NVIDIA.

ANDROID LIFECYCLE FOR APPLICATION DEVELOPERS: GUIDELINES AND TIPS

May 2011

Developer Whitepaper

DOCUMENT CHANGE HISTORY

Document Number

Version	Date	Authors	Description of Change
01	30 May, 2011	Lbishop	Initial release

TABLE OF CONTENTS

ANDROID LIFECYCLE BASICS IN PRACTICE

TOP-LEVEL ITEMS

Android applications must conform to the Android OS's concepts of application lifecycle. Applications failing to code for application lifecycle can be less responsive, less stable and less compatible across the range of Android devices than an app that even handles simple lifecycle events. This whitepaper is designed to provide guidelines and tips for application developers with respect to handling common lifecycle events.

Note: This document is not a replacement for a basic understanding of Android application development. It is merely additional, practical documentation on Android application lifecycle behavior in real applications on commercial devices. For Android application basics, visit http://developer.android.com/index.html

Android applications are based on several components, the most visible of which is the Activity class ("visible" is literal, because it implements all UI and views in an Android application). Android specifies the lifecycle of each Activity in an application in detail, known as the Android Activity Lifecycle.

Simple Android applications, especially 3D games that handle all user interfaces in the 3D engine tend to be single-Activity Android applications. As a result, the concepts of "application lifecycle" and "Activity lifecycle" can be considered as one and the same. We will do so for the purposes of this document.

The Android lifecycle is exposed to applications via a set of callback member functions in the application's Java code. At a top level, the basic sequence of Android lifecycle callbacks follow a sort of "stack", along with a set of additional callbacks that appear to happen outside of this strictly hierarchical sequence. We note the items that should likely be handled in each. This list is not exhaustive and covers mainly the common ones.

The Lifecycle Hierarchy Events

The following events follow a basic hierarchy as indicated by indentation. They are all override-able members of Activity.

onCreate: called to set up the Java class for the instance of the app

> **onStart**: technically, called to initiate the "visible" lifespan of the app; at any time between **onStart** and **onStop**, the app may be visible. We can either be **onResume'd** or **onStop'ped** from this state. Note that there is also an event for **onRestart**, which is called before **onStart** if the application is transitioning from **onStop** to **onStart** instead of being started from scratch.
>
> > **onResume**: technically, the start of the "foreground" lifespan of the app, but this does not mean that the app is fully visible and should be rendering – more on that later
> >
> > **onPause**: the app is losing its foregrounded state; this is normally an indication that something is fully covering the app. On versions of Android before Honeycomb, once we returned from this callback, we could be killed at any time with no further app code called. We can either be **onResume'd** or **onStop'ped** from this state
>
> **onStop**: the end of the current visible lifespan of the app – we may transition to **on(Re)Start** to become visible again, or to **onDestroy** if we are shutting down entirely. Once we return from this callback, we can be killed at any time with no further app code called on any version of Android.

onDestroy: called when the Java class is about to be destroyed. Once this function is called, there is only one option for transition (other than being killed): **onCreate**.

The non-Hierarchy Events

Another set of important events are not hierarchical; we have seen them come in various levels of the previously described hierarchy events. These events cover two categories: window focus and surface status.

Window Focus Callbacks

Window focus generally indicates whether an app's window is the top-most, visible and interact-able window in the system. There are two functions that relate window focus, and they seem to be redundant for apps with a single major view: **`Activity::onWindowFocusChanged`** and **`View::onWindowFocusChanged`**. Each of these functions is passed a Boolean that indicates whether the callback is denoting focus gained or lost. Starting from **`onCreate`**, focus is assumed not to be held, so applications should be looking for an initial focus gained callback. In general, focus gain seems to be indicated only with an app that is "resumed", that is between **onResume** and **onPause** callbacks. However, as can be seen from callback sequences later in this document, an application may not receive a focus lost message before its **onPause** callback is called. In fact, in cases of quick shutdown for configuration changes, the system may go from resumed and focused all the way to **onDestroy** without ever indicating focus lost. So **onPause** and **onWindowFocusChanged** must be used in tandem to determine the visible and interact-able state of the app. It is this lack of focus lost callbacks at expected times which places the window focus events in the "non-hierarchical" category.

Surface Callbacks

The other forms of non-hierarchical callbacks of great importance to 3D apps are the surface callbacks. There are three of them, and they are associated with an application's **SurfaceView**(s). Since most 3D games will use a single, full-screen **SurfaceView**, we will assume a singleton. This singleton **SurfaceView** is the host of the application's EGL Surface for rendering, and is thus pivotal to the app.

