Oculus VR, Inc.

Oculus Software Development Kit Overview

This document details installation, configuration, and integration of the Oculus Rift developer kit hardware and software.

Authors: Nate Mitchell, Michael Antonov

Version: 0.01

Last Edited: January 23, 2013

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

Table of Contents

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 software 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++ core of the Oculus SDK, not the Unreal Engine 3 (UE3) or Unity Integrations.

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

Oculus Rift Hardware Setup

Aside from the Oculus SDK, you'll need the Oculus Rift Development Kit (DK) hardware. The DK includes an Oculus Rift development headset, control box, required cabling, and additional pairs of lenses for different vision characteristics.

2.3 Display Specifications

- 7'' viewing area diagonally
- 1280x800 resolution (720p), split between two eyes for 640x800 per eye.
- 64 mm fixed distance between lens centers
- 60hz LCD panel
- DVI-D Single Link
- HDMI 1.3+
- USB 2.0 Full Speed+

2.4 Tracker Specifications

- Up to 1000hz sampling rate
- Gyroscope
- Magnetometer
- Accelerometer

2.5 Additional Vision Lenses

The Rift comes installed with lenses for developers 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 skip to Section 2.6: Adjusting Assembly Distance below.

There are two additional pairs of lenses included with the kit for near-sighted users. These lenses provide two generic vision settings for moderately nearsighted and very nearsighted users. While they may not work perfectly for all nearsighted users, they should help some nearsighted developers use the headset without glasses or contacts.

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

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

Note that if your eyes have special characteristics like astigmatism, the lenses may not be enough to correct your vision. In this case, we recommend wearing contact lenses or glasses. Note that using glasses will cut down on your effective field of view.

2.5.1 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 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!

If you change 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. More detailed information on adjusting the distance of the fixture in the “Distance Adjustment” section of the document.

2.6 Assembly Distance Adjustment

The headset has a distance adjustment feature that allows you to change the distance of the fixture that holds the screen and lenses for different facial characteristics or different vision lenses. For example, if the default lenses are too close to your eyes, you should adjust the fixture outward, moving the lenses and the screen away from your face. You can also adjust the assembly to give more room for eyeglasses.

Note: Everyone should take the time to adjust the headset's fixture setting to be as comfortable as possible while keeping the lenses as close as possible to the eyes.

Changing the location of the fixture does not change the optical characteristics; it simply changes where your eyes need to be for the optimal field of view. Remember that the optimal field of view is always as close to the lenses as possible without having your eye touching the lens.

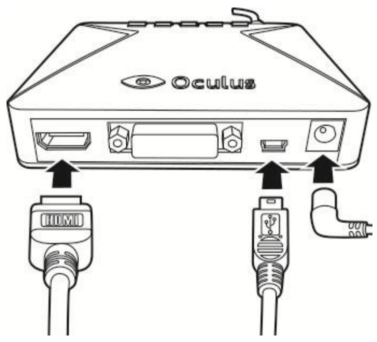
2.6.1 Adjusting the Assembly

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 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 screen mechanism toward the display to move the assembly farther away from the user. After changing one side, ensure that the other side is set to the same setting!

2.7 Control Box Setup

The headset is connected to the control box by a 6ft cable. The control box takes in the 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.

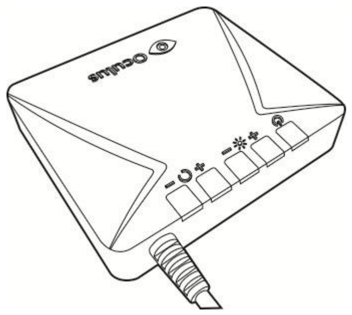
Note: There should only be one video-out cable running to the control box at a time (DVI **or** HDMI, not both).

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. Rift screen will only stay on when all three cables are connected.

2.7.1 Adjusting Brightness and Contrast

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



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

2.8 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 works best for you. Regardless of the monitor configuration, is it currently not possible to see the desktop clearly inside the Rift, as it requires stereo rendering and distortion only available while rendering the game scene.

Regardless of whether you're mirroring or extending your desktop, the resolution of the Rift should be set to 1280x800 (720p).

3 Oculus Rift SDK Setup

3.1 System Requirements

3.1.1 Operating Systems

The Oculus SDK currently supports Windows XP SP3, Vista, 7, and 8.

3.1.2 Minimum System Requirements

There are no specific computer hardware requirements for the Oculus SDK. We recommend that developers use a computer with a modern graphics card. A good benchmark to run Unreal Engine 3 and Unity at 60 frames per second (60 FPS) with vertical sync and stereo 3D enabled. If the game engine can run with these settings without dropping frames, it should be sufficient for Oculus Rift development!

- Windows XP SP3, 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 Intel HD 4000, have the shader and graphics capabilities to run minimal Rift demos, their rendering throughput may be inadequate for full-scene 60fps VR rendering with stereo and distortion. Developers targeting this hardware will need to be very conscious of scene geometry, as low-latency rendering at 60fps is critical for a usable VR experience.

