NATIVE GLOBE SAMPLE APPLICATION

March 2012

Developer Whitepaper

DOCUMENT CHANGE HISTORY

Document Number

Version	Date	Authors	Description of Change
01	3 March, 2012	Lbishop	Initial release

TABLE OF CONTENTS

INTRODUCTION

The Globe (**`sample_apps\native_globe`**) application is a simple Android NDK NativeActivity-based application that demonstrates simple 3D rendering in native code using Android. It also demonstrates reasonable reactions to common Android lifecycle events. The Globe application is written using the NVIDIA-expanded version of android_native_app_glue, nv_native_app_glue, but the basic concepts of handling Android lifecycle events are applicable to Java/native and pure native NDK applications independent of the particular framework library.

HIGHLIGHTS OF THE APPLICATION

The highlights of the Globe application at the user level include:

- Interactive 3D rendering of a shader-based "globe" object.
- Multiple OpenGL ES-rendered UI "dialogs" leading the user through the application's responses to Android lifecycle events. These are designed to simulate the in-game-engine rendered UIs common to most 3D games, rather than using Android's Java-based UI rendering.
- Correct, responsive handling of in-application device orientation changes.

HIGHLIGHTS OF THE APPLICATION CODE

In terms of code-level inspection, the Globe application provides quite a few useful examples to developers, including:

- Handling of major Android lifecycle events corresponding to common user actions, including:
  - Background'ing the application with the Home button
  - Pressing the Back button
  - Suspending and Resuming the device with the power button
  - Changing the device orientation at any time
- Touch-based user input
- Sound play/pause
- Examples of how to handle the Back button; the application chooses to "eat" the event internally or pass the event to Android, depending on the particular UI state
- Use of the nv_native_app_glue example framework

APPLICATION UI SCREEN AND TRANSITIONS

The Globe app has four basic UI screens, which are designed to be little more than placeholders to demonstrate different application modes. The four screens are:

- Startup screen
- Gameplay screen
- Auto-pause screen
- Quit confirmation screen

Each of these screens has a different set of basic behaviors.

THE SCREENS

The next few pages show screenshots of the four screens along with information about each screen and the common transitions.

Startup Screen

The Startup Screen is a placeholder for an application "splash" screen. The "Globe Demo" text and the "Tap to start" text are the only visual differences to the Gameplay Screen. The Startup Screen's main purpose is to serve as an indication that the application has been restarted from scratch.

- Tapping on this screen moves to the Gameplay Screen
- Pressing the Back button transitions to the Quit Confirmation Screen

Gameplay Screen

The Gameplay Screen is a stand-in for real game play interaction. In this mode, you can drag on the screen to rotate the globe interactively. In a real game, this would be active gameplay (in a race, playing a level, etc).

- Pressing the Back button transitions to the Auto-pause Screen

Auto-pause Screen

The Auto-pause screen is shown in multiple cases, normally when the application is resumed from an invisible or hidden case like suspend/resume and selecting a backgrounded instance of the application from the list of apps. It presents a dialog to the user to select between resuming to gameplay mode or quitting.

- Selecting Resume will transition to the Gameplay Screen
- Selecting Quit will transition to the Quit Confirmation Screen

