Native Virtual Reality Software Development

v. 0.4

1. Introduction

This document provides a quick guide to the native SDK, which includes several sample projects and an overview of the native source code, and provides a basis for creating your own native virtual reality applications. 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 more elaborate environments such as Unity. The native SDK may not be feature rich, but it provides the basic infrastructure to get started with your own high performance virtual reality experience.

Before using this SDK and the documentation, if you have not already done so, 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 applications.

2. 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.

2.1 SDK Sample Overview

This SDK includes a few sample apps, some of which are implemented natively using the Android Native Development Kit (NDK), and some of which are implemented using Unity.

For a list of these sample apps please review the “Initial SDK Setup” section in *Device and Environment Setup Guide*.

These sample applications and associated data can be installed to the device by using one of the following methods with the device connected via USB:

1. Run installtophone.bat from your Oculus Mobile SDK directory, e.g.: C:\Dev\Oculus\Mobile\installToPhone.bat

2. Issue the following commands from C:\Dev\Oculus\Mobile\:

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

3. Copy using Windows Explorer (may be faster in some cases)

2.2 Native Sample Descriptions

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/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 document 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 one, 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 documentation 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

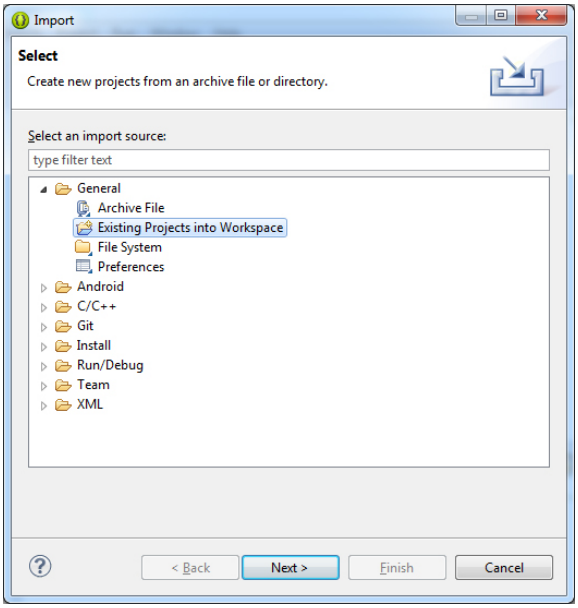
By default VrScene loads the Tuscany scene from the Oculus demos, which can be navigated using a gamepad. However, VrScene accepts Android Intents to view different .ovrscene files, so it can also be used as a generic scene viewer during development. New scenes can be created in Autodesk 3DS Max, Maya, 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.

2.3 Importing Native Samples in Eclipse

In order 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

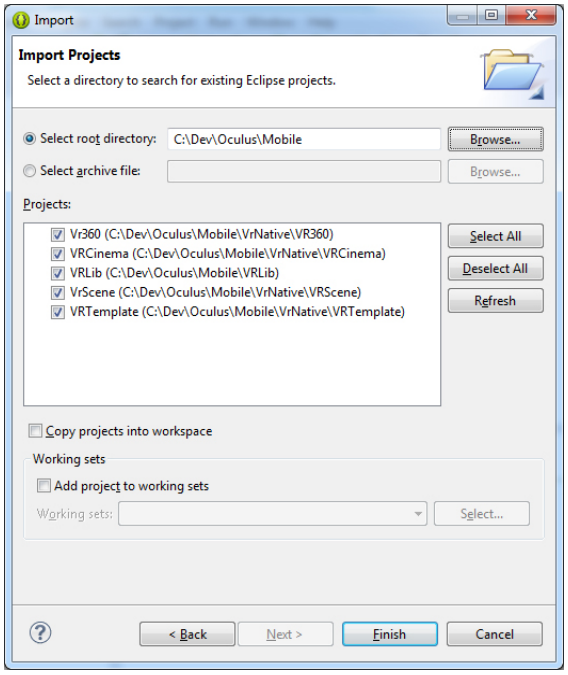
Device and Environment Setup Guide for ADT and Eclipse before importing or you will receive errors.

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



[Eclipse]

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



[Eclipse]

4. Browse to the location where the Oculus mobile software is that you wish to import. Setting your “Root Directory” and keeping all your apps under this folder will enumerate all of your apps.

5. Select the apps to import. In general, you should select the VrLib app if you have not previously imported it because 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*.

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

3. Native Source Code Overview

The following tables provide descriptions of the native source code files that are part of the framework.

