NVIDIA Khronos Apps SDK

Version 100

Contents

Introduction

The NVIDIA Tegra Khronos Apps SDK ("Apps SDK") is a high-level set of demos and samples that demonstrate some of the multimedia features of the Tegra platform as exposed by the Khronos APIs:

- OpenGL ES 2.0
- OpenGL ES 1.x
- OpenMAX IL
- OpenKODE Core
- EGL

These samples are provided as source code, along with the required art assets and a set of support libraries. Build files for the supported platforms are also supplied.

Owing to the cross-platform nature of the Khronos APIs, several platforms are (or soon will be) supported by this SDK:

- Tegra 250 devkit hardware running the Windows CE 6.0 OS
- NVIDIA Windows PC-based Khronos Emulator
- Tegra 250 devkit hardware running Linux (***coming soon!***)

Each of these platforms has different requirements for development PC hardware and installed software, which will be detailed in coming sections.

> **Note:** Developers who have not yet received their Tegra 250 devkit hardware can still use this Apps SDK to familiarize themselves with the Khronos APIs owing to the SDK's support of the PC-based emulator. This can be an easy way to begin basic application development without the need to run on a Tegra devkit immediately.

Installing the SDK Software

The prerequisites and install process differs by platform. The following sections will cover each supported platform.

Note that the windows-based platforms (the Windows PC emulator and the Tegra running Windows CE 6.0) are supported by a single, shared SDK installer (`tegra_khronos_apps_sdk_<version>.msi`) and tree, and thus it is assumed that most developers developing for the Tegra devkit with Windows CE 6.0 will ***also*** have installed support for the Windows PC emulator. However, developers ***without*** Tegra 250 devkit hardware can still compile and run the samples and demos in this SDK on the Windows PC

emulator, adding support for the Tegra devkit running Windows CE 6.0 at a later date when they procure devkit hardware.

Windows PC Emulator

Prerequisites

Using the Apps SDK on the Windows PC-based emulator requires that the "X86 Windows OpenGL ES 2.0 emulator support pack" (`win_x86_es2emu_<version>.msi`) and its documentation (`win_x86_es2emu_<version>.pdf`) be downloaded and successfully installed on the development PC. The latest version of this pack is available from http://developer.nvidia.com/tegra

The included documentation lists the prerequisites for using that platform support pack on a particular development system.

Installing the Windows-based SDK

Once the platform support pack for the emulator is installed on the development PC, it is possible to install and compile/use the SDK and its demos on the PC. Simply download the Tegra Khronos Apps SDK Windows-based installer (`tegra_khronos_apps_sdk_<version>.msi`) from the SDK pages at http://developer.nvidia.com/tegra, run it on the development PC, and follow the installer's on-screen instructions. Note that once installed, this same SDK tree can support both the PC-based emulator and the Tegra devkit running Windows CE 6.0 (with the correct platform support pack installed as described in the next section).

Tegra Devkit Running Windows CE 6.0

Using the Apps SDK with the Tegra devkit running Windows CE 6.0 requires that the "CE6 Tegra 250 support pack" (`ce6_tegra_250_<version>.msi`) and its documentation (`ce6_tegra_250_<version>.pdf`) be downloaded and successfully installed on the development PC. The latest version of this pack is available from http://developer.nvidia.com/tegra

The included documentation lists the prerequisites for using that platform support pack on a particular development system.

Installing the Windows-based SDK

If not already downloaded and installed previously (e.g. when setting up to use the PC-based emulator), install and compile/use the SDK and its demos on the PC. Simply download the

Tegra Khronos Apps SDK Windows-based installer (`tegra_khronos_apps_sdk_<version>.msi`) from the SDK pages at http://developer.nvidia.com/tegra, run it on the development PC, and follow the installer's on-screen instructions.

Tegra Devkit Running Linux

Coming soon!

Additional Documentation

In addition to this documentation, there are other documents available on the Tegra Developer Zone (http://developer.nvidia.com/tegra) that are of interest to users of this Apps SDK:

- Developers using the SDK on the PC-based emulator should consult the documentation for that platform support pack (`win_x86_es2emu_<version>.pdf`), as it documents differences between the emulator and the Tegra platform itself w.r.t. the Khronos APIs
- Developers using the SDK on the Tegra 250 devkit hardware should consult the documentation for the platform support pack they are using (e.g. `ce6_tegra_250_<version>.pdf`), as it will document known issues on that pack's OS image.
- OS platform-independent documentation on programming for the Tegra using the Khronos APIs can also be downloaded from the Tegra Developer Zone, such as the Tegra OpenGL ES 2.0 Programming Guide (`tegra_gles2_development.pdf`).
- General Khronos documentation may be found at the Khronos website (http://www.khronos.org/)

SDK Tree Overview

This section gives a brief overview of major directories and projects in the SDK.

\3rdparty

This is the folder where all non-NVIDIA, third-party libraries reside. These currently include only the `libjpeg` library.

\inc

Contains the header files within subfolders for each third-party library.

\lib

Contains the generated link-time libraries for each third-party library.

\src

Contains full source trees for each third-party library.

\build (Windows and CE 6.0 only)

build_sdk.sln

This is a Visual Studio 2005 solution file which is the 'master' solution for building all elements of the SDK, including the simple *samples*, the more interactive *demos*, and all of the helper libraries that are used in various cases. Each of the samples, demos, and libraries are listed with basic details in this section, with more in-depth coverage in later sections.

glslf.bat

This is a helper batch file for compiling a single fragment shader text file into Tegra-compatible binary format, via the Tegra-specific Cg compiler and the shader-fixup executable. It is primarily designed for advanced users building custom build scripts and toolchains.

glslv.bat

This is a helper batch file for compiling a single vertex shader text file into a Tegra-compatible binary format, via the Tegra-specific Cg compiler and the shader-fixup executable. It is primarily designed for advanced users building custom build scripts and toolchains.

UpdateBinaryShaders.bat

This helper batch file is used to run a post-processing pass on all text-based shader files in the media directory, first compiling them to binary format using the Cg compiler, and then doing additional adjustments to the binary shaders specific for running binary shaders on the Tegra device. It is generally run just before preparing to copy content from the SDK to a staging directory or SD card, or otherwise in preparation for running applications on the Tegra devkit itself.

\demos

This folder contains the source code for the main interactive demo applications, all of which are built on top of the `nv_main` application framework and leverage OpenKODE Core and OpenGL ES 2.0. Most also use the `nv_ui` library for 2D user interface elements and support *gesture* input events generated into the OpenKODE event queue by `nv_main` and the `nv_gesture` library. While most of the demos will present a user interface on the PC-based emulator, as noted previously the PC-based emulator does not have OpenMAX support, and thus any related functionality will be absent.

addrbook

This demo leverages the `nv_ui` library to create an interactive address-book-like scrolling interface. It specifically creates a subclass of `NvUIScrollPane` to manage the 'physics' of a touch-scrollable list interface with a longer-than-the-screen list of rows/cells, and to simplify the handling of gesture inputs including taps, doubletaps, and side-swipes of list cells, and dragging and flicking the list. It uses the UI framework for everything rendered on screen, including backgrounds, bars, buttons, characters, and strings.

photoviewer

This demo implements the basic feature set for a photo browser application. It live-scans for JPEGs in a `data` subfolder, loads and creates custom thumbnail textures (using a combination of

the nv_jpeg and nv_hhdds libraries), and starts up by displaying a thumbnail 'overview' mode (with custom code for layout/rendering, use of the nv_gesture library for all interactions with the list including taps, drags, and flicks). You can jump from a thumbnail into 'fullscreen' single-photo mode, which allows you to use drag/flick gestures to flip one at a time through the photo library, as well as self-hiding onscreen controls for navigation and starting/stopping a timed slideshow mode.