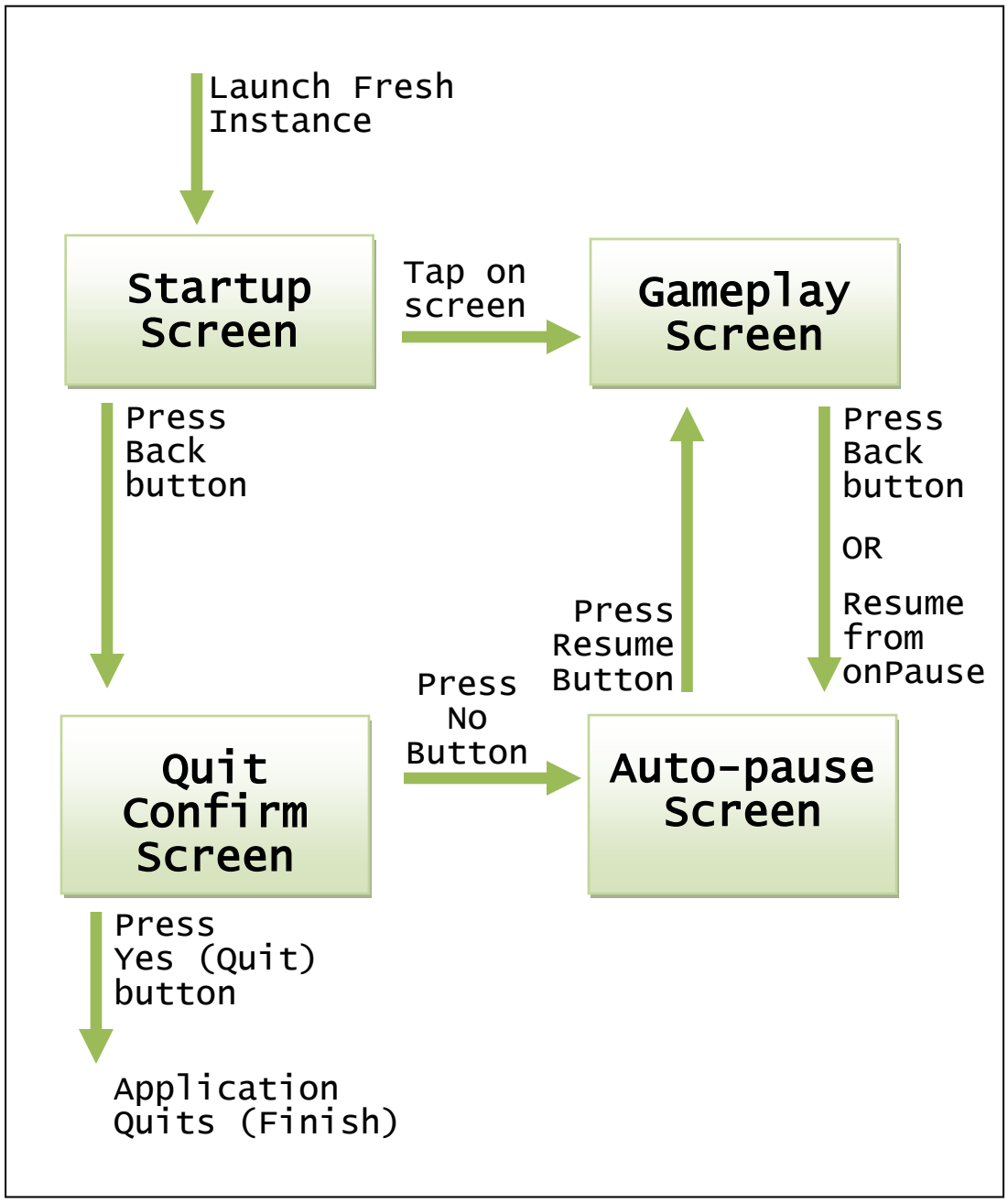
Quit Confirmation Screen

The Quit Confirmation Screen is the last chance for the user to avoid quitting the application. It is shown when the user selects Quit from the Auto-pause Screen or when the user presses the Back button at the Startup Screen

- Pressing the No button will transition to the Auto-pause Screen
- Pressing the Yes (Quit) button will quit (Finish) the application (Java **`onDestroy`**).

UI TRANSITION GRAPH

The four screens follow the basic flow shown below:



THE BASIC APPLICATION FLOW

At its core, the Globe app is a simple 3D application that renders a pair of concentric spheres with shaders (Earth's surface and a cloud layer) and allows for basic touch interaction. The interesting parts of Globe have to do with its handling of Android lifecycle events.

There are two basic parts to the Java/native JNI code interaction. Most of this interaction is handled by the Android support API, **`NativeActivity`**, which is in turn wrappered away from the application (in most cases) by NVIDIA's version of the Google sample framework, **`nv_native_app_glue`**. All of the passing of input events and lifecycle events down to the native code from the app process's Java class is handled by **`NativeActivity`** and **`nv_native_app_glue`**. In addition, **`nv_native_app_glue`** handles the signaling from the Globe app back up to Java when lifecycle events have been handled and whether key/button events have been "eaten" by the app or need to be passed on to Android. These **`nv_native_app_glue`** interactions are detailed in a later section of this document.

In addition to **`NativeActivity/nv_native_app_glue's`** event signaling, the Globe app also makes use of **`nv_native_app_glue's`** convenience queries. These queries are simple Booleans that **`nv_native_app_glue`** keeps track of for the application based on the lifecycle events that have been delivered. There is no unique information from these convenience functions; their return values could be shadowed by the Globe app simply by tracking the events posted to the queue and setting its own flags. However, since most applications can make use of similar flags, **`nv_native_app_glue`** tracks these states as a convenience to avoid duplicated code.

EXPLICITLY-HANDLED LIFECYCLE EVENTS

The Globe app handles most lifecycle cases in practice by overriding only a few Android lifecycle events. The aforementioned use of **`nv_native_app_glue's`** convenient state-tracking conditional functions avoids the need for the application to explicitly track these. As a result, most of the heavy-lifting of the lifecycle cases actually occurs in the application's rendering code, which is detailed later in this section.

