

Oculus™

**Mobile Development
Documentation**

Copyrights and Trademarks

© 2015 Oculus VR, LLC. All Rights Reserved.

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

Contents

Introduction to Mobile VR Development	5
Getting Started	5
System and Hardware Requirements	5
SDK Contents	6
Application Signing	8
VrPlatform Entitlement Checks	8
Contact	9
Device and Environment Setup	10
Introduction	10
Install the SDK	10
Device Setup	10
Setting up your System to Detect your Android Device	10
Configuring your Android Device for Debugging	11
Android Development Environment Setup	12
Android Development Software Setup for Windows	12
Android Development Software Setup for OS X	18
Troubleshooting	24
Device Troubleshooting	24
Environment Troubleshooting	25
Native Development Guide	28
Introduction	28
Native Samples	28
SDK Sample Overview	28
Importing Native Samples in Eclipse	29
Native Source Code	30
Overview	30
Native User Interface	31
Input Handling	32
Native SoundManager	32
Creating New Native Applications	33
Template Project Using The Application Framework	33
Integration with Third-Party Engines	34
Android Manifest Settings	34
Migrating from Earlier Versions	35
Mobile VR Application Development	37
Introduction to Mobile VR Design	37
Performance Advice for Early Titles	37
Frame Rate	38
Scenes	38
Resolution	39
Hardware Details	39
Universal Menu	40
Reserved User Interactions	40
Implementation Overview	41
Runtime Threads	41
Power Management	42

Fixed Clock Level API	42
Power Management and Performance	43
Power State Notification and Mitigation Strategy	44
Front Buffer Rendering	44
TimeWarp	45
TimeWarp Minimum Vsyncs	45
Consequences of not rendering at 60 FPS	46
TimeWarp Chromatic Aberration Correction	47
TimeWarp Debug Graph	47
User Interface Guidelines	47
In a Word: Stereoscopic!	48
The Infinity Problem	48
Depth In-Depth	48
Gazing Into Virtual Reality	49
Media and Assets	50
Mobile VR Media Overview	50
Introduction	50
Panoramic Stills	50
Panoramic Videos	50
Movies on Screens	51
Media Locations	53
Oculus Media Applications	53
Native VR Media Applications	53
Oculus Cinema Theater Creation	55
How to Create and Compile a Movie Theater FBX	55
Detailed Instructions	56
FBX Converter	61
Overview	61
Command-Line Interface	64
Optimization	66
Testing and Troubleshooting	68
Oculus Testing Tools and Procedures	68
Developer Mode: Running Apps Outside of the Gear VR Headset	68
Oculus Remote Monitor	68
Local Preferences	74
Android Debugging	75
Adb	75
Logcat	77
Application Performance Analysis	78
Performance Analysis	78
Application Performance	78
Rendering Performance: Tracer for OpenGL ES	81
Revision	84

Introduction to Mobile VR Development

Welcome to mobile application development for Gear VR!

This section will help you get oriented to the world of VR development before you jump in.

Getting Started

To become acquainted with the Oculus VR environment and with using Gear VR, we recommend beginning with the [Samsung Gear VR User Manual](#), which covers topics including:

- Health and Safety
- Device Features and Functionality
- Connecting the Headset
- Navigation and App Selection

Ready to start developing?

The [Device and Environment Setup Guide](#) will lead you through setting up and configuring your development environment.

If you are primarily interested in developing a Native Mobile VR Application, focus on the [Native Development Guide](#) (included in this document).

If you are primarily interested in developing a Unity Mobile VR Application, focus on the [Unity Integration Guide](#) (available at the [Developer Site](#)).

We recommend that all developers review the [Mobile VR Application Development guide](#) for performance guidelines and best practices.

For both Native and Unity development, we also recommend:

- [Android Debugging](#)
- [Performance Analysis and Performance Guidelines](#)
- [Design Guidelines](#)

You will find Gear VR submission information and other helpful documents at <https://developer.oculus.com>.

Thank you for joining us at the forefront of virtual reality!

System and Hardware Requirements

Please begin by making sure that you are using supported hardware and devices for this release of the Oculus Mobile SDK v0.5.

Operating System Requirements

The Oculus Mobile SDK currently supports Windows 7 and Mac OS X.

Minimum System Requirements

The following computer system requirements for the Oculus Mobile SDK are based on the Android SDK system requirements:

- Windows 7
- Mac OS: 10.6+ (x86 only)
- 2.0+ GHz processor
- 2 GB system RAM

Supported Devices

- Samsung Note 4

Target Device Requirements

- API Level
 - 19 (Android 4.4.2)
- VR Hardware
 - 3.0 Class 2 Bluetooth gamepad (see below)

Bluetooth Gamepad

A Bluetooth gamepad (3.0 Class 2) is necessary for testing the sample applications which come with this release. You may use the Samsung EI-GP20 gamepad or a Moga Pro, or another compatible gamepad.

For more information about the Samsung EI-GP20, including specifications and key bindings, see the following: <http://developer.samsung.com/s-console>.

Bluetooth Keyboard

It is useful (but not required) to have a Bluetooth keyboard during development. The Logitech K810 is known to function well.

SDK Contents

Included with this SDK, you will find the following:

- VrLib, the native framework for building high-performance VR Applications.
- Unity Standalone Integration for adding the VR framework to your Unity project.
- Example native and Unity Projects with source to provide a model for creating your own VR applications.
- Several pre-built sample applications, some implemented in native and some in Unity.

Sample Applications and Media

 **Note:** The sample applications included with the SDK are provided as a convenience for development purposes. Some of these apps are similar to apps available for download from the Oculus Store. Due to the potential for conflict with these versions, we do not recommend running these sample apps on the same device on which you have installed your retail Gear VR Innovator experience. Please take care to secure the retail media content bundled with the SM-R320. It will be difficult if not impossible to replace.

Table 1: Sample Native Applications

Application	Description
	A viewer for panoramic stills.
Oculus 360 Photos	
	A viewer for panoramic videos.
Oculus 360 Videos	
	Plays 2D and 3D movies in a virtual movie theatre.
Oculus Cinema	
	Loads a scene which can be navigated using a gamepad.
VrScene	

Table 2: Sample Unity Applications

Application	Description
	A simple game example in which blocks can be thrown to knock over structures and collect stars.
BlockSplosion	
	An example app that renders a scene and character from Shadowgun by Madfinger Games.
Shadowgun, by Madfinger Games	

For the Mobile SDK, we have limited pre-loaded media. Please add your own media by consulting the following table and media creation guidelines for compatibility:

Table 3: Sample Media

Application	Path for Media on the SD Card
Oculus Cinema - 2D Movie	Movies\ DCIM\ Oculus \Movies\My Videos
Oculus Cinema - 3D Movie	Movies\3D DCIM\3D Oculus\Movies\My Videos \3D
Oculus 360 Video - 360-degree panoramic video	Oculus\360Videos
Oculus 360 Photo - 360-degree static photos	Oculus\360Photos
Note: non-360 degree photos will not render properly.	

For more information on media management, see [Mobile Media VR Overview](#).

Application Signing

Application signing is a key part of the development process.

All mobile VR applications must be signed by an Android digital certificate in order to install and run on an Android device, and they must also be signed with an Oculus Signature File (osig) during development to gain complete access to the phone's full VR capabilities.

For more information on application signing, see "Create Your Signature Files" in the [Oculus Mobile Submission Guidelines](#).

VrPlatform Entitlement Checks

Mobile SDKs v0.4.3 and later include support for entitlement checking with VrPlatform. This feature is typically used to protect applications sold through the store from unauthorized distribution. During an entitlement check, the application's package name is compared with a list of applications that the currently logged-in user is entitled to use. This check is performed invisibly and automatically with the Oculus background service.

If the application package name is not found on the user's entitlement list, the application is killed, and Home automatically launches to display an error message. If the check runs into any error, applications are treated as though an entitlement were not found, in order to prevent circumvention.

Possible failures in this check are:

1. No service installed/problem connecting to service.
2. Invalid service signature.
3. Calling app is neither developer (osig) nor VR signed.
4. No user logged in.

5. User is not entitled.

 **Note:** Entitlement checking may be left enabled during development, as any development build with a valid osig file will automatically skip the check. When Oculus signs your package for release, we will test the integration to confirm that your application's entitlement is verified correctly.

Native: Java

Copy the vrplatlib.jar library from /sdk/VRPlatform/libs/ into your project's /libs/ directory and add to your project's build path.

In your activity's `onCreate` method, add the following line:

```
OVREntitlementChecker.doAutomatedCheck(this);
```

Native: C++

Follow the procedure described in the “Native: Java” section above. Every native project must subclass `VrActivity`. This is usually called `MainActivity`, so add the entitlement check in the `onCreate` as with Java.

Unity

Entitlement checking is disabled by default in Unity. To enable entitlement checking:

1. Select *Edit > Project Settings > Player*.
2. In the *PlayerSettings* window, select the Android icon and expand the *Other Settings* tab.
3. In the *Scripting Define Symbols* field, add `USE_ENTITLEMENT_CHECK`.

Verifying Entitlement Integration

You can verify your integration by checking the logcat output after starting your app.

For example:

```
D/OVREntitlementChecker(19779) : Package oculussig verified, entitlement check was skipped.
```

 **Note:** The term "oculussig" is equivalent to "osig."

Contact

Questions?

Visit our developer support forums at <https://developer.oculus.com>.

Our Support Center can be accessed at <https://support.oculus.com/>.

Device and Environment Setup

A guide to setting up your device and environment for mobile VR application development.

Introduction

Welcome to the Oculus VR Mobile Software Development Kit! This SDK will demonstrate how to implement high-performance, high-quality and fully-immersive virtual reality applications for Samsung Gear VR.

Install the SDK

Begin by installing the mobile SDK archive.

The mobile SDK is composed of a compressed archive in .zip format which contains both source and media files: ovr_mobile_sdk_<version>.zip.

Once downloaded, extract the .zip file into a directory of your choice (e.g., C:\Oculus\Mobile).

Device Setup

This section will provide information on how to setup your supported device and gamepad for running, debugging, and testing your Gear VR application.

Please review the [System and Hardware Requirements](#) above for the list of supported devices for this SDK release.



Note: This information is accurate at the time of publication of this document. Unfortunately, we cannot guarantee the consistency or reliability of any of the third-party applications discussed in these pages, nor can we offer support for any of the third-party applications we describe.

Setting up your System to Detect your Android Device

You must set up your system to detect your Android device over USB in order to run, debug, and test your application on an Android device.

If the device is not automatically detected by your system when connected over USB, you may need to manually update the drivers. More information can be found in the “Using Hardware Devices” section at <http://developer.android.com/tools/device.html>.

Windows

If you are developing on Windows, you need to install a USB driver for adb. For an installation guide and links to OEM drivers, see the Android [OEM USB Drivers](#) document.

Samsung Android drivers may be found on their developer site: <http://developer.samsung.com/android/tools-sdks/Samsung-Android-USB-Driver-for-Windows>

Windows may automatically detect the correct device and install the appropriate driver when you connect your device to a USB port on your computer. However, if Windows is unable to detect your device, you may still need to update the drivers through the Windows Device Manager, even if your device was automatically detected.

Access the Device Manager through the Windows Control Panel. If the device was automatically detected, it will show up under *Portable Devices* in the Device Manager. Otherwise, look under *Other Devices* in the Device Manager and select the device to manually update the driver.

To verify that the driver successfully recognized the device, open a command prompt and type the command:

```
adb devices
```

 **Note:** You will need to successfully setup your Android development environment in order to use this command. For more information, see the next section: [Android Development Environment Setup](#)

If the device does not show up, verify that the device is turned on with enough battery power, and that the driver is installed properly.

Mac OS

If you are developing on Mac OS X, you do not need to install USB drivers.

Your Samsung device may display a notification recommending you install [Android File Transfer](#), a handy application for transferring files between OS X and Android.

Configuring your Android Device for Debugging

In order to test and debug applications on your Android device, you will need to enable specific developer options on the device.

Developer Options

Developer options may be found under: *Home -> All Apps -> Settings -> System -> Developer options*.

Developer options may be hidden by default. If so, you can expose these options with the following steps:

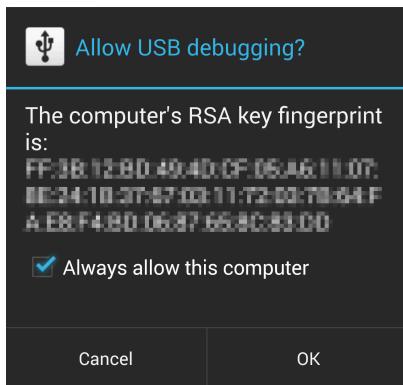
1. Go to *Home -> All Apps -> Settings -> System -> About device*.
2. Scroll down to *Build number*.
3. Press *Build number* seven times.

You should be informed that *Developer options* has been enabled.

Once you have found *Developer options*, enable the following:

USB Debugging: This will allow the tools to install and launch deployed apps over USB.

You should see the screen shown on the accompanying figure.



 **Note:** If the above screen does not appear, ensure that your system recognizes the device and toggle *USB Debugging* off then back on.

Check *Always allow this computer* and hit *OK*.

To purge the authorized whitelist for USB Debugging, press *Revoke USB debugging authorizations* from the *Developer options* menu and press *OK*.

Allow mock locations: This will allow you to send mock location information to the device (convenient for apps which use Location Based Services).

Verify apps via USB: This will check installed apps from ADB/ADT for harmful behavior.

Display Options

The following display options are found in: *Home -> Apps -> Settings -> Sound and Display*

Lock screen/Screen Security/Screen lock: When set to *None* the Home screen is instantly available, without swipe or password. Useful to quickly get in and out of the phone.

Display/Screen timeout: Set the time to your desired duration. Useful if you are not actively accessing the device but wish to keep the screen awake longer than the default 30 seconds.

See [Android Debugging](#) for more information.

Android Development Environment Setup

This section describes setup and configuration of the Android Development Environment necessary for building Oculus Android mobile applications.

 **Note:** As of February 2015, Android has officially deprecated the Eclipse-based SDK in favor of the Android Studio system. We are in the process of updating our developer instructions for integration with their new IDE. Until that is complete, we recommend that developers continue to download and install the tools and versions referenced in this guide. When this guide was published, they were still available from download at the specified addresses. Please check back for updates.

Android Development Software Setup for Windows

In order to develop Android applications, you must have the following software installed on your system:

1. Java Development Kit (JDK)
2. Android Development Tools (ADT) Bundle

3. Android Native Development Kit (NDK)
4. Apache Ant

Java Development Kit (JDK)

The Java Development Kit is a prerequisite for the Android Eclipse IDE (which comes with the ADT Bundle) as well as Apache Ant.

The latest version which has been tested with this release is JDK 8u11. You may download the appropriate version for your OS from the following location: <http://www.oracle.com/technetwork/java/javase/downloads/java-archive-javase8-2177648.html>

 **Note:** As of October 2014, the Windows x86 version of the JDK appears to be incompatible with Eclipse and cannot be used.

Once downloaded and installed, add the environment variable JAVA_HOME, which should be set to the JDK install location. Depending on where it was installed, it may look like:

- C:\Program Files\Java\jdk1.8.0_11 if you have installed the x64 version, or
- C:\Program Files (x86)\Java\jdk1.8.0_11 if you have installed the x86 version.

Make sure your JAVA_HOME variable does not include quotation marks. Based on the default installation path of Java SE 8u11, the correct syntax when using `set` from the command line is:

```
set JAVA_HOME=C:\Program Files (x86)\Java\jdk1.8.0_11
```

Do not use quotes with the `set` command, even though the path has a space in it. Also, *be sure to use your actual path* - this example is based on the default installation path, but your path may vary.

Android Development Tools Bundle

The Android Development Tools (ADT) Bundle includes everything needed to begin developing Java Android Apps:

- Android SDK Tools
- Android Platform Tools
- Latest Android Platform
- Eclipse with integrated ADT Plugin
- Latest System Image for Emulator

The ADT bundle comes in either a 32-bit or 64-bit version. This must match the JDK option you selected above. Download the appropriate version of the ADT Bundle at the following locations:

- [32-bit .zip download](#)
- [64-bit .zip download](#)

Once downloaded, unpack the zip file and save to your Android development folder, e.g.:

```
C:\Dev\Android\adt-bundle-<os_platform>\
```

Add the ADT SDK tools and platform-tools to your PATH, e.g.

- C:\Dev\Android\android_adt_bundle_20140702\sdk\tools
- C:\Dev\Android\android_adt_bundle_20140702\sdk\platform-tools

Add the environment variable ANDROID_HOME which should be set to your Android SDK location, e.g.:

- C:\Dev\Android\android_adt_bundle_20140702\sdk

```
set ANDROID_HOME=C:\Dev\Android\adt-bundle-windows-x86-20140702\sdk
```

Installing Additional Packages and Tools

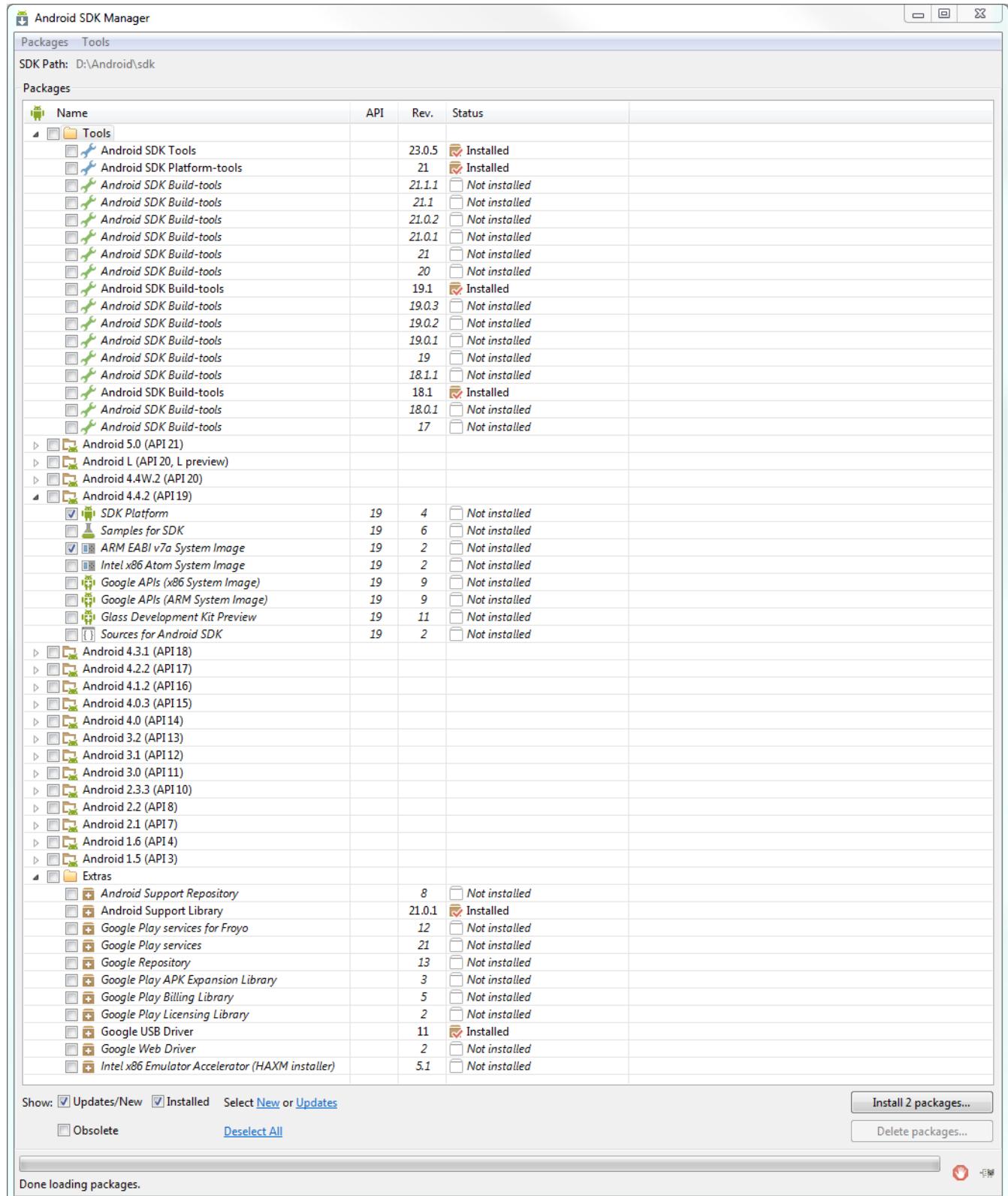
The Android SDK does not include everything you will need for development by default. You must download additional packages via the SDK Manager before you begin.

 **Note:** Before launching the SDK Manager, make sure you have installed the JDK as outlined above. SDK Manager may fail to launch if it has not been installed.

1. Launch the Android SDK Manager located at: C:\Dev\Android\adt-bundle-<os_platform>\SDKManager.exe
You may also launch it via Eclipse by selecting the SDK Manager in the toolbar.
2. Under the *Tools* section within *Packages*, select the following (if they are unselected):
 - Android SDK Tools Rev. 23.0.5
 - Android SDK Platform-tools Rev. 21
 - Android SDK Build-tools Rev. 20
3. Our current Android build target is Android 4.4.2 (API 19). Select at least the following under the API 19 section:
 - SDK Platform
 - ARM EABI v7a System Image
 - Sources for Android SDK: optional, but invaluable for dealing with the Android API.
4. Finally, under *Extras* at the bottom, select the following:
 - Android Support Library Rev. 21.0.1

- Google USB Driver Rev. 11

Figure 1: Android SDK Manager



5. Open Android 4.4.2 (API 19) and select the items for *SDK Platform* and *ARM EABI v7a System Image*. Note that Oculus Mobile SDK projects use API level 19.

6. Click *Install X Packages*, where X is the number of selected packages (it may vary based on what needs to be updated).
7. On the next dialog, choose *Accept License*.
8. Click *Install* to install and update packages.

You may install additional packages above API 19 if you wish. Android SDK Tools 23.0.2, Android SDK Platform-tools 20 and Android SDK Build-tools 20 have all been verified to work correctly with the SDK.

If you have problems compiling after an update, close the Android SDK Manager and re-open it to make sure you have everything required for the installed packages. In some cases you may get the warning “Could not load definitions from resource emma_ant.properties”. This is usually due to missing components in the install or improper file permissions. If the problem persists after verifying everything is up to date, try deleting the local.properties files from your project folders and updating them in each project folder with the command:

```
android update project -p .
```

Then rebuild.

Verify Eclipse Configuration

Verify that Eclipse has the proper location for your Android Development Tools.

Launch the Eclipse executable which came with the ADT bundle, e.g.: C:\Dev\Android\android_adt_bundle_20140702\eclipse\eclipse.exe.

1. In the Eclipse menu go to *Window -> Preferences -> Android*.
2. The Android Preferences *SDK Location* should be set to the location where you installed the ADT bundle SDK in the previous section, e.g.: C:\Dev\Android\android_adt_bundle_20140702\sdk.

Android Native Development Kit

The Android Native Development Kit (NDK) is a toolset that allows you to implement parts of your app using native code languages such as C and C++. It is used extensively by the sample applications which come with this release.

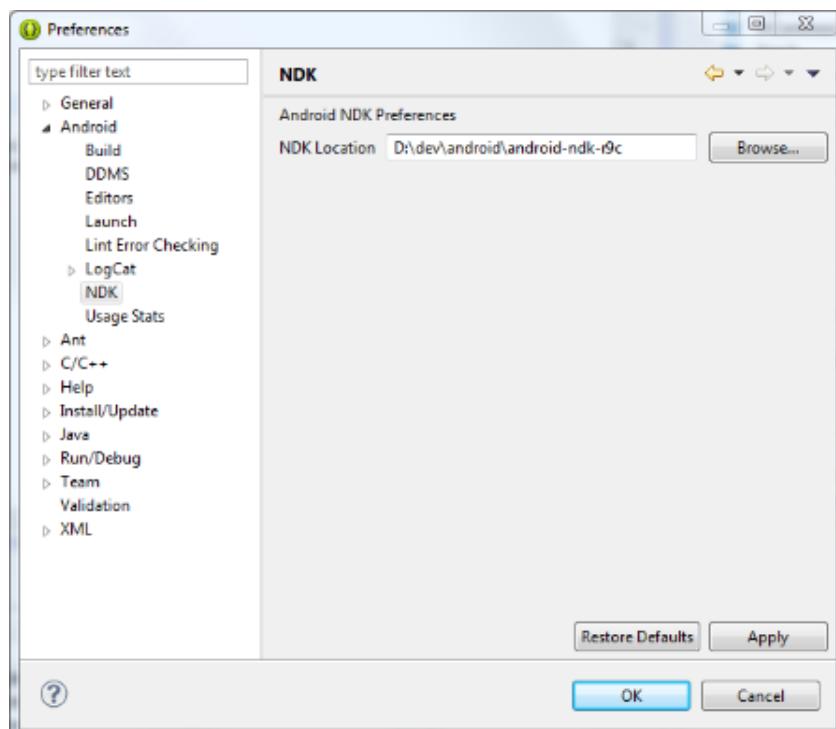
1. Download the latest version of NDK at the following location:<https://developer.android.com/tools/sdk/ndk/index.html>
2. Once downloaded, install NDK to your Android development folder, e.g.: C:\Dev\Android\android-ndk-r10c\.
3. Add the NDK location to your PATH, e.g.: C:\Dev\Android\android-ndk-r10c\.
4. Add the environment variable ANDROID_NDK, which should be set to your Android NDK location, e.g.: C:\Dev\Android\android-ndk-r10c.

```
set ANDROID_NDK=C:\Dev\Android\android-ndk-r10c
```

Configure Eclipse for NDK Usage

1. Start Eclipse from the location it was installed to, e.g.: C:\Dev\Android\adt-bundle-windows-x86_64-20140702\eclipse\eclipse.exe.
2. Download and install the Eclipse ADT Plugin.
 1. In the Eclipse menu go to: *Help -> Install New Software*.
 2. Click *Add* in the top-right corner.
 3. In the *Add Repository* dialog that appears, enter “ADT Plugin” in the *Name* field and the following URL: <https://dl-ssl.google.com/android/eclipse>
 4. Click *OK*.

