NVIDIA Tegra Android Samples

Version 2.0

Contents

Introduction

The Android Tegra samples pack contains sample applications focusing on demonstrating features of the Tegra platform using the Android Native Development Kit (NDK). The goal is to show developers how they could make use of the features we expose in Android in an application that is either being ported from an existing C/C++ code base or written in C/C++ from scratch, as well as provide helper libraries to make the transition to Android as easy as possible.

Samples Tree Overview

```
\apps
      \es2_globe
      \es2_water
      \event_accelerometer
      \event_lorenz
      \jniperftest
      \multitouch
```

These NDK samples demonstrate use of the accelerometer, JNI performance, OpenGLES2, multitouch input, as well as an event loop-based native application framework on top of the NDK.

```
\apps-sdk9
      \native_globe
      \native_lorenz
```

These NDK samples demonstrate use of the new native-only app frameworks `NativeActivity` and `native_app_glue` added to the NDK in r5 and supported on SDK level 9 and higher. Note that these applications will fail to install on OS images earlier than Android 2.3 (Gingerbread), as this is the first version of Android to support this feature.

\build

\platforms

Additional header and library files used by native code. Please note that EGL headers and library shipped in this folder are for debugging purpose only (using PerfHUD ES). Before shipping your application, be sure and recompile your application using the Android NDK headers and libraries.

\docs

Contains additional documentation besides this introductory document.

\libs

Contains helper library code used by the sample applications. In particular the NvActivity framework and its subclasses define a simple interface to make it as easy as possible to port existing C/C++ applications that interact well with Android without having to write much Java code. There are also helper functions for loading files and textures and managing the render loop. It also handles touch, multitouch, keyboard and accelerometer input. For more details of the implementation see the source code documentation and look at the source code of the samples to see how the helper libraries can be used. Finally, an additional library, `nv_event` implements an event loop framework for easier porting of classic event-driven interactive applications.

\libs-sdk9

Contains helper library code used by the SDK level 9 (`NativeActivity`) sample applications. In particular, there is an implementation of the APK-based file handler library that relies on SDK 9 NDK features in native code, instead of Java-level helper code.

\prebuilt

Contains prebuilt APK files of all the sample applications.

```
\tools
      \app_create
```

A Cygwin/bash shell script tool designed to make it easy to create new template Java/JNI applications. Applications created with this tool are ready to run and form a framework into which new or existing application code can be added. See the later section on "Creating new applications" for details on the use of this tool.

```
\install_samples.bat
```

A script that deploys the prebuilt samples along with any required data to an ADB-connected device. See the later section on "Installing the samples to the target device" for details on the use of this script. Note that this script installs the pre-built, shipped APKs from the `prebuilt` directory. It does not install locally-rebuilt APKs, which will reside in each application's `bin` subdirectory. The apps in the `prebuilt` directory will remain unchanged from the originally shipped binaries.

Installation

System Requirements

Please refer to the getting NVIDIA Tegra Android Getting Started document for details on the system requirements.

Prerequisites

Throughout this document it is assumed that all the required software and the "nice to have" components as described in the NVIDIA Tegra Android Getting Started document have been installed.

Installing the Android samples

Unzip the package into for example `c:\android\samples`

Required environment variables

These environment variables should be set either by right clicking My Computer →Properties→Advanced→Environment Variables, or directly in your Eclipse workspace via the menu item Window→Preferences→C/C++→Environment.

If you set these via My Computer, you'll have to restart Eclipse for these settings to take effect.

NDKROOT

Set the environment variable NDKROOT to the installation path of the NDK. For example `C:\android\android-ndk-r5b`

Windows specific

CYGWIN_HOME

Set the environment variable CYGWIN_HOME to the root of your Cygwin installation. For example `c:\cygwin`

CYGWIN (optional)