Photoviewer also has an example 'transition' system for implementing specialized effects using custom geometry, shaders, and code to design a progressive change between two visuals.

videoplayer

This demo implements the basics of a portable video player. It's primary interface is a 3D 'spinner' interface for navigating DVD-style cover art, based on the nv_tileflow library which supports gesture-based navigation (handled under the covers in connection with the nv_gesture library), that allows the user to drag/flick through a library of videos. Selecting a video switches into fullscreen playback of the associated video file using OpenMAX IL.

visu

This is a standalone sample showing off a music visualization effect using a custom OpenMAX IL component to grab live sample data and visualize it via custom OpenGL ES shaders.

\docs

Contains additional SDK documentation beyond this main document.

\licenses

Contains the license documents for third party code and content contained in the SDK.

tegra_sdk_libs.chm

This is a Microsoft Help file with information extracted from some of the SDK helper libraries, including nv_bitfont, nv_config, nv_hhdds, nv_shader, and nv_ui. For each header/library, the documentation generally covers a general description of the feature set, all of the public classes, methods, and functions and their basic use, common enums and defines, special structures, and the like. The code samples that take advantage of the libraries are in most cases

the best 'documentation' of actual use, but this supplementary information should help fill in the gaps.

\libs

This folder contains the NVIDIA Helper Libraries (described elsewhere), most with full source code.

\inc

Contains the header files within subfolders for each helper library.

\lib

Contains the generated link-time libraries for each helper library.

\src

Contains subdirectories with the source code and project files for building the helper libraries.

\make (Linux only)

Contains the shared build-definition makefiles (`*.mk`) for building on the Linux platform.

\media

This folder contains two different sets of files. The first set is the runtime content (including configuration files, textures/images, 3D object data, GLSL-E shaders, etc.) used by sample and demo applications.

The second set is the applications/executables themselves as built from Visual Studio. The root of this folder is where PC emulator (`Win32` platform) sample and demo executables will be generated, in order to enable simple execution and debugging from Visual Studio.

\data

This is the 'read-write' contents folder, with subfolders for any individual OpenKODE applications that require them.

\resources

This is the 'read-only' contents folder, with subfolders for each individual OpenKODE application that has any specific content. It also contains `nvmain.txt`, the master configuration file for controlling certain runtime features of the `nv_main` framework.

\bin\wince\nvap

This is where all sample/demo executables built from Visual Studio, targeting the Tegra CE 6.0 platform (`STANDARDSDK_500` platform), will be output by default.

\samples

\opengles1x

> **Note:** OpenGL ES 1.x is not implemented on the PC-based emulator, and thus samples will not run on a PC (even though they may compile and link due to stub libraries).

kodecube

This is a standalone (non-`nv_main`) OpenGL ES 1.x sample that shows proper EGL setup for ES 1.x support, and the creation and animated rendering of a texture cube.

\opengles20

These demos were all designed to specifically show how to get certain features up and running under OpenGL ES 2.0. All of them sit on the `nv_main` shared framework, and thus will have similar 'flow' of execution, and benefit from the framework's built-in features.

basic

This sample draws a simple colored triangle, spinning in a circle. It is a great starting example of implementing on top of `nv_main`. It also shows use of the `nv_shader` helper library for loading shader programs and setting up vertex attribute bindings.

cpp

(Also known as "demo") This is the `basic` demo but rewritten and compiled as a C++ program. It illustrates the fact that all `nv_main`-related functions and variables need to be designated as extern "C" for proper linking with the library.

fbo

This sample extends off of `basic`, and shows the preferred method for offscreen-rendering under ES 2.0. It shows how to set up a framebuffer object (FBO) render target to allow dynamically-rendered graphics to serve as a texture. Specifically, it sets the FBO as the rendering target and draws a spinning triangle, switches back to the primary rendering target, binds the FBO's associated texture, and renders a spinning quad to the screen, thus showing the FBO contents.

font

Another `nv_main`-based sample that shows how to use the `nv_bitfont` library to render text. It demonstrates system initialization and global state setting, creation of string-reference objects, setting various properties on those objects, and rendering those strings out to an OpenGL ES context. It shows many features of `nv_bitfont` including using multiple font textures, the use of 'split fonts' to render normal and bold styles from one font texture, text output alignment, multi-line text handling, colored text, and drop-shadows.

frag_bench

A sample that makes it easy to try different fragment shaders rendered to the full screen with an adjustable degree of overdraw. The sample includes a convenient config file that makes it possible to set the uniforms and even the samplers available to the fragment shader without having to modify or recompile the sample code. The sample is designed to make it easy for developers to benchmark fill-heavy shaders and how changes to these shaders can affect performance on the Tegra device. The sample prints a running frame rate and fill rate value sampled over the course of several seconds.

glpath_basic

The `glpath_basic` sample demonstrates how to work with the new `GL_NV_draw_path` extension (*which is only available on Tegra*). It shows the following key features:

- The overall draw_path rendering process, including the NVIDIA logo as three paths, and a star-shaped path used as a mask.
- Masking applied via render-to-texture of the mask shape, then a fragment shader for the masked shapes using a '`discard`' statement to knock out fragments based on the alpha value of the mask.
- One approach to antialiasing, using supersampling.

glpath_font

The glpath_font sample demonstrates how to use the GL_NV_draw_path extension to render high-quality text. Feature set is similar to glpath_basic.

glpath_gradient

The glpath_gradient sample demonstrates basic use of GL_NV_draw_path extension (*again, only available on Tegra*), along with an efficient implementation of linear and radial gradients (via a fragment shader using a one-dimensional texture) in a simple helper library. Note that while this sample does not show the path supersampling used in the glpath_basic sample, the gradients show the benefit of hardware trilinear sampling.

gpu_mesh

A sample showing the power of the Tegra platform to do per-vertex animations inside of its vertex hardware unit via OpenGL ES 2.0's GLSL-ES. This sample renders an animated "sheet-like" mesh to which any desired texture can be applied.

kd_egl_simple

This sample shows the most basic OpenKODE Core + EGL + OpenGL ES 2.0 application, which is dependent upon neither the SDK helper libraries nor nv_main.

mars

This nv_main-based OpenGL ES sample shows more complex data loading and rendering, having multiple 3D objects, each individually textured and animated, and a shader that does lighting calculations. It leverages the nv_file library for setting up *file search paths* and opening data files that may live in varying locations within those folders. It also utilizes the nv_hhdds library for loading textures stored in DDS files (commonly used for DXT-compressed textures), and the nv_math library for working with 3D matrices.

native_render (PC-based emulator and CE 6.0 only)

This is an nv_main-wrapped sample that shows one method for bridging Windows native graphics and OpenGL ES. It creates a Windows HBITMAP (using CreateDIBSection) based off the EGL-created native window, shows two different ways to 'draw' to the bitmap (raw poking of pixels in the bitmap memory versus using GDI to render colored circles), and then renders the contents of the bitmap by uploading it to a pre-allocated texture and mapping the texture onto a full-screen quad.

particles

A sample showing how the Tegra platform's vertex shading unit can be used to create compelling particle animations without any per-vertex, per-frame CPU load. The example animates the particle positions entirely in the vertex hardware and uses OpenGL ES 2.0 point primitives to minimize the number of vertices that need to be processed to one per particle.

toon

This `nv_main`-based sample shows the use of more advanced GLSL shaders to do 'toon' style visualization of a 3D object. It also leverages the `nv_file` library for open files in various locations, the `nv_math` library for manipulation of 3D matrices, the `nv_bitfont` library for onscreen textual status, and the `nv_gesture` library and OpenKODE for handling tap, drag, and flick events.

win_egl_simple

This sample shows how to use Win32 HWNDs with EGL (on both the Windows PC-based emulator and Tegra CE 6.0 platforms), as well as how to use native Win32 without OpenKODE Core for applications, including how pointer/mouse events can be used.

\openkodecore

gesture_input

This is an `nv_main`-based sample that introduce using the `nv_gesture` library for all finger/mouse input, the gesture-based events it generates into the OpenKODE event queue, and how to handle them. Specifically, it covers press, release, hold, drag, tap, double tap, flick, and multi-zoom events, as they are generated. It also uses `nv_bitfont` to render status text indicating each gesture event generated.

pointer_input

This is another `nv_main`-based sample that shows how to manage OpenKODE 'pointer' (mouse, touchscreen, etc.) input events. It caches information from the input event, and during rendering it draws a square at the last pointer location, if the pointer was in 'selected' (down) state.

\openmax

> **Note:** OpenMAX IL is not implemented on the PC-based emulator, and thus samples will not run on PC (even though they may compile and link due to stub libraries).

audiocapture

This is a standalone (non-`nv_main`) OpenMAX IL sample that shows how to capture audio data.

componentinfo

This is another standalone sample that shows how to probe which components are available and able to provide a specific role in a component graph.

customcomponent

This is another standalone sample illustrating custom component development. NVIDIA provides a backbone for custom component creation to make the process as easy as possible for component writers. This backbone manages buffers, tunneling and other basic OpenMAX IL operations in the most efficient way so that developers can concentrate on their custom component's processing of data between its in port(s) and out port(s).

jpegencode

A standalone sample that renders a single frame to a framebuffer object (mapped to an EGLImage), which is then encoded multiple times in succession to a set of JPEG image files by the JPEG-encoding hardware in Tegra.

mpegencode

A standalone sample that renders a simple OpenGL ES 2.0 animation (moving rectangle) to a framebuffer object (mapped to an EGLImage), which is then encoded frame-by-frame into an MPEG file by the MPEG-encoding hardware in Tegra.

simplemp3playback

This is another standalone OpenMAX sample that shows how to set up a graph to play an audio file.

simpleplayback

This is another standalone OpenMAX sample that shows how to set up a graph to play a video file.

\openvg

> **Note:** OpenVG is not implemented on the PC-based emulator, and thus samples will not run on PC (even though they may compile and link due to stub libraries).

openvg_helloworld

This is a standalone (non-`nv_main`) OpenVG sample that shows proper EGL setup for OpenVG use, the creation of VG paint objects (solid colors and gradients) and VG paths (the NVIDIA logo, in a few parts), and animated rendering in OpenVG using paints and paths as well as 2D transformations.

Building the SDK

Following section describes the procedure to build the components of the Khronos Apps SDK

PC-based Khronos Emulator

- To build the entire SDK (helper libraries and apps) for the PC-based emulator, open `build_sdk.sln` in VS2005. You can also use VS2008 (projects will be auto converted to VS2008 format).
- Choose Win32 platform from the drop down menu.
- Choose release or debug configuration to build from the drop down menu.
- Build the entire solution.
- The binaries will be placed in `tegra_khr_apps_sdk\media` folder.
- The PC-based OpenGL ES 2.0 emulator is based upon the desktop GLSL shader compiler. The demos built for the emulator can only load GLSL-ES source-based shaders and compile them on at run time. Consequently, there is no need to compile shaders offline to run applications on this platform.

Tegra Windows CE 6.0

Building Libraries and Applications

- You can build the SDK for Tegra WinCE 6.0 using the same solution file as for the PC-based emulator.
- Choose STANDARDSDK_500 (ARMV4I) platform from the drop down menu.
- Choose release or debug configuration to build from the drop down menu.
- Build the entire solution.
- The binaries will be placed in `tegra_khr_apps_sdk\media\bin\wince\nvap` folder.

Pre-compiling Shaders

The Tegra is a mobile platform, and to lower the memory footprint of the driver, it can support pre-compiled shaders, built using the shader compiler tools that are shipped with this SDK. ***This is not a required step***, as the Tegra OpenGL ES 2.0 driver can compiled shaders from source code at runtime, but pre-compiling shaders can save runtime memory and CPU cycles.

- The source vertex shader files should have extension `*.vert`, `*.glslv` or `*.vs`. The source fragment shader files should have extension `*.frag`, `*.glslf` or `*.fs`. Generally, the output of this process will be `*.nvbv` and `*.nvbf` binary vertex and fragment shaders respectively.
- To compile all shaders in `tegra_khr_apps_sdk\media\[data, resources]` path, use `UpdateBinaryShaders.bat` in `tegra_khr_apps_sdk\build`. The script can be launched from a command line or by double-clicking. It compiles the shader files based on the file extension and outputs the compiled shader file in the same path with the same name but different extension (`nvbv` or `nvbf`, as appropriate).
- To compile shaders individually or to manually script the process, you can use the batch files `glslf.bat` for fragment shaders and `glslv.bat` for vertex shaders. Both are located in `tegra_khr_apps_sdk\build`. The first argument (required) is the path/name of the source shader to be compiled. The second argument is optional and is the path/name of the output binary shader. If not specified, the output file path/name is the same as the source path/name, with the source filename extension replaced with `nvbv` or `nvbf`, as appropriate.

Linux

Coming soon!

Installing and Running the SDK Samples and Demos

PC-based Khronos Emulator (Win32)

On the PC-based emulator, the applications are automatically built into the tree `tegra_khr_apps_sdk/media`, and are thus immediately ready to run, assuming that the environment has been set up correctly to add the DLL paths directed in the install guide chapters. Thus, the emulator versions of the samples and demos can be run in one of two ways:

- Launch the application from Visual Studio as you would any other project.
- Navigate to `tegra_khr_apps_sdk/media` and double click any demo executable.

The demos should be able to locate their textures, configuration files and shaders automatically.

Tegra Windows CE 6.0

Debugging the samples and demos on the Tegra devkit requires several steps initially:

1. Building the application (and binary shaders, if used) for the Tegra
2. Getting the media/content/shaders onto the devkit filesystem
3. Setting MSVC++ to deploy the applications to the same directory as the app media
4. Deploying and debugging the app

Verifying the Compiled Executables and Shaders

The previous sections in this document describe how to build the demos and (if desired) the pre-compiled shaders for the Tegra platform. Build the demos (release build) and (if desired/required) the shaders for the Tegra Windows CE 6.0 platform and then check for:

- The resulting executables in
  `tegra_khr_apps_sdk/media/bin/wince/nvap/`
- The resulting `*.nvbv` and `*.nvbf` binary shaders in the `shaders` subdirectory of each demo in
  `tegra_khr_apps_sdk/media/resources`.

If the desired executables and shaders are found in the SDK tree on the development PC, then the build stages were successful.

Getting Content onto the Devkit

While MSVC++ will automatically deploy the executable for a given demo over ActiveSync to the device, to avoid long deployment stages, we do not use MSVC++ deployment to deploy art assets and shaders for each demo. Those are assumed to be accessible on the device before a debugging session is started.

Selecting the Content Location

There are three common ways to store the media on the devkit's file system:

SD card: Content is copied to a FAT32-formatted SD card on the development PC and then inserted into the SD card slot on the devkit (located between the mini-USB and Ethernet jacks). The default mounted name of an SD card on the devkit is "\Storage Card", unless a root directory with this name already exists. In the case of an existing "\Storage Card" directory on the devkit, the SD card will mount to "\Storage Card2".

USB drive: Content is copied to a FAT32-formatted USB drive on the development PC and then inserted into a powered USB hub connected to the devkit (the use of a powered hub is recommend unless the drive is low-power). The default mounted name of a USB drive on the devkit is "\Hard Disk", unless a root directory with this name already exists. In the case of an existing "\Hard Disk" directory on the devkit, the drive will mount to "\Hard Disk2".

