Investigation of three different techniques of Android native API access from Unity

Unity is a multi-platform gaming engine for developing gaming apps for mobile, desktop and web environments. With different Virtual and Augmented reality software libraries, it has entered the world of VR/AR apps using device's camera by turning it to VR/AR Camera. Power of Unity gives developers the access to write the code once and publish application end product for multiple platforms. It uses its own built-in UI components, C#, and JavaScript scripting for achieving cross-platform compatbility. When Unity is used for Android e.g. gaming, usually developers do not need to call Android native API/Java API from their apps. Since, Unity gives simple access to device properties, vibration, screen orientation, anti-piracy check, activity indicator and so on. However, there are some situations where Unity developers should call those Android native APIs (specifically Java). For example, in VR/AR apps, developers may use device's NFC reader to interact with toy or physical object such as reading NFC tags from those objects, or using other device sensors, or launching Android OS's chooser Intent to share information generated from Unity app, etc.

We have investigated three different techniques to achieve native API access. Firstly, we tried using only JNI (Java Native Interface) calls to Android native APIs. As seen from its name, C# JNI API provides developers with an interface to access Android Java APIs. Secondly, there is an option to export Unity Project as an Android Project, performing Java API calls in the exported project and generating final .APK (Android Package) app installation file. Finally, we have developed a custom Java plug-in, in which we have written required Java code snippets and called those snippets from C# script using few lines of JNI calls. In this report, we considered advantages and disadvantages of those techniques with respect to one other in terms of performance, stability, implementation time, maintenance possibility, code clarity and other perspectives.

Technique 1: Using only JNI Calls or higher level methods for android native API access from Unity

To interact with Android native OS components, Unity provides API which acts out as bridge between Unity's C# API and Android Java API. For example, the following Java code snippet creates an instance of java.lang.String initialized with a "string", and retrieves the hash value for that string.

```
java.lang.String someString = new java.lang.String("string");
int hashCode = someString.hashCode();
```

It can be converted to equivalent C# snippet using the following:

```
AndroidJavaObject jo = new AndroidJavaObject("java.lang.String", "string");
int hashCode = jo.Call<int>("hashCode");
```

There are other ways achieving the same thing above such as using raw JNI through the AndroidJNI methods and AndroidJNIHelper class together with Android JNI in C# scripts. The way we used above is the most-convenient higher-level API constructed with AndroidJavaObject and AndroidJavaClass classes. The AndroidJavaObject and AndroidJavaClass Unity API classes automate a lot of tasks when using JNI calls, and they also use caching to make calls to Java faster. The combination of AndroidJavaObject and AndroidJavaClass is built on top of AndroidJNI and AndroidJNIHelper, but

also has some additional functionality. These classes also have static methods that are used to access static members of Java classes. Using AndroidJavaObject and AndroidJavaClass is not only decreasing the code implementation time, but also helping for code clarity with few lines of calls. We only consider this higher-level APIs in this report.

The following example starts with AndroidJavaClass instead of AndroidJavaObject in order to access a static member of com.unity3d.player.UnityPlayer rather than to create a new object. Then the static field "currentActivity" is accessed but this time AndroidJavaObject is used as the generic parameter. This is because the actual field type android.app.Activity is a subclass of java.lang.Object, and any non-primitive type must be accessed as AndroidJavaObject. The exceptions to this rule are strings, which are accessed directly even though they don't represent a primitive type in Java.

The method getCacheDir() can then be called on the Activity object to get the File object representing the cache directory, getCanonicalPath() can then be called to get a string representation.

```
AndroidJavaClass unityPlayerClass
                 = new AndroidJavaClass("com.unity3d.player.UnityPlayer");

AndroidJavaObject currentActivityObject
        = unityPlayerClass.GetStatic<AndroidJavaObject>("currentActivity");

Debug.Log(currentActivityObject.Call <AndroidJavaObject>("getCacheDir")
                        .Call<string>("getCanonicalPath"));
```