Android lifecycle events not listed in the following subsections are not used explicitly by the Globe app.

APP_CMD_INIT_WINDOW and APP_CMD_WINDOW_RESIZED

When a new or changed window is provided to the application, it is passed to the EGL helper class instance via **`setWindow`**. Then, a forced render is requested. This will cause the system to check for an actual resize or replacement of the screen during the normal update and render sequence. No actual querying of the new size needs to be done at this time. The update and render frame functions handle this in a unified manner.

APP_CMD_TERM_WINDOW

When the window is closed, the application sets a NULL window on the EGL helper class instance, which will cause it to unbind the surface and context (if bound) and delete the rendering surface. The rendering context is kept resident to limit the work needed if and when the app receives a new surface.

APP_CMD_LOST_FOCUS and APP_CMD_PAUSE

The Globe app's reaction to either of these events is to check if the app is in Gameplay mode. If so, the application is transitioned to the auto-pause screen. Otherwise, the mode is left as-is, as the other screens are not live-rendered and are safe for backgrounded, non-focused lifecycle states. Also, the actual rendering code is specifically designed to avoid actively (repeatedly) rendering 3D except in Gameplay mode. However, a single force render is queued to ensure that the new screens are visible even if the mode is unchanged.

APP_CMD_CONFIG_CHANGED

Since configuration change can indicate that the screen has been rotated, especially on some Gingerbread devices, the application triggers a forced render when receiving this message.

HANDLING INPUT EVENTS

AINPUT_EVENT_TYPE_KEY

The main interest in the handling of the key events is with the Back button. Specifically, the handler returns 1 or 0 from the event function for the Back button event depending on mode. Returning 1 causes the **`nv_native_app_glue`** framework to "eat" the Back button event; it is not passed to Android. Returning 0 tells the **`nv_native_app_glue`** to pass the Back button event on to Android. Android, in turn, receiving the Back button, "finishes" or exits the application's Activity. This could also be handled by always eating the Back button events and manually calling the NativeActivity function **`ANativeActivity_finish`** to initiate the exit sequence.

In Gameplay mode, the Back button is eaten and handled by the app: it transitions to Auto-pause mode. In Auto-pause Screen or Startup Screen mode, Back button is eaten and handled by the app: it transitions to Quit Confirm Screen mode. Finally, in Quit Confirm Screen mode, the back button is passed on to Android to make the app quit.

AINPUT_EVENT_TYPE_MOTION

Touch events are handled in two different manners, depending on the UI mode. In Gameplay mode, the touch events are passed to the application function `touchEvent`, which implements virtual-sphere rotation of the globe object. In Startup Screen mode, touch input transitions the UI to Gameplay mode. In Auto-pause and Quit Confirmation modes, all touch input is handled by the GLES-rendered UI. The position of the clicks is compared to "hit boxes" that roughly surround the various "yes/no" areas in the GLES-rendered dialogs and the resulting UI transition is taken, possibly involving exit via **`ANativeActivity_finish`**.

THE RENDERING CODE

If there is any real core to the lifecycle handling in the Globe app, it is the code that handles per-frame rendering. The key function is **`renderFrame`**. **`renderFrame`** handles not only the rendering of a frame, but also just-in-time initialization of EGL and of the app's OpenGL ES resources. There are two modes of operation:

"Allocate if Needed" **`(allocateIfNeeded == true)`**. In this mode, the rendering code will request that the NVIDIA sample pack helper class **`NvEGLUtil`** set up EGL, its context and its surface. If this fails, the function will return failure. If EGL is already set up or can be set up, then the function will check if the application's OpenGL ES resources have already been loaded. If not, they are loaded. The function then renders a frame and swaps.

"Render only if ready" **`(allocateIfNeeded == false)`** In this mode, the function only renders if EGL, the context and surface and the app's OpenGL ES resources are already allocated. It will not allocate any of these if they are not present – it will return without rendering if all is not already in place.

This automatically handles setting up (and re-setting if lost) resources only as needed. This does not explicitly avoid rendering when we are not visible or not focused. The main loop code handles this by checking the **`nv_native_app_glue`** convenience function **`nv_app_status_interactable()`** before rendering. This function returns true only when the app is resumed and focused, which will block the app from rendering at inappropriate times.