5. In the *Available Software* dialog, select the checkbox for *Developer Tools* and click *Next*.
 6. Click *OK*.
 7. In the next window, you will see a list of tools to be download. Click *Next*.
 8. Read and accept the license agreements, then click *Finish*. Click *OK* if you get any prompts regarding the security and authenticity of the software.
 9. When the installation completes, restart Eclipse.
3. Configure the NDK path:
1. In the Eclipse menu, go to: *Window -> Preferences -> Android*.
 2. Under *Android*, select *NDK*.
 3. Set the *NDK Location* field to the directory where the NDK is installed.



If the NDK option under the Android section is missing, something went wrong with the Eclipse ADT Plugin installation. Full instructions and troubleshooting information may be found here: <http://developer.android.com/sdk/installing/installing-adt.html#Troubleshooting>

Apache Ant

Apache Ant is a Java library and command-line build system. It provides a number of built-in tasks which simplify building Java projects. The Apache Ant project is part of the Apache Software Foundation.

The latest version which has been tested with this release is Apache Ant 1.9.3 and is available for download at the following location: <http://ant.apache.org/bindownload.cgi>

 **Note:** The Ant download page is explicit about verifying the Ant binary, but this is not strictly necessary for using Ant or for getting Android development up and running.

Once downloaded, unpack the zip file and save to your Android development folder, e.g.: C:/Dev/Android/apache-ant-1.9.3.

Next, add the Ant bin folder to your PATH, e.g.: C:\Dev\Android\apache-ant-1.9.3\bin.

For more information on using Ant to automate building Android Apps, see: <http://www.androidengineer.com/2010/06/using-ant-to-automate-building-android.html>

Android Development Software Setup for OS X

In order to develop Android applications, you must have the following software installed on your system:

1. Xcode
2. Android Development Tools (ADT) Bundle
3. Android Native Development Kit (NDK)
4. Apache Ant

Xcode

Before installing any Android development tools, you must install Xcode.

Once Xcode is installed, some of the following steps (such as installing Apache Ant or the JDK) may be unnecessary - some versions of OS X (10.5 and later) include Apache Ant, and some do not. On Mavericks 10.9, Ant does not appear to be installed by default or included with Xcode 5.0.2.

Java Development Kit

The Java Development Kit (JDK 8) is a prerequisite for the Android Eclipse IDE (included with the ADT Bundle) as well as Apache Ant.

Install the JDK if it is not already present on your system. If you already installed Xcode, this step may be unnecessary.

The latest version tested with this release is JDK 8 - it may be downloaded at the following location: <http://www.oracle.com/technetwork/java/javase/downloads/java-archive-javase8-2177648.html>.

Android Development Tools Bundle

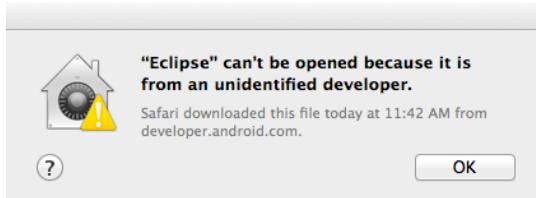
The Android Development Tools ADT Bundle includes almost everything needed to begin developing Android Apps.

The ADT Bundle includes:

- Android SDK Tools
- Android Platform Tools
- Latest Android Platform
- Eclipse with integrated ADT Plugin
- Latest System Image for Emulator

 **Note:** The tilde character followed by a forward slash (~/) is shorthand for the current user's home folder. It is used throughout this section.

1. In your home folder, create a new folder named "dev". To get to home folder in OS X, open Finder and press CMD+Shift+H.
2. Download the ADT package at http://dl.google.com/android/adt/adt-bundle-mac-x86_64-20140702.zip.
3. Once downloaded, unzip the ADT archive (if necessary) and save to your "dev" folder.
4. Browse to ~/dev/adt-bundle-<version>/eclipse and double-click the Eclipse icon. You may receive a warning like the following:

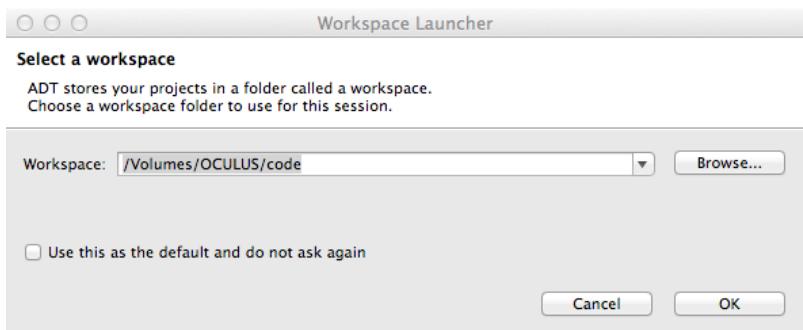


 **Note:** Please refer to Apple's knowledge base regarding this security feature and safeguards for apps downloaded and installed from the internet.

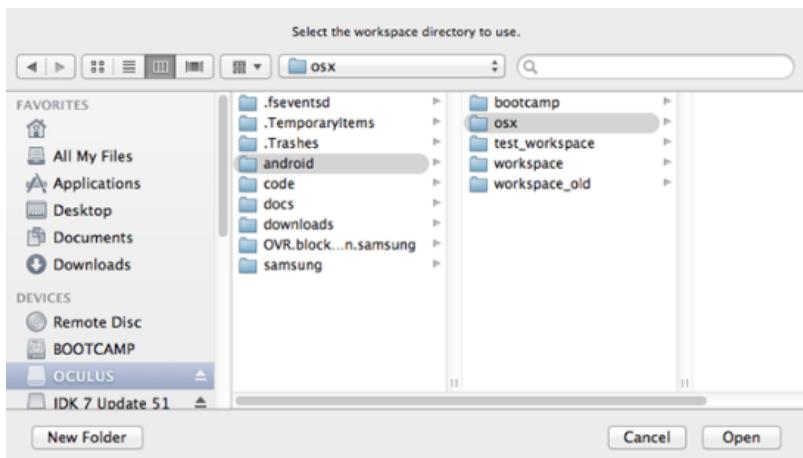
- To remedy this, right click the Eclipse icon and choose Open. You will get a warning again, but it will now present an *Open* option:



- Select *Open* to start Eclipse. If you get a message that Eclipse needs JRE 6 in order to run, allow it to download and install JRE 6. Next, you should see a window asking you to select your workspace:



- Select *Browse* and locate or create the directory for your workspace:

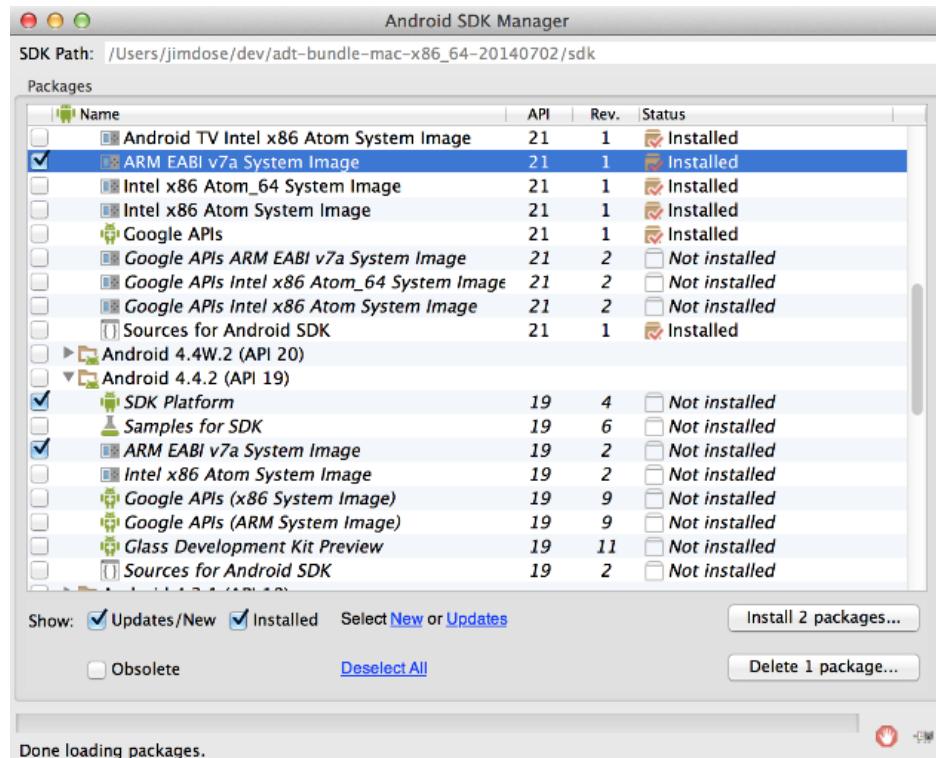


- If you plan to always use this same workspace, you can select the checkbox labeled *Use this as the default and do not ask again*. Choose *OK*.

Installing Additional Packages and Tools

The Android SDK does not include everything you will need for development by default. You will need to download additional packages via the SDK Manager before you begin.

In Eclipse go to *Window -> Android SDK Manager* to open a window similar to the following:



1. Note that some packages are already selected for installation by default. Leave these projects selected.
2. If not already selected under *Tools*, select *Android SDK Tools*, *Android SDK Platform-tools*, and *Android SDK Build-tools*.
3. Open *Android 4.4.2 (API 19)* and select the items for *SDK Platform* and *ARM EABI v7a System Image* under it.
4. Click *Install X Packages*, where *X* is the number of selected packages (it may vary based on what needs to be updated).
5. On the next dialog, choose *Accept License*.
6. Click *Install* to install and update packages.

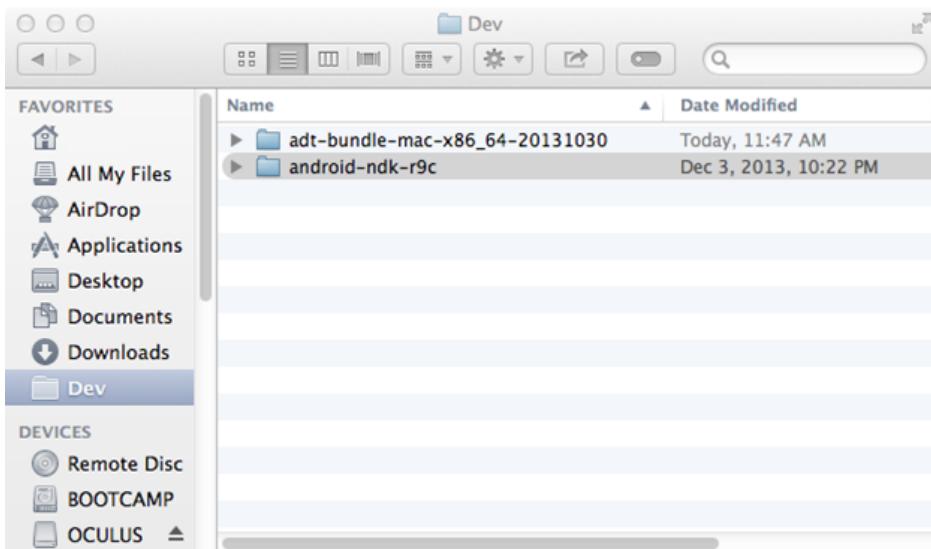
You may install additional packages above API 19 if you wish. *Android SDK Tools 23.0.2*, *Android SDK Platform-tools 20* and *Android SDK Build-tools 20* have all been verified to work correctly with the SDK.

Android Native Development Kit

The 2.4 Android Native Development Kit (NDK) is a toolset that allows you to implement parts of your app using native code languages such as C and C++. It is used extensively by the sample applications which come with this release.

The latest version which has been tested with this release is NDK 10 - it is available for download at the following location: <https://developer.android.com/tools/sdk/ndk/index.html>.

Once downloaded, extract the NDK to your home/dev folder (~/dev). Your dev folder should look something like the following:

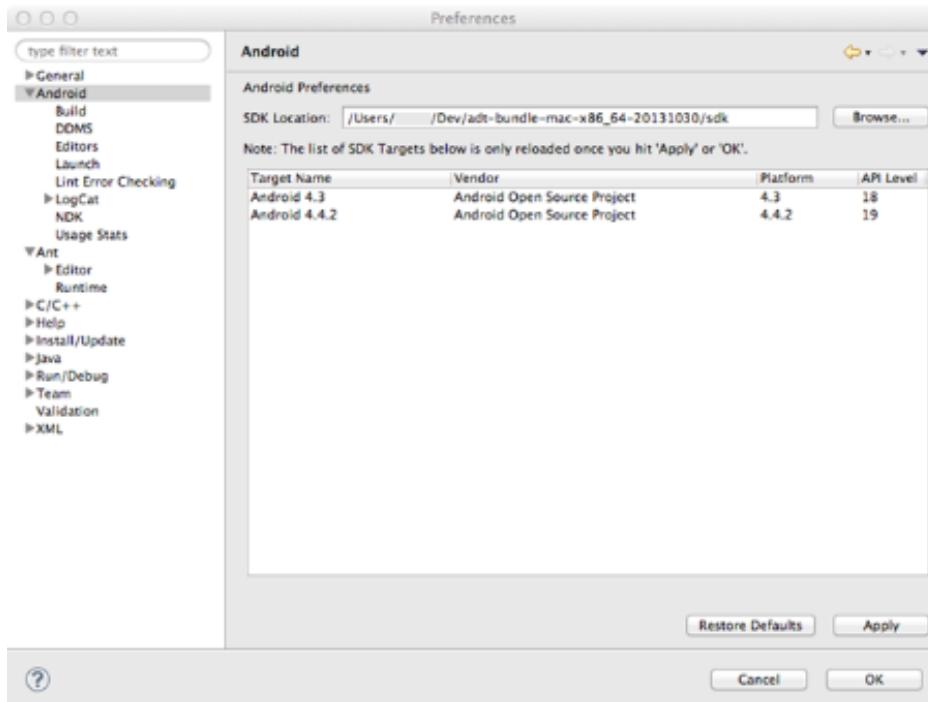


Note that the ADT bundle and the NDK are extracted into their own folders within the Dev folder (in this case, folders that were in the root of their archive). The names of the folders, where the ADT bundle and the NDK reside, are not vitally important. As long as they are in separate folders and are not extracted directly into the Dev folder, any conflict between the two packages can be avoided. It is recommended you use the above naming scheme so that there is no question which version of the ADT bundle and the NDK are installed.

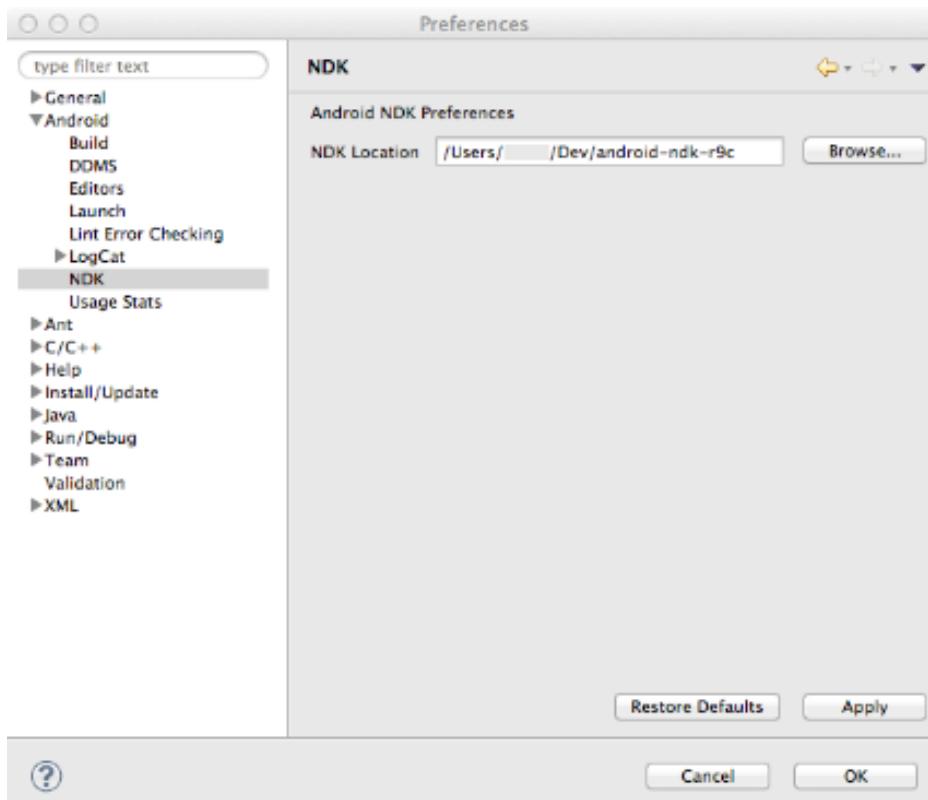
You can read more about installation and use of the NDK here: <http://developer.android.com/tools/sdk/ndk/index.html#installing>.

Configure Eclipse for NDK Usage

1. Launch Eclipse and go to the *ADT -> Preferences* window.
2. Under *Android*, verify that Eclipse knows the location of the Android Development Tools. If the *NDK option* is missing in the *Preferences -> Android* section, update the Eclipse ADT Plugin as follows.
 1. Download and install the Eclipse ADT Plugin.
 - a. In the Eclipse menu go to: *Help->Install New Software*.
 - b. Click *Add* in the top-right corner.
 - c. In the *Add Repository* dialog that appears, enter “ADT Plugin” in the *Name* field and the following URL: <https://dl-ssl.google.com/android/eclipse>
 - d. Click *OK*.
 - e. In the *Available Software* dialog, select the checkbox for *Developer Tools* and click *Next*.
 - f. Click *OK*.
 - g. In the next window, you will see a list of tools to be download. Click *Next*.
 - h. Read and accept the license agreements, then click *Finish*. Click *OK* if you get any prompts regarding the security and authenticity of the software.
 - i. When the installation completes, restart Eclipse.
 2. Configure the NDK path.
 - a. In the Eclipse menu go to: *Window -> Preferences -> Android*.
 - b. Under *Android* select *NDK*.
 - c. Set the *NDK Location* field to the directory where the NDK is installed.



3. Under *Android -> NDK* set the NDK folder to the location where you installed the NDK:



4. To set environment variables specifying the locations of Android SDK and NDK on your system, and to add Android tools to your global path:

1. Launch a terminal.

- At the prompt type the following three lines:

```
echo 'export ANDROID_HOME=~/dev/adt-bundle-mac-x86_64-20140702/sdk' >> ~/.profile
echo 'export ANDROID_NDK=~/dev/android-ndk-r9c' >> ~/.profile
echo 'export PATH=$PATH:$ANDROID_HOME/tools:$ANDROID_HOME/platform-tools:$ANDROID_NDK' >> ~/.profile
```

- To verify that the Android NDK environment is set up correctly, launch a new terminal and go to the samples/hello-jni folder in the Android NDK. Execute the command `ndk-build`, which will compile the sample hello-jni application if everything is set up correctly.

Apache Ant

Apache Ant is a Java library and command-line build system. It provides a number of built-in tasks which simplify building Java projects. The Apache Ant project is part of the Apache Software Foundation.

- Download Apache Ant here: <http://ant.apache.org/bindownload.cgi>.

The latest version which has been tested with this release is Apache Ant 1.9.3. If you have already installed Xcode, you may be able to skip this step.

- Once downloaded, unzip Ant (or copy/move it if your web browser auto-expands .zip files for you) into your Dev folder alongside the ADT bundle and the NDK. As with the other packages, Ant should be in its own folder within Dev.
- Set environment variables to specify the locations of JRE and ADT on your system, and add Ant to your path.

- Launch a terminal

- Type the following three lines at the prompt:

```
echo 'export ANT_HOME=~/dev/apache-ant-1.9.3' >> ~/.profile
echo 'export JAVA_HOME=$(/usr/libexec/java_home)' >> ~/.profile
echo 'export PATH="$PATH":~/dev/apache-ant-1.9.3/bin' >> ~/.profile
```

For additional information about the JAVA_HOME folder and how to determine its location, see:

- <http://stackoverflow.com/questions/18144660/what-is-path-of-jdk-on-mac>
- http://www.mkyong.com/java/how-to-set-java_home-environment-variable-on-mac-os-x/

- Verify the \$PATH variable and *_HOME variables are set correctly.

- Launch a new terminal window (from Terminal, press Cmd+N). BE SURE to do this from a new Terminal window. The original Terminal window where you set the environment variables will not have an updated version of the environment variables, because it hasn't re-run ~/.profile.

- At the prompt type:

```
echo $PATH
```

- You should see the full path with your Ant bin folder at the end.

- At the same terminal window prompt, type:

```
echo $ANT_HOME
```

- Verify the Ant home path is the folder you installed it to. This particular path should NOT have /bin on the end.

- To ensure the Ant binaries are accessible through the path, type `ant -version`

- You should see the Ant version information output as in the screen capture below.

If you receive any errors, verify that the paths are all correct. If you have to correct anything, you can edit the `~/.profile` file from Terminal using the following command:

```
sudo nano ~/.profile
```

```
localhost-MacBook-Pro-2:~ user0$ echo $PATH
/usr/bin:/bin:/sbin:/usr/local/bin:/Users/user0/dev/apache-ant-1.9.3/bin
localhost-MacBook-Pro-2:~ user0$ echo $ANT_HOME
/Users/user0/dev/apache-ant-1.9.3
localhost-MacBook-Pro-2:~ user0$ ant -version
Apache Ant(TM) version 1.9.3 compiled on December 23 2013
localhost-MacBook-Pro-2:~ user0$
```

You should see something like the following screen where you can use the arrow keys to navigate the file and edit text:

```
GNU nano 2.0.6          File: /Users/User0/.profile
export ANT_HOME=~/dev/apache-ant-1.9.3
export JAVA_HOME=$(/usr/libexec/java_home)
export PATH="$PATH":~/dev/apache-ant-1.9.3/bin
```

When you are finished editing, press **Ctrl+X** to save, answer **Y** to overwrite, and press **Enter** to select the current file.

Troubleshooting

Troubleshooting your device and environment.

Device Troubleshooting

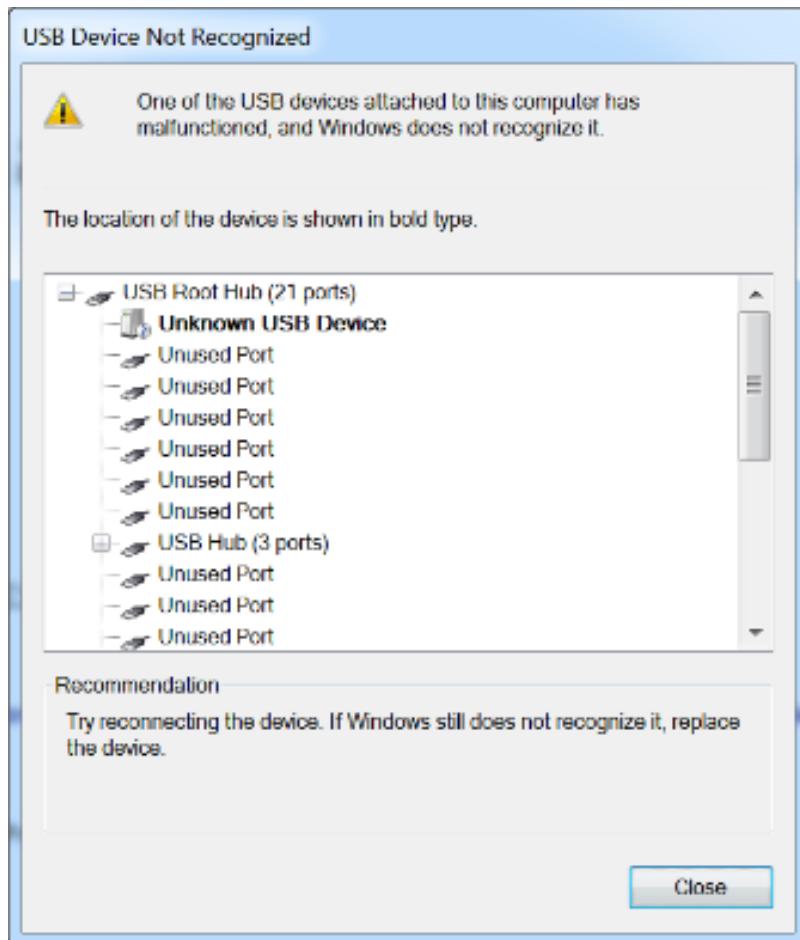
Troubleshooting your Samsung device.

View is tilted

If the headset view is tilted to the left or right, there is probably an issue with sensor calibration. Take off the headset and place it face down on a flat surface for around 5 seconds to prompt the device to reorient the sensors. When you put the device back on your head, the view should be level.

