OCULUS VR, INC

SOFTWARE DOCUMENTATION

SDK Overview

Authors:
Michael Antonov
Nate Mitchell
Andrew Reisse
Lee Cooper
Steve LaValle

Date:
March 28, 2013

Oculus VR, Inc. 19800 MacArthur Blvd Suite 450 Irvine, CA 92612

Contents

1 Introduction

Thanks for downloading the Oculus Software Development Kit (SDK)!

This document will detail how to install, configure, and use the Oculus SDK. The Oculus SDK includes all of the components that developers need to integrate the Oculus Rift with their game engine or application. The core of the SDK is made up of source code and binary libraries. The Oculus SDK also includes documentation, samples, and tools to help developers get started.

This document focuses on the C++ API of the Oculus SDK. Integration with the Unreal Engine 3 (UE3) and Unity game engine is available as follows:

- Unity integration is available as a separate package from the Oculus Developer Center.
- Unreal Engine 3 integration is also available as a separate package from the Oculus Developer Center. You will need a full UE3 license to access the version of Unreal with Oculus integration. If you have a full UE3 license, you can email support@oculusvr.com to be granted download access.

2 Oculus Rift Hardware Setup

In addition to the Oculus SDK, you will also need the hardware provided by the Oculus Rift Development Kit (DK). The DK includes an Oculus Rift development headset (Rift), control box, required cabling, and additional pairs of lenses for different vision characteristics.

2.1 Display Specifications

- 7 inch diagonal viewing area
- 1280×800 resolution (720p). This is split between both eyes, yielding 640×800 per eye.
- 64mm fixed distance between lens centers
- 60Hz LCD panel
- DVI-D Single Link
- HDMI 1.3+
- USB 2.0 Full Speed+

2.2 Tracker Specifications

- Up to 1000Hz sampling rate
- Three-axis gyroscope, which senses angular velocity
- Three-axis magnetometer, which senses magnetic fields
- Three-axis accelerometer, which senses accelerations, including gravitational

2.3 Additional Vision Lenses

The Rift comes installed with lenses for users with 20/20 or farsighted vision. If your vision is 20/20 or farsighted, you won't need to change your lenses and you can proceed to Section 2.4.

For nearsighted users, two additional pairs of lenses are included with the kit. Although they may not work perfectly for all nearsighted users, they should enable most people to use the headset without glasses or contact lenses.

The medium-depth lenses are for users who are moderately nearsighted. The shortest-depth lenses are for users who are very nearsighted. We recommend that users experiment with the different lenses to find the ones that work best for them.

The lenses are also marked with the letters 'A', 'B', and 'C' to aid identification. The recommended lenses are as follows:

Lenses	Designed for
A	20/20 or farsighted
B	Moderately nearsighted
C	Very nearsighted

Note: If your eyes have special characteristics such as astigmatism, the provided lenses may not be sufficient to correct your vision. In this case, we recommend wearing contact lenses or glasses. Note, however, that using glasses will cut down on your effective field of view.

2.3.1 Changing Vision Lenses

Note: Changing the lens may cause dust or debris to get inside the Rift. We strongly recommend changing the lenses in the cleanest space possible! Do not store the Rift without lenses installed.

To change lenses, first turn the headset upside down (this is to minimize the amount of dust and debris that can enter the headset) and gently unscrew the lenses currently attached to the headset. Unscrewing the lenses doesn't require much pressure; a light touch is most effective. The right lens unscrews clockwise. The left lens unscrews counterclockwise.

Place the old lenses in a safe place, then take the new lenses and install them the same way you removed the original pair. Remember to keep your headset upside down during this process. Once the new lenses are securely in place, you're all set!

After changing the lenses, you may need to adjust the distance of the assembly that holds the screen and lenses closer or farther away from your face. This is covered next.

2.4 Screen Distance Adjustment

The headset has an adjustment feature that allows you to change the distance of the fixture that holds the screen and lenses from your eyes. This is provided to accommodate different facial characteristics and vision lenses. For example, if the lenses are too close to your eyes, then you should adjust the fixture outward, moving the lenses and the screen away from your face. You can also use this to provide more room for eyeglasses.

Note: Everyone should take some time to adjust the headset for maximum comfort. While doing so, an important consideration is that the lenses should be situated as close to your eyes as possible. Remember that the maximal field of view occurs when your eyes are as close to the lenses as possible without actually touching them.

2.4.1 Changing The Screen Distance Adjustment

There are two screw mechanisms of either side of the headset that can be adjusted using a coin. These screws control the location of the screen assembly. The setting for the two screw mechanisms should always match unless you're in the process of adjusting them.

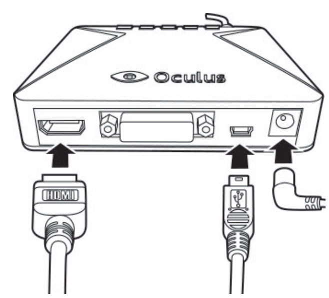
Turn the screw mechanism toward the lenses to bring the assembly closer to the user. Turn the screw mechanism toward the display to move the assembly farther away from the user. After changing one side, ensure that the other side is turned to the same setting!

2.5 Control Box Setup

The headset is connected to the control box by a 6ft cable. The control box takes in video, USB, and power, and sends them out over a single cord to minimize the amount of cabling running to the headset.

1. Connect one end of the video cable (DVI or HDMI) to your computer and the other end to the control box.

   Note: There should only be one video-out cable running to the control box at a time (DVI or HDMI, not both).
2. Connect one end of the USB cable to your computer and the other to the control box.
3. Plug the power cord into an outlet and connect the other end to the control box.



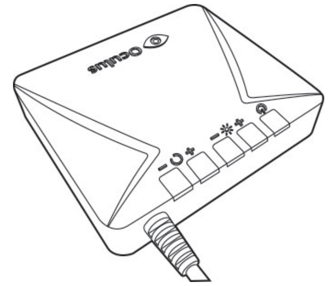
You can power on the DK using the power button on the top of the control box. A blue LED indicates whether the DK is powered on or off. The Rift screen will only stay on when all three cables are connected.

2.5.1 Adjusting Brightness and Contrast

The brightness and contrast of the headset can be adjusted using the buttons on the top of the control box.

Looking from the back side:

- The leftmost buttons adjust the display's contrast.
- The neighboring two adjust the display's brightness.
- The rightmost button turns the power on and off.



2.6 Monitor Setup

Once the Oculus Rift is connected to your computer, it should be automatically recognized as an additional monitor and Human Input Device (HID).

The Rift can be set to mirror or extend your current monitor setup using your computer's display settings. We recommend using the Rift as an extended monitor in most cases, but it's up you to decide which configuration works best for you. This is covered in more detail in Appendix A. Regardless of the monitor configuration, is it currently not possible to see the desktop clearly inside the Rift. This would require stereo rendering and distortion correction, which is only available while rendering the game scene.

Whether you decide to mirror or extend your desktop, the resolution of the Rift should always be set to 1280×800 (720p).

3 Oculus Rift SDK Setup

3.1 System Requirements

3.1.1 Operating Systems

The Oculus SDK currently supports Windows Vista, Windows 7, and Windows 8.

3.1.2 Minimum System Requirements

There are no specific computer hardware requirements for the Oculus SDK; however, we recommend that developers use a computer with a modern graphics card. A good benchmark is to try running Unreal Engine 3 and Unity at 60 frames per second (FPS) with vertical sync and stereo 3D enabled. If this is possible without dropping frames, then your configuration should be sufficient for Oculus Rift development!

The following components are provided as a guideline:

- Windows Vista or Windows 7
- 2.0+ GHz processor
- 2 GB system RAM
- Shader Model 3.0-compatible video card.