Note that "interactable" in this case is defined to include "focused". This may not always be the case when a forced-render is needed. For example, if the user activates the Android status bar's notifications menu, this translucent menu will appear blended on top of the application's window. The menu display causes the application to lose focus, but still be visible. If the user then rotates their device, the screen will rotate to a different orientation and aspect ratio. Aspect change requires a forced redraw of the 3D scene, even though the application is not focused and thus is not interactable. The system handles this by allowing the request for a forced render to cause a redraw even if the application is not focused. However, unlike other rendering calls, this rendering call sets **`allocateIfNeeded`** to false, so that the forced rendering of the non-focused app only happens if it already has allocated EGL/GLES resources.

EXPERIMENTS WITH THE GLOBE APPLICATION

Users and developers can conduct some interesting experiments with the Globe application. The following sections discuss these experiments and what they mean for other applications and the Android application lifecycle.

BASIC LIFECYCLE EXPERIMENTS

Some of the basic cases that can be tested to see the way that the Globe application behaves include:

- Using the Back button to trigger the Auto-pause dialog. This is an example of an application "eating" an input event and not passing it back to Android.
- Rotating the device. This shows how the application rotates its rendering without being shut down.
- Suspend/Resume. This shows how the application handles the cessation and restarting of rendering with Suspend/Resume/Unlock, as well as the transition from Gameplay to Auto-pause when focus is lost.
- Backgrounding with the Home button and then bringing the application back from the list of running apps.

ORIENTATION AND FORCING ORIENTATION

By default, the shipping Globe application handles orientation changes and allows the screen to rotate. Since the UI is compact and simple, and the 3D rendering is confined to a square (rather than rectangular) subset of the screen, handling rotation is trivial. All the app has to do is listen for **`surfaceChanged`** events and adjust the **`glViewport`** and rendering aspect ratio accordingly. Since the app sets the manifest key:

```
android:configChanges="keyboardHidden|orientation"
```

It simply receives **`surfaceChanged`** events for each rotation (note that the app could also choose to override **`onConfigurationChanged`** in the app Java code and generate USER events on these rotations, but in the case of the globe, we have no need to understand the rotation itself, only the new screen size.

For many apps, this is not the case. The UI and/or the 3D rendering may be explicitly designed for one orientation or the other. To explore this option, one can add a single line to the AndroidManifest.xml for the Globe app. The existing block:

```
android:configChanges="keyboardHidden|orientation">
```

becomes

```
android:configChanges="keyboardHidden|orientation"
android:screenOrientation="landscape">
```

This causes the application to be forced to landscape orientation. The same key also works for portrait orientation with **`"portrait"`** as the value. Note that the application may still (in some cases) see brief moments at startup and suspend/resume where a pair of **`APP_CMD_WINDOW_RESIZED`** events will provide a portrait (wrong) and then landscape (correct) orientation. However, the application will not receive these in normal operation, even if the device is rotated.

Note that the selected forced portrait or landscape orientation is fixed for the device in this case. It may not match the particular portrait or landscape orientation in which the user is currently holding the device, requiring them to turn the tablet or phone without good reason. As of API level 9, Gingerbread, applications may request one of two orientations that lock the application to the landscape orientation closest to the device's current orientation or the portrait orientation closest to the device's current orientation. This locks an app's aspect ratio without forcing the user to a single way of holding the device. These tags are **`sensorLandscape`** and **`sensorPortrait`**.

This is a useful experiment in the Globe application to see the difference in behavior, and may also make the Globe app behave in a manner closer to the developer's intended application.

NOTICES

Notice

ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE.

Information furnished is believed to be accurate and reliable. However, NVIDIA Corporation assumes no responsibility for the consequences of use of such information or for any infringement of patents or other rights of third parties that may result from its use. No license is granted by implication of otherwise under any patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all other information previously supplied. NVIDIA Corporation products are not authorized as critical components in life support devices or systems without express written approval of NVIDIA Corporation.

HDMI

HDMI, the HDMI logo, and High-Definition Multimedia Interface are trademarks or registered trademarks of HDMI Licensing LLC.

ROVI Compliance Statement

NVIDIA Products that support Rovi Corporation's Revision 7.1.L1 Anti-Copy Process (ACP) encoding technology can only be sold or distributed to buyers with a valid and existing authorization from ROVI to purchase and incorporate the device into buyer's products.

This device is protected by U.S. patent numbers 6,516,132; 5,583,936; 6,836,549; 7,050,698; and 7,492,896 and other intellectual property rights. The use of ROVI Corporation's copy protection technology in the device must be authorized by ROVI Corporation and is intended for home and other limited pay-per-view uses only, unless otherwise authorized in writing by ROVI Corporation. Reverse engineering or disassembly is prohibited.

OpenCL

OpenCL is a trademark of Apple Inc. used under license to the Khronos Group Inc.

Trademarks

NVIDIA, the NVIDIA logo, and <add all the other product names listed in this document> are trademarks and/or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright