Extending Moai

Patrick Meehan
July 1, 2011

In this paper I'll discuss various ways to add custom functionality to Moai. From least to most involved:

1. **Lua modules:** As Moai leaves full Lua functionality in tact, it is possible to load custom or 3rd party Lua modules at runtime or to link in Lua modules by building them into Moai.
2. **Lua runtime injection:** Moai's Lua runtime is available through the Aku interface via a call to AKUGetLuaState (). By obtaining the Lua state, it is possible to inject custom data, functions and modules.
3. **Lua wrappers:** It is possible to extend any Moai object by wrapping its factory method and modifying its instance table after creation. This can be used to greatly simplify the initialization of and interaction with frequently used objects.
4. **Object binding with USLuaObject:** Moai is a loosely coupled framework of objects. It is possible to use Moai's low level class binding system to create custom objects that do not rely on any part of the Moai framework.
5. **Moai framework extension:** The Moai framework consists of simulation objects based on a small set of base classes that integrate with data structures for dependency graph execution, spatial hierarchy, spatial partitioning, rendering and simulation. A working knowledge of these subsystems is necessary to write a framework extension. That said, I designed the framework to be easily extended and maintained (at least by me). While I can't speak for anyone else's experience, I personally find it's not much trouble to extend the framework.

I won't go into the first two methods of extension here. There is plenty of information about the Lua runtime and how to write modules for it. Bear in mind that the USLuaObject binding system on which Moai is based assumes that any Lua userdata is a reference to a class based on USLuaObject. While type safety is guaranteed across objects using USLuaObject, no such guarantees are made about third party userdata.

Lua Wrappers

Any instance of a Moai class (or any class based on USLuaObject) has a unique instance table that may be extended by the user. This table sits between the instance's userdata and its internal implementation and may be used to mask or alter the object's default implementation.

The typical pattern is to create a factory table that contains a custom new () method. So instead of calling MOAIThing.new (), the user will call FOOThing.new (). Internally, FOOThing.new ()will create a new instance using MOAIThing.new () and then decorate it with custom methods and members before returning it to the user.

An alternative approach is to wrap instances of Moai objects entirely in custom instance tables and manage all access to them.

Object Binding with USLuaObject

USLuaObject is a base class that may be used to bind any derived class to the Lua runtime.

To expose a custom C++ class to Lua:

1. Inherit USLuaObject.
2. Wirte your Lua interface. My convention is to declare all Lua C methods as private static class methods.
3. Add DECL_LUA_FACTORY or DECL_LUA_SINGLETON to the public section of your class declaration.
4. In your class's constructor(s), notify USLuaObject's run time type information system by calling the RTTI_BEGIN, RTTI_EXTEND and RTTI_END macros. These work for multiple inheritance as well.
5. Overload RegisterLuaClass () and RegisterLuaFuncs () to add your Lua C methods. If you are inheriting other Lua bound classes, be sure to call their implementations of RegisterLuaClass () and RegisterLuaFuncs () explicitly to inherit their methods.
6. Register your class with the Lua runtime using the REGISTER_LUA_CLASS macro. You will need to do this each time a new Lua runtime is created.

That might seem like a lot of work, but in practice it isn't too bad: inherit from a class, add a macro to your declaration, a macro to your constructor, overload two methods and register with the runtime. The two most common gotchas here are forgetting to add the RTTI_ macros to the class constructor and forgetting to register the class with the runtime. If you work from an existing file or a code template, everything else should be a snap.

Of course, there's more to understand before working with USLuaObject. It's important to know how the binding works under the hood.

When an object class is registered with the Lua runtime, its 'class table' is added to Lua's global namespace. The class table is a Lua table that contains information about a class and any global methods pertaining to the class. This is the same table passed in to RegisterLuaClass (). It is initialized only once.

A bound class may be a singleton or a factory. If it is a factory, then its class table will contain a new () method that may be used to create instances of the class. The new () method is added before RegisterLuaClass () is called by the system and may therefore be overloaded or removed.

When a user calls the new () method of a factory class, a new instance of that class is allocated. Instead of adding the entire instance to the Lua runtime as a userdata, the class is allocated normally (using the C++ 'new' operator) and a handle to the class is allocated by the Lua runtime as a userdata. Using a handle for the userdata instead of the instance itself allows the Lua garbage collector to break the binding when the instance goes out of scope yet allows the instance to remain in memory if it is still being used. On the C++ side, class instances are tracked by reference counting. The userdata handle simply adds a reference to the instance. When it is garbage collected, the instance will be deleted if there are no other references to it.

