Oculus Rift Developer Guide

Copyrights and Trademarks

Contents

LibOVR Integration

The Oculus SDK is designed to be as easy to integrate as possible. This section 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.

Integrating LibOVR

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

1. Initialize LibOVR.
2. Enumerate Oculus devices, create the `ovrHmd` object, and start sensor input.
3. Integrate head-tracking into your application's view and movement code. This involves:
    a. Reading data from the Rift sensors through or `ovrHmd_GetTrackingState` or `ovrHmd_GetEyePoses`.
    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.
    b. For SDK rendered distortion, configure rendering based on system rendering API pointers and viewports.
    c. For client rendered distortion, create the necessary distortion mesh and shader resources.
5. Modify application frame rendering to integrate HMD support and proper frame timing:
    a. Make sure your engine supports multiple rendering views.
    b. Add frame timing logic into the render loop to ensure that motion prediction and timewarp work correctly.
    c. Render each eye's view to intermediate render targets.
    d. Apply distortion correction to render target views to correct for the optical characteristics of the lenses (only necessary for client rendered distortion).
6. Customize UI screens to work well inside of the headset.

Initialization and Sensor Enumeration

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

Review the following code:

```
// Include the OculusVR SDK
#include <OVR_CAPI_0_5_0.h>

void Application()
{
    if (ovr_Initialize(NULL))
    {
        ovrHmd hmd = ovrHmd_Create(0);

        if (hmd)
        {
            // Get more details about the HMD.
            ovrSizei resolution = hmd->Resolution;
            ...

            // Do something with the HMD.
            ...
            ovrHmd_Destroy(hmd);
        }

        ovr_Shutdown();
    }
}
```

As you can see from the code,`ovr_Initialize` must be called before using any of the API functions, and`ovr_Shutdown`must be called to shut down the library before you exit the program. In between these function calls, you are free to create HMD objects, access sensors, and perform application rendering.

In this example,`ovrHmd_Create(0)` creates the first available HMD. `ovrHmd_Create` accesses HMDs by index, which is an integer ranging from 0 to the value returned by`ovrHmd_Detect`. Users can call`ovrHmd_Detect`any time after library initialization to re-enumerate the connected Oculus devices. Finally,`ovrHmd_Destroy`must be called to clear the HMD before shutting down the library.

If no Rift is plugged in during detection,`ovrHmd_Create(0)` will return a null handle. In this case, you can use`ovrHmd_CreateDebug`to create a virtual HMD of the specified type. Although the virtual HMD will not provide any sensor input, it can be useful for debugging Rift compatible rendering code, and doing general development without a physical device.

The`ovrHmd`handle is actually a pointer to an`ovrHmdDesc` struct that contains information about the HMD and its capabilities, and is used to set up rendering. The following table describes the fields:

Field	Type	Description
`Type`	ovrHmdType	Name of the manufacturer.
`ProductName`	const char*	Name of the manufacturer.
`Manufacturer`	const char*	Name of the manufacturer.
`VendorId`	short	Vendor ID reported by the headset USB device.

Field	Type	Description
`ProductId`	short	Product ID reported by the headset USB device.
`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 tracking camera frustum.
`CameraFrustumVFovInRadians`	float	The vertical FOV of the position tracking camera frustum.
`CameraFrustumNearZInMeters`	float	The distance from the position tracking camera to the near frustum bounds.
`CameraFrustumNearZInMeters`	float	The distance from the position tracking camera to the far frustum bounds.
`HmdCaps`	unsigned int	HMD capability bits described by `ovrHmdCaps`.
`TrackingCaps`	unsigned int	Tracking capability bits describing whether orientation, position tracking, and yaw drift correction are supported.
`DistortionCaps`	unsigned int	Distortion capability bits describing whether timewarp and chromatic aberration correction are supported.
`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.
`EyeRenderOrder`	ovrEyeType[]	Preferred eye rendering order for best performance. Using this value can help reduce latency on sideways scanned screens.
`Resolution`	ovrSizei	Resolution of the full HMD screen (both eyes) in pixels.
`WindowsPos`	ovrVector2i	Location of the monitor window on the screen. Set to (0,0) if not supported.
`DisplayDeviceName`	const char *	System specific name of the display device.

Field	Type	Description
`DisplayId`	int	System specific ID of the display device.

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 an external camera to track headset position. The information from each of these sensors is combined through a process known as sensor fusion to determine the motion of the user's head in the real world, and to synchronize the user's virtual view in real-time.

To use the Oculus sensor, you first need to initialize tracking and sensor fusion by calling `ovrHmd_ConfigureTracking`. This function has the following signature:

```
ovrBool  ovrHmd_ConfigureTracking(ovrHmd hmd, unsigned int supportedTrackingCaps,
                                              unsigned int requiredTrackingCaps);
```

`ovrHmd_ConfigureTracking` takes two sets of capability flags as input. These both use flags declared in `ovrTrackingCaps`. `supportedTrackingCaps` describes the HMD tracking capabilities that the application supports 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, then `ovrHmd_ConfigureTracking` returns false.

After tracking is initialized, you can poll sensor fusion for head position and orientation by calling `ovrHmd_GetTrackingState`. These calls are demonstrated by the following code:

```
 // Start the sensor which provides the Rift's pose and motion.
ovrHmd_ConfigureTracking(hmd, ovrTrackingCap_Orientation |
                              ovrTrackingCap_MagYawCorrection |
                              ovrTrackingCap_Position, 0);

// Query the HMD for the current tracking state.
ovrTrackingState ts  = ovrHmd_GetTrackingState(hmd, 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 camera-based position tracking. If you are using a DK2 headset and the DK2 camera is not available during the time of the call, but is plugged in later, the camera is automatically enabled by the SDK.

After the sensors are initialized, the sensor state is obtained by calling `ovrHmd_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 camera.

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, `ovrHmd_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 `ovrHmd_GetTrackingState` is the current time or earlier then the tracking state returned will be based on the latest sensor readings with no prediction. In a production application, however, you should use one of the real-time computed values returned by `ovrHmd_BeginFrame` or `ovrHmd_BeginFrameTiming`. 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.

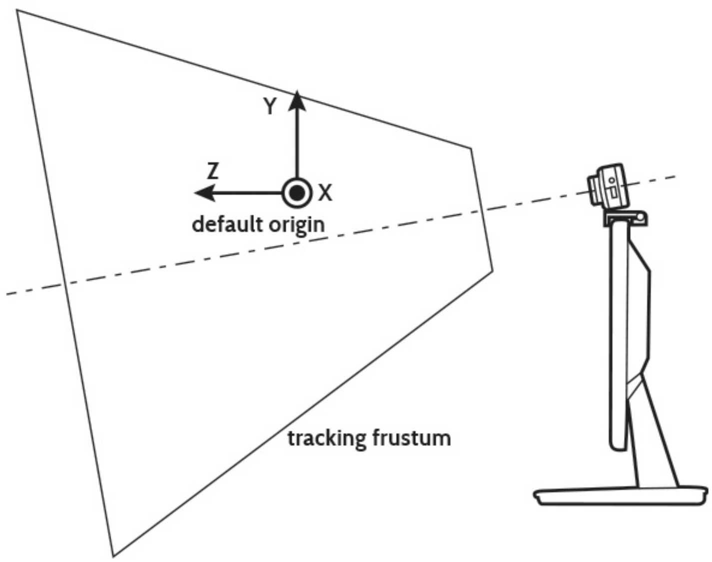
Approximate values for these parameters can be accessed through the `ovrHmdDesc` struct as follows:

```
ovrHmd      hmd = ovrHmd_Create(0);

if (hmd)
{
    // Extract tracking frustum parameters.
    float frustomHorizontalFOV = hmd->CameraFrustumHFovInRadians;
    ...
```

The following figure shows the DK2 position tracking camera 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 camera axis (and hence the tracking frustum) are shown tilted downwards slightly, the tracking coordinate system is always oriented horizontally such that the and axes are parallel to the ground.

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

Note: This can be modified using the API call `ovrHmd_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 `ovrHmd_GetTrackingState`. The returned `ovrTrackingState` struct contains several items relevant to position tracking:

- `HeadPose`—includes both head position and orientation.
- `CameraPose`—the pose of the camera relative to the tracking origin.

- `LeveledCameraPose`— the pose of the camera 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`—this is set when the position tracking camera is connected and functioning properly.
- `ovrStatus_PositionTracked`—flag that is set only when the headset is being actively tracked.
- `ovrStatus_CameraPoseTracked`—this is set after the initial camera 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 camera).
- The exterior of the headset is partially or fully occluded from the tracking camera'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 must integrate code that displays a health and safety warning when the device is used.

This warning must appear 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 6 seconds afterwards.

The warning displays automatically as an overlay in SDK Rendered mode. In App rendered mode, it is left for developers to implement. To support timing and rendering the safety warning, we've added two functions to the C API: `ovrHmd_GetHSWDisplayState` and `ovrHmd_DismissHSWDisplay`.

`ovrHmd_GetHSWDisplayState` reports the state of the warning described by the `ovrHSWDisplayState` structure, including the displayed flag and how much time is left before it can be dismissed. `ovrHmd_DismissHSWDisplay` should be called in response to a keystroke or gamepad action to dismiss the warning.

The following code snippet illustrates how health and safety warning can be handled:

```
    // Health and Safety Warning display state.
    ovrHSWDisplayState hswDisplayState;
    ovrHmd_GetHSWDisplayState(HMD, &hswDisplayState);

    if (hswDisplayState.Displayed)
    {
        // Dismiss the warning if the user pressed the appropriate key or if the
user
        // is tapping the side of the HMD.
        // If the user has requested to dismiss the warning via keyboard or
controller input...
        if (Util_GetAndResetHSWDismissedState())
            ovrHmd_DismissHSWDisplay(HMD);
        else
        {
            // Detect a moderate tap on the side of the HMD.
            ovrTrackingState ts = ovrHmd_GetTrackingState(HMD,
ovr_GetTimeInSeconds());

            if (ts.StatusFlags & ovrStatus_OrientationTracked)
            {
                const OVR::Vector3f v(ts.RawSensorData.Accelerometer.x,
                                      ts.RawSensorData.Accelerometer.y,
                                      ts.RawSensorData.Accelerometer.z);

                // Arbitrary value and representing moderate tap on the side of
the DK2 Rift.
                if (v.LengthSq() > 250.f)
                    ovrHmd_DismissHSWDisplay(HMD);
            }
        }
    }
```

With the release of 0.4.3, 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 if the Windows OS is 32-bit:

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

If the Windows OS is 64-bit, the path will be slightly different:

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

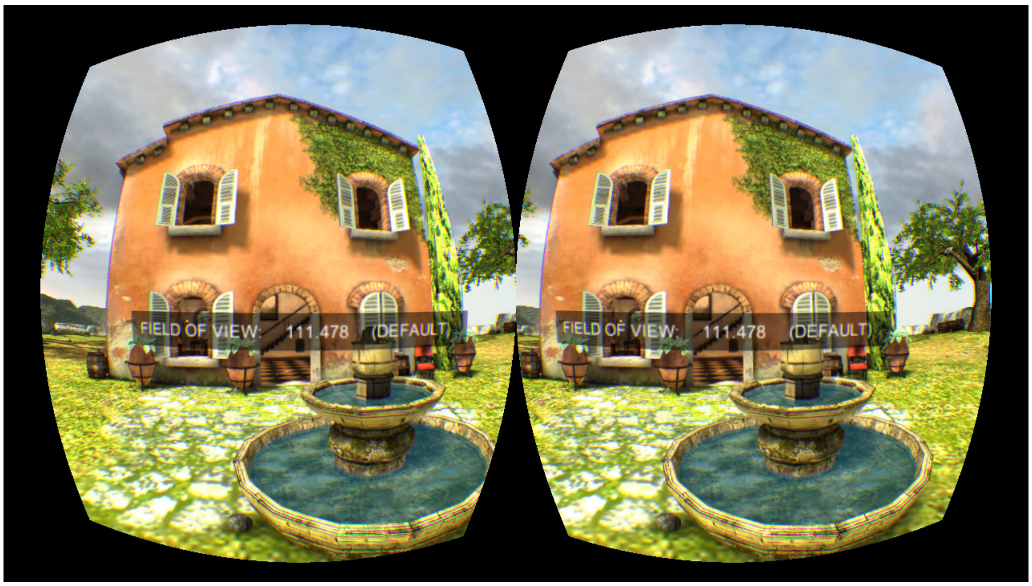
Setting the value of `HSWToggleEnabled` to 1 enables the Disable Health and Safety Warning check box in the Advanced Configuration panel of the Oculus Configuration Utility. For non-Windows builds, setting an environment variable named `Oculus LibOVR HSWToggleEnabled` must be created with the value of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

Rendering to the Oculus Rift

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

Setting this up can be tricky, but proper distortion correction is critical to achieving an immersive experience.

Figure 3: OculusWorldDemo Stereo Rendering



The Oculus C API provides two types of distortion correction: SDK distortion rendering and Client (application-side) distortion rendering. For each type, the application renders stereo views into individual render textures or into a single combined one. The differences appear in the way the APIs handle distortion, timing, and buffer swap:

- With the SDK distortion rendering approach, the library takes care of timing, distortion rendering, and buffer swap (the `Present` call). To make this possible, developers provide low level device and texture pointers to the API, and instrument the frame loop with `ovrHmd_BeginFrame` and `ovrHmd_EndFrame` calls that do all of the work. No knowledge of distortion shaders (vertex or pixel-based) is required.
- With Client distortion rendering, distortion must be rendered by the application code. This is similar to the approach used in SDK Version 0.2. However, distortion rendering is now mesh-based. In other words, the distortion is encoded in mesh vertex data rather than using an explicit function in the pixel shader. To support distortion correction, the Oculus SDK generates a mesh that includes vertices and UV coordinates used to warp the source render target image to the final buffer. The SDK also provides explicit frame timing functions used to support timewarp and prediction.

Rendering to the Oculus Rift

The Oculus Rift requires the scene to be rendered in split-screen stereo with half 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 Distortion

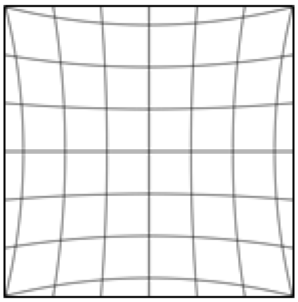
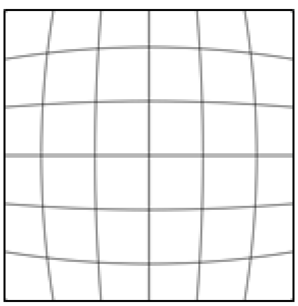


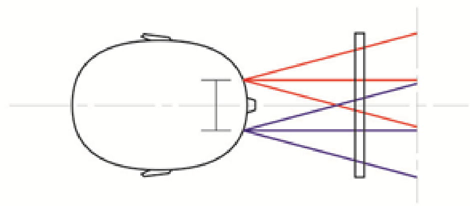
Figure 5: Barrel Distortion



To counteract this distortion, the software must apply 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 software must also correct 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 6: 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 to the Rift.

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::ViewAdjust` 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.

SDK Distortion Rendering

The Oculus SDK provides SDK Distortion Rendering as the recommended path for presenting frames and handling distortion.

With SDK rendering, developers render the scene into one or two render textures, passing these textures into the API. Beyond that point, the Oculus SDK handles the rendering of distortion, calling Present, GPU synchronization, and frame timing.

The following are the steps for SDK rendering:

1. Initialize:
    a. Modify your application window and swap chain initialization code to use the data provided in the struct e.g. Rift resolution etc.
    b. Compute the desired FOV and texture sizes based on `ovrHMDDesc` data.
    c. Allocate textures in an API-specific way.
    d. Use `ovrHmd_ConfigureRendering` to initialize distortion rendering, passing in the necessary API specific device handles, configuration flags, and FOV data.
    e. Under Windows, call `ovrHmd_AttachToWindow` to direct back buffer output from the window to the HMD.
2. Set up frame handling:
    a. Call `ovrHmd_BeginFrame` to start frame processing and obtain timing information.
    b. Perform rendering for each eye in an engine-specific way, rendering into render textures.

c. Call `ovrHmd_EndFrame` (passing in the render textures from the previous step) to swap buffers and present the frame. This function will also handle timewarp, GPU sync, and frame timing.

3. Shutdown:

a. You can use `ovrHmd_ConfigureRendering` with a null value for the `apiConfig` parameter to shut down SDK rendering or change its rendering parameters. Alternatively, you can just destroy the `ovrHmd` object by calling `ovrHmd_Destroy`.

Render Texture Initialization

This section describes the steps involved in initialization.

As a first step, you determine the rendering FOV and allocate the required render target textures. The following code sample shows how the OculusRoomTiny demo does this:

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

   const int eyeRenderMultisample = 1;
   pRendertargetTexture = pRender->CreateTexture(
                                    Texture_RGBA | Texture_RenderTarget |
eyeRenderMultisample,
                                    renderTargetSize.w, renderTargetSize.h,
NULL);
   // The actual RT size may be different due to HW limits.
   renderTargetSize.w = pRendertargetTexture->GetWidth();
   renderTargetSize.h = pRendertargetTexture->GetHeight();
```

The code first determines the render texture size 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 `hmd->DefaultEyeFov`). The function `ovrHmd_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. The sample then calls `CreateTexture` to allocate the texture in an API-specific way. Under the hood, the returned texture object will wrap either a D3D texture handle or OpenGL texture id. Because video hardware may have texture size limitations, we update `renderTargetSize` based on the actually allocated texture size. Although use of a different texture size may affect rendering quality and performance, it should function properly if the viewports are set up correctly. The Frame Rendering section of this guide describes viewport setup.

