Android System for Unity
Version 1.4

Unity is a very extensible game engine, but lacks of good integration with native functionalities of the systems running on.

Specifically for Android, access to system information is very poor, as well as interaction with system apps and services.

Even worse, the only way to call C# methods from Java code requires developer to:

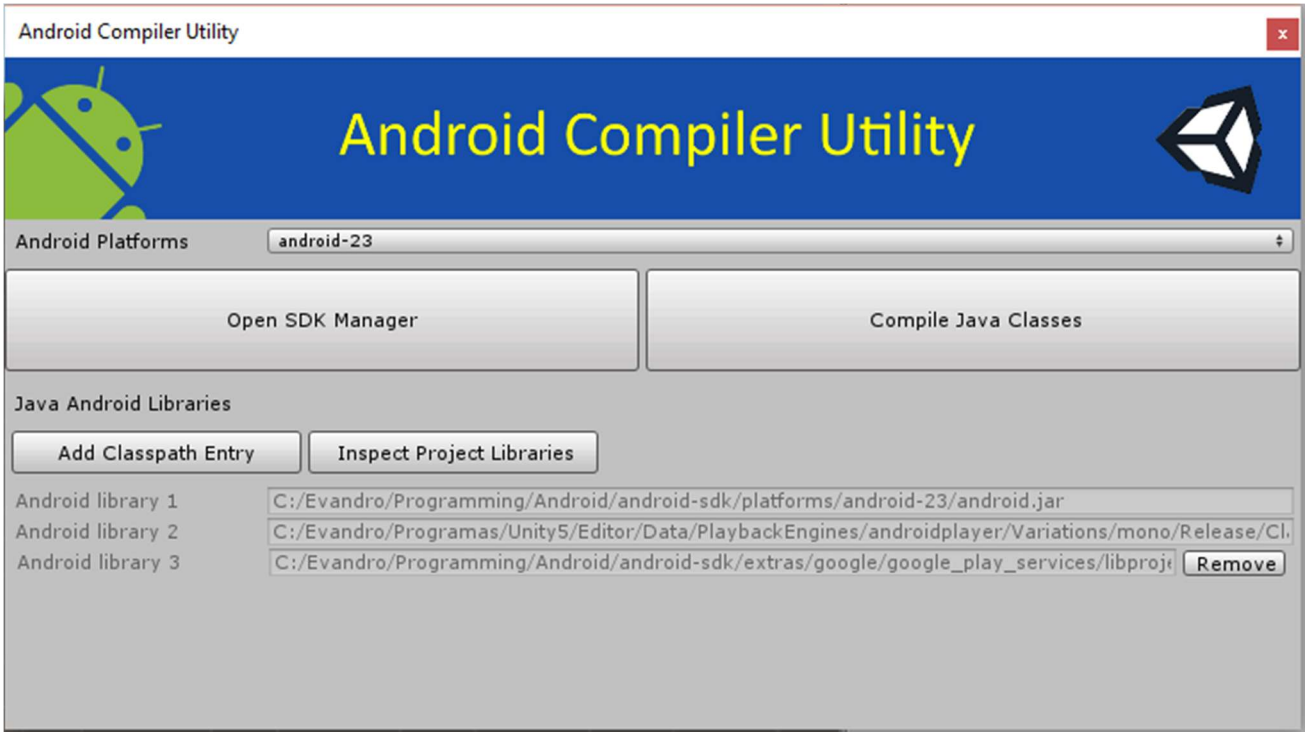
- Create and manage a library on Android side;
- Compress the data in a string to call **UnityPlugin.UnitySendMessage**;
- Parse the data received in Mono side.

This makes the developer to lose focus on your game and get worried about system functionalities.

Android System allows developers to interact with native Android functionalities without any Java code. You can listen events with delegates, parse the data as it comes and even start and interact with Android services directly.