Device not (or no longer) recognized

Even if a device worked fine previously, it may enter a state in which it is no longer recognized. When using Windows you may get the following dialog:



In most cases, rebooting the device should be sufficient.

Windows appears to sometimes auto-update or auto-rollback the correct Samsung USB driver when the phone is reconnected. If this occurs, you should see a Windows notification that it is installing a driver when you connect your phone via USB. Should this happen, reinstall the correct driver.

Device regularly reconnects

The device may enter a state in which it reconnects regularly, or in the middle of sessions.

In Eclipse you may get the following message:

```
[<data> <time> - DeviceMonitor] Adb connection Error: An existing connection was  
forcibly closed by the remote host
```

Solution: reboot the device.

Environment Troubleshooting

Troubleshooting your Android environment.

Ant build failure

When building your project using the Ant build tools, you may run into build.xml errors such as the following:

```
C:\Dev\Android\android_adt_bundle_20140702\sdk\tools\ant\build.xml:653: The
following error occurred while executing this line:
```

```
C:\Dev\Android\android_adt_bundle_20140702\sdk\tools\ant\build.xml:698: null
returned: 1
```

If you add the `-verbose` option to the Ant build, you will see an error such as:

```
invalid resource directory name: C:\Dev\Android\VrTestApp\bin\res/crunch
```

This appears to happen for projects which refer to library projects - there is currently no explanation why it occurs. To fix the problem, delete the bin/res/crunch folders that are generated in VrLib/ and VrTestApp/.

Eclipse Problems

Spaces in Android tool paths.

Make sure there are NO SPACES in any of your Android tool paths. If there are, follow all of the installation instructions from the start without spaces in any paths. The exception is the Java Development Kit on Windows, which by default installs to "C:\Program Files (x86)" for 32-bit or "C:\Program Files" for 64-bit. Those paths are acceptable as long as JAVA_HOME is set appropriately (without quotes -- see the note in the section IV.1.1).

Build/clean error due to missing NDK path.

Eclipse sometimes loses the NDK path even though you set the path during the software installation process. When this happens you may get a build/clean error similar to the one below:

```
**** Clean-only build of configuration Default for project VrExperiments ****
sh ndk-build clean
Error: Cannot run program "sh": Launching failed
**** Build Finished ****
```

Solution: set the NDK path at: *Menu -> Window -> Preferences -> Android -> NDK*.

Unable to launch due to errors

Although you just successfully built your application, you get a message that the project contains errors that need to be fixed before you can launch the application.

Solution: go to the Problems tab (typically at the bottom) and delete all the errors.

Compatibility Warnings When Starting Eclipse

If you already have an existing installation of the Android Development Tools prior to 23.0.2, you may see compatibility errors when starting Eclipse depending on which packages you downloaded from the SDK Manager. In some cases you may be able to fix these compatibility issues by going to *Eclipse -> Help -> Install New Software*. Add the URL <https://dl-ssl.google.com/android/eclipse/> in the *Work with:* field and select *Add*. Name the repository "Eclipse", select the packages that appear under *Developer Tools* and choose *Next*. If you receive installation errors, you can try to resolve these (you should be offered some resolution steps) or try installing one package at a time. If you continue to receive errors, try the steps below.

Make sure you have downloaded the latest ADT bundle and verify that all of the required environment variables (including PATH) point to the location of the new bundle. Launch the SDK Manager from the new ADT bundle installation and download the required packages as indicated in section IV. Launch Eclipse from the new bundle, go to *Window -> Preferences* and verify that the SDK Location field there points to the location where you

installed the new version of the ADT bundle. Re-verify that all paths in Eclipse and all system environment variables are correct and reboot.

Missing NDK Option in Android Menu

At this point you should verify that the NDK path is set correctly in the *Eclipse Settings* as shown in section IV.1.5 (Windows), IV.2.5 (OS X), or IV.3.3. (Ubuntu). If NDK does not appear under *Preferences -> Android*, make sure you have all of the latest development tools installed.

In Eclipse, go to *Help -> Install New Software*. In some versions of ADT the existing links for Android Development Tools Update Site appear to be broken (the URL is paired with the wrong URL). If any of the pre-defined links for *Work with:* field do not allow proper installation, click *Add...*, input "eclipse" for Name: and in the *Location* field type:

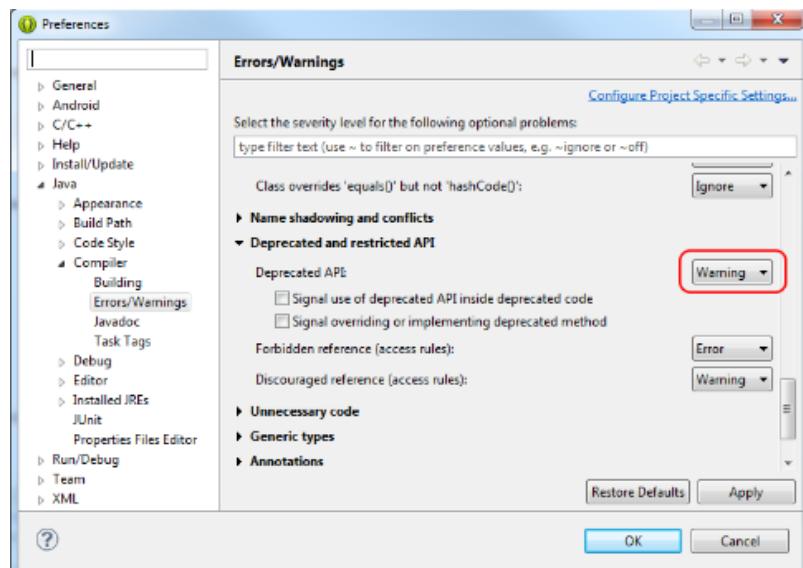
```
https://dl-ssl.google.com/android/eclipse/
```

Download all of the packages under *Development Tools* and complete the installation steps.

Java Build Errors

The SDK currently uses some deprecated Java interfaces. If you see build errors (listed under Errors and preceded by a red X icon) related to Java deprecation, such as "The type ActivityGroup is deprecated", then set deprecation errors to warnings. Go to *Window -> Preferences -> Java -> Compiler -> Errors/Warnings* and scroll down to the *Deprecated and Restricted API* section. Next to *Deprecated API*, change *Error* to *Warning* or *Ignore* as shown below:

Figure 2: Eclipse



Native Development Guide

Welcome to the Native Development Guide

Introduction

This document describes the Oculus native mobile SDK, which provides a framework for creating your own native virtual reality applications.

This SDK includes several sample projects which provide an overview of the native source code.

While native software development is comparatively rudimentary, it is also closer to the metal and allows implementing very high performance virtual reality experiences without the overhead of elaborate environments such as you would find with a typical game engine. It is not feature rich, but it provides the basic infrastructure you will need to get started with your own high-performance virtual reality experience.

Before using this SDK and documentation, please review the [Device and Environment Setup Guide](#) to ensure that you are using a supported device, and to ensure that your Android Development Environment and mobile device are configured and set up to build and run Android VR applications.

Native Samples

This release provides a simple application framework and a set of sample projects that prove out virtual reality application development on the Android platform and demonstrate high-performance virtual reality experiences on mobile devices.

SDK Sample Overview

The Mobile SDK includes several sample apps, some of which are native implementations built with the Android Native Development Kit (NDK), and some of which are implemented using Unity. For a list of these sample apps see [Mobile SDK Contents](#).

To install these sample applications and associated data to your mobile device, perform the following steps:

1. Connect to the device via USB.
2. Run `installtophone.bat` from your Oculus Mobile SDK directory, e.g.: `C:\Dev\Oculus\Mobile\installToPhone.bat`.
3. Issue the following commands from `C:\Dev\Oculus\Mobile\`:

```
adb push sdcard_SDK /sdcard/  
adb install -r *.apk
```

4. Alternately, you may copy the files directly onto your mobile device using Windows Explorer, which may be faster in some cases.

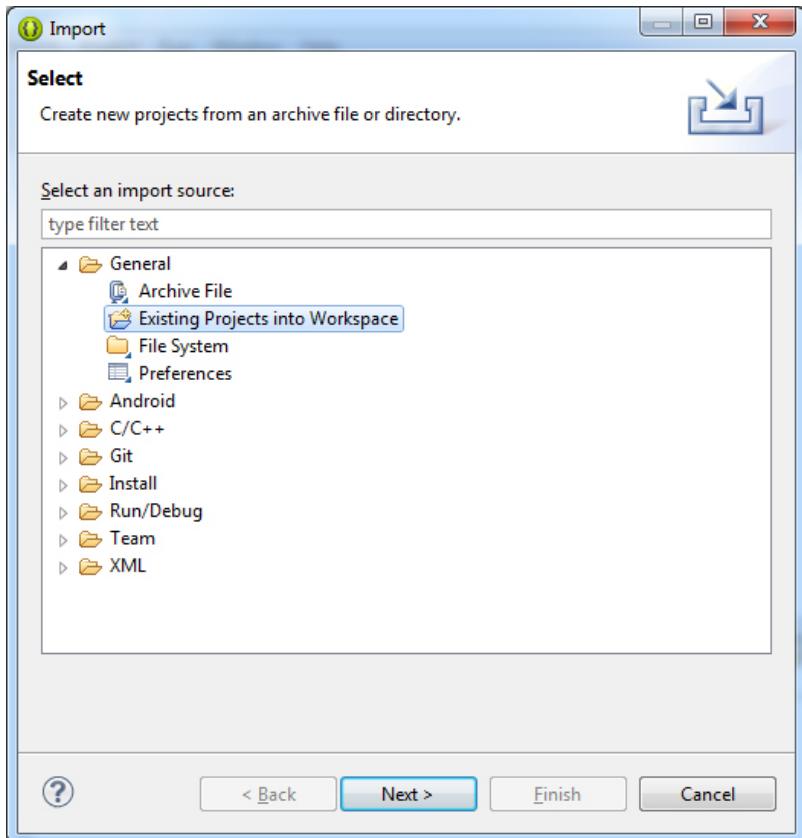
Importing Native Samples in Eclipse

To work with or modify the Oculus VR sample application source code, the samples must first be imported into Eclipse.

Make sure you have followed the configuration steps in the [Device and Environment Setup Guide](#) before importing or you will receive errors.

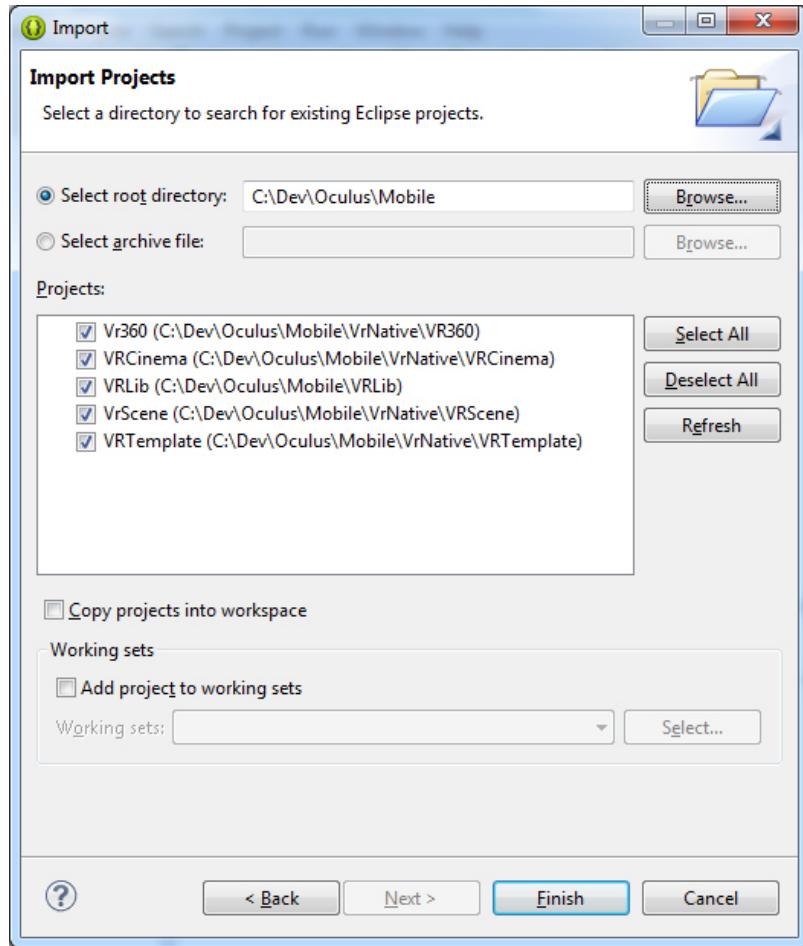
1. In the Eclipse menu, choose *File -> Import*. You should see a dialog similar to that shown.
2. Open the *General* folder and select *Existing Projects into Workspace*, then click *Next*.

Figure 3: Eclipse



3. You should now see the *Import Projects* dialog:

Figure 4: Eclipse



4. Browse to the location containing the Oculus mobile software that you wish to import. Specifying a *Root Directory* and keeping all of your apps under this folder will enumerate all of your apps.
5. Select the apps to import. If you have not previously imported the VRLib app, we recommend doing so, as other apps depend on it.
6. After selecting apps to import, make sure you have not checked *Copy projects into workspace* if you intend to directly work on the existing apps.
7. Select *Finish*.



Note: We recommend that you disable building automatically when Eclipse loads your workspace. To do so, go to *Project -> Build Automatically* and deselect the option.

Native Source Code

This section describes Gear VR native source code development.

Overview

The native source code can be broken down into three main components as shown in the following table.

Component	Description	Source code folder
Virtual Reality API	Minimal API for VR.	VRLib/jni/VrApi/
Integrations	Glue for third party engines/environments such as Unity.	VRLib/jni/Integrations/
Application Framework	Framework and support code for native applications.	VRLib/jni/

The Virtual Reality API provides a minimal set of entry points for enabling VR rendering in native applications and third-party engines.

Examples of integrations built with third party engines may be found in the VRLib/jni/Integrations/ folder.

The Application Framework includes code for rendering, user interfaces, sound playback, and more. It is not meant to provide the functionality of a full-fledged engine, but it does provide structure and a lot of useful building blocks for building native applications.

Native User Interface

Applications linking to VrLib have access to the VrMenu interface code. The VrMenu system is contained in VRLib/jni/VRMenu. Menus are represented as a generic hierarchy of menu objects. Each object has a local transform relative to its parent and local bounds.

VRMenu may be used to implement menus in a native application, such as the Folder Browser control used in Oculus 360 Photos and Oculus 360 Videos.

Parameter Type	Description
eVRMenuObjectType const type	An enumeration indicating the type of the object. Currently this can be VRMENU_CONTAINER, VRMENU_STATIC and VRMENU_BUTTON.
VRMenuComponentParms const & components	An object pointing to function objects that are called when a particular button event occurs.
VRMenuSurfaceParms const & surfaceParms	Specifies image maps for the menu item. Each image map is specified along with a SURFACE_TEXTURE_* parameter. The combination of surface texture types determines the shaders used to render the menu item surface. There are several possible configurations: diffuse only, diffuse + additive, diffuse + color ramp, and diffuse + color ramp + color ramp target, et cetera. See jni/VRMenu/VRMenuObjectLocal.cpp for details.
char const * text	The text that will be rendered for the item.
Posef const & LocalPose	The position of the item relative to its parent.
Vector3f const & localScale	The scale of the item relative to its parent.
VRMenuld_t id	A unique identifier for this object. This can be any value as long as it is unique. Negative values are used by the default menu items and should be avoided unless the default app menu items are completely overloaded.

Parameter Type	Description
VRMenuObjectFlags_t const flags	Flags that determine behavior of the object after creation.
VRMenuObjectInitFlags const initFlags	Flags that govern aspects of the creation process but are not referenced after menu object creation.

Input Handling

Input to the application is intercepted in the Java code in `VrActivity.java` in the `dispatchKeyEvent()` method. If the event is NOT of type `ACTION_DOWN` or `ACTION_UP`, the event passed to the default `dispatchKeyEvent()` handler. If this is a volume up or down action, it is handled in Java code. Otherwise the key is passed to the `buttonEvent()` method. `buttonEvent()` passes the event to `nativeKeyEvent()`.

`nativeKeyEvent()` posts the message to an event queue, which is then handled by the `AppLocal::Command()` method. This calls `AppLocal::KeyEvent()` with the event parameters.

For key events other than the back key, the event is first passed to the GUI System, where any system menus or menus created by the application have a chance to consume it in their `OnEvent_Impl` implementation by returning `MSG_STATUS_CONSUMED`, or pass it to other menus or systems by returning `MSG_STATUS_ALIVE`.

If not consumed by a menu, the native application using VRLib is given the chance to consume the event by passing it through `VrAppInterface::OnKeyEvent()`. Native applications using the VRLib framework should overload this method in their implementation of `VrAppInterface` to receive key events. If `OnKeyEvent()` returns true, VRLib assumes the application consumed the event and will not act upon it.

`AppLocal::KeyEvent()` is also partly responsible for detecting special back key actions, such as long-press and double-tap and for initiating the wait cursor timer when the back key is held. Because tracking these special states requires tracking time intervals, raw back key events are consumed in `AppLocal::KeyEvent()` but are re-issued from `AppLocal::VrThreadFunction()` with special event type qualifiers (`KEY_EVENT_LONG_PRESS`, `KEY_EVENT_DOUBLE_TAP` and `KEY_EVENT_SHORT_PRESS`).

`VrThreadFunction()` calls `BackKeyState.Update()` to determine when one of these events should fire. When a back key event fires it receives special handling depending. If a double-tap is detected, the Gear VR sensor state will be reset and the back key double-tap event consumed. If the key is not consumed by those cases, the universal menu will get a chance to consume the key. Otherwise, the back key event is passed to the application through `VrAppInterface::OnKeyEvent()` as a normal key press.

Native SoundManager

Use `SoundManager`, a simple sound asset management class, to easily replace sound assets without recompilation.

`SoundManager` is controlled by a JSON file in which sounds are mapped as key-value pairs, where a value is the actual path to the wav file. For example:

```
"sv_touch_active" : "sv_touch_active.wav"
```

In code, we use the key to play the sound, which `SoundManger` then resolves to the actual asset. For example:

```
app->PlaySound( "sv_touch_active" );
```

The string “`sv_touch_active`” is first passed to `SoundManager`, which resolves it to an absolute path, as long as the corresponding key was found during initialization.

The following two paths specify whether the sound file is in the res/raw folder of VrLib (e.g., for sounds that may be played from any app, such as default sounds or Universal Menu sounds), or the assets folder of a specific app:

```
"res/raw/ sv_touch_active.wav"
```

or

```
"assets/ sv_touch_active.wav"
```

If SoundManager fails to resolve the passed-in string within the App->PlaySound function, the string is passed to playSoundPoolSound in the VrActivity class in Java. In playSoundPoolSound, we first try to play the passed-in sound from res/raw, and if that fails, from the current assets folder. If that also fails, we attempt to play it as an absolute path. The latter allows for sounds to be played from the phone's internal memory or SD card.

The JSON file loaded by SoundManager determines which assets are used with the following scheme:

1. Try to load sounds_assets.json in the Oculus folder on the sdcard: sdcard/Oculus/sound_assets.json
2. If we fail to find the above file, we the load the following two files in this order: res/raw/sound_assets.json
assets/sound_assets.json

The loading of the sound_assets.json in the first case allows for a definition file and sound assets to be placed on the SD card in the *Oculus* folder during sound development. The sounds may be placed into folders if desired, as long as the relative path is included in the definition.

For example, if we define the following in sdcard/Oculus/sound_assets.json:

```
"sv_touch_active" : "SoundDev/my_new_sound.wav"
```

we would replace all instances of that sound being played with our new sound within the SoundDev folder.

The loading of the two asset definition files in the second step allows for overriding the VrLib sound definitions, including disabling sounds by redefining their asset as the empty string. For example:

```
"sv_touch_active" : ""
```

The above key-value pair, if defined in an app's sound_assets.json (placed in its asset folder), will disable that sound completely, even though it is still played by VrLib code.

Creating New Native Applications

This section will get you started with creating new native applications for Gear VR.

Template Project Using The Application Framework

The VrTemplate project is set up for exploratory work and as a model for setting up similar native applications. VrTemplate is the best starting place for creating your own mobile app.

We include "make_new_project.bat" and "make_new_project.sh" to simplify renaming the project name already set in the template. To use these scripts, the sed utility must be installed. Please refer to the [sed documentation](#) and follow the brief instructions below.

- Download the setup program from <http://gnuwin32.sourceforge.net/downlinks/sed.php>.

- Add the PATH where you installed sed. e.g.: C:\Program Files (x86)\GnuWin32\bin.
- Alternately, you can copy these files into the VrTemplate folder: libiconv2.dll, libintl3.dll, regex2.dll, sed.exe.

To create your own mobile app based on VrTemplate, perform the following steps:

1. Run C:\Dev\Oculus\Mobile\VRTemplate\make_new_project.bat, passing the name of your new app and your company as parameters. For example:

```
make_new_project.bat VrTestApp YourCompanyName
```

2. Your new project will now be located in C:\Dev\Oculus\Mobile\VrTestApp. The packageName will be set to com.YourCompanyName.VrTestApp.
3. Navigate to your new project directory. With your Android device connected, execute the “run.bat” located inside your test app directory to verify everything is working.
4. run.bat should build your code, install it to the device, and launch your app on the device. One parameter controls the build type:
 - run debug: Generates a build for debugging.
 - run release: Generates a build for release.
 - run clean: Removes files generated by the build.

The Java file VrTemplate/src/oculus/MainActivity.java handles loading the native library that was linked against VrLib, then calls nativeSetAppInterface() to allow the C++ code to register the subclass of VrApplInterface that defines the application. See VrLib/jni/App.h for comments on the interface.

The standard Oculus convenience classes for string, vector, matrix, array, et cetera are available in the Oculus LibOVR, located at VrLib/jni/LibOVR/. You will also find convenience code for OpenGL ES texture, program, geometry, and scene processing that is used by the demos.

Integration with Third-Party Engines

It is generally easy to pull code into the VrTemplate project, but some knowledge of the various VR-related systems is necessary to integrate them directly with your own engine.

The file VrLib/jni/VrApi/VrApi.h provides the minimum API for VR applications. The code in VrLib/jni/Integrations/Unity/UnityPlugin.cpp may be used as a model for integrating mobile VR in your own engine.

Android Manifest Settings

Configure your manifest with the necessary VR settings, as shown in the following manifest segment.

```
<application android:theme="@android:style/Theme.Black.NoTitleBar.Fullscreen" >
<meta-data android:name="com.samsung.android.vr.application.mode"
    android:value="vr_only"/>
<activity android:screenOrientation="landscape"
    android:launchMode="singleTask"
    android:configChanges="screenSize|orientation|keyboardHidden|keyboard">
</activity>
</application>
<uses-sdk android:minSdkVersion="19" android:targetSdkVersion="19" />
<uses-feature android:glEsVersion="0x00030000" />
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
```

- The Android theme should be set to the solid black theme for comfort during application transitioning: Theme.Black.NoTitleBar.Fullscreen
- The vr_only meta data tag should be added for VR mode detection.

- The required screen orientation is landscape: `android:screenOrientation="landscape"`
- It is recommended that your configChanges are as follows: `android:configChanges="screenSize|orientation|keyboardHidden|keyboard"`
- The `minSdkVersion` and `targetSdkVersion` are set to the API level supported by the device. For the current set of devices, the API level is 19.
- Do not add the `noHistory` attribute to your manifest.
- `READ_EXTERNAL_STORAGE` permission is needed for reading the appropriate lens distortion file for the device.

Applications submission requirements may require additional adjustments to the manifest. Please refer to the submission guidelines available in our [Developer Center](#).

Migrating from Earlier Versions

This section will help you upgrade from previous SDK revisions.

The 0.5.0 Mobile SDK introduces several major changes that necessitate updates to the Native app interface and development workflow.

The Universal Menu is now no longer a part of VrLib. This allows modifications to the Universal Menu without requiring each app to upgrade to the latest SDK. The Universal Menu is now part of the Oculus System Activities application and is downloaded and updated alongside Oculus Home and Horizon. Make sure you update your version of Home in order to test your application with the new Universal Menu.

The VrApi has been streamlined and re-factored for future modularity and ease of development.

Native Developers

The native AppInterface has changed slightly to handle more robust intent passing. `nativeSetAppInterface` and `OneTimeInit` now take three additional arguments:

1. Public static native long `nativeSetAppInterface(VrActivity act, String fromPackageNameString, String commandString, String uriString);`
2. Virtual void `OneTimeInit(const char * fromPackage, const char * launchIntentJSON, const char * launchIntentURI);`
3. Your Vr Activity `OnCreate()` call will need to pass in the following data to `nativeSetAppInterface`:

```
Intent intent = getIntent();
String commandString = VrLib.getCommandStringFromIntent( intent );
String fromPackageNameString = VrLib.getPackageNameStringFromIntent( intent );
String uriString = VrLib.getUriStringFromIntent( intent );
appPtr = nativeSetAppInterface( this, fromPackageNameString, commandString,
    uriString );
```

Additionally, you will no longer need the following in your manifest file:

```
<activity android:name="com.oculusvr.vrlib.PlatformActivity"
    android:theme="@android:style/Theme.Black.NoTitleBar.Fullscreen"
    android:launchMode="singleTask"
    android:screenOrientation="landscape"
    android:configChanges=" screenSize|orientation|keyboardHidden|keyboard" >
<uses-permission android:name="android.permission.CAMERA" />
```

Head-Neck Model (HNM) Changes

The default eye height is now 1.675 m and the default interpupillary distance is now 64 mm, in agreement with the default values for the Desktop SDK.