Although many lower end and mobile video cards, such as the Intel HD 4000, have the shader and graphics capabilities to run minimal Rift demos, their rendering throughput may be inadequate for full-scene 60 FPS VR rendering with stereo and distortion. Developers targeting this hardware will need to be very conscious of scene geometry because low-latency rendering at 60 FPS is critical for a usable VR experience.

If you are looking for a portable VR workstation, we've found that the Nvidia 650M inside of a MacBook Pro Retina provides enough graphics power for our demo development.

3.2 Installation

The latest version of the Oculus SDK is available at `http://developer.oculusvr.com`.

The naming convention for the Oculus SDK release package is `ovr_packagetype_major.minor.build`. For example, the initial build was `ovr_lib_0.1.1.zip`.

3.3 Windows

Extract the package to your computer. We recommend extracting it to a memorable location, for example `C:/Oculus`.

3.4 Directory Structure

The installed Oculus SDK package contains the following subdirectories:

`/3rdParty`	Third party SDK components used by samples, such as TinyXml.
`/Doc`	SDK Documentation, including this document.
`/LibOVR`	Libraries, source code, projects, and makefiles for the SDK.
`/LibOVR/Include`	Public include header files, including `OVR.h`. Header files here reference other headers in `LibOVR/Src`.
`/LibOVR/Lib`	Pre-built libraries for use in your project.
`/LibOVR/Src`	Source code and internally referenced headers.
`/Samples`	Samples that integrate and leverage the Oculus SDK.

3.5 Compiler Settings

The LibOVR libraries do not require exception handling or RTTI support, thereby allowing your game to disable these features for efficiency.

3.6 Makefiles, Projects, and Build Solutions

Development partners who have the source code can rebuild the LibOVR libraries using the projects and solutions in the `LibOVR/Projects` directory. Projects and makefiles are divided by platform.

3.6.1 Windows

The Visual Studio 2010 solution and project files are provided with the SDK:

- `/Samples/LibOVR_Samples_Msvc2010.sln` is the main solution that allows you to build and run all of the samples.

- `/LibOVR/Projects/Win32` contains the project needed to build the LibOVR library itself (for developers that have access to the full source).

3.7 Terminology

Interpupillary distance (IPD)	The distance between the eye pupils. The default value in the SDK is 64mm which corresponds to the average human distance, but values of 54mm to 72mm are possible.
Field of view (ϕ_{fov})	The full vertical viewing angle used to configure rendering. This is computed based on the eye distance and display size.
Aspect ratio (a)	The ratio of horizontal resolution to vertical resolution. The aspect ratio for each eye on the Oculus Rift is 0.8.
k_0, k_1, k_2	Optical radial distortion coefficients.
Multisampling	Hardware anti-aliasing mode supported by many video cards.

4 Getting Started

Your developer kit is unpacked and plugged in, you've installed the SDK, and you are ready to go. Where is the best place to begin?

If you haven't already, take a moment to adjust the Rift headset so that it's comfortable for your head and eyes. More detailed information about configuring the Rift can be found in Section 2.

Once your hardware is fully configured, the next step is to test the development kit. The SDK comes with a set of full-source C++ samples designed to help developers get started quickly. These include:

- **OculusWorldDemo** - A visually appealing Tuscany scene with on-screen text and controls.
- **OculusRoomTiny** - A minimal C++ sample showing sensor integration and rendering on the Rift.
- **SensorBoxTest** - A 3D rendered box that demonstrates sensor fusion by tracking and displaying the rotation of the Rift.

We recommend running the pre-built OculusWorldDemo as a first-step in exploring the SDK. You can find a link to the executable in the root of the Oculus SDK installation.

4.1 OculusWorldDemo

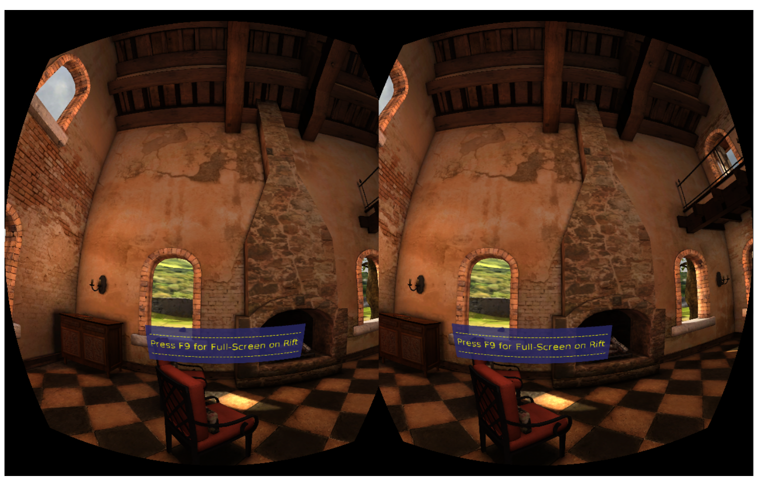


Figure 1: Screenshot of the OculusWorldDemo application.

4.1.1 Controls

Key or Input	Movement	Key	Function
W, S	Move forward, back	F1	No stereo, no distortion
A, D	Strafe left, right	F2	Stereo, no distortion
Mouse Move	Look left, right	F3	Stereo and distortion
Left Gamepad Stick	Move	F9	Hardware full-screen (low latency)
Right Gamepad Stick	Turn	F11	Windowed full-screen (no blinking)

Key(s)	Function	Keys	Function
R	Reset sensor orientation	Insert, Delete	Change interpupillary distance
G	Toggle grid overlay	PageUp, PageDown	Change aspect ratio
Spacebar	Toggle debug info overlay	[,]	Change field of view
Esc	Cancel full-screen	Y, H	Change k_1 coefficient
-, +	Adjust eye height	U, J	Change k_2 coefficient

4.1.2 Using OculusWorldDemo

Once you've launched OculusWorldDemo, take a moment to look around using the Rift and double check that all of the hardware is working properly. You should see an image similar to the screenshot in Figure 1. Press F9 or F11 to switch rendering to the Oculus Rift.

- **F9** - Switches to hardware full-screen mode. This will give best possible latency, but will blink monitors as Windows changes display settings. If no image shows up in the Rift, then press F9 again to cycle to the next monitor.
- **F11** - Instantly switches the rendering window to the Rift portion of the desktop. This mode has lower latency and no vsync, but is convenient for development.

If you're having problems (for example no image in the headset, no head tracking, and so on), then see the developer forums on the Oculus Developer Center. These should help for resolving common issues.

There are a number of interesting things to take note of during your first trip inside OculusWorldDemo. First, the level is designed "to scale". Thus, everything appears to be roughly the same height as it would be in the real world. The sizes for everything, including the chairs, tables, doors, and ceiling, are based on measurements from real world objects. All of the units are measured in meters.

Depending on your actual height, you may feel shorter or taller than normal. The default eye-height of the player in OculusWorldDemo is 1.78 meters (5ft 10in), but this can be adjusted using the '+' and '-' keys.

As you may have already concluded, the scale of the world and the player is critical to an immersive VR experience. This means that players should be a realistic height, and that art assets should be sized proportionally. More details on scale can be found in the "Oculus Best Practices Guide" document. Among other things, the demo includes simulation of a basic head model, which causes head rotation to introduce additional displacement proportional to the offset of eyes from the base of the neck. This displacement is important for improving realism and reducing disorientation.

4.2 Using the SDK Beyond the OculusWorldDemo

4.2.1 Software Developers and Integration Engineers

If you're integrating the Oculus SDK into your game engine, we recommend starting by opening the samples solution (`/Samples/LibOVR_Samples_Msvc2010.sln`), building the projects, and experimenting with the provided sample code.

OculusRoomTiny is a good place to start because its source code compactly combines all critical features of the Oculus SDK. It contains logic necessary to initialize LibOVR core, access Oculus devices, implement head-tracking, sensor fusion, head modeling, stereoscopic 3D rendering, and distortion shaders.