Device-local storage: Because it is not physically removable, in order to copy content to the device's local storage (sometimes called "NAND Flash"), the devkit must be connected to the development PC via USB (ActiveSync or WMDC, as described in the "Viewing/Transferring Files onto the Devkit" section of the Tegra CE 6.0 Platform Support Pack documentation). Content can be copied into a directory created in the root of the device file system ("My Device").

Note that depending on where the content is mounted, the deployment of the content and some of the configuration files for the demos may need to be changed.

Copying the Content to the Chosen Location

Having selected SD card, USB drive, or a named directory on the devkit itself, the content can be copied to the target. We will assume that the target card, drive or devkit directory is `<target>`.

1. Copy the `data` and `resources` trees (including those directories, not just the contents thereof):

```
tegra_khr_apps_sdk/media/data/
tegra_khr_apps_sdk/media/resources/
```

   into the desired directory, giving

```
<target>/data
<target>/resources
```

2. There *may* be additional 3rd-party DLLs required by some demos, depending on the SDK release. If the following path exists in the SDK tree

```
tegra_khr_apps_sdk/3rdparty/bin/wince/nvap/release/
```

   Then copy the contents of it into

```
<target>/
```

The Tegra OpenKODE implementation bases its paths relative to the location of the current application. This generally means that pure OpenKODE applications can have their executable and data trees anywhere on the file system of a device, so long as the relative location of executable to data tree is fixed.

However, the OpenMAX based demos in this SDK use native paths to reference files. On Windows CE, all paths are absolute. As a result, in order to function correctly "out of the box", the configuration files for several of the OpenMAX demos must be changed if the `resources/data` trees are not accessible on the devkit in the directory "\Storage Card". Note that placing the data in the "\Storage Card" directory is possible in two cases:

1) Using an SD card with `resources` and `data` in the root of the SD card

2) Using devkit-local storage, manually creating the directory "\Storage Card" on the devkit. (There must be no SD card inserted in the devkit)

> **Note:** As mentioned, manually creating a "\Storage Card" directory on the devkit will cause any SD card that is later inserted to be mounted as "\Storage Card2" unless the manually-created "\Storage Card" directory is first deleted!

If the selected <target> will not appear on the devkit as "\Storage Card", then the following configuration files or source files will need to be edited (and if the changes are to source files, to recompile the demo) to change "/Storage Card" to the chosen mount point (e.g. "/Hard Drive" or "/demos"):

Resources:

```
resources\videoplayer\config.txt
```

Source Files (note that in most cases below, either the source file can be modified or a command-line argument can specify the path to the file):

```
tegra_khr_apps_sdk\samples\openmax\src\audiocapture\main.c
tegra_khr_apps_sdk\samples\openmax\src\customcomponent\main.c
tegra_khr_apps_sdk\samples\openmax\src\jpegencode\main.c
tegra_khr_apps_sdk\samples\openmax\src\simplemp3playback\main.c
tegra_khr_apps_sdk\samples\openmax\src\simpleplayback\main.c
```

Setting MSVC++ to Deploy Correctly

As mentioned previously, the OpenKODE application executables must be deployed to the mount point of <target> on the devkit, so that the .exe files are in the same location as the data and resources directories. To make this easier, the Apps SDK's solution files are set to deploy the applications to a directory specified by the system environment variable `$(NV_KHR_SDK_REMOTE_DIR)`, which the SDK installer defaults to "\Storage Card". If the <target> selected in the previous steps is not "\Storage Card" (e.g. a USB drive was used and the <target> is "\Hard Disk"), then follow these steps:

1) Exit any instances of MSVC++ (which only re-reads the environment at launch)
2) Change the value of system environment variable `$(NV_KHR_SDK_REMOTE_DIR)` to the mounting oath of <target> on the devkit
3) Restart MSVC++

Now, if you select "debug" or "deploy" for any of the projects in `build_sdk.sln`, the executable should be deployed to the desired location on the device. Verify this using remote file viewer or File Explorer on the devkit before proceeding.

Controls

Common Controls

The following controls are common to all `nv_main`-based application, which includes all sample applications and demos unless otherwise noted in the application's specific control listings.

Exiting the Demos

On all platforms, the CTRL-Q key combination will exit `nv_main`-based application.

On the PC-based emulator, all applications can be exited by closing the window with the normal Windows close button.

Applications that use another method for exiting will be listed explicitly in the following sections.

Sample Application Controls

Many of the sample applications are very simple; as a result they have extremely few controls. Often the controls consist of nothing but the `nv_main` defaults.

Basic

No controls beyond the defaults afforded by `nv_main`.

CPP

No controls beyond the defaults afforded by `nv_main`.

FBO

No controls beyond the defaults afforded by `nv_main`.

Font

There are numerous additional keyboard controls afforded by the demo:

- **ENTER:** Change the text for the bouncing string
- **Up/Down Arrows:** Move the shadow offsets
- **Left/Right Arrows:** Make the multiline text box narrower/wider

Frag bench

No controls beyond the defaults afforded by `nv_main`.

GLPath Basic

There are numerous additional controls afforded by the demo:

- **Mouse Click:** Cycle rendering between unmasked, static/cached masked, and dynamic masked
- **M Key:** Toggle z-based masking off and on
- **S Key:** Toggle supersampled-AA off and on

GLPath Font

- **Mouse Click:** Cycle rendering between unmasked, static/cached masked, and dynamic masked
- **A Key:** Pause/restart animation
- **C Key:** Toggle coverage-sampled AA off and on
- **S Key:** Toggle supersampled-AA off and on

GLPath Gradient

No controls beyond the defaults afforded by `nv_main`.

GPU Mesh

Tapping on the screen or pressing the ENTER key will make the mesh transition from flat, fullscreen to scaled-down and animated in a waving motion. Tapping (or pressing ENTER) again will return the mesh to flat, fullscreen.

KD EGL Simple

This is not an `nv_main` application, so the `nv_main` controls are not supported. Click the mouse to exit.

Mars

No controls beyond the defaults afforded by `nv_main`.

Native Render

Dragging the mouse pointer will draw one of two types of trails: 'pixel' pokes, or 'GDI' circles. It starts in 'pixel' mode.

Particles

No controls beyond the defaults afforded by `nv_main`.

Toon

Clicking the main mouse button or pressing the ENTER key will cause the shader to cycle from toon shading with outline, to normal smooth shading with outline, to just normal smooth

shading. If you press and drag across the width of the screen, you can interactively control the rotation of the object. If you do a 'flick' gesture across the width of the screen, you will set the direction and speed of spinning.

Win EGL Simple

This is not an `nv_main` application, so the `nv_main` controls are not supported. Click the mouse to exit.

Kode Cube

Kode Cube is not an `nv_main` application. Any tap on the touchscreen or button press will exit the application.

Pointer Input

Pointer Input draws a colored quad to the screen directly under the current position of the mouse pointer.

Gesture Input

Gesture Input draws 'hourglass' markers in response to input, but specifically tied into gesture events. It shows a red marker for the current input location, but additionally shows a blue marker for the start location of a drag or flick. As each 'transition' of gesture state occurs, a scrolling text block at the right edge is updated with the gesture event name and UID, to help better understand the gesture system and events. Also, at the bottom of the screen is an output showing the coordinates of the start of the gesture, plus a second value indicating delta for drag and velocity for flick.

Audio Capture

Audio Capture has no controls. It exits automatically after grabbing ten seconds of audio. It is not an `nv_main` application. It accepts an optional command-line argument: the name of the audio file to write. Note that this path is a native WinCE device path, not an OpenKODE path.

Componentinfo

Componentinfo has no controls. It exits automatically after printing the supported OpenMAX IL components.