The HNM has been re-parameterized to use a 2D vector (depth/height) rather than angle/length which is also in agreement with the Desktop SDK.

Any references to HeadModelLength or HeadModelAngle should be replaced as follows:

```
HeadModelLength->HeadModelDepth  
HeadModelAngle->HeadModelHeight
```

Mobile VR Application Development

Welcome to the Guidelines and Performance Guide

Introduction to Mobile VR Design

This section contains guidelines for VR application development in the unique domain of mobile development.

Performance Advice for Early Titles

Be conservative on performance. Even though two threads are dedicated to the VR application, a lot happens on Android systems that we can't control, and performance has more of a statistical character than we would like. Some background tasks even use the GPU occasionally. Pushing right up to the limit will undoubtedly cause more frame drops, and make the experience less pleasant.

You aren't going to be able to pull off graphics effects under these performance constraints that people haven't seen years ago on other platforms, so don't try to compete there. The magic of a VR experience comes from interesting things happening in well-composed scenes, and the graphics should largely try not to call attention to themselves.

Even if you consistently hold 60 FPS, more aggressive drawing consumes more battery power, and subtle improvements in visual quality generally aren't worth taking 20 minutes off the battery life for a title.

Keep rendering straightforward. Draw everything to one view, in a single pass for each mesh. Tricks with resetting the depth buffer and multiple camera layers are bad for VR, regardless of their performance issues. If the geometry doesn't work correctly - all rendered into a single view (FPS hands, et cetera) - then it will cause perception issues in VR, and you should fix the design.

You can't handle a lot of blending for performance reasons. If you have limited navigation capabilities in the title and can guarantee that the effects will never cover the entire screen, then you will be ok.

Don't use alpha tested / pixel discard transparency -- the aliasing will be awful, and performance can still be problematic. Coverage from alpha can help, but designing a title that doesn't require a lot of cut out geometry is even better.

Most VR scenes should be built to work with 16 bit depth buffer resolution and 2x MSAA. If your world is mostly pre-lit to compressed textures, there will be little difference between 16 and 32 bit color buffers.

Favor modest "scenes" instead of "open worlds". There are both theoretical and pragmatic reasons why you should, at least in the near term. The first generation of titles should be all about the low hanging fruit, not the challenges.

The best-looking scenes will be uniquely textured models. You can load quite a lot of textures -- 128 Megs of textures is okay. With global illumination baked into the textures, or data actually sampled from the real world, you can make reasonably photo realistic scenes that still run 60 FPS stereo. The contrast with much lower fidelity dynamic elements may be jarring, so there are important stylistic decisions to be made.

Panoramic photos make excellent and efficient backdrops for scenes. If you aren't too picky about global illumination, allowing them to be swapped out is often nice. Full image-based lighting models aren't performance-practical for entire scenes, but are probably okay for characters that can't cover the screen.

Frame Rate

Thanks to the [asynchronous TimeWarp](#), looking around in the Gear VR will always be smooth and judder-free at 60 FPS, regardless of how fast or slow the application is rendering. This does not mean that performance is no longer a concern, but it gives a lot more margin in normal operation, and improves the experience for applications that do not hold perfectly at 60 FPS.

If an application does not consistently run at 60 FPS, then animating objects move choppier, rapid head turns pull some black in at the edges, player movement doesn't feel as smooth, and gamepad turning looks especially bad. However, the asynchronous TimeWarp does not require emptying the GPU pipeline and makes it easier to hold 60 FPS than without.

Drawing anything that is stuck to the view will look bad if the frame rate is not held at 60 FPS, because it will only move on eye frame updates, instead of on every video frame. Don't make heads up displays. If something needs to stay in front of the player, like a floating GUI panel, leave it stationary most of the time, and have it quickly rush back to center when necessary, instead of dragging it continuously with the head orientation.

Scenes

Per scene targets:

- 50k to 100k triangles
- 50k to 100k vertices
- 50 to 100 draw calls

An application may be able to render more triangles by using very simple vertex and fragment programs, minimizing overdraw, and reducing the number of draw calls down to a dozen. However, lots of small details and silhouette edges may result in visible aliasing despite MSAA.

It is good to be conservative! The quality of a virtual reality experience is not just determined by the quality of the rendered images. Low latency and high frame rates are just as important in delivering a high quality, fully immersive experience, if not more so.

Keep an eye on the vertex count because vertex processing is not free on a mobile GPU with a tiling architecture. The number of vertices in a scene is expected to be in the same ballpark as the number of triangles. In a typical scene, the number of vertices should not exceed twice the number of triangles. To reduce the number of unique vertices, remove vertex attributes that are not necessary for rendering.

Textures are ideally stored with 4 bits per texel in ETC2 format for improved rendering performance and an 8x storage space reduction over 32-bit RGBA textures. Loading up to 512 MB of textures is feasible, but the limited storage space available on mobile devices needs to be considered. For a uniquely textured environment in an application with limited mobility, it is reasonable to load 128 MB of textures.

Baking specular and reflections directly into the textures works well for applications with limited mobility. The aliasing from dynamic shader based specular on bumped mapped surfaces is often a net negative in VR, but simple, smooth shapes can still benefit from dynamic specular in some cases.

Dynamic lighting with dynamic shadows is usually not a good idea. Many of the good techniques require using the depth buffer (expensive on a mobile GPU) with a tiling architecture. Rendering a shadow buffer for a single parallel light in a scene is feasible, but baked lighting and shadowing usually results in better quality.

To be able to render many triangles, it is important to reduce overdraw as much as possible. In scenes with overdraw, it is important that the opaque geometry is rendered front-to-back to significantly reduce the number of shading operations. Scenes that will only be displayed from a single viewpoint can be statically sorted to guarantee front-to-back rendering on a per triangle basis. Scenes that can be viewed from multiple vantage points may need to be broken up into reasonably sized blocks of geometry that will be sorted front-to-back dynamically at run-time.

Resolution

Due to distortion from the optics, the perceived size of a pixel on the screen varies across the screen. Conveniently, the highest resolution is in the center of the screen where it does the most good, but even with a 2560x1440 screen, pixels are still large compared to a conventional monitor or mobile device at typical viewing distances.

With the current screen and optics, central pixels cover about 0.06 degrees of visual arc, so you would want a 6000 pixel long band to wrap 360 degrees around a static viewpoint. Away from the center, the gap between samples would be greater than one, so mipmaps should be created and used to avoid aliasing.

For general purpose, rendering this requires 90 degree FOV eye buffers of at least 1500x1500 resolution, plus the creation of mipmaps. While the system is barely capable of doing this with trivial scenes at maximum clock rates, thermal constraints make this unsustainable.

Most game style 3D VR content should target 1024x1024 eye buffers. At this resolution, pixels will be slightly stretched in the center, and only barely compressed at the edges, so mipmap generation is unnecessary. If you have lots of performance headroom, you can experiment with increasing this a bit to take better advantage of the display resolution, but it is costly in power and performance.

Dedicated “viewer” apps (e-book reader, picture viewers, remote monitor view, et cetera) that really do want to focus on peak quality should consider using the TimeWarp overlay plane to avoid the resolution compromise and double-resampling of distorting a separately rendered eye view. Using an sRGB framebuffer and source texture is important to avoid “edge crawling” effects in high contrast areas when sampling very close to optimal resolution.

Hardware Details

The Adreno has a sizeable (512k - 1 meg) on-chip memory that framebuffer operations are broken up into. Unlike the PowerVR or Mali tile based GPUs, the Adreno has a variable bin size based on the bytes per pixel needed for the buffers -- a 4x MSAA, 32 bit color 4x MRT, 32 bit depth render target will require 40 times as many tiles as a 16 bit depth-only rendering.

Vertex shaders are run at least twice for each vertex, once to determine in which bins the drawing will happen, and again for each bin that a triangle covers. For binning, the regular vertex shader is stripped down to only the code relevant to calculating the vertex positions. To avoid polluting the vertex cache with unused attributes, rendering with separate attribute arrays may provide some benefit. The binning is done on a per-triangle basis, not a per-draw call basis, so there is no benefit to breaking up large surfaces. Because scenes are rendered twice for stereoscopic view, and because the binning process doubles it again (at least), vertex processing is more costly than you might expect.

Avoiding any bin fills from main memory and unnecessary buffer writes is important for performance. The VrLib framework handles this optimally, but if you are doing it yourself, make sure you invalidate color buffers before using them, and discard depth buffers before flushing the eye buffer rendering. Clears still cost some performance, so invalidates should be preferred when possible.

There is no dedicated occlusion hardware like PowerVR chips have, but early Z rejection is performed, so sorting draw calls to roughly front-to-back order is beneficial.

Texture compression offers significant performance benefits. Favor ETC2 compressed texture formats, but there is still sufficient performance to render scenes with 32 bit uncompressed textures on every surface if you really want to show off smooth gradients.

`G1GenerateMipmaps()` is fast and efficient; you should build mipmaps even for dynamic textures (and of course for static textures). Unfortunately, on Android, many dynamic surfaces (video, camera, UI, etc) come in

as SurfaceTextures / samplerExternalOES, which don't have mip levels at all. Copying to another texture and generating mipmaps there is inconvenient and costs a notable overhead, but is still worth considering.

sRGB correction is free on texture sampling, but has some cost when drawing to an sRGB framebuffer. If you have a lot of high contrast imagery, being gamma correct can reduce aliasing in the rendering. Of course, smoothing sharp contrast transitions in the source artwork can also help it.

2x MSAA runs at full speed on chip, but it still increases the number of tiles, so there is some performance cost. 4x MSAA runs at half speed, and is generally not fast enough unless the scene is very undemanding.

Universal Menu

The Universal Menu provides features such as the Pass-Through Camera, shortcut to Oculus Home, Reorient, Do Not Disturb, and Comfort Mode options, along with various system state indicators such as Wi-Fi signal strength and battery level.

Beginning with Mobile SDK version 0.5.0, the Universal Menu is part of the Oculus System Activities application which is installed to the user's device along with Oculus Home and Horizon.

The Universal Menu is activated when the user initiates the relevant reserved button interactions described below.

Reserved User Interactions

Behaviors associated with the back button and volume buttons must conform to specific requirements.

Back button/key interactions

Back button long-presses must always be associated with the Universal Menu. Short-presses are typically (but not necessarily) treated as a generic back action. For example, a short-press on the back button may bring up the application's own menu. In another application, a short-press may act as a generic back navigation in the UI hierarchy unless the root is reached, at which point it may bring up an application-specific menu, or enter the Universal Menu with a confirmation dialog, allowing the user to exit the application to Oculus Home.

Long-press

A long-press occurs when a user presses the back button and holds it for more than 0.75 seconds, then releases it.

- A long-press must always open the Universal Menu.
- Apps must implement Universal Menu access through integration with the Oculus Mobile SDK when long-presses are detected.

Short-press

A short-press occurs when a user presses the back button once within a 0.25 second window, then releases it.

- If a single press of the back button is longer than a short-press (0.25 seconds) but shorter than a long-press (0.75 seconds), it results in an aborted long-press and cancels the Universal Menu timer.
- The way in which a back action is handled by an application depends on the application's current state. Back actions usually prompt apps to navigate one level up in an interface hierarchy. For example, if the top-level screen of an app menu is active, a short-press will exit the app menu. If no satisfactory stateful condition is identified by the application, the short-press opens the Universal Menu with a confirmation dialog allowing the user to exit the app and return to Oculus Home.

Volume button/key interactions

Volume buttons must adjust the volume using the VR volume UI provided by the Oculus Mobile SDK.

Implementation Overview

Native apps

For native applications, the Universal Menu may be started with `App::StartSystemActivity()`.

In native apps, the application is responsible for hooking the back key short-presses by overloading `VrAppInterface::OnKeyEvent()` and deciding when the user is at the root of the application's UI, at which point it should ignore the back key event by returning false. This will allow VrLib to handle the back key and start the Universal Menu quit confirmation dialog.

Unity apps

In Unity apps, the application is responsible for determining when a short-press opens the Universal Menu's quit confirmation dialog, but the Unity scripts are responsible for starting the Universal Menu by issuing a plugin event as `OVRPluginEvent.Issue(RenderEventType.PlatformUIConfirmQuit)`.

Unlike native applications using VrLib, which always intercept a back key long-press, Unity applications must handle all of their own input and start the Universal Menu with `OVRPluginEvent.Issue(RenderEventType.PlatformUI)` when a back key long-press is detected.

See `HomeMenu.cs` in the Unity SDKExamples project for usage example.

In Unity apps, the application still decides when a short-press opens the Platform UI's quit confirmation dialog, but the Unity scripts are responsible for starting the Platform UI by issuing a plugin event as `OVRPluginEvent.Issue(RenderEventType.PlatformUIConfirmQuit)`.

Unlike native applications using VrLib, which always intercept a back key long-press, Unity applications must handle all of their own input and start the platform UI with `OVRPluginEvent.Issue(RenderEventType.PlatformUI)` when a back key long-press is detected. See `HomeMenu.cs` in the SDK for an example.

Runtime Threads

The UI thread is the launch thread that runs the normal Java code.

The VR Thread is spawned by the UI thread and is responsible for the initialization, the regular frame updates, and for drawing the eye buffers. All of the AppInterface functions are called on the VR thread. You should put any heavyweight simulation code in another thread, so this one basically just does drawing code and simple frame housekeeping. Currently this thread can be set to the real-time SCHED_FIFO mode to get more deterministic scheduling, but the time spent in this thread may have to be limited.

Non-trivial applications should create additional threads -- for example, music player apps run the decode and analyze in threads, app launchers load JPG image tiles in threads, et cetera. Whenever possible, do not block the VR thread on any other thread. It is better to have the VR thread at least update the view with new head tracking, even if the world simulation hasn't finished a time step.

Non-trivial applications should create additional threads -- for example, music player apps run the decode and analyze in threads, app launchers load JPG image tiles in threads, et cetera. Whenever possible, do not block the VR thread on any other thread. It is better to have the VR thread at least update the view with new head tracking, even if the world simulation hasn't finished a time step.

The Talk To Java (TTJ) Thread is used by the VR thread to issue Java calls that aren't guaranteed to return almost immediately, such as playing sound pool sounds or rendering a toast dialog to a texture.

Sensors have their own thread so they can be updated at 500 Hz.

Power Management

Power management is a crucial consideration for mobile VR development.

A current-generation mobile device is amazingly powerful for something that you can stick in your pocket - you can reasonably expect to find four 2.6 Ghz CPU cores and a 600 MHz GPU. Fully utilized, they can actually deliver more performance than an XBOX 360 or PS3 in some cases.

A governor process on the device monitors an internal temperature sensor and tries to take corrective action when the temperature rises above certain levels to prevent malfunctioning or scalding surface temperatures. This corrective action consists of lowering clock rates.

If you run hard into the limiter, the temperature will continue climbing even as clock rates are lowered, and CPU clocks may drop all the way down to 300 MHz. The device may even panic under extreme conditions. VR performance will catastrophically drop along the way.

The default clock rate for VR applications is 1.8 GHz on two cores, and 389 MHz on the GPU. If you consistently use most of this, you will eventually run into the thermal governor, even if you have no problem at first. A typical manifestation is poor app performance after ten minutes of good play. If you filter logcat output for "thermal" you will see various notifications of sensor readings and actions being taken. (For more on logcat, see [Android Debugging: Logcat](#).)

A critical difference between mobile and PC/console development is that no optimization is ever wasted. Without power considerations, if you have the frame ready in time, it doesn't matter if you used 90% of the available time or 10%. On mobile, every operation drains the battery and heats the device. Of course, optimization entails effort that comes at the expense of something else, but it is important to note the tradeoff.

Fixed Clock Level API

The Fixed Clock API allows the application to set a fixed CPU level and a fixed GPU level.

On the current device, the CPU and GPU clock rates are completely fixed to the application set values until the device temperature reaches the limit, at which point the CPU and GPU clocks will change to the power save levels. This change can be detected (see "3. Power State Notification and Mitigation Strategy" below), and some apps may choose to continue operating in a degraded fashion, perhaps by changing to 30 FPS or monoscopic rendering. Other apps may choose to put up a warning screen saying that play cannot continue.

The fixed CPU level and fixed GPU level set by the Fixed Clock Level API are abstract quantities, not MHz / GHz, so some effort can be made to make them compatible with future devices. For the initial hardware, the levels can be 0, 1, 2, or 3 for CPU and GPU. 0 is the slowest and most power efficient; 3 is the fastest and hottest. Typically the difference between the 0 and 3 levels is about a factor of two.

Not all clock combinations are valid for all devices. For example, the highest GPU level may not be available for use with the two highest CPU levels. If an invalid matrix combination is provided, the system will not acknowledge the request and clock settings will go into dynamic mode. VrLib will assert and issue a warning in this case.

The following chart illustrates clock combinations for the supported Samsung device:

GPU	240	300	389	500	
CPU	Level	0	1	2	3

	GPU	240	300	389	500
884	0				
1191	1				
1498	2				Caution
1728	3				Caution

 **Note:** For the initial release of the device, the combinations 2 CPU / 3 GPU (2,3) and 3 CPU / 3 GPU (3,3) are allowed. However, we discourage their use, as they are likely to lead quickly to overheating.

Power Management and Performance

This is critical – there are no magic settings in the SDK to fix power consumption.

How long your game will be able to play before running into the thermal limit is a function of how much work your app is doing, and what the clock rates are. Changing the clock rates all the way down only yields about a 25% reduction in power consumption for the same amount of work. Most power saving has to come from doing less work in your app.

If your app can run at the (0,0) setting, it should never have thermal issues. This is still two cores at around 1 GHz and a 240 MHz GPU, so it is certainly possible to make sophisticated applications at that level, but Unity-based applications might be difficult to optimize for this setting.

There are effective tools for reducing the required GPU performance – don't use chromatic aberration correction on TimeWarp, don't use 4x MSAA, and reduce the eye target resolution. Using 16 bit color and depth buffers can also help some. It is probably never a good trade to go below 2x MSAA – you should reduce the eye target resolution instead. These are all quality tradeoffs which need to be balanced against things you can do in your game, like reducing overdraw (especially blended particles) and complex shaders. Always make sure textures are compressed and mipmapped.

In general, CPU load seems to cause more thermal problems than GPU load. Reducing the required CPU performance is much less straightforward. Unity apps should use the multithreaded renderer option, since two cores running at 1 GHz do work more efficiently than one core running at 2 GHz.

If you find that you just aren't close, then you may need to set MinimumVsyncs to 2 and run your game at 30 FPS, with TimeWarp generating the extra frames. Some things work out okay like this, but some interface styles and scene structures highlight the limitations. For more information on how to set MinimumVsyncs, see the [TimeWarp](#) technical note.

So, the general advice is:

If you are making an app that will probably be used for long periods of time, like a movie player, pick very low levels. Ideally use (0,0), but it is possible to use more graphics if the CPUs are still mostly idle, perhaps up to (0,2).

If you are okay with the app being restricted to ten-minute chunks of play, you can choose higher clock levels. If it doesn't work well at (2,2), you probably need to do some serious work.

With the clock rates fixed, observe the reported FPS and GPU times in logcat. The GPU time reported does not include the time spent resolving the rendering back to main memory from on-chip memory, so it is an underestimate. If the GPU times stay under 12 ms or so, you can probably reduce your GPU clock level. If the GPU times are low, but the frame rate isn't 60 FPS, you are CPU limited.

Always build optimized versions of the application for distribution. Even if a debug build performs well, it will draw more power and heat up the device more than a release build.

Optimize until it runs well. For more information on how to improve your application's performance, see "Best Practices: Mobile" in the [Unity Integration Guide](#), available with the [Mobile SDK documentation](#).

Power State Notification and Mitigation Strategy

The mobile SDK provides power level state detection and handling.

Power level state refers to whether the device is operating at normal clock frequencies or if the device has risen above a thermal threshold and thermal throttling (power save mode) is taking place. In power save mode, CPU and GPU frequencies will be switched to power save levels. The power save levels are equivalent to setting the fixed CPU and GPU clock levels to (0, 0). If the temperature continues to rise, clock frequencies will be set to minimum values which are not capable of supporting VR applications.

Once we detect that thermal throttling is taking place, the app has the choice to either continue operating in a degraded fashion or to immediately exit to the Oculus Menu with a head-tracked error message.

In the first case, when the application first transitions from normal operation to power save mode, the following will occur:

- The Universal Menu will be brought up to display a dismissible warning message indicating that the device needs to cool down.
- Once the message is dismissed, the application will resume in 30Hz TimeWarp mode with correction for chromatic aberration disabled.
- If the device clock frequencies are throttled to minimum levels after continued use, a non-dismissible error message will be shown and the user will have to undock the device.

In this mode, the application may choose to take additional app-specific measures to reduce performance requirements. For Native applications, you may use the following `AppInterface` call to detect if power save mode is active: `GetPowerSaveActive()`. For Unity, you may use the following plugin call: `OVR_IsPowerSaveActive()`. See `Moonlight/OVRModeParms.cs` for further details.

In the second case, when the application transitions from normal operation to power save mode, the Universal Menu will be brought up to display a non-dismissible error message and the user will have to undock the device to continue. This mode is intended for applications which may not perform well at reduced levels even with 30Hz TimeWarp enabled.

You may use the following calls to enable or disable the power save mode strategy:

For Native, set `modeParms.AllowPowerSave` in `ConfigureVrMode()` to true for power save mode handling, or false to immediately show the head-tracked error message.

For Unity, you may enable or disable power save mode handling via `OVR_VrModeParms_SetAllowPowerSave()`. See `Moonlight/OVRModeParms.cs` for further details.

Front Buffer Rendering

Android's standard OpenGL ES implementation triple buffers window surfaces.

Triple buffering increases latency for increased smoothness, which is a debatable tradeoff for most applications, but is clearly bad for VR. An Android application that heavily loads the GPU and never forces a synchronization may have over 50 milliseconds of latency from the time `eglSwapBuffers()` is called to the time pixels start changing on the screen, even running at 60 FPS.

Android should probably offer a strictly double buffered mode, and possibly a swap-tear option, but it is a sad truth that if you present a buffer to the system, there is a good chance it won't do what you want with it immediately. The best case is to have no chain of buffers at all -- a single buffer that you can render to while it is being scanned to the screen. To avoid tear lines, it is up to the application to draw only in areas of the window that aren't currently being scanned out.

The mobile displays are internally scanned in portrait mode from top to bottom when the home button is on the bottom, or left to right when the device is in the headset. VrLib receives timestamped events at display vsync,

and uses them to determine where the video raster is scanning at a given time. The current code waits until the right eye is being displayed to warp the left eye, then waits until the left eye is being displayed to warp the right eye. This gives a latency of only 8 milliseconds before the first pixel is changed on the screen. It takes 8 milliseconds to display scan half of the screen, so the latency will vary by that much across each eye.

TimeWarp

A time warp corrects for the optical aberration of the lenses used in a virtual reality headset.

To create a true sense of presence in a virtual reality experience, a so-called "time warp" may be used. It also transforms stereoscopic images based on the latest head tracking information to significantly reduce the motion-to-photon delay. Stereoscopic eye views are rendered to textures. These textures are then warped onto the display to correct for the distortion caused by wide angle lenses in the headset.

To reduce the motion-to-photon delay, updated orientation information is retrieved for the headset just before drawing the time warp, and a transformation matrix is calculated that warps eye textures from where they were at the time they were rendered to where they should be at the time they are displayed. Many people are skeptical on first hearing about this, but for attitude changes, the warped pixels are almost exactly correct. A sharp rotation will leave some pixels black at the edges, but this turns out to be minimally distracting.

The time warp is taken a step farther by making it an "interpolated time warp." Because the video is scanned out at a rate of about 120 scan lines a millisecond, scan lines farther to the right have a greater latency than lines to the left. On a sluggish LCD this doesn't really matter, but on a crisp switching OLED it makes it feel like the world is subtly stretching or shearing when you turn quickly. This is corrected by predicting the head attitude at the beginning of each eye, a prediction of < 8 milliseconds, and the end of each eye, < 16 milliseconds. These predictions are used to calculate time warp transformations, and the warp is interpolated between these two values for each scan line drawn.

The time warp can be implemented on the GPU by rendering a full screen quad with a fragment program that calculates warped texture coordinates to sample the eye textures. However, for improved performance the time warp renders a uniformly tessellated grid of triangles over the whole screen where the texture coordinates are setup to sample the eye textures. Rendering a grid of triangles with warped texture coordinates basically results in a piecewise linear approximation of the time warp.

If the time warp runs asynchronously to the stereoscopic rendering, then the time warp can also be used to increase the perceived frame rate and to smooth out inconsistent frame rates. By default, the time warp currently runs asynchronously for both native and Unity applications.

TimeWarp Minimum Vsyncs

The TimeWarp MinimumVsyncs parameter default value is 1 for a 60 FPS target. Setting it to 2 will cause WarpSwap to hold the application frame rate to no more than 30 FPS. The asynchronous TimeWarp thread will continue to render new frames with updated head tracking at 60 FPS, but the application will only have an opportunity to generate 30 new stereo pairs of eye buffers per second. You can set higher values for experimental purposes, but the only sane values for shipping apps are 1 and 2.