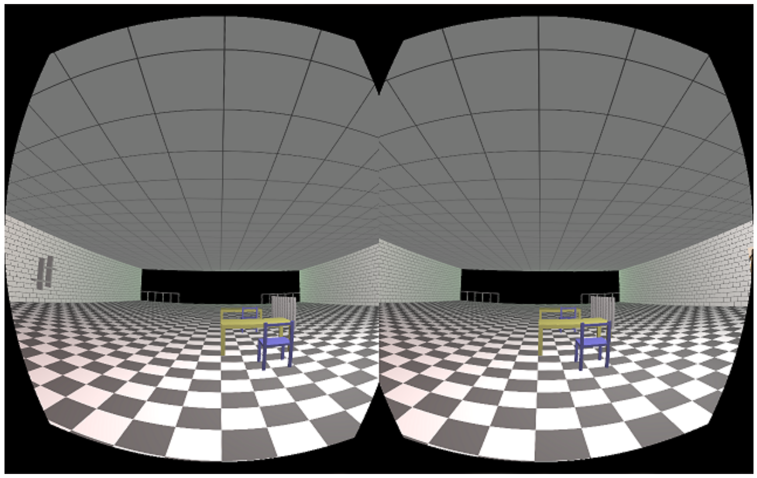


Figure 2: Screenshot of the OculusRoomTiny application.

OculusWorldDemo is a more complex sample. It is intended to be portable and supports many more features including: windowed/full-screen mode switching, XML 3D model and texture loading, movement collision detection, adjustable distortion and view key controls, 2D UI text overlays, and so on. This is a good application to experiment with once you are familiar with Oculus SDK basics.

Beyond experimenting with the provided sample code, you should continue to follow this document. We'll cover important topics including the Oculus kernel, initializing devices, head-tracking, rendering for the Rift, and minimizing latency.

4.2.2 Artists and Game Designers

If you're an artist or game designer unfamiliar in C++, we recommend downloading UE3 or Unity along with the corresponding Oculus integration. You can use our out-of-the-box integrations to begin building Oculus-based content immediately.

The "Unreal Engine 3 Integration Overview" document and the "Unity Integration Overview" document, available from the Oculus Developer Center, detail the steps required to set up your UE3/Unity plus Oculus development environment.

We also recommend reading through the "Oculus Best Practices Guide", which has tips, suggestions, and research oriented around developing great VR experiences. Topics include control schemes, user interfaces, cut-scenes, camera features, and gameplay. The "Best Practices Guide" should be a go-to reference when designing your Oculus-ready games.

Aside from that, the next step is to get started building your own Oculus-ready games! Thousands of other developers, like you, are out there building the future of virtual reality gaming. You can reach out to them by visiting `http://developer.oculusvr.com/forums`.

5 LibOVR Integration Tutorial

If you've made it this far, you are clearly interested in integrating the Rift with your own game engine. Awesome. We are here to help.

We've designed the Oculus SDK to be as easy to integrate as possible. This section outlines a basic Oculus integration into a C++ game engine or application. We'll discuss initializing the LibOVR kernel, device enumeration, head tracking, 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 draw sample integration code from when in doubt about a particular system or feature.

5.1 Outline of Integration Tasks

To add Oculus support to a new game, you'll need to do the following:

1. Initialize LibOVR.
2. Enumerate Oculus devices, creating HMD device and sensor objects.
3. Integrate head-tracking into your game's view and movement code. This involves:
   (a) Reading data from the Rift's sensors through the SensorFusion class.
   (b) Applying the calculated Rift orientation to the camera view, while combining it with other controls.
   (c) Modifying movement and game play to consider head orientation.
4. Modify game rendering to integrate the HMD, including:
   (a) Stereoscopic 3D rendering for each eye.
   (b) Correctly computing projection, ϕ_{fov}, and other parameters based on the HMD settings.
   (c) Applying a pixel shader to correct for optical distortion.
5. Customize UI screens to work well inside of the headset.

We'll first take a look at obtaining sensor data, since it's a relatively easy to setup. Then we'll move on to the more involved subject of rendering.

5.2 Initialization of LibOVR

We initialize LibOVR's core by calling `System::Init`, which will configure logging and register a default memory allocator (that you can override).

```
#include "OVR.h"
using namespace OVR;
System::Init(Log::ConfigureDefaultLog(LogMask_All));
```

Note that System::Init must be called before any other OVR_Kernel objects are created, and System::Destroy must be called before program exit for proper cleanup. Another way to initialize the LibOVR core is to create a System object and let its constructor and destructor take care of initialization and cleanup, respectively. In the cases of OculusWorldDemo and OculusRoomTiny, the init and destroy calls are invoked by the OVR_PLATFORM_APP macro.

Once the system has been initialized, we create an instance of OVR::DeviceManager. This allows us to enumerate detected Oculus devices. All Oculus devices derive from the DeviceBase base class which provides the following functionality:

1. It supports installable message handlers, which are notified of device events.
2. Device objects are created through DeviceHandle::CreateDevice or more commonly through DeviceEnumerator<>::CreateDevice.
3. Created devices are reference counted, starting with a RefCount of 1.
4. A device's resources are cleaned up when it is Released, although its handles may survive longer if referenced.

We use DeviceManager::Create to create a new instance of DeviceManager. Once we've created the DeviceManager, we can use DeviceManager::EnumerateDevices to enumerate the detected Oculus devices. In the sample below, we create a new DeviceManager, enumerate available HMDDevice objects, and store a reference to the first active HMDDevice that we find.

```
Ptr<DeviceManager> pManager;
Ptr<HMDDevice>     pHMD;
pManager = *DeviceManager::Create();
pHMD     = *pManager->EnumerateDevices<HMDDevice>().CreateDevice();
```

We can learn more about a device by using DeviceBase::GetDeviceInfo(DeviceInfo* info). The DeviceInfo structure is used to provide detailed information about a device and its capabilities. DeviceBase::GetDeviceInfo is a virtual function, therefore subclasses like HMDDevice and SensorDevice can provide subclasses of DeviceInfo with information tailored to their unique properties.

In the sample below, we read the vertical resolution, horizontal resolution, and screen size from an HMDDevice using HMDDevice::GetDeviceInfo with an HMDInfo object (subclass of DeviceInfo).

```
HMDInfo hmd;
if (pHMD->GetDeviceInfo(&hmd))
{
    MonitorName    = hmd.DisplayDeviceName;
    EyeDistance    = hmd.InterpupillaryDistance;
    DistortionK[0] = hmd.DistortionK[0];
    DistortionK[1] = hmd.DistortionK[1];
    DistortionK[2] = hmd.DistortionK[2];
    DistortionK[3] = hmd.DistortionK[3];
}
```

The same technique can be used to learn more about a SensorDevice object. Now that we have information about the HMDDevice, the next step is to setup rendering for the Rift.

5.3 Leveraging Sensor Data

The Oculus tracker includes a gyroscope, accelerometer, and magnetometer. We combine the information from these sensors through a process known as "Sensor Fusion" in order to determine the orientation of the player's head in the real world, and to synchronize the player's virtual perspective in real-time.

The Rift orientation is reported as a set of rotations in a right-handed coordinate system, as illustrated in Figure 3.

