Moai Project Setup

In this article I'll give a tour of Moai's project structure and discuss some typical ways to use the Moai framework from a development environment and project organization standpoint. To proceed, you'll need a good Lua text editor and an IDE. We use Xcode, Visual Studio and Eclipse internally, but if you're feeling adventurous you can certainly move the project to your IDE of choice.

As you probably know, an application consists of a compiled executable along with a set of data resources (usually graphics and sound). The exact terminology and method for bundling the executable with its resources varies from platform to platform, but the basic concept is the same.

A Moai executable has two parts: the Moai framework and the host. The Moai framework is the body of C++ code officially maintained and supported by Zipline Games. The host is a custom application that (at minimum) sets up a platform specific environment for the framework to run in, including an OpenGL surface to render to. The host also exposes any available input devices to the framework. The host is usually written in the language of choice for a given platform: Objective-C on iPhone, C, C++ or C# on Windows, C or C++ on Linux and Java on Android.

In the Moai source tree, we provide sample host implementations for iPhone, Android and GLUT. GLUT is a minimalist, open source, cross platform UI framework for writing OpenGL sample programs. As useful as it is for writing samples, GLUT is probably not the correct choice for writing a commercial desktop game. You should instead write a native host using a cross platform library such as Qt.

The Moai framework, when initialized by the host, will create a Lua runtime and can be passed Lua scripts. Thanks to Moai's Lua bindings, these scripts can create and manipulate any of the Moai objects a game designer might care about.

As I mentioned, an application contains both the executable (Moai framework and host) and data resources. Because Lua scripts can be loaded and executed at runtime (without the need for recompilation), the Lua scripts that make up your game logic are also data resources - just like images and sounds.

Loading Lua scripts at runtime is a great help during development, and may even be suitable for finished games under certain circumstances. That said, these scripts will exist in your resource folder as plaintext, readable by anyone. In some cases this won't matter, but in other cases you may wish to obfuscate your resources or bundle them into your project's executable itself; I'll discuss some ways to do so at the end of this article.

There are four basic approaches to using Moai depending on your skill set and goals:

Game Designer (Lua only): In this case, you don't care about anything under the hood. You will use Moai as an 'off the shelf' solution that will allow you to begin scripting and playing your game immediately without any need for Objective-C, C++ or Java.

Game Designer and Application Developer (Lua and a custom host): This is an approach more suitable for commercial games. While it's certainly possible to use Moai with one of the supplied sample hosts, there may be platform specific features you need for your game that the sample hosts do not provide. For example, video capture or access to a camera roll are features you might want to add to your host and then expose through Lua. While we may certainly add features like these down the road, you also have the freedom to roll up your sleeves and add them yourself without having to dig into the guts of Moai.

Game Designer and Game Engine Developer (Lua, C++ and a custom host): If you are an experienced game developer, you may recognize Moai's utility as a game library but wish to extend it with your own objects and algorithms. You can do this using Moai's Lua to C++ binding system, or inject your own into the Lua runtime; it doesn't matter to Moai. We'd recommend taking this approach as an optimization; most of the studios using Moai now either implement their whole games in Lua or prototype in Lua and then move performance critical sections of their code to C++ after carefully profiling on the device - and taking all other measures to optimize Lua first.

Game Engine Developer (C++ only): I'm mentioning this option only because it's possible. Even without Lua, Moai is a richly featured 2D game development framework. All of the Moai objects can be instantiated and manipulated directly in C++ without ever binding to the Lua runtime. That said, this isn't the intended use of Moai and so there are some challenges to doing this. The first is that we're putting almost all of our documentation efforts around the Lua binding, as most developers will be using Lua. The second is that many of the Moai objects do not offer parity between the Lua API and the C++ API – the Lua API calls are implemented as private static class methods and are not suitable for calling without a valid Lua runtime. If you are determined to use Moai directly from C++, this is easy enough to correct, but will require digging into and refactoring portions of the C++ APIs. (If you decide to do this, you are welcome to contribute these changes to the project.)

The Development Environment

To develop using Moai you will need a decent Lua text editor and an IDE that can build for your platform of choice. How much you use of each depends on your approach.

Lua only: For this approach you only need a Moai executable (or binaries and sample IDE projects if targeting mobile). On Windows and OSX you can use the pre-built host that comes with the Moai SDK. We'll make one available for Linux in the near future, too. As of this writing, on iPhone and Android you will still have to open an IDE project, add your script and content resources and build your application. Don't worry; I'll walk you through how to do that later in this article.

Lua and a custom host: For this approach you'll be building with an IDE, but you'll link to Moai as a static library. Static libraries for each platform along with sample host projects are included in the Moai SDK.