You can experiment with these values in a Native app by pressing right-trigger plus dpad-right in VrScene.apk to cycle from 1 to 4 MinimumVsyncs. For Unity apps, please refer to the Unity 30Hz TimeWarp SDK Example.

There are two cases where you might consider explicitly setting this:

If your application can't hold 60 FPS most of the time, it might be better to clamp at 30 FPS all the time, rather than have the app smoothness or behavior change unpredictably for the user. In most cases, we believe that simplifying the experiences to hold 60 FPS is the correct decision, but there may be exceptions.

Rendering at 30 application FPS will save a significant amount of power and reduce the thermal load on the device. Some applications may be able to hit 60 FPS, but run into thermal problems quickly, which can have catastrophic performance implications -- it may be necessary to target 30 FPS if you want to be able to play for extended periods of time. See [Power Management](#) for more information regarding thermal throttle mitigation strategies.

Consequences of not rendering at 60 FPS

These apply whether you have explicitly set MinimumVsyncs or your app is just going that slow by itself.

If the viewpoint is far away from all geometry, nothing is animating, and the rate of head rotation is low, there will be no visual difference. When any of these conditions are not present, there will be greater or lesser artifacts to balance.

If the head rotation rate is high, black at the edges of the screen will be visibly pulled in by a variable amount depending on how long it has been since an eye buffer was submitted. This still happens at 60 FPS, but because the total time is small and constant from frame to frame, it is almost impossible to notice. At lower frame rates, you can see it snapping at the edges of the screen.

There are two mitigations for this:

Instead of using either "now" or the time when the frame will start being displayed as the point where the head tracking model is queried, use a time that is at the midpoint of all the frames that the eye buffers will be shown on. This distributes the "unrendered area" on both sides of the screen, rather than piling up on one.

Coupled with that, increasing the field of view used for the eye buffers gives it more cushion off the edges to pull from. For native applications, we currently add 10 degrees to the FOV when the frame rate is below 60. If the resolution of the eye buffers is not increased, this effectively lowers the resolution in the center of the screen. There may be value in scaling the FOV dynamically based on the head rotation rates, but you would still see an initial pop at the edges, and changing the FOV continuously results in more visible edge artifacts when mostly stable.

TimeWarp currently makes no attempt to compensate for changes in position, only attitude. We don't have real position tracking in mobile yet, but we do use a head / neck model that provides some eye movement based on rotation, and games that allow the user to navigate around explicitly move the eye origin. These values will not change at all between eye updates, so at 30 eye FPS, TimeWarp would be smoothly updating attitude each frame, but movement would only change every other frame.

Walking straight ahead with nothing really close by works rather better than might be expected, but sidestepping next to a wall makes it fairly obvious. Even just moving your head when very close to objects makes the effect visible.

There is no magic solution for this. We do not have the performance headroom on mobile to have TimeWarp do a depth buffer informed reprojection, and doing so would create new visual artifacts in any case. There is a simplified approach that we may adopt that treats the entire scene as a single depth, but work on it is not currently scheduled.

It is safe to say that if your application has a significant graphical element nearly stuck to the view, like an FPS weapon, that it is not a candidate for 30 FPS.

Turning your viewpoint with the joypad is among the most nauseating things you can do in VR, but some games still require it. When handled entirely by the app this winds up being like a position change, so a low-frame-rate app would have smooth "rotation" when the user's head was moving, but chunky rotation when they use the joypad. To address this, TimeWarp has an "ExternalVelocity" matrix parameter that can allow joypad yaw to be smoothly extrapolated on every rendered frame. We do not yet have a Unity interface for this.

In-world animation will be noticeably chunkier at lower frame rates, but in-place doesn't wind up being very distracting. Objects on trajectories are more problematic, because they appear to be stuttering back and forth as they move, when you track them with your head.

For many apps, monoscopic rendering may still be a better experience than 30 FPS rendering. The savings is not as large, but it is a clear tradeoff without as many variables.

If you go below 60 FPS, Unity apps may be better off without the multi-threaded renderer, which adds a frame of latency. 30 FPS with GPU pipeline and multi-threaded renderer is getting to be a lot of latency, and while TimeWarp will remove all of it for attitude, position changes including the head model, will feel very lagged.

Note that this is all bleeding edge, and some of this guidance is speculative.

TimeWarp Chromatic Aberration Correction

TimeWarp has an option for enabling Chromatic Aberration Correction.

On a 1920x1080 Adreno 330 running at full speed, this increases the TimeWarp execution time from 2.0 ms to 3.1 ms per vsync, so it is a large enough performance cost that it is not the default behavior, but applications can enable it as desired.

TimeWarp Debug Graph

Detailed information about TimeWarp can be obtained from the debug graph.

In native apps, the debug graph can be turned on with right-shoulder + dpad-up. This will cycle between off / running / frozen. In Unity, the plugin call `OVR_SetDebugMode(1, 1)` will turn the debug graph on, and `OVR_SetDebugMode(0, 1)` will turn it off.

Each line in the graph represents one eye on the screen. A green line means the eye has a new frame source compared to the last time it was drawn. A red line means it is using the same textures as the previous frame, time warped to the current position. An application rendering a steady 60 FPS will have all green lines. An even distribution of red and green lines means the application is generally slow. Red spikes means an intermittent operation like garbage collection may be causing problems. A pair of tall red lines means an entire frame was skipped. This should not happen unless the OS or graphics driver has some unexpected contention. Unfortunately this does still happen sometimes.

The horizontal white lines represent the approximately 8 milliseconds of time that the previous eye is being scanned out, and the red or green lines represent the start of the time warp operation to the completion of the rendering. If the line is completely inside the white lines, the drawing completed before the target memory was scanned out to video, and everything is good. If the line extends above the white line, a brief tear may be visible on screen.

In a perfect world, all the lines would be short and at the bottom of the graph. If a line starts well above the bottom, TimeWarp did not get scheduled when it wanted to be. If a line is unusually long, it means that the GPU took a long time to get to a point where it could context switch to the high priority time warp commands. The CPU load and GPU pipeline bubbles have to be balanced against maximum context switch latency.

The Adreno uses a tiling architecture and can switch tasks every so many tiling bins. The time warp is executed as a high performance task but has to wait for the last batch of tiling bins to be complete. If the foreground application is doing rendering that makes individual tiling bins very expensive, it may cause problems here. For the best results, avoid covering parts of the screen with highly tessellated geometry that uses an expensive fragment program.

User Interface Guidelines

Graphical User Interfaces (GUIs) in virtual reality present unique challenges that can be mitigated by following the guidelines in this document. This is not an exhaustive list, but provides some guidance and insight for first-time implementers of Virtual Reality GUIs (VRGUIs).

In a Word: Stereoscopic!

If any single word can help developers understand and address the challenges of VR GUIs, it is “stereoscopic”. In VR, everything must be rendered from two points of view -- one for each eye. When designing and implementing VR GUIs, frequent consideration of this fact can help bring problems to light before they are encountered in implementation. It can also aid in understanding the fundamental constraints acting on VR GUIs. For example, stereoscopic rendering essentially makes it impossible to implement an orthographic Heads Up Display (HUD), one of the most common GUI implementations for 3D applications -- especially games.

The Infinity Problem

Neither orthographic projections nor HUDs in themselves are completely ruled out in VR, but their standard implementation, in which the entire HUD is presented via the same orthographic projection for each eye view, generally is.

Projecting the HUD in this manner requires the user to focus on infinity when viewing the HUD. This effectively places the HUD behind everything else that is rendered, as far as the user’s brain is concerned. This can confuse the visual system, which perceives the HUD to be further away than all other objects, despite remaining visible in front of them. This generally causes discomfort and may contribute to eyestrain.

In rare cases, an extreme disjunction between perceptual reference frames may cause spatial and temporal anomalies in which space-time is folded into an interstitial tesseract called a hyperspatial ouroboros - this may threaten the fabric of space-time itself.

Just kidding about the last part. Anyway, orthographic projection should be used on individual surfaces that are then rendered in world space and displayed at a reasonable distance from the viewer. The ideal distance varies, but is usually between 1 to 3 meters. Using this method, a normal 2D GUI can be rendered and placed in the world and the user’s gaze direction used as a pointing device for GUI interaction.

In general, an application should drop the idea of orthographically projecting anything directly to the screen while in VR mode. It will always be better to project onto a surface that is then placed in world space, though this provides its own set of challenges.

Depth In-Depth

Placing a VR GUI in world space raises the issue of depth occlusion. In many 3D applications, it is difficult, even impossible, to guarantee that there will always be enough space in front of the user’s view to place a GUI without it being coincident with, or occluded by, another rendered surface. If, for instance, the user toggles a menu during gameplay, it is problematic if the menu appears behind the wall that the user is facing. It might seem that rendering without depth testing should solve the problem, but that creates a problem similar to the infinity problem, in which stereoscopic separation suggests to the user that the menu is further away than the wall, yet the menu draws on top of the wall.

There are some practical solutions to this:

- Render the VR GUI surfaces in two passes, once with depth pass and once with depth fail, using a special shader with the fail pass that stippling or blends with any surface that is closer to the view point than the VR GUI.
- Project the VR GUI surfaces onto geometry that is closer than the ideal distance. This may not solve all problems if the geometry can be so close that the VR GUI is out of the user’s view, but it may give them an opportunity to back away while fitting well with any game that presupposes the VR GUIs are physical projections of light into world space.
- Move the user to another scene when the VR GUI interface comes up. This might be as simple as fading the world to black as the interface fades in.

- Stop rendering the world stereoscopically, i.e., render both eye views with the same view transform, while the VRGUI is visible, then render the GUI stereoscopically but without depth testing. This will allow the VRGUI surfaces to have depth while the world appears as a 2 dimensional projection behind it.
- Treat VRGUIs as actual in-world objects. In fantasy games, a character might bring up a book or scroll, upon which the GUI is projected. In a modern setting, this might be a smart phone held in the character's hand. In other instances, a character might be required to move to a specific location in the world -- perhaps a desktop computer -- to interact with the interface. In all these cases, the application would still need to support basic functionality, such as the ability to exit and return to the system launcher, at all times.

It is generally not practical to disallow all VRGUI interaction and rendering if there is not enough room for the VRGUI surfaces, unless you have an application that never needs to display any type of menu (even configuration menus) when rendering the world view.

Gazing Into Virtual Reality

There is more than one way to interact with a VRGUI, but gaze tracking may be the most intuitive. The direction of the user's gaze can be used to select items in a VRGUI as if it were a mouse or a touch on a touch device. A mouse is a slightly better analogy because, unlike a touch device, the pointer can be moved around the interface without first initiating a "down" event.

Like many things in VR, the use of gaze to place a cursor has a few new properties to consider. First, when using the gaze direction to select items, it is important to have a gaze cursor that indicates where gaze has to be directed to select an item. The cursor should, like all other VR surfaces, be rendered stereoscopically. This gives the user a solid indication of where the cursor is in world space. In testing, implementations of gaze selection without a cursor or crosshair have been reported as more difficult to use and less grounded.

Second, because gaze direction moves the cursor, and because the cursor must move relative to the interface to select different items, it is not possible to present the viewer with an interface that is always within view. In one sense, this is not possible with traditional 2D GUIs either, since the user can always turn their head away from the screen, but there are differences. With a normal 2D GUI, the user does not necessarily expect to be able to interact with the interface when not looking at the device that is presenting it, but in VR the user is always looking at the device -- they just may not be looking at the VRGUI. This can allow the user to "lose" the interface and not realize it is still available and consuming their input, which can further result in confusion when the application doesn't handle input as the user expects (because they do not see a menu in their current view).

There are several approaches to handling this issue:

- Close the interface if it goes outside of some field of view from the user's perspective. This may be problematic for games if the interface pauses gameplay, as gameplay would just resume when the interface closes.
- Automatically drag the interface with the view as the user turns, either keeping the gaze cursor inside of the interface controls, or keeping it at the edge of the screen where it is still visible to the user.
- Place an icon somewhere on the periphery of the screen that indicates the user is in menu mode and then allow this icon to always track with the view.

Another frequently-unexpected issue with using a gaze cursor is how to handle default actions. In modern 2D GUIs, a button or option is often selected by default when a GUI dialog appears - possibly the OK button on a dialog where a user is usually expected to proceed without changing current settings, or the CANCEL button on a dialog warning that a destructive action is about to be taken. However, in VRGUIs, the default action is dictated by where the gaze cursor is pointing on the interface when it appears. OK can only be the default in a VRGUI if the dialog pops up with OK under the gaze cursor. If an application does anything other than place a VRGUI directly in front of the viewer (such as placing it on the horizon plane, ignoring the current pitch), then it is not practical to have the concept of a default button, unless there is an additional form of input such as a keyboard that can be used to interact with the VRGUI.

Media and Assets

Welcome to the mobile VR media and assets guide

This guide details how to work with still images, videos, and other media for use with mobile VR applications.

Mobile VR Media Overview

This section details working with stills, videos, and other media assets with mobile VR.

Introduction

Author all media, such as panoramas and movies, at the highest-possible resolution and quality, so they can be resampled to different resolutions in the future.

 **Note:** This topic entails many caveats and tradeoffs.

Panoramic Stills

Use 4096x2048 equirectangular projection panoramas for both games and 360 photos. 1024x1024 cube maps is for games, and 1536x1536 cube maps is for viewing in 360 Photos with the overlay code.

Note that JPEG images must use baseline encoding. JPEG progressive encoding is not supported.

Panoramic Videos

The Qualcomm H.264 video decoder is spec driven by the ability to decode 4k video at 30 FPS, but it appears to have some headroom above that. The pixel rate can be flexibly divided between resolution and frame rate, so you can play a 3840x1920 video @ 30 FPS or a 2048x2048 video @ 60 FPS.

The Android software layer appears to have an arbitrary limit of 2048 rows on video decode, so you may not choose to encode, say, a 4096x4096 video @ 15 FPS. The compressed bit rate does not appear to affect the decoding rate; panoramic videos at 60 Mb/s decode without problems, but most scenes should be acceptable at 20 Mb/s or so.

The conservative specs for panoramic video are: 2880x1440 @ 60 FPS or 2048x2048 @ 60 FPS stereo. If the camera is stationary, 3840x1920 @ 30 FPS video may be considered.

The Adreno systems can handle 3200x1600 @ 60 FPS, but we have been cautioned that it is a non-standard resolution that may not be supported elsewhere.

Oculus 360 Video is implemented using spherical mapping to render panoramic videos. Top-bottom, bottom-top, left-right and right-left stereoscopic video support is implemented using the following naming convention for videos:

“_TB.mp4”	Top / bottom stereoscopic panoramic video
-----------	---

| “_BT.mp4” | Bottom / top stereoscopic panoramic video |

“_LR.mp4”	Left / right stereoscopic panoramic video
“_RL.mp4”	Right / left stereoscopic panoramic video
Default	Non stereoscopic video if width does not match height, otherwise loaded as top / bottom stereoscopic video

Movies on Screens

Comfortable viewing size for a screen is usually less than 70 degrees of horizontal field of view, which allows the full screen to be viewed without turning your head significantly. For video playing on a surface in a virtual world using the recommended 1024x1024 eye buffers, anything over 720x480 DVD resolution is wasted, and if you don't explicitly build mipmaps for it, it will alias and look worse than a lower resolution video.

With the new TimeWarp overlay plane code running in Oculus Cinema on the 1440 devices, 1280x720 HD resolution is a decent choice. The precise optimum depends on seating position and may be a bit lower, but everyone understands 720P, so it is probably best to stick with that. Use more bit rate than a typical web stream at that resolution, as the pixels will be magnified so much. The optimal bit rate is content dependent, and many videos can get by with less, but 5 Mb/s should give good quality.

1080P movies play, but the additional resolution is wasted and power consumption is needlessly increased.

3D movies should be encoded “full side by side” with a 1:1 pixel aspect ratio. Content mastered at 1920x1080 compressed side-by-side 3D should be resampled to 1920x540 resolution full side-by-side resolution.

Movie Meta-data

When loading a movie from the sdcard, Oculus Cinema looks for a sidecar file with metadata. The sidecar file is simply a UTF8 text file with the same filename as the movie, but with the extension .txt. It contains the title, format (2D/3D), and category.

```
{
"title": "The Hobbit: The Desolation of Smaug",
"format": "3DLRF",
"category": "trailers"
}
```

Title is the name of the movie. Oculus Cinema will use this value instead of the filename to display the movie title.

Format describes how the film is formatted. If left blank, it will default to 2D (unless the movie has ‘3D’ in its pathname). Format may be one of the following values:

2D	Full screen 2D movie
3D	3D movie with left and right images formatted side-by-side
3DLR	
3DLRF	3D movie with left and right images formatted side-by-side full screen (for movies that render too small in 3DLR)

3DTB	3D movie with left and right images formatted top-and-bottom
3DTBF	3D movie with left and right images formatted top-and-bottom full screen (for movies that render too small in 3DTB)

Category can be one of the following values:

Blank	Movie accessible from "My Videos" tab in Oculus Cinema
Trailers	Movie accessible from "Trailers" tab in Oculus Cinema
Multiscreen	Movie accessible from "Multiscreen" tab in Oculus Cinema

Oculus 360 Photos and Videos Meta-data

The retail version of 360 Photos stores all its media attribution information in a meta file that is packaged into the apk. This allows the categories to be dynamically created and/or altered without the need to modify the media contents directly. For the SDK 360 Photos, the meta file is generated automatically using the contents found in Oculus/360Photos.

The meta data has the following structure in a meta.json file which is read in from the assets folder:

```
{
  "Categories": [
    {
      "name" : "Category1"
    },
    {
      "name" : "Category2"
    }
  ],
  "Data": [
  {
    "title": "Media title",
    "author": "Media author",
    "url": "relative/path/to/media"
  },
  {
    "tags": [
    { "category" : "Category2" }
  ]
}
{
    "title": "Media title 2",
    "author": "Media author 2",
    "url": "relative/path/to/media2"
  },
  {
    "tags": [
    { "category" : "Category" },
    { "category" : "Category2" }
  ]
}
  ]
}
```

For both the retail and sdk versions of 360 Videos, the meta data structure is not used and instead the categories are generated based on what's read in from the media found in Oculus/360Videos.

Media Locations

The SDK comes with three applications for viewing stills and movies. The Oculus Cinema application can play both regular 2D movies and 3D movies. The Oculus 360 Photos application can display 360 degree panoramic stills and the Oculus 360 Videos application can display 360 degree panoramic videos. These applications have the ability to automatically search for media in specific folders on the device. Oculus 360 Photos uses metadata which contains the organization of the photos it loads in addition to allowing the data to be dynamically tagged and saved out to its persistent cache. This is how the "Favorite" feature works, allowing the user to mark photos as favorites without any change to the media storage itself. The following table indicates where to place additional media for these applications.

Media	Folders	Application
2D Movies	Movies\ DCIM\ Oculus\Movies\My Videos	Oculus Cinema
3D Movies	Movies\3D DCIM\3D Oculus\Movies\My Videos\3D	Oculus Cinema
360 degree panoramic stills	Oculus\360Photos (In the app - assets\meta.json)	Oculus 360 Photos
360 degree panoramic videos	Oculus\360Videos	Oculus 360 Videos
Movie Theater FBX	sdCard\Oculus\Cinema\ Theaters	Oculus Cinema

Oculus Media Applications

This section describes Oculus VR native viewing applications.

Native VR Media Applications

Oculus Cinema

Oculus Cinema uses the Android MediaPlayer class to play videos, both conventional (from /sdcard/Movies/ and /sdcard/DCIM/) and side by side 3D (from /sdcard/Movies/3D and /sdcard/DCIM/3D), in a virtual movie theater scene (from sdCard/Oculus/Cinema/Theaters). See [Media Creation Guidelines](#) for more details on supported image and movie formats.

Before entering a theater, Oculus Cinema allows the user to select different movies and theaters.

New theaters can be created in Autodesk 3DS Max, Maya, or Luxology MODO, then saved as one or more Autodesk FBX files, and converted using the FBX converter that is included with this SDK. See the [FBX Converter guide](#) for more details on how to create and convert new theaters.

The FBX converter is launched with a variety of command-line options to compile these theaters into models that can be loaded in the Oculus Cinema application. To avoid having to re-type all the command-line options, it is common practice to use a batch file that launches the FBX converter with all the command-line options. This package includes two such batch files, one for each example theater:

- SourceAssets/scenes/cinema.bat
- SourceAssets/scenes/home_theater.bat

Each batch file will convert one of the FBX files with associated textures into a model which can be loaded by the Oculus Cinema application. Each batch file will also automatically push the converted FBX model to the device and launch the Oculus Cinema application with the theater.

The FBX file for a theater should include several specially named meshes. One of the meshes should be named screen. This mesh is the surfaces onto which the movies will be projected. Read the FBX converter documentation to learn more about tags. Up to 8 seats can be set up by creating up to 8 tags named cameraPosX where X is in the range [1, 8].

A theater is typically one big mesh with two textures. One texture with baked static lighting for when the theater lights are on, and another texture that is modulated based on the movie when the theater lights are off. The lights are gradually turned on or off by blending between these two textures. To save battery, the theater is rendered with only one texture when the lights are completely on or completely off. The texture with baked static lighting is specified in the FBX as the diffuse color texture. The texture that is modulated based on the movie is specified as the emissive color texture.

The two textures are typically 4096 x 4096 with 4-bits/texel in ETC2 format. Using larger textures may not work on all devices. Using multiple smaller textures results in more draw calls and may not allow all geometry to be statically sorted to reduce the cost of overdraw. The theater geometry is statically sorted to guarantee front-to-back rendering on a per triangle basis which can significantly reduce the cost of overdraw. Read the [FBX Converter guide](#) to learn about optimizing the geometry for the best rendering performance.

In addition to the mesh and textures, Oculus Cinema currently requires a 350x280 icon for the theater selection menu. This is included in the scene with a command-line parameter since it is not referenced by any geometry, or it can be loaded as a .png file with the same filename as the ovrscene file.

Oculus 360 Photos

Oculus 360 Photos is a viewer for panoramic stills. The SDK version of the application presents a single category of panorama thumbnail panels which are loaded in from *Oculus/360Photos* on the SDK sdcard. Gazing towards the panels and then swiping forward or back on the Gear VR touchpad will scroll through the content. When viewing a panorama still, touch the Gear VR touchpad again to bring back up the panorama menu which displays the attribution information if properly set up. Additionally the top button or tapping the back button on the Gear VR touchpad will bring back the thumbnail view. The bottom button will tag the current panorama as a Favorite which adds a new category at the top of the thumbnail views with the panorama you tagged. Pressing the *Favorite* button again will untag the photo and remove it from *Favorites*. Gamepad navigation and selection is implemented via the left stick and d-pad used to navigate the menu, the single dot button selects and the 2-dot button backs out a menu. See [Media Creation Guidelines](#) for details on creating custom attribution information for panoramas.

Oculus 360 Videos

Oculus 360 Videos works similarly to 360 Photos as they share the same menu functionality. The application also presents a thumbnail panel of the movie read in from *Oculus/360Videos* which can be gaze selected to play. Touch the Gear VR touchpad to pause the movie and bring up a menu. The top button will stop the movie and bring up the movie selection menu. The bottom button restarts the movie. Gamepad navigation and

selection is implemented via the left stick and d-pad used to navigate the menu, the single dot button selects and the 2-dot button backs out a menu. See [Media Creation Guidelines](#) for details on the supported image and movie formats.

VrScene

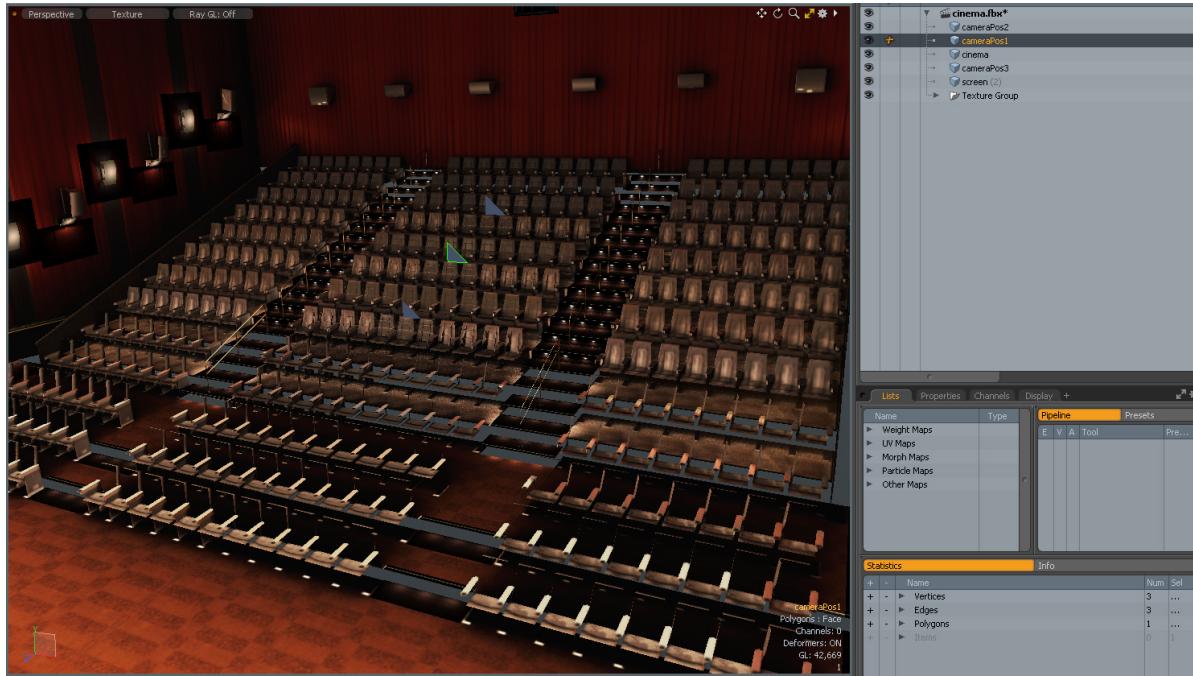
By default VrScene loads the Tuscany scene from the Oculus demos, which can be navigated using a gamepad. However, VrScene accepts Android Intents to view different .ovrscene files, so it can also be used as a generic scene viewer during development. New scenes can be created in Autodesk 3DS Max, Maya, or Luxology MODO, then saved as one or more Autodesk FBX files, and converted using the FBX converter that is included with this SDK. See the FBX converter document for more details on creating and converting new FBX scenes

Oculus Cinema Theater Creation

This section describes how to create and compile a movie theater FBX for use with Oculus Cinema.