This coordinate system uses the following axis definitions:

- Y is *Up Positive*
- X is *Right Positive*
- Z is *Back Positive*

Rotations are counter-clockwise (CCW) when looking in the negative direction of the axis. This means they are interpreted as follows:

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

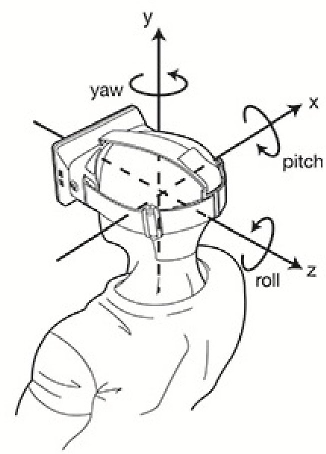


Figure 3: The Rift coordinate system

The gyroscope reports the rate of rotation (angular velocity) around X, Y, and Z axes, in radians/second. This provides the most valuable data for head orientation tracking. By continuously accumulating angular velocity samples over time, the Oculus SDK can determine the direction of the Rift relative to where it began.

To integrate head-tracking, first we need a `SensorDevice` object to read from. If we have a reference to an HMD, we get a reference to the associated sensor using `HMDDevice::GetSensor` as follows:

```
Ptr<SensorDevice> pSensor;
pSensor = *pHMD->GetSensor();
```

We can get more information about the sensor using `SensorDevice::GetInfo`.

The `SensorFusion` class accumulates Sensor notification messages to keep track of orientation. This involves integrating the gyroscope data and then using the other sensors to correct for drift. `SensorFusion` provides the orientation as a quaternion, from which users can obtain a rotation matrix or Euler angles. There are two ways to receive updates from the SensorFusion class:

1. We can manually pass MessageBodyFrame messages to the OnMessage() function.
2. We can attach SensorFusion to a SensorDevice. This will cause the SensorFusion instance to automatically handle notifications from that device.

```
SensorFusion SFusion;
if (pSensor)
    SFusion.AttachToSensor(pSensor);
```

Once an instance of SensorFusion is attached to a SensorDevice, we can use it to get relevant data from the Oculus tracker through the following functions:

```
// Obtain the current accumulated orientation.
Quatf    GetOrientation() const
// Obtain the last absolute acceleration reading, in m/s^2.
Vector3f GetAcceleration() const
// Obtain the last angular velocity reading, in rad/s.
Vector3f GetAngularVelocity() const
```

In most cases, the most important data coming from SensorFusion will be the orientation Quaternion provided by `GetOrientation`. We'll make use of this to update the virtual view to reflect the orientation of the player's head. We'll also account for the orientation of the sensor in our rendering pipeline.

```
// We extract Yaw, Pitch, Roll instead of directly using the orientation
// to allow "additional" yaw manipulation with mouse/controller.
Quatf hmdOrient = SFusion.GetOrientation();
float yaw = 0.0f;
hmdOrient.GetEulerABC<Axis_Y, Axis_X, Axis_Z>(&yaw, &EyePitch, &EyeRoll);
EyeYaw += (yaw - LastSensorYaw);
LastSensorYaw = yaw;
// NOTE: We can get a matrix from orientation as follows:
Matrix4f hmdMat(hmdOrient);
```

Developers can also read the raw sensor data directly from the SensorDevice, bypassing SensorFusion entirely, by using `SensorDevice:: SetMessageHandler(MessageHandler* handler)`. The `MessageHandler` delegate will receive a `MessageBodyFrame` every time the tracker sends a data sample. A `MessageBodyFrame` instance provides the following data:

```
Vector3f Acceleration;   // Acceleration in m/s^2.
Vector3f RotationRate;   // Angular velocity in rad/s^2.
Vector3f MagneticField;  // Magnetic field strength in Gauss.
float    Temperature;    // Temperature reading on sensor surface, in degrees Celsius.
float    TimeDelta;      // Time passed since last Body Frame, in seconds.
```

5.4 User Input Integration

Head-tracking will need to be integrated with an existing control scheme for many games to provide the most comfortable, intuitive, and usable interface for the player.

For example, in a standard First Person Shooter (FPS), the player 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; they need a way to reorient themselves so that they are always comfortable (the same way we turn our bodies if we want to look behind ourselves for more than a brief glance).

As a result, developers should carefully consider their control schemes and how to integrate head-tracking when designing games for VR. The OculusRoomTiny application provides a source code sample for integrating Oculus head tracking with the aforementioned, standard FPS control scheme.

5.5 Rendering Configuration

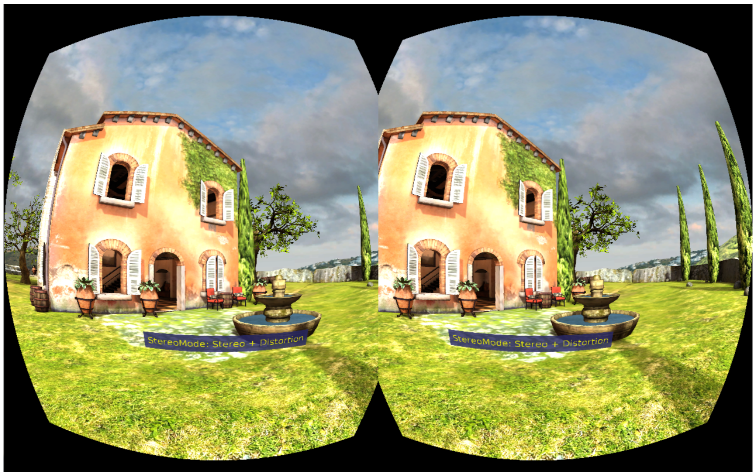


Figure 4: OculusWorldDemo stereo rendering.

As you may be aware, Oculus rendering requires split-screen stereo with distortion correction for each eye to account for the Rift's optics. Setting this up can be tricky, but immersive rendering is what makes the Rift magic come to life. We separate our rendering description into several sections:

- Section 5.5.1 introduces the basics of HMD stereo rendering and projection setup.
- Section 5.5.2 covers distortion correction, describing the pixel shader and its associated parameters.
- Sections 5.5.3 and 5.5.4 round out the discussion by explaining the scaling and field of view math necessary for the scene to look correct.
- Finally, Section 5.5.5 introduces the StereoConfig utility class that does a lot of the hard work for you, hiding math complexity behind the scenes.

Aside from changes to the game engine's rendering pipeline, 60 FPS low latency rendering is also critical for immersive VR. We cover VSync, latency, and other performance requirements in Section 5.5.6.

5.5.1 Rendering Stereo

The Oculus Rift requires the game scene to be rendered in split-screen stereo, with half the screen used for each eye. When using the Rift, your left eye sees the left half of the screen, whereas the right eye sees the right half. This means that your game will need to render the entire scene twice, which can be achieved with logic similar to the following pseudo code:

```
// Render Left Eye Half
SetViewport(0, 0, HResolution/2, VResolution);
SetProjection(LeftEyeProjectionMatrix);
RenderScene();

// Render Right Eye Half
SetViewport(HResolution/2, 0, HResolution, VResolution);
SetProjection(RightEyeProjectionMatrix);
RenderScene();
```

Note that the $reprojection\ stereo\ rendering$ technique, which relies on left and right views being generated from a single fully rendered view, is not usable inside of an HMD because of significant artifacts at object edges.

Unlike stereo TVs, rendering inside of the Rift does not require off-axis or asymmetric projection. Instead, projection axes are parallel to each other as illustrated in Figure 5. This means that camera setup will be very similar to that normally used for non-stereo rendering, except you will need to shift the camera to adjust for each eye location.

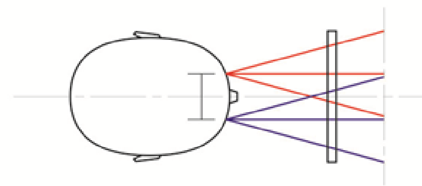


Figure 5: HMD eye view cones.