C++: To use Moai as a C++ library, you can link to one of the pre-built binaries and simply include the full SDK header files from the open source code tree, but you'll probably be better off having access to the full Moai source and build. I'd recommend using one of the existing IDE projects if you can, but you can certainly port the code base to any IDE of your choice. If you do this, you may wish to pre-build (or set up your own build) for the 3rd party libraries Moai relies on.

If you're coming to Moai from an authoring environment like Flash, Game Maker or Corona you may find yourself looking for a Lua IDE and supporting tools such as a Lua debugger and a Moai 'player' or device simulator. High level tools such as these are a good thing to have, but they aren't something we're focusing on now. Our immediate concern with Moai is developing the framework and the feature set of Moai and Moai Cloud. Once you get over the transition, working without these higher level tools might not be as onerous as you'd think; remember that there are a number of commercial games already being developed for Moai by designers not using anything higher level than Notepad++.

Before moving on, I should mention that there are at least two contributors we know of working on fully integrated Lua IDEs for Moai with a built-in simulator and debugger. This is a stand-alone project based on Mono that will be contributed as a sister project under an open source license. This project is still very early, but we are doing what we can to support its developer and may feature it officially when it is nearer to completion.

Moai SDK

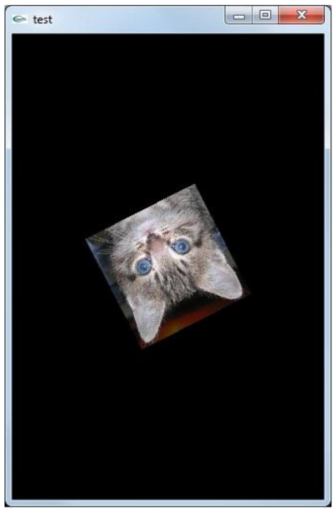
The Moai SDK contains binaries, samples and projects for all the supported platforms. In the next sections I'll walk you through using the binaries on Windows and OSX as well as building for Android and iPhone.

The SDK has everything you need to start prototyping games using the GLUT host or building your own host using Visual Studio 2008 or 2010. You can also build and run apps on Android using Eclipse and the Android SDK/NDK.

To get started, download the SDK zip file and unpack it. Open the 'samples' folder. The samples are a set of extremely basic code snippets that demonstrate some of the most frequently used objects in the Moai framework.

In the samples folder, open 'basics/anim-basic' and execute 'run.bat' or 'run.sh'. You should see a console window open followed by a graphics window that again shows a spinning cathead. The run script files are simply a convenience to save you the trouble of launching the default Moai host and passing in a Lua file.

Now that Moai is working, let’s set up a project you can use for experimentation. Start by creating a new folder somewhere in your system; in your documents folder or on the desktop, for example. Copy the entire contents of the ‘anim-basic’ folder into your new folder. When you’re done you should have a copy of the run scripts, main.lua, and cathead.png in your new folder. By default, the run scripts use a relative path to the Moai executables in the SDK. They also pass a relative reference to a sample ‘config.lua’ file as the first parameter. Since Moai executes all the Lua files you pass it in order, this demonstrates how to set up configuration files which run before your main game files. You’ll probably want to edit your run scripts and update them to continue pointing at the Moai executables and the ‘config.lua’ file in your SDK (or simply create your own config set up or omit it entirely). You should also consider adding the binaries to your system path so you can easily access them from anywhere. Once you’ve configured this, execute the run script again. If you see the cat head spin, then everything is set up and working correctly.



Now that you have your own project in which to experiment, open ‘main.lua’ and try changing some of the numbers there. By changing the numbers you should be able to modify the shape of the window, the shape of the cat head, and the speed (and direction) in which it spins.

Building with Visual Studio

The code and project that builds the GLUT host you’ve been using to spin cat heads is included in the Moai SDK. If you know C or C++, you can use this as the starting point for writing a custom host or a device simulator.

First, locate the Moai host solution directory for the version of Visual Studio you have installed. Open the ‘hosts’ folder and choose either the ‘vs2008’ or the ‘vs2010’ subfolder.

The Visual Studio solution contains two projects: ‘moai’ and ‘moai-untz.’ Each is configured to reference Moai as a statically linked library. The ‘moai’ project builds the Moai GLUT sample host without any sound support. It should build and run as-is without any additional configuration. The ‘moai-untz’ project is packaged separately as it requires

DirectX for sound support, which we do not redistribute. To get DirectX, download and install the latest DirectX SDK from Microsoft's developer page. Once installed, you will need to add the search paths for the DirectX include and library folders to either the 'moai-untz' project or to Visual Studio's global search paths. A readme file is included in the vs20xx folder explaining exactly how to set up the DirectX paths.