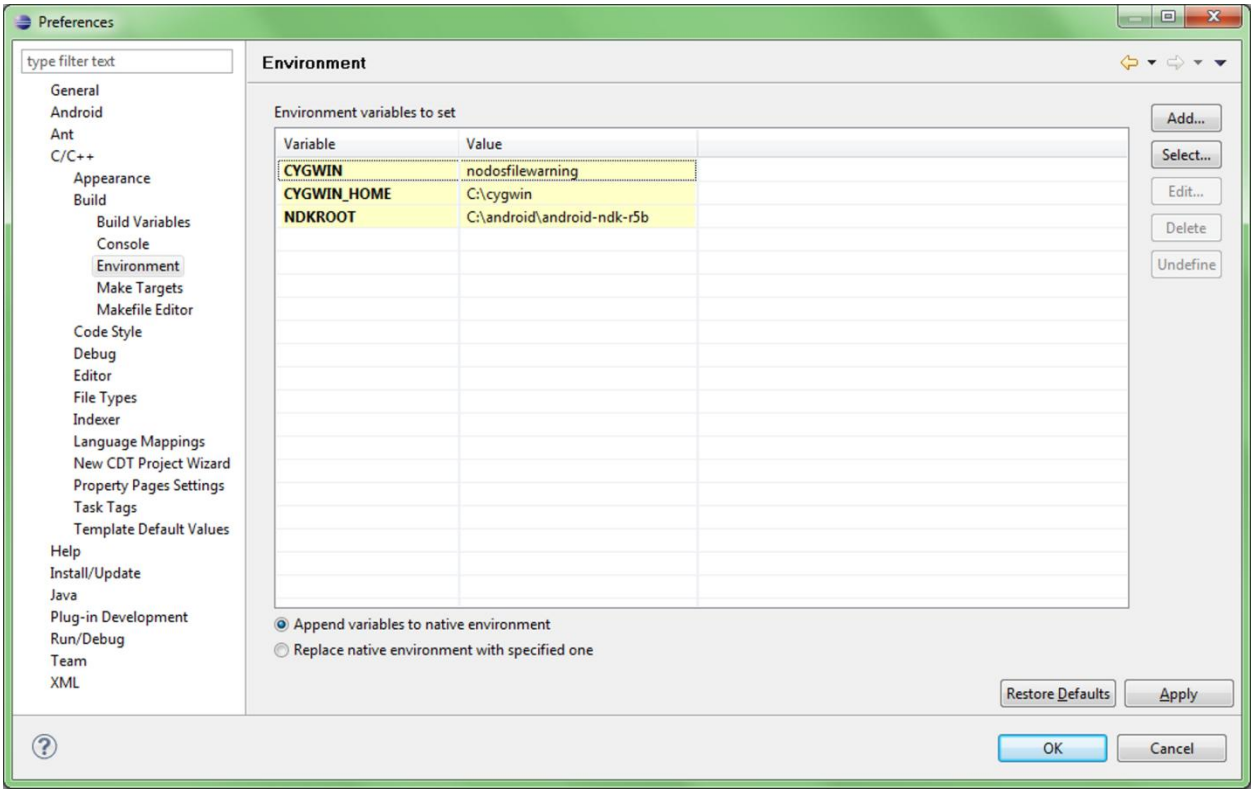
If Cygwin complains about "MS-DOS style path detected", setting the environment variable CYGWIN to "`nodosfilewarning`" will make the warning go away.

OS X and Linux specific

PLATFORM

Set the PLATFORM variable to either linux-x86 or darwin-x86 for Linux and OS X respectively.

An example of these variables set up in Eclipse on Windows might look like the following:



Installing the samples on the target device

To install the samples, please make sure adb is in your path and then double click the ***install_samples.bat*** file in the ***tools*** subfolder. This will execute the appropriate ADB commands to install all the sample applications and required data. After running the script the applications can now be found in the application list on the device.
If the applications do not appear, please look at the console output of the script as a pointer to what went wrong.

Note: If you have previously installed an earlier version of this SDK to your devkit hardware, you may receive errors such as:

"[INSTALL_PARSE_FAILED_INCONSISTENT_CERTIFICATES]".

If you see this error, uninstall the samples on the Tegra device by going to Settings→Applications→Manage Applications, then select the application you want to uninstall and press the Uninstall button. Try installing the samples again.

Running the samples

es2_globe

The `es2_globe` application is a 3d graphics intensive application that shows common idioms for high performance application development, such as index and vertex buffer object use. The globe can be interactively dragged via the touchscreen, or with a mouse connected to the devkit. Tapping (or mouse-clicking) in the top-left corner toggles between rendering from small, simple vertices and doing the computation of all attributes in the vertex shader, or simplifying the vertex shader and rendering from larger vertices. Tapping in the bottom left corner toggles the cloud layer on and off. Tapping in the top-right corner toggles the parallax mapping effect. Tapping in the bottom-right corner toggles the normal-mapped lighting on and off.

event_accelerometer

This sample shows how to properly transform Android accelerometer data with respect to the device's orientation.