Let's recapitulate some of methods which are present in AndroidJavaClass and AndroidJavaObject classes. To access static members of the specific Java class, AndroidJavaClass is used to construct C# interface from the class name. AndroidJavaClass is the Unity representation of a generic instance of java.lang.Class. On the other hand, AndroidJavaObject is the Unity representation of a generic instance of java.lang.Object. It can be used as type-less interface to an instance of any Java class. Public methods:

Call	Calls a Java method on an object (non-static).
CallStatic	Call a static Java method on a class.
Dispose	IDisposable callback.
Get	Get the value of a field in an object (non-static).
GetRawClass	Retrieves the raw jclass pointer to the Java class.Note: Using raw JNI functions requires advanced knowledge of the Android Java Native Interface (JNI).
GetRawObject	Retrieves the raw jobject pointer to the Java object.Note: Using raw JNI functions requires advanced knowledge of the Android Java Native Interface (JNI).
GetStatic	Get the value of a static field in an object type.

Set Set the value of a field in an object (non-static).

SetStatic Set the value of a static field in an object type.

Instances of UnityEngine.AndroidJavaObject and UnityEngine.AndroidJavaClass have a one-to-one mapping to an instance of java.lang.Object and java.lang.Class (or their subclasses) on the Java side, respectively. They essentially provide 3 types of interaction with the Java side:

- Call a method
- Get the value of a field
- Set the value of a field

The call is separated into two categories: A call to a 'void' method, and call to a method with non-void return type. A generic type is used to represent the return type of those methods which return a non-void type. The Get and Set always take a generic type representing the field type.

When AndroidJavaClass is used, developers should make use of static Java method counterparts such as CallStatic, GetStatic, SetStatic methods. For example, com.unity3d.player.UnityPlayer.currentActivity call can be achieved with AndroidJavaClass as shown above. For accesing Java public instance methods and public instance variables defined in java.lang.Class such as **public String getName()** or **public T cast(Object obj)**, one should make use of Call and Get methods above. The important thing is not to confuse CallStatic with GetStatic and Call with Get, since GetStatic and Get refers to fields in the class; static public class variables and public instance variables respectively. On the other hand, CallStatic and Call is used to invoke respective methods only. It should be noted that, e.g. calling Java public static method with GetStatic will not have any effect on Unity side.

AndroidJavaObject's use case refers to usage of Java objects. For example, **currentActivityObject.Call <AndroidJavaObject>("getCacheDir")** as shown in above examples. The utilization of Get and Call is the same as in AndroidJavaClass, since they will request specific Java object's public instance variables and public instance methods accordingly. GetStatic and CallStatic are not so relevant, because they will just access public static variables and public static methods through object's dependence to a specific class. For example, if there is a public static method go() in Java class Foo, then it will achieve the same effect of calling Foo.go() through new Foo().go(), because new Foo() is an instance of Java class Foo. Again, difference between GetStatic, Get, CallStatic and Call should be handled carefully. GetStatic and Get methods should not necessarily call getter methods in Java API, since they are reserved for fields defined in Java classes.

It is obvious that in simple tasks such as accessing application cache directory's canonical path, JNI only implementation in C# scripts will satisfy the needs. But, for complex tasks such as reading NFC tags'NDEF formatted text information natively from Android device, a developer should put more effort and invest more time to complete them.

Instances of AndroidJavaClass and AndroidJavaObject are using a lot of resources; those objects should be disposed as soon as they are out of use for performance and stability reasons. Both of these classes implement Unity's C# IDisposable interface and Dispose() method is available to call as soon as a developer are done with the objects. For example the below snippet should be,

```
AndroidJavaClass unityPlayerClass
            = new AndroidJavaClass("com.unity3d.player.UnityPlayer");

AndroidJavaObject currentActivityObject
 = unityPlayerClass.GetStatic<AndroidJavaObject>("currentActivity");

Debug.Log(currentActivityObject
            .Call <AndroidJavaObject>("getCacheDir"));
```

followed by the following snippet,

```
//dispose objects
unityPlayerClass.Dispose();
currentActivityObject.Dispose();
```