Customcomponent

Customcomponent has no controls. It exits automatically after playing back ten seconds of the provided audio file at half-speed.

Jpegencode

Jpegencode has no controls. It exits automatically after encoding the rendered image to 30 JPEG files.

mpegencode

Mpegencode has no controls. It exits automatically after encoding the animation sequence to an MPEG file.

Simplemp3playback

Simplemp3playback has no controls. It exits automatically after playing ten seconds of the supplied MP3 file.

simpleplayback

Simpleplayback has no controls. It exits automatically after playing ten seconds of the supplied video file.

Demo Application Controls

Address Book

The controls for the Address Book / gesture demo are as follows:

- Dragging up and down with the mouse scrolls the list up and down.
- Flicking up and down with the mouse will slide the list automatically.
- Clicking the mouse while the list is sliding will stop it immediately.
- Clicking on a name changes its color
- Double-clicking on a name changes it another color
- Swiping across a name (drag horizontally across it) changes to a third color

Photo Viewer

In Wall/Thumbnail mode, the controls are:

- Dragging or flicking left or right with the mouse scrolls the thumbnail set left and right
- Double-clicking on a thumbnail will zoom in on that thumbnail
- Double-clicking when zoomed in will zoom back out
- Clicking on a thumbnail will either open up the folder (for a folder thumbnail), or enter fullscreen mode (for a photo thumbnail)
- Clicking the Back button will go up to a previous level if inside a folder
- The ESC key goes back one level

In Fullscreen, single-photo mode, the controls are:

- Dragging more than a quarter of the screen left or right switches to the next/previous photo in the set.
- Flicking left or right to quickly switches to the next/previous photo.
- Double-clicking while in fullscreen mode zooms in on the thumbnail.
- Click in fullscreen mode when the control bar is hidden will slide it back up into position.
- The right/left buttons on control bar switch to the next/previous photo.
- Click the play icon to start slideshow mode, which automatically plays through photos in the current set.
- Clicking the pause icon (which replaces Play when in slideshow mode) will stop the slideshow.
- Dragging or Flicking during slideshow mode will stop the slideshow.
- Clicking the Back button will go back to the Wall/Thumbnail mode
- Pressing the left and right arrow keys will skip forward and backwards through the set
- Pressing the Enter key will leave single-photo mode
- Pressing the Space bar will pause or restart the slideshow
- The ESC key returns to wall/thumbnail mode

Video Player

In video list "spinner" mode, the controls are:

- Dragging/flicking the mouse left or right spins through the collection of video cover art.
- The left and right arrow keys spin through the collection of video cover art.
- Double-clicking a video cover will start video playback of that title.
- The Enter key will start video playback of the centered title.

In video playback mode, the controls are:

- Clicking the mouse will stop playback and return to the video cover art spinner.
- If the video mode is EGLImage, or if video is displayed on external HDMI display only, then a UI bar is shown with controls for pausing/playing, stopping, and seeking in the video.

The list of videos and associated cover images may be configured via the text files in:

```
/data/videoplayer/config*.txt
```

Note that video paths in these configuration files are native WinCE device paths, not OpenKODE paths.

Visu

No controls beyond the defaults afforded by `nv_main`.

An optional command line argument of

```
-music <filename>
```

Will use the desired MP3 file as the audio. The default music file is

```
/Storage Card/data/mediaplayer/music/nv_sample_mp3_320.mp3
```

Note that this path is a native WinCE device path, not an OpenKODE path.

Basic Application Development

Common Build Settings

Include Paths

`tegra_khr_apps_sdk/samples/shared/src` – If using nv_main

`tegra_khr_apps_sdk/libs/inc` – For our helper libraries

For the Tegra Platform:

`$(NV_WINCE_T2_PLAT)/include` – For GLES2/EGL etc headers

`$(NV_WINCE_T2_PLAT)/include/OpenMAX/il` – For OpenMAX IL headers

For the PC-based emulator:

`$(NV_WINGL_X86_PLAT)/include` – For GLES2/EGL etc headers

`$(NV_WINGL_X86_PLAT)/include/OpenMAX/il` – For OpenMAX IL headers

Library Paths

For the Tegra Platform:

`$(NV_WINCE_T2_PLAT)/lib/release` – Khronos libraries

`tegra_khr_apps_sdk/libs/lib/WINCE/NVAP/release` – Our helper libraries

For the PC-based emulator:

`$(NV_WINGL_X86_PLAT)/lib/release` – Khronos libraries

`tegra_khr_apps_sdk/libs/lib/winxp/x86/release` – Our helper libraries

> **Note:** The OS support packs only provide Khronos link libraries in the build release flavor, but for our helper libraries "release" in the pathname can optionally be changed to "debug" to link with the libraries in that flavor.

Preprocessor defines for nv_main based applications

NEED_CUSTOM_PRE_EGL_INIT

Define this if the application needs to perform some operations before EGL has been initialized. The function `KDint nv_pre_egl_init(void);` will have to be implemented by the application and is the function that nv_main calls before EGL is initialized.

USE_EGLTIMER

Define this to make the application use the extension `EGL_NV_perfmon` for all `nv_main` provided time stamps. Time stamps provided in KD events are not affected by this define. For further information on the `EGL_NV_perfmon` extension see the OpenGL ES 2.0 development / OpenGL ES extensions section of this document.

Avoiding QVGA windows on Windows Mobile

By default, running an application on a Windows Mobile operating system will show up as QVGA. To avoid this, these three files can be added to your project:

`tegra_khr_apps_sdk/samples/shared/src/nv_main/nv_main_win32resource.h`

`tegra_khr_apps_sdk/samples/shared/src/nv_main/nv_main_win32resource.rc`

`tegra_khr_apps_sdk/samples/shared/src/nv_main/nv_main_win32resource_ceux.bin`

> **Note:** Although located in the `nv_main` directory, using the rest of the `nv_main` functionality is not required when using these previously mentioned resource files.
>
> **Note:** For more information, search the internet for "HI_RES_AWARE"

Adding support for floating point instructions

The Microsoft Visual Studio 2008 compiler added support for targeting the vector floating point (vfp) unit on ARM processors via a command line option. If you are developing an application that uses heavy floating point operations, it is recommended that you enable this option.

To do so, choose the Project->Properties menu item, navigate to Configuration Properties->C/C++->Command Line, and add `/QRfpe-` in the Additional Options text box.

Building and Running on the PC-based emulator

Once you have set up your project, you are now ready to build and run it. When building for the PC-based emulator, make sure that the active platform is "Win32". This can be ensured by choosing the menu item *Build->Configuration Manager…*and selecting Win32 from the "Active solution platform" dropdown menu.

To build the solution, select "Build Solution" from the Build menu.

To run the application, select the desired application as the "Startup Project" by right-clicking on the project in the solution explorer and selecting "Set as Startup Project". Then, hit F5 on your keyboard.

> **Note:** Only OpenGL ES 2 and OpenKODE applications will be emulated on the PC-based emulator. While stub driver library files for Windows PC are provided for example for OpenMAX IL to make the applications compile and link, they will not provide any functionality other than setting an error.

Building, Deploying and Running on the Tegra Devkit

When building for the Tegra devkit, make sure that the active platform is "STANDARDSDK_500 (ARMV4I)". This can be ensured by choosing the menu item

Build->Configuration Manager…

and selecting STANDARDSDK_500 from the "Active solution platform" dropdown menu.

To build the solution, select "Build Solution" from the Build menu.

Running standalone

The Tegra 250 devkit supports SD cards and USB drives on which you can put your executable and resource files for easy exchange between your device and PC. If you are developing an OpenKODE application, it will expect the resources and data directories to be in the same

location as the executable, which means that if you put your executable in the root of the SD card or USB drive, then that's where the data and resources directories should go too.