I'm going to assume that if you're doing this you already know how to use Visual Studio and have goals in mind for your own Moai host, so I won't go into further detail here.

Running on Android with Eclipse

To run on Android you'll need Eclipse (configured with the Android development environment) and an Android device.

I believe it is also feasible to work under IntelliJ IDEA, about which I have heard good things. We don't have an IntelliJ reference project at the moment, but may offer one in the near future.

To get going on Android with Eclipse:

1. Follow the Android development environment set up instructions provided at the official website: http://developer.android.com/sdk/index.html. There are several steps in this process, but their documentation should answer your questions. You'll only need to set up their SDK to run the sample host; the NDK they provide is only necessary if you wish to build from source. This topic is covered later in the document.
2. Once you've set up your Android environment, you'll need to import the sample host project. Right-click in the 'Package Explorer' area of Eclipse, and click 'Import…' In the 'Import' dialog box, expand the 'General' folder and choose 'Existing Projects into Workspace.' In the next dialog box, click the 'Browse…' button. Select the folder 'hosts/eclipse/android-project' and click ok. You should see the project called 'MoaiSample' appear in the projects list. Click the 'Finish' button.
3. To get the app to build each time, you should touch a code file and save the changes. Otherwise, Eclipse may not recognize that anything needs to be done when you execute the build command.
4. Building an Android project in Eclipse can be tricky at first. The build generates a folder called 'gen' which is required to launch the app. Unfortunately, you sometimes have to build a few times in Eclipse to get the 'gen' folder to show up properly. Until it appears, the app will show build errors. This situation can repeat itself each time you clean the project. The best way to get around this is to build the project, then refresh your project tree. You should see the 'gen' folder appear there after the first couple of tries.
5. You'll need an Android device to see Moai running. This is because the emulator included in the Android SDK does not support OpenGL as of this writing. The MoaiSample host app is configured to build for Android 2.2.

There is still one more thing to do before you can deploy to Android. Unfortunately, the Android application format does not offer a file system image to us in which to place our Lua scripts and resources. An Android application is actually a zipped directory from which application resources are dynamically loaded using Android OS functions that bypass a file system and are therefore incompatible with Moai. As of this writing, our current solution is to create a resource bundle of our own that contains our desired project directory structure. Our Moai resource bundle is simply a .zip file that is included as a single resource under the Android project's 'res/raw' directory. When the Android application is installed, the Moai bundle is extracted and unzipped to the system's writeable storage (usually the phone's flash memory card). This preserves our directory structure, enabling Moai to run normally.

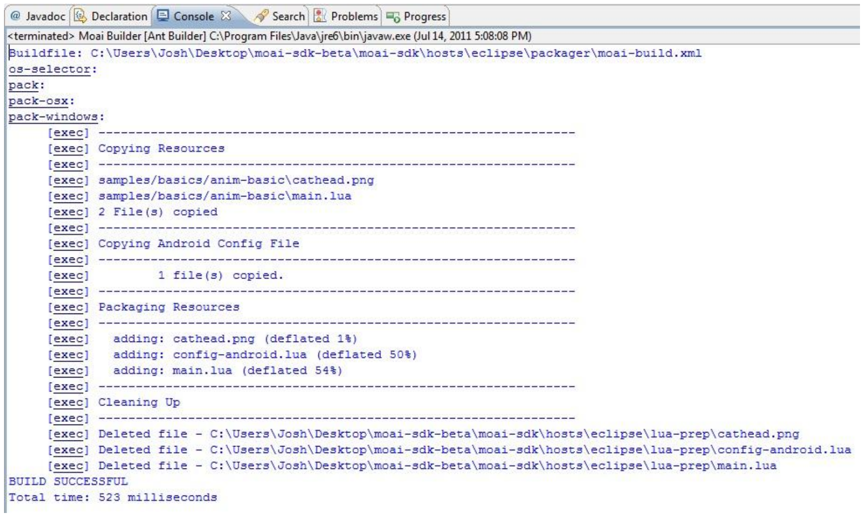
In future versions of Moai we may offer an alternative system for loading scripts and assets to be compatible with the Android application bundle.

An alternative to consider is writing a custom Android host that downloads and unpacks your Moai project bundle from a server (such as Moai Cloud). This will avoid the increase in size that accompanies our original method.