To get correct rendering on the Rift, the game needs to use the physically appropriate field of view (ϕ_{fov}), calculated based on the Rift's dimensions. The parameters needed for stereo rendering are reported from LibOVR in `OVR::HMDInfo` as follows:

Member Name	**Description**
HScreenSize, VScreenSize	Physical dimensions of the entire HMD screen in meters. Half HScreenSize is used for each eye. The current physical screen size is 149.76 x 93.6mm, which will be reported as 0.14976f x 0.0935f.
VScreenCenter	Physical offset from the top of the screen to eye center, in meters. Currently half VScreenSize.
EyeToScreenDistance	Distance from the eye to the screen, in meters. This combines distances from the eye to the lens, and from the lens to the screen. This value is needed to compute the ϕ_{fov} correctly.
LensSeparationDistance	Physical distance between the lens centers, in meters. Lens centers are the centers of distortion; we will talk about them later in Section 5.5.2
InterpupillaryDistance	Configured distance between eye centers.
HResolution, VResolution	Resolution of the entire HMD screen in pixels. Half the HResolution is used for each eye. The reported values are 1280×800 for the DK, but we are determined to increase this in the future!
DistortionK	Radial distortion correction coefficients, discussed in Section 5.5.2.

So, how do we use these values to setup projection? For simplicity, let us focus on rendering for the left eye and ignore the distortion for the time being. Before you can draw the scene you'll need to take several steps:

1. Set the viewport to cover the left eye screen area.
2. Determine the aspect ratio a and ϕ_{fov} based on the reported HMDInfo values.
3. Calculate the center projection matrix $\mathbf{P}$ based on a and ϕ_{fov}.
4. Adjust the projection matrix $\mathbf{P}$ based on interpupillary distance.
5. Adjust the view matrix $\mathbf{V}$ to match eye location.

Setting up the viewport is easy, simply set it to `(0,0, HResolution/2, VResolution)` for the left eye. In most 3D graphics systems, the clip coordinates [-1,1] will be mapped to fill the viewport, with (0,0) corresponding to the center of projection.

Ignoring distortion, Rift half-screen aspect ratio a and vertical FOV ϕ_{fov} are determined by

$$a = \frac{HResolution}{2 \cdot VResolution} \tag{1}$$

and

$$\phi_{fov} = 2 \arctan\left(\frac{VScreenSize}{2 \cdot EyeToScreenDistance}\right). \tag{2}$$

We form the projection matrix, $\mathbf{P}$, based on a and ϕ_{fov}, as

$$\mathbf{P} = \begin{bmatrix} \frac{1}{a \cdot \tan(\phi_{fov}/2)} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(\phi_{fov}/2)} & 0 & 0 \\ 0 & 0 & \frac{z_{far}}{z_{near}-z_{far}} & \frac{z_{far} \cdot z_{near}}{z_{near}-z_{far}} \\ 0 & 0 & -1 & 0 \end{bmatrix}, \tag{3}$$

in which z_{near} and z_{far} are the standard clipping plane depth coordinates. This common calculation can be done by the `Matrix4f::PerspectiveRH` function in the Oculus SDK, the `gluPerspective` utility function in OpenGL, or `D3DXMatrixPerspectiveFovRH` in Direct3D.

The projection center of $\mathbf{P}$ as computed above falls in the center of each screen, so we need to modify it to coincide with the center of the eye instead. This adjustment can be done in final clip coordinates, computing the final left and right projection matrices as illustrated below. Let h denote the absolute value of horizontal offset to account for eye separation. This can be used in a transformation matrix

$$\mathbf{H} = \begin{bmatrix} 1 & 0 & 0 & \pm h \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{4}$$

which is applied at the end to obtain $\mathbf{P}' = \mathbf{H}\,\mathbf{P}$. In the upper right corner of $\mathbf{H}$, the term h appears for the left-eye case, and $-h$ appears for the right eye. The particular horizontal shift in meters is

$$h_{meters} = \frac{HScreenSize}{4} - \frac{IPD}{2}. \tag{5}$$

In screen coordinates,

$$h = \frac{4\,h_{meters}}{HScreenSize}. \tag{6}$$

In terms of screen size, this adjustment is significant: Assuming 64mm IPD and 149.76mm screen size of the 7" Rift, each eye projection center needs to be translated by about 5.44mm towards the center of the device. This is a critical step for correct stereo rendering.

Assuming that the original non-stereo game view transform $\mathbf{V}$ falls at the center between the eyes, the final adjustment we have to make is shift that view horizontally to match each eye location:

$$\mathbf{V}' = \begin{bmatrix} 1 & 0 & 0 & \pm\frac{1}{2}IPD \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{V}. \tag{7}$$

It is important that this shift is done by half of the interpupillary distance in world units IPD. Please refer to your game's content or design documents for the conversion to or from real-world units. In Unreal Engine 3, for example, 2 units = 1 inch. In Unity, 1 unit = 1 meter.

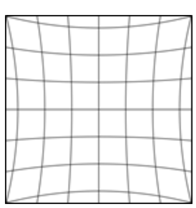
We can now present a more complete example of this stereo setup, as it is implemented inside of the `OVR::Util::Render::StereoConfig` utility class. It covers all of the steps described in this section with exception of viewport setup.

```
HMDInfo& hmd = ...;
Matrix4f viewCenter = ...;

// Compute Aspect Ratio. Stereo mode cuts width in half.
float aspectRatio = float(hmd.HResolution * 0.5f) / float(hmd.VResolution);

// Compute Vertical FOV based on distance.
float halfScreenDistance = (hmd.VScreenSize / 2);
float yfov = 2.0f * atan(halfScreenDistance/HMD.EyeToScreenDistance);

// Post-projection viewport coordinates range from (-1.0, 1.0), with the
// center of the left viewport falling at (1/4) of horizontal screen size.
// We need to shift this projection center to match with the eye center
// corrected by IPD. We compute this shift in physical units (meters) to
// correct for different screen sizes and then rescale to viewport coordinates.
float viewCenter             = hmd.HScreenSize * 0.25f;
float eyeProjectionShift     = viewCenter - hmd.InterpupillaryDistance*0.5f;
float projectionCenterOffset = 4.0f * eyeProjectionShift / hmd.HScreenSize;

// Projection matrix for the "center eye", which the left/right matrices are based on.
Matrix4f projCenter = Matrix4f::PerspectiveRH(yfov, aspect, 0.3f, 1000.0f);
Matrix4f projLeft   = Matrix4f::Translation(projectionCenterOffset, 0, 0) * projCenter;
Matrix4f projRight  = Matrix4f::Translation(-projectionCenterOffset, 0, 0) * projCenter;

// View transformation translation in world units.
float    halfIPD  = hmd.InterpupillaryDistance * 0.5f;
Matrix4f viewLeft = Matrix4f::Translation(halfIPD, 0, 0) * viewCenter;
Matrix4f viewRight= Matrix4f::Translation(-halfIPD, 0, 0) * viewCenter;
```

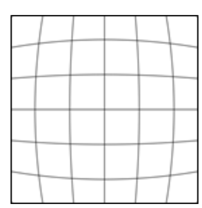
With all of this setup done, you should be able to see the 3D world and converge on it inside of the Rift. The last, and perhaps more challenging step, will be correcting for distortion due to the lenses.

5.5.2 Distortion Correction

The optical lens used inside of the rift magnifies the image, increasing the field of view. It also generates a radial pincushion distortion that warps the image as illustrated in the image on the left below.



Pincushion Distortion



Barrel Distortion

For the Oculus Rift DK, distortion needs to be corrected in software by warping the image with a barrel distortion, as seen in the image on the right. When the two distortions are combined, the barrel distortion will cancel out the lens pincushion effect, producing straight lines.