Note that some applications may choose to use **GLSurfaceView** to wrap their OpenGL ES rendering structure. While we do not assume this use in the course of this document (we wish to handle the rendering more directly), the surface callbacks remain in place.

The three callbacks are: **surfaceCreated**, **surfaceChanged**, and **surfaceDestroyed**. These three callbacks form a hierarchy within themselves, as you will always receive a **surfaceCreated** callback to indicate a surface is available for the **SurfaceView**, one or more **surfaceChanged** callbacks to indicate that format or (most commonly) width and/or height have changed, and then a **surfaceDestroyed** callback to indicate that the **SurfaceView** no longer makes a surface available.

These are pivotal to 3D applications because the application's EGL Surface may only exist between the surf**a**ceCreated and **surfaceDestroyed** callbacks. Furthermore, since most Android EGL implementations require that you bind a non-NULL surface whenever you bind a non-NULL context, until you have a valid surface, you cannot bind a context and thus cannot load any 3D resources.

While these functions form their own hierarchy, they can interleave into the overall hierarchy of the lifecycle in rather complex ways. Like window focus, we generally see windows initially created (**surfaceCreate**) by an app in the resumed state. However, in practice, we see **surfaceDestroyed** callbacks after **onPause** or even after **onDestroy**. As a result, applications may need to handle shutting down EGL before their surfaces are destroyed.

ADDITIONAL LIFECYCLE CONCEPTS AND REALITIES

Being “Killable”

The Android developer documentation makes reference to an application being in a “killable” state. While Android tries to keep the process of an application resident even after it has exited (i.e. after **`onDestroy`**), it does need to be able to kill these processes in low-resource situations to reclaim memory. The states in which an application is killable differ per OS version. On all versions of Android, applications that have returned from **`onStop`** or **`onDestroy`** are silently killable. On versions of Android prior to Honeycomb, applications that had returned from **`onPause`** were also killable. Being killable simple means that Android reserves the right to terminate your application’s process at any time without running even another instruction of your app’s code. In other words, if you have any state that must be recoverable (such as a player’s game progress, items, awards, etc) you must save those to persistent storage no later than the last callback before entering a killable state.

In addition, while applications can run native threads even when they are in a killable state and even post-onDestroy, this is to be avoided, since the process kill will also kill those threads. This could cause all manner of corruption and shutdown issues.

Callback-Handling AND ANR’s

It may seem possible to avoid becoming killable by choosing not to return from the last callback before entering the killable state (e.g. **`onPause`** or **`onStop`**). This will not work. In fact, it will have disastrous consequences for your application that highlight another aspect of application lifecycle. Applications are required by Android to handle any posted input events within 5 seconds. Since that input is delivered on the same thread (the main or UI thread) as the lifecycle callbacks, blocking for 5 seconds in a lifecycle callback will break this rule.

When an application breaks the “5 second rule”, the device user will be given an Application Not Responding or “ANR” error dialog box. This will give the user the option of waiting for your (assumedly crashed) application to continue or else kill your application. This is to be avoided in any application. Applications should carefully plan their callback code to avoid making blocking calls that could take an unknown amount of time. This includes such operations as large storage reads/writes and especially network operations, which should be done in spawned threads.

onPause versus onStop

In addition to the lifecycle state items previously mentioned, there is one other important difference between **`onPause`** and **`onStop`** callbacks. The **`onPause`** callback halts the visible UI thread, and thus any time spent in **`onPause`** will actively block the app's UI interaction. Thus, it is important to make **`onPause`** as responsive as possible while still saving absolutely key application state.

The **`onStop`** callback, on the other hand, is called once the app is no longer visible. Thus, while it is important to avoid ANRs from this callback, it is not pivotal to return immediately from **`onStop`**. Spending a little more time in **`onStop`** will not cause application/device responsiveness issues.

Rendering and Threading

Related to callbacks and the potential for ANR's, it is important for applications to do their rendering in a secondary thread, either in Java or in native. If a native thread is to be calling OpenGL and EGL, it should be sure that it calls up to Java from native to bind the EGL context. The thread that is calling OpenGL ES must be the same thread that bound the context. The main UI thread should not bind the context and then have some other thread in native calling OpenGL ES. This is likely to cause problems. NVIDIA's Tegra Android Samples Packs include examples of exposing EGL call wrappers in Java that can be called from native threads to make this easier.