Fortunately, the sample host has a script which handles packaging a Lua resource folder automatically. In order for this script to execute properly, you must complete the configuration steps described in 'hosts/eclipse/README.txt.' These steps include adding ant-contrib to Eclipse, and ensuring you have command line zip exposed on your path.

The sample host uses a file called 'moai-target' to define the folder in which your Lua resources are located. During the build process, the contents of this folder are automatically packaged and placed in 'res/raw' as described above. By default, 'moai-target' is pointing at the 'anim-basic' spinning cat head script.

Now that everything is set up, build the project. The console output should show your Lua resources being copied. If everything was done correctly, the application will deploy to your Android device, and you should see the cat head appear and spin.



```
<terminated> Moai Builder [Ant Builder] C:\Program Files\Java\jre6\bin\javaw.exe (Jul 14, 2011 5:08:08 PM)
Buildfile: C:\Users\Josh\Desktop\moai-sdk-beta\moai-sdk\hosts\eclipse\packager\moai-build.xml
os-selector:
pack:
pack-osx:
pack-windows:
     [exec] ----------------------------------------------------------------
     [exec] Copying Resources
     [exec] ----------------------------------------------------------------
     [exec] samples/basics/anim-basic\cathead.png
     [exec] samples/basics/anim-basic\main.lua
     [exec] 2 File(s) copied
     [exec] ----------------------------------------------------------------
     [exec] Copying Android Config File
     [exec] ----------------------------------------------------------------
     [exec]         1 file(s) copied.
     [exec] ----------------------------------------------------------------
     [exec] Packaging Resources
     [exec] ----------------------------------------------------------------
     [exec]   adding: cathead.png (deflated 1%)
     [exec]   adding: config-android.lua (deflated 50%)
     [exec]   adding: main.lua (deflated 54%)
     [exec] ----------------------------------------------------------------
     [exec] Cleaning Up
     [exec] ----------------------------------------------------------------
     [exec] Deleted file - C:\Users\Josh\Desktop\moai-sdk-beta\moai-sdk\hosts\eclipse\lua-prep\cathead.png
     [exec] Deleted file - C:\Users\Josh\Desktop\moai-sdk-beta\moai-sdk\hosts\eclipse\lua-prep\config-android.lua
     [exec] Deleted file - C:\Users\Josh\Desktop\moai-sdk-beta\moai-sdk\hosts\eclipse\lua-prep\main.lua
BUILD SUCCESSFUL
Total time: 523 milliseconds
```

Running on iOS with Xcode

To build and deploy to an iOS device you need to use Xcode and the iOS SDK. You can find the latest iOS SDK at Apple's developer website. Once you have it installed, open the Moai Xcode sample project located at 'hosts/xcode-ios/moai.xcodeproject'. The sample project is set up to link to the Moai static libraries in 'bin/ios'.

Before building, let's take a look at the Xcode project structure.

As you can see, there is a small set of code files under the 'Classes' folder at the top of the project. These are the files that make up the Moai host. There is also a folder called 'Other Sources.' This folder contains the 'main' for the iOS app and the precompiled header.

The 'Resources' folder contains the app icons required by iOS, and a few other standard resource files. It also contains a subfolder 'build' which contains a file called 'moai-target' and a folder reference called 'lua.' By default, these two files should be shown in red because they don't exist until you build. I'll explain the function of this folder in greater detail below.

Finally, the 'Frameworks' contains references to the frameworks and static libraries required by the targets and the 'Products' folder contains references to the products built by the targets.

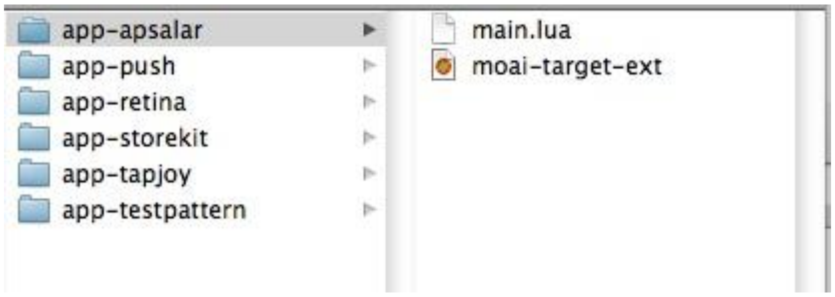
The project group structure outlined above is not our invention; it is the standard layout recommended by Apple for iOS application projects.

Now that we've covered the project structure, let's discuss how our Lua resources are packaged into the host.

