Using SWIG to talk to Java on Android (and get access to the Android API)

Summary

In this tutorial, we're going to use SWIG to generate Java wrappers for our C/C++ code and through that get access to the native Android functionalities. Even though Orx provides access to a great many platform functionalities that are commonly required by games, there's always just a few more things required by a game running on a device and we're going to tackle that here.

SWIG is a nice tool that generates wrappers for C/C++ code for a multitude of languages, please see the full list of supported languages in their website. In this tutorial, SWIG will do the arduous and error-prone task of writing JNI wrappers for us.

Getting to Know SWIG

Even though this is not a tutorial on SWIG per se, let's get familiar with it with an isolated example first. You don't need to perform the steps in this section, since we don't need them for our stated goal in this tutorial. Say we have a C++ class that we would like to use in another language:

MyClass.h

```
class MyClass {
public:
  void SayHello();
};
```

MyClass.cpp

```
#include <iostream>
#include "MyClass.h"
void MyClass::SayHello() {
  std::cout << "Hello From MyClass!" << std::endl;
}
```

In order to wrap this library, we need a SWIG interface definition file. SWIG's interface file syntax is a superset of C++. On top of C++ it lets you write some directives to fine-tune the interface generation process. For our `MyClass` class we're going to use the following file:

MyModule.i

```
%module MyModule
%{
#include "MyClass.h"
%}
%include "MyClass.h"
```

In `MyModule.i`, we're telling SWIG to generate a "module"(library, gem etc.) named `MyModule` in the target language. We're then telling it to include our `MyClass.h` file in two different ways; one with `#include` inside `%{` and `%}`. This `#include` statement will go directly into the generated bindings file. In fact, whatever's withing the `%{` and `%}` braces will go verbatim to be compiled as C/C++ code. This is needed so that the generated bindings actually include our class. The other include is specified with `%include`, and that is a directive for SWIG itself and it means that SWIG should parse `MyClass.h` and generate bindings for whatever it finds there.

Since this is an isolated example, let's go one step further and wrap this class for Python. Install SWIG for your platform, and make sure that you're able to call it from a command prompt, and call the following command in a terminal in the folder that you have the `MyClass` files:

```
swig -python -c++ MyModule.i
```

So, in order, we're telling swig to generate a Python module, for a C++ library using the interface definition in `MyModule.i`. Here's what it generates for us:

- `MyModule_wrap.cxx` : A C++ file that contains the bindings so that Python can call our functions
- `MyModule.py` : A Python module that exposes the bound C++ functions in a natural way

SWIG will generate different things for different languages, but the pattern is always the same: Some C++ code to bind functions, some target language code to expose a natural interface for the bound functions. In order to use the generated module, we first need to compile the generated C++ binding file. With g++ on Linux that would look like: (If you're using another compiler/platform you'll need to do something similar, but you don't have to follow this step in order to benefit from the tutorial, our aim isn't Python anyway :))

```
 g++ MyModule_wrap.cxx MyClass.cpp -I /usr/include/python2.7/ -fPIC -shared
-o _MyModule.so
```

So, we're telling g++ to compile our own C++ source (`MyClass.cpp`) along with SWIG generated bindings file (`MyModule_wrap.cxx`). `-I /usr/include/python2.7` is needed so that `MyModule_wrap.cxx` can find the python headers needed for the bindings. Finally, we're telling it to generate a dynamic library (and not an executable) with the arguments `-fPIC -shared` named `_MyModule.so`. The name of the generated library is important here, the Python generator for SWIG needs the compiled library to be called `_<ModuleName>.<Library extension of your platform>`.

Now, in the same folder, you can start a python session and issue:

```
import MyModule
myObject = MyModule.MyClass()
myObject.SayHello()
```

Output:

```
 Hello From MyClass!
```

Just for fun, if we call swig with `swig -java -c++ MyModule.i` swig will generate:

- `MyModule_wrap.cxx`: as before
- `MyModuleJNI.java`: a JNI file to expose the bindings to Java
- `MyModule.java`: a Java class to hold the C++ functions (we don't have any, we just have `MyClass`'s methods.)
- `MyClass.java`: to expose `MyClass` to Java

Now we would compile that with:

```
 g++ MyModule_wrap.cxx MyClass.cpp -I /usr/lib/jvm/java-8-oracle/include/ -I
/usr/lib/jvm/java-8-oracle/include/linux -fPIC -shared -o libMyModule.so
```

and happily use our new Java library. For a longer introduction to SWIG please head over to their their starting tutorial.

Defining Our Interaction with the Platform

Now that we're experts of SWIG, we can move onto our task of interfacing Orx with the native Android platform. We need to define such a scheme that from the point of view of Orx we'll be calling regular C/C++ functions, but those functions will actually be executing Java code on Android, and whatever on PC etc. Note that this is the inverse of what we have done in the previous section, we've called a C++ method from Python. Luckily, SWIG supports communication in that direction too. The way SWIG supports that is through allowing inheritance of C++ classes in the target Language. So, we're going to define a `Platform` class in C++, expose it through SWIG, and derive from it in Java to implement platform specific functionality. This way, we can also implement the same functionality on different platforms and expose them through the same interface.