3.1.3 Recommend System Requirements

- ?

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.

3.2.1 Windows

Extract the package to your computer. We recommend extracting it to a memorable location (eg. C:\Oculus).

3.3 Directory Structure

Oculus\LibOVR – Libraries, source code, projects, and makefiles for the SDK.

Oculus\LibOVR\Include

Oculus\LibOVR\Lib

Oculus\LibOVR\Src

Oculus\Samples – Samples which integrate and leverage the Oculus SDK. More information on these samples is provided in the “Samples” section of this document.

Oculus\Tools – Applications for configuring the Oculus Rift and the Oculus SDK development environment.

3.4 Compiler Settings

3.5 Makefiles, Projects, and Build Solutions

If you have the source code, you can rebuild the LibOVR libraries using the projects and solutions in the LibOVR\Projects\ directory. Projects and makefiles are divided by platform.

3.5.1 Windows

- The Visual Studio 2010 solution for LibOVR is in Projects\Win32.

3.6 Terminology

Term	***Definition***
Interpupillary distance (IPD)	Distance between the eye pupils. The average and default is 64mm, but values of 54 to 72mm are possible.

Field of view (FOV)	Full vertical viewing angle used to configure rendering, computed based on the eye distance and display size.
Aspect ratio	Relationship of screen resolution (width/height). Oculus Rift one-eye screen aspect ratio is 0.8.
K0, K1, K2	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're ready to go. Where's the best place to begin?

If you haven't already, take a moment and 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: Oculus Hardware Setup.

Now that your hardware is fully configured, the next step is test the development kit!

The SDK includes a set of basic samples designed to help developers get started quickly. The most important of these samples is **Oculus World Demo.**

4.3 Oculus Room

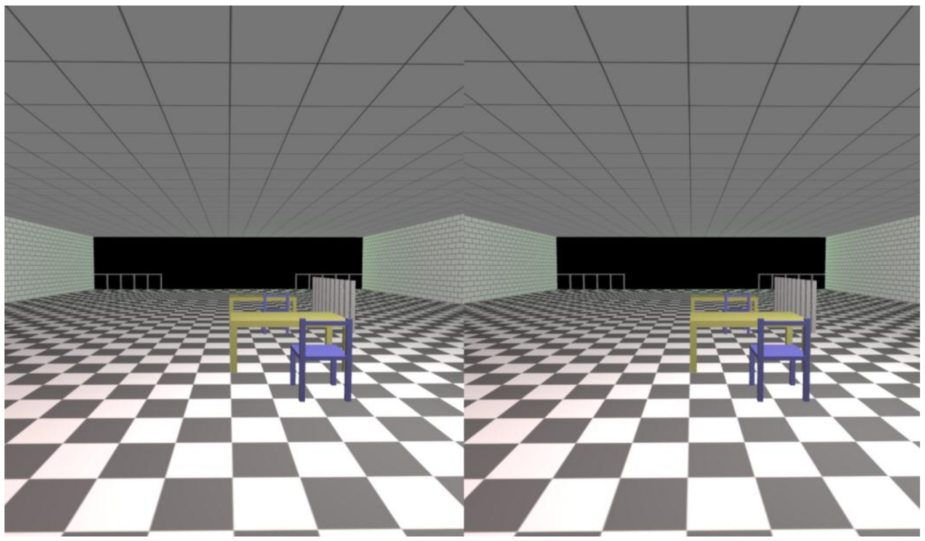


Figure 1: Screenshot of Oculus Room application.

Oculus World Demo is a simple DirectX C++ application that integrates all the features of the Oculus SDK. It serves three purposes:

1. It incorporates all the features of the SDK. This makes it an excellent application to test the Rift developer kit's basic functionality (stereo rendering, head tracking).

2. It demonstrates how to integrate the SDK with a new or existing application. Source code for the Oculus World Demo is included as part of the SDK.

3. It includes developer-centric features including toggles for stereoscopic/monoscopic rendering, various stereo implementations, and an overlay grid to measure distortion characteristics. More details on these features below.

We recommend running the pre-built Oculus World Demo as a first-step in exploring the SDK. You can find the executable in Oculus/Samples/ OculusWorldDemo.

Oculus World Demo's source code includes samples demonstrating initialization of the LibOVR core, access to Oculus devices, head-tracking, sensor fusion, head modeling, stereoscopic 3D rendering, and distortion shaders.