This document is separated in two parts: the first explains the services exposed by this plugin, and the second presents the low-level features, that allows deeper interaction with Android OS.

Android Compiler Utility



The **Android Compiler Utility** allows developers to compile custom Java classes into JAR libraries, without need of export project to Eclipse. The utility window allows choosing the Android platform to compile classes against, and add other JAR libraries to classpath, including the main Android JAR library and the Unity JAR library.

All java files will be included in a single library, called **unityandroidsystem.jar**. The files created by developer must be inside **Assets/Plugins/Android/java** path, in a package structure like any Java project.

Compiling java code

The **Open SDK Manager** button will open the Android SDK Manager, to download new Android platforms and packages.

The **Compile Java Classes** button will compile all java files and pack them to the **unityandroidsystem.jar** file. The classes compiles with libraries included in classpath entries. The first entry is main Android library, and changes when Android platform change. The second is the Unity Android library, and are unchangeable.

To add new classpath entries, use the **Add Classpath Entry**. The JAR file could be inside the project or in any other path in developer's machine. You can remove the entry with **Remove** button, at the right of the path.

Preprocessor

The utility also contains an asset preprocessor, that intercepts changes in **.java** files, and try to compile the files.

Compile native code

The native sources are in the package too. However, there is not a native compiling tool. If you need to alter the source, we suggest using the **ndk-build** command from Android NDK package.

Create a new native library and configure the compiler requires understanding the NDK structure and libraries, as well knowledge of parameters available in **Android.mk** and **Application.mk** files inside this plugin. The utilities for native development is in the roadmap. If you need help to change current library or develop a new one, please send an e-mail to fourthskyinteractive@gmail.com.

Services

Telephony

The **Telephony** class is used to initiate phone calls from application, send SMS messages and pick any contact from phone's contacts.

To make a phone call, you must use the **MakeCall** method. The first parameter is the number to call, and the second is boolean **true** to call immediatly or **false** to just open the caller application (requires ***android.permission.CALL_PHONE*** in AndroidManifest.xml):

```
Telephony.MakeCall ("55550009", true );
```

To send SMS messages, use the **SendSMS** method (requires ***android.permission.SEND_SMS*** in AndroidManifest.xml):

```
Telephone.SendSMS (phoneNumber, message,
    (sentOK) => {
        if (sentOK)
            messageStatus = "SMS sent successfully";
        else
            messageStatus = "Failed to send SMS";
    },
    (deliveredOK) => {
```

```
            if (deliveredOK)
                messageStatus = "SMS delivered sccessfully";
            else
                messageStatus = "Failed to deliver SMS";
        }
);
```

The first argument is the phone number to send the SMS, the second is the text message to send.

The third and fourth parameters are optional. The third parameters is a delegate called when the message was sent, and the last parameter is a delegate called when the message was delivered to the recipient.

In the version 1.4, you can receive SMS on Unity code, using **ListenForSms** method (requires ***android.permission.RECEIVE_SMS*** in AndroidManifest.xml):

```
Telephony.ListenForSms ((msg) => {
     // Mostrar apenas a primeira mensagem
     messageStatus = "Receive SMS from " +
                      msg[0].OriginAddress +  ": " +
                      msg[0].MessageBody;
     }
);
```

Picker

The **Picker** class allows easy access to images, videos and other media types. It opens the gallery, allowing the user to choose the media object, and returns it through a callback. Also, you can take a picture from the native camera app, and return it to the application in the same way.

Below is and example that pick and image from the gallery and is assigned to some material as the main texture (requires ***android.permission.READ_EXTERNAL_STORAGE*** in AndroidManifest.xml):

```
Picker.PickImageFromGallery( (Texture2D tex) => {
                                obj.material.mainTexture = tex;
                              } );
```

Also, you can access the contact list, and return a **PhoneContact** instance, containing the name and phone of the contact. Remember that you need to add **READ_CONTACTS** permission to AndroidManfest.xml to access the internal Android's contact database.

