developer WORLD

February 2011

How to access the Xperia™ PLAY touch pad through the Android™ NDK

First edition (February 2011)

Document history

Change history

2011-02-13	Version 1.0.0	First version

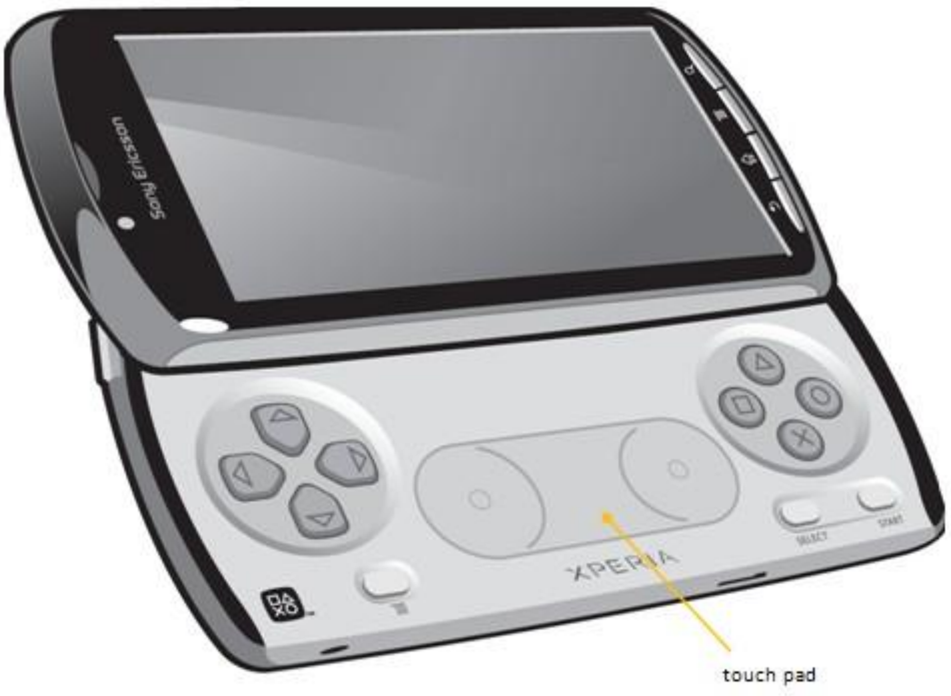
Contents

Introduction

You can use native code, that is, code written in C or C++, in conjunction with the Android™ Native Development Kit (NDK) to access touch events from the Xperia™ PLAY touch pad. (Note that you cannot use the Android SDK to access these touch events from the Xperia™ PLAY touch pad.) This article shows you how to do that.

The Xperia PLAY™ touch pad and touch events

One of the input devices provided with the Sony Ericsson Xperia™ PLAY is a touch pad.



The touch pad gives you a lot of flexibility. As a game developer, you can design the touch pad to act as an analog gaming device such as a joystick. Some contexts where the touch pad can be used in this way include:

- First-person shooters. The two analog joystick areas of the touch pad can be used much like dual joysticks for directional movement and aim control.

- Role-playing games. The touch pad can allow more than 8 directional movements for a character.
- Racing games. The touch pad can be used for smooth steering and for vehicle controls.
- Fighting games. The touch pad can be used to direct complex movement patterns.

You can also set things up so that the entire touch pad area can be used as an aiming device for a shooting game. Some other examples where the entire area of the touch pad can be used include:

- Pictionary type games. The touch pad can be used as a drawing space.
- Computer mouse simulators. The touch pad can be used to simulate computer mouse movements such as button selections or drag and drop actions.

However, for the touch pad to perform as an input device for a game, Android 2.3 needs to recognize touch events and pass them to the game for processing. Unfortunately, there are no APIs in the Android 2.3 SDK that can detect touch events from a touch pad. The APIs in the Android SDK can recognize touch events only from input devices that are associated with a display. The touch screen is associated with a display, but the touch pad is not. So the Android SDK can handle touch events from the former, but not the latter. But there is a solution. You can use native code, that is, code written in C or C++, in your Android game to handle touch events from the XperiaTM PLAY touch pad.