After creating the handle, two Lua tables are connected to it in a metatable chain. The first as an empty table called the 'instance table.' This is a table created just for the new instance of the class. It implements both the __index and __newindex metamethods to allow users to extend the instance. While it is bound to the handle as a metatable, the instance table will survive garbage collection of the handler and persist until the instance itself is deleted: the class instance keeps a strong reference to its instance table but a weak reference to its handle.

Another metatable is then attached on top of the instance table. This table is known as the 'member table' and contains pointers to the methods and any read-only members of the instance. The member table is initialized once by RegisterLuaFuncs () and is shared by all instances of a class.

So in summary, from Lua's perspective an instance looks like a metatable stack. From bottom to top:

Userdata Handle – Handle that refers to the class instance.
Instance Table – Per instance table for use by the scripter.
Member Table – Shared by all instances of a given class.

The userdata handle is what is returned by new ().

This implementation favors size over speed: only one table of function pointers is required per class. The disadvantage is that resolving a C method call requires not one but two metatable lookups. For performance critical loops, it may be faster to extract the C function pointers from the instance before any performance critical sections of the code and then call them directly, without the metatable lookup.

When writing your C functions, to retrieve an instance of a class derived from USLuaObject given a userdata, the userdata must first be cast to a pointer of type USLuaObject from which the USLuaObject custom RTTI may be used to dynamically cast to the desired type (using USCast<>() or the AsType<>() class method). Be aware that the initial cast from userdata is unsafe.

To make casting userdata a little easier, USLuaState (a wrapper on lua_State) provides a GetLuaObject<>() template method that retrieves a userdata and performs the dynamic cast for you.

If you are not planning to persist your objects beyond the scope of the Lua garbage collector then you can stop reading. If, however, you wish your objects to remain in memory even after they fall out of Lua's scope then you will need to use reference counting. USLuaObject exposes Retain () and Release () methods for this purpose. As a convenience, the smart pointers USRef<> and USWeak<> are also provided. USRef<> will pin the object it points to. USWeak<> will not and will automatically be nullified when the object is deleted, so be sure to always check for nil before dereferencing.

Moai Framework Extension

All Moai framework classes are derived from USLuaObject, so you should be comfortable with that system before continuing. Obviously, a good understanding of Moai's architecture and subsystems also helps!

Here's bird's eye view of some of Moai's subsystems and concepts:

1. **The action tree:** This is a tree of timers that is executed at the start of every game loop. These are the only Moai objects that receive a time step.
2. **The dependency graph:** This is a node graph that is evaluated every frame, after actions have been applied. Only nodes that have been scheduled for update are processed.
3. **The spatial hierarchy:** This is a directed acyclic graph of affine transformations used to describe the spatial relationships between nodes.
4. **Animation:** Moai's animation engine is based on curves and dependency node attributes. Curves may be connected directly to attributes and driven by timers. Alternatively, curves may be collected into animations via the MOAIAnim object. MOAIAnim derives from MOAITimer and will apply the values of its curves directly to target attributes on sets of objects.
5. **The Sim:** The Sim is a singleton that controls Moai's simulation. It runs the game loop and exposes controls for setting the size of the simulation step.
6. **Props:** Props are the atomic objects that make up a scene. A prop is a combination of a transform and usually some kind of geometry. Props are held in partitions (data structures used to speed up spatial queries) and are operated on by rendering methods. Note that props are not necessarily renderable: props may also be used to implement collision geometry, regions, cells, lights, sound emitters – anything that needs a both a transform and inclusion in a partition.
7. **Decks:** Decks are indexed collections of geometry that may be instanced by props. Deck contents usually consist of graphics geometry, but may be used for others kinds of primitives as well, such as collision surfaces.
8. **Shaders:** As of this writing, Moai still uses OpenGL ES 1.1, which doesn't support shaders. For this reason, the concept of a 'shader' in Moai is a placeholder for the real thing. A Moai shader is simply an object that manages a Prop's render

state beyond geometry and texturing. Conceptually, shaders should manage all of a Prop's graphical state, including texturing. As Moai develops, more emphasis will be put on using shaders to control all aspects of rendering.

