An Introduction to jit.gl.lua

jit.gl.lua is an OpenGL Jitter object that runs Lua scripts for drawing in 3D. It is similar in nature to the **js** object but has alot more features for 3D grahpics. As an example, **jit.gl.lua** provides direct access to much of the OpenGL API, which otherwise is only available through **jit.gl.sketch** or writing C externals. In addition to the OpenGL functionality, **jit.gl.lua** provides full access to all Jitter objects including **jit.matrix** and **jit.submatrix** for doing matrix processing in Lua scripts.

Why would I want to use it?

The biggest reason to use **jit.gl.lua** is to have the ability to mix Jitter objects with direct OpenGL commands. For complex 3D scenes, the constant OpenGL state changes that jit.gl.* objects do can cause serious performance hits. Often only specific sections of OpenGL state need to be changed from the drawing of one 3D object to another. Current hooks in Jitter for doing this don't scale well and are inadequate for advanced users. Strategies such as jit.gl.multiple are limited by the **jit.matrix** structure of its input data and funnelling everything through **jit.gl.sketch** is inelegant at best and another performance bottleneck at worst.

Additionally, Lua is a lightweight, fast, and highly expressive scripting language. Lua's only data structure is an associative table that serves as array, map, and structuring element for language constructs. The concept of a table is further extended in Lua to the idea of metatables - special tables that handle meta-events and can be used to model relationships between objects in Lua. Lua itself is not object oriented but is in fact a functional programming language where functions are first class types, behaving the same as numbers or strings. Despite the fact that Lua has no built in support for object oriented concepts like inheritance, metatables and a little functional programming can be used to easily develop such concepts in Lua and can in fact be customized to the task at hand to more closely follow say Objective-C's idea of OO over C++'s notions.

Other conveniences of **jit.gl.lua** include a fully C-based vector math library for 2-, 3-, and 4-dimensional vector operations as well as quaternion and matrix math, all of which are crucial for generating 3D graphics. There are also bindings for **jit.matrix** and **jit.submatrix** for manipulating matrix data in Lua scripts.

Accessing Jitter in Lua

Most Jitter objects are available in Lua scrips via the commands jit.new() (e.g. jit.new("jit.gl.gridshape", "foo") where "foo" is the context). The exceptions to this rule are jit.matrix, jit.submatrix, and jit.listener. These objects all have special constructors that must be used instead. In Lua terms, Jitter objects are of type Userdata. Userdata simply means that the underlying object is allocated in C and not by Lua as Lua only allocates a handle on the object. Because Jitter objects in Lua are Userdata, they have a table associated with them and in the case of Jitter objects a metatable as well. In fact almost all of the functionality of a Jitter object is in its metatable. This is where methods and attributes of the object are stored.

Accessing attrbiutes and methods of a Jitter object is equivalent to accessing fields of a table. As an example, the following are equivalent:

```
local xfade = jit.new("jit.xfade")
print(xfade.xfade .. " is the same as " .. xfade["xfade"])
```

Although methods are accessed as fields in a table as well, there is a slight difference to how they are called in practice. Jitter object methods need access to the instance of the Jitter object that is calling the method. In languages like C++, the instance of a class is passed in automatically as a 'this' variable. In C, generally the first argument of the function is the instance. In Lua, the ':' syntax for calling a method automatically passes in the instance as the first argument to the function. The following are equivalent way to call xfade's matrixcalc method:

```
local xfade = jit.new("jit.xfade")
xfade:matrixcalc(mat1, mat2)
xfade.matrixcalc(xfade, mat1, mat2)
```

In the case of jit.gl.lua, the metatables of Jitter objects has been designed such that the Lua syntax resembles **JS** in order to ease the language transition. So in fact,

```
xfade:matrixcalc(mat1, mat2)
xfade.matrixcalc(mat1, mat2)
```

will produce identical results although the former sytax hews more to the standard Lua style.

When a Jitter object's attributes or methods are accessed, its metamethods are triggered to properly handle the situation. The attributes and methods of an object are stored in the metatable's 'attributes' and 'methods' fields.

Accessing jit.gl.lua in Lua

The instance of **jit.gl.lua** that a Lua script is running in can be accessed and controlled just like any other Jitter object. It is stored in a special global variable 'this'. Alternatively, jit.gl.lua's attributes can be set as if they were global variables in a script. For example, setting jit.gl.lua's autwatch attribute could be done the following ways:

```
autowatch = 1     this.autowatch = 1
```

jit.gl.lua Lua-specific Attributes

jit.gl.lua has a number of attributes that control how the Lua environment behaves and what it can do. They control the operation of the Lua Virtual Machine and the number of inlets and outlets a **jit.gl.lua** object has.