Using native code to handle touch events

There are two important elements in using native code to handle touch events from the XperiaTM PLAY touch pad: the Android Native Development Kit (NDK) and the NativeActivity class in the Android SDK. For further information about the NDK, see What is the NDK? on the Android Developers site. For further information about the NativeActivity class, see Native Activity on the Android Developers site.

The NDK lets you create native code for your Android application. The NativeActivity class enables you to write an Android activity in native code. In addition, the NativeActivity class handles the communication between the Android framework and your native code. The class notifies your native code of any activity lifecycle callbacks such as onCreate() or onPause(). You can implement the callbacks in your native code to handle these events when they occur. The NDK provides a header file, native_activity.h, that defines the callback interface and data structures that you need to implement in your native activity.

Threading considerations

One obstacle in using the `NativeActivity` class is that in taking this approach your native activity runs on the same thread as your Android application. If a callback implementation in your native activity blocks, it blocks your entire application until the callback completes its processing. So be careful – when you use the `NativeActivity` class, you need to ensure that your callback implementations do not block. For example, you could create another thread for your main application loop and, if needed, synchronize the threads when the registered callbacks are called.

The native-app-glue helper library

However, there is an easier way to implement your native activity. The NDK offers a static helper library named native-app-glue that handles the thread creation for you. The helper library, which is defined in the android_native_app_glue.h header file, is built on top of the native_activity.h interface. Significantly, the native-app-glue code spawns a separate thread to handle events created from registered callbacks or input events. In addition, the helper library interface provides a facility called a looper that listens for activity lifecycle events as well as input events coming from the input queue attached to the activity. In general, it's easier to process events through the looper and the native-app-glue interface than directly through callbacks. If you process events through callbacks, you have to deal more explicitly with thread synchronization and perform actions such as destroying the old input queue when receiving a new one.

Let's examine how to implement a native activity using the native-app-glue helper library so that you can detect and process touch events on the Xperia™ PLAY touch pad.

Using native-app-glue to process touch events

The native-app-glue approach is pretty simple. All you need to do is:

1. Provide a main entry point for your native activity and register two callbacks for processing input and system events.
2. Get events from the looper.
3. Process the events as appropriate.

When you implement a native activity using the native-app-glue library, you need to provide a function named android_main(). This is the main entry point for the native activity. The native-app-glue library calls this function when the native activity is created. The function runs in its own thread, that is, apart from the main thread of the application.

When the native-app-glue library calls the android_main() function, it passes it a pointer to an android_app structure that is defined in the android_native_app_glue.h header file.

```
struct android_app {
       void* userData;
       void (*onAppCmd)(struct android_app* app, int32_t cmd);
       int32_t (*onInputEvent)(struct android_app* app, AInputEvent* event);
       ANativeActivity* activity;
       AConfiguration* config;
       void* savedState;
       size_t savedStateSize;
       ALooper* looper;
       ARect contentRect;
```

```
        int activityState;
        int destroyRequested;
       …
}
```

Pay particular attention to the following parts of the structure:

- userData. Your native activity can place a pointer here to an object that maintains state data for the application.
- onAppCmd. Points to your function that processes system events and commands sent from the application's main thread.
- onInputEvent. Points to your function that processes input events.
- Looper. Holds an instance of an ALooper object that listens for activity lifecycle events, such as pause and resume, as well as input events from the application's input queue.

After you have the android_main function in place and fill the android_app structure with pertinent data, all your native application needs to do is process the events detected by the looper.

A touch pad code example

Let's take a look at an example of a native activity that uses the native-app-glue library. The native activity is part of a project named TouchpadNAGlue, which you can use to build the application. The example is included in the touchpadexamples.zip file, which you can download from the XperiaTM PLAY section of Developer World. You'll find the native activity in the main.c file in the project's jni directory.