During the build process, a package script runs which reads the contents of the 'moai-target' file in the 'Resources/build' project folder in order to locate the project's Lua resources. The script copies the entire contents of this target folder into the Lua reference folder (also found in the 'Resources/build' project folder) and marks the contents as read-only (to indicate they should not be edited). The standard iOS bundle creation continues and the Lua resources are included in your package under a folder called 'lua.' The AppDelegate file then executes a file called 'main.lua' in the 'lua' folder of your package. You can change this behavior by editing the AppDelegate file if you wish. This set up allows you to quickly try out any of the samples provided in the SDK. Simply point it at a new folder and the MoaiSample host will play the contents of that folder.

There are a few additional features built into this packaging script. First, you can include any number of folder references, one per line, and the package script will grab the contents of each and combine them in your Lua reference folder. This is useful if you want to place general files somewhere besides your project folder.

Another feature of the packaging script is the ability to define folder dependencies within the folders themselves. For example, the ‘samples/iphone/app-apsalar’ sample depends on the Apsalar Lua module included elsewhere within the SDK. If you look in this folder, you’ll notice a file called ‘moai-target-ext.’ When a target folder is packaged, the script checks for the existence of this ‘moai-target-ext’ file. If it’s there, the package script is called recursively on the folders listed in this extension file. This allows you to point your main ‘moai-target’ file at the ‘app-apsalar’ folder and have its dependency copied automatically.

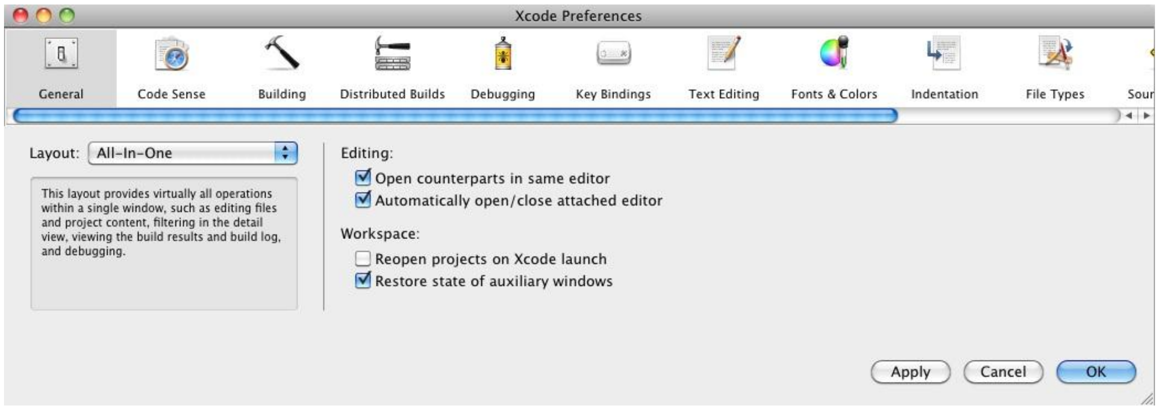


Before deploying to a device, let’s build and run for the simulator. To do this (using Xcode 3), select ‘Simulator’ and ‘Debug’ from the build configuration dropdown. Now select ‘Build and Debug’ from the ‘Build’ menu. You should see some build output in the ‘Build Results’ view of Xcode. A few seconds after the build is finished, the iOS simulator should appear and show you the spinning cat head of which we’re all so fond.

To build and deploy on an iOS device, simply select ‘Device’ from the build configuration dropdown instead of ‘Simulator.’

Note that if your Xcode 3 doesn’t look like mine, leaving you angry and confused, you may not be set up to run Xcode in its ‘all in one’ mode. Take comfort in the fact that I was also angry and confused before discovering Xcode’s all in one mode.

To set up the all in one mode, go to ‘General/Preferences…’ and select the ‘General’ tab. From the ‘Layout’ drop down select ‘All-In-One.’ When you reopen the Moai solution, Xcode should now be in all in one mode.



Layout: All-In-One

This layout provides virtually all operations within a single window, such as editing files and project content, filtering in the detail view, viewing the build results and build log, and debugging.

Editing:
- [✓] Open counterparts in same editor
- [✓] Automatically open/close attached editor

Workspace:
- [ ] Reopen projects on Xcode launch
- [✓] Restore state of auxiliary windows

Note: you cannot edit this preference if you have any open projects in Xcode, so close your projects beforehand.

Configuring a New Xcode Project

To get started building your own Moai projects for iOS, the best approach is to copy the Moai samples project and adapt it to your needs.

Create a new folder to contain your own project and resources. Copy the Moai Xcode project and its subfolders into the new folder.