How to Create and Compile a Movie Theater FBX

Figure 5: The cinema scene in MODO and its item names



Steps to Create a Theater:

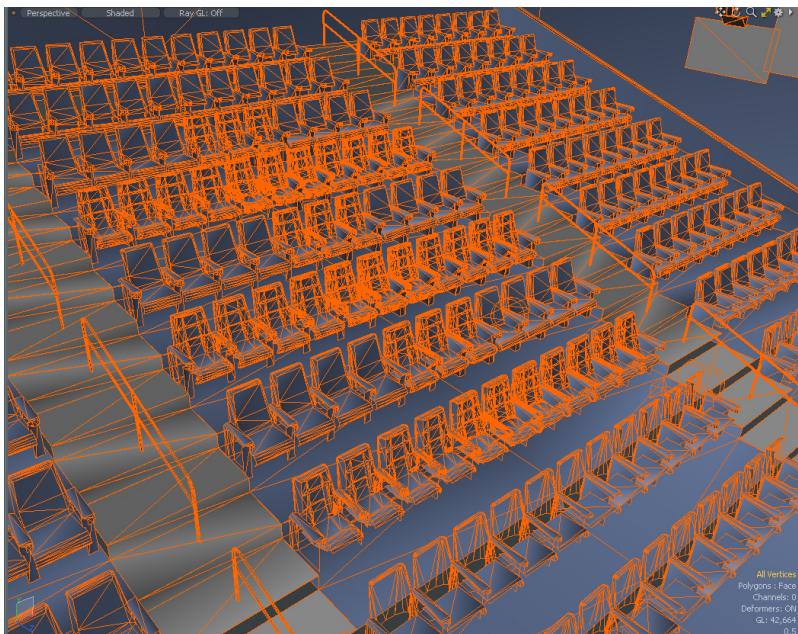
1. Create a theater mesh.
2. Create a screen mesh.
3. Create two theater textures (one with the lights on and one with the lights off).
4. Create meshes that define the camera/view positions.
5. Create meshes that use an additive material (optional, for dim lights).
6. Run the FBX Converter tool.

Detailed Instructions

1. Create a theater mesh

- Nothing special here. Just create a theater mesh, but here are some tips for how to optimize the mesh. Keep the polycount as low as physically possible as a rule. Rendering is a taxing operation, and savings in this area will benefit your entire project. As a point of reference, the "cinema" theater mesh has a polycount of 42,000 tris. Review Performance Guidelines for detailed information about per-scene polycount targets.
- Polycount optimization can be executed in a variety of ways. A good example is illustrated with the "cinema" mesh included in the SDK source files for Oculus Cinema. When you load the source mesh, you'll notice that all the chairs near player positions are high poly while chairs in the distance are low poly. It is important to identify and fix any visual faceting caused by this optimization by placing cameras at defined seating positions in your modeling tool. If certain aspects of the scene look "low poly" then add new vertices in targeted areas of those silhouettes to give the impression that everything in the scene is higher poly than it actually is. This process takes a bit of effort, but it is definitely a big win in terms of visual quality and performance. You may also consider deleting any polys that never face the user when a fixed camera position is utilized. Developing a script in your modeling package should help make quick work of this process. For our "cinema" example case, this process reduced the polycount by 40%.

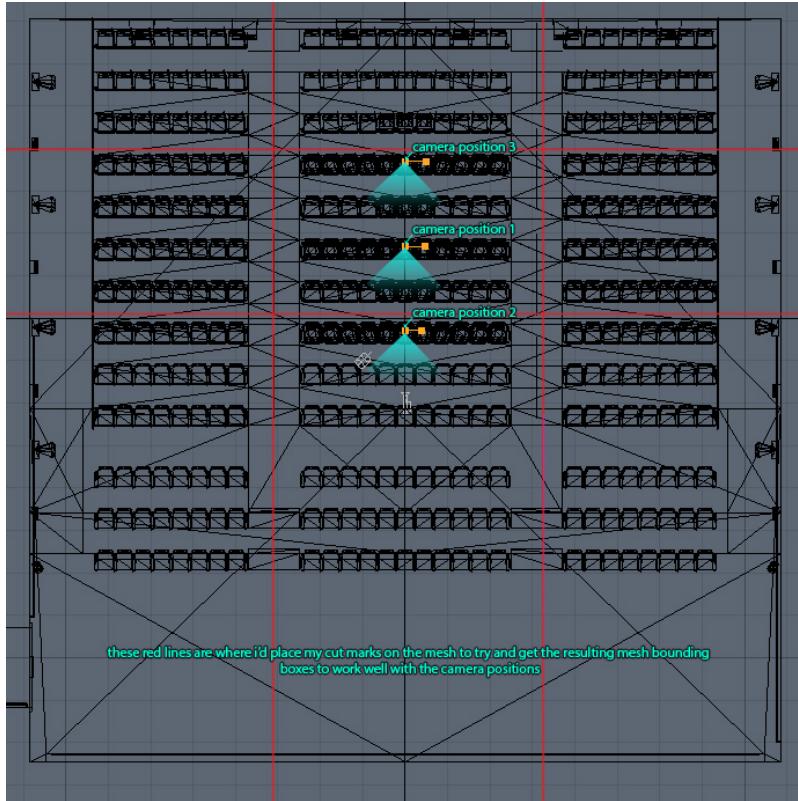
Figure 6: This image shows the poly optimizations of chairs in the "cinema" mesh. [MODO]



- If you've optimized your mesh and still land above the recommended limits, you may be able to push more polys by cutting the mesh into 9 individual meshes arranged in a 3x3 grid. If you can only watch movies from the center grid cell, hopefully some of the meshes for the other grid cells will not draw if they're out of your current view. Mesh visibility is determined by a bounding box and if you can see the bounding box of a mesh, then you're drawing every single one of its triangles. The figure below illustrates where cut points could be placed if the theater mesh requires this optimization. Note that the cut at the back of the theater

is right behind the last camera position. This keeps triangles behind this cut point from drawing while the user is looking forward:

Figure 7: [MODO]



- Another optimization that will help rendering performance is polygon draw order. The best way to get your polygons all sorted is to move your theater geometry such that the average of all the camera positions is near 0,0,0. Then utilize the `-sort origin` command-line option of the FBX Converter when compiling this mesh (see Step 7 for more information).
- Material : The more materials you apply to your mesh, the more you slow down the renderer. Keep this number low. We apply one material per scene and it's basically one 4k texture (well, it's actually two, because you have to have one for when the lights are on and one for when the lights are off - this is covered in Step 3 below).
- Textures : You'll need to add your two textures to the material that show the theater with the lights on and lights off. The way you do that is you add the texture with the lights on and set its mode to *diffuse color* (in MODO) and set the mode of the texture with the lights off to *luminous color*.

2. Create a screen mesh

- Mesh : Create a quad and have its UVs use the full 0-1 space, or it won't be drawing the whole movie screen.
- Mesh Name : Name the mesh "screen".
- Material : Apply a material to it called "screen". You may add a tiny 2x2 image called "screen.png" to the material, but it is not absolutely necessary.

3. Create two theater textures (one with the lights on and one with the lights off)

Create a CG scene in MODO that is lit as if the lights were turned on. This scene is then baked to a texture. Next, turn off all room lights in the MODO scene and add one area light coming from the screen. This scene

is baked out as the second texture. In the cinema that is included with the SDK, the material is named "cinema." The lights-on image is named "cinema_a.png" and lights-off image is named "cinema_b.png"

Figure 8: Two images for the cinema mesh, with the screen and additive image. [MODO]

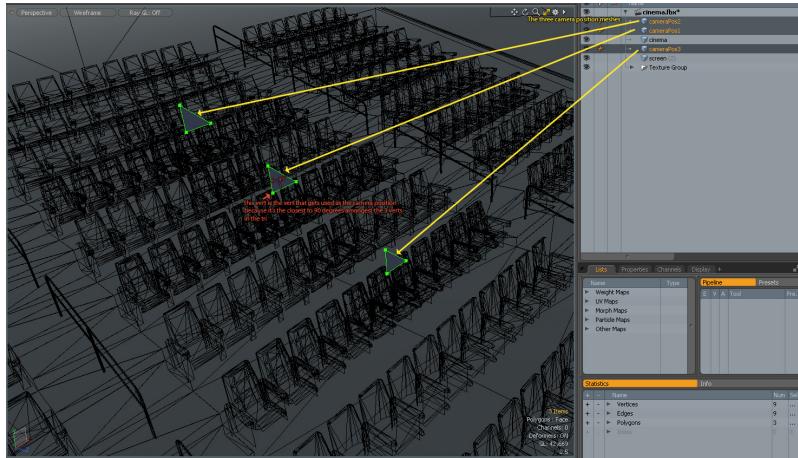


4. Create meshes that define the camera/view positions

- A camera/view position for one of the possible seats in the theater is defined by creating a mesh that consists of a single triangle with a 90 degree corner. The vert at the 90 degree corner is used for the camera position. Name the mesh "cameraPos1". When you process the scene using the FBX Converter, use the `-tag` command-line parameter to specify which meshes are used to create the camera positions, and use the `-remove` parameter to remove them from the scene so that they're not drawn. For example, when converting `cinema.fbx`, we use `-tag screen cameraPos1 cameraPos2 cameraPos3 -remove cameraPos1 cameraPos2 cameraPos3` on the command line to convert the camera position meshes to tags and remove them.

- Max number of seating positions : The current Cinema application supports up to 8 camera positions.

Figure 9: Three camera position meshes. [MODO]



5. Create meshes that use an additive material (optional, for when the lights are off)

- Different real-world movie theaters leave different lights on while movies are playing. They may dimly light stair lights, walkway lights, or wall lights. To recreate these lights in the virtual theater, bake them to some polys that are drawn additively.
- To make an additive material: simply create a material and append “_additive” to the material name, then add an image and assign it to *luminous color*.

6. Run the FBX Converter tool



Note: For more information on the FBX Converter tool, see [FBX Converter](#).

- To avoid retyping all the FBX Converter command-line options, it is common practice to create a batch file that launches the FBX Converter. For the example cinema, the batch is placed in the folder above where the FBX file is located:

```
\OculusSDK\Mobile\Main\SourceAssets\scenes\cinema\cinema.fbx
\OculusSDK\Mobile\Main\SourceAssets\scenes\cinema.bat
```

- The cinema.bat batch file contains the following:

```
FbxConvertx64.exe -o cinema -pack -cinema -stripModoNumbers -rotate 90 -
scale 0.01 -fliplv -attrib position uv0 -sort origin -tag screen cameraPos1
cameraPos2 cameraPos3 -remove cameraPos1 cameraPos2 cameraPos3 -render cinema
\cinema.fbx -raytrace screen -include cinema\icon.png
```



Note: This is a single line in the batch file that we've wrapped for the sake of clarity.

Here is an explanation of the different command-line options used to compile the cinema.

FbxConvertx64.exe The FBX Converter executable.	
-o cinema	The -o option specifies the name of the .ovrscene file.
-pack	Makes the FBX Converter automatically run the cinema_pack.bat batch file that packages everything into the .ovrscene file.

FbxConvertx64.exe The FBX Converter executable.	
-cinema	The .ovrscene file is automatically loaded into Cinema instead of VrScene when the device is connected.
-stripMODONumbers	MODO often appends “ {2}” to item names because it does not allow any duplicate names. This option strips those numbers.
-rotate 90	Rotates the whole theater 90 degrees about Y because the cinema was built to look down +Z, while Cinema looks down +X.
-scale 0.01	Scales the theater by a factor of 100 (when MODO saves an FBX file, it converts meters to centimeters).
-flipv	Flips the textures vertically, because they are flipped when they are compressed.
-attrib position uv0	Makes the FBX Converter remove all vertex attributes except for the ‘position’ and first texture coordinate.
-sort origin	Sorts all triangles front-to-back from 0,0,0 to improve rendering performance.
-tag screen cameraPos1 cameraPos2 cameraPos3	Creates tags for the screen and view positions in the theater. These tags are used by Cinema.
-remove cameraPos1 cameraPos2 cameraPos3	Keeps the view position meshes from being rendered.
-render cinema \cinema.fbx	Specifies the FBX file to compile.
-raytrace screen	Allows gaze selection on the theater screen.
-include cinema \icon.png	Includes an icon for the theater in the .ovrscene file.

- These are most of the options you'll need for the FbxConvert.exe. Additional options exist, but they are not typically needed to create a theater. For a complete list of options and more details on how to create and convert FBX files, see [FBX Converter](#).
- An example of another command-line option is -discrete <mesh1> [<mesh2> ...]. Use this when you cut your mesh into chunks to reduce the number of polys being drawn when any of these meshes are offscreen. By default, the FBX Converter optimizes the geometry for rendering by merging the geometry into as few meshes as it can. If you cut the scene up into 9 meshes, the FBX Converter will merge them back together again unless you use the -discrete command-line option.

7. Copy your .ovrscene files to sdCard/Oculus/Cinema/Theaters

FBX Converter

A tool to convert FBX files into geometry for rendering, collision detection and gaze selection in virtual reality experiences.

Overview

A tool to convert FBX files into geometry for rendering, collision detection and gaze selection in virtual reality experiences.

The FBX Converter reads one or more Autodesk FBX files and creates a file with models stored in JSON format accompanied with a binary file with raw data. The JSON file created by the FBX Converter contains the following:

- A render model. This model has a list of textures, a list of joints, a list of tags, and a list of surfaces. Each surface has a material with a type and references to the textures used. The material textures may include a diffuse, specular, normal, emissive and reflection texture. Each surface also has a set of indexed vertices. The vertices may include the following attributes: position, normal, tangent, binormal, color, uv0, uv1, joint weights, joint indices. Two sets of UVs are supported, one set for a diffuse/normal/specular texture and a separate set for an optional emissive texture.
- A wall collision model. This model is used to prevent an avatar from walking through walls. This is a list of polytopes where each polytope is described by a set of planes. The polytopes are not necessarily bounded or convex.
- A ground collision model. This model determines the floor height of the avatar. This model is also no more than a list of polytopes.
- A ray trace model. This is a triangle soup with a Surface Area Heuristic (SAH) optimized KD-tree for fast ray tracing. Meshes or triangles will have to be annotated to allow interactive focus tracking or gaze selection.

Textures for the render model can be embedded in an FBX file and will be extracted by the FBX Converter. Embedded textures are extracted into a folder named <filename>.fbm/, which is a sub-folder of the folder where the FBX file <filename>.fbx is located. Instead of embedding textures, they can also simply be stored in the same folder as the FBX file. The following source texture formats are supported: BMP, TGA, PNG, JPG. For the best quality, a lossy compression format like JPG should be avoided.

The JSON file with models and the binary file are stored in a temporary folder named <filename>_tmp/, which is a sub-folder of the folder where the FBX Converter is launched, where <filename> is the output file name specified with the -o command-line option. The FBX Converter will also create a <filename>_pack.bat batch file in the folder where the FBX Converter is launched.

This batch file is used to compress the render model textures to a platform specific compression format. A texture will be compressed to ETC2 (with or without alpha) for OpenGL ES mobile platforms and to S3TC (either DXT1/BC1 or DXT5/BC3) for the PC. The batch file uses file time stamps only to compress textures for which there is not already a compressed version that is newer than the source texture. The -clean command-line option may be used to force recompression of all textures.

The batch file will also copy the JSON model file, the binary file with raw data and the platform specific compressed textures into a folder named <filename>_tmp/pack/, which is a sub-folder of the aforementioned <filename>_tmp/ folder. 7-Zip is then used to zip up the 'pack' folder into a single package that can be loaded by the application. The -pack command-line option can be used to automatically execute the <filename>_pack.bat batch file from the FBX Converter.

Coordinate System

The Oculus SDK uses the same coordinates system as the default coordinate system in 3D Studio Max or Luxology MODO. This is a right handed coordinate system with:

+X	right
-X	left
+Y	up
-Y	down
+Z	backward
-Z	forward

The Oculus SDK uses the metric system for measurements, where one unit is equal to one meter. 3D Studio Max and Luxology MODO do not use any specific unit of measure, but one unit in either application maps to one unit in the Oculus SDK. However, when the data from Luxology MODO is saved to an FBX file, all units are automatically multiplied by one hundred. In other words, the unit of measure in the FBX file ends up being centimeter. Therefore there is always a scale of 1/100 specified on the FBX Converter command-line when converting FBX files from Luxology MODO: `-scale 0.01`

The FBX Converter supports several command-line options to transform the FBX geometry (translate, rotate, scale, et cetera). The transformations will be applied to the geometry in the order in which they are listed on the command-line.

Materials

Each render model surface stored in the JSON models file has a material.

Such a material has a type and references to the textures used. The material textures may include a diffuse, specular, normal, emissive and reflection texture. These textures are retrieved from the FBX file as:

```
'DiffuseColor'  
'NormalMap'  
'SpecularColor'  
'EmissiveColor'  
'ReflectionColor'
```

Most modeling tools will map similarly named textures to the above textures in the FBX file. For instance, using Luxology MODO, the 'Emissive color' texture is mapped to the 'EmissiveColor' texture in the FBX file.

During rendering the diffuse texture is multiplied with the emissive texture as follows:

```
color = DiffuseColor(uv0) * EmissiveColor(uv1) * 1.5
```

Surface reflections look into a cube map (or environment map). The textures for the 6 cube map sides should be named:

```
<name>_right.<ext>  
<name>_left.<ext>  
<name>_up.<ext>  
<name>_down.<ext>  
<name>_backward.<ext>  
<name>_forward.<ext>
```

The reflection texture 'ReflectionColor' should be set to one of these 6 textures used to create the cube map. The FBX Converter automatically picks up the other 5 textures and combines all 6 textures into a cube map.

The normal map that is used to calculate the surface reflection is expected to be in local (tangent) space. During rendering the color of reflection mapped materials is calculated as follows:

```

surfaceNormal = normalize( NormalMap(uv0).x * tangent +
                           NormalMap(uv0).y * binormal +
                           NormalMap(uv0).z * normal )
reflection = dot( eyeVector, surfaceNormal ) * 2.0 * surfaceNormal - eyeVector
color = DiffuseColor(uv0) * EmissiveColor(uv1) * 1.5 +
        SpecularColor(uv0) * ReflectionColor(reflection)

```

The material type is one of the following:

1. opaque
2. perforated
3. transparent
4. additive

The first three material types are based on the alpha channel of the diffuse texture. The `-alpha` command-line option must be used to enable the 'perforated' and 'transparent' material types. Without the `-alpha` command-line option, the alpha channel of the diffuse texture will be removed.

The 'additive' material type cannot be derived from the textures. An additive texture is specified by appending `_additive` to the material name in the FBX file.

Animations

There is currently not a full blown animation system, but having vertices weighted to joints is still very useful to programmatically move geometry, while rendering as few surfaces as possible. Think about things like the buttons and joystick on the arcade machines in VrArcade. An artist can setup the vertex weighting for skinning, but the FBX Converter also has an option to rigidly bind the vertices of a FBX source mesh to a single joint. In this case the joint name will be the name of the FBX source mesh. The meshes that need to be rigidly skinned to a joint are specified using the `-skin` command-line option. There is currently a limit of 16 joints per FBX file.

The FBX Converter can also apply some very basic parametric animations to joints.

These simple animations are specified using the `-anim` command-line option. The types of animation are `rotate`, `sway` and `bob`. One of these types is specified directly following the `-anim` command-line option. Several parameters that define the animation are specified after the type. For the `rotate` and `sway` these parameters are `pitch`, `yaw` and `roll` in degrees per second. For the `bob` the parameters are `x`, `y` and `z` in meters per second. Following these parameters, a time offset and scale can be specified. The time offset is typically used to animate multiple joints out of sync and the time scale can be used to speed up or slow down the animation. Last but not least, one or more joints are specified to which the animation should be applied.

When a mesh is rigidly skinned to a joint using the `-skin` command-line option, the FBX Converter stores the mesh node transform on the joint. This mesh node transform is used as the frame of reference (pivot and axes) for animations.

Tags

A tag is used to define a position and frame of reference in the world.

A tag can, for instance, be used to define a screen or a view position in a cinema. A tag can also be used to place objects in the world.

The `-tag` command-line option is used to turn one or more FBX meshes from the render model into tags. The name of a tag will be the name of the mesh. The position and frame of reference are derived from the first triangle of the mesh and are stored in a 4x4 matrix. The position is the corner of the triangle that is most orthogonal. The edges that come out of this corner define the first two basis vectors of the frame of reference.

These basis vectors are not normalized to maintain the dimensions of the frame. The third basis vector is the triangle normal vector.

Multiple tags can be created by specifying multiple FBX mesh names after the `-tag` command-line option. The `-tag` command-line option does not remove the listed meshes from the render model. The `-remove` command-line option can be used to remove the meshes from the render model.

Command-Line Interface

The FBX Converter is a command-line tool.

To run the FBX Converter open a Windows Command Prompt, which can be found in the Windows Start menu under *All Programs -> Accessories*. A command prompt can also be opened by typing `cmd` in the Windows Run prompt in the Start menu. Once a command prompt has been opened, we recommend launching the FBX Converter from the folder where the source FBX files are located.

The FBX Converter comes with the following tools:

<code>FbxConvert64.exe</code>	(from Oculus VR)
<code>TimeStamp64.exe</code>	(from Oculus VR)
<code>PVRTexTool/*</code>	(version 3.4, from the PowerVR SDK version 3.3)
<code>7Zip/*</code>	(version 9.20, from www.7-zip.org)

The `FbxConvert64.exe` is the executable that is launched by the user. The other executables are directly or indirectly used by the `FbxConvert64.exe` executable.

Options

The FBX Converter supports the following command-line options:

Command	Description
<code>-o <output></code>	Specify the name for the .ovrscene file. Specify this name without extension.
<code>-render <model.fbx></code>	Specify model used for rendering.
<code>-collision <model.fbx meshes></code>	Specify model or meshes for wall collision.
<code>-ground <model.fbx meshes></code>	Specify model or meshes for floor collision.
<code>-raytrace <model.fbx meshes></code>	Specify model or meshes for focus tracking.
<code>-translate <x> <y> <z></code>	Translate the models by x,y,z.
<code>-rotate <degrees></code>	Rotate the models about the Y axis.
<code>-scale <factor></code>	Scale the models by the given factor.
<code>-swapXZ</code>	Swap the X and Z axis.
<code>-flipU</code>	Flip the U texture coordinate.
<code>-flipV</code>	Flip the V texture coordinate.
<code>-stripModoNumbers</code>	Strip duplicate name numbers added by MODO.
<code>-sort <+ -><X Y Z origin></code>	Sort geometry along axis or from origin.
<code>-expand <dist></code>	Expand collision walls by this distance. Defaults to 0.5
<code>-remove <mesh1> [<mesh2> ...]</code>	Remove these source meshes for rendering.
<code>-atlas <mesh1> [<mesh2> ...]</code>	Create texture atlas for these meshes.
<code>-discrete <mesh1> [<mesh2> ...]</code>	Keep these meshes separate for rendering.

Command	Description
-skin <mesh1> [<mesh2> ...]	Skin these source meshes rigidly to a joint.
-tag <mesh1> [<mesh2> ...]	Turn 1st triangles of these meshes into tags.
-attrib <attr1> [<attr2> ...]	Only keep these attributes: [position, normal, tangent, binormal, color, uv0, uv1, auto].
-anim <rotate> <pitch> <yaw> <roll> <timeoffset> <timescale> <joint1> [<joint2> ...] -anim <sway> <pitch> <yaw> <roll> <timeoffset> <timescale> <joint1> [<joint2> ...] -anim <bob> <X> <Y> <Z> <timeoffset> <timescale> <joint1> [<joint2>...]	Apply parametric animation to joints.
-ktx	Compress textures to KTX files (default).
-pvr	Compress textures to PVR files.
-dds	Compress textures to DDS files.
-alpha	Keep texture alpha channels if present.
-clean	Delete previously compressed textures.
-include <file1> [<file2> ...]	Include these files in the package.
-pack	Automatically run <output>_pack.bat file.
-zip <x>	7-Zip compression level (0=none, 9=ultra).
-fullText	Store binary data as text in JSON file.
-noPush	Do not push to device in batch file.
-noTest	Do not run a test scene from batch file.
-cinema	Launch VrCinema instead of VrScene.
-expo	Launch VrExpo instead of VrScene.