The android_main() function

Recall that the android_main() function is the main entry point for the native activity. When the native-app-glue library calls the android_main() function, it passes it a pointer to an android_app structure that is defined in the android_native_app_glue.h header file. Here's what the android_main() function declaration in the touch pad code example looks like:

```
android_main( struct android_app* state )
```

With the android_app structure passed in, the native function can set various pointers in the structure to track the state of the application as well as process input events. Here are the pointers that the touch pad example sets in android_main():

```
        struct ENGINE engine;

        state->userData     = &engine;
        state->onAppCmd     = engine_handle_cmd;
        state->onInputEvent = engine_handle_input;
        engine.app         = state;
```

The &engine pointer refers to the address of a structure that the native activity sets up to hold the shared state for the application:

```
struct ENGINE
{
    struct android_app* app;

    int         render;
    EGLDisplay  display;

    EGLSurface  surface;
    EGLContext  context;
    Int         width;
    Int         height;
…
};
```

The pointer to engine_handle_cmd refers to a function that handles commands from the main thread in the application. The pointer to engine_handle_input refers to a function that handles input events such as motion events from the touch pad.

The event loop

After setting pointers in the android_app structure, the android_main() function in the touch pad code example runs an event loop to receive and process events. Remember that looper mentioned earlier? It goes about its business listening for activity lifecycle events as well as input events from the input queue attached to the activity. To read and process the events from the looper, the android_main() function in the touch pad example creates a loop that calls the Alooper method ALooper_pollAll(). The method polls for all pending events from the looper.

```
while( 1 )
{
    int ident;
    int events;
    struct android_poll_source* source;

   while( (ident = ALooper_pollAll( engine.render ? 0 : -1, NULL,
   &events,
   (void**)&source) ) >= 0 )
   {
```

The source parameter in the call to ALooper_pollAll() points to a structure named android_poll_source, which contains information to call one of the callbacks registered in the android_app structure. The native activity then processes the event by calling the process() function, which will call one of the registered callbacks. Both the android_poll_source structure and the process() function are defined in the android_native_app_glue.h header file.

```
           if( source != NULL )
           {
               source->process( state, source );
           }
           …
```

For input events, the process function calls the function pointer android_app::onInputEvent(), which has been set in the touch pad code example to the engine_handle_input() function. For activity lifecycle and other system events, the process function calls the function pointer android_app:: onAppCmd(), which has been set to the engine_handle_command() function. (You'll learn more about the engine_handle_command() function in the *Processing activity lifecycle events* section later in this article.)

Processing input events

The engine_handle_input () function in the touch pad code example handles input events. The first thing the function does is determine whether the event is a motion event. If it is, the function gets the source of the event and a count of the number of pointers associated with the event.

```
engine_handle_input( struct android_app* app, AInputEvent* event )
{
   struct ENGINE* engine = (struct ENGINE*)app->userData;
   if( AInputEvent_getType(event) == AINPUT_EVENT_TYPE_MOTION )
   {
      engine->render          = 1;
      int nPointerCount= AMotionEvent_getPointerCount( event );
      int nSourceId           = AInputEvent_getSource( event );
      int n;
```

Then for each pointer, the function gets the pointer ID and action code. Even though the function gets the action code for each pointer, the action code does not differ between pointers in one event . So you don't necessarily need to fetch the action code for every pointer. You may move it outside of the pointer loop.

```
      for( n = 0 ; n < nPointerCount ; ++n )
        {
              int nPointerId= AMotionEvent_getPointerId( event, n );
           int nAction= AMOTION_EVENT_ACTION_MASK &
                      AMotionEvent_getAction( event );
```

The functions, structures, and constants that begin with A, such as AMotionEvent_getPointerCount(), AInputEvent, and AINPUT_EVENT_TYPE_MOTION are defined in the input.h header file exposed by the NDK.