9. **Layers:** Layers are rendering algorithms that operate on sets of Props. Layers are Props themselves and, as such, may be included in other layers. The only type of Layer at the time of this writing is MOAILayer2D. This is a layer that simply queries a partition using the view rectangle then sorts the objects it finds and renders them in order. More sophisticated layer types may be added later.
10. **The Input Manager:** The Moai input manager exposes a set of input devices and sensors that may be configured by host applications using the Aku library. To extend Moai with additional sensor types, Aku should also be extended to make integration with the host easier. Moai's sensor implementations buffer input events and then push them to Lua during the game loop. This way individual input events are available to users for callbacks. Users may also poll input state at the speed of the game loop.
11. **Logging:** Moai doesn't directly contain log strings. Instead, Moai exposes log event types for which users may register custom messages and control which messages they receive. When extending Moai, it may be desirable to add custom log messages. Following Moai's pattern will make it easier to manage message text and localize should the need arise.
12. **Particles:** MOAIParticleSystem is derived from MOAIProp. It may be extended with custom emitters and forces or by adding new particle bytecode commands.
13. **Physics:** Moai has bindings for both Chipmunk and Box2D. Both libraries resist easy extension, but there is no reason that separate, game specific physics cannot be added independently of these.

The objects most likely to be derived from are probably MOAIAction, MOAIProp, MOAIProp2D, MOAIDeck and MOAITransformBase. It may also be interesting to derive from MOAINode, should you wish to create new objects that also work with Moai's animation system. Of course, plenty of Moai objects also just derive directly from USLuaData.

MOAIAction

Inherit MOAIAction and implement OnUpdate (). Once started, OnUpdate () will be called with the current time step. It is safe for you to stop the action from within OnUpdate () at any time.

MOAIActions may be in several states. 'Done' means the action has no more work to do. 'Active' means the state is in the action tree. 'Busy' means the action is in the action tree and isn't done.

By default, an action will report its state as not being done to users for as long as it has children that aren't done. For this reason, it's a good idea to also implement IsDone (). This method should simply indicate whether or not the action has more work to perform.

This condition, when true, will cause an action to be automatically stopped by the action tree after calling its OnUpdate () method.

You may also override OnStart () and OnStop () to trap the action's start and stop events.

When deriving from MOAIAction, also consider extending MOAITimer. MOAITimer is an action that tracks elapsed time and exposes a richer set of controls to the user. MOAIAnim and MOAIEaseDriver are examples of classes that derive from MOAITimer.

MOAIProp

Inherit MOAIProp and implement Draw () and/or DrawDebug (). GatherSurfaces () is temporarily unused. In future versions of Moai, it will be and there will be more virtual methods to implement, depending on what kind of prop you are writing.

You will also probably want to implement OnDepNodeUpdate () for your Prop. This method is called only when the Prop (which derives from MOAINode) is scheduled for update. For this reason, make liberal use of ScheduleUpdate () in any mutators on your Prop.

In your Prop's constructor, be sure to set a type mask for the prop using SetMask (). This is an optimization that will let Moai's systems know what the Prop can do.

Also be sure to call UpdateBounds () in your Prop's OnDepNodeUpdate () method with the current bounding box (in world space) of your prop and one of BOUNDS_EMPTY, BOUNDS_GLOBAL or BOUNDS_OK. This will update the Prop's location in the partition. BOUNDS_EMPTY means the Prop will never be returned in a spatial query. BOUNDS_GLOBAL means it will always be returned. BOUNDS_OK means the Prop will be returned only when its bounds overlap the query.

The most common gotcha when implementing MOAIProp is forgetting to call UpdateBounds (). In that case, the Prop will not be rendered even if added to a layer. If a Prop's Draw () method isn't being called, UpdateBounds () should be the first thing you check.

MOAIProp2D

MOAIProp2D is already set up to work with decks and to integrate well with some of Moai's other 2D features. MOAIProp2D has its own implementation of OnDepNodeUpdate (). If you provide your own, be sure to call MOAIProp2D's version as well.

When deriving from MOAIProp2D, override GetLocalFrame () if you do not want to use the default implementation. The default implementation uses the bounds of the currently selected Deck item as its frame.

A Note About Drawing

If you are creating a new Prop or Deck type, chances are you will be drawing. Right now, Moai's Props and Decks mostly draw textured quads. Because Moai draws so many quads and because these quads move frequently and independently, we don't have a good case for using static vertex buffers, which are more suited for large chunks of geometry.