The `-collision`, `-ground` and `-raytrace` command-line options may either specify a separate FBX file or a list of meshes from the FBX file specified with the `-render` command-line option. If the collision and ray-trace meshes are in the same FBX file as the to be rendered meshes but the collision and ray-trace surface should not be rendered, then these meshes can be removed for rendering using the `-remove` command-line option.

Note that the `-collision`, `-ground`, `-raytrace`, `-remove`, `-atlas`, `-discrete`, `-skin` and `-tag` command-line options accept wild cards like `*` and `?`. For instance, to make all surfaces discrete use: `-discrete *`

Batch Execution

Instead of typing all the command-line options on the command prompt, it is common practice to use a batch file to launch the FBX Converter with a number of options. This allows for quick iteration on the assets while consistently using the same settings. The following is the contents of the batch file that was used to convert the FBX for the home theater:

```
FbxConvertx64.exe -o home_theater -pack -stripModoNumbers -rotate 180 -scale 0.01
  -translate 0.45 0 -3 -swapxz -flipv -sort origin -tag screen -render
  home_theater\home_theater.fbx -raytrace screen
```

Troubleshooting

The FBX Converter prints several things on the screen such as configuration options and warnings and errors. Warnings (e.g., missing textures) are printed in yellow, and errors (e.g., missing executables) are printed in red.

Optimization

The FBX Converter implements various command-line options that can be used to optimize the geometry for rendering.

Reducing Draw Calls

The FBX Converter automatically merges FBX meshes that use the same material such that they will be rendered as a single surface. At some point it may become necessary to automatically break up surfaces for culling granularity. However, currently it is more important to reduce the number of draw calls due to significant driver overhead on mobile platforms. Source meshes that need to stay separate for some reason can be flagged using the `-discrete` command-line option of the FBX Converter.

To further reduce the number of draw calls, or to statically sort all geometry into a single surface, the FBX Converter can also create one or more texture atlases using the `-atlas` option. This option takes a list of FBX source meshes that need to have their textures combined into an atlas. Multiple atlases can be created by specifying the `-atlas` command-line option multiple times with different mesh names. Textures that are placed in an atlas cannot be tiled (repeated) on a mesh and the texture coordinates of the source mesh need to all be in the $[0, 1]$ range.

Reducing Vertices

During conversion, the FBX Converter displays the total number of triangles and the total number of vertices of the render geometry. The number of vertices is expected to be in the same ballpark as the number of triangles. Having over two times more vertices than triangles may have performance implications. The number of unique vertices can be reduced by removing vertex attributes that are not necessary for rendering. Unused vertex attributes are generally wasteful and removing them may increase rendering performance just by improving GPU vertex cache usage.

An FBX file may store vertex attributes that are not used for rendering. For instance, vertex normals may be stored in the FBX file, but they will not be used for rendering unless there is some form of specular lighting. The FBX file may also store a second set of texture coordinates that are not used when there are no emissive textures. The `-attrib` command-line option of the FBX Converter can be used to keep only those attributes that are necessary to correctly render the model. For instance, if the model only renders a diffuse texture with baked lighting then all unnecessary vertex attributes can be removed by using `-attrib position uv0`.

The `-attrib` command-line option also accepts the `auto` keyword. By using the `auto` keyword the FBX Converter will automatically determine which vertex attributes need to be kept based on the textures specified per surface material. The `auto` keyword can be specified in combination with other vertex attributes. For instance: `-attrib auto color` will make sure that the color attribute will always be kept and the other vertex attributes will only be kept if they are needed to correctly render based on the specified textures. The following table shows how the FBX Converter determines which attributes to keep when the `auto` keyword is specified:

<code>position</code>	always automatically kept
<code>normal</code>	if <code>NormalMap</code> or <code>SpecularColor</code> texture is specified
<code>tangent</code>	if <code>NormalMap</code> texture is specified
<code>binormal</code>	if <code>NormalMap</code> texture is specified

uv0	if DiffuseColor or SpecularColor texture is specified
uv1	if EmissiveColor texture is specified
color	never automatically kept

Reducing Overdraw

To be able to render many triangles, it is important to minimize overdraw as much as possible. For scenes or models that do have overdraw, it is very important that the opaque geometry is rendered front-to-back to significantly reduce the number of shading operations. Scenes or models that will only be displayed from a single viewpoint, or a limited range of view points, can be statically sorted to guarantee front-to-back rendering on a per triangle basis.

The FBX Converter has a `-sort` command-line option to statically sort all the geometry. The `-sort` option first sorts all the vertices. Then it sorts all the triangles based on the smallest vertex index. Next to sorting all the triangles this also results in improved GPU vertex cache usage.

The `-sort` option can sort all geometry along one of the coordinate axes or it can sort all geometry from the origin. Sorting along an axis is useful for diorama-like scenes. Sorting from the origin is useful for theater-like scenes with a full 360 view.

Sorting along an axis is done by specifying `+` or `-` one of the coordinate axis (X, Y or Z). For instance, to sort all geometry front-to-back along the X axis use: `-sort +x`

Sorting from the origin can be done by specifying `+` or `-` `origin`. For instance, to sort all geometry front-to-back from the origin use: `-sort +origin`

For sorting from the origin to be effective, the origin of the FBX model or scene must be the point from which the model or scene will be viewed. Keep in mind that sorting from the origin happens after any translations have been applied to the FBX geometry using the `-translate` command-line option. In other words, when using the `-sort +origin` command-line option in combination with the `-translate` option, the scene will be sorted from the translated position instead of the original FBX origin.

Scenes that can be viewed from multiple vantage points may need to be manually broken up into reasonably sized blocks of geometry that will be dynamically sorted front-to-back at run-time. If the meshes the scene is broken up into use the same material, then the `-discrete` command-line option can be used to keep the meshes separate for rendering.

Testing and Troubleshooting

Welcome to the testing and troubleshooting guide.

Oculus Testing Tools and Procedures

Welcome to the testing and troubleshooting guide.

Developer Mode: Running Apps Outside of the Gear VR Headset

It may be useful to run VR applications during development without inserting your phone in the Gear VR headset, which can be a time-consuming process when making iterative changes. To run a VR application without loading it into the headset, you must enable Developer mode.

To enable Developer Mode:

- Go to Application Manager
- Select *Gear VR Service*
- Select *Manage Storage*
- Click on *VR Service Version* several times until the *Developer Mode* toggle shows up
- Toggle *Developer Mode*

 **Note:** Before enabling Developer Mode, you must have built an apk signed with your OSIG file and installed it to your phone. Otherwise, you will see an error message saying "You are not a developer!"

Oculus Remote Monitor

A monitoring tool for mobile development.

OVR::Capture

The Capture library is a low overhead remote monitoring system designed to help debug behavior and performance issues in mobile VR applications. It is capable of both real-time and offline inspection of collected data. Support is built into VrAPI by default and accessible when your device has VR Developer Mode enabled (see below).

Oculus Remote Monitor

The Monitor client (available for Mac OS X, Linux x86_64, and Windows) is the application that connects to VR applications running on remote devices. It is responsible for capturing, storing and displaying the data streamed in from the remote device.

Known Issues

- On slow networks, Capture's internal FIFO can fill up, causing stalls in the target application. This is easily detected in Monitor by looking for long stalls in the "OVR::Capture" thread in the performance/chart view. If this happens, it is best to disable "Capture Frame Buffer" before connecting to a remote device, as this feature consumes a large amount of network bandwidth. For this reason, we also recommend using a dedicated 802.11n+ network in close proximity, or tether directly to the device.

- GPU Zones are currently disabled due to driver issues related to timer queries.
- Integrated systrace support is currently disabled, as it is under development.
- Rare hangs may occur in remote applications if connection is unexpectedly severed.
- Some VPNs will break the auto-discovery mechanism. If you cannot connect to a device and your Client PC is connected to a VPN, you may need to disconnect from the VPN.

Setup

Network Setup

Oculus Remote Monitor uses a UDP-broadcast-based auto-discovery mechanism to locate remote hosts and then a TCP connection to access the capture stream. For this reason, the host and the client must be on the same subnet, and the network must not block or filter UDP broadcasts or TCP connections. If you are on a large corporate network that may have such restrictions, it is often ideal to set up a dedicated network or tether directly to your device. Furthermore, frame buffer capturing is extremely bandwidth intensive. If your signal strength is low or you have a lot of interference or traffic on your network, you may need to disable “Capture Frame Buffer” before connecting to improve capture performance.

Device Setup

For security reasons the Capture Library won't accept connections unless VR Developer Mode is turned on and Capture support is enabled in your device's Oculus Developer Preferences file.

VR Developer Mode

1. Go to *Application Manager > Gear VR Service > Manage Storage*.
2. Tap on the version number until the *Developer Mode* checkbox appears.
3. Tap the *Developer Mode* checkbox.

Oculus Developer Preferences

Append the line `dev_enableCapture 1` to the file “/sdcard/.oculusprefs” on your target device. A simple way to set the contents of this file is by issuing the following commands in your Terminal / Command Line:

```
adb shell
echo dev_enableCapture 1 > /sdcard/.oculusprefs
```

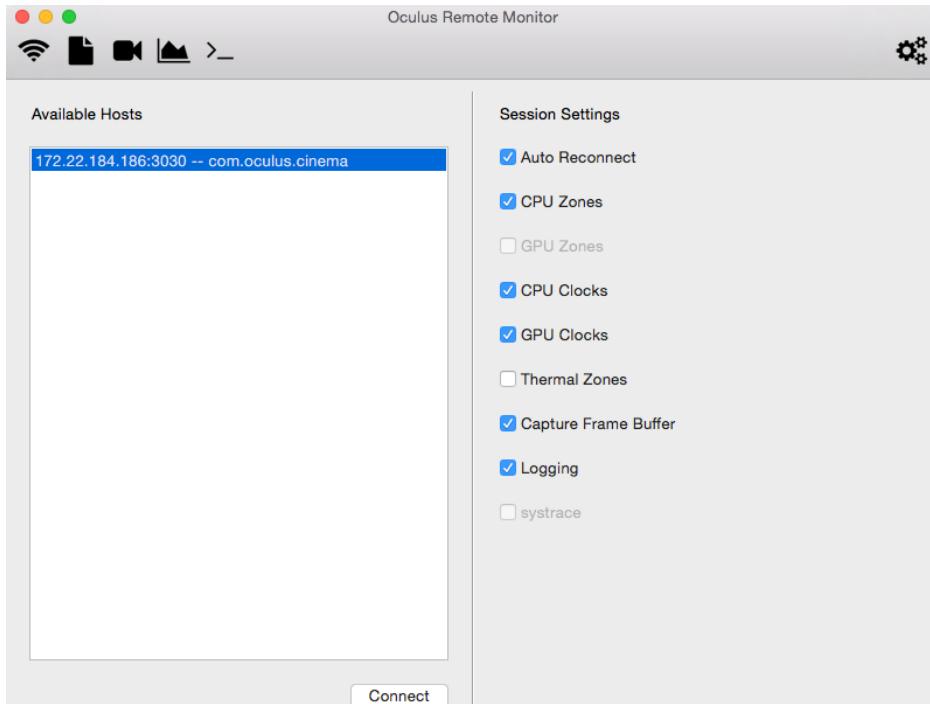
It is always a good idea to delete the Developer Preferences file when finished to prevent Capture support from turning on unexpectedly. This can be done simply:

```
adb shell rm /sdcard/.oculusprefs
```

Usage

Connecting

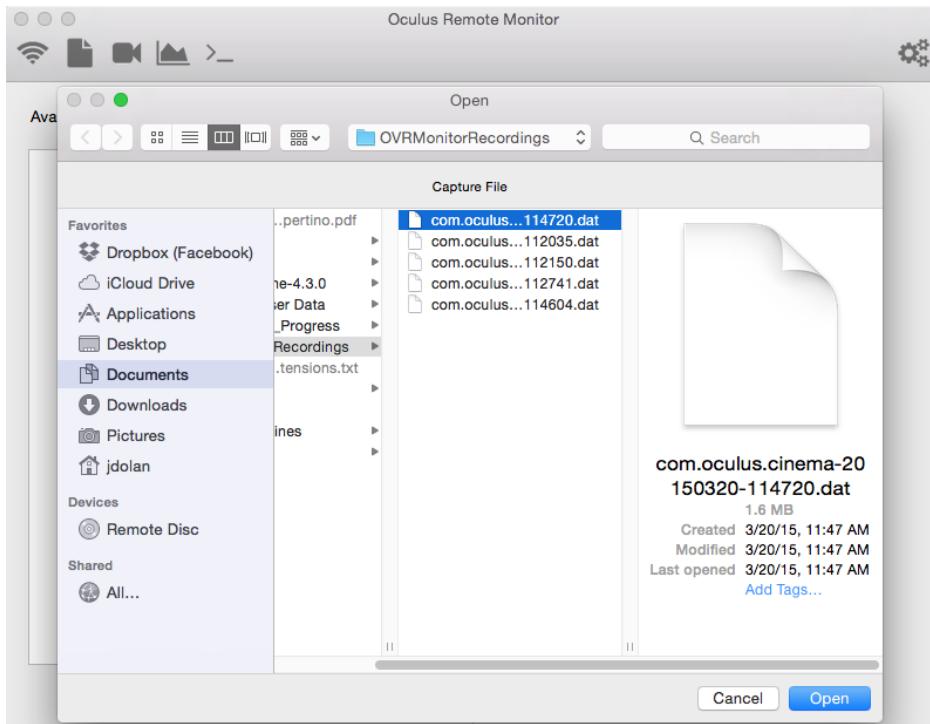
If the host and client are on the same subnet and the network is configured correctly (see 'Network Setup' above), Oculus Remote Monitor will automatically discover any compatible running application on the network. Simply select a host, toggle your desired *Session Settings*, and click *Connect* to begin capturing and viewing data.



Capture Files

Upon each connection, Oculus Remote Monitor will automatically compress and store the data stream to disk under a unique filename in the format "package-YYYYMMDD-HHMMSS.dat". By default, these files are stored under your *Documents* directory in a folder called *OVRMonitorRecordings*, although the path can be changed in Monitor's settings.

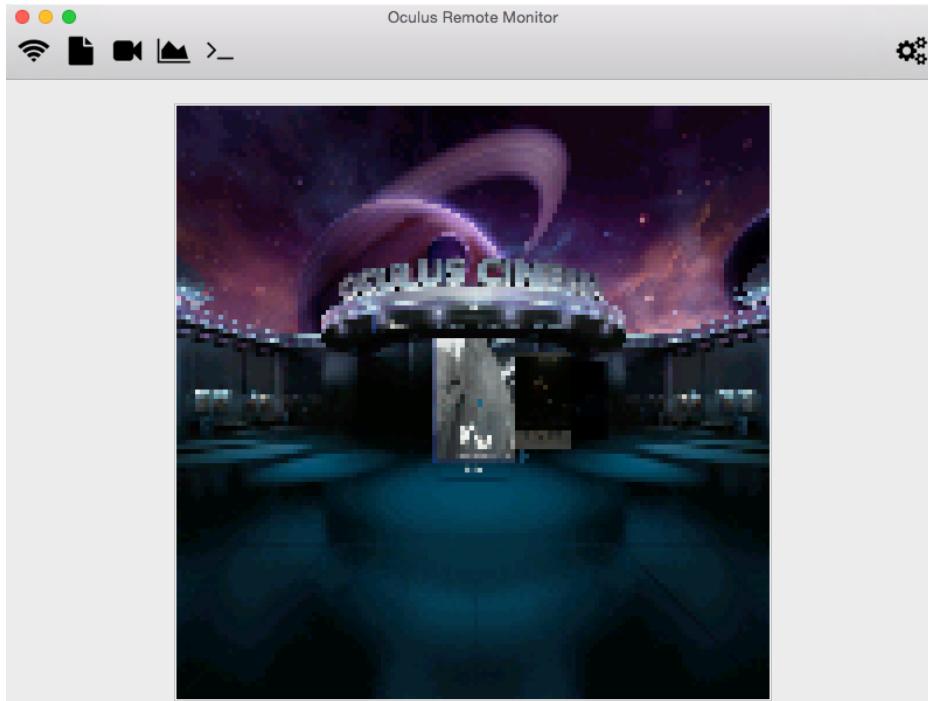
Old capture files may be opened and played back by clicking on the File icon in the toolbar.



Frame Buffer Viewer

The Frame Buffer Viewer provides a mechanism to inspect the frame buffer in real-time, as the data is received, which is particularly useful for monitoring play test sessions. When connected to a remote session or streaming from a capture file, click on the Camera icon in the toolbar to get instant access to the most recent frame buffer.

When enabled, the Capture library will stream a downscaled pre-distortion eye buffer across the network. While the quality may seem excessively low, the purpose of downscaling rather than using a higher-quality compression scheme is to reduce overhead on the host device as much as possible while still giving the client application context as to what is happening on screen at any given moment. 128x128@16-bits is the current default. The Monitor application recompresses the Frame Buffer to save memory and disk space when dealing with large capture sets.



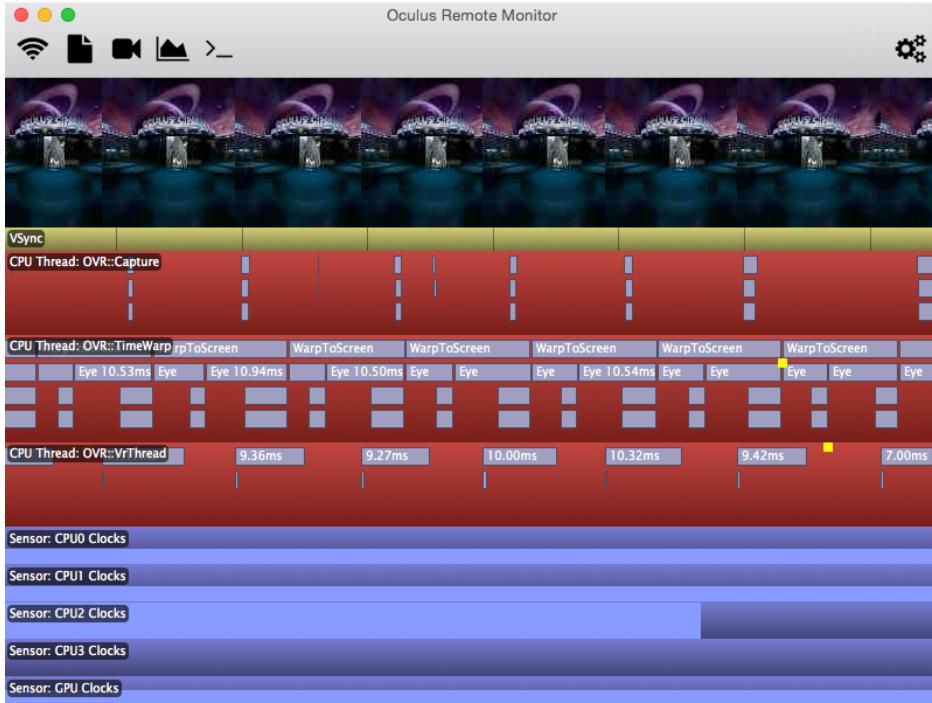
Performance Data

The Performance Data Viewer provides both real-time and offline inspection of CPU/GPU events, sensor readings, console messages/warnings/errors and frame buffer captures on a single contiguous timeline. By default, VrAPI has a minimal number of events embedded to help diagnose VR-specific scheduling issues, though this may change in the future.

Pressing the space bar toggles between real-time timelines scrolling and freezing at a specific point in time. This allows the user to go back and forth quickly between watching events unfold in real-time and stopping to focus in on a particular point of interest without stopping or restarting.

Scrolling the mouse wheel zooms the view in and out.

Clicking and dragging pans the timeline forwards or backwards in time.

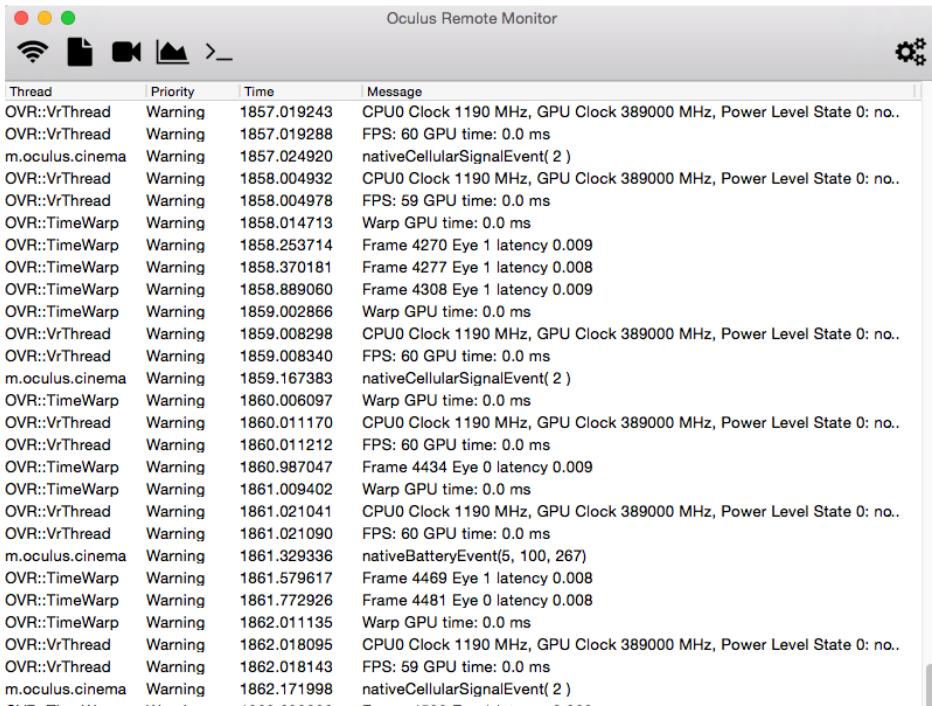


The above screenshot shows only some of the data available to capture.

- Frame Buffer - Provides screen captures of the pre-distorted frame buffer, timestamped the moment immediately before the frame was handed off to the TimeWarp context. The left edge of each screenshot represents the point in time in which it was captured from the GPU.
- VSync - Displays notches on every driver v-sync event.
- CPU Thread - Hierarchical visualization of wall clock time of various functions inside VrAPI. Log messages will be displayed on their corresponding CPU thread as little icons that you can mouse over to display the message: blue circles for messages, yellow squares for warnings, and red diamonds for errors.
- Sensor - General sensor data visualizer. CPU and GPU clocks are visualized in the screenshot shown above, but other data may be displayed here, such as thermal sensors, IMU data, et cetera.

Logging

VrAPI reports various messages and error conditions to Android's Logcat, and to the Oculus Remote Monitor as well, which also provides the thread and timestamp of each message. The Logging Viewer provides raw access to this data.



The screenshot shows the Oculus Remote Monitor application window. The title bar reads "Oculus Remote Monitor". The main area is a table with four columns: "Thread", "Priority", "Time", and "Message". The table contains numerous rows of log entries, mostly "Warning" priority, from various threads like OVR::VrThread, m.oculus.cinema, and OVR::TimeWarp. The log details system performance metrics such as CPU and GPU clock speeds, power levels, and frame latencies.