Configure Rendering

After determining FOV, you can initialize SDK rendering.

To initialize SDK rendering, call `ovrHmd_ConfigureRendering`. This also generates the `ovrEyeRenderDesc` structure that describes all of the details needed to perform stereo rendering.

Note: In client-rendered mode, use the `ovrHmd_GetRenderDesc` call instead.

In addition to the input `eyeFovIn[]` structures, this requires a render-API dependent version of `ovrRenderAPIConfig` that provides API and platform specific interface pointers. The following code shows an example of what this looks like for Direct3D 11:

```
    // Configure D3D11.
    RenderDevice* render = (RenderDevice*)pRender;
    ovrD3D11Config d3d11cfg;
    d3d11cfg.D3D11.Header.API         = ovrRenderAPI_D3D11;
    d3d11cfg.D3D11.Header.RTSize      = Sizei(backBufferWidth, backBufferHeight);
    d3d11cfg.D3D11.Header.Multisample = backBufferMultisample;
    d3d11cfg.D3D11.pDevice            = pRender->Device;
    d3d11cfg.D3D11.pDeviceContext     = pRender->Context;
    d3d11cfg.D3D11.pBackBufferRT      = pRender->BackBufferRT;
    d3d11cfg.D3D11.pSwapChain         = pRender->SwapChain;

    if (!ovrHmd_ConfigureRendering(hmd, &d3d11cfg.Config,
 ovrDistortionCap_Chromatic |

 ovrDistortionCap_TimeWarp |

 ovrDistortionCap_Overdrive,
                                  eyeFov, EyeRenderDesc))
        return(1);
```

With D3D11, `ovrHmd_ConfigureRendering` requires the device, context, back buffer and swap chain pointers. Internally, it uses these to allocate the distortion mesh, shaders, and any other resources necessary to correctly output the scene to the Rift display.

Similar code is used to configure rendering with OpenGL. The following code shows a Windows example:

```
    // Configure OpenGL.
    ovrGLConfig cfg;
    cfg.OGL.Header.API         = ovrRenderAPI_OpenGL;
    cfg.OGL.Header.RTSize      = Sizei(hmd->Resolution.w, hmd->Resolution.h);
    cfg.OGL.Header.Multisample = backBufferMultisample;
    cfg.OGL.Window             = window;
    cfg.OGL.DC                 = dc;

    ovrBool result = ovrHmd_ConfigureRendering(hmd, &cfg.Config, distortionCaps,
                                               eyesFov, EyeRenderDesc);
```

In addition to setting up rendering, starting with Oculus SDK 0.4.0, Windows must call `ovrHmd_AttachToWindow` to direct its swap-chain output to the HMD through the Oculus display driver. This requires a single call:

```
    // Direct rendering from a window handle to the Hmd.
    // Not required if ovrHmdCap_ExtendDesktop flag is set.
    ovrHmd_AttachToWindow(hmd, window, NULL, NULL);
```

With the window attached, we are ready to render to the HMD.

Frame Rendering

When used in the SDK distortion rendering mode, the Oculus SDK handles frame timing, motion prediction, distortion rendering, end frame buffer swap (known as `Present` in Direct3D), and GPU synchronization.

To do this, it makes use of three functions that must be called on the render thread:

- `ovrHmd_BeginFrame`
- `ovrHmd_EndFrame`
- `ovrHmd_GetEyePoses`

As suggested by their names, calls to `ovrHmd_BeginFrame` and `ovrHmd_EndFrame` enclose the body of the frame rendering loop. `ovrHmd_BeginFrame` is called at the beginning of the frame, returning frame timing information in the `ovrFrameTiming` struct. Values within this structure are useful for animation and correct sensor pose prediction. `ovrHmd_EndFrame` should be called at the end of the frame, in the same place that you would typically call Present. This function takes care of the distortion rendering, buffer swap, and GPU synchronization. The function also ensures that frame timing is matched with the video card VSync.

In between `ovrHmd_BeginFrame` and `ovrHmd_EndFrame`, you will render both of the eye views to a render texture. Before rendering each eye, you should get the latest predicted head pose by calling `ovrHmd_GetEyePoses`. This will ensure that each predicted pose is based on the latest sensor data. We also recommend that you use the `ovrHMDDesc::EyeRenderOrder` variable to determine which eye to render first for that HMD, since that can also produce better pose prediction on HMDs with eye-independent scanout.

The `ovrHmd_EndFrame` function submits the eye images for distortion processing. Because the texture data is passed in an API-specific format, the `ovrTexture` structure needs some platform-specific initialization.

The following code shows how `ovrTexture` initialization is done for `D3D11` in OculusRoomTiny:

```
    ovrD3D11Texture EyeTexture[2];

    // Pass D3D texture data, including ID3D11Texture2D and
 ID3D11ShaderResourceView pointers.
    Texture* rtt = (Texture*)pRendertargetTexture;
    EyeTexture[0].D3D11.Header.API            = ovrRenderAPI_D3D11;
    EyeTexture[0].D3D11.Header.TextureSize    = RenderTargetSize;
    EyeTexture[0].D3D11.Header.RenderViewport = EyeRenderViewport[0];
    EyeTexture[0].D3D11.pTexture              = pRendertargetTexture-
>Tex.GetPtr();
    EyeTexture[0].D3D11.pSRView               = pRendertargetTexture-
>TexSv.GetPtr();

    // Right eye uses the same texture, but different rendering viewport.
    EyeTexture[1]                             = EyeTexture[0];
    EyeTexture[1].D3D11.Header.RenderViewport = EyeRenderViewport[1];
```

Alternatively, here is OpenGL code:

```
    ovrGLTexture EyeTexture[2];
    ...
    EyeTexture[0].OGL.Header.API            = ovrRenderAPI_OpenGL;
    EyeTexture[0].OGL.Header.TextureSize    = RenderTargetSize;
    EyeTexture[0].OGL.Header.RenderViewport = eyes[0].RenderViewport;
    EyeTexture[0].OGL.TexId                 = textureId;
```

Note: In addition to specifying the texture related pointers, we are also specifying the rendering viewport. Storing this value within the texture structure that is submitted every frame allows applications to change render target size dynamically, if desired. This is useful for optimizing rendering performance. In the sample code a single render texture is used with each eye mapping to half of the render target size. As a result the same `pTexture` pointer is used for both `EyeTexture` structures but the render viewports are different.

With texture setup complete, you can set up a frame rendering loop as follows:

```
    ovrFrameTiming hmdFrameTiming = ovrHmd_BeginFrame(hmd, 0);

    pRender->SetRenderTarget ( pRendertargetTexture );
    pRender->Clear();

    ovrPosef headPose[2];

    for (int eyeIndex = 0; eyeIndex < ovrEye_Count; eyeIndex++)
    {
        ovrEyeType eye          = hmd->EyeRenderOrder[eyeIndex];
        headPose[eye]           = ovrHmd_GetEyePoses(hmd, eye);

        Quatf      orientation = Quatf(headPose[eye].Orientation);
        Matrix4f   proj        = ovrMatrix4f_Projection(EyeRenderDesc[eye].Fov,
                                                        0.01f, 10000.0f, true);

        // * Test code *
        // Assign quaternion result directly to view (translation is ignored).
        Matrix4f   view = Matrix4f(orientation.Inverted()) *
 Matrix4f::Translation(-WorldEyePos);

        pRender->SetViewport(EyeRenderViewport[eye]);
        pRender->SetProjection(proj);
        pRoomScene->Render(pRender,
 Matrix4f::Translation(EyeRenderDesc[eye].ViewAdjust) * view);
    }

    // Let OVR do distortion rendering, Present and flush/sync.
    ovrHmd_EndFrame(hmd, headPose, eyeTextures);
```

As described earlier, frame logic is enclosed by the begin frame and end frame calls. In this example both eyes share the render target. Rendering is straightforward, although there a few points worth noting:

- We use `hmd->EyeRenderOrder[eyeIndex]` to select the order of eye rendering. Although not required, this can improve the quality of pose prediction.
- The projection matrix is computed based on `EyeRenderDesc[eye].Fov`, which are the same FOV values used for the rendering configuration.
- The view matrix is adjusted by the `EyeRenderDesc[eye].ViewAdjust` vector, which accounts for IPD in meters.
- This sample uses only the Rift orientation component, whereas real applications should make use of position as well. Please refer to the OculusRoomTiny or OculusWorldDemo source code for a more comprehensive example.