You may wish to remove the 'moai-target' file and its accompanying scripts. You may instead prefer to package your Lua resources in another way. If this is the case, simply remove the files 'moai-target,' 'mt.default,' 'package.sh,' and the folder 'lua' from your project folder in Finder. You'll also want to delete the 'Resources/build' folder in your Xcode project tree. To disable the build script, expand the 'Targets' area in your project tree, and then expand the 'MoaiSample' target. Select and delete the build step called 'Run Script' to remove the call to package.sh.

Open the 'Frameworks' group and look to see if any of the Moai libraries are highlighted in red. To fix the broken links, right click on the target icon and select 'Get Info.' On the 'General' tab you will see the references to external frameworks. All of the broken links should be visible under 'Linked Libraries.' Select these and remove them. Now re-add them by clicking the plus icon at the bottom left of the window.

Now that the missing libraries have been added, we also need to fix the broken path to the Aku headers. Close the target's info window. Right click on the project root, choose 'Get Info' and select the 'Build' tab. Make sure that 'All Configurations' and 'All Settings' are selected. Scroll down in this window until you located the 'Search Paths' section. In this section you will find the 'Header Search Paths' setting. The original relative path to the Moai 'include' folder is there. You will need to change this to a new path (relative or absolute) to the Moai 'include' folder.

If you've removed the packaging script, you'll need to add the Lua files for your new project. Return to Finder and add a project subfolder called 'lua.'

To add your Lua files to the project we recommend using a folder reference. To do this, right click on the 'Resources' group in your project and choose 'Add/Existing Files…' Use the file selector dialog that appears to select the 'lua' folder you just created in your project folder in Finder. Choose 'Add,' select the 'Create Folder References for any added folders' radio button and then click the 'Add' button at the bottom of the dialog. If you did this step correctly, you should see a folder reference appear in your project. Unlike regular project groups, the new folder reference is blue.

A folder reference is convenient because any files you add under the referenced folder in Finder will be automatically included in your project and sent to the app bundle each time you build.

Building from Source

Building Moai from source is more advanced than using the binaries or writing a custom host. That said, the reference IDE projects included in the source tree build Moai and all of its dependencies for each supported platform, so while getting the source and doing a build from scratch is more involved than using the binaries, you may find the process easier than configuring and building other open source projects (depending on the project, of course).

To get the Moai source, download it from https://github.com/moai/moai-beta/downloads and unpack it to a convenient location. Alternatively, you can clone or fork the Moai source tree git repository: git@github.com:moai/moai-beta.git. I personally recommend this approach over a download, as you will find it much easier to stay current with the project and contribute your own improvements by sending us a pull request. You can read about using github here: http://help.github.com/.

The root of the source project is organized with a folder for 3rd party library projects, the Moai source code itself and a directory for each of our supported IDEs. There is also a folder for building our distributable items, such as the Moai SDK and the documentation.

The 3rd party source projects are included for a few reasons. First, by including them we can make sure that you immediately have everything you need to build the exact same version of Moai that we build. Second, keeping a snapshot of the 3rd party projects we're using ensures version compatibility. The trade-off is that we don't automatically get the benefits up 3rd party updates and bug fixes.

To make it easy to move to later versions of the 3rd party libraries, our policy is to change as little of those libraries as possible. When a library must be configured or changed, our approach is to duplicate its 'includes' folder, give it a platform specific name and place it ahead of the project's standard 'includes' folder on the header search path. We then put our configuration and changes in the copy instead of the original.

Building from Source with Visual Studio

Go ahead and open the folder and solution for the version of Visual Studio you are using. The solution contains projects to build a suite of executables, the Moai core library and extensions, and sub-projects for the open source 3rd party libraries Moai depends on.

Take a moment to look at the project structure. The open source projects have been assembled and included into the solution. This saves us the inconvenience of having to wrangle the build tools for each of these projects independently: everything you need to build them is right here at your fingertips.

In the 'Libraries' folder you'll see the various libraries that make up Moai itself. The 'uslscore' and 'uslsext' are utility libraries on which Moai is partially based. Don't worry about those for now. The 'moaiext-luaext' project is a set of Aku interfaces for initializing some of the additional Lua modules we decided to bundle with Moai. The 'moaiext-untz' project contains the Untz extension.

In the ‘Products’ folder, you’ll see ‘moaicore’ which contains the entire Moai framework and Lua bindings.

The ‘Utilities’ folder contains project to build the various utility apps that ship with Moai.

To build the GLUT host and its dependencies, just right click on the ‘Utilities/moai’ project and choose ‘build’ from the context menu. Sip your beverage of choice and watch as several hundred files are compiled and linked. If everything went according to plan, you should have a shiny new build of the GLUT host and the Moai libraries.