JNIPerfTest

This application has no controls; it'll print some statistics about JNI performance out to logcat and will automatically exit when finished.

event_lorenz

`event_lorenz` shows an application using the SDK's `nv_event` framework; it also requires a keyboard connected to the devkit in order to interact with it. Pause the demo by pressing "p", switch to wireframe rendering by pressing "w", press "o" to view the model from the outside and press esc to exit.

multi

This shows differently colored and oriented triangles centered on each multitouch input – thus, two fingers will show as two different triangles, one under each finger.

native_globe

`native_globe` is written to behave the same way as `es2_globe`, but uses the pure-native code `NativeActivity` system added with Android's SDK 9 and NDK r5.

native_lorenz

`native_lorenz` is written to behave the same way as `event_lorenz`, but uses the pure-native code `NativeActivity` system. Pause the demo by pressing "p" or the volume up button, switch to wireframe rendering by pressing "w" or the volume down button, press "o" to view the model from the outside, or drag on the screen to control the time like a slider.

es2_water

Tap somewhere on the touchscreen to change the gravity vector. The direction of the gravity vector has the same direction as the vector between the center of the screen and the touch point. Press esc to exit the application.

Android Lifecycle Behavior of the Samples

Various samples behave differently from one another when the BACK and/or HOME buttons are pressed. The following section lists the expected behavior of each sample in these cases:

es2_globe

The globe app is based on `nv_event` and its "supports pause/resume" mode. The globe app is able to initialize its 3D resources quickly, so it simply releases all rendering resources on pause and reloads them on resume. This allows it to correctly handle pause/resume.

event_accelerometer

The `event_accelerometer` app's behavior is based on `nv_event's` "default: do not handle pause/resume" mode. In this mode, pause forces "finish" of the app. The process is not force-killed by the application. Launching the app again works correctly; no static data issues exist.

JNIPerfTest

If left to run to completion of all tests, the app closes its window, but the app process is still resident. Re-launching the app functions correctly, as static data is re-initialized. Pressing the BACK button is ignored, and while the HOME button will send the window away, the tests continue. Basically, the application does not include any lifecycle management, since it is designed to be a performance benchmark that runs to completion. It does, however, correctly handle relaunching without killing its own process.

event_lorenz

The nv_event-based version of the es2_lorenz app (present in previous versions of the NVIDIA Tegra Android Samples SDK, now deprecated) uses nv_event's "supports pause/resume" mode. The event_lorenz app is able to initialize its 3D resources quickly, so it simply releases all rendering resources on pause and reloads them on resume. This allows it to correctly handle pause/resume.

multi

multi always stays resident after being sent away with home or back, but when re-launched it re-runs initialization of the EGL/GLES code and is capable of re-launch.

native_globe

native_globe is written to behave the same way as es2_globe, but uses the pure-native lifecycle events sent by the android_native_app_glue wrapper (provided by Google) for NativeActivity. Specifically, the INIT_WINDOW and TERM_WINDOW messages are used to signal renderer creation and shutdown. The focus-based messages LOST_FOCUS and GAINED_FOCUS are used to signal the halting and restarting of rendering.

native_lorenz

native_lorenz is written to behave the same way as event_lorenz, but uses the pure-native lifecycle events sent by the android_native_app_glue wrapper for NativeActivity. Specifically, the INIT_WINDOW and TERM_WINDOW messages are used to signal renderer creation and shutdown. The focus-based messages LOST_FOCUS and GAINED_FOCUS are used to signal the halting and restarting of rendering.

es2_water

The es2_water app is based on nv_event and its "supports pause/resume" mode. The es2_water app is able to initialize its 3D resources quickly, so it simply releases all rendering resources on pause and reloads them on resume. This allows it to correctly handle pause/resume.

Compiling the samples

Setting up the Eclipse workspace

1. Create a new workspace (in Eclipse click File→Switch Workspace→Other…)

2. Select the menu Window→Preferences→Java→Build Path→Classpath Variables and add `ANDROID_JAR` with the path to the Éclair android.jar. For example `C:\android\android-sdk-windows\platforms\android-9\android.jar`

3. Set the path to the Android SDK via the menu Window→Preferences→Android→Android SDK Location. For example `c:\android\android-sdk-windows`

4. Import projects by selecting File→Import→General→Existing projects into workspace, and selecting the NVIDIA Android samples root. For example `c:\android\samples`. Make sure that the option "Copy projects into workspace" is NOT checked.