USER ACTIONS AND CALLBACK SEQUENCES

The most common callback sequences for common user actions are listed in the following sub-sections.

Sequencing Differences

As noted previously, while **`onCreate`**, **`onStart`**, and **`onResume`** (and their matching "downward" callbacks **`onDestroy`**, **`onStop`** and **`onPause`**) are defined by spec to happen in a nested ordering, focus events and window created/update/destroy events are not specified to fall in definite locations in these sequences. You should be prepared to see such differences as:

- The window can gain focus before or after the surface is created
- The window can lose focus before or after the surface is destroyed
- These cases do not imply each other (i.e. all four possible outcomes have been seen in practice)

Thus, it is important to flag all of these changes and have an update function that is called on each change to check the current state flags and act when the combinations of important state are entered.

Note: these are merely common examples - some devices may produce slightly different results, as we will document in the section “Surprising Application Lifecycle Events and how to Handle them”.

Launching a New Application

When a new instance of an application is launched, the Java class for the Activity is created (the process itself may actually have still be resident from an earlier run of the app), and the full system is brought up, including a valid, visible, focused window.

```
+-onCreate
+-onStart
+-onResume
+-surfaceCreated
+-surfaceChanged: 1366, 695
+-onWindowFocusChanged (TRUE)
```

At this point, the application should be rendering, playing its sounds and accepting input.

Pressing the Back Button

An application receives the back button event and has two choices – it can either:

1. “Eat” the event, handling it internally by stepping backwards in its UI structure. It then does not pass the event to the super-class’s **onKeyDown**, and returns true (“handled”) from its own **onKeyDown**. The handling of the event is complete
2. Allow Android to process the event, in which case it will pop the application off the UI stack and destroy the application’s Activity, essentially quitting the application (although the app’s process will likely keep running).

```
+-onPause
+-onWindowFocusChanged (FALSE)
+-surfaceDestroyed
+-onStop
+-onDestroy
```

This is quite a heavyweight case. For this reason, most applications will “eat” the Back event internally and provide dialogs for the user to confirm before quitting actually happens.

Pressing the Home Button

The Home button sends the app down in the UI stack. The application's Activity still exists, but is completely hidden and may be killed at any time. Additionally, it loses its rendering surface (which is not an issue, since it is not visible in any case).

```
+-onPause
+-onWindowFocusChanged (FALSE)
+-surfaceDestroyed
+-onStop
```

While the application is stopped, it may be resumed at any time by selecting the app from the list of resident applications.

Resuming a "Home'd" Application

If an application sent further down the UI stack by a Home event is selected from the list of running applications before Android chooses to kill it, the app is "restarted", which calls not only start but also a special onRestart callback that differentiates between a newly-launched instance and a resumed one.

```
+-onRestart
+-onStart
+-onResume
+-surfaceCreated
+-surfaceChanged: 1366, 695
+-onWindowFocusChanged (TRUE)
```

The application is once again ready for rendering, sound and interaction.

Opening a Status Icon Pop-up

Pulling down a pull-down status bar or tapping on a status icon causes some UI items to be composited on top of the application. Thus, the application is visible and can render, but is not focused and cannot receive input.

```
+-onWindowFocusChanged (FALSE)
```

This is the lightest-weight version of being down-shifted.

Closing a Status Icon Pop-up

When a pending partial-screen UI element on top of your app is dismissed, focus is regained.

```
+-onWindowFocusChanged (TRUE)
```

Suspend/Resume

Suspend and resume happen in a three-step sequence and thus are best discussed together. Basically, when the device is suspended with the power button or the screen-saver timeout expires, the device suspends. If the power button is pressed again, it resumes to the lock-screen. At this point, if the user unlocks the device, the application is resumed. If the user waits several seconds without unlocking the lock screen, the device will suspend again.

Note: Suspend/Resume is the most variable of all of the cases. See the later section on "Surprising Application Lifecycle Events and how to Handle Them" for practical details on handling this case.

Suspending the Device

Most commonly, when suspended, the application will be paused and will lose focus.

```
+-onPause
+-onWindowFocusChanged (FALSE)
```

Since this is explicitly a low-power state, the application should have stopped all rendering and sound, and likely any background processing that isn't 100% required to keep the app alive.