The shorter way of doing this would be to utilize C# using keyword which guarantees objects will be disposed as soon as they are not required. More concise equivalent,

```
using (AndroidJavaClass unityPlayerClass
      = new AndroidJavaClass("com.unity3d.player.UnityPlayer"))
{
    using(AndroidJavaObject currentActivityObject
      = unityPlayerClass.GetStatic<AndroidJavaObject>("currentActivity"))
    {

      Debug.Log(currentActivityObject
            .Call <AndroidJavaObject>("getCacheDir"));
    } // using

} // using
```

Technique 2: Export of Unity Project as an Android Project, performing Java API calls in the exported project and generating final .APK file

The second technique we have tested is to export Unity project to Android Studio. The purpose of this technique is to reference future methods in Unity through JNI calls, write the respective code in exported project in Android Studio and generate final APK. As of Unity version 5.6.1f1, it supports three Android build systems: **Gradle, ADT** and **Internal.** Unity exports projects in two formats: **ADT** (Google Android project) and **Gradle**. The **Internal** build system creates an APK by invoking Android SDK utilities in a specific order. In this build system, an APK is automatically deployed to a device if a developer selects **Build and Run** right from Unity Editor.

The Gradle build system uses Gradle to build an APK or export a project in Gradle format, which can then be imported to Android Studio. When a developer select this build system, Unity goes through the same steps as the Internal build system excluding resource compilation with AAPT and running DEX. Unity then generates the *build.gradle* file (along with the other required configuration files). After import to Android Studio, Android Studio invokes the Gradle executable, passing it the task name and the working directory. The APK is then built by Gradle there.

Imported project in Android Studio will have the same structure as native Android applications. In a simple case, the developer will see there activities namely UnityPlayerActivity, UnityPlayerNativeActivity and UnityPlayerProxyActivity. This activity classes are generated as of Unity version 5.6.1f1 and Android Studio projects show that UnityPlayerNativeActivity and UnityPlayerProxyActivity are deprecated. Therefore, MAIN and LAUNCHER activity should be defined as UnityPlayerActivity in the manifest. Therefore, the methods or variables that will be accessed from Unity (usually future methods and/or variables are already referenced through JNI calls before their original implementation in Android Studio) are placed in UnityPlayerActivity. For example, if boolean check of NFC sensor‘s enabled/disabled state is implemented using only JNI calls, it will look as follows:

```
//helper method to detect whether the NFC is enabled/disabled
public bool IsNfcEnabled()
{
    AndroidJavaClass unityPlayerClass
      = new AndroidJavaClass("com.unity3d.player.UnityPlayer");

    AndroidJavaObject currentActivityObject
      = unityPlayerClass.GetStatic<AndroidJavaObject>("currentActivity");

    AndroidJavaClass contextClass
      = new AndroidJavaClass ("android.content.Context");

    //context.getSystemService() returns java.lang.Object
    AndroidJavaObject plainObject
      = currentActivityObject.Call <AndroidJavaObject>("getSystemService",
        contextClass.GetStatic<string>("NFC_SERVICE"));

   //perform java casting using Utils class. Cast from Object to NfcManager
    AndroidJavaObject nfcManagerObject
      = Utils.Cast(plainObject, "android.nfc.NfcManager");

   //get default adapter of NfcManager
   AndroidJavaObject nfcAdapterObject
      = nfcManagerObject.Call<AndroidJavaObject>("getDefaultAdapter");

   if (nfcAdapterObject != null && nfcAdapterObject.Call<bool> ("isEnabled"))
   {
            //dispose all objects
            unityPlayerClass.Dispose();
            currentActivityObject.Dispose ();
            contextClass.Dispose ();
            plainObject.Dispose ();
            nfcManagerObject.Dispose ();
            nfcAdapterObject.Dispose ();
            return true;
    } // if
    else
    {
            //dispose all objects
            unityPlayerClass.Dispose();
            currentActivityObject.Dispose ();
            contextClass.Dispose ();
            plainObject.Dispose ();
            nfcManagerObject.Dispose ();
            nfcAdapterObject.Dispose ();
            return false;
     } // else

} // isNfcEnabled
```