5. It is recommended that Project→Build Automatically be ***disabled***, as it can cause failed builds.

6. Build everything by selecting project→Build All.

7. **WARNING** – the rebuilt APK files will NOT be placed in `prebuilt`. They will be placed in the `bin` subdirectory for each app. Reinstalling the APKs from the `prebuilt` tree will simply reinstall the APKs shipped with the SDK, not the newly-built copies!

Note: Android SDK's API9 (or newer) is required in order to build the `NativeActivity`-based samples (`native_lorenz and native_globe, as well as any samples generated with the "native" template for app_create` tool). Keep that in mind when setting the value of the `ANDROID_JAR` variable in step 2.

Note: Sometimes the build step must be done multiple times to resolve errors about "missing required source folder: 'gen'". If you after building twice still get "The project cannot be built until build path errors are resolved", try editing a .java file in that project, save the change and build again.

Note: Another option for the above errors is to exit and restart Eclipse which will often allow the "build all" to complete the build process and clear the errors.

Note: If the compiler gives the error "must override a superclass method" for overloaded run functions, please make sure that the Compiler compliance level is set to 1.6. Go to Preferences→Java→Compiler and select 1.6 in the Compiler compliance level drop down list.

Note: If there still are errors after this, make sure everything is set up correctly according to this guide and check the problems view for more details about the error by selecting the menu item Window→Show View→Problems.

Note: If you have installed the prebuilt samples using the install_samples.bat script, you may receive the error

"[INSTALL_PARSE_FAILED_INCONSISTENT_CERTIFICATES]".

If you see this error, uninstall the samples on the Tegra device by going to Settings→Applications→Manage Applications, then select the application you want to uninstall and press the Uninstall button. Try running/installing the samples from Eclipse again.

Creating new applications

In order to ease the creation of new Android JNI native/Java samples that can be built from Eclipse, the SDK includes a bash shell script that generates new projects from a selection of templates. This tool is called `app_create`. It can create new applications in subdirectories of `apps/` that can be used as the basis for more complex application projects. To use the script, do the following:

1. Choose a path name for the application, e.g. "`my_app`". A directory of the given name under `apps/` (or `apps-sdk9/` in the case of `native` template apps) must **NOT** already exist prior to running the script, or the script will fail. This is a safety measure to avoid overwriting existing code.
2. Choose a Java class name for the application, e.g. "`MyApp`"
3. Select your desired application template. The options are:
   - `basic`: similar to the framework used for most of the current samples, a simple template with JNI exposed to the application
   - `nv_event`: A more complex framework, used to create the `event_lorenz` sample, wherein the JNI code is wrapped in a library and the application is presented with a more classic "event loop" and "event queue"
   - `native`: Supporting only Android 2.3 (Gingerbread) and newer OS images, applications created from this template have no Java code. The app is purely native and is based on `NativeActivity` and `android_native_app_glue`.
4. Open a Cygwin/bash shell to the SDK's `tools\app_create` directory
5. Run the script
   `./app_create.sh <app path name> <app java name> <template name>`

e.g.

```
./app_create.sh my_app MyApp nv_event
```

This will result in the new application being created in the indicated subdirectory of `apps/` or `apps-sdk9/` for `native` template-based apps. To load the new application into Eclipse for building, simply re-run the steps previously mentioned for "setting up the Eclipse workspace", selecting only the new project from the project import dialog. The new application project should be added to your workspace.

The project may be treated exactly like any other project in the SDK. New source files may be added to the app's `jni` directory, an `assets` directory may be created and filled with assets for the APK, and additional settings may be added to the `AndroidManifest.xml`.

All three versions of the application have the same basic features: they render a spinning GLES2 triangle and provide basic keyboard and touch controls. The difference in the versions of the application lie in the methods used to implement them. The `basic` template renders and handles events directly in JNI-called threads, while the `nv_event` version handles events and rendering in native-created threads to avoid stalling Android and avoid having JNI code in the application source.