Resuming the Device

When the device is resumed with the power button, the application is also resumed. However, the lock screen covers the application's window. No focus change is returned to the application. The application should not play sound or render at this point. Within a few seconds, it may be paused again if the lock screen times out.

```
+-onResume
```

Unlocking the Lock Screen

Once the lock screen is unlocked, the application is focused again. Since it is now resumed and focused, the application should consider itself signaled to begin rendering, playing sound and accepting input.

```
+-onWindowFocusChanged (TRUE)
```

Alternative Sequence

Note that in some cases, only the **`onPause`** is received on suspend, in which case, the focus lost callback actually comes when the device is resumed to the lock screen. In other words, resuming sends an indication that you have been resumed (**`onResume`**) and then an indication that you are hidden (focus lost). And the unlock case remains unchanged (focus regained).

BASIC ANDROID LIFECYCLE RECOMMENDATIONS

The previous sections provided a basic introduction to the practical meaning of the most important application lifecycle callbacks, as well as describing the most common ordering of the calls and how they are triggered by user actions. However, the most important question surrounding these callbacks is, "How do I map common game/application actions to these callbacks?"

ALL A GAME NEEDS

At its most basic, in terms of lifecycle, most games need to know little more than:

- When can I create my game data and/or rendering resources?
- When should I be rendering or not rendering (playing/not playing sound)?
- When should I move to an auto-pause dialog?
- When do I need to shut down?

Generally, this breaks down to knowing:

- When an EGL context can be created, when it can be bound, when it is lost
- When the app has a valid rendering surface that is visible and interactable
- When something has caused the app not to be the top-most UI element and when it comes back
- When the system is shutting down the application

The following sections will discuss important concepts to answer these questions.

COMMON STEPS AND WHEN THEY CAN BE DONE

Loading and Deleting Game Data

Technically, game data other than OpenGL ES resources can be loaded at any time from **`onCreate`** beyond. Actually, application copies of OpenGL ES resources could be loaded in **`onCreate`** as well, but they cannot be loaded to OpenGL ES until it is initialized. So it may or may not make sense to load the rendering resources so early.

Note that depending on your manifest settings, the application may be brought up and shut down several times during initial launch (see the later section "Surprising Application Lifecycle Events and how to Handle Them"). If you choose not to use the settings to avoid this, you may wish to delay the loading of any data until you know that the game is fully launched. Doing so can avoid doubling or tripling the load time if the system launches and re-launches the app several times.

Initializing and Deleting EGL, Contexts, Surfaces and Rendering Resources

Technically, the following steps can be done at any time between **`onCreate`** and **`onDestroy`**, as they do not require any specific resources from the app:

1. **`eglInitialize`**
2. Querying, sorting and selection of the **`EGLConfig`**
3. Creation of the **`EGLContext`** for OpenGL ES

However, the context cannot be bound until there is an **`EGLSurface`** that can be bound at the same time. The **`EGLSurface`** in turn cannot be created until there is an Android surface available to attach the **`EGLSurface`**. Such a surface is provided by the **`surfaceCreated`** callback. As a result, in order to fully set up OpenGL ES, we must be between **`surfaceCreated`** and **`surfaceDestroyed`**.

Indeed, since an **`EGLContext`** must be bound in order to load OpenGL ES resources (which in turn requires an **`EGLSurface`**), this means that we cannot load our 3D resources for the application until we receive **`surfaceCreated`**. We should note further that it is best to wait for **`surfaceChanged`** to indicate that the surface has positive width and height as well; some cases can generate temporary 0x0 surfaces.

When we receive a **`surfaceDestroyed`** callback, we must immediately unbind the **`EGLSurface`** and **`EGLContext`** (**`eglMakeCurrent`** with **`display`**, **`NULL`**, **`NULL`**) and must stop rendering. However, we do not need to destroy the **`EGLContext`** at this time. It can be kept resident even when it is not bound. When a new surface is available we can rebind the existing context and we will not have lost any of our rendering resources. This can avoid an expensive asset reloading process in some lifecycle cases like suspend/resume.

We should still consider deleting the **`EGLContext`** (and thus freeing resources) in onStop, even if the context is not bound. If the context is bound, it may be best to release the GLES resources manually using GLES calls. However, if there is no surface and thus no way to bind the context at the point at which we want to delete the GLES resources, the best option is to destroy the **`EGLContext`** manually using **`eglDestroyContext`**.

