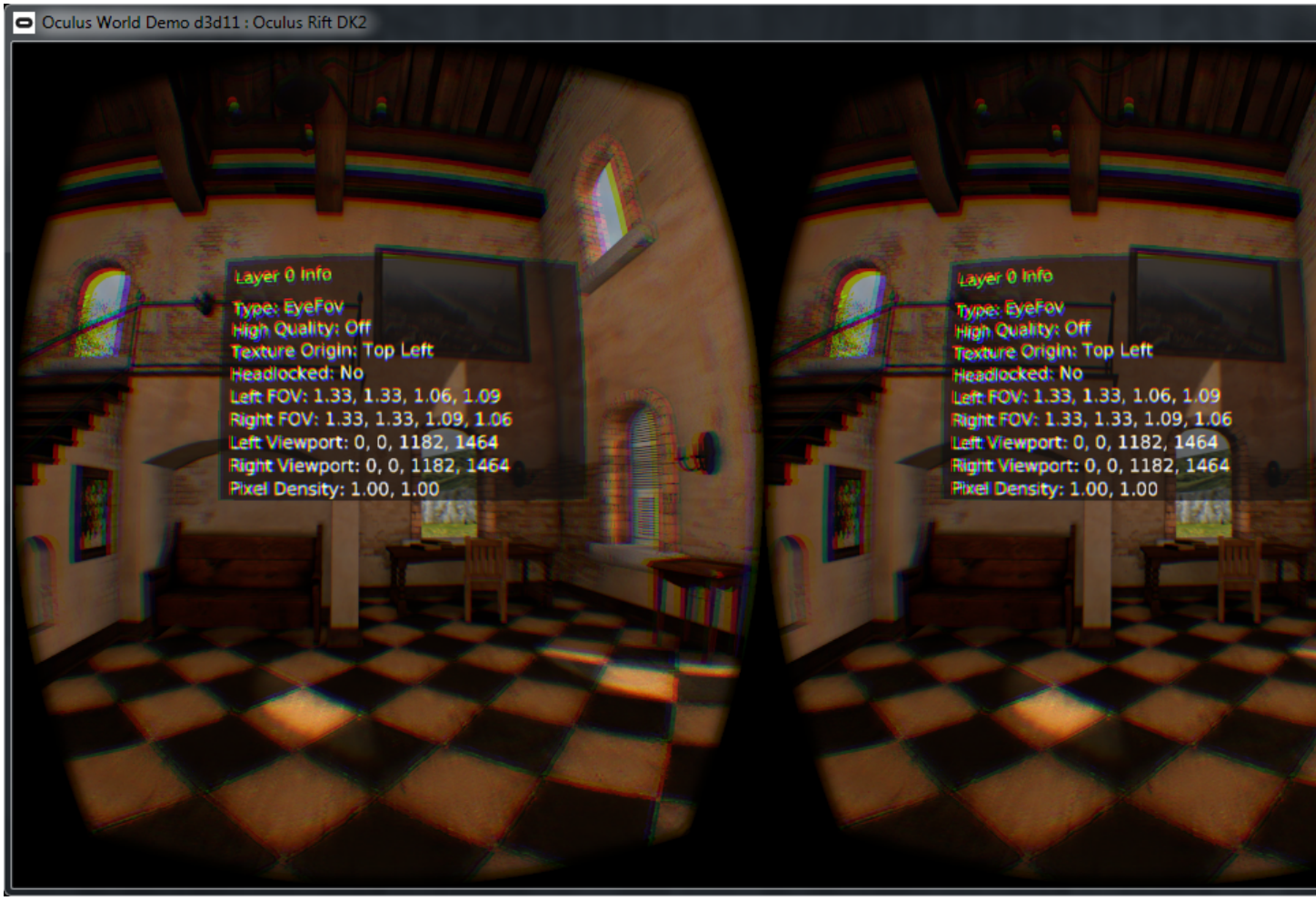
5. If you selected Stereo Debug HUD, configure the mode, size, position, and color from the Stereo Debug HUD options.

   The following is an example of the Stereo Debug HUD:



6. 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:



7. Put on the headset and view the results.