If you want to build ‘moai-untz’ you will have to install DirectX. The ‘moai-untz’ project makes no assumptions about where you’ve installed DirectX, so you will have to either add the search paths for DirectX includes and libraries to the project itself or add them globally to your install of Visual Studio. I recommend the latter method as this will not alter the project structure. A readme file is included in the vs20xx folder explaining exactly how to set up the DirectX paths. Please note that ‘moai-untz’ also builds a product called ‘moai’ and will overwrite (or be overwritten by) the product of ‘moai.’

Building from Source with Xcode

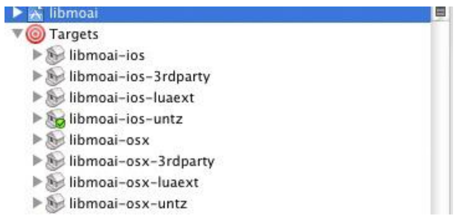
There are three Xcode projects in the source tree: ‘ios,’ ‘osx’ and ‘libmoai.’ Due to a quirk of Xcode 3, you will need to build libmoai indirectly using either ‘ios’ or ‘osx.’ This quirk may have been corrected in Xcode 4. Although I’ve ensured that Moai can build under Xcode 4 (by having our build server use Xcode 4 for each release), we have not switched to Xcode 4 for daily use. We plan a switch soon as Xcode 3 support is being dropped in iOS 5.0.

The ‘ios’ project contains the source for the iOS host and a targeting script used to select and run the sample projects. It also contains an external reference to the ‘libmoai’ project. If you are using Xcode 3, only the products from the external project will show up. If you are using Xcode 4, you should see the entire ‘libmoai’ project and its contents inlined in the ‘ios’ project.

The ‘osx’ project follows a similar structure to the ‘ios’ project. Like the ‘ios’ project there is also an external reference to the ‘libmoai’ project.

Before building either project, let’s take a look at the structure of the ‘libmoai’ project. If you are using Xcode 3, double click the external reference to open the project; if you are using Xcode 4, you can peruse the project’s contents from within the containing project.

The ‘libmoai’ project has eight targets, each a static library:



The ‘libmoai’ target builds the Moai static library. The ‘libmoai-3rdparty’ project contains all of the 3rd party code that Moai depends on. The ‘libmoai-luaext’ project contains the optional Lua extensions available for Moai. The ‘libmoai-untz’ project contains the Moai framework extension for the Untz sound library.

The source files included in the ‘libmoai’ project are grouped into the Moai source and the 3rd party libraries. Like the Visual Studio project, all of the 3rd party sources are included for building with the IDE.

The source code under the ‘moaicore’ group contains the Moai framework and its Lua bindings. The sources under ‘uslscore’ and ‘uslsext’ are legacy utility libraries on which Moai is partially based. The sources under ‘moaiext-luaext’ and ‘moaiext-untz’ are Moai extensions. They implement the Aku host binding API for initializing the Lua extensions and the Untz runtime respectively. The sources under ‘moaiext-iphone’ are an iOS specific extension that combines C++ and Objective-C to add native iOS features to the Moai framework. These include objects for push notification, in-app purchases and device specific information.

As I mentioned before, to build for iOS or OSX, use the ‘ios’ or ‘osx’ projects. This ensures the project settings are resolved correctly for each platform under Xcode 3.

Building from Source with Android NDK

To build the Moai source code for Android you’ll need the Native Development Kit (NDK). Unfortunately, if you’re using Windows you’ll also need Cygwin since the NDK relies upon Unix bash commands. Word on the street is that Google is planning to abandon Cygwin for future releases of the NDK, but for now you’ll just have to suffer along (unless you like Cygwin. In which case: enjoy.)

Download Cygwin here: http://www.cygwin.com/
Download the Android NDK here: http://developer.android.com/sdk/ndk/index.html.

You should unzip the contents of the NDK package to a location with no spaces in its path. Spaces can negatively impact the scripts you’ll be running. Cygwin’s installer is more complex. It features a large number of modules which you can selectively install. As of this writing, the Moai build process relies upon several executables which are not installed by default. To be safe, you can install the entire Cygwin developer folder. There is a readme file located in ‘eclipse/libmoai’ which describes all the things you need to do to get things up and running under Cygwin. If you run into any issues, please refer to this file for additional instructions. If you find the build process breaking because a certain executable cannot be found, you can probably find information about getting the missing file by searching online.