One common way to handle initialization by making a function that returns "true" if we are completely ready to load resources, allocating/initializing/failing as needed/possible. This can handle partial initializations; for example, it can handle the case where we had a surface, lost it due to **`surfaceDestroyed`** and now have a new Android surface (and can thus create and bind a new **`EGLSurface`**). The pseudo-code might be as follows:

```
bool EGLReadyToRender()
{
    // If we have a bound context and surface, then EGL is ready
    if ((Is EGL Context Bound?) == false)
    {
        // If we have not bound the context and surface,
        // do we even _have_ a surface?
        if ((Is EGL Surface Created?) == false)
        {
            // No EGL surface; is EGL is set up at all?
            if ((Are EGL Context/Config Initialized?) == false)
            {
                // Init EGL, the Config and Context here…
                if (failed)
                    return false;

                // Mark that we have a new context - this will
                // be important later, as a new context bound
                // means we can/must load content into it
            }

            if ((Is there a valid Android surface?) == false)
                return false;

            // Create the EGLSurface surface here…
            if (failed)
                return false;
        }

        // We have a surface and context, so bind them

        // eglMakeCurrent with the surface and context here…
        if (failed)
            return false;
    }

    // One way or another (free or requiring some amount of setup)
    // We have a fully initialized set of EGL objects…  Off we go

    return true;
}
```

Rendering

In general, an app should only render when it is fully visible and interact-able. In Android lifecycle terms, this tends to mean:

- The app is resumed (between **`onResume`** and **`onPause`**)
- The window is focused (**`onWindowFocusChanged(TRUE)`**)

- The EGL objects are in place (see the previous section, but this basically implies that we're between **`surfaceCreated`** and **`surfaceDestroyed`** and have a bound surface and context)

There are cases where we might render outside of being the focused window and in rare cases when **`onPause'd`**, but those tend to be:

- Singletons (single frame rendered) to ensure that an auto-pause screen is rendered prior to stepping out of the way.
- Low-rate animations for when the window is visible but not focused

The Case of Suspend/Resume

A very important case to consider is the previously-discussed suspend/resume. Note the sequence: when the device is suspended, the app receives onPause and focus loss, the signal to stop rendering and playing music. When the device is resumed, onResume is given. But the app is NOT visible. It must not render and must not play sounds yet. The lock screen covers it. Users often power up to the lock screen in quiet areas to look at their clock and messages. If the app began blasting music, it could be a major usability issue. It is not until the user unlocks the device and the lock screen goes away that the focus regained callback is given. Only when both focus and onResume have both been given should rendering and sound begin again. However, the application should not resume to active gameplay on unlock – see the next section for more discussion of this.

Auto-pause

In general, gameplay should be paused to some auto-pause screen whenever the user would have lost the ability to control the game for any period of time. Commonly, this is the case when you are regaining window focus from any form of loss. This screen should be kept active even once focus is regained. Do not restart active gameplay just because focus is regained.

If you handle window focus regain by simply enabling the gameplay again, the user may be caught off guard and left to catch up. When receiving an **`onWindowFocusChanged(TRUE)`** after a **`onWindowFocusChanged(FALSE)`**, it is best to show an autopause screen to allow the user to resume gameplay on their own time.

Saving Game Progress

Game progress will be lost if it is not saved prior to the application being killed. Since an application is killable in all Android variants after returning from **`onStop`**, and is killable on pre-Honeycomb after returning from **`onPause`**, these are the obvious callbacks to use to save game state to persistent storage.

Resuming versus Restarting

In addition, it may make sense to save a flag in **`onPause`** or onStop that indicates that the current save game is an auto-save, and not a user-initiated save. Then, if the user exits out of gameplay or entirely out of the game with explicit decisions, clear this "autosaved" flag.

When your application is launched, you can check if this is a "fresh" launch initiated by a user or simply your app being relaunched because it was killed silently by Android. In the latter case, you may want to resume directly to auto-pause in the saved game, since that is presumably where the user last sat before the game was interrupted. This can make the game appear seamless with respect to getting auto-killed.

SOME SIMPLE MAPPINGS

While the following items do, to some degree, oversimplify the situation, for app developers just starting out, this section provides some basic guidelines for what to do. These guidelines can assist in "bringing up" an app; once brought up, significant lifecycle testing should be done on a range of devices.