Frame Timing

When used in the SDK distortion rendering mode, the Oculus SDK handles frame timing, motion prediction, distortion rendering, end frame buffer swap (known as `Present`in Direct3D), and GPU synchronization.

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. Miscomputing these values 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`. However, it should rarely be necessary because simulation and motion prediction should rely completely on the frame timing values.

Render frame timing is managed at a low level by two functions:

- `ovrHmd_BeginFrameTiming`—called at the beginning of the frame; and returns a set of timing values for the frame.
- `ovrHmd_EndFrameTiming`—implements most of the actual frame vsync tracking logic. It must be called at the end of the frame after swap buffers and GPU Sync.

With SDK Distortion Rendering, `ovrHmd_BeginFrame`and`ovrHmd_EndFrame` call the timing functions internally and do not need to be called explicitly. Regardless, you will still use the`ovrFrameTiming` values returned by `ovrHmd_BeginFrame`to perform motion prediction and sometimes waits.

`ovrFrameTiming`provides the following set of absolute times values associated with the current frame:

`DeltaSeconds`	float	The amount of time passed since the previous frame (useful for animation).
`ThisFrameSeconds`	double	Time that this frame's rendering started.
`TimewarpPointSeconds`	double	Time point, during this frame, when timewarp should start.
`NextFrameSeconds`	double	Time when the next frame's rendering is expected to start.
`ScanoutMidpointSeconds`	double	Midpoint time when this frame will show up on the screen. This can be used to obtain head pose prediction for simulation and rendering.
`EyeScanoutSeconds[2]`	double	Times when each eye of this frame is expected to appear on screen. This is the best pose prediction time to use for rendering each eye.

Some of the timing values are used internally by the SDK and might not need to be used directly by your application. For example, the `EyeScanoutSeconds[]` values are used internally by `ovrHmd_GetEyePoses` to report the predicted head pose when rendering each eye. However, there some cases in which timing values are useful:

- When using timewarp, to ensure the lowest possible latency, the `ovrHmd_EndFrame` implementation will pause internally to wait for the timewarp point. If the application frame rendering finishes early, you might decide to execute other processing to manage the wait time before the `TimewarpPointSeconds` time is reached.
- If both simulation and rendering are performed on the same thread, then simulation might need an earlier head Pose value that is not specific to either eye. This can be obtained by calling `ovrHmd_GetSensorState` with `ScanoutMidpointSeconds` for absolute time.

- `EyeScanoutSeconds[]` values are useful when accessing pose from a non-rendering thread. This is discussed later in this guide.

Client Distortion Rendering

In the client distortion rendering mode, the application applies the distortion to the rendered image and makes the final Present call.

This mode is intended for application developers who want to combine the Rift distortion shader pass with their own post-process shaders for increased efficiency. It is also useful if you want to retain fine control over the entire rendering process. Several API calls are provided which enable this, while hiding much of the internal complexity.

Set Up Rendering

The first step is to create the render texture that the application will render the undistorted left and right eye images to.

The process here is essentially the same as the SDK distortion rendering approach. Use the `ovrHmdDesc`struct to obtain information about the HMD configuration and allocate the render texture (or a different render texture for each eye) in an API-specific way. This was described previously in the Render Texture Initialization section of this document.

The next step is to obtain information regarding how the rendering and distortion should be performed for each eye. This is described using the`ovrEyeRenderDesc` struct. The following table describes the fields:

Field	Type	Description
`Eye`	ovrEyeType	The eye that these values refer to (`ovrEye_Left`or`ovrEye_Right`).
`Fov`	ovrFovPort	The field of view to use when rendering this eye view.
`DistortedViewport`	ovrRecti	Viewport to use when applying the distortion to the render texture.
`PixelsPerTanAngleAtCenter`	ovrVector2f	Density of render texture pixels at the center of the distorted view.
`ViewAdjust`	ovrVector3f	Translation to be applied to the view matrix.

Call ovrHmd_GetRenderDesc for each eye to fill in ovrEyeRenderDesc as follows:

```
// Initialize ovrEyeRenderDesc struct.
ovrFovPort eyeFov[2];

...

ovrEyeRenderDesc eyeRenderDesc[2];

EyeRenderDesc[0] = ovrHmd_GetRenderDesc(hmd, ovrEye_Left, eyeFov[0]);
EyeRenderDesc[1] = ovrHmd_GetRenderDesc(hmd, ovrEye_Right, eyeFov[1]);
```

Set Up Rendering

In client distortion rendering mode, the application is responsible for executing the necessary shaders to apply the image distortion and chromatic aberration correction.

In previous SDK versions, the SDK used a fairly complex pixel shader running on every pixel of the screen. However, after testing many methods, Oculus now recommends rendering a mesh of triangles to perform the corrections. The shaders used are simpler and therefore run faster, especially when you use higher resolutions. The shaders also have a more flexible distortion model that allows us to use higher-precision distortion correction.

OculusRoomTiny is a simple demonstration of how to apply this distortion. The vertex shader looks like the following:

```
float2 EyeToSourceUVScale, EyeToSourceUVOffset;
float4x4 EyeRotationStart, EyeRotationEnd;
float2 TimewarpTexCoord(float2 TexCoord, float4x4 rotMat)
{
    // Vertex inputs are in TanEyeAngle space for the R,G,B channels (i.e. after
 chromatic
    // aberration and distortion). These are now "real world" vectors in
 direction (x,y,1)
    // relative to the eye of the HMD. Apply the 3x3 timewarp rotation to these
 vectors.
    float3 transformed = float3( mul ( rotMat, float4(TexCoord.xy, 1, 1) ).xyz);

    // Project them back onto the Z=1 plane of the rendered images.
    float2 flattened = (transformed.xy / transformed.z);

    // Scale them into ([0,0.5],[0,1]) or ([0.5,0],[0,1]) UV lookup space
 (depending on eye)
    return(EyeToSourceUVScale * flattened + EyeToSourceUVOffset);
}

void main(in float2 Position    : POSITION,    in float timewarpLerpFactor :
 POSITION1,
          in float Vignette     : POSITION2,   in float2 TexCoord0         :
 TEXCOORD0,
          in float2 TexCoord1   : TEXCOORD1,   in float2 TexCoord2         :
 TEXCOORD2,
          out float4 oPosition  : SV_Position, out float2 oTexCoord0       :
 TEXCOORD0,
          out float2 oTexCoord1 : TEXCOORD1,   out float2 oTexCoord2       :
 TEXCOORD2,
          out float  oVignette  : TEXCOORD3)
{
    float4x4 lerpedEyeRot = lerp(EyeRotationStart, EyeRotationEnd,
 timewarpLerpFactor);
    oTexCoord0  = TimewarpTexCoord(TexCoord0,lerpedEyeRot);
    oTexCoord1  = TimewarpTexCoord(TexCoord1,lerpedEyeRot);
    oTexCoord2  = TimewarpTexCoord(TexCoord2,lerpedEyeRot);
    oPosition = float4(Position.xy, 0.5, 1.0);
    oVignette = Vignette;   /* For vignette fade */
}
```

The position XY data is already in Normalized Device Coordinates (NDC) space (-1 to +1 across the entire framebuffer). Therefore, the vertex shader simply adds a 1 to W and a default Z value (which is unused because depth buffering is not enabled during distortion correction). There are no other changes.

`EyeToSourceUVScale` and `EyeToSourceUVOffset` are used to offset the texture coordinates based on how the eye images are arranged in the render texture.