Both pincushion and barrel distortion can be modeled by the following distortion function, defined for radius r from the center of distortion:

$$r' = r(k_0 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

Here, the resulting radius r' is computed based on the original radius r and fixed coefficients $k_{0..3}$. The coefficients are positive for a barrel distortion. The radius r' is used to modify the sample location in the render surface generated when we rendered the application. As a result of applying the distortion, pixels are pulled towards the center of the lens, with the amount of displacement increasing with radius. In OculusWorldDemo, this is implemented by the following Direct3D10 pixel shader:

```
Texture2D    Texture : register(t0);
SamplerState Linear  : register(s0);
float2       LensCenter;
float2       ScreenCenter;
float2       Scale;
float2       ScaleIn;
float4       HmdWarpParam;

// Scales input texture coordinates for distortion.
float2 HmdWarp(float2 in01)
{
    float2 theta  = (in01 - LensCenter) * ScaleIn; // Scales to [-1, 1]
    float  rSq    = theta.x * theta.x + theta.y * theta.y;
    float2 rvector= theta * (HmdWarpParam.x + HmdWarpParam.y * rSq +
                             HmdWarpParam.z * rSq * rSq +
                             HmdWarpParam.w * rSq * rSq * rSq);
    return LensCenter + Scale * rvector;
}
float4 main(in float4 oPosition : SV_Position, in float4 oColor : COLOR,
            in float2 oTexCoord : TEXCOORD0) : SV_Target
{
    float2 tc = HmdWarp(oTexCoord);
    if (any(clamp(tc, ScreenCenter-float2(0.25,0.5),
                      ScreenCenter+float2(0.25, 0.5)) - tc))
        return 0;
    return Texture.Sample(Linear, tc);
};
```

This shader is designed to run on a quad covering one half of the screen, while the input texture spans both the left and right eyes. The input texture coordinates, passed in as `oTexCoord`, range from (0,0) for the top left corner of the Oculus screen, to (1,1) at the bottom right. This means that for the left eye viewport, `oTexCoord` will range from (0,0) to (0.5,1). For the right eye it will range from (0.5,0) to (1,1).

The distortion function used by `HmdWarp` is, however, designed to operate on [-1,1] unit coordinate range, from which it can compute the radius. This means that there are a number of variables needed to scale and center the coordinates properly to apply the distortion. These are:

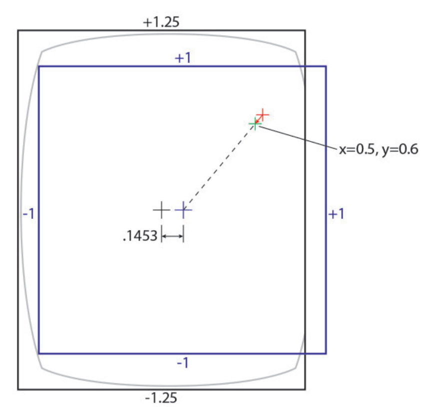
ScaleIn	Rescale input texture coordinates to [-1,1] unit range, and corrects aspect ratio.
Scale	Rescale output (sample) coordinates back to texture range and increase scale so as to support sampling outside of the screen.
LensCenter	Shifts texture coordinates to center the distortion function around the center of the lens:

$$LensCenter = 4\left(\frac{\frac{HScreenSize}{4} - \frac{LensSeparationDistance}{2}}{HScreenSize}\right) \tag{8}$$

HmdWarpParam	Distortion coefficients (`DistortionK[]`)
ScreenCenter	Texture coordinate for the center of the half-screen texture. This is used to clamp sampling, preventing pixel leakage from one eye view to the other.

The following diagram illustrates the left eye distortion function coordinate range, shown as a blue rectangle, as it relates to the left eye viewport coordinates. As you can see, the center of distortion has been shifted to the right in relation to the screen center to align it with axis through the center of the lense. For the 7" screen and 64mm lense separation distance, viewport shift is roughly 0.1453 coordinate units. These parameters may change for future headsets and so this should always be computed dynamically.

The diagram also illustrates how sampling coordinates are mapped by the distortion function. A distortion unit coordinate of (0.5,0.6) is marked as a green cross; it has a radius of ≈ 0.61. In the example shown, this maps to a sampling radius of ≈ 0.68 post-distortion, illustrated by a red cross. As a result of the distortion shader, pixels in the rendered image move towards the center of distortion, or from red to green in the diagram. The amount of displacement increases the further out we go from the distortion center.



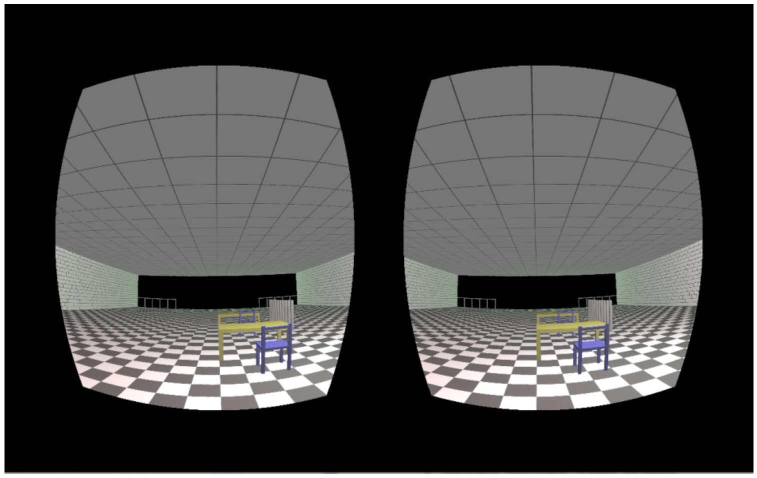
Hopefully it's clear from this discussion, that the barrel distortion pixel shader needs to run as a post-process on the rendered game scene image. This has several implications:

- The original scene rendering will need to be done to a render-target.
- The scene render-target will need to be larger than the final viewport, to account for the distortion pulling pixels in towards the center.
- Field of view (FOV) and image scale will need to be adjusted do accommodate for the distortion.

We will now discuss the distortion scale, render target, and FOV adjustments necessary to make things look correct inside of the Rift.

5.5.3 Distortion Scale

If you run the distortion shader on the original image render target that is the same size as the output screen you will get an image similar to the following:



Here, the pixels at the edges have been pulled in towards the center, with black being displayed outside, where no pixel data was available. Although this type of rendering would look acceptable within the Rift, a significant part of the FOV is lost as large areas of the screen go unused. How would we make the scene fill up the entire screen?

The simplest solution is to increase the scale of the input texture, controlled by the Scale variable of the distortion pixel shader discussed earlier. As an example, if we want to increase the perceived input texture size by 25% we can adjust the sampling coordinate Scale by a factor of (1/1.25) = 0.8. Doing so will have several effects:

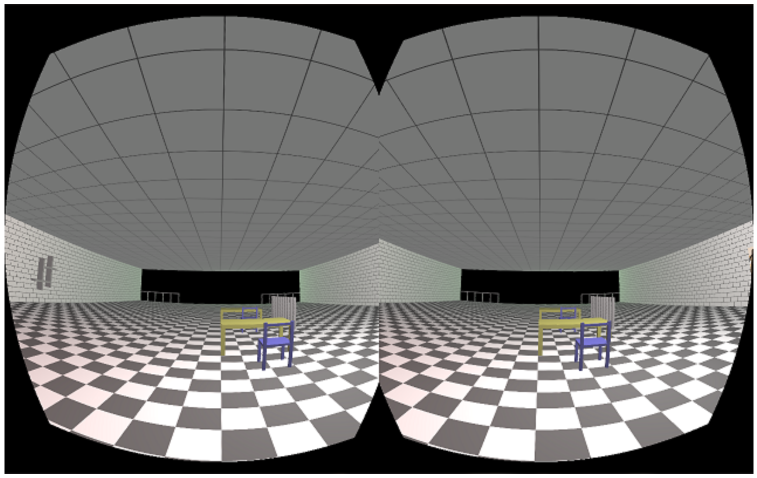
- The size of the post-distortion image will increase on screen.
- The required rendering FOV will increase.
- The quality of the image will degrade due to sub-sampling from the scaled image, resulting in blocky or blurry pixels around the center of the view.