Once the files are on the removable storage, you can launch your application by navigating to it through My Device (in "/Storage Card" for SD cards and "/Hard Disk" for USB drives) and double tapping it to launch.

Building the provided sample applications and libraries

When building the provided sample applications and helper libraries, please use the solution file:

```
tegra_khr_apps_sdk/build/build_sdk.sln
```

Building the solution twice might be necessary to solve all dependencies.

Tegra SDK Helper Libraries

The nv_main App Framework

While located with the sample code, `nv_main` is for all intents a 'helper library' of sorts. All of the OpenGL ES 2.0 applications in the SDK (including "samples" and "demos" subdirectories) are implemented on top of a simple application 'framework', based on the shared `nv_main.c` file (and its matching header). The nv_main framework handles most of the basics needed to get an OpenKODE Core application up and running, including a `kdMain` function -- the entry point for all Core apps – that implements the initialization and setup of an EGL window, surface, and context for OpenGL-ES rendering, runs the main OpenKODE event loop, dispatches individual OpenKODE events, and handles all normal application shutdown/cleanup.

Other optional features of `nv_main` include specifying use of an external display device, the ability to display live FPS data overlaid on your application's output, the ability to do 'benchmark runs' which execute your app for a specific number of frames or seconds and then automatically exits, and the ability for applications to change the timeout for the main loop's `kdWaitEvent` call (such as when they lose or gain focus).

> **Note:** All `nv_main`-based applications can be exited by one of two methods:
>
> **PC emulator-only:** click the "close" button on the title bar of the application window
>
> **All platforms:** Press CTRL-Q on the keyboard

Required Application Functions

The framework requires that each application implement four special-named functions, all with C linkage, in order for it to call into application-specific code. Let's look at each of these framework functions, and how they should be used.

```
KDint nv_init(void);
```

The application should use this function to set up any data loading, texture setup, etc. It is called after all of the EGL and OpenGL-ES setup have been done, so in the rare case you need to directly access the EGL data structures stored in the global `gEGLHandles`, they will all be valid. The application should return `EGL_TRUE` upon completing successful initialization, and `EGL_FALSE` if there is a failure.

```
KDint nv_handle_message(const KDEvent *);
```

This function is called with each Khronos event that is received by the app's window. The application must handle each event or pass it to `kdDefaultEvent`. The application should return `EGL_TRUE` to continue, or `EGL_FALSE` to cause the app to exit.

```
KDint nv_render(KDust drawTime, KDint drawWidth, KDint drawHeight,
              KDboolean forceRedraw);
```

The application should use the function to do their per-frame rendering. The app should not call `eglSwapBuffers` in this function. If needed, the `nv_main` code will do so. The application should return `EGL_TRUE` if a swap is required and `EGL_FALSE` if not (e.g. if the application did not need to redraw). The arguments are as follows:

`drawTime`: The current time in nanoseconds

`drawWidth`, `drawHeight`: the width and height of the drawing surface in pixels at the time of the draw call.

`forceRedraw`: `KD_TRUE` if the system requires the application to redraw.

```
KDint nv_cleanup(void);
```

The application should use this function to release all application data and resources. It is called before all of the EGL and OpenGL ES items have been destroyed. The application should return `EGL_TRUE` on success and `EGL_FALSE` on failure.

In addition, apps must define the following string in their code – it will be used as the window caption of the default window:

```
const KDchar* nv_window_caption;
```

The Main Loop

The framework implements a fairly simple startup, main loop, and shutdown process as follows (pseudo code):

```
kdMain
{
    // Calls nv_pre_egl_init, which does nothing by default.
    // An app can replace it with its own by defining the
    // #define NEED_CUSTOM_PRE_EGL_INIT, and then defining its
    // own nv_pre_egl_init function which would be called here.

    // egl_init() sets up the EGL display and config.

    // The main window is created via OpenKODE.
    // The size, caption, etc are set and the window is activated.

    // egl_window_init() sets up the windowed EGL surface and
    // OpenGL ES context.

    // The app's nv_init() is called to do custom initialization.

    while (!quit)
    {
        // Loop on the OpenKODE event queue, passing each event to the
        // app's nv_handle_message().  If it returns KD_FALSE,
        // exit the main app loop.
        // When the event queue is empty or the max total per-iteration
        // time has been exhausted, call the app's nv_render(),
        // and swap buffers if it returns a non-zero result.
    }

    // The app's nv_cleanup() is called to allow the app to delete
    // any OpenGL ES resources, or other custom data it created.

    // egl_window_cleanup() is called to destroy the OpenGL ES context
    // and the EGL surface.
    // The window is destroyed.

    // egl_cleanup() is called to shut down EGL.
}
```

(Emulated) OpenKODE ATX_keyboard Support

The `ATX_keyboard` extension is an OpenKODE Core extension to provide full keyboard support. Note that this extension is currently a draft spec, not yet approved. Some Tegra OS images (such as Linux images) support the extension directly. On Windows CE 6.0 and the Windows PC-based emulator, the extension is not currently supported in the OS itself. However, partial support for the extension has been added to the nv_main code to simulate it. Applications based on nv_main will receive `KD_EVENT_INPUT_KEY_ATX` on the Windows CE 6.0 and Windows PC-based emulator platforms. Applications using their own `kdMain` entry point

and not using nv_main will NOT receive these events on those Windows-based platforms (they will receive them on Linux), because nv_main implements the simulated extension. Note that the extension itself supports both `KD_EVENT_INPUT_KEY_ATX` and `KD_EVENT_INPUT_KEYCHAR_ATX` messages; the simulated extension only supports `KD_EVENT_INPUT_KEY_ATX` events. For details, see the function `NvAtxKeyboardSetup()`, which installs a "hooked" Windows event handler to capture key events and translate them. If needed, this code can be ported to other applications by the developer.

Additional nv_main functions

These functions and variables are declared and defined in `nv_main` and made available to applications:

```
void nv_wait_timeout(KDust waitTime);
```

This function can be used by an app to request a different `kdWaitEvent` timeout (depending on other settings, `nv_main` may decide to ignore the requested change..). Applications should, for example, use this to adjust their amount of CPU cycling when not focused. This is the maximum amount of time that the application will "sleep" between `nv_render` calls. If user input is pending, it will be sent immediately, independent of this setting. Applications that do not need to do animations, etc when no input is pending can set this value to larger times to avoid wasting CPU time.

```
KDint nv_swapinterval(KDint interval);
```

This function can be used by an app to request a particular swap interval/vsync option (depending on other state, nv_main may ignore the request..).

```
void nv_app_rotation(KDint rot);
```

This function can be used by an app to indicate to `nv_main`'s input handling that it is rotating itself into the render buffer (and thus input to it should be rotated)

```
void nv_app_designinput(KDboolean needsRescale);
```

This function can be used by an app to request that its input event coordinates (gestures and/or pointer) be rescaled into design space instead of EGL/window input space.

```
const KDEvent *NvMainRescaleGesture(const KDEvent *ev);
```

This function manually converts (rescales) a gesture event (could be others in the future) from EGL/window input coordinate system into design coordinates, returning the result in a temporary `KDEvent`. This function is ***automatically*** called on gesture events if `nv_app_designinput` has been set to `KD_TRUE`.

```
KDboolean NvMainClear(GLenum flags);
```