In order to optimize drawing and attempt to minimize render state changes, most of Moai's drawing is done using the USDrawBuffer object. This is a singleton that buffers OpenGL state changes and only passes them through to the graphics hardware when absolutely necessary. USDrawBuffer also offers a configurable vertex buffer for batching primitives. When the vertex format, primitive type or graphics state changes, the buffer is flushed.

To get the most benefit, USDrawBuffer's vertex buffer is only force flushed at the end of each render pass. This means that if you only draw using the methods provided by USDrawBuffer, then everything will be fine. If, however, you need to bypass USDrawBuffer and use OpenGL directly, be sure to call USDrawBuffer::Flush () before you make any direct changes to the graphics state (including loading matrices) and USDrawBuffer::Reset () when you are done. If you don't, subsequent drawing performed using USDrawBuffer may be corrupted.

Of course, for all 2D drawing USDrawBuffer is recommended. USDrawBuffer will eventually be extended to wrap all OpenGL state.

MOAIDeck

MOAIDeck is just a collection of objects that may be drawn or queried in some way. All decks may be indexed by a single number or an array of numbers (in the case of a tiled grid).

Single objects such as MOAIGfxQuad2D are represented as decks of only one object.

To create a new Deck, inherit MOAIDeck then implement any of the virtual methods the deck supports. Typically you will at least implement Draw () and/or DrawDebug ().

If your deck is to be used by MOAIProp2D, also implement Bind () and GetBounds (). It is good practice to implement these anyway. Bind () should set up the Deck's rendering state and return true or false based on whether or not the deck is in a renderable state. You can use Bind () to abort rendering if the Deck is not ready. GetBounds () should return the model space bounding box for a Deck item based on index. MOAIProp2D will use this to set the dimensions and partition placement of the Deck item.

As an optimization, you may also set the Deck's content mask. You should do this in the Deck's constructor. This mask will automatically be applied to any Prop to which the deck is attached. It will then be used to filter queries for the prop based on the Deck's capabilities. For example, Decks meant to be used as collision data or regions may not implement Draw () but might implement DrawDebug ().

MOAITransformBase

This class is one of the simplest to extend. The only requirement is to at some point update the two affine member transforms, mLocalToWorldMtx and mWorldToLocalMtx. These updates will typically be performed in OnDepNodeUpdate (), assuming you want your transform to work with the dependency graph. Of course any other method of causing the transform to update may be used.

An example of extending MOAITransformBase may be seen in MOAICpBody and MOAIBox2DBody, both of which implement MOAITransformBase so that they may be used as parents to MOAIProp (and other transforms).

MOAINode

When extending MOAINode, you must implement OnDepNodeUpdate (). This method is called only after the node has been scheduled. OnDepNodeUpdate () should do any processing the node needs to complete prior to being accessed by dependent nodes.

If you want to define attributes to be driven by Moai's animation system, you will also need to implement ApplyAttrOp (). Attributes are virtual parameters, so they must be resolved using a method on each class that supports them. Moai uses an opaque 'operation' object to inspect and modify attributes. Any operation on an attribute can be supported by passing a value to the operation object and then storing the result. See MOAITransform for an example.

A Note About Multiple Inheritance

The Moai framework makes use of multiple inheritance. In particular, USLuaObject is often joined with a stand-alone class to create a new, Lua-bound class.

So far the decision to use multiple inheritance hasn't given us any trouble. When I decided to use it, it was to build the architecture that conceptually made the most sense to me. Having spent more time developing games on Moai and maintaining the code base, I'm still satisfied with this decision.

That, some things to keep in mind:

Use virtual inheritance to make sure you don’t wind up with extra copies of your base classes.

You may encounter compiler warnings about ambiguous resolution of virtual methods. For example, virtual serialization methods may be implemented by multiple classes in your inheritance graph. If you don’t explicitly implement these methods in your leaf class, you will get warnings. To get rid of the warnings, implement the virtual methods in question in your leaf class.

Be sure to also call any virtual initialization methods explicitly for parent classes. For example, in RegisterLuaFuncs () be sure to call the same for each class you are deriving from before registered the derived class’ Lua functions.

Finally, when using multiple inheritance don’t forget to inform USLuaObject’s RTTI about which classes you are deriving from in your constructor. If you don’t then RTTI will break and you will have trouble getting your class back from the Lua runtime using supported methods.