Two ANativeWindow + Shared EGL Contexts/GL Buffers + Blitting from one to another has inconsistent behavior

Assigned Bug P2 + Add Hotlist adexe s nau

STATUS UPDATE No update yet. Edit

DESCRIPTION jo...@unity3d.com created issue #1 Apr 4, 2020 09:18AM

The goal was to perform rendering of different animating images to different hardware displays for an XR-type application.

The approach that was taken was to have the launch Activity create it's own EGL context to render an image with GLES 3.2 to a framebuffer. Both the context and the color buffer were to be shared with a second activity so that OpenGL shared contexts could be constructed. Once the contexts had shared memory, we could easily blit different regions of the color buffer to each Activity surface all on the GPU for lowest latency.

The activities that were launched were using NativeActivity/android_native_app_glue as a Java/C++ intermediary, and all of the operative render code was written in C++.

Due to the nature of NativeActivity/android_native_app_glue, we had to make the build both into a separate *.so to avoid name collisions with JNI. In addition to this, each were running in a separate thread due to what android_native_app_glue does under the hood. However, since we were building the activities to share memory, and to get around the aforementioned limitations, we built them to share `extern` variables. These extern variables share the EGL context of the launch Activity, and the GL color buffer. That way, once the second activity detects that the two GL identifiers are non-zero, it could then unblock and construct the second EGL context.

There are a few issues that are happening now.

The first is that functionality is inconsistent intermittently in builds. When behavior is incorrect, it will either continue to show a mirror image in the second screen, or it will show a black screen, or will show the first blit frame locked.

Interestingly, we've noticed that if we use a build machine that produces a build with the correct behavior, it tends to produce it for all devices. Another machine with a very similar setup might tend towards producing incorrect behavior. If we deploy from machines that produce correct builds onto the same devices did not run correctly, the run correctly and vice versa, as if it matters which build machine it came from.

Even more peculiar is that the correct/incorrect behavior for a particular build machine appears to change in about a week's time, but in different ways. A few weeks back, I was experiencing the correct image, but the second image would lock-up and stop blitting if the first activity received touch controls. The week before that, incorrect behaviour meant that it was only blitting the first frame.

This varying behavior, including sanity checks such as regressing back to points where behavior was different and still getting the new behavior, has suggested a threading issue somewhere.

We've implemented OpenGL non-CPU blocking synch fences to try and rule out thread access issues, but even then, one thread is only writing to the color buffer and the other is only reading, so it would make sense that they wouldn't step on each other's toes.

We also considered there could be issues with putting Activities on the same stack on different hardware screens, but have made tests that suggest that the NativeActivities are not thread blocking each other unless the second is waiting for the first's shared data.

Other issues that we're seeing are more consistently reproducible, but fall into similar categories. These are more obviously tied to the activity lifecycle. If the image is created and blitting properly, then we pause the task, the second activity's animating image locks up. This should makes sense because you'd expect that if launch activity is paused, which is generating the image, then the second activity wouldn't have a source of new images to replace on its surface. However, the porblem starts when we resume the application, the first activity is able to resume, but the second activity stays locked up. From what I can tell from digging through source, when android_native_app_glue detects pausing/resuming, it signals a wait condition on the thread. It may be that the thread is seizing up, but it could even be an issue with EGL shared contexts.

- We've been building on MacOS Catalina 10.15.3 and 10.15.4
- Android Studio 3.6.1 (Feb 26, 2020 build)
- Targeting OpenGL ES 3.2 and EGL 1.5
- Gradle 5.6.4
- SDK is 28
- NDK is 20.0.5594570
- JDK is 8u152

- Main device is a Samsing Galaxy S9+ SM-G965U1
- Android 10 Build QP1A.190711.020.G965U1UES7DTB2
- Feb 1, 2020 Patch

Links (1) Hide all

"https://developer.android.com/reference/android/app/NativeActivity?hl=en#MET... " jr...@ #4

Reporter	jo...@unity3d.com
Type	Bug
Priority	P2
Severity	S2
Status	Assigned
Access	Default access View
Assignee	jo...@unity3d.com
Verifier	--
Collaborators	
CC	id...@google.com jo...@unity3d.com nd...@google.com
AOSP ID	--
ReportedBy	--
Found In	--
Targeted To	--
Verified In	--
In Prod	

COMMENTS

All comments ↓ Oldest first

en...@google.com <en...@google.com> #2 Apr 4, 2020 10:25AM

Reassigned to jr...@google.com.

jreck: are you the right person to be talking to about shared GL contexts?

---

jr...@google.com <jr...@google.com> #3 Apr 4, 2020 10:37AM

> Due to the nature of NativeActivity/android_native_app_glue, we had to make the build both into a separate *.so to avoid name collisions with JNI. In addition to this, each were running in a separate thread due to what android_native_app_glue does under the hood. However, since we were building the activities to share memory, and to get around the aforementioned limitations, we built them to share `extern` variables. These extern variables share the EGL context of the launch Activity, and the GL color buffer. That way, once the second activity detects that the two GL identifiers are non-zero, it could then unblock and construct the second EGL context.

I don't understand. Why are you using multiple .so's? How do you have name collisions?

> functionality is inconsistent intermittently in builds

If it's varying based off of *build* it sounds like you have a build problem, not a framework problem. This is probably related to the above name collision. *.so's are not really isolated, so you could still have that fundamental collision and it's just a coinflip how it gets resolved instead of being a clear build error.

I'd recommend as a first step getting this to compile as a single .so. There shouldn't be anything about NativeActivity that prevents that, and android_native_app_glue looks like it's just a static library that's bundled into your build so you should be able to resolve that dependency however you need.

---

jr...@google.com <jr...@google.com> #4 Apr 4, 2020 10:48AM

Reassigned to jo...@unity3d.com.

To elaborate on "There shouldn't be anything about NativeActivity that prevents that" the specific thing is NativeActivity allows specifying a different function name for the entry point:

https://developer.android.com/reference/android/app/NativeActivity?hl=en#META_DATA_FUNC_NAME

So you should be able to cleanly have multiple NativeActivities all provided by a single .so, just give them different entry-point names. I'm not familiar with android_native_app_glue, but it looks like it's just a small helper library. So I'd suggest just forking it to fit your needs. It's not a platform API and doesn't use any platform internals, so there's nothing special about it.