Following is an example (requires ***android.permission.READ_CONTACTS*** in AndroidManifest.xml):

```
Picker.PickContact( (PhoneContact contact) => {
                    Telephony.MakeCall(contact.phoneNumber);
                     } );
```

In version 1.4, you can also pick an image from the camera as follows:

```
Picker.PickImageFromCamera( (Texture2D tex) => {
                              obj.material.mainTexture = tex;
                              } );
```

IMPORTANT: To use **Picker** service, you need to use the custom Android activities from the **unityandroidsystem.jar** in the plugin's package. To more information, follow the configuration advice from **OnActivityResult** section in this document.

NFC

This feature allows immediate sharing of small data only by aproaching two phones. It can be used to easily exchange connection info between two devices, and even exchange game itens like power-ups.

Before using NFC, the application need to check if the feature is supported with **NFC.Supported**. Also, you must add the permission ***android.permission.NFC*** to AndroidManifest.xml.

The service can publish messages as string or raw binary. To publish a string message, you must simply call:

```
NFC.Publish( "NFC test message" );
```

Or, to to publish messages with arbitrary data, use **Publish** method which receives an array of **Record** objects:

```
Record rec = Record.CreateText( "NFC test message" );
NFC.Publish( new Record[] { rec } );
```

The **Record** object has other static methods to create other message types from NFC standard, like binary, URI and Android Application Record.

And to receive messages, just register a listener calling **Subscribe** method, with a callback that receives and array of **Record** objects:

```
NFC.Subscribe( (Record[] records) => {
          nfcReceivedMessage = "NFC message received: " +
                               records[0].StringValue;
     }
);
```

In version 1.4, you can also listen for NFC tags. To do this, there's a new method **ListenNFCTag**:

```
NFC.ListenNFCTag( (Record[] records) => {
     string type    = records[0].TypeASCII;
     string tnf         = records[0].TypeNameFieldASCII;
     string payload = records[0].PayloadASCII;

     nfcReceivedMessage = "NFC tag " +
                          tnf +
                          " (type - " + type + "): " +
                          payload;
     }
);
```

Due to issues with Unity lifecycle, when **ListenNFCTag** is called, a dialog will be presented, waiting for interaction with a NFC tag.

IMPORTANT: To use **Picker** service, you need to use the custom Android activities from the **unityandroidsystem.jar** in the plugin's package. To more information, follow the configuration advice from **OnNewIntent** section in this document.

Low Level APIs

Broadcast Receiver

Many system events in Android, like device boot or a headset connection, are delivered through broadcast messages. Any application can register a receiver and handle the information passed, and so trigger an event based on the message. A list with some events that Android broadcasts is presented below:

- Device boot and shutdown;
- Battery level;
- Headset plugged and unplugged;
- Wi-fi networks found;
- Phone call or SMS received.

Following is a code to allow the application to receive the event of battery low:

```
public BroadcastReceiver receiver;
public void RegisterForBatteryEvent() {
    receiver = new BroadcastReceiver();
    receiver.OnReceive +=
                (context, intent) => {

          string action = intent.Call<string>("getAction");
                  if (BATTERY_LOW == action) {
                    // Quit application
                    Application.Quit();
                  }
                };

    receiver.Register(BroadcastActions.ACTION_BATTERY_LOW);
}
```

As broadcast receiver action runs in the application's main thread, it is important to do not run long operations on it. The ability to run broadcast actions in a background thread will be available in a future release.