Since we really don't want the quality to degrade, the size of the source render target can be increased by the same amount to compensate. For the 1280×800 resolution of the Rift, a 25% scale increase will require rendering a 1600×1000 buffer. Unfortunately this incurs a 1.56 times increase in the number of pixels in the source render target. Due to the nature of the distortion function, the area required increases exponentially with radius. However, we don't need to completely fill the very corners of the screen where the user cannot see. Evidently there are some trade-offs that can be made between the covered field of view, quality, and rendering performance.

For the 7" Rift, we recommend picking the scale that fits close to the left side of the screen. This gives you the maximum horizontal FOV without filling the pixels at the top of the screen, which are not visible to most users.

Recall the distortion function:

$$r' = r(k_0 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$



The scale is then simply

$$s = \frac{r'}{r} \quad \text{in which } r \text{ is the polar coordinate of the farthest point to fit.}$$

For example, to fit the left side of the display $(-1, 0)$:

$$r = -1 - LensCenter$$

For OculusWorldDemo, the actual distortion scale factor is computed inside of the `StereoConfig::updateDistortionOffsetAndScale` function by fitting the distortion radius to the left edge of the viewport. The `LensCenter` is the same as used in the shader above.

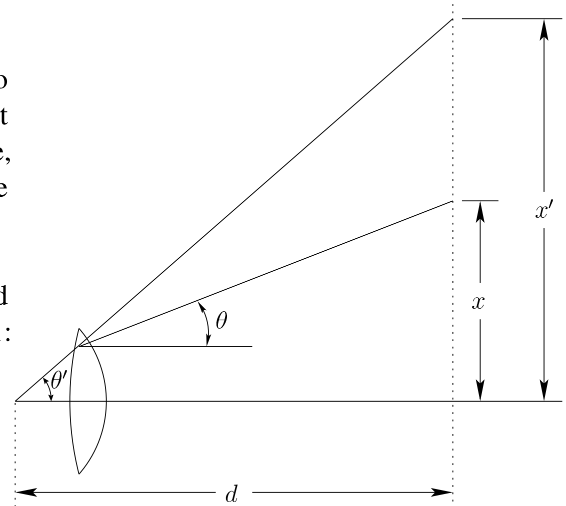
5.5.4 Distortion and FOV

With distortion scale and offset properly computed, the remaining thing is to compute the proper FOV. For this, we need to examine the geometry of the Rift projection as illustrated in the diagram on the right.

The diagram illustrates the effect that a lens introduces into an otherwise simple calculation of the FOV, assuming that the user is looking at the screen through the lens. Here, d specifies the 'eye to screen' distance and x is half the vertical screen size ($VScreenSize/2$).

In the absence of the lens, the FOV can be easily computed based on the distances d and x, as described in Section 5.5.1:

$$\phi_{fov} = 2\arctan(x/d)$$



The lens, however, increases the perceived screen size from x to x', where x' can be computed through the distortion function. Thus, the distorted ϕ'_{fov} is calculated as follows:

$$x' = x(k_0 + k_1x^2 + k_2x^4 + k_3x^6)$$

$$\theta' = \arctan(x'/d)$$

$$\phi'_{fov} = 2 \cdot \theta'$$

In our case, we want to compute the field of view of the distorted render target (RT), which is affected by the distortion scale s and may not necessarily match the screen edge. Assuming that both the RT and the display have the same aspect ratios, it is enough to adjust the screen size to get the correct perceived rendering FOV:

$$x'' = \frac{s \cdot VScreenSize}{2}$$

$$\phi_{fov} = 2 \arctan x''/d$$

At first glance, it may seem strange that the rendering FOV equation bypasses the distortion function. However, we must remember that we are computing perceived RT size x'' inside of the Rift, where the optical lens cancels out the effect of the shader distortion. Under these conditions, `DistortionScale` accurately represents half the size of the render target, assuming [-1,1] unit coordinate range before scaling.

5.5.5 StereoConfig Utility Class

If setting up the projection and distortion scaling seems like a lot of work, you'll be happy to learn that the Oculus SDK comes with a set of full-source utility classes that do a lot of the work for you. The most important class for rendering setup is `StereoConfig`, located inside of the `OVR::Util::Render` namespace. This class works as follows:

1. First, create an instance of `StereoConfig` and initialize it with `HMDInfo`, viewport, distortion fit location, IPD and other desired settings.
2. `StereoConfig` computes the rendering scale, distortion offsets, FOV and projection matrices.
3. `StereoConfig::GetEyeRenderParams` returns the projection matrix and distortion settings for each eye. These can be used for rendering stereo inside of the game loop.

`StereoConfig` class is used for rendering setup inside of the OculusRoomTiny and OculusWorldDemo samples. Here's an example of the initialization you need to get started:

```
using namespace OVR::Util::Render;

HMDDevice*   pHMD = ...;
StereoConfig stereo;
HMDInfo      hmd;
float        renderScale;

// Obtain setup data from the HMD and initialize StereoConfig
// for stereo rendering.
pHMD->GetDeviceInfo(&hmd);

stereo.SetFullViewport(Viewport(0,0, Width, Height));
stereo.SetStereoMode(Stereo_LeftRight_Multipass);
stereo.SetHMDInfo(hmd);
stereo.SetDistortionFitPointVP(-1.0f, 0.0f);

renderScale = stereo.GetDistortionScale();
```

As you can see, after all parameters are initialized, `GetDistortionScale` computes the rendering scale that should be applied to the render texture. This is the scale that will maintain one-to-one rendering quality at the center of the screen while simultaneously scaling the distortion to fit its left edge.

Based on this computed state, you can get left and right eye rendering parameters as follows:

```
StereoEyeParams leftEye  = stereo.GetEyeRenderParams(StereoEye_Left);
StereoEyeParams rightEye = stereo.GetEyeRenderParams(StereoEye_Right);

// Left eye rendering parameters
Viewport        leftVP         = leftEye.VP;
Matrix4f        leftProjection = leftEye.Projection;
Matrix4f        leftViewAdjust = leftEye.ViewAdjust;
```

You can use the resulting `Viewport` and projection matrix directly for rendering the scene. `ViewAdjust` should be a post-transform applied after the game's view matrix to properly shift the camera for the left or right eye.

5.5.6 Rendering Performance

Aside from changes to the game engine's renderer to account for the Rift's optics, there are two other requirements when rendering for VR:

- The game engine should run at least 60 frames per second without dropping frames.
- Vertical Sync (vsync) should always be enabled to prevent the player from seeing screen tearing.

These may seem arbitrary, but our experiments have shown them to be important for a good VR experience. A player can easily tell the difference between 30 FPS and 60 FPS when playing in VR because of the immersive nature of the game. The brain can suspend disbelief at 60 FPS. At 30 FPS, the world feels choppy. Vertical sync is also critical. Since the Rift screen covers all of the player's view, screen tearing is very apparent, and causes artifacts that break immersion.

6 Optimization

6.1 Latency

Minimizing latency is crucial to immersive VR and low latency head tracking is part of what sets the Rift apart. We define latency as the time between movement of the player's head, and the updated image being displayed on the screen. We call this latency loop "motion-to-photon" latency. The more you can minimize motion-to-photon latency in your game, the more immersive the experience will be for the player.

Two other important concepts are actual latency and perceived latency.

Actual latency is equivalent to motion-to-photon latency. It is the latency in the system at the hardware and software level.

Perceived latency is how much latency the player perceives when using the headset. Perceived latency may be less than actual latency depending on the player's movements and by employing certain techniques in software.