- **`onCreate`**
  - Game data can be loaded here, but be careful about loading too much heavyweight data this early in the process, as device configuration lifecycle issues can cause the app to be shutdown after **`onCreate`** but before the window and surface are ready.
- **`surfaceCreated`**
  - The initial **`surfaceCreated`** callback is the earliest that the application can create its **`EGLSurface`** and thus is the earliest that it can bind a context, load OpenGL ES resources and render.
  - However, applications should consider delaying any rendering until the app is "fully" ready and on-screen (next bullet)
- When ***all*** of the following are true:
    - We are between calls to **`onResume`** and **`onPause`**
    - We are between calls to **`surfaceCreated`** and surfaceDestroyed
    - We are between calls to **`onWindowFocusChanged(TRUE)`** and **`onWindowFocusChanged(FALSE)`**
  - This is a safe time to be rendering and expecting input
- **`onWindowFocusChanged(FALSE)`**
  - Always auto-pause gameplay
  - Stop sounds/music
  - Strongly consider stopping rendering; at least throttle back

- **`onPause`**
  - Stop all rendering, even if you do not do so on focus loss
- **`onResume`** after **`onPause`**
  - Prepare to begin rendering again
  - Do not restart rendering or audio until you also get a “focus regained”, as the lock screen may quickly “time out” because the user had just resumed the device to look at the clock, not to start playing again.
- **`onWindowFocusChanged(TRUE)`** after **`onResume`**
  - This is a reasonable time to move an auto-pause screen, start rendering again and playing sounds
  - This is generally the signal that you are refocused and visible again (e.g. the lock screen has been unlocked and you are visible)
- **`onStop`**
  - Google now recommends freeing all graphics/3D resources here (either through GLES calls if the **`EGLContext`** is still bound, or via **`eglDestroyContext`** if the context has been unbound because the rendering surface was destroyed).
- **`onDestroy`**
  - Release all of your native resources

SURPRISING APPLICATION LIFECYCLE EVENTS AND HOW TO HANDLE THEM

There are several confusing application lifecycle cases we have seen on numerous commercial devices and OS versions that we recommend developers test with their applications. This document will list confusing cases that we have frequently seen cause problems for real, commercial applications during the QA process. It will also discuss what an application can do to better handle the situation. For each of these cases, we recommend that developers use the simple Lifecycle application in the NVIDIA Tegra Android Samples Pack, editing the app as needed to replicate the exact case. This is not meant to be exhaustive – it merely lists some items that very commonly slip through most developers in-house testing.

BASIC CONCEPTS BEHIND THE PROBLEMATIC CASES

Configuration Changes

Configuration changes in Android refer to a set of system-level events that can strong affect application and UI behavior. These include such changes as:

- Orientation change (e.g. portrait to landscape)
- Screen size/depth change (e.g. HDMI hot-plug)
- Locale changes (e.g. country code, language)
- Input device changes (e.g. hot-plugging a keyboard or laptop dock)

Many (most?) of the issues discussed in the document revolve around device orientation changes, (either intentional, unintentional, or implicit based on device behavior in other lifecycle events). By default, if an application specifies no AndroidManifest.xml keys for configuration change handling, any configuration change will actually result in the application being completely shut down (**`onDestroy`**) and then re-launched in the new configuration. Needless to say, for a large 3D game, this can be extremely expensive. However, there are other options, whether the application wishes to support both orientations (landscape and portrait) or only one.

"Forcing" Landcape or Portrait Orientation

It is possible to request that your application always run in landscape or portrait mode with the `AndroidManifest.xml` keys:

```
android:screenOrientation="landscape"
```

or

```
android:screenOrientation="portrait"
```

On some OS versions and devices, this will keep the application from ever seeing any portrait configurations or seeing any issues. However on other devices (we have seen this on Motorola Atrix and other devices), this tag behaves somewhat differently. Specifically, while it ensures that your application will "settle" to the requested orientation, on these devices and OS versions, we find that there may be considerable "thrash" between multiple orientations owing to spurious configuration changes.

On these devices, in several specific lifecycle cases discussed later in this document, the application may receive multiple configuration changes between the desired orientation and the opposite orientation. These spurious configuration changes can cause issues for applications in several manners:

1) With the default behavior, each config change (sometimes as many as three in a row) will shut down and restart the app

2) Some of these "false starts" in one configuration or another may or may not complete all the way to a valid surface before shutting down, confusing some apps that expect each new launch to come all the way up to a fully working rendering surface and focus

3) Some of these "false starts" may launch all the way up to a full rendering surface with dimensions opposite the desired orientation before quickly shutting down again. Needless to say, these incorrectly-sized "false starts" can be very confusing to an application.