Once the NDK and Cygwin are installed, you can run the appropriate build script located in the ‘eclipse/libmoai’ folder manually if you choose. However, the proper build script will be automatically run when you build the project in Eclipse (as long as you’re using the sample host included in the git repository; the sample host in the Moai SDK uses a precompiled library). This means the Eclipse project should just work once you've fully set up the environment.

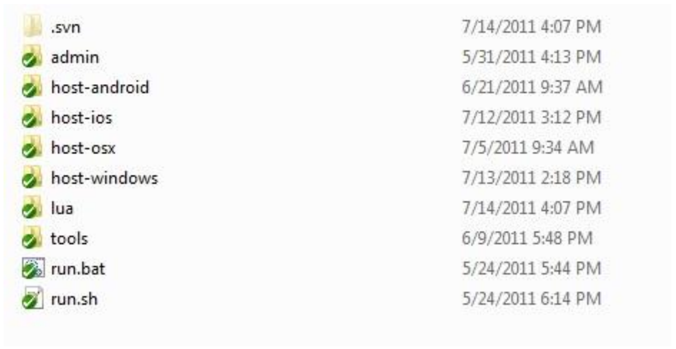
You'll be in for a wait as the entire Moai source tree and its 3rd party libraries must be built. Under Cygwin this is especially slow. Once the build is complete, you should have a new Moai shared library in 'eclipse/libmoai/libs/armeabi'. This library is already referenced by the Eclipse project and will be linked in to your Android application.

Using the Moai Lua Modules

At the time of this writing, Moai ships with several modules written in Lua. These modules are SDKs (and their support modules) for using the partner services provided by Tapjoy and Apsalar. To use these modules, include the Lua module source code (and the modules it uses) in your Lua project as you would any other Lua code. You may then use Lua's package system to load the modules using the 'require' directive at the start of your application.

Setting Up for Cross Platform Development

Our recommendation for structuring your project folders for cross platform development is to create a root folder for the project and to put all of your Lua and game assets in a subfolder. For each build system or IDE you intend to use, create a separate folder alongside the Lua folder to house your platform specific IDE projects. In the root of your project folder, you can also add any scripts and utilities you need to use during development.



```
.svn            7/14/2011 4:07 PM
admin           5/31/2011 4:13 PM
host-android    6/21/2011 9:37 AM
host-ios        7/12/2011 3:12 PM
host-osx        7/5/2011 9:34 AM
host-windows    7/13/2011 2:18 PM
lua             7/14/2011 4:07 PM
tools           6/9/2011 5:48 PM
run.bat         5/24/2011 5:44 PM
run.sh          5/24/2011 6:14 PM
```

Since we like to prototype on Windows, this is usually a batch file to run the Lua code using the Windows Moai host. If you are frequently updating the Moai host, you may also want to place a copy of the host binary under a local 'bin' folder in your project to ensure that you won't be subject to breaking changes. This is good practice anyway, as your project will be still able to run even after many updates to Moai.

Packaging Lua Code

There are a few different ways to approach packaging your Lua scripts for inclusion in your app. This first (and simplest) is to simply copy your Lua script folder verbatim into the app bundle (on iPhone), ship them in a resource folder that installs with your app (in the case of a desktop app) or store them in an archive and unpack them to writable storage (on Android devices).

The shortcoming with the simple method is that all of your Lua scripts are available on the device in plaintext. While this is fine if you want to encourage users to mod your game, in some cases you may want to obfuscate your code to make accessing it more difficult.

Before we go on, I should stress that there is a big difference between obfuscation and encryption. Any savvy would-be-hacker, with enough time and effort, can decompile your game code. This is true of code written in any language, so we aren't introducing any new security vulnerabilities. The point of obfuscating your Lua code is to prevent the casual hacker from gaining easy access to your game logic.

You have several options for obfuscation:

Running Lua files through luac: You can compile your Lua files using luac. These files are no longer in plaintext and therefore are harder to change.

Wrapping compiled Lua files into your executable: After compiling all of your Lua files using luac, you can convert them to hex, dump them into header files, and compile this data in alongside your source code. This approach ensures you don't have script files sitting in resource folders, but it requires the implementation of a custom loader for the Lua chunks you've put into the header files. You won't be able to use Lua's loadfile command with this set up.

Encrypting your compiled Lua files: After compiling all of your Lua files using luac, you can also encrypt your them. You must then hide your encryption key somewhere. The easiest approach is to simply hide the key in your app, but it's also the most vulnerable.

None of these options is truly secure. No matter which technique you use, you are still vulnerable to hacking on some level. The only thing you can do is raise the level of sophistication and time necessary in order to compromise the system. It's probably best to prevent casual hacking of plaintext files, but anything more advanced will require your own research into the issues surround digital rights management.