In version 1.4, it's now possible to listen for application to listen broadcasts in background. If application is not active when broadcast is received, it will be stored. Following is an example of how to handle a broadcast received in background:

```
public BroadcastReceiver receiver;
AndroidJavaObject intent =
                BroadcastReceiver.GetNextBackgroundBroadcasts();
if (intent != null) {
     string action = intent.Call<string>("getAction");
     string message = intent.Call<string>("getStringExtra",
"message");
     broadcastMessage = "Background broadcast message (" +
                         action +
                         ") received: " +
                         message;
} else {
     broadcastMessage = "No pending background broadcast";
}
```

Also, you need to register the broadcast listener in AndroidManifest.xml, with the action to listen to. Following is an example of configuration:

```
<application>
     …
 <receiver android:name="">
  <intent-filter>
   <action android:name="com.android.vending.INSTALL_REFERRER" />
  </intent-filter>
 </receiver>
     …
</application>
```

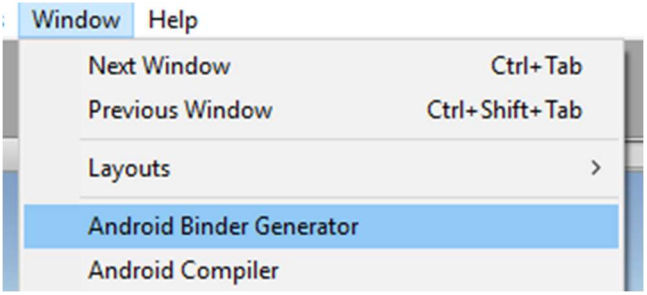
Many broadcast actions return some data, for example the battery level or name of connected headset. To get a list of available broadcast actions and data received in each action, see the list in Android Intent class documentation here. If you need more information, you can consult the documentation for BroadcastReceiver in this link.

Service Connection and Binder wrappers

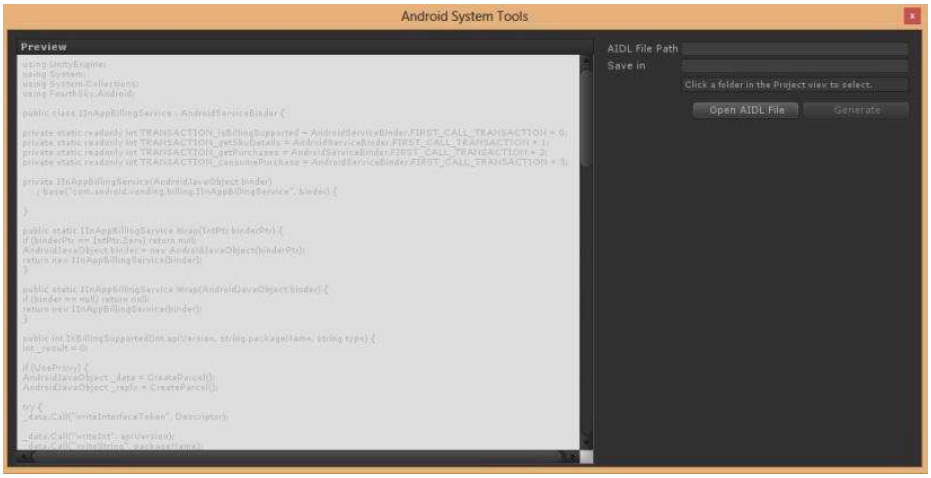
To interact with some Android services, first you have to establish a "connection" with it, and then use the object representing the connection, called Binder. The Binder calls operations on service, and receive responses from it.

To use services, first we have to create C# wrappers for binders, using the AIDL file that represents the interface to interact with the service.

In Unity, select **Window -> Android Binder Generator**:



Opening Android Binder Generator window, first you have to choose and AIDL file. Click in "Open AIDL file" button, and choose the **IInAppBillingService.aidl** file in **ANDROID_SDK/extras/google/play_billing/in-app-billing-v03** directory, inside Android SDK path (if this path does not exists, download using **SDK Manager**). The C# wrapper generated is showed in Preview pane:



Finally, choose a directory in Project view and click in **Generate**. The wrapper class will be generated, and can be edited if needed.