Handling Configuration Changes in a Lightweight Manner

The `AndroidManifest.xml` key:

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

specifies that the application wishes to manually support orientation change events. If this key is set for an application's Activity, then rather than "bounce" (shut down and restart) the application's Activity on each orientation change, the application will receive two forms of notification in quick succession:

1) The Activity's **`onConfigurationChanged`** callback (if overridden) will be called with the new configuration.

2) The Activity's Views (specifically their **`SurfaceView Holders`**) will receive **`surfaceChanged`** callbacks.

In practice, we find that many applications do not even need to supply overrides for **`onConfigurationChanged`** (although having one installed can be useful for debugging), as they can simply change their **`glViewport`** and any aspect-ratio-handling in the **`surfaceChanged`** callback.

This simple flag and **`surfaceChanged`** handling code can often allow applications that have the ability to render their UI, etc in portrait and landscape modes to easily support fast orientation change. This flag avoids having the app shut down and thus can avoid having the app lose or destroy its EGL Context. Since the EGL Context stays resident, there is no need to do expensive reloading of 3D resources.

With this flag, note that the application may still receive **`surfaceChanged`** callbacks for the previously mentioned "false starts" with incorrect surface sizes. However, since the application receives only these change messages and not full shutdown/restart, these unexpected (temporary) surfaces of the wrong size can often be quickly ignored.

However, applications setting this flag do need to be prepared for **`surfaceChanged`** callbacks that happen more frequently than just once per **`surfaceCreated`**/**`surfaceDestroyed`** pairing.

COMMON LIFECYCLE CASES THAT EXHIBIT CONFIGURATION ISSUES

The following sections describe common issues we see with lifecycle events causing unexpected configuration changes. In both cases, the answer is actually pretty simple: using the `configChanges` tag in the manifest, with or without forcing the orientation with the `screenOrientation` tag. However, we present the full sequences with and without the potential fix, so that developers can better understand the expected results.

Launch a Forced-Landscape Application with Device in Portrait Orientation

Many games and media applications force landscape orientation using the AndroidManifest.xml key:

```
android:screenOrientation="landscape"
```

And then assume that they will always be launched/resumed in the desired orientation with no anomalies or multiple launches. This is the case on some OS images and devices, which can lead to confusion on platforms that do not behave as simply. For example, on some devices, an application that forces landscape with the above key and does NOT declare that it wants to receive config changes explicitly through the **`onConfigurationChanged`** callback will see the following lifecycle callbacks when launched from portrait orientation:

```
I/System.out( 1355): +-onCreate
I/System.out( 1355): +-onStart
I/System.out( 1355): +-onResume
I/System.out( 1355): +-onPause
I/System.out( 1355): +-onStop
I/System.out( 1355): +-onDestroy
I/System.out( 1355): +-onCreate
I/System.out( 1355): +-onStart
I/System.out( 1355): +-onResume
I/System.out( 1355): +-surfaceCreated
I/System.out( 1355): +-surfaceChanged: 1366, 695
I/System.out( 1355): +-onWindowFocusChanged (TRUE)
```

Note that the system is brought all the way up to **onResume** before being shut down again. One answer is to simply add the aforementioned tag (`android:configChanges`) to handle the configuration change internally via a callback. In this case, the result is much simpler:

```
I/System.out( 1532): +-onCreate
I/System.out( 1532): +-onStart
I/System.out( 1532): +-onResume
I/System.out( 1532): +-onConfigurationChanged
I/System.out( 1532): +-surfaceCreated
I/System.out( 1532): +-surfaceChanged: 1366, 695
I/System.out( 1532): +-onWindowFocusChanged (TRUE)
```

For many 3D apps, there will not even be a need to override **onConfigurationChanged**. Note that the configuration changed callback comes before the creation of the window, and thus there is no problem for a 3D app, which bases its rendering buffer on the **surfaceCreated** and **surfaceChanged** callbacks. Merely the addition of the manifest key may be sufficient.

This can be more complex with applications that have advanced Android UI layouts using XML. But for many 3D applications, this is not the case, and thus there is no complication.