The pixel shader is as follows:

```
Texture2D Texture   : register(t0);
SamplerState Linear : register(s0);

float4 main(in float4 oPosition  : SV_Position, in float2 oTexCoord0 : TEXCOORD0,
            in float2 oTexCoord1 : TEXCOORD1, in float2 oTexCoord2 : TEXCOORD2,
            in float oVignette : TEXCOORD3)   : SV_Target
{
    // 3 samples for fixing chromatic aberrations
    float R = Texture.Sample(Linear, oTexCoord0.xy).r;
    float G = Texture.Sample(Linear, oTexCoord1.xy).g;
    float B = Texture.Sample(Linear, oTexCoord2.xy).b;
    return (oVignette*float4(R,G,B,1));
}
```

The pixel shader samples the red, green, and blue components from the source texture where specified, and combines them with a shading. The shading is used at the edges of the view to give a smooth fade-to-black effect rather than an abrupt cut-off. A sharp edge triggers the motion-sensing neurons at the edge of our vision and can be very distracting. Using a smooth fade-to-black reduces this effect substantially.

As you can see, the shaders are very simple, and all the math happens during the generation of the mesh positions and UV coordinates. To generate the distortion mesh, call `ovrHmd_CreateDistortionMesh`. This function generates the mesh data in the form of an indexed triangle list, which you can then convert to the data format required by your graphics engine. It is also necessary to call `ovrHmd_GetRenderScaleAndOffset` to retrieve values for the constants `EyeToSourceUVScale` and `EyeToSourceUVOffset` used in the vertex shader. For example, in OculusRoomTiny:

```
//Generate distortion mesh for each eye
for ( int eyeNum = 0; eyeNum < 2; eyeNum++ )
{
    // Allocate & generate distortion mesh vertices.
    ovrDistortionMesh meshData;
    ovrHmd_CreateDistortionMesh(hmd,
                                eyeRenderDesc[eyeNum].Eye,
 eyeRenderDesc[eyeNum].Fov,
                                distortionCaps, &meshData);

    ovrHmd_GetRenderScaleAndOffset(eyeRenderDesc[eyeNum].Fov,
                                   textureSize, viewports[eyeNum],
                                   (ovrVector2f*)
 DistortionData.UVScaleOffset[eyeNum]);

    // Now parse the vertex data and create a render ready vertex buffer from it
    DistortionVertex * pVBVerts = (DistortionVertex*)OVR_ALLOC(
                                       sizeof(DistortionVertex) *
 meshData.VertexCount );
    DistortionVertex * v        = pVBVerts;
    ovrDistortionVertex * ov    = meshData.pVertexData;
    for ( unsigned vertNum = 0; vertNum < meshData.VertexCount; vertNum++ )
    {
        v->Pos.x = ov->Pos.x;
        v->Pos.y = ov->Pos.y;
        v->TexR  = (*(Vector2f*)&ov->TexR);
        v->TexG  = (*(Vector2f*)&ov->TexG);
```

```
        v->TexB  = (*(Vector2f*)&ov->TexB);
        v->Col.R = v->Col.G = v->Col.B = (OVR::UByte)( ov->VignetteFactor *
 255.99f );
        v->Col.A = (OVR::UByte)( ov->TimeWarpFactor * 255.99f );
        v++; ov++;
    }

    //Register this mesh with the renderer
    DistortionData.MeshVBs[eyeNum] = *pRender->CreateBuffer();
    DistortionData.MeshVBs[eyeNum]->Data ( Buffer_Vertex, pVBVerts,
                                        sizeof(DistortionVertex) *
 meshData.VertexCount );

    DistortionData.MeshIBs[eyeNum] = *pRender->CreateBuffer();
    DistortionData.MeshIBs[eyeNum]->Data ( Buffer_Index, meshData.pIndexData,
                                        sizeof(unsigned short) *
 meshData.IndexCount );

    OVR_FREE ( pVBVerts );
    ovrHmd_DestroyDistortionMesh( &meshData );
}
```

For extra performance, this code can be merged with existing post-processing shaders, such as exposure correction or color grading. However, to ensure that the shader and mesh still calculate the correct distortion, you should do this before and after pixel-exact checking. It is very common to get something that looks plausible, but even a few pixels of error can cause discomfort for users.

Game Rendering Loop

In client distortion rendering mode, the application is responsible for executing the necessary shaders to apply the image distortion and chromatic aberration correction.

The following code demonstrates this:

```
    ovrHmd hmd;
    ovrPosef headPose[2];

    ovrFrameTiming frameTiming = ovrHmd_BeginFrameTiming(hmd, 0);

    pRender->SetRenderTarget ( pRendertargetTexture );
    pRender->Clear();

    for (int eyeIndex = 0; eyeIndex < ovrEye_Count; eyeIndex++)
    {
        ovrEyeType eye = hmd->EyeRenderOrder[eyeIndex];
        headPose[eye] = ovrHmd_GetEyePoses(hmd, eye);

        Quatf orientation = Quatf(eyePose.Orientation);
        Matrix4f proj = ovrMatrix4f_Projection(EyeRenderDesc[eye].Fov,
                                               0.01f, 10000.0f, true);

        // * Test code *
        // Assign quaternion result directly to view (translation is ignored).
        Matrix4f view = Matrix4f(orientation.Inverted()) *
 Matrix4f::Translation(-WorldEyePosition);

        pRender->SetViewport(EyeRenderViewport[eye]);
```

```
        pRender->SetProjection(proj);

        pRoomScene->Render(pRender,
Matrix4f::Translation(EyeRenderDesc[eye].ViewAdjust) * view);
    }

    // Wait till time-warp point to reduce latency.
    ovr_WaitTillTime(frameTiming.TimewarpPointSeconds);


    // Prepare for distortion rendering.
    pRender->SetRenderTarget(NULL);
    pRender->SetFullViewport();
    pRender->Clear();

    ShaderFill distortionShaderFill(DistortionData.Shaders);
    distortionShaderFill.SetTexture(0, pRendertargetTexture);
    distortionShaderFill.SetInputLayout(DistortionData.VertexIL);

    for (int eyeIndex = 0; eyeIndex < 2; eyeIndex++)
    {
        // Setup shader constants
        DistortionData.Shaders->SetUniform2f("EyeToSourceUVScale",
        DistortionData.UVScaleOffset[eyeIndex][0].x,
DistortionData.UVScaleOffset[eyeIndex][0].y);
        DistortionData.Shaders->SetUniform2f("EyeToSourceUVOffset",
        DistortionData.UVScaleOffset[eyeIndex][1].x,
DistortionData.UVScaleOffset[eyeIndex][1].y);

        ovrMatrix4f timeWarpMatrices[2];
        ovrHmd_GetEyeTimewarpMatrices(hmd, (ovrEyeType) eyeIndex,
headPose[eyeIndex],
                                      timeWarpMatrices);

        DistortionData.Shaders->SetUniform4x4f("EyeRotationStart",
Matrix4f(timeWarpMatrices[0]));
        DistortionData.Shaders->SetUniform4x4f("EyeRotationEnd",
Matrix4f(timeWarpMatrices[1]));

        // Perform distortion
        pRender->Render(&distortionShaderFill,
                        DistortionData.MeshVBs[eyeIndex],
DistortionData.MeshIBs[eyeIndex]);
    }

    pRender->Present( VSyncEnabled );
    pRender->WaitUntilGpuIdle();  //for lowest latency
    ovrHmd_EndFrameTiming(hmd);
```

Multi-Threaded Engine Support

Modern applications, particularly video game engines, often distribute processing over multiple threads.

When integrating the Oculus SDK, make sure to call the API functions in the appropriate manner and to manage timing correctly for accurate HMD pose prediction. This section describes two multi-threaded scenarios that you can use. Hopefully the insight provided will enable you to handle these issues correctly, even if your application's multi-threaded approach differs from those presented. As always, if you require guidance, please visit developer.oculusvr.com.

One of the factors that dictates API policy is our use of the application rendering API inside of the SDK (e.g., Direct3D). Generally, rendering APIs impose their own multi-threading restrictions. For example, it is common to call core rendering functions from the same thread that was used to create the main rendering device. As a result, these limitations impose restrictions on the use of the Oculus API.

These rules apply:

- All tracking interface functions are thread-safe, allowing the tracking state to be sampled from different threads.
- All rendering functions including the configure and frame functions, are not thread-safe. You can use `ConfigureRendering` on one thread and handle frames on another thread, but you must perform explicit synchronization because functions that depend on configured state are not reentrant.
- All of the following calls must be done on the render thread (the thread used by the application to create the main rendering device):
  - `ovrHmd_BeginFrame` (or `ovrHmd_BeginFrameTiming`)
  - `ovrHmd_EndFrame`
  - `ovrHmd_GetEyePoses`
  - `ovrHmd_GetEyeTimewarpMatrices`

Update and Render on Different Threads

It is common for video game engines to separate the actions of updating the state of the world and rendering a view of it.