Thread	Priority	Time	Message
OVR::VrThread	Warning	1857.019243	CPU0 Clock 1190 MHz, GPU Clock 389000 MHz, Power Level State 0: no..
OVR::VrThread	Warning	1857.019288	FPS: 60 GPU time: 0.0 ms
m.oculus.cinema	Warning	1857.024920	nativeCellularSignalEvent(2)
OVR::VrThread	Warning	1858.004932	CPU0 Clock 1190 MHz, GPU Clock 389000 MHz, Power Level State 0: no..
OVR::VrThread	Warning	1858.004978	FPS: 59 GPU time: 0.0 ms
OVR::TimeWarp	Warning	1858.014713	Warp GPU time: 0.0 ms
OVR::TimeWarp	Warning	1858.253714	Frame 4270 Eye 1 latency 0.009
OVR::TimeWarp	Warning	1858.370181	Frame 4277 Eye 1 latency 0.008
OVR::TimeWarp	Warning	1858.889060	Frame 4308 Eye 1 latency 0.009
OVR::TimeWarp	Warning	1859.002866	Warp GPU time: 0.0 ms
OVR::VrThread	Warning	1859.008298	CPU0 Clock 1190 MHz, GPU Clock 389000 MHz, Power Level State 0: no..
OVR::VrThread	Warning	1859.008340	FPS: 60 GPU time: 0.0 ms
m.oculus.cinema	Warning	1859.167383	nativeCellularSignalEvent(2)
OVR::TimeWarp	Warning	1860.006097	Warp GPU time: 0.0 ms
OVR::VrThread	Warning	1860.011170	CPU0 Clock 1190 MHz, GPU Clock 389000 MHz, Power Level State 0: no..
OVR::VrThread	Warning	1860.011212	FPS: 60 GPU time: 0.0 ms
OVR::TimeWarp	Warning	1860.987047	Frame 4434 Eye 0 latency 0.009
OVR::TimeWarp	Warning	1861.009402	Warp GPU time: 0.0 ms
OVR::VrThread	Warning	1861.021041	CPU0 Clock 1190 MHz, GPU Clock 389000 MHz, Power Level State 0: no..
OVR::VrThread	Warning	1861.021090	FPS: 60 GPU time: 0.0 ms
m.oculus.cinema	Warning	1861.329336	nativeBatteryEvent(5, 100, 267)
OVR::TimeWarp	Warning	1861.579817	Frame 4469 Eye 1 latency 0.008
OVR::TimeWarp	Warning	1861.772926	Frame 4481 Eye 0 latency 0.008
OVR::TimeWarp	Warning	1862.011135	Warp GPU time: 0.0 ms
OVR::VrThread	Warning	1862.018095	CPU0 Clock 1190 MHz, GPU Clock 389000 MHz, Power Level State 0: no..
OVR::VrThread	Warning	1862.018143	FPS: 59 GPU time: 0.0 ms
m.oculus.cinema	Warning	1862.171998	nativeCellularSignalEvent(2)

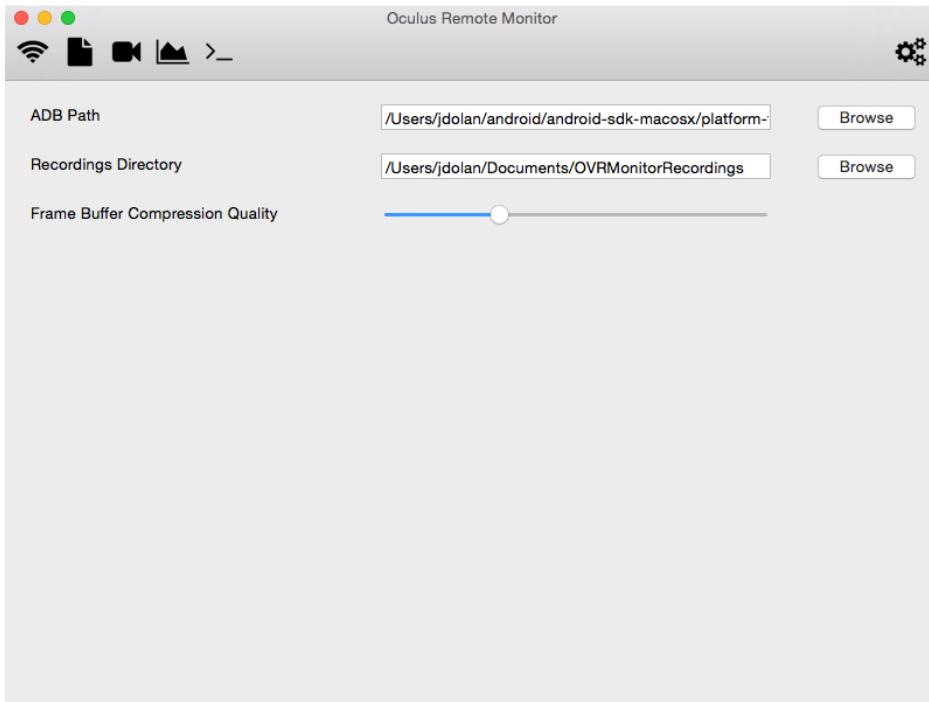
Settings

Monitor has a few persistent settings that may be adjusted by clicking on the gears icon in the upper-right of the toolbar.

You may set *ADB Path* to a desired path at any time. If it does not point to a valid executable when Monitor runs, it will attempt to locate a valid copy of ADB by checking to see if the environment variable ADB points to a specific copy. If that fails, Monitor will search under ANDROID_HOME.

Recordings Directory specifies the location in which Monitor will automatically store capture files when connected to a remote host. The default location is in the current user's *Documents* directory under *OVRMonitorRecordings*.

Frame Buffer Compression Quality is used for client-side recompression of the frame buffer. This helps offload compression load from the host while allowing for significant savings in memory usage on the client. Lower quality settings provide greater memory savings, but may result in blocking artifacts in the frame buffer viewer.



Local Preferences

Local Preferences are a set of key-value pairs written to `/sdcard/.oculusprefs` that are useful for storing debug settings that are tied to a device, rather than a particular application or user.

For security reasons, Local Preferences are not allowed unless Developer Mode is enabled for your device. To enable Developer Mode for your device, please refer to [Developer Mode](#).

The following Local Preferences are currently available via the Unity integration and VR app framework:

Preference	Acceptable Values	Function
<code>dev_enableCapture</code>	"0" or "1"	Enables support for Oculus Remote Monitor to connect to the application.
<code>dev_cpuLevel</code>	"0", "1", "2", "3"	Changes the fixed cpu level.
<code>dev_gpuLevel</code>	"0", "1", "2", "3"	Changes the fixed GPU level.
<code>dev_gpuTimings</code>	"0" or "1"	Turns on GPU timings in logcat (off by default due to instability).

To set a local preference value, you may do the following via command line:

```
adb shell "echo dev_gpuTimings 1 > /sdcard/.oculusprefs"
```

You may also set multiple values at once:

```
adb shell "echo dev_cpuLevel 1 dev_gpuLevel 2 > /sdcard/.oculusprefs"
```

After setting the values, pause and then resume your app for the Local Preferences file to be updated.

 **Note:** Remember to clear or remove your local prefs file when finished so there are no surprises.

For native applications, you may query your own Local Preferences debug option with the following:

```
const char * myDebugOptionStr =
    ovr_GetLocalPreferenceValueForKey( "myDebugOption", "0" );

if ( atoi( myDebugOptionStr ) > 0 )
{
// perform debug code
}
```

Android Debugging

A guide to Android debugging for mobile VR development.

Adb

This document describes utilities, tips and best practices for improving debugging for any application on Android platforms. Most of these tips apply to both native and Unity applications.

Android Debug Bridge (adb) is included in the Android SDK and is the main tool used to communicate with an Android device for debugging. We recommend familiarizing yourself with it by reading the official documentation located here: <http://developer.android.com/tools/help/adb.html>

Using adb

Using adb from the OS shell, it is possible to connect to and communicate with an Android device either directly through USB, or via TCP/IP over a WIFI connection. The Android Software Development Kit and appropriate device drivers must be installed before trying to use adb (see [Device and Environment Setup Guide](#) for more information).

To connect a device via USB, plug the device into the PC with a compatible USB cable. After connecting, open up an OS shell and type:

```
adb devices
```

If the device is connected properly, adb will show the device id list such as:

```
List of devices attached
ce0551e7          device
```

Adb may not be used if no device is detected. If your device is not listed, the most likely problem is that you do not have the correct Samsung USB driver - see [Device and Environment Setup Guide](#) for more information. You may also wish to try another USB cable and/or port.

```
- waiting for device -
```

Note that depending on the particular device, detection may be finicky from time to time. In particular, on some devices, connecting while a VR app is running or when adb is waiting for a device, may prevent the device from being reliably detected. In those cases, try ending the app and stop adb using Ctrl-C before reconnecting the

device. Alternatively the adb service can be stopped using the following command after which the adb service will be automatically restarted when executing the next command:

```
adb kill-server
```

Multiple devices may be attached at once, and this is often valuable for debugging client/server applications. Also, when connected via WIFI and also plugged in via USB, adb will show a single device as two devices. In the multiple device case adb must be told which device to work with using the `-s` switch. For example, with two devices connected, the `adb devices` command might show:

```
List of devices attached
ce0551e7          device
10.0.32.101:5555  device
```

The listed devices may be two separate devices, or one device that is connected both via WIFI and plugged into USB (perhaps to charge the battery). In this case, all adb commands must take the form:

```
adb -s <device id> <command>
```

where `<device id>` is the id reported by `adb devices`. So, for example, to issue a logcat command to the device connected via TCP/IP:

```
adb -s 10.0.32.101:5555 logcat -c
```

and to issue the same command to the device connected via USB:

```
adb -s ce0551e7
```

Connecting adb via WIFI

Connecting to a device via USB is generally faster than using a TCP/IP connection, but a TCP/IP connection is sometimes indispensable, especially when debugging behavior that only occurs when the phone is placed in the Gear VR, in which case the USB connection is occupied by the Gear VR jack.

To connect via TCP/IP, first determine the IP address of the device and make sure the device is already connected via USB. You can find the IP address of the device under Settings -> About device -> Status. Then issue the following commands:

```
adb tcpip <port>
adb connect <ipaddress>:<port>
```

For example:

```
> adb tcpip 5555
restarting in TCP mode port: 5555
> adb connect 10.0.32.101:5555
connected to 10.0.32.101:5555
```

The device may now be disconnected from the USB port. As long as `adb devices` shows only a single device, all adb commands will be issued for the device via WIFI.

To stop using the WIFI connection, issue the following adb command from the OS shell:

```
adb disconnect
```

Logcat

The Android SDK provides the logcat logging utility, which is essential for determining what an application and the Android OS are doing.

To use logcat, connect the Android device via USB or WIFI, launch an OS shell, and type:

```
adb logcat
```

If the device is connected and detected, the log will immediately begin outputting to the shell. In most cases, this raw output is too verbose to be useful. Logcat solves this by supporting filtering by tags. To see only a specific tag, use:

```
adb logcat -s <tag>
```

This example:

```
adb logcat -s VrApp
```

will show only output with the “VrApp” tag.

In the native VRLib code, messages can generally be printed using the LOG() macro. In most source files this is defined to pass a tag specific to that file. Log.h defines a few additional logging macros, but all resolve to calling __android_log_print().

Using Logcat to Determine the Cause of a Crash

Logcat will not necessarily be running when an application crashes. Fortunately, it keeps a buffer of recent output, and in many cases a command can be issued to logcat immediately after a crash to capture the log that includes the backtrace for the crash:

```
adb logcat > crash.log
```

Simply issue the above command, give the shell a moment to copy the buffered output to the log file, and then end adb (Ctrl+C in a Windows cmd prompt or OS X Terminal prompt). Then search the log for “backtrace:” to locate the stack trace beginning with the crash.

If too much time has elapsed and the log does not show the backtrace, there a full dump state of the crash should still exist. Use the following command to redirect the entire dumpstate to a file:

```
adb shell dumpstate > dumpstate.log
```

Copying the full dumpstate to a log file usually takes significantly longer than simply capturing the currently buffered logcat log, but it may provide additional information about the crash.

Getting a Better Stack Trace

The backtrace in a logcat capture or dumpstate generally shows the function where the crash occurred, but does not provide line numbering. To get more information about a crash, the Android Native Development Kit (NDK) must be installed. When the NDK is installed, the ndk-stack utility can be used to parse the logcat log or dumpstate for more detailed information about the state of the stack. To use ndk-stack, issue the following:

```
ndk-stack -sym <path to symbol file> -dump <source log file> > stack.log
```

For example, this command:

```
ndk-stack -sym VrNative\Oculus360Photos\obj\local\armeabi-v7a -dump crash.log > stack.log
```

uses the symbol information for Oculus 360 Photos to output a more detailed stack trace to a file named stack.log, using the backtrace found in crash.log.

Application Performance Analysis

A guide to performance analysis during mobile VR application development.

Performance Analysis

This document contains information specific to application performance analysis.

While this document is geared toward native application development, most of the tools presented here are also useful for improving performance in Android applications developed in Unity or other engines.

Application Performance

This section describes tools, methods and best practices for evaluating application performance.

In this section we review FPS Report, SysTrace, and NDK Profiler.

FPS Report

FPS logging with logcat.

The number of application frames per second, the GPU time to render the last eye scene rendering, and the GPU time to render the eye TimeWarp are reported to logcat once per second (for more information on logcat, see [Android Debugging](#)). Note that the GPU time reported does not include the time spent resolving the rendering back to main memory from on-chip memory, so it is an underestimate.

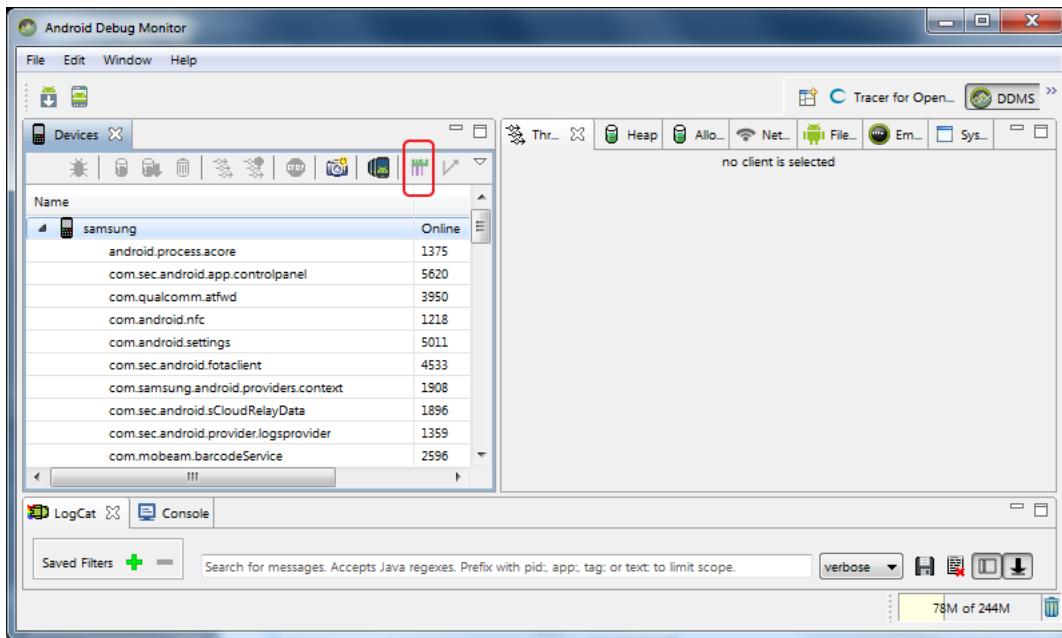
If the reported GPU time is over about 14 milliseconds, actual drawing is limiting the frame rate, and the resolution / geometry / shader complexity will need to be reduced to get back up to 60 FPS. If the GPU time is okay and the application is still not at 60 FPS, then the CPU is the bottleneck. In Unity, this could be either the UnityMain thread that runs scripts, or the UnityGfxDevice thread that issues OpenGL driver calls. Systrace is a good tool for investigating CPU utilization. If the UnityGfxDevice thread is taking longer blocks of time, reducing the number of draw call batches is the primary tool for improvement.

SysTrace

SysTrace is the profiling tool that comes with the Android Developer Tools (ADT) Bundle. SysTrace can record detailed logs of system activity that can be viewed in the Google Chrome browser.

With SysTrace, it is possible to see an overview of what the entire system is doing, rather than just a single app. This can be invaluable for resolving scheduling conflicts or finding out exactly why an app isn't performing as expected.

Under Windows: the simplest method for using SysTrace is to run the `monitor.bat` file that was installed with the ADT Bundle. This can be found in the ADT Bundle installation folder (e.g., `C:\android\adt-bundle-windows-x86_64-20131030`) under the `sdk/tools` folder. Double-click `monitor.bat` to start Android Debug Monitor.



Select the desired device in the left-hand column and click the icon highlighted in red above to toggle Systrace logging. A dialog will appear enabling selection of the output .html file for the trace. Once the trace is toggled off, the trace file can be viewed by opening it up in Google Chrome.

You can use the WASD keys to pan and zoom while navigating the HTML doc. For additional keyboard shortcuts, please refer to the following documentation: <http://developer.android.com/tools/help/systrace.html>

NDK Profiler

Use NDK Profiler to generate gprof-compatible profile information.

The Android NDK profiler is a port of gprof for Android.

The latest version, which has been tested with this release, is 3.2 and can be downloaded from the following location: <https://code.google.com/p/android-ndk-profiler/>

Once downloaded, unzip the package contents to your NDK sources path, e.g.: C:\Dev\Android\android-ndk-r9c\sources.

Add the NDK prebuilt tools to your PATH, e.g.: C:\Dev\Android\android-ndk-r9c\toolchains\arm-linux-androideabi-4.8\prebuilt\windows-x86_64\bin.

Android Makefile Modifications

1. Compile with profiling information and define NDK_PROFILE

```
LOCAL_CFLAGS := -pg -DNDK_PROFILE
```

2. Link with the ndk-profiler library

```
LOCAL_STATIC_LIBRARIES := android-ndk-profiler
```

3. Import the android-ndk-profiler module

```
$(call import-module, android-ndk-profiler)
```

Source Code Modifications

Add calls to `monstartup` and `moncleanup` to your `Init` and `Shutdown` functions. Do not call `monstartup` or `moncleanup` more than once during the lifetime of your app.

```
#if defined( NDK_PROFILE )
extern "C" void monstartup( char const * );
extern "C" void moncleanup();
#endif

extern "C" {

void Java_oculus_VrActivity2_nativeSetAppInterface( JNIEnv * jni, jclass clazz ) {

#if defined( NDK_PROFILE )
setenv( "CPUPROFILE_FREQUENCY", "500", 1 ); // interrupts per second, default
100
monstartup( "libovrapp.so" );
#endif

app->Init();
}

void Java_oculus_VrActivity2_nativeShutdown( JNIEnv *jni ) {

app->Shutdown();

#if defined( NDK_PROFILE )
moncleanup();
#endif
}

} // extern "C"
```

Manifest File Changes

You will need to add permission for your app to write to the SD card. The `gprof` output file is saved in `/sdcard/gmon.out`.

```
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
```

Profiling your App

To generate profiling data, run your app and trigger the `moncleanup` function call by pressing the Back button on your phone. Based on the state of your app, this will be triggered by `OnStop()` or `OnDestroy()`. Once `moncleanup` has been triggered, the profiling data will be written to your Android device at `/sdcard/gmon.out`.

Copy the `gmon.out` file to the folder where your project is located on your PC using the following command: `adb pull /sdcard/gmon.out`

To view the profile information, run the `gprof` tool, passing to it the non-stripped library, e.g.:

```
arm-linux-androideabi-gprof obj/local/armeabi/libovrapp.so
```

For information on interpreting the `gprof` profile information, see the following: <http://sourceware.org/binutils/docs/gprof/Output.html>

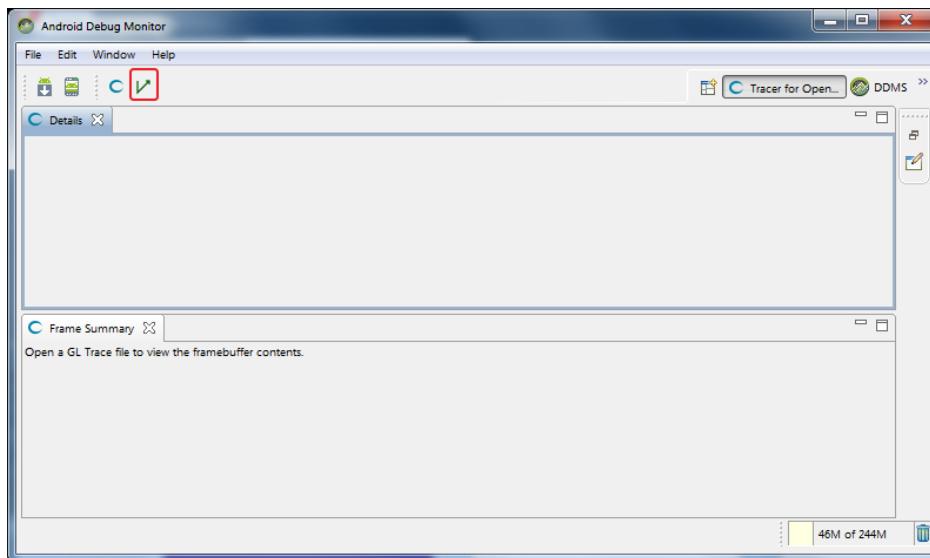
Rendering Performance: Tracer for OpenGL ES

Use Tracer for OpenGL ES to capture OpenGL ES calls for an application.

Tracer is a profiling tool that comes with the ADT Bundle.

1. Under Windows: the simplest method for using Tracer is to run the `monitor.bat` file that is installed with the ADT Bundle. This can be found in the ADT bundle installation folder (e.g., `C:\android\adt-bundle-windows-x86_64-20131030`) under the `sdk/tools` folder. Just double-click `monitor.bat` to start Android Debug Monitor.
2. Go to Windows -> Open Perspective... and select Tracer for OpenGL ES.
3. Click the Trace Capture button shown below.

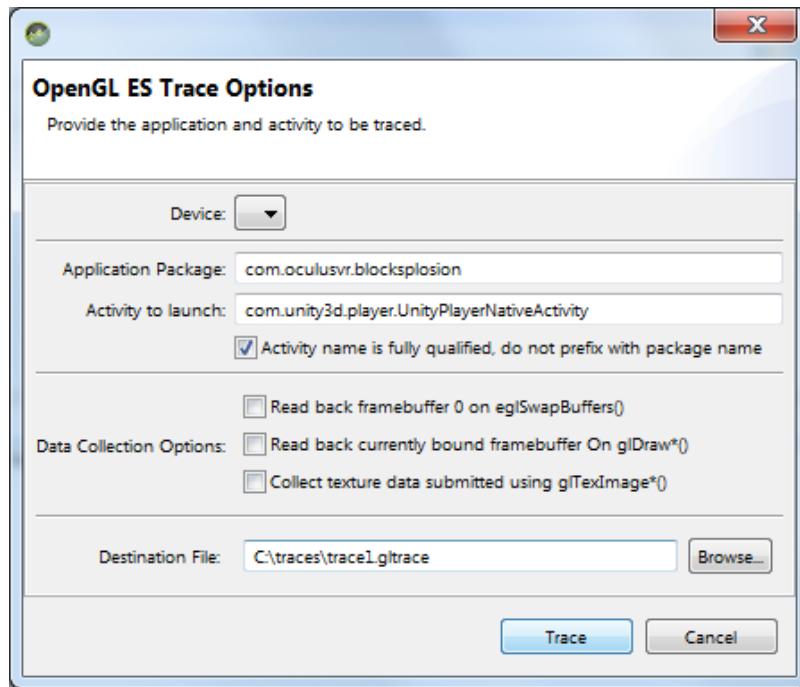
Figure 10: Tracer for OpenGL ES



4. Fill in the Trace Options and select Trace.

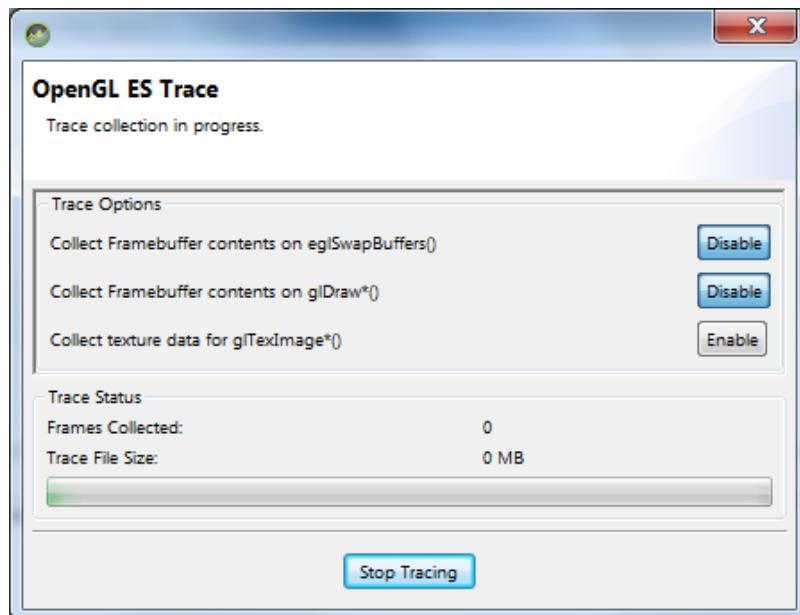


Note: A quick way to find the package name and Activity is to launch your app with logcat running. The Package Manager `getActivityInfo` line will display the information, e.g., `com.Oculus.Integration/com.unity3d.player.UnityPlayerNativeActivity`.



 **Note:** Selecting any Data Collection Options may cause the trace to become very large and slow to capture.

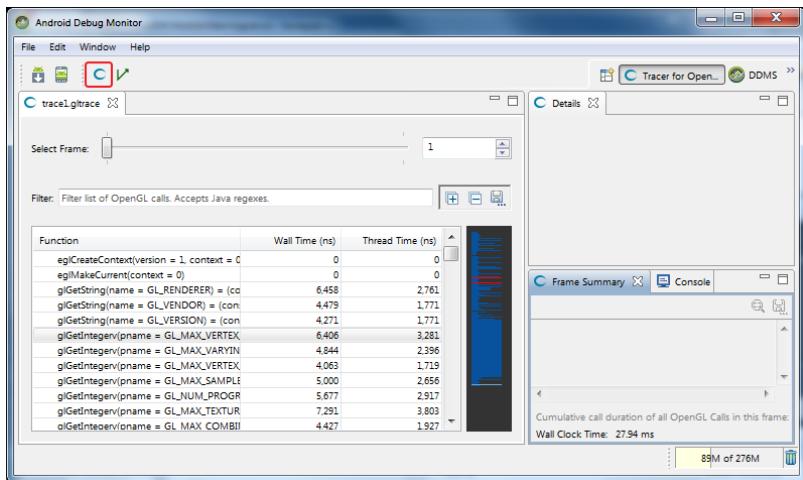
Tracer will launch your app and begin to capture frames.



5. Enable the following options:

- Collect Framebuffer contents on eglSwapBuffers()
- Collect Framebuffer contents on glDraw*()

6. Once you have collected enough frames, choose Stop Tracing.



7. Select the button highlighted above in red to import the .gltrace file you just captured. This can be a lengthy process. Once the trace is loaded, you can view GL commands for every captured frame.

Revision

Released March 31, 2015