4.3.1 Movement Controls

Key	***Function***
W	Moves the player forward
S	Moves the player backward
A	Strafes the player left
D	Strafes the player right
Mouse Move	Controls horizontal player look
Left analog stick (Gamepad)	Move forward/ backward; strafe left/right.
Right analog stick (Gamepad)	Controls horizontal player look

4.3.2 Developer Hot-Keys

General

Key	***Function***
F1	No stereo, no distortion
F2	Stereo, no distortion
F3	Stereo and distortion
F4	Stereo (LeftDouble) and distortion
F8	Toggle MSAA
F9	Toggle HW Full-Screen (Low Latency)
F11	Fake Desktop Full-Screen (No Blinking, VSync)
R	Reset the sensor (calls SensorFusion.Reset())
Spacebar	Toggle overlay with debug info

Field of view, interpupillary distance, and aspect ratio adjustments

Key	***Function***
[	Increase field of view
]	Decrease field of view
Insert	Increase interpupillary distance
Delete	Decrease interpupillary distance
PageUp	Increase aspect ratio
PageDown	Decrease aspect ratio

Distortion adjustments

Key	***Function***
H	Decrease K1 in distortion calculation
Y	Increase K1 in distortion calculation
J	Decrease K2 in distortion calculation
U	Increase K2 in distortion calculation

4.4 Using Oculus World Demo

Once you've launched Oculus World Demo, take a moment to look around using the Rift and double check that all your hardware is working properly. You should see an image similar to the screenshot above. Press F9 or F11 key to switch rendering to Oculus Rift.

If you're having problems (eg. no image in the headset, no head tracking, etc…), take a look at the troubleshooting section of this document and the developer forums on the Oculus Developer Center. These are both excellent resources for resolving common issues.

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

Depending on your height, you may feel shorter or taller than normal. The default eye-height of the player in Oculus World Demo is 1.78 meters (5'10). If you're feeling adventurous, you can use ',' and '.' keys to adjust your height.

As you may have already concluded, the scale of the world and player is critical to an immersive VR experience. This means that players should be a realistic height and that art assets are 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.

4.5 Using the SDK Reading Beyond Oculus Room

4.5.1 Software Developers and Integration Engineers

If you're integrating the Oculus SDK into your game engine, 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.

4.5.2 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-content immediately.

The Unreal Engine 3 Integration Overview and the Unity Integration Overview, available from the Oculus Developer Center, detail all the steps required to setup your UE3/Unity + Oculus development environment.

We also recommend reading through the Oculus Best Practices Guide, which has tips, suggestions, and research 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! There are thousands of other developers out there, like you, who are 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're clearly interested in integrating the Rift with your own game engine. Awesome. We're here to help.

We've designed the Oculus SDK to be as easy to integrate as possible. The following 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. Advanced topics, including head-modeling and predictive tracking are covered later in this document.

Many of the code samples below are taken directly from Oculus World Demo's source code (available in Oculus\LibOVR\Samples\ OculusWorldDemo\). Oculus World Demo is a great place to draw sample integration code from when in doubt about a particular system or feature.

5.1 Outline of Integration Tasks

- Initialization of LibOVR

- Enumeration of Oculus devices
- Rendering
    - Distortion
    - Stereoscopic 3D
    - Projection, field of view, and other parameters
- Head-tracking
    - Reading from the Oculus tracker
    - Applying orientation, position to player view

5.2 *Initialization of LibOVR*

We initialize LibOVR's core by calling System::Init(). System::Init() will initialize LibOVR's default memory allocator. We can override the default memory allocator by passing in a reference to a different memory allocator.

```
#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's core is to create a System object and let its constructor and destructor take care of initialization and cleanup, respectively.

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

1. 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 RefCount of 1.
4. 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 HMDDevices, 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 more detailed information about a device and its capabilities. *DeviceBase::GetDeviceInfo(DeviceInfo* info)* 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 the vertical resolution, horizontal resolution, and screen size from an HMDDevice using HMDDevice::GetDeviceInfo() with an HMDInfo object (subclass of DeviceInfo).