The above method exposes a lot of resource-heavy objects which is not optimal in terms of performance and implemetation effort/time. However, if one can reference future Java method from Unity, the above method could be simplified as follows:

```
//helper method to detect whether the NFC is enabled/disabled
public bool IsNfcEnabled()
{
   using (AndroidJavaClass unity
      = new AndroidJavaClass ("com.unity3d.player.UnityPlayer"))
   {
      //currentActivity is UnityPlayerActivity exported to Andr. Studio
     using(AndroidJavaObject currentActivityObject =
                unity.GetStatic<AndroidJavaObject>("currentActivity"))
     {
       //call a method that will be defined in An. Studio after export
       return currentActivityObject.Call<bool>("isNfcEnabled");

     } // using
   } // using

} // isNfcEnabled
```

The respective method that will be called by the above method is inside UnityPlayerActivity Java class residing in exported project and looks like this:

```
public boolean isNfcEnabled()
{
    NfcManager manager = (NfcManager) getSystemService(Context.NFC_SERVICE);
    NfcAdapter adapter = manager.getDefaultAdapter();
    if (adapter != null && adapter.isEnabled()) {
        return true;
    } // if
    else
    {
        return false;
    } // else

} // isNfcEnabled
```

It is clear that, through this technique a developer can effectively reduce the complexity required in JNI only call implemetations. Having written methods on Java side is also effective in terms of code clarity where few lines of code will be needed for execution in C# script.

Apart from many advantages of this technique, there is one important disadvantage. According to our implementations, as compared to Android apps (.apk files) exported right from Unity, exporting Unity Android Project from Android Studio to an .apk file can affect original size and positions of objects defined in Unity scenes which is an unwanted behaviour.

Technique 3: Writing custom Java plug-in project, migrating all Java related code there, and calling optimal/shortcut methods from C# scripts with few lines of JNI calls

The last technique we have tested is to write custom Java Plug-in project and import it as .jar or .aar file to Unity project. In this project a developer can write optimal/shortcut methods using Java and call them from C# scripts using few lines of JNI calls. For this technique we look up JAR plug-ins and AAR plug-ins in a comparable manner.

JAR (Java Archive) plug-ins are primarily used to enable interaction with the Android OS or to call methods written in Java from within C# scripts. They can only contain Java code (for example, they can't contain Android resources), which makes their use very limited. Using Java plug-ins Unity uses the Java Native Interface (JNI) both when calling code from Java and when interacting with Java or the Java VM (Virtual Machine) from native code or C# scripts.

Android Archive (AAR) plug-ins are bundles that include compiled Java and native (C/C++) code, resources, and an Android Manifest. The .aar file itself is a zip archive which contains all of the Assets. Plug-in project's AndroidManifest.xml gets automatically merged with the Unity's main manifest file when the project is built. Per Unity documentation, AAR is the recommended plug-in format for Unity Android applications.

Before exporting plug-in project as .jar or .aar file, a developer should consider extending UnityPlayerActivity to insert the required Java methods there.

When developing a Unity Android application, it is possible to extend the standard UnityPlayerActivity class (the primary Java class for the Unity Player on Android) by using plug-ins. An application can override any and all of the basic interaction between the Android OS and the Unity Android application.

Two steps are required to override the default activity:

- Create the new Activity which derives from UnityPlayerActivity;
- Modify the Android Manifest to have the new activity as the application's entry point.