The nv_event framework

Prior to Android 2.3's addition of NativeActivity, Android and most NDK frameworks used callbacks down into native code to implement each input event, each rendered frame, etc. Since shipping devices on Android 2.2 cannot support this new functionality, Marketplace applications may still need this kind of framework. There are several issues with this "callback per operation" model:

- Many applications do not fit this model, as the classic interactive rendering model tends to involve queued events and a loop in the application code for alternatively processing events and rendering
- Application code may not be prepared for the multi-threaded nature of Android's callbacks
- Android becomes less responsive, unresponsive or can even throw ANR (Application Not Responding) exceptions if a thread calling into native code from Java over JNI spends too long in native code without returning.

This SDK includes a paired set of library/Java class code that provides an event loop/event queue API on top of the Android callbacks. In addition, this library implements application setup and can avoid the need for any JNI code in the application itself. The `nv_event` framework consists of several basic native code APIs:

- **Event Queues:** Functions to read pending input and system events from a queue that is automatically filled by the framework. These functions are thread-safe.
- **EGL Functions**: Functions to bind and unbind the application's OpenGL ES 2.0 context to the current thread, as well as a function to swap buffers.
- **"main"-like Entrypoint**: An entrypoint that the application defines in native code that is called inside of a natively-created thread. This function can implement the classic event and render loop, and returns only when the application exits, rather than once per frame. Because it is invoked within its own thread, there is a greatly reduced risk of making the main thread and thus the Android UI less responsive.

Entrypoints in nv_event

Applications use `nv_event` by:

- Subclassing a trivial class from `NvEventQueueActivity` (the `app_create.sh` tool does this for you)
- Linking the `libnvevent.a` library and its dependent libraries (the `app_create.sh` tool does this for you)
- Implementing the two required application entrypoint functions (the `app_create.sh` tool creates stub versions of these for you to fill out)

There are two required entrypoints that any `nv_event` application must define; they are declared in `nv_event.h`:

```
extern int32_t NVEventAppInit(int32_t argc, char** argv);
```

Called by the framework in the main JNI thread from within a JNI function call, applications should use this function to call any initialization that must query application Java classes. Examples of code requiring initialization from within this function are documented in the libraries documentation, and include `nv_shader_init()` in `nv_shader.h`. Applications should avoid heavyweight initialization or loading of data in this function.

```
extern int32_t NVEventAppMain(int32_t argc, char** argv);
```

Called from within its own native-created thread, this function should be thought of as the "`main`" or `WinMain` equivalent. Applications should do their data loading, event looping, rendering and shutdown code from within this function, returning from the function only on application shutdown. While this thread is native and does not have to return to Java to avoid

stalling Android, the thread is linked to the JVM and can make JNI calls through the use of nv_thread.h's NVThreadGetCurrentJNIEnv().

Handling Pause and Resume

NvEvent runs in two different modes for handling Android lifecycle pause and resume events. "Quit on pause" and "support pause/resume". Each requires slightly different coding on the developer's part

The "quit on pause" mode is currently the default owing to historical reasons in the SDK. Note that new applications should consider using "support pause/resume" mode described below whenever possible. "Quit on pause" is set in java by setting the member supportPauseResume to false (the default) in onCreate. In this mode, NvEvent will automatically generate an NV_EVENT_QUIT event on pause. The application should shut down and return from NVEventAppMain, at which point NvEvent will finish (this will not kill the process; it simply does an Activity finish). NvEvent will also finish its Activity if and when the application returns from NVEventAppMain of its own accord. The app in this case cannot handle being sent to the background. It will simply quit in this case.

The "support pause/resume" mode is enabled by explicitly setting supportPauseResume to true in onCreate. In this mode, NvEvent will generate an NV_EVENT_RESUME event when the app is started or resumed, and will generate an NV_EVENT_PAUSE event when paused. Note that ***in this mode, the application must not initialize EGL and GLES resources until it receives an NV_EVENT_RESUME*** event. ***Do not initialize EGL or GLES prior to the event loop in this mode, or the calls will fail***. In this mode, NV_EVENT_RESUME indicates both resumption after pause and initial window readiness. On each NV_EVENT_PAUSE event, the application should release all GLES and EGL resources before returning from the event handler. The Globe app shows an example of this behavior. In this mode, the app will not exit on pause, and will instead be given the chance to keep running. However, applications should, in general go into a "sleep" mode when paused, waiting for events and looking for a quit or resume event before doing any significant work.