```
HMDInfo    HMDInfo;
if (pHMD->GetDeviceInfo(&HMDInfo))
{
     MonitorName              = HMDInfo.DisplayDeviceName;
     EyeDistance              = HMDInfo.InterpupillaryDistance;
     DistortionK0             = HMDInfo.DistortionK0;
     DistortionK1             = HMDInfo.DistortionK1;
     DistortionK2             = HMDInfo.DistortionK2;
}
```

The same technique can be used to learn more about a SensorDevice.

Once we have information about the HMDDevice, the next step is to setup our rendering for the Rift.

5.3 Leveraging Sensor Data

The Oculus tracker includes a gyroscope, accelerometer, and magnetometer. When the data from these devices is fused, we can determine the orientation of the player's head in the real world and synchronize the player's virtual perspective in real-time.

The Rift's orientation is reported as a set of rotations in a right-handed coordinate system, as follows:

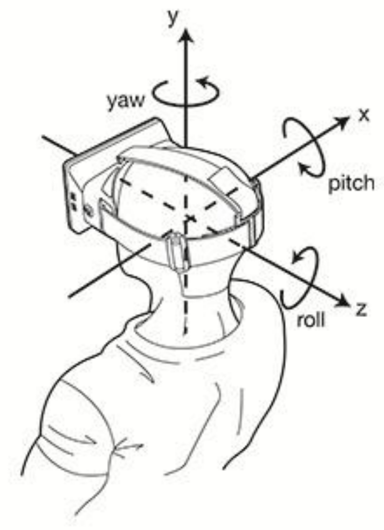


Figure 2: Diagram of Oculus Coordinate System

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) while looking in the negative direction of the axis. This means they are interpreted as follows:
- Roll is rotation around Z, counter-clockwise (tilting left) in XY plane.
- Yaw is rotation around Y, positive for turning left.
- Pitch is rotation around X, positive for pitching up.

The gyroscope, which reports the rate of rotation (angular velocity) around X, Y and Z axes in radians/second, provides the most valuable data for head orientation tracking. By constantly 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 to read from. If we have a reference to an HMD, we get a reference to the associated sensor using *HMDDevice::GetSensor();*

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

We can get more information about the sensor using *SensorDevice::GetInfo(DeviceInfo* info),* the same way we did for the HMDDevice.

The SensorFusion class accumulates Sensor notification messages to keep track of orientation, which involves integrating the gyroscope data and correcting with gravity. 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:
- We can manually pass MessageBodyFrame messages to the OnMessage() function
- We can attach SensorFusion to a SensorDevice, which will cause the SensorFusion instance to automatically handle notifications from that device.

```
SensorFusion        SFusion;
if (pSensor)
{
    SFusion.AttachToSensor(pSensor);
    SFusion.SetDelegateMessageHandler(this);
}
```

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

```
// Obtain the current accumulated orientation.
Quatf          GetOrientation() const
// Obtain the last absolute acceleration reading, in m/s^2.
SF 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 GetOrientation(). We'll use the orientation to update the virtual view to reflect the orientation of the the player's head. We'll 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. Using the Rift, the player can now look left, right, up, and down using their head. However, player's should not be required to turn their heads 180 degrees constantly; they need a way to reorient themselves so that they're 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 within those schemes when designing games for VR.

The RoomTest demo provides a source code sample for integrating Oculus head tracking with the aforementioned, standard FPS control scheme.

5.5 User Interfaces

- **User interfaces in stereo**
  - **Configuring "orthographic" surface in stereo**
    - **Convergence**
      - **Faking projection on a surface 3ft away, while considering eye distance.**

- **Useful viewable area, background colors, positioning (is this best practices?)**
  - **Depth perception**

5.6 Rendering Configuration

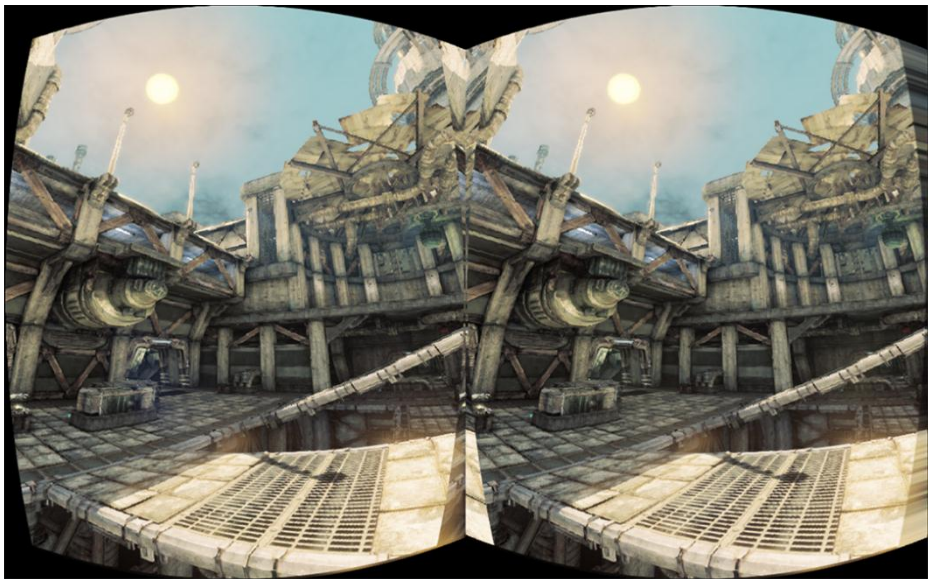


Figure 3: Unreal Engine 3® with Oculus Rift Rendering Enabled

Proper rendering inside the Rift requires that developers implement the following features:

- Individual images for each eye, rendered in a split-screen approach. Each image should take up half of the Rift's display.
- Distortion shader to correct for the optical distortion from the Rift's lenses. This will make the images appear undistorted to the user.
- Stereoscopic 3D – The images for each eye should be offset in the virtual space to provide the user with stereoscopic 3D depth. Stereoscopic 3D is crucial to the immersive nature of the Rift.

Without these collective rendering changes, the image rendered inside the Rift will appear incorrect.

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:

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

These may seem arbitrary, but they're important for a good 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.

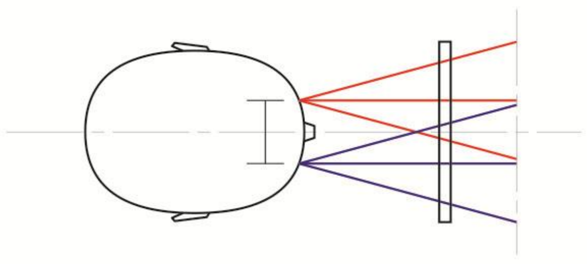
Vertical sync is also critical. Since the Rift screen covers all of the players view, screen tearing is very obvious causes trails and immediately breaks immersion.

5.6.1 Rendering Stereo

Oculus Rift requires the game scene to be rendered in split-screen stereo, with half the screen used for each eye. With Rift on, your left eye sees the left half of the screen, while the right eye sees the right half. This means that the 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();
```

Unlike stereo TVs, rendering inside of the Rift does not require Off-axis or asymmetric projection, relying on simple parallel projection instead. This means that camera setup will be very similar to the one normally used for non-stereo rendering, except you will need to shift the camera to adjust for eye location. More specifically, the 3D projection matrix used for each half of the screen will need to be computed based on the following:



- Rift Aspect Ratio
- Rift Vertical Field Of View (FOV)
- User Inter-pupillary Distance (IPD)

The Aspect Ratio and FOV are standard arguments used by graphics APIs to configure the perspective projection matrix, a task handled by Matrix4f::PerspectiveRH() function in Oculus SDK. The equivalent behavior is provided gluPerspective() utility function in OpenGL and by D3DXMatrixPerspectiveFovRH() in Direct3D. IPD is the distance between end user eye pupils that is used to properly adjust rendering so that eyes can converge on a stereo image.

As you may realize, IPD will vary from user to user, while FOV will change based on the lens cup chosen or physical eye distance from the lens. This means you can’t just hard code these values and move on – the scene that looks good for you may be uncomfortable for another user, or even become misaligned in a future version of the Rift. Instead, projection setup values need to be computed based on the geometry of the HW, considering screen size and user settings. Luckily, all of the values needed to make this possible are reported by Oculus SDK insider of the OVR::HMDInfo class. They are summarized in the table below:

Member Name	Description
HResolution, VResolution	Resolution of the entire HMD screen in pixels. Half the HResolution is used

	for each eye. The reported values are 1280x800 for DK, but we are determined to increase this in the future!
HScreenSize, VScreenSize	Physical dimensions of the entire active screen area in meters. Half HScreenSize is used for each eye. 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 eyeball to screen, in meters. This combines distances from the eye to the lens and from lens to screen. This value is needed to compute correct FOV.
LensSeparationDistance	Physical distance between lens centers, in meters. Lens centers are the centers of distortion and we will talk about them later.
InterpupillaryDistance	Configured distance between eye centers, as we discussed below.
DistortionK0, DistortionK1, DistortionK2	Radial distortion correction coefficients, discussed in the section on distortion.

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:

a) Set the viewport to cover left eye screen area.
b) Determine the Aspect Ratio and FOV based on the reported HMDInfo values.
c) Generate a projection matrix based on the Aspect Ratio and FOV.
d) Adjust projection matrix based on IPD, so that projection center matches the center of the eye.
e) Adjust view matrix to match eye location.
f) Apply projection and view matrices during rendering.

Retting up the viewport is easy - just set if to (0,0, HResolution/2, VResolution) for the left eye. In most graphics APIs, this pixel area will map to [-1,1] post-projection viewport coordinate space for both axes, with center of projection being at (0,0).

Ignoring distortion, Aspect Ratio and FOV are determined by the following formulas:

AspectRatio = (HScreenSize * 0.5) / VScreenSize

YFovRadians = 2 * atan((VScreenSize * 0.5) / EyeToScreenDistance)

Given these values, we can use Matrix4f::PerspectiveRH function to generate the perspective projection matrix that will project the 3D geometry to out viewport coordinates. The center of this projection will be at (0,0) coordinate, in the middle of the projected screen. We now need to shift this center so that it matches the eye center.

The average human IPD is 64mm, while the distance between half-screen centers on the 7" Rift DK is 149.76/2 = 74.88mm. This means that the each eye projection center needs to be translated by about

5.44mm towards the center of the device if it is to be perfectly aligned. Of course, if shifting is to modify the projection matrix, we need to convert it into viewport coordinates, as follows:

```
    float viewCenter          = HMD.HScreenSize * 0.25f;
    float eyeProjectionShift = viewCenter - InterpupillaryDistance*0.5f;
    ProjectionCenterOffset    = 4.0f * eyeProjectionShift / HMD.HScreenSize;
```

Although this last step moved projection to the center of the eye, we still need to make sure that the world that the eye sees is rendered from its own point of view. As compared to a non-stereo rendering, this means that the left camera view left by half the IPD and the right camera view to the right. This is easily achieved my adding appropriate translation as the final transformation of the view matrix. Of course, applying the IPD here will require converting it from meters to your game's coordinate system. In Oculus SDK samples this conversion is unnecessary, since the world is already scaled in meters.

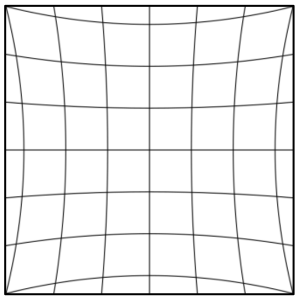
Here's a full sample of stereo setup necessary for one eye:

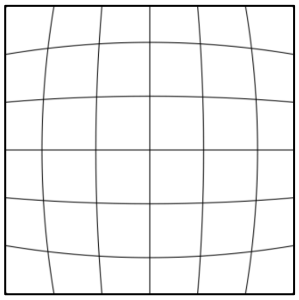
```
    HMDInfo& hmd = ...;
    Matrix4f viewCenter = ...;

    // Compute Aspect Ratio. Stereo mode cuts width in half.
    float aspect = 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,
             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,
             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 it for distortion.

5.6.2 Distortion Correction

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


Pincushion Distortion


Barrel Distortion

For the Oculus Rift DK, distortion needs to be corrected in the software by warping the image with a barrel distortion, as seen in the image on the right. When the two distortions are combined, 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:

R = k0 * r + k1 * r^3 + k2 * r^5

Here, the source sampling radius 'R' is computed based on the original radius 'r' and fixed coefficients k0, k1 and k2; these coefficients are positive for a barrel distortion. As the result of the distortion, pixels are pulled towards the center of the lens, with the amount of displacement increasing with radius. In Oculus SDK World Demo, this is implemented by the following 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);
};
```

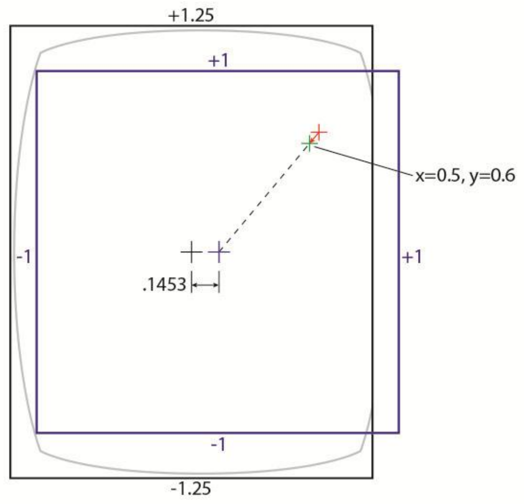
This shader is designed to run on a quad covering one half of the screen, while the input texture spans both left and right eyes. The input texture coordinates, passed in as oTextCoord, range from (0,0) for the top left corner of the Oculus screen, to (1,1) bottom right. This means that for the left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

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 that are used to scale and center the coordinates to properly 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.
- HmdWarpParam – Stores distortion coefficients.
- ScreenCenter – Texture coordinate for center of the half-screen texture; used to clamp sampling, preventing pixel leakage from one eye to the other.

The following diagram numerically illustrates the left eye distortion function coordinate range, shown as a blue rectangle, as it relates to the left eye viewport coordinates. As can we seen, the center of distortion has been shifted to the right in relation to screen center to account for narrower IPD. For the 7" screen and 64mm IPD, viewport shift is roughly 0.1453 coordinate units (which should, of course, be computed dynamically).

The diagram also illustrates how a 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. This may map to a sampling radius of ~0.68 post-distortion, illustrated by a red cross. As a result of the shader, pixels are moved towards the center of distortion, of from red to green in the diagram. The amount of displacement increases the further out you go.



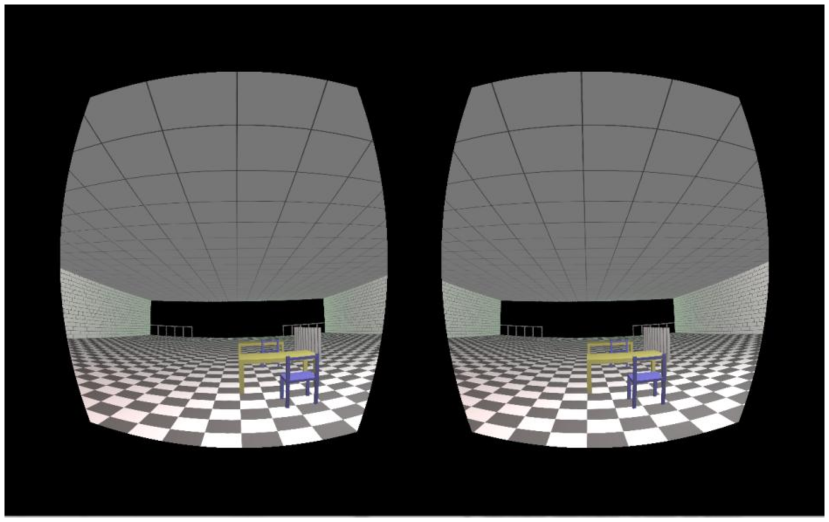
As should be obvious from this discussion, the barrel distortion pixel shader needs to run as a post-process on the entire game scene. This has several implications for rendering:

- The original scene rendering will need to be done on a render-target.
- Scene render-target will need to be larger than the final viewport, due to distortion pulling pixels in towards the center.
- FOV and image scale will need to be adjusted do accommodate for distortion.

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

5.6.3 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:



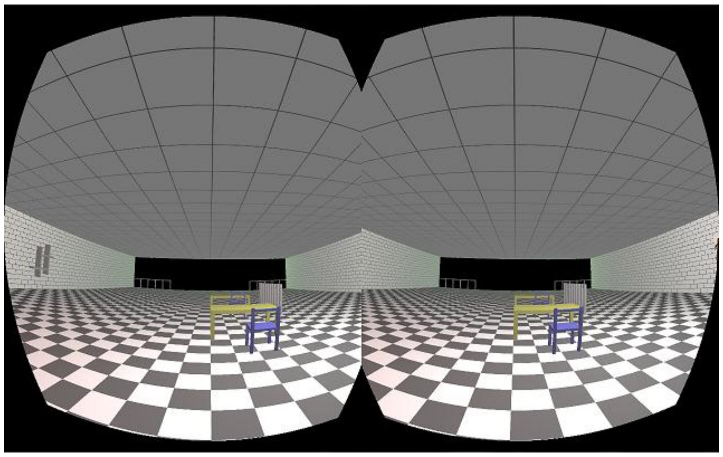
Here, the image the pixels at the edges have been pulled in towards the center, with black being displayed outside, where no pixel data was available. While this type of rendering would look ok within the Rift, a significant part of the FOV is lost as large areas of the screen go unused. How would we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

The simple solution this 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 this 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 scaling, resulting in blocky or blurry pixels at the center of the scene.

Since we really can't afford for the quality to degrade, the size of the source render target needs to be increased by the same amount to compensate. For the 1280x800 resolution of the Rift a 25% scale increase will require rendering a 1600x1000 buffer – with an obvious 1.56x increase to the rendering cost due to filling fill up more of the screen. Due to the nature of the distortion function, the area required increases exponentially with radius; however, we don't need to completely fill out the very corners of the screen where the users can't see. This means that there are some tradeoffs between the covered field of view, quality and rendering performance that can be made.

For the 7” Rift, we recommend picking the scale that fits close the left side of the screen. This gives you the maximum horizontal FOV without filling up the tops of the screen, which is not visible for most users:



For the Oculus World Demo, the actual distortion scale factor is computed inside of the StereoConfig::updateDistortionOffsetAndScale() by fitting the distortion radius to the left edge of the viewport. To make this convenient, the StereoConfig class supports distortion scale fitting to a user specified viewport coordinate with the following logic:

```
float stereoAspect = 0.5f * float(FullView.w) / float(FullView.h);
float dx           = DistortionFitX - Distortion.XCenterOffset;
float dy           = DistortionFitY / stereoAspect;
float fitRadius    = sqrt(dx * dx + dy * dy);
Distortion.Scale   = Distortion.DistortionFn(fitRadius)/fitRadius;
```

Here, the (DistortionFitX, DistortionFitY) is the left half-screen viewport-relative coordinate that will “fit” to the same pixel location in both input and output textures. This value, which is the (-1,0) for the left side of the display, is used to compute the radius to the distortion center that is scaled by the distortion function.

```
float DistortionConfig::DistortionFn (float s) const
{
    // This should match distortion equation used in shader.
    float ssq   = s * s;
    float scale = s * (K[0] + K[1] * ssq + K[2] * ssq * ssq + K[3] * ssq * ssq * ssq);
    return scale;
}
```

The output of the `Distortion. DistortionFn (fitRadius)` distortion function will return the further-out radius R’ used for the sampling, so Distortion.Scale == R’/fitRadius gives us the scale that will fit to same point in both textures. This can be proven easily as follows:

sample_uv(r) = DistortionFn(r) * (1/DistortionScale) = R’ * (r/R’) = r

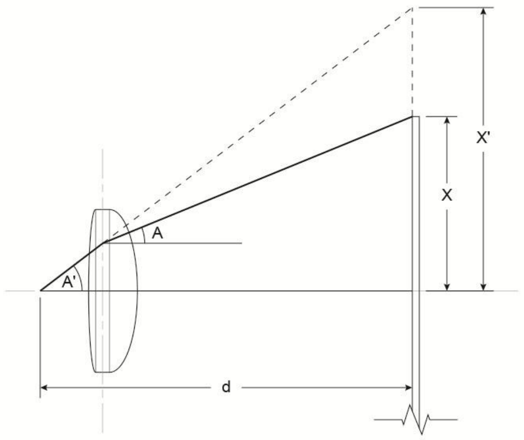
The only other value of interest in the distortion calculation Distortion.XCenterOffset, which specifies the horizontal distortion offset relative to viewport center. This can be computed easily as follows:

```
    // Compute the distortion center viewport shift based on the lens location.
    float lensOffset        = HMD.LensSeparationDistance * 0.5f;
    float lensShift         = HMD.HScreenSize * 0.25f - lensOffset;
    float lensViewportShift = 4.0f * lensShift / HMD.HScreenSize;
    Distortion.XCenterOffset= lensViewportShift;
```

5.6.4 Distortion FOV

With Distortion scale and offset properly computed, the only thing left is to compute proper field of view (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 from the lens. Here, d specifies the eye to screen distance, while X is half the vertical screen size (VScreenSize / 2).



In absence of the lens, the FOV can be easily computed simply based on the distances d and X, as described by 5.6.1:

YFovRadians = 2 * atan(X / d)

The optical lens, however, increases the perceived screen size from X to X’, where X’ can be computed through the distortion function. Since vertical FOV is twice the angle A’, it can be computed as follows:

X’ = DistortionFunction(X)
A’ = atan(X’ / d)
YFovRadians = A’ * 2

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

```
    float percievedHalfRTSize = (HMD.VScreenSize / 2) * Distortion.Scale;
    float YFov                = 2.0f * atan(percievedHalfRTSize/HMD.EyeToScreenDistance);
```

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

6 Optimization

5.7 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 the movement of player's head and the updated image displayed on screen. We call this latency loop 'movement-to-photons latency'. The more we can minimize 'movement-to-photons' latency in our 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 the movement-to-photons latency. It is the latency in the system at the hardware and software level.

- 'Perceived latency' is measured by how much latency the player **perceives** when using the headset. Perceived latency will always be **equal to or less than** the 'actual latency' depending on the player's movements and the techniques used 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. More information on predictive tracking can be found in the Advanced Topics section of this document.

At Oculus, we believe the sweet spot for compelling VR to be sub-40ms of latency. If there's more than 40ms, your brain isn't completely immersed in the environment. Obviously, the closer we are to 0ms, the better.

For the Rift developer kit, we expect the actual latency to be approximately 30-40ms depending on the image being drawn on the screen. It may be faster or slower depending on how much the pixels need to change. For example, a change from black to dark brown may take 5ms but a change from black to white may take 20ms.

Image the player is staring at a bright, snow-covered field. The player then quickly turns his head 90 degrees to the left and is suddenly cast into darkness. This represents a worst-case scenario because it will take an approximately worst-case amount of time for the pixels to make the long change from white to black.

The table below provides a breakdown of the worst-case actual latency when using Rift developer kit assuming a game running 60 FPS with vsync enabled.

Stage	***Event***	***Event Duration***	***Worst Case Total Latency***
Start	Oculus tracker sends data	N/A	0ms
Transit	Computer receives tracker data	~2ms	~2ms
Processing	Game engine renders latest frame (60 FPS w/ vysnc)	~0 – 16.67ms	~19ms
Processing	Display controller writes latest frame to LCD, top to bottom	~16.67ms	~36ms
Processing	Simultaneously, pixels switching colors adding additional latency of	~0 - 15ms	~51ms
END	Latest frame complete; presented to user	N/A	~51ms

Again, these numbers represent the actual latency assuming a game running 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.
  - Don’t buffer any frames!
  - Force flush hardware rendering to prevent video cards from buffering frames.

Advanced Topics

- **Sensor Fusion**

- **Sensor Messages**

- **Head/Neck Modeling**

```
float headBaseToEyeHeight     = 0.15f;  // Vertical height of eye from base of head
float headBaseToEyeProtrusion = 0.09f;  // Distance forward of eye from base of head
Vector3f eyeCenterInHeadFrame(0.0f, headBaseToEyeHeight, -headBaseToEyeProtrusion);
Vector3f shiftedEyePos = EyePos + rollPitchYaw.Transform(eyeCenterInHeadFrame);
// Bring the head back down to original height
shiftedEyePos.y -= eyeCenterInHeadFrame.y;
View = Matrix4f::LookAtRH(shiftedEyePos, shiftedEyePos + forward, up);
```

- **Threading**

- **Multisampling**

Troubleshooting

Q. I’m unable to download the Oculus SDK from the Oculus Developer Center.

A. Developers have to be registered for the Developer Center before they can download the SDK. If you’re still having problems downloading the latest Oculus SDK, please refer to the forums for support.