In this tutorial, we're going to expose the vibration functionality of the Android device to Orx. For this, we define the following base class for the platform:

Platform.h

```
class Platform {
public:
  Platform(){};
  virtual ~Platform(){};
  virtual void ~Vibrate(int length_in_millisec) = 0;
};

void SetPlatform(Platform *);
Platform * GetPlatform();
```

Platform.cpp

```
#include "Platform.h"
Platform * platform; // a global platform pointer
void SetPlatform(Platform * p) {
  platform = p;
}
Platform * GetPlatform() {
  return platform;
}
```

The most important thing in the `Platform` class is the `Vibrate` method, which will allow us to invoke the vibration feature in our game **given a platform instance**. One part of the magic is how we get a platform object. When the game is running on a PC, we'll get the platform object that implements the functionality on PC, and on Android we'll get an instance of our SWIG generated Java-derivable awesome class.

The way we get different `Platform` objects on different platforms is through the `SetPlatform` and `GetPlatform` functions. During game initialization, a platform-specific `Platform` object will be injected using the `SetPlatform` function, and the game will call the methods of the `Platform` object obtained by calling the `GetPlatform` function.

We can now go ahead and implement the platform on PC, we don't need SWIG for that: (If you don't care at all about PC, you can just ignore this)

main.cpp

```
#include <iostream>
#include "Platform.h"

// ... Your awesome game

class PCPlatform: public Platform {
  void Vibrate(int length) {
    std::cout << "Bzzzzzz!!! Look this idiot is trying to vibrate a PC!" <<
std::endl;
  }
};

int main(int argc, char **argv) {
#ifdef COMPILING_FOR_PC
  SetPlatform(new PCPlatform);
#endif

  orx_Execute(argc, argv, Init, Run, Exit);
}
```

Notice the `COMPILING_FOR_PC` flag we use to check if we're running on PC. You need to make sure in your project configuration that this flag is defined while you're compiling for PC.

Now, anywhere in our game, we can just say `GetPlatform()→Vibrate(1000);` to invoke the vibration feature!

Implementing AndroidPlatform

Now, we're going to expose `Platform.h` using SWIG, so that we can derive from `Plaform` in Java and set an instance of that Java class as the `Platform` using `SetPlatform`. So, we prepare the following SWIG interface file:

PlatformModule.i

```
%module(directors="1") PlatformModule
%{
#include "Platform.h"
%}

%feature("director");

%include "Platform.h"
```

This is similar to our first interface file, but we've added the “director” feature. This is the feature that enables us to extend the `Platform` class in Java. As a side note, notice that we've called our module `PlatformModule`. What you call the module is not that important, but if the name of your module coincides with one of your classes, that causes a problem in Java, since the module itself is also exposed as a class.

Having this interface definition file, we call SWIG as follows:

```
swig -java -c++ -package platform PlatformModule.i
```

The only difference here is the `-package platform` argument, which tells SWIG to generate the Java classes with `package platform`. This means that we need to put them under a folder called `platform` in the Java source files path. These are the files generated by SWIG with that call:

- `PlatformModule_wrap.cxx`: The bindings file, we need to make sure that our `Android.mk` compiles this
- `PlatformModuleJNI.java`, `PlatformModule.java`, `Platform.java`: These should be available in a `platform` folder on the java source path.

From this point on, I'll take the Orx android-native demo application as a reference. If your project layout is different, please translate the instructions accordingly. In that project, you'd be putting `Platform.h`, `Platform.cpp` and `PlatformModule_wrap.cxx` in the `app/src/jni` folder, and `PlatformModuleJNI.java`, `PlatformModule.java` and `Platform.java` files in the `app/src/main/java/platform` folder. (You'll need to create the `platform` folder yourself)

Make sure that you modify your `Android.mk` so that `Platform.cpp` and `PlatformModule_wrap.cxx` both get compiled. And for the sake of this tutorial, you also need to add the vibration permission to your `app/src/main/AndroidManifest.xml` with the line `<uses-`

```
permission android:name="android.permission.VIBRATE"/>
```

Now we can finally implement the vibration functionality on the Android platform:

app/src/main/java/org/orxproject/AndroidPlatform.java

```
package org.orxproject.orxtestnative;

import android.os.Vibrator;
import android.content.Context;
import platform.Platform;

public class AndroidPlatform extends Platform {
  private Context context;
  public AndroidPlatform(Context context) {
    this.context = context;
  }
  public void Vibrate(int duration) {
    context.runOnUiThread(new Runnable() { // Since this will be called from
the game
      Vibrator v = (Vibrator)
context.getSystemService(Context.VIBRATOR_SERVICE);
      v.vibrate(duration);
    });
  }
}
```

We also need to set an instance of `AndroidPlatform` as the game's platform before the game begins; so in `app/src/main/java/org/orxproject/orxtestnative/MainActivity.java`:

```
...

import platform.PlatformModule;

...

public class MainActivity extends NativeActivity {

...

  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    PlatformModule.SetPlatform(new AndroidPlatform(this));
    ...
  }

...
```

```
}
```

Notice how `SetPlatform` is a member of the `PlatformModule` class. That's how SWIG exposes C++'s free functions in Java, static methods of the module's main class.

That's it folks, now you can go and call `GetPlatform()→Vibrate(n)` freely anywhere in your game!