Using EGL in nv_event

Nv_event applications may or may not use EGL/GLES. Applications wishing to use EGL/GLES in NvEvent must first initialize EGL and a GLES context using NVEventEGLInit. Then, the NVEventAppMain thread or any nv_thread-created thread can bind the context and surface via NVEventEGLMakeCurrent(). Before binding to a second thread or before returning

from the main function, the bound thread should call `NVEventEGLUnmakeCurrent()`. A thread currently bound can call `NVEventEGLSwapBuffers()` to display the current backbuffer.

The time to call `NVEventEGLInit` and `NVEventEGLMakeCurrent` depends upon the pause/resume handling mode. Applications using the default "quit on pause" mode can initialize EGL at any time in `NVEventAppMain`. Applications running in "support pause/resume" mode must wait to receive an `NV_EVENT_RESUME` event in their main loop before initializing EGL. Doing so earlier in the application's lifecycle, or doing so between the receipt of an `NV_EVENT_PAUSE` event and an `NV_EVENT_RESUME` event will cause an exception in the app.

Event handling in nv_event

`nv_event` supports the reading of queued events via the function `NVEventGetNextEvent()`. This function is capable of returning immediately with or without an event, blocking until the next event is available, or waiting at most a given number of milliseconds for an event before retuning "no event". This gives applications the flexibility to avoid "spinning" on the event loop if they do not need to render at a high rate.

The `nv_event` system currently supports the following events. Some of these events (as noted) are not enabled by default, owing to their high rates of firing or their larger amounts of data to be copied:

`NV_EVENT_KEY`

Represents a key press or release, with keycodes matching the supplied enumerants in `nv_event.h`. Note that key repeats are not sent in this case. Also, key down events are generated for many non-character keys, such as shift.

`NV_EVENT_CHAR`

Represents the Unicode character of a key or multi-key press. Not all keys generate a character event, but unlike key events, character events will auto-repeat. A press-and-release of a single key can generate three events. Key down, character, and key up.

`NV_EVENT_TOUCH`

A single-touch touchscreen or mouse event. Note that mice are currently represented as touch devices. No motion events are generated unless there is a button down, and only one button is supported.

`NV_EVENT_MULTITOUCH`

Multi-touch (currently 1 or 2 fingers) events. These events are not generated by default. However, if the application's Java code is modified such that the member variable

`wantsMultitouch` is set to `true` in the function override `onCreate`, then multi-touch events will replace single-touch events for that application.

`NV_EVENT_ACCEL`

Accelerometer force vector events. These come in at a constant rate when enabled, and are thus disabled by default. However, if the application's Java code is modified such that the member variable `wantsAccelerometer` is set to `true` in the function override `onCreate`, then the accelerometer events will be posted to the event queue.

`NV_EVENT_WINDOW_SIZE`

Sent by the system when the window (and thus the OpenGL ES 2.0 rendering surface) has changed size. An instance of this event will be sent immediately upon startup to indicate that the rendering surface is render for rendering.

`NV_EVENT_QUIT`

Sent by the system when the application is requested to exit. The application should exit the event loop immediately and shut down, returning from the `NVEventAppMain()` function. Currently, there is no way for an application to cleanly exit other than by being requested to do so by Android (e.g. pressing the Home button). This will be fixed in an upcoming version of the framework.

`NV_EVENT_RESUME`

Sent by the system when the app's window becomes visible and focused, either for the first time (during launch) or following a pause event. The handler for this event should initialize EGL and GLES resources if those APIs are being used. This event is only sent to the application in "support pause/resume" mode.

`NV_EVENT_PAUSE`

Sent by the system when the app's window is no longer visible. The handler for this event should release GLES and EGL resources and unbind the EGL context if those APIs are being used. This event is only sent to the application in "support pause/resume" mode.

Loading texture images in nv_event

`Nv_event` includes a pair of functions that can load and release GLES-friendly texture data from APK assets or /data files. The functions are `NVEventGetTextureData()` and `NVEventReleaseTextureData()`, and they can be called as follows to load textures:

```
//Temporary variables for loading pixel data
unsigned char* pixels;
unsigned int width, height, format, type;
void *data;
unsigned int id;

glGenTextures((GLsizei)1, &id);

glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, id);
data = NVEventGetTextureData("tex.jpg", pixels,
                             width, height, format, type);
glTexImage2D(GL_TEXTURE_2D, (GLint)0, (GLint)format,
                     (GLsizei)width, (GLsizei)height, (GLint)0,
                     (GLenum)format, (GLenum)type,
                     (const GLvoid*)pixels);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
NVEventReleaseTextureData(data, pixels);
```

The calls to `Get` and `Release` must be balanced in order to avoid leaks. See the included library documentation for details.