Apps should use this function to clear their window buffer, passing in the same flags they would pass to `glClear`. In addition, even if the app does not need to clear each frame, it should call this function with a flag of 0, as `nv_main` may have other operations that require an aux buffer to be cleared. The function returns `KD_TRUE` if a clear was done, or `KD_FALSE` if not.

```
void NvMainSwap(KDuint mode);
```
If an application needs to manually swap buffers (`nv_main` does it automatically at the end of a `KD_TRUE`-returning `nv_render` call), it should use this function. The parameter is a bitmask, and for the moment should be `0xFF` if you want HUDs, or 0 if you don't (the flags are not yet defined..)

```
void NvMainSetCSAAMask(GLboolean mask);
void NvMainSetCSAAOperation(GLenum operation);
```
Advanced applications that wish to adjust their CSAA modes per object manually should do so using this function. If CSAA is not active, then these functions will do nothing. Apps should, when possible, avoid calling the CSAA functions directly, but rather should call these versions, so state may be tracked

nv_bitfont

The `nv_bitfont` library is a fairly simple implementation of a bitmap-font renderer, which has been built up in features bit by bit. At its core, it loads up one or more bitmap font texture files (a DDS file containing a texture formed of an 8x8 grid of character glyphs) along with a '.abc' data file that contains the 'spacing' data for each character. Currently, all included fonts are formatted as A8 data for improved quality over compressed formats. The font files are located in `\media\resource\shared\fonts`, and include multiple sans-serif typefaces (generally at two resolutions), plus a 'console' font for monospaced output (also in two resolutions, but lower so that `nv_main` use of a console font is only a small overhead for the functionality it provides).

In terms of general approach, `nv_bitfont` manages a given 'string' you want to output in repeated frames (and thus want to 'cache' and allow it to be optimized) as a 'bftext' abstract structure. Each bftext has properties you can set on it, including of course a character string, which font/typeface to use, a pixel size, output alignment (vertical and horizontal), position in (generally) screen coordinates, a bounding box for alignment and wrapping, multi-line output settings (automatically word wraps text that exceeds the specified bounding box, for example), string base and multiplied colors, and drop-shadow options (offset and color). The character string is also parsed for certain 'escape codes' embedded in the non-visible character range, allowing for certain (predefined) color changes on-the-fly in one string, as well as the ability to have bold vs. normal style runs in a string (assuming the bitmap font was generated with support for 'bold' output).

The bftext objects are highly optimized for rendering speed, using vertex buffer objects to cache generated vertex data for rasterizing each character glyph (and shadows). Changing the

position or basic alignment of the string does not require any recalculation, and any other changes that do require data be recached will queue up until the string is next drawn, calculate and cache once, from then on rendering in optimized fashion.

In tying `nv_bitfont` in with the `nv_ui` library, two other abilities were added to the system. The first is that you can specify the output 'screen' resolution – however, this would be better stated as output 'buffer' resolution, as it can be used to properly match the size of an FBO for pre-rendering text into a texture for mapping onto say a 3D object. The second feature is that you can specify a rotation for the overall system, typically in 90-degree increments. Combining the two enables outputting text to any size and orientation target surface, allowing a 'portrait' application to basically rotate itself into a 'landscape' framebuffer, for example.

For a specific example that exercises pretty much every major feature of the `nv_bitfont` library, please refer to the `font` sample (`\samples\opengles20\font`).

For further details of functions and variables you can use, please refer to `nv_bitfont` information in the SDK Libraries help file (`\docs\tegra_sdk_libs.chm`).

nv_check

The nv_check library includes a set of simple functions for converting EGL, GL and OpenKODE Core error enumerants into strings for error logging. It also includes functions that can check for standing errors in EGL and GL, printing an error message with a provided prefix note if an error exists. It also includes "wrapper" code to make it easier to log GL ES FBO status (complete, error, etc) and to log a message if an expected shader uniform or attribute is not found in the currently bound shader program.

nv_config

The nv_config library implements support for block-based configuration text files. A simplistic example of a base configuration block would be something like:

```
"Options"
{
       // Here is a comment about our option that has two arguments
       Gadget 10 20
}
```

This defines a base block "Options", under which there is one entry "One", which has two additional parameters available. To access this, you would load your options file using `NvConfigCreateFromFile`, and then use a variety of helper functions to 'retrieve' a given sub-block or line within the config file. Some pseudo-code for the sample block above might be something like:

```
matchingLine = NvConfigFindMatchingLine(configFile, "Options", 0);
if (matchingLine >= 0)
{
    if (NvConfigFindDelimitedBlock(configFile, "{", "}", KD_TRUE,
            matchingLine, -1, &blockRange)
    {
        matchingLine = NvConfigFindMatchingLineInRange(&blockRange, "Gadget");
        if (matchingLine >= 0)
        {
            firstArg = NvConfigGetIntToken(configFile, matchingLine, 1);
            secondArg = NvConfigGetIntToken(configFile, matchingLine, 2);
        }
    }
}
```

To see more specific examples of using the `nv_config` library, you can look at the `nv_main` shared framework, or many of the demos like `videoplayer` and `frag_bench`, that use the library to process custom configuration files.

For further details of functions and variables exported for use, please refer to `nv_config` information in the SDK Libraries help file (`\docs\tegra_sdk_libs.chm`).

nv_file

The `nv_file` library serves to abstract the file-system specifics for the programmer, make it easier to work with files within the restrictions of OpenKODE, and make it easier to use content from different directories without needing to specify full paths. At the base level, an application calls the `NvFSAppendSearchPath` function, one or more times, to set up the potential sources for data files. The application can later call `NvFSOpen` (instead of `kdOpen`/`fopen`) to try to open a named file, and `nv_file` will search the specified paths to locate (and then open) the requested file. All the major file-access functions have `NvFS` wrappers/implementations, and they should be used instead of the 'native' file functions if you are using `NvFSOpen`.

nv_gesture

The `nv_gesture` library is used heavily by all of our demo applications and a few of the samples, as well as a few libraries, so there are many good references for how to interact with gestures.

Simply put, the `nv_gesture` library works by taking in finger/pointer/mouse input, sample by sample, tracking it over time, and running an internal state machine to determine what best defines what the user has done or is doing. The library includes a critical a helper function (`generate_gesture_event`) which is called by `nv_main` in response to user input if the `GENERATE_GESTURES` define is set in a given project file (currently set in source in `nv_main.c`).

The 'support' code in nv_main takes OpenKODE pointer input events, translates them into the structure that nv_gesture wants, and calls the helper function to do the real 'heavy lifting' of communicating with various internal functions of the library in order to track and report the current 'inferred gesture' state. The generate_gesture_event function will return a "gesture event", a first-class OpenKODE event object, if there is useful state to pass along. If it does, nv_main will posts that gesture event into the normal OpenKODE event queue, where it eventually makes its way into application-specific code (and then possibly is passed back 'out' into nv_ui, or nv_tileflow, or other libraries that manage specific user interaction). The gesture_input sample application is a great place to see 'live' what the gesture system is generating, and when.

If you want to see source code that handles gestures, you can look at the photoviewer demo (as it directly handles gestures for the most part), or the addrbook demo (which shows how to let the nv_ui framework take complete care of handling interactions). Overall, a number of the classes in the nv_ui framework are great examples of handling and responding to gesture input. For some specific uses, look at NvUIButton for basic press to release tracking, or look as NvUIScrollPane for drag and flick handling (in addition to tap, double-tap).

One other useful place to look is the main application-facing header file:

\libs\inc\nv_gesture\nv_gesture_event.h

It contains the NvGestureKind enumerants with all the types of gestures the system currently tracks and reports, as well as the KDEventInputGestureNv structure which is the custom event data that is attached to a given gesture event passed through the OpenKODE event system to an application.

nv_glesutil

The nv_glesutil library is a central source of commonly-used helper functions. Its major areas of functionality are loading of DDS image files (and hands-off uploading to OpenGL ES for you), writing out screen-shot images as BMP files, and the management of custom OpenGL ES 2.0 contexts.