You might wonder why there's a need for a bitwise AND (&) operation. The operation is needed because only the lowest 8 bits returned by MotionEvent_getAction() actually represent the action code. To get the action code, the engine_handle_input() function performs a bitwise AND operation. The two operands of the AND operation are the result returned by AMotionEvent_getAction() and the constant, AMOTION_EVENT_ACTION_MASK. The constant is a bit mask of the part of the action code that represents the action itself.

Although this article focuses on touch events from the touch pad, the TouchpadNAGlue sample application can respond to touch events from the touch screen as well as the touch pad. So the native activity in the application first tests to see which input device is associated with the event.

```
struct TOUCHSTATE *touchstate = 0;

if( nSourceId == AINPUT_SOURCE_TOUCHPAD )
    touchstate = engine->touchstate_pad;
else
    touchstate = engine->touchstate_screen;
```

The major purpose of the engine_handle_input () function is determining what type of touch event took place. The touch events it looks for are defined by the following constants:

AMOTION_EVENT_ACTION_DOWN	The user touched the touch pad or touch screen with the primary pointer (finger). This represents the start of the touch gesture.
AMOTION_EVENT_ACTION_POINTER_DOWN	The user touched the touch pad or touch screen with a non-primary pointer.
AMOTION_EVENT_ACTION_POINTER_UP	The user released a non-primary pointer from the touch pad or touch screen.
AMOTION_EVENT_ACTION_UP	The user released the primary pointer from the or touch screen. This represents the end of the touch gesture.
AMOTION_EVENT_ACTION_CANCEL	The user cancelled the touch gesture or the application lost focus during a gesture.

Here, for example, is how the engine_handle_input () function handles user touches on the touch pad or touch screen.

```
if( nAction == AMOTION_EVENT_ACTION_DOWN || nAction ==
        AMOTION_EVENT_ACTION_POINTER_DOWN )
    {
        touchstate[nPointerId].down = 1;
    }
    …
    if (touchstate[nPointerId].down == 1)
    {
        touchstate[nPointerId].x = AMotionEvent_getX( event, n );
        touchstate[nPointerId].y = AMotionEvent_getY( event, n );
    }
```

The AMotionEvent_getX()and AMotionEvent_getY() functions get the X and Y coordinate offsets of the touch positions on the touch pad or touch screen.

At this point, the touch pad code example can take some visible action. And the action it takes is to display a small square on the screen that represents the current touch position on the touch pad or touch screen -- or two squares that represent the touch positions for two finger touches. When the user's finger moves to a new position on the touch pad, the square moves accordingly. When both fingers move on the touch pad, so too do both squares.

Running the touch pad example

Let's see how the touch pad code example does that for touch gestures.

Rendering touch gestures

The rendering of the touch gestures is handled by the engine_draw_frame() function in the native activity. The function takes advantage of OpenGL to render graphics to the screen. The location and data format of the array of vertex coordinates to use for rendering the squares is done with the following code:

```
static GLfloat square[] =
{
     -25, -25,
     25, -25,
     -25, 25,
     25, 25
};


engine_draw_frame( struct ENGINE* engine )
  {

      …
```

```
        glVertexPointer( 2, GL_FLOAT, 0, square );
```

And here is the code that renders the squares based on touches on the touch pad.

```
static const float padx_scale = 1.0f/966.0f;
static const float pady_scale = 1.0f/360.0f;
glColor4f(1.0f, 1.0f, 0.0f, 1.0f);
for( i = 0; i < 64; ++i )
{
    if( engine->touchstate_pad[i].down == 0 )
        continue;

    glPushMatrix();

        glTranslatef( ((float)vPort[2]) *
                      (((float)engine->touchstate_pad[i].x) *
                      padx_scale),
                      ((float)vPort[3]) *
                      (((float)engine->touchstate_pad[i].y) *
                      pady_scale),0.0f );
        glDrawArrays( GL_TRIANGLE_STRIP, 0, 4 );
    glPopMatrix();
}
```

The engine_draw_frame() function needs to translate the touch position on the touch pad to a display position on the screen, factoring in the screen dimensions. In other words, it needs to translate the position from the touch pad coordinate system to the screen coordinate system.

