Custom App/APK Resources

DEF-2382

OVERVIEW

Some games/apps need to bundle non-game resources located inside the App (iOS) or APK (Android) folders/packages. Examples include;

- Some ad extensions, ex. AdMob, expect external files (*.plist or *.xml) to be available inside the root of the package.
- Localizing app names on iOS require language specific .plist-files (with localized app name) inside subfolders of the App-folder/-package.
- Having screen size independent launch screens (would decrease the size of the app compared to static image launch screens) on iOS the user need to supply a compiled XCode storyboard (storyboardc).
- Dynamic libraries (.dll, .dylib etc) that needs to ship with the app/game and be placed in such way that they are accessable by the binary.

Our current implementation only support custom resources that are bundled inside the darc file.

REQUIREMENTS

1. Should be able to specify custom resources and handle them in a generic way.
2. Need to handle file collisions (ex if both dependency-lib-1 and dependency-lib-2 supply the same custom resource output MyApp.app/ApaBepa.plist).
3. Some resources should only be included in the bundled package on specific platforms (no need to have Info.plist on Android).
4. There also needs to be an option to include certain resources for all platforms.

SOLUTIONS

Option in Game Project

We currently have an option for "**`custom_resources`**" in the game.project file for bundling resources inside the darc file. One solution would be to add a new "**`bundle_resources`**" for each platform, and also one "general" entry under the **`[project]`** category.

Example:

```
[project]
bundle_resources = /res/
```

Pros

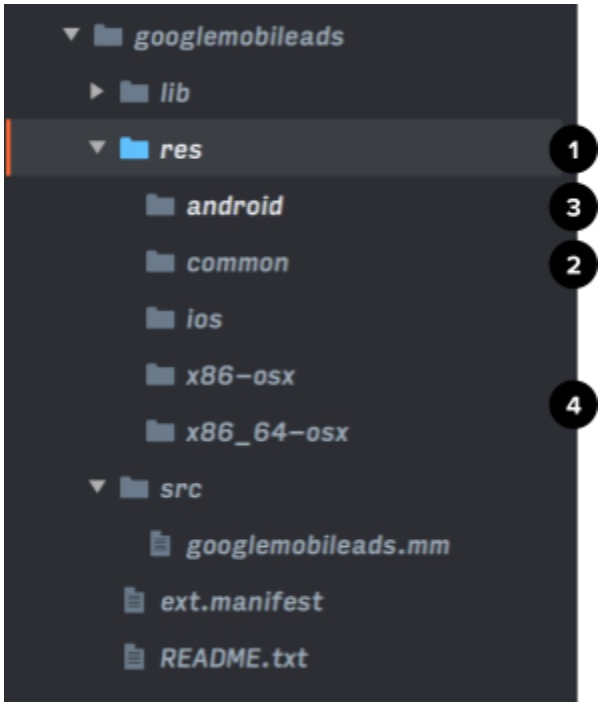
- Familiar setup as current custom_resources

Cons

- Adding a dependency/extension that needs (and supplies) custom bundle resources will need manual setup in the game.project file.

Extensions Resource Folder

We could leverage the extension system with custom resources being part of the folder structure defining an extension.



1. Just like the extensions folder structure have `src/`, `lib/` and `frameworks/` we add a new `res/` folder which include files/resources that will be bundled into the packages.
2. **common** - Resources that will be packaged on all platforms.
3. **android** - Resources that will be packaged on all android targets (currently armv7 only).
4. **x86-osx, x86_64-osx** - Resources that will be packaged for specific archs on OSX.

Pros

- Adding dependencys/extensions that needs (and supplies) custom bundle resources will automatically work, ie. automatically bundled for each platform without any manual changes to the project/game.project.

Cons

- Adding custom resources to a project without an extension will not work out of the box. The user would need to create an empty “dummy” extension with the `res/` folder to include resources.

Solution 3: Combine 1 & 2

Have the ability to have bundle resources both specified as an extension folder structure, but also specify a project resource folder with the same structure. (This will make it easier to add bundle resources for projects not using extensions, or want to supply resources expected by certain extensions.)

Common for Solutions

All resources (common + platform specific) are copied during build/bundling (there is no need to upload them if using the extension solution since the bundling still happens on the client).

We need to handle collisions of resources for both solutions;

- **Alt 1**, generate an error when a conflict occurs, just as regular resources when they have outputs that conflict;
  `Conflicting output resource 'build/default/Test.plist‘ generated by the following input files: [ext1/res/common/Test.plist] <-> [ext1/res/x86_64-darwin/Test.plist]`

- **Alt 2**, as Alt1 but add the option to specify an “override” resources folder as explained in Solution 1.

- **Alt 3**, add ability to exclude resources

```
[project]
exclude_bundle_resources = /extension1/res/ios/GoogleAds.plist
```

IMPLEMENTATION

After design meeting with Mathias, Andreas and Jakob we decided to go forward with **Solution 3** in combination with **Alt 3** for collision handling.

WIP!