To make a plug-in with the new activity code and add it to the Unity project a developer must perform the following steps:

1. Extend the UnityPlayerActivity. The UnityPlayerActivity.java file is found at */Applications/Unity/Unity.app/Contents/PlaybackEngines/AndroidPlayer/src/com/unity3d/player* on Mac and *C:\Program Files\Unity\Editor\Data\PlaybackEngines\AndroidPlayer\src\com\unity3d\player* on Windows. To extend the UnityPlayerActivity one must locate the classes.jar included with Unity. It is found in the installation folder (usually C:\Program Files\Unity\Editor\Data (on Windows) or /Applications/Unity (on Mac)) in a subfolder called PlaybackEngines/AndroidPlayer/Variations/mono or il2cpp/Development or Release/Classes/. Then classes.jar should be added to classpath to compile the new Activity. After compilation of the Activity source file and it should be packaged into a JAR or AAR package, and copied into Unity project folder such as Assets/Plugins/Android.
2. Create a new or modify an existing Android Manifest to set the new activity as the entry point of the application. Place or update the AndroidManifest.xml file in the Assets/Plugins/Android folder of Unity project.

The following is an example of new Activity file:

```
package com.denkwerk.nfcplugin;

import android.content.Context;
import android.nfc.NfcAdapter;
import android.nfc.NfcManager;
import android.os.Bundle;

import com.unity3d.player.UnityPlayerActivity;

public class NFCPluginTest extends UnityPlayerActivity
{
    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
        // call UnityPlayerActivity.onCreate()
        super.onCreate(savedInstanceState);
    } // onCreate

    //helper method to check NFC state
    public boolean isNfcEnabled()
    {
        NfcManager manager
                = (NfcManager) getSystemService(Context.NFC_SERVICE);
        NfcAdapter adapter = manager.getDefaultAdapter();
        if (adapter != null && adapter.isEnabled()) {
            return true;
        } // if
        else
        {
            return false;
        } // else
    } // isNfcEnabled

} // class NFCPluginTest
```

Then Unity Manifest XML file and plug-in Project Manifest XML file should be updated as follows:

```
<activity
    android:name="com.denkwerk.nfcplugin.NFCPluginTest"
    android:label="@string/app_name"
    android:configChanges
      ="fontScale|keyboard|keyboardHidden|locale|mnc|mcc|
        navigation|orientation|screenLayout|screenSize|
        smallestScreenSize|uiMode|touchscreen" >

    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>

    <meta-data android:name="unityplayer.UnityActivity"
        android:value="true" />

    <meta-data
        android:name="unityplayer.ForwardNativeEventsToDalvik"
        android:value="false" />

</activity>
```

The above xml snippet is used instead of original com.unity3d.player.UnityPlayerNativeActivity or com.unity3d.player.UnityPlayerActivity activity element in the manifest:

```
<activity android:name="com.unity3d.player.UnityPlayerNativeActivity"
    android:label="@string/app_name" >

    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>

    <meta-data
        android:name="unityplayer.UnityActivity"
        android:value="true" />

    <meta-data
        android:name="unityplayer.ForwardNativeEventsToDalvik"
        android:value="false" />

</activity>
```

The correspondence between two projects' manifests is essential; otherwise a developer can face manifest merge issues e.g. in .aar export of plug-in project.

Above two snippets are enough for JAR plug-in, since .jar file does not contain Android specific files such as Android Manifest and isNfcEnabled() method can be easily called using the way presented in Technique 2. For .jar export, if plug-in project uses other JAR libraries through its libs folder and/or through Gradle dependencies (Android Studio), they should be exported along with plug-in project's .jar file; otherwise a developer should import those dependent libraries again in Unity e.g. in Assets/Plugins/Android folder. Therefore, it is optimal to generate fat JAR (jar with all of its dependencies) of the plug-in project for convenience. This all-in-one fat archive generation should also be performed for .aar files.