Sleep and Resume with a Fixed-Orientation Application

On some devices, we have seen particularly problematic sequences when putting a device to sleep and then waking it up again. The "expected" sequence (from the Android docs) is:

```
I/System.out(  437): +-onCreate
I/System.out(  437): +-onStart
I/System.out(  437): +-onResume
I/System.out(  437): +-onConfigurationChanged
I/System.out(  437): +-surfaceCreated
I/System.out(  437): +-surfaceChanged: 1366, 695
I/System.out(  437): +-onWindowFocusChanged (TRUE)
   ➔ Sleep
I/System.out(  437): +-onPause
   ➔ Resume
I/System.out(  437): +-onResume
I/System.out(  437): +-onWindowFocusChanged (FALSE)
   ➔ Unlock the Screen
I/System.out(  437): +-onWindowFocusChanged (TRUE)
```

This is not the sequence seen on some devices, however… With a forced layout (`screenOrientation`) and ***without*** the configuration changed callback tag `configChanges`, some devices will provide extremely "noisy" sequences involving multiple configuration changes:

➔ **Sleep**

```
I System.out: +-onPause
I System.out: +-onStop
I System.out: +-onDestroy
I System.out: +-surfaceDestroyed
I System.out: +-onCreate 0
I System.out: +-onStart
I System.out: +-onResume
I System.out: +-onPause
I System.out: +-surfaceCreated
I System.out: +-surfaceChanged: 540, 884
```

➔ **Resume**

```
I System.out: +-onResume
```

➔ **Unlock the Screen**

```
I System.out: +-onPause
I System.out: +-onStop
I System.out: +-onDestroy
I System.out: +-surfaceDestroyed
I System.out: +-onCreate 0
I System.out: +-onStart
I System.out: +-onResume
I System.out: +-onWindowFocusChanged (TRUE)
I System.out: +-surfaceCreated
I System.out: +-surfaceChanged: 960, 464
I System.out: +-onPause
I System.out: +-onStop
I System.out: +-onDestroy
I System.out: +-surfaceDestroyed
I System.out: +-onCreate 0
I System.out: +-onStart
I System.out: +-onResume
I System.out: +-onWindowFocusChanged (TRUE)
I System.out: +-surfaceCreated
I System.out: +-surfaceChanged: 540, 884
I System.out: +-onPause
I System.out: +-onStop
I System.out: +-onDestroy
I System.out: +-surfaceDestroyed
I System.out: +-onCreate 0
I System.out: +-onStart
I System.out: +-onResume
I System.out: +-onWindowFocusChanged (TRUE)
I System.out: +-surfaceCreated
I System.out: +-surfaceChanged: 960, 464
```

Quite a wild ride for the application, especially if they start loading 3D resources each time.

The solution is the same as the previous case: Adding the configuration callback tag `android:configChanges` causes the sequence on the problematic devices to simplify significantly. With these changes, the device still sends multiple screen-size changes, but they do not generate the problematic shutdown/startup "bounces":

➔ **Sleep**

```
I System.out: +-onPause
```

➔ **Resume**

```
I System.out: +-surfaceChanged: 540, 884
I System.out: +-onWindowFocusChanged (FALSE)
I System.out: +-onResume
```

➔ **Unlock the Screen**

```
I System.out: +-onWindowFocusChanged (TRUE)
I System.out: +-surfaceChanged: 960, 464
I System.out: +-surfaceChanged: 540, 884
I System.out: +-surfaceChanged: 960, 464
```

These resolution changes are much easier to handle quickly than a full shutdown, all by simply setting the flag to receive configuration changes for orientation.

Honeycomb and the Volume Button

In previous versions of Android, pressing the volume up/down button showed a volume "overlay" and changed the device volume. There was no lifecycle messaging to the application, as the slider could not be dragged – it was merely a visual representation of the hardware volume button presses. However, as of Honeycomb, that behavior has changed. The volume buttons launch a temporary dialog with one or more sliders that you can actually touch and interact with. As a result, if you do not "eat" the volume button events and change the volume yourself, you will receive a focus lost/gained pair of events. Focus will be lost on the first press of a volume key when the dialog is launched, and focus will be regained when the volume "slider" dialog times out an fades away.

This can be confusing for some apps, as the behavior differs per OS. Workaround options include:

- Handling the volume buttons yourself in your game code and "eating" the events, thus keeping the case from ever happening.
- Allowing the app to be auto-paused and requiring the user to un-pause when they are done.
- Making all cases in your app where focus is lost and regained while spending the entire time resumed (i.e. not the HOME or Suspend cases, where the app is also **onPause'd**) pause the game only for the duration of the loss of focus. In other words, in the case where **onWindowFocusChanged** is called with **FALSE** and then **TRUE**, without any call to onPause happening before or during the loss of focus. The

issue here is that this will cause your game to immediately restart in gameplay when the user throws away a settings dialog, etc that caused the focus loss. This can be jarring to the user as they have to rush to catch up with the active game.

NOTICES