To do this, the function first scales down the touch position on the touch pad to a value between 0 and 1. The padx_scale and pady_scale variables contain the values to scale the x and y coordinates of the touch position, respectively. The padx_scale value is 1.0f/966.0f, where 966 is the touch pad resolution in the x dimension. The pady_scale value is 1.0f/360.0f, where 360 is the touch pad resolution in the y dimension.

After the x position is scaled down, the function multiplies it by the x coordinate value of the viewport, vPort. The scaled-down y position is multiplied by the y coordinate value of the viewport.

The glTranslatef() function tells OpenGL to render at the specified position when it starts rendering. The glDrawArrays() function tells OpenGL to render the square to the screen.

Processing activity lifecycle events

The engine_handle_cmd () function in the touch pad code example handles activity lifecycle events sent through the looper. These events begin with APP_CMD and are defined in the android_native_app_glue.h header file. Here are the activity lifecycle events that the engine_handle_cmd () function handles:

APP_CMD_SAVE_STATE	Save the current state.

APP_CMD_INIT_WINDOW	Initialize the window for this native activity.
APP_CMD_TERM_WINDOW	Terminate the window for this native activity. It is being hidden or closed.
APP_CMD_GAINED_FOCUS	The window for this activity gained focus. Rendering can begin.
APP_CMD_LOST_FOCUS	The window for this activity lost focus. Rendering must stop.

Here, for example, is the code in the engine_handle_cmd () function that processes the APP_CMD_INIT_WINDOW command.

```
engine_handle_cmd( struct android_app* app, int32_t cmd )
switch( cmd )
{
    case APP_CMD_INIT_WINDOW:
    if( engine->app->window != NULL )
    {
        engine->has_focus = 1;
        engine_init_display( engine );
        engine_draw_frame( engine );
    }
    break;
    …
}
```

In this case, the function set the focus state of the application. It then calls the engine_init_display() function to initialize the display through various OpenGL and EGL attributes such as those in the following code snippet.

```
engine_init_display( struct ENGINE* engine )
{
 EGL
const EGLint attribs[] =
{
EGL_SURFACE_TYPE, EGL_WINDOW_BIT,
EGL_BLUE_SIZE, 5,
EGL_GREEN_SIZE, 6,
EGL_RED_SIZE, 5,
EGL_NONE
};
```

Notice that the function sets attributes for a 16-bit RGB565 color scheme. However, you will not always receive a surface with these attributes. You can get the actual pixel format of the window surface by calling the ANativeWindow_getFormat() function, and configure EGL accordingly.

Last, the engine_handle_cmd () calls the engine_draw_frame() function to draw the frame.

Other approaches

Of course, you don't have to use the native-app-glue helper library in your application. You can handle touch events from the Xperia™ PLAY touch pad through a native activity and the NDK (and no helper library). In this case, your game would handle events through callbacks. Remember though that your game will have to deal more explicitly with thread synchronization and perform actions such as setting up the input looper. You can find an example of this approach in a project named TouchpadNA. The example is included in the touchpadexamples.zip file which you can download from the Xperia™ PLAY section of Developer World. You'll find the native activity in the main.c file in the project's jni directory.

And you can couple your native activity with a Java activity class. You can find an example of this approach in a project named TouchpadNAJava. The example is included in the touchpadexamples.zip file, which you can download from the Xperia™ PLAY section of Developer World. You'll find the native activity in the main.c file in the project's jni directory and the Java activity class in the src directory.

More information

1. Download the Android NDK>>
2. Learn more about the Android NDK>>
3. Learn more about the NativeActivity class>>