Returning from the app's NVEventAppMain function

In either pause/resume mode, exiting the NVEventAppMain function should cause the app's process to exit.

Frequently asked questions

How can I set my app to run as home screen app?

In Android any app can be set to run as the home screen app by setting the category elements in the activity's intent-filter in the `AndroidManifest.xml` file as

```
<category android:name="android.intent.category.HOME"/>
<category android:name="android.intent.category.DEFAULT"/>
```

For the application to be able to be used as a home screen app, the board will have to be rebooted after the app has been installed. After reboot, since there will be more than one application with the home screen intent, the ActivityManager will prompt the user to select one of them to be used as the home screen. It will list the default home screen app (Home) and all other home screen apps that you have installed on the device.

It also gives an option to use the selected app by default for that activity. If this option is checked the user will not be prompted to choose from available home screen apps in subsequent boots. Otherwise the user will be prompted to choose an app each time device reboots (unless the apps are uninstalled).

http://justanapplication.wordpress.com/2009/08/22/a-standalone-android-runtime-launching-helloworld-the-easy-way/ shows how to change the `HelloAndroid` app to run as the home screen app.

Please note that it is up to the app to do the home screen tasks. The above changes only sets the app to run after reboot, it does not automatically add any home screen related features in your app (like Widget handling, listing app icons etc). Your app should handle any UI/home screen features explicitly.

Shaders not working on non-Tegra devices?

We have found some differences between Tegra and other shader compilers.

On Tegra, fragment shaders are not required to specify the precision of each floating point variable or default floating point precision. But this may be required by other shader compilers, and will generate a shader link error if not specified.

On Tegra while calculating the value of `gl_Position.x`, `gl_Position.y` and `gl_Position.z`, vertex shaders are not required to explicitly calculate `gl_Position.w`. But this may be required by other shader compilers.

Getting IllegalArgumentException in eglCreateContext() on non-Tegra devices?

Confirm that `eglChooseConfig()` returned at least one matching configuration. If not, find out which configurations are supported on the non-Tegra device that you are testing. For example, on Motorola DROID, `eglChooseConfig()` might return matching configs when requesting a 24-bit depth size instead of 16-bits.

Change History

01 March, 2011 Release

- Two samples applications have been added that use Android SDK 9's `NativeActivity` and `native_app_glue` classes.

- Builds of native code are now done using the NDK's `ndk-build`, making it possible to seamlessly support NDK r4b and r5b.
- Most applications now handle Android application lifecycle cases correctly and do not force process exit with `exit(0)`, which is not recommended.
- The multitouch demo now correctly handles up to 10 touch points at once.
- Fragment shaders that were missing default precision specifiers have been fixed.
- All apps have been upgraded to build to at least SDK8 (Froyo) as a minimum.
- A possible infinite loop in pause/resume event handling in `NvEvent` has been fixed.
- Expanded Android lifecycle documentation and `NvEvent` documentation.

19 November, 2010 Release

- nv_event now supports pause/resume. Any new apps made using the app_create script will inherit this support. Older applications wishing to update should set supportPauseResume equal to true in the constructor of their activity class , add an event handler for NV_EVENT_PAUSE and NV_EVENT_RESUME, and slightly alter the strategy for waiting for events. Use app_create to make a dummy project in order to see the changes on the C++ side. Apps may want to remove the orientation property in their manifest as well. NOTE: when device rotation occur, FOV must be adjusted to maintain proper object scale. The event_accelerometer sample shows how to do this when using "field of view" values in event_accelerometer/jni/render.cpp.
- A new accelerometer app, named event_accelerometer, was created to show how to process accelerometer data with respect to device orientation and how to use that data to orient an object.

10 June, 2010 Release

- Verified the code with the latest Android NDK (R4) and SDK (R6) and ADT plugin (0.97).
- Fix to the multi-touch handling in `nv_event`. Previous system worked correctly for single-touch, but multi-touch was being reported erratically.

27 May, 2010 Release

- Added the `nv_event` system, related `app_create` tool and the `event_lorenz` sample.
- Expanded all support libraries docs.

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com