In addition, executing these on separate threads (mapped onto different cores) allows them to execute concurrently and use more of the available CPU resources. Typically the update operation executes AI logic and player character animation which, in VR, requires the current headset pose. For the rendering operation, it needs to determine the the left and right eye view transform to render, which also needs the head pose. The main difference between the two is the level of accuracy required. The AI logic only requires a moderately accurate head pose. For rendering, it is critical that the head pose is very accurate and that the image is displayed on the screen matches as closely as possible. The SDK employs two techniques to ensure this. The first technique is prediction, where the application can request the predicted head pose at a future point in time. The `ovrFrameTiming` struct provides accurate timing information for this purpose. The second technique is Timewarp, where we wait until a very short time before presenting the next frame to the display, perform another head pose reading, and re-project the rendered image to take account of any changes in predicted head pose that occurred since the head pose was read during rendering.

Generally, the closer we are to the time that the frame is displayed, the better the prediction of head pose at that time will be. It is perfectly fine to read head pose several times during the render operation, each time passing in the same future time that the frame will display (in the case of calling `ovrHmd_GetFrameTiming`), and each time receiving a more accurate estimate of the future head pose. However, for Timewarp to function correctly, you must pass the actual head pose that was used to determine the view matrices when you make the call to `ovrHmd_EndFrame` (for SDK distortion rendering) or `ovrHmd_GetEyeTimewarpMatrices` (for client distortion rendering).

When obtaining the head pose for the update operation, it is usually sufficient to get the current head pose (rather than the predicted one). This can be obtained with:

```
ovrTrackingState ts  = ovrHmd_GetTrackingState(hmd, ovr_GetTimeInSeconds());
```

The next section describes a scenario that uses the final head pose to render from a non-render thread, which requires prediction.

Render 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). The difference between this and the former scenario is that the non-render thread needs to obtain accurate predictions of head pose.

To do this, it needs an accurate estimate of the time until the frame being processed appears on the screen. Furthermore, due to the asynchronous nature of this approach, while the render thread is rendering a frame, the main thread might be processing the next frame. As a result, the application must associate the head poses that were obtained in the main thread with the frame, such that when that frame is being rendered by the render thread, the application is able to pass the correct head pose transforms into `ovrHmd_EndFrame` or `ovrHmd_GetEyeTimewarpMatrices`. For this purpose, we introduce the concept of a `frameIndex` which is created by the application, incremented each frame, and passed into several of the API functions.

Essentially, there are three additional things to consider:

1. The main thread needs to assign a frame index to the current frame being processed for rendering. This is used in the call to `ovrHmd_GetFrameTiming` to return the correct timing for pose prediction etc.
2. The main thread should call the thread safe function `ovrHmd_GetTrackingState` with the predicted time value.
3. When the rendering commands generated on the main thread are executed on the render thread, pass in the corresponding value of `frameIndex` when calling `ovrHmd_BeginFrame`. Similarly, when calling `ovrHmd_EndFrame`, pass in the actual pose transform used when that frame was processed on the main thread (from the call to `ovrHmd_GetTrackingState`).

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.
    ovrFrameTiming frameTiming = ovrHmd_GetFrameTiming(hmd, frameIndex);

    // Get the corresponding predicted pose state.
    ovrTrackingState state = ovrHmd_GetTrackingState(hmd,
 frameTiming.ScanoutMidpointSeconds);

    ovrPosef pose = state.HeadPose.ThePose;

    SetFrameHMDData(frameIndex, pose);
```

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

    ...

}

void RenderThreadProcessing()
{

    int frameIndex;
    ovrPosef pose;

    GetFrameHMDData(&frameIndex, &pose);

    // Call begin frame and pass in frameIndex.
    ovrFrameTiming hmdFrameTiming = ovrHmd_BeginFrame(hmd, frameIndex);

    // Execute actual rendering to eye textures.
    ovrTexture eyeTexture[2]);

    ...

    ovrPosef renderPose[2] = {pose, pose};

    ovrHmd_EndFrame(hmd, pose, eyeTexture);
}
```

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.

Advanced Rendering Configuration

The SDK has been 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 `ovrHmd_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 = ovrHmd_GetFovTextureSize(hmd, ovrEye_Left,
                                   hmd->DefaultEyeFov[0], pixelsPerDisplayPixel);
    Sizei recommenedTex1Size = ovrHmd_GetFovTextureSize(hmd, ovrEye_Right,
                                   hmd->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] = { hmd->DefaultEyeFov[0], hmd->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;
```

IFor SDK distortion rendering, this data is passed into `ovrHmd_ConfigureRendering` as follows (code shown is for the D3D11 API):

```
    ovrEyeRenderDesc eyeRenderDesc[2];

    ovrBool result = ovrHmd_ConfigureRendering(hmd, &d3d11cfg.Config,
                                               ovrDistortion_Chromatic |
ovrDistortion_TimeWarp,
                                               eyeFov, eyeRenderDesc);
```

Alternatively, in the case of client distortion rendering, you would call `ovrHmd_GetRenderDesc` as follows:

```
    ovrEyeRenderDesc eyeRenderDesc[2];

    eyeRenderDesc[0] = ovrHmd_GetRenderDesc(hmd, ovrEye_Left, eyeFov[0]);
    eyeRenderDesc[1] = ovrHmd_GetRenderDesc(hmd, ovrEye_Right, eyeFov[1]);
```

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 `ovrHmd_EndFrame` (in the case of SDK rendering using the `ovrTexture` structure) or `ovrHmd_GetRenderScaleAndOffset` (in the case of client rendering). However, using `ovrHmd_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.

Advanced Rendering Configuration

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 centerline as the right edge.

This is because humans, as well as the Rift, have a wider FOV when looking outwards. When you look inwards, towards your nose, 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 = hmd->DefaultEyeFov[ovrEye_Left];

ovrFovPort maxLeftFOV = hmd->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 36.

The chosen FOV values should be passed into `ovrHmd_ConfigureRendering` for SDK side distortion or `ovrHmd_GetRenderDesc` for client distortion rendering.

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. Here is some code for handling this most restrictive case:

```
ovrFovPort fovLeft = hmd->DefaultEyeFov[ovrEye_Left];
ovrFovPort fovRight = hmd->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 = ovrHmd_GetFovTextureSize(hmd, ovrEye_Left,
                                                     fovBoth,
 pixelsPerDisplayPixel);
Sizei recommenedTex1Size = ovrHmd_GetFovTextureSize(hmd, 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 fairly modest 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 render target that is 2000x1056 pixels in size, surprisingly large!

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 gives 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 games can run at this native resolution at full speed, and 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 `ovrHmd_GetFovTextureSize` function. This could also be based on a slider presented in the applications render settings. This determines the relative size of render targetpixels 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 = ovrHmd_GetFovTextureSize(hmd, 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, and 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 = hmd->DefaultEyeFov[ovrEye_Left];
ovrFovPort defaultFovRight = hmd->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 = ovrHmd_GetFovTextureSize(hmd, ovrEye_Left  newFovLeft,
                                                    pixelsPerDisplayPixel);
Sizei recommenedTex1Size = ovrHmd_GetFovTextureSize(hmd, 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 ability.

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

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 `ovrHmd_EndFrame` or `ovrHmd_GetRenderScaleAndOffset`.
- Render the scene only once to this 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 should 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!

Oculus API Changes

This section describes API changes for each version release.

Changes Since Release 0.2

The Oculus API has been significantly redesigned since the 0.2.5 release, with the goals of improving ease of use, correctness and supporting a new driver model.

The following is the summary of changes in the API:

- All of the HMD and sensor interfaces have been organized into a C API. This makes it easy to bind from other languages.
- The new Oculus API introduces two distinct approaches to rendering distortion: SDK Rendered and Client Rendered. As before, the application is expected to render stereo scenes onto one or more render targets. With the SDK rendered approach, the Oculus SDK then takes care of distortion rendering, frame present, and timing within the SDK. This means that developers don't need to setup pixel and vertex shaders or worry about the details of distortion rendering, they simply provide the device and texture pointers to the SDK. In client rendered mode, distortion rendering is handled by the application as with previous versions of the SDK. SDK Rendering is the preferred approach for future versions of the SDK.
- The method of rendering distortion in client rendered mode is now mesh based. The SDK returns a mesh which includes vertices and UV coordinates which are then used to warp the source render target image to the final buffer. Mesh based distortion is more efficient and flexible than pixel shader approaches.
- The Oculus SDK now keeps track of game frame timing and uses this information to accurately predict orientation and motion.
- A new technique called Timewarp is introduced to reduce motion-to-photon latency. This technique re-projects the scene to a more recently measured orientation during the distortion rendering phase.

The table on the next page briefly summarizes differences between the 0.2.5 and 0.4 API versions.