In case of any of the situations below, the **Generate** button is disabled:

- The wrapper is already generated;
- Another class with the same name of the wrapper exists in the project;
- Name of the wrapper is invalid;
- The target path is empty or invalid.

Following is an example that uses the wrapper generated from **IInAppBillingService.aidl** to check if the service is supported in the device:

```
// Action string
public static readonly string BILLING_ACTION =
          "com.android.vending.billing.InAppBillingService.BIND";
IInAppBillingService service = null;

public void BindBillingService() {
    // Create service connection
    ServiceConnection connection = new ServiceConnection();

    // Bind delegate methods
    connection.OnServiceConnected +=
          (AndroidJavaObject namePtr, AndroidJavaObject binder) {
               if (binderPtr == IntPtr.Zero) {
                    Debug.Log("Something's wrong");
               }

               // Wrap binder pointer with generated wrapper
               service = IInAppBillingService.Wrap(binder);
               Debug.Log("Billing service connected");

               // Check if service is supported
               int responseCode =
                         service.IsBillingSupported(3,
                                                    packageName,
                                                    "inapp");
               if (responseCode == 0)
                    Debug.Log("Billing service is supported");
               else
                    Debug.Log("Billing service is unsupported");
          };

    connection.OnServiceDisconnected +=
          (AndroidJavaObject name) {
               // Clear wrapper
               service.Dispose();
               service = null;

               Debug.Log("Billing service disconnected");
          } ;


    // Connect to service
    connection.Bind(BILLING_ACTION);
}
```

Custom Activities

This package provide some custom activities that needs to be used to enable services that uses **OnActivityResult** and **OnNewIntent**. This topic will learn how to configure your project to use these activities.

OnActivityResult

Sometimes you want to interact with another Android app, like camera, contacts or image gallery, pick an item from them, and return the chosen item to the calling application.

In Android, calls **startActivityForResult** from the Activity, specifying the application we want to get data. The return values are received in the implementation of **onActivityResult** method in the calling application.

The Android System plugin implements the same behaviour, ported to Unity C#. You can see an example in **Picker** service. The service calls the gallery (or contacts application, if picking a contact), and returns the choosen image as a **Texture2D** object.

To enable application to receive **OnActivityResult** calls, the developer needs to configure **AndroidManifest.xml** (or use the one from plugin package) to use the custom Unity Activities created for **Android System.**

OnNewIntent

This is a very particular callback, and their uses is better explained in Android documentation. Currently, the only use for this callback is receiving messages using the **NFC** service.

To enable application to receive **OnNewIntent** calls, the developer needs to include in **AndroidManifest.xml** (or use the one from plugin package) the **UnityNFCActivityReceiver** activity. The reason for this is that the flow for NFC and onNewIntent activity method does not works with **Unity3D** lifecycle, so the need to handle this callback in another activity, and then returns fast to application.

Roadmap

We have some great ideas for the future releases, some of them include:

- A tool to create customized AndroidManifest.xml, to add permissions and other configurations;
- A generator to wrap any Android class or interface and call their methods in C#;
- Creating and handling status bar notifications in Mono C#.

Changes in document

1.4

- **NFC** now supports tags reading
- Pick images from camera with Picker (NEW!!!)
- Listen Broadcasts in background
- Added API and utility to compile Java classes and native files
- Source code of the Java for this plugin is provided, and can be compiled with compiler included in this plugin

1.3

- All callbacks now returns instances of **AndroidJavaObject** instead of **IntPtr** pointers;
- In addition, the callbacks can be lambdas or instance methods.
- Added services: **Picker**, **NFC**, and **Telephony**;
- Added explanation about **OnNewIntent** callback

1.2

- **Editor classes now wrapped in DLL library, to allow execution even with some compilation errors**
- **AndroidSystem.ConstructJavaObjectFromPtr** – due to changes in AndroidJavaObject class since Unity 4.2, it's recommended (mandatory in 4.2) to use this method to create instances of AndroidJavaObject from IntPtr values