Virtual Reality API

Source Files	Classes / Types	Description
VrApi\VrApi.cpp VrApi\VrApi.h VrApi\VrApi_local.h	ovrModeParms ovrHmdInfo ovrMobile HMTMountState_t batteryState_t	Minimum necessary API for mobile VR.
VrApi\HmdInfo.cpp VrApi\HmdInfo.h	hmdInfoInternal_t	Head Mounted Display Info.
VrApi\DirectRender.cpp VrApi\DirectRender.h	DirectRender	Multi-window front buffer setup.
VrApi\Vsync.cpp VrApi\Vsync.h	VsyncState	Interface to the raster vsync information.
VrApi\TimeWarp.cpp VrApi\TimeWarp.h VrApi\TimeWarpLocal.h VrApi\TimeWarpParms.h	TimeWarpInitParms TimeWarp TimeWarpImage TimeWarpParms	TimeWarp correct for optical distortion & reduction of motion-to-photon delay.
VrApi\ImageServer.cpp VrApi\ImageServer.h	ImageServer ImageServerRequest ImageServerResponse	Listen for requests to capture and send screenshots for viewing testers.
VrApi\LocalPreferences.cpp VrApi\LocalPreferences.h		Interface for device-local preferences.

Application

Source Files	Classes / Types	Description
App.h App.cpp AppLocal.h AppRender.cpp	VrAppInterface App AppLocal	Virtual application interface and local implementation. (Native counterpart to VrActivity).
PlatformActivity.cpp		Native implemnetation for the PlatformActivity. Also implements several other menus.
TalkToJava.cpp TalkToJava.h	TalkToJava TalkToJavaInterface	Thread and JNI management for background Java calls.
Input.h	VrInput VrFrame	Input and frame data passed each frame.
KeyState.cpp KeyState.h	KeyState	Keypress tracking.

Rendering

Source Files	Classes / Types	Description
GlUtils.cpp GlUtils.h	eglSetup_t	OpenGL headers, setup and convenience functions.
GlTexture.cpp GlTexture.h	GlTexture	OpenGL texture loading.
GlGeometry.cpp GlGeometry.h	GlGeometry	OpenGL geometry setup.
GlProgram.cpp GlProgram.h	GlProgram	OpenGL shader program compilation.
EyeBuffers.cpp EyeBuffers.h	EyeBuffers	Handling of different eye buffer implementations.

EyePostRender.cpp EyePostRender.h	EyePostRender	Rendering on top of an eye render.
ModelFile.cpp ModelFile.h	ModelFile ModelTexture ModelJoint ModelTag	Model file loading.
ModelRender.cpp ModelRender.h	ModelDef SurfaceDef MaterialDef ModelState	OpenGL rendering path.
ModelCollision.cpp ModelCollision.h	CollisionModel CollisionPolytope	Basic collision detection for scene walkthroughs.
ModelView.cpp ModelView.h	OvrSceneView ModelInScene	Basic viewing and movement in a scene.
SurfaceTexture.cpp SurfaceTexture.h	SurfaceTexture	Interface to Android SurfaceTexture objects.
BitmapFont.cpp BitmapFont.h	BitmapFont BitmapFontSurface	Bitmap font rendering.
DebugLines.cpp DebugLines.h	OvrDebugLines	Debug line rendering.

Ray-Tracing

Source Files	**Classes / Types**	**Description**
RayTracer\RtTrace.cpp RayTracer\RtTrace.h	RtTrace	Ray tracer using a KD-Tree.
RayTracer\RtIntersect.cpp RayTracer\RtIntersect.h	RtIntersect	Basic intersection routines.

User Interface

Source Files	Classes / Types	Description
GazeCursor.cpp GazeCursor.h	OvrGazeCursor	Global gaze cursor.
FolderBrowser.cpp FolderBrowser.h	OvrFolderBrowser	Content viewing and selection via gaze and swipe.
VRMenu\GlobalMenu.cpp VRMenu\GlobalMenu.h	GlobalMenu	Main menu that appears when pressing the Gear VR button.
VRMenu\VRMenuMgr.cpp VRMenu\VRMenuMgr.h	VRMenuMgr	Menu manager.
VRMenu\VRMenu.cpp VRMenu\VRMenu.h	VRMenu	Container for menu components and menu event handler.
VRMenu\VRMenuComponent.cpp VRMenu\VRMenuComponent.h	VRMenuComponent	Menu components.
VRMenu\VRMenuObject.h VRMenu\VRMenuObjectLocal.cpp VRMenu\VRMenuObjectLocal.h	VRMenuObject VRMenuObjectParms VRMenuSurfaceParms VRMenuComponentParms	Menu object.
VRMenu\VRMenuEventHandler.cpp VRMenu\VRMenuEventHandler.h	VRMenuEventHandler VRMenuEvent	Menu event handler.
VRMenu\SoundLimiter.cpp VRMenu\SoundLimiter.h	SoundLimiter	Utility class for limiting how often sounds play.

Utility

Source Files	Classes / Types	Description
VrCommon.cpp VrCommon.h		Common utility functions.
ImageData.cpp ImageData.h		Handling of raw image data.
MemBuffer.cpp MemBuffer.h	MemBuffer MemBufferFile	Memory buffer.
MessageQueue.cpp MessageQueue.h	MessageQueue	Thread communication by string commands.
Log.cpp Log.h	LogCpuTime LogGpuTime	Macros and helpers for Android logging and CPU and GPU timing.
SearchPaths.cpp SearchPaths.h	SearchPaths	Management of multiple content locations.
SoundManager.cpp SoundManager.h	OvrSoundManager	Sound asset manager using JSON definitions.

3.1 Native User Interface

Applications linking to VRLib have access to the VRMenu interface code. By default this interface hooks the long-press behavior to bring up an application menu that includes the above options. 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 a local bounds.

The main application menu (hereafter referred to as “universal menu”) is defined in jni/VRMenu/AppMenu.h and jni/VRMenu/GlobalMenu.cpp. By default this creates an application menu that includes the Home, Passthrough Camera, Reorient, Do Not Disturb and Comfort Mode options, along with various system state indicators such as WIFI and battery level..

VRMenu can be and is used to implement additional menus in a native applications, such as the Folder Browser control used in Oculus 360 Photos and Oculus 360 Videos.

PlatformActivity.cpp provides several examples of creating multiple native menus using separate VRMenu objects.

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 will be 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, etc. 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.
VRMenuId_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.
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.

3.2 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.

4. New Native Applications

4.1 Template Project

There is a VrTemplate project that is setup for exploratory work and to set up new similar native applications. VrTemplate is the best starting place for creating your own mobile app. 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 as a parameter, e.g.:

```
make_new_project.bat VrTestApp
```

Note: You will need to have cygwin installed for the sed utility.

2. With your Android device connected, execute the “run.bat” (“run.sh” in linux) located inside your test app directory to verify everything is working.
3. run.bat (“run.sh” in linux) should build your code, install it to the device, and launch your app on the device. There is one parameter for controlling the type of build you’re doing. Use `run debug` to generate a build for debugging. Use `run release` to generate a build for release. Use `run clean` to remove files generated by the build.

You may also test your app within Oculus Home by performing the following steps:

1. Browse to the location where you installed the SDK, e.g., C:\OculusUnity\Mobile
2. Open the Oculus Home JSON config file : .OculusHomeDebug\OculusHomeDebugConfig.json
3. Look for the “Your Test App” item under “HomePageItems”
4. Replace the packageName with your application’s package name, i.e., com.yourcompany.vrTestApp
5. Push the updated JSON config file to the following location on your device: /sdcard/.OculusHomeDebug
6. Reload Oculus Home and insert the device into the Gear VR.
7. Gaze and highlight the *Vr Test App* tile on the front page of the Home Menu. Tap the touchpad to launch the app.

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 `VrAppInterface` 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/. There is also convenience code for OpenGL ES texture, program, geometry, and scene processing that is used by the demos.

4.2 Integration

While it should be easy to pull code into the VrTemplate project, some knowledge about the different VR related systems is necessary to integrate them directly inside your own engine. The file VrLib/jni/VrApi/VrApi.h provides the minimum API for VR applications. The code in VrLib/jni/Integrations/UnityPlugin.cpp can be used as an example for integrating mobile VR in your own engine.

4.3 Android Manifest Settings

You will need to 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:configChanges="screenSize|orientation|keyboardHidden|keyboard">
  </activity>
</application>
<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">
</activity>
<uses-sdk android:minSdkVersion="19" android:targetSdkVersion="19" />
<uses-feature android:glEsVersion="0x00030000" />
<uses-permission android:name="android.permission.CAMERA" />
```

- 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.

- `PlatformActivity` represents the Universal Menu and is activated when the user long-presses the HMT button. The Universal Menu is implemented in VrLib and simply requires the activity to be included in your manifest.
- CAMERA permission is needed for the pass-through camera in the Universal Menu.
- **Do not** add the `noHistory` attribute to your manifest.

Note that submission requirements will have a few adjustments to these settings. Please refer to the submission guidelines available in our Developer Center: https://developer.oculus.com

Last Update: Nov 10, 2014