We're always working to reduce actual and perceived latency in our hardware and software pipeline. For example, in some cases we're able to reduce perceived latency by 20ms or more using a software technique called predictive tracking.

Although 60ms is a widely cited threshold for acceptable VR, at Oculus we believe the threshold for compelling VR to be below 40ms of latency. Above this value you tend to feel significantly less immersed in the environment. Obviously, in an ideal world, the closer we are to 0ms, the better.

For the Rift developer kit, we expect the actual latency to be approximately 30ms to 50ms. This depends partly on the screen content. For example, a change from black to dark brown may take 5ms but a larger change in color from black to white may take 20ms.

Stage	**Event**	**Event Duration**	**Worst Case Total Latency**
Start	Oculus tracker sends data	N/A	0ms
Transit	Computer receives tracker data	$\approx 2\text{ms}$	$\approx 2\text{ms}$
Processing	Game engine renders latest frame (60 FPS w/ vysnc)	≈ 0 to 16.67ms	$\approx 19\text{ms}$
Processing	Display controller writes latest frame to LCD (top to bottom)	$\approx 16.67\text{ms}$	$\approx 36\text{ms}$
Processing	Simultaneously, pixels switching colors	≈ 0 to 15ms	$\approx 51\text{ms}$
End	Latest frame complete; presented to user	N/A	$\approx 51\text{ms}$

Again, these numbers represent the actual latency assuming a game running at 60 FPS with vsync enabled. Actual latency will vary depending on the scene being rendered. Perceived latency can be reduced further. As developers, we want to do everything we can to reduce latency in this pipeline. Techniques for Reducing Latency:

- Run at 60 FPS (remember that vsync should always be enabled for VR).
- Minimize swap-chain buffers to a maximum of 2 (the on screen and off screen buffers).
- Reduce the amount of rendering work where possible. Multi-pass rendering and complex shaders

increase the rendering latency and hence the time between reading the HMD orientation and having the frame ready to display.

- Reduce render command buffer size. By default the driver may buffer several frames of render commands in order to batch GPU transfers and smooth out variability in rendering times. This needs to be minimized. One technique is to make a rendering call that blocks until the current frame is complete. This can be a "block until render queue empty event" or a command that reads back a property of the rendered frame. While blocking, we're preventing additional frames from being submitted and hence buffered in the command queue.

A Display Device Management

A.1 Display Identification

Display devices identify themselves and their capabilities using EDID[1]. When the device is plugged into a PC, the display adapter reads a small packet of data from it. This includes the manufacturer code, device name, supported display resolutions, and information about video signal timing. When running Microsoft Windows, the display is identified and added to a list of active display devices which can be used to show the Windows' desktop.

The display within the Oculus Rift interacts with Windows in the same was as a typical PC monitor. It too provides EDID information which identifies it as having a manufacturer code of 'OVR', a model ID of 'Rift DK1', and support for several display resolutions including its native 1280×800 at 60Hz.

A.2 Display Configuration

After connecting a Rift to the PC it's possible to modify the display settings through the Windows Control Panel[2].

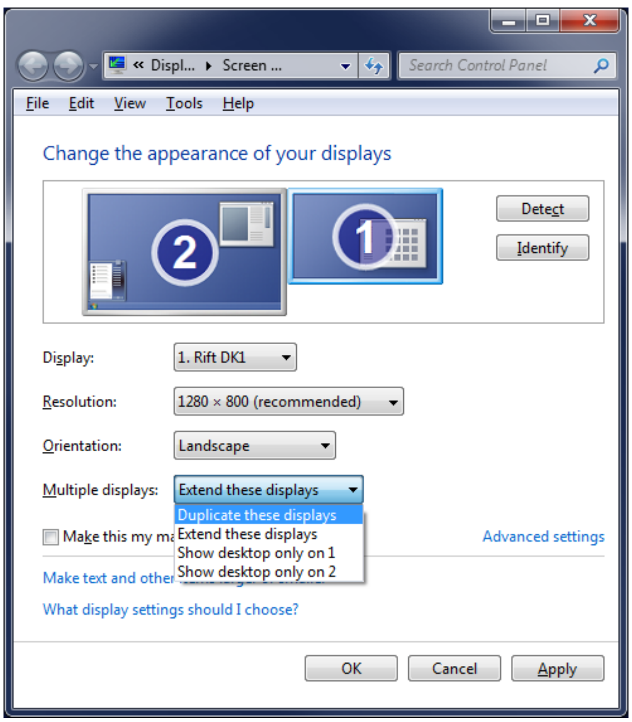


Figure 6: Screenshot of the Windows "Screen Resolution" dialog.

Figure 6 shows the "Screen Resolution" dialog for a PC with the Rift display and a PC monitor connected. In this configuration there are four modes that can be selected as show in the figure. These are duplicate mode, extended mode, and standalone mode for either of the displays.

[1] Extended Display Identification Data

[2] Under Windows 7 this can be accessed via Control Panel : All Control Panel Items : Display : Screen Resolution

A.2.1 Duplicate Display Mode

In duplicate display mode the same portion of the Windows desktop is shown on both displays, and they adopt the same resolution and orientation settings. Windows attempts to choose a resolution which is supported by both displays, while favoring the native resolutions described in the EDID information reported by the displays. Duplicate mode is a potentially viable mode in which to configure the Rift, however it suffers from vsync issues. See Section A.4 for details.

A.2.2 Extended Display Mode

In extended mode the displays show different portions of the Windows desktop. The Control Panel can be used to select the desired resolution and orientation independently for each display. Extended mode suffers from shortcomings related to the fact that the Rift is not a viable way to interact with the Windows desktop, nevertheless it is the current recommended configuration option. The shortcomings are discussed in more detail in Section A.4.

A.2.3 Standalone Display Mode

In standalone mode the Windows desktop is displayed on just one of the plugged in displays. It is possible to configure the Rift as the sole display, however is becomes impractical due to problems interacting with the Windows desktop.

A.3 Selecting A Display Device

Reading of EDID information from display devices has been found to occasionally be slow and unreliable. In addition, EDID information may be cached leading to problems with stale data. As a result, display devices may sometimes become associated with incorrect display names and resolutions, with arbitrary delays before the information becomes current.

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

The following code outlines how to create an HMD device and get the display device name:

```
DeviceManager* pManager = DeviceManager::Create();

HMDDevice* pHMD = pManager->EnumerateDevices<HMDDevice>().CreateDevice();

HMDInfo info;
pHMD->GetDeviceInfo(&HMDInfo);

char* displayName = info.DisplayDeviceName;
```

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`.

In order to display the video output on the Rift, the application needs to match the display name determined above to a device object. 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);
```

If the Rift display device is not detected but the Rift is detected through USB, then we return an empty display name string. In this case your application could attempt to locate it using additional information, for example display resolution. The GetDeviceInfo call in the API returns the native resolution of the Rift display. This can be used to match one of the available display devices.

In general, due to the uncertainty associated with identifying the Rift display device, it may make sense to incorporate functionality into your application to allow the user to choose the display manually, for example from a drop-down list of enumerated display devices. One additional root cause of the above scenario, aside from EDID issues, is that the user failed to plug in the Rift video cable. We recommend appropriate assistance inside your application to help users troubleshoot an incorrectly connected Rift device.

A.4 Rift Display Considerations

There are several considerations when it comes to managing the Rift display within the existing Windows display framework.

A.4.1 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 Direct3D "Present" call) will only occur 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 that we've found to be currently exposed in Windows or through display adapter drivers.

A.4.2 Extended Mode Problems

When extended display mode is used in conjunction with the Rift, the Windows desktop will extend partly 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 Windows 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.

A.4.3 Observing Rift Output On A Monitor

Sometimes it's desirable to be able to see the same video output on the Rift and on an external monitor. This can be 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 Windows OS.

A.4.4 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 in order 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.