Functionality	0.2 SDK APIs	0.4 SDK C APIs
`Initialization`	`OVR::System::Init`, `DeviceManager`, `HMDDevice`, `HMDInfo`.	`ovr_Initialize`, `ovrHmd_Create`, `ovrHmd` handle and `ovrHmdDesc`.
`Sensor Interaction`	`OVR::SensorFusion` class, with `GetOrientation` returning `Quatf`. Prediction amounts are specified manually relative to the current time.	`ovrHmd_ConfigureTracking`, `ovrHmd_GetTrackingState` returning `ovrTrackingState`. `ovrHmd_GetEyePoses` returns head pose based on correct timing.
`Rendering Setup`	`Util::Render::StereoConfig` helper class creating `StereoEyeParams`, or manual setup based on members of `HMDInfo`.	`ovrHmd_ConfigureRendering` populates `ovrEyeRenderDesc` based on the field of view. Alternatively, `ovrHmd_GetRenderDesc` supports rendering setup for client distortion rendering.
`Distortion Rendering`	App-provided pixel shader based on distortion coefficients.	Client rendered: based on the distortion mesh returned by `ovrHmd_CreateDistortionMesh`. (or) SDK rendered:

Functionality	0.2 SDK APIs	0.4 SDK C APIs
		done automatically in `ovrHmd_EndFrame`.
`Frame Timing`	Manual timing with current-time relative prediction.	Frame timing is tied to vsync with absolute values reported by `ovrHmd_BeginFrame` or `ovr_BeginFrameTiming`.

Changes Since Release 0.3

A number of changes were made to the API since the 0.3.2 Preview release.

These are summarized as follows:

- Removed the method `ovrHmd_GetDesc`. The `ovrHmd` handle is now a pointer to a `ovrHmdDesc` struct.
- The sensor interface has been simplified. Your application should now call `ovrHmd_ConfigureTracking` at initialization and `ovrHmd_GetTrackingState` or `ovrHmd_GetEyePoses` to get the head pose.
- `ovrHmd_BeginEyeRender` and `ovrHmd_EndEyeRender` have been removed. You should now use `ovrHmd_GetEyePoses` to determine predicted head pose when rendering each eye. Render poses and `ovrTexture` info is now passed into `ovrHmd_EndFrame` rather than `ovrHmd_EndEyeRender`.
- `ovrSensorState` struct is now `ovrTrackingState`. The predicted pose Predicted is now named `HeadPose`. `CameraPose` and `LeveledCameraPose` have been added. Raw sensor data can be obtained through `RawSensorData`.
- `ovrSensorDesc` struct has been merged into ovrHmdDesc.
- Addition of `ovrHmd_AttachToWindow`. This is a platform specific function to specify the application window whose output will be displayed on the HMD. Only used if the `ovrHmdCap_ExtendDesktop` flag is false.
- Addition of `ovr_GetVersionString`. Returns a string representing the libOVR version.

There have also been a number of minor changes:

- Renamed `ovrSensorCaps` struct to `ovrTrackingCaps`.
- Addition of `ovrHmdCaps::ovrHmdCap_Captured` flag. Set to `true` if the application captured ownership of the HMD.
- Addition of `ovrHmdCaps::ovrHmdCap_ExtendDesktop` flag. The display driver is in compatibility mode (read only).
- Addition of `ovrHmdCaps::ovrHmdCap_NoMirrorToWindow` flag. Disables mirroring of HMD output to the window. This may improve rendering performance slightly (only if 'Extend-Desktop' is off).
- Addition of `ovrHmdCaps::ovrHmdCap_DisplayOff` flag. Turns off HMD screen and output (only if 'ExtendDesktop' is off).
- Removed `ovrHmdCaps::ovrHmdCap_LatencyTest` flag. Was used to indicate support of pixel reading for continuous latency testing.
- `AdditionofovrDistortionCaps::ovrDistortionCap_Overdriveflag. Overdrivebrightness` transitions to reduce artifacts on DK2 displays.
- Addition of `ovrStatusBits::ovrStatus_CameraPoseTracked` flag. Indicates that the camera pose is successfully calibrated.

Display Device Management

This section was original written when management of the Rift display as part of the desktop was the only option.

With the introduction of the Oculus Display Driver, the standard approach is now to select Direct HMD Access from Apps mode and let the SDK manage the device. However, until the driver matures, it might still be necessary to switch to one of the legacy display modes which requires managing the display as part of the desktop. For this reason, this section is still in this guide for reference purposes.

Display Identification

Display devices identify themselves and their capabilities using Extended Display Identification Data (EDID).

The display within the Oculus Rift interacts with the system in the same way as a typical PC monitor. It also provides EDID information which identifies it as having a OVR manufacturer code, a Rift DK1model, and support for several display resolutions (including the native 1280 × 800 at 60Hz).

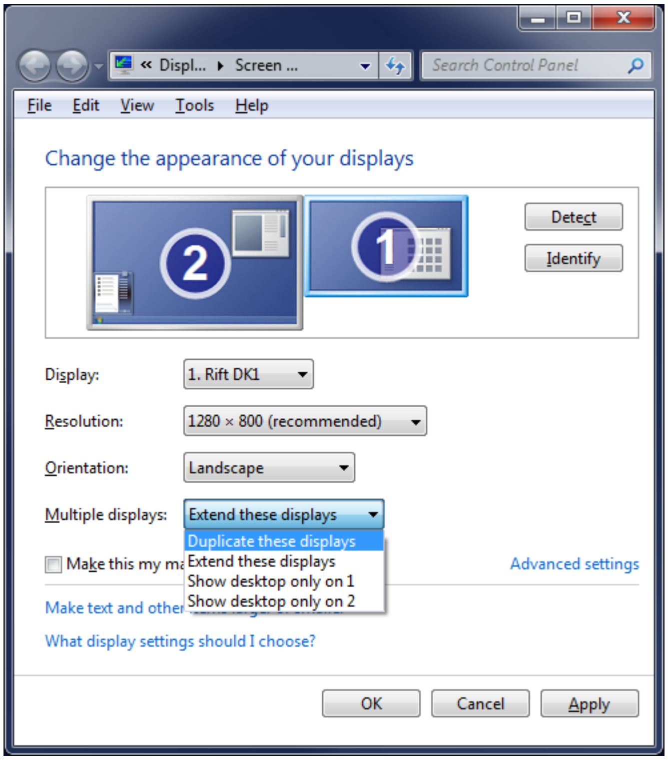
Display Configuration

After connecting a Rift to the PC, you can modify the display settings through the Windows Control Panel.

In Windows 7, select Control Panel -> All Control Panel Items -> Display -> Screen Resolution. In MacOS, select System Preferences -> Displays. In Ubuntu Linux, select System Settings -> Displays.

The following figure shows how to configure a Windows PC with the Rift display and a monitor. In this configuration, there are four available modes: duplicate mode, extended mode, standalone mode the monitor, and standalone mode for the Rift.

Figure 7: Windows Screen Resolution Dialog Box



Duplicate Display Mode

In duplicate display mode, the same portion of the desktop is shown on both displays; each device uses the same resolution and orientation settings. Your computer attempts to choose a resolution that is supported by both displays, while favoring the native resolutions described in the EDID information reported by the displays. Duplicate mode can be useful for configuring the Rift. However, it suffers from vsync issues.

Extended Display Mode

In extended mode, the displays show different portions of the desktop. You can use the Control Panel to set the desired resolution and orientation independently for each display. Extended mode suffers from shortcomings due to the fact that the Rift is not a viable way to interact with the desktop. Nevertheless, it is the current recommended configuration option. For more information about the shortcomings, see *Rift Display Considerations* on page 45.

Standalone Display Mode

In standalone mode, the desktop is displayed on either the monitor or the Rift. It is possible to configure the Rift as the sole display, but this becomes impractical when interacting with the desktop.

Selecting A Display Device

Reading of EDID information from display devices can occasionally be slow and unreliable. In addition, EDID information may be cached, leading to problems with old data.

As a result, display devices can become associated with incorrect display names and resolutions, with arbitrary delays before the information becomes current.

Because of these issues, we adopt an approach that attempts to identify the Rift display name among the attached display devices. However, we do not require that it is found for an HMD device to be created using the API.

If the Rift display device is not detected, but the Rift is detected through USB, then an empty display name string is returned. When this happens, your application can attempt to locate it using additional information, such as the display resolution.

In general, due to the uncertainty associated with identifying the Rift display device, you might want to incorporate functionality into your application that allows the user to choose the display manually, such as from a drop-down list of display devices. One possible cause of the this scenario, aside from EDID issues, is that the user failed to plug in the Rift video cable. Make sure you have appropriate assistance within your application to help users troubleshoot an incorrectly connected Rift device.