The most-used feature is certainly the NvCreateTextureFromDDSEx function, which leverages the nv_hhdds library to load a DDS file from disk, then turns around and uploads it as a texture to OpenGL ES, and returns basic texture details to the caller including the generated OpenGL texture ID of the internally-allocated texture.

The library code and headers are the best source for details.

nv_hhdds

The `nv_hhdds` library supports loading of textures stored in a DDS file, ranging from DXT-compressed, to raw RGB, to more specialized formats like A8. It also supports loading of mipmap levels, or cubemap faces. The library also includes functions to let the developer compress RGB(A) texture data into any of the DXT formats on-the-fly, supports internal mipmap generation, and can also write out DDS-format files as well.

> **Note:** The `nv_hhdds` library was not designed for shipping products, rather for initial rapid porting or prototyping, and as such the compression quality is nowhere near that of other (larger) libraries, nor at the level of offline tools like plugins for Photoshop. However, `nv_hhdds` is small, and it sacrifices quality for performance, and is thus well suited to early stages of work on Tegra – especially if you need to load textures in an existing format, but want (or need) the memory savings of DXT compression for runtime.

Any of the OpenGL ES 2.0 samples or demos that loads textures generally does so by leveraging `nv_hhdds` directly or indirectly. In some cases, it is 'hidden use', provided under the covers by another library such as `nv_ui` for user interface elements, or targeted use of further helper functions in something like `nv_glesutil` for one-shot file-load and GL-upload.

For further details of functions and variables exported for use, please refer to `nv_hhdds` information in the SDK Libraries help file (`\docs\tegra_sdk_libs.chm`).

nv_jpeg

The `nv_jpeg` library serves to abstract the underlying methods used for JPEG decompression (which may be software or hardware based), and exports a very simple functional interface for loading a JPEG file from disk.

nv_math

The `nv_math` library is a set of simple helper functions for working with certain complex data structures and calculations. It has support functions for working with quaternions, vectors, and matrices, as well as managing and evaluating certain classes of higher-order curves. It also includes a very-useful function for calculating the seconds, in floating-point, between two `KDust` timestamps.

The `nv_math` code and headers are the best place to look for specific functionality, though many of the samples are also good sources of use-cases.

nv_math_cpp

The `nv_math_cpp` library is a C++ oriented helper library for working with things like vectors, quaternions, and matrices as C++ objects. It takes advantage of operator overloading in limited cases to simplify the handling of standard operations like matrix or vector multiplication.

The `nv_math_cpp` code and headers are the best source for specific information/details.

nv_mediaplayer

The `nv_mediaplayer` library provides a common interface for audio and video use cases, such as play/pause/stop, seeking, and changing the volume, all through a simple API. For users who just want to get up and running quickly with playing media files on the Tegra devkit, the `nv_mediaplayer` library header is worth a quick look – as is the `videoplayer` demo. For more advanced developers, the library source code serves as a reasonable demonstration of implementing certain media features using OpenMAX IL.

nv_mediawall

The nv_mediawall library manages basic functionality for a wall-like user interface component. It is the basis of the thumbnail browsing mode in the `photoviewer` application, but was written with customization in mind using a basic Model-View-Controller separation. The library separates the specific data source feeding into the wall (i.e., photos, or music, or videos, or other content), from the actual rendered representation (the "cells", which could have a variety of different visuals/layouts), from the user input that manipulates the position/navigation of the wall.

The library also demonstrates how to interact with 3D objects using one form of 'picking', by rendering the main scene objects with a special shader that colors each one differently, then reading back the 'clicked' pixel from the framebuffer to see what color was there, and thus which object was picked.

nv_shader

The nv_shader library was designed to abstract the differences of loading of GLSL-E shaders on the PC-based emulator and the actual Tegra devkit, and encapsulate some best-approach code into a set of helpful functions.

Rather than duplicate other documentation, please look at the OpenGL ES 2.0 Development chapter in the Programming Guide for specific use of nv_shader to aid your GL development.

In addition, you can look at pretty much any of the OpenGL ES 2.0 samples or our demos, as they all take advantage of the library.

For further details of functions and variables exported for use, please refer to `nv_shader` information in the SDK Libraries help file (`\docs\tegra_sdk_libs.chm`).

nv_surface

The `nv_surface` library is one example of encapsulating OpenGL ES 2.0 framebuffer object (FBO) handling within a set of C++ classes. The library abstracts creation of the FBO and linked texture, preparing to render to the FBO, restoring a prior render target, and preparing to render FBO contents via the linked texture.

The major blocks of code/functionality are in `NvTexture2DSurface`, which leverages multiple inheritance to be both a `NvSurface` and an `Nv2DTexture`, whichever is the desired/needed 'view' of the object.

Other samples and demos use alternate approaches to managing FBOs. The `fbo` sample does all steps directly, to demonstrate in one file how easily FBOs can be used, whereas the `photoviewer` demo instead has a `FrameBuffer` class (used by all of the major TransitionEffect subclasses – and also used by `visu`) that wraps up the creation and cleanup pieces (FBO and its linked texture), leaving the bind/restore calls to the developer.

nv_tileflow

The `nv_tileflow` library is used by the `videoplayer` demo for its 3D 'spinner' interface that uses an arc of 'tiles' (a 'tileflow') to visually represent media. The `nv_tileflow` library handles loading application-specified 3D geometry for tiles and the shaders for the various lighting, shadowing, and mirror effects. It also manages an abstract list of tiles (that is, the client app knows what the a given tile 'index' means, but `nv_tileflow` just knows as set of indices and the matching textures for each), as well as the current 'visual' state of the tileflow, and of course handles rendering of the tileflow.

Note that the `nv_tileflow` library only indirectly handles the dragging and flicking of the list – while it implements a kind of 'physics' system for spinning the tiles, it leaves it to each application to manage actual user interaction (that is, event handling, generally via gesture events generated from `nv_gesture`), and the application must translate user input into the appropriate 'changes' in the tileflow state.

nv_ui

The `nv_ui` library is a 2D GUI system built in C++, where everything derives from the base `NvUIElement` class. Elements are positioned in 2D screen space for natural layout, can handle OpenKODE events, can be grouped in an instance of the `NvUIContainer` subclass to manage entire sets of elements as one (handling events, drawing, repositioning, etc.). Other subclasses include graphics (loaded from texture files on disk, or in memory), text blocks (leveraging the `nv_bitfont` library), frames, progress/thermometer bars, buttons, and sliders.

The best examples of its use are the SDK demos, such as `addrbook`, `camera`, and `photoviewer`. All of the demos rely heavily upon the `nv_ui` library for their user interfaces.

For details on functions and variables exported to the developer, please refer to the coverage of `nv_ui` in the SDK Libraries help file (`\docs\tegra_sdk_libs.chm`).

The "Launcher"

The SDK libraries, `nv_main` and some of the samples and demos make reference to the "Launcher". This is a demo that previously shipped with the WindowsCE/XP-only Tegra SDK. This composited UI demo is not yet shipping as a part of the new Tegra Khronos Apps SDK, but the "hooks" used by some other demos are retained from previous SDKs to allow it to be shipped at a later date. Currently, when reading the `nv_main` code in particular, developers can simply make the assumption that the global variable `g_runningInLauncher` will be `KD_FALSE`.

Known Issues on Current OS Support Packs

OS-independent

Native_render

The shipping version of native_render does not include keyboard or mouse controls to change the rendering mode. The next SDK update will include such controls.

CE6 Tegra 250 Support Pack Version 5265393

Simpleplayback

- The simpleplayback demo experiences a lockup on exit

Videoplayer

- EGLSurface video rendering mode is not supported

- Playing videos using overlay mode produces a lockup on exit

Jpegencode and Mpegencode

- Both of these demos currently function and generate output files. However, the files produced include significant color-channel shifting

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