For .aar export, along with fat AAR (all dependencies have to be in the libs folder of .aar file), there are some tweaks that developer should perform. First case is about manifest merging, since plug-in project and Unity project's manifests will be merged during the built. If there are some inconsistencies between two manifests, Unity project will not able to be built. There are some important facts to consider during manifest merge operation to avoid merge errors:

1. Both projects' manifest elements' package sub-element have to be the same.
2. Requested permissions and features have to be the same.
3. minSdkVersion and targetSdkVersion should be consistent in both manifests.
4. Plug-in project should avoid using AppCompat support library, because Unity activities are derived from android.app.Activity and during build, support library compilation are discarded.
5. It is optimal to set application's theme as android:theme="@android:style/Theme.NoTitleBar.Fullscreen" in both projects which is Unity's default.
6. References to resources and assets in plug-in project's manifest file should be removed. According to Unity documentation; Unity treats any subfolder of

Assets/Plugins/Android as a potential Android Library, and disables Asset importing from within these subfolders. If a developer needs to add Assets to Unity application that should be copied unchanged into the output package, they should be imported into the Assets/Plugins/Android/assets directory. They appear in the assets/ directory of the final APK, and are accessed by using the getAssets() Android API from your Java code. This is the same for resources using the method getResources().

7. To build and generate .aar or .jar file, plug-in project has to have Unity's classes.jar as a dependency in its libs folder (unless given through Gradle dependencies). However, having this jar existed in final fat JAR or fat AAR will cause collision with Unity, because Unity is also compiling this jar during build. Therefore, during .jar or .aar export, this jar should be excluded in Gradle build script.
8. Probably, the easiest way of having the consistent manifest file with plug-in project is to copy-paste the content of Unity's manifest file there, and make necessary changes and copy-paste to Unity again.

Higher compression rate of project's fat AAR (without assets and resources) makes it has less size as compared to corresponding fat JAR, however after final APK is generated, the size of the application will be the same. AAR files with resources will increase the size of final APK file. Therefore, if .aar export is used and resources/assets are not necessary, it is convenient to get rid of them in the plug-in project where fat JAR should do the job as well. The importance of AAR file shines when there is a need for resources and assets.

Unity also gives an opportunity to pass data from Java to Unity using UnitySendMessage method:

```
using UnityEngine;

public class NewBehaviourScript : MonoBehaviour
{
   void Start ()
   {
      AndroidJNIHelper.debug = true;
      using (AndroidJavaClass jc
            = new AndroidJavaClass("com.unity3d.player.UnityPlayer"))
       {
         jc.CallStatic("UnitySendMessage", "Main Camera",
                        "JavaMessage", "NewMessage");

       } // using
    } //Start

    void JavaMessage(string message)
    {
       Debug.Log("message from java: " + message);

    } // JavaMessage

} // class NewBehaviourScript
```

The Java class com.unity3d.player.UnityPlayer has a static method UnitySendMessage. It is used in Java to pass data to Unity. Although UnitySendMessage is called from within Unity, it relays the message using Java, and then Java calls back to the native/Unity code to deliver the message to the object named “Main Camera”. This object has a script attached which contains a method called JavaMessage.

Conclusion

After careful comparison, it is obvious that using JNI only calls to Java APIs is not the optimal technique in case of accomplishing complex tasks. According to Unity documentation, AndroidJavaObject and AndroidJavaClass are computationally expensive methods (as are any methods that use raw JNI). It is convenient to keep the number of transitions between managed and native/Java code to a minimum, for better performance and also code clarity. On the other hand, the Mono garbage collector should release all created instances of AndroidJavaObject and AndroidJavaClass after use, but it is advisable to keep them in a using(){} statement to ensure they are deleted as soon as possible. Without this, a developer cannot be sure when they are being destroyed. If AndroidJNIHelper.debug is set to true, Unity’s debug output will show a record of the garbage collector’s activity.

For solving complex tasks, either export of Unity project should be done as in Technique 2 or Plug-in should be developed as in Technique 3. However, exporting Unity project to Android studio can sometimes affect pre-defined positions and sizes of objects in Unity scenes, therefore sounds not optimal. For the technique 2, additional research should be performed in which cases, original Unity project is affected. If plug-in project is considered, then it is better to use AAR plug-in in case of using plug-in project’s resources and assets; otherwise fat JAR plug-in would satisfy the needs.

References

https://docs.unity3d.com/Manual/AndroidJARPlugins.html

https://docs.unity3d.com/Manual/AndroidAARPlugins.html

https://docs.unity3d.com/Manual/AndroidUnityPlayerActivity.html