Windows

If there are no EDID issues and we detect the Rift display device successfully, then we return the display name corresponding to that device, for example `\\.\DISPLAY1\Monitor0`.

To display the video output on the Rift, the application needs to match the display name determined above to an object used by the rendering API. Unfortunately, each rendering system used on Windows (OpenGL, Direct3D 9, Direct3D 10-11) uses a different approach. OpenGL uses the display device name returned in HMDInfo, Direct3D requires the display name must be matched against a list of displays returned by D3D. You can use the DesktopX and DesktopY members of HMDInfo to position a window on the Rift if you do not want to use fullscreen mode.

When using Direct3D 10 or 11, the following code shows how to obtain an IDXGIOutput interface using the DXGI API:

```
IDXGIOutput* searchForOculusDisplay(char* oculusDisplayName)
{
    IDXGIFactory* pFactory;
    CreateDXGIFactory(__uuidof(IDXGIFactory), (void**)(&pFactory));

    UInt32 adapterIndex = 0;
    IDXGIAdapter* pAdapter;

    // Iterate through adapters.
    while (pFactory->EnumAdapters(adapterIndex, &pAdapter) !=
                                DXGI_ERROR_NOT_FOUND)
    {
        UInt32 outputIndex = 0;
        IDXGIOutput* pOutput;

        // Iterate through outputs.
        while (pAdapter->EnumOutputs(outputIndex, &pOutput) !=
                                DXGI_ERROR_NOT_FOUND)
        {
            DXGI_OUTPUT_DESC outDesc;
            pOutput->GetDesc(&outDesc);
            char* outputName = outDesc.DeviceName;

            // Try and match the first part of the display name.
            // For example an outputName of "\\.\DISPLAY1" might
            // correspond to a displayName of "\\.\DISPLAY1\Monitor0".
            // If two monitors are setup in 'duplicate' mode then they will
            // have the same 'display' part in their display name.
```

```
            if (strstr(oculusDisplayName, outputName) == oculusDisplayName)
            {
                return pOutput;
            }
        }
    }

    return NULL;
}
```

After you've successfully obtained the IDXGIOutput interface, you can set your Direct3D swap-chain to render to it in fullscreen mode using the following:

```
IDXGIOutput* pOculusOutputDevice = searchForOculusDisplay(oculusDisplayName);
pSwapChain->SetFullscreenState(TRUE, pOculusOutputDevice);
```

When using Direct3D 9, you must create the Direct3DDevice with the "adapter" number corresponding to the Rift (or other display that you want to output to). This next function shows how to find the adapter number:

```
unsigned searchForOculusDisplay(const char* oculusDisplayName)
{
for (unsigned Adapter=0; Adapter < Direct3D->GetAdapterCount(); Adapter++)
{
MONITORINFOEX Monitor;
Monitor.cbSize = sizeof(Monitor);
if (::GetMonitorInfo(Direct3D->GetAdapterMonitor(Adapter), &Monitor) &&
 Monitor.szDevice[0])
{
DISPLAY_DEVICE DispDev;
memset(&DispDev, 0, sizeof(DispDev));
DispDev.cb = sizeof(DispDev);
if (::EnumDisplayDevices(Monitor.szDevice, 0, &DispDev, 0))
{
if (strstr(DispDev.DeviceName, oculusDisplayName))
{
return Adapter;
}
}
}
}
return D3DADPATER_DEFAULT;
}
```

Unfortunately, you must re-create the device to switch fullscreen displays. You can use a fullscreen device for windowed mode without recreating it (although it still has to be reset).

MacOS

When running on MacOS, the `ovrHMDDesc` struct contains a `DisplayID` which you should use to target the Rift in fullscreen. If the Rift is connected (and detected by libovr), its DisplayId is stored, otherwise DisplayId contains 0. The following is an example of how to use this to make a Cocoa `NSView` fullscreen on the Rift:

```
CGDirectDisplayId RiftDisplayId = (CGDirectDisplayId) hmd->DisplayId;
```

```
NSScreen* usescreen = [NSScreen mainScreen];
NSArray* screens = [NSScreen screens];
for (int i = 0; i < [screens count]; i++)
{
    NSScreen* s = (NSScreen*)[screens objectAtIndex:i];
    CGDirectDisplayID disp = [NSView displayFromScreen:s];

    if (disp == RiftDisplayId)
        usescreen = s;
}

[View enterFullScreenMode:usescreen withOptions:nil];
```

You can use the `WindowPos` member of `ovrHMDDesc` to position a window on the Rift if you do not want to use fullscreen mode.

Rift Display Considerations

There are several considerations when it comes to managing the Rift display on a desktop OS.

Duplicate Mode VSync

In duplicate monitor mode, it is common for the supported video timing information to be different across the participating monitors, even when displaying the same resolution. When this occurs, the video scans on each display will be out of sync and the software vertical sync mechanism will be associated with only one of the displays. In other words, swap-chain buffer switches (for example following a `glSwapBuffers` or Direct3D `Present` call) only occurs at the correct time for one of the displays, and 'tearing' will occur on the other display. In the case of the Rift, tearing is very distracting, and so ideally we'd like to force it to have vertical sync priority. Unfortunately, the ability to do this is not something currently exposed in system APIs.

Extended Mode Problems

When extended display mode is used in conjunction with the Rift, the desktop partly extebds onto the regular monitors and partly onto the Rift. Since the Rift displays different portions of the screen to the left and right eyes, it is not suited to displaying the desktop in a usable form, and confusion may arise if icons or windows find their way onto the portion of the desktop displayed on the Rift.

Observing Rift Output on a Monitor

Sometimes, it can be useful to see the same video output on the Rift and on an external monitor. This is particularly useful when demonstrating the device to a new user, or during application development. One way to achieve this is through the use of duplicate monitor mode as described above. However, we don't currently recommend this approach due to the vertical sync priority issue. An alternative approach is through the use of a DVI or HDMI splitter. These take the video signal coming from the display adapter and duplicate it such that it can be fed to two or more display devices simultaneously. Unfortunately, this can also cause problems depending on how the EDID data is managed. Specifically, with several display devices connected to a splitter, which EDID information should be reported back to the graphics adapter? Low cost HDMI splitters have been found to exhibit unpredictable behavior. Typically they pass on EDID information from one of the attached devices, but exactly how the choice is determined is often unknown. Higher cost devices may have explicit schemes (for example they report the EDID from the display plugged into output port one), but these can cost more than the Rift itself! Generally, the use of third party splitters and video switching boxes means that Rift EDID data may not be reliably reported to the OS and libovr will not be able to identify the Rift.

Windows: Direct3D Enumeration

As described above, the nature of the Rift display means that it is not suited to displaying the Windows desktop. As a result you might be inclined to set standalone mode for your regular monitor to remove the Rift from the list of devices displaying the Windows desktop. Unfortunately, this also causes the device to no longer be enumerated when querying for output devices during Direct3D setup. As a result, the only viable option currently is to use the Rift in extended display mode.

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,2 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”).

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.

To set up the gamepad for Mac:

1. Put the controller into 'DirectInput' mode by moving the switch on the front of the controller to the 'D' 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. OSX should recognize the controller and install any necessary drivers automatically.

Sony PlayStation DUALSHOCK3 Controller

To set up the controller for Mac:

1. Turn off any nearby PS3 consoles.
2. Go to System Preferences -> Bluetooth.
3. Make sure the 'On' and 'Discoverable' check boxes are checked.
4. Plug the controller into the Mac using the USB cable.
5. Press the 'PS' Button in the middle of the controller for 3 seconds and then remove the USB cable.
6. After removing the cable, the controller should immediately appear in the device list. If a dialog appears requesting a passcode enter 'xxxx' and then press Pair.
7. Click on the gear symbol beneath the list of Bluetooth devices and select Add to Favorites.
8. Click on the gear symbol once more and select Update Services.
9. Quickly turn Bluetooth off and then immediately back on again.
10. Press the 'PS' Button on the controller. The controller status should now appear as 'Connected'.

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 tracking camera 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.

Sensor Fusion Details

The most important part of sensor fusion is the integration of angular velocity data from the gyroscope.

In each tiny interval of time, a measurement of the angular velocity arrives:

$$\omega = (\omega_x, \omega_y, \omega_z).$$

In each tiny interval of time, a measurement of the angular velocity arrives:

$$\ell = \sqrt{\omega_x^2 + \omega_y^2 + \omega_z^2}$$