autogarbage	If set, **jit.gl.lua** will clear all variables from the Lua Virtual Machine before reloading a script
autowatch	If set, **jit.gl.lua** will automatically reload a script when it has been modified
context	Gets the address of jit.gl.lua's render context (if 0, then it doesn't have one)
gc	Turns the Lua garbage collector on and off
inlets	Sets the number of inlets **jit.gl.lua** has
memsize	Reports how much memory Lua is using
outlets	Sets the number of outlets **jit.gl.lua** has

jit.gl.lua Utility Methods

Lua scripts in **jit.gl.lua** have access to a number of globally available utility methods to help manage the scripting environment. They are

importfile()	Imports other Lua scripts in the Max search path into the current script
outlet()	Sends a message out the specified outlet of jit.gl.lua
print()	Replaces Lua's print function with a new one that prints to the Max window
stackDump()	Print the contents of the Lua stack to the Max window

Special Scripting Methods

In order for Lua scripts to properly interact with the patcher environment, the script has to be able to receive signals from particluar events that might affect its operation. As a simple illustration, display lists created before an OpenGL context has been initialized will crash Max. To automatically detect when a valid context is available, certain methods can be defined in the Lua script that will be called when this happens. The following methods when defined will be called depending on the signals that **jit.gl.lua** recieves from the patcher it's in:

closebang()	Called when **jit.gl.lua** gets deleted
dest_changed()	Called when the render context **jit.gl.lua** is attached to gets initialized or changed
dest_closing()	Called when **jit.gl.lua** is disconnected from its current rendering context
draw()	Called everytime **jit.gl.lua** gets the draw() commands
loadbang()	Called when a patcher is fully loaded
script_load()	Called whenever a Lua script is (re)loaded

The Global Metatable

Just as Jitter objects have metatables, the global environment also has a metatable. In Lua, the variable _G hold the global environment. **jit.gl.lua** automatically attaches a default metatable to _G to facilitate the creation of global metamethods. The global metatable can be used to programmatically generate a patcher interface for Lua scripts. Instead of creating a function for each message that a patcher can send to a script, the functions can be generated with a metafunction. When potential messages to a script are small this may seem insignificant, but as a project grows and the underlying code morphs, the interface can automatically adapt with the use of global metamethods

OpenGL

OpenGL methods are available in **jit.gl.lua** scripts under the 'gl' and 'glu' tables. The Lua OpenGL bindings are based on the LuaGL (http://luagl.sourceforge.net/) project. LuaGL implements almost all functions up to OpenGL 1.5. Although this is not the latest version of OpenGL, all of the new functionality (shaders, FBOs, etc) is already encapsualted in Jitter objects and is actually better managed in C rather than in Lua directly. The Lua version of OpenGL functions and constants follows the pattern that the functions have the same name except that there is a '.' between the prefix 'gl' and the rest of the name and that constants are instead strings and don't have the 'GL_' prefix.

As an example, drawing a line from (-1., 0, 0) to (1, 0, 0) in Lua could be done the following way:

```
gl.Begin("LINES")
gl.Vertex(-1., 0, 0)
gl.Vertex(1., 0, 0)
gl.End()
```

For functions like gluPerspective() that have the 'glu' prefix, the Lua equivalent would be 'glu.Perspective()'.

Jitter OpenGL

While most of Jitter's OpenGL functionality is exposed via the jit.gl.* objects, there are a few areas that are used internally in Jitter object but not exposed for external access. Since Lua scripts are the internals of a jit.gl.* object, these methods have been made available. These functions can be found in the table 'jit.gl'. Their C equivalents are mostly found in jit.drawinfo.h in the SDK. The functions are:

jit.gl.active_textures	Returns the number of currently active texture units
jit.gl.begincapture	Sets up and initializes render to texture
jit.gl.bindtexture	Binds a texture
jit.gl.endcapture	Ends render to texture
jit.gl.screentoworld	Converts coordinates in screen space into coordinates in world space
jit.gl.unbindtexture	Unbinds a texture
jit.gl.worldtoscreen	Converts coordinates in world space into coordinates in screen space

There is also one attribute in **jit.gl.lua** particular to OpenGL. **jit.gl.lua** has a context attribute that is read-only. It reports the address of the context that **jit.gl.lua** currently belongs to. If it is 0, it doesn't currently belong to a context. This attribute is useful for storing context dependent reources like display lists that would otherwise be invalid when jit.gl.lua's dest_changed method gets triggered because of a context change.

this.context	Address of OpenGL context that **jit.gl.lua** currently belongs to.

Jitter OpenGL Support

Since many systems can have widely varying support for the various OpenGL extensions, **jit.gl.lua** makes Jitter's t_jit_gl_support mechanism available to Lua scripts. t_jit_gl_support can be accessed by indexing the jit.gl.support table. Accessing this table invokes a metatable which returns the value of the relevant extension.

Jitter OpenGL Error Reporting

When something doesn't draw properly in OpenGL, it is often difficult to pinpoint exactly what part of the code failed. Using the function jit.gl.error(), errors caught by OpenGL can be reported back to the Max window with detailed information as to what exactly went wrong.

Vector Math

To aid in building 3D structures, **jit.gl.lua** provides access to most of the vector math functions found in jit.vecmath.h in the Jitter SDK. These include 2-, 3-, and 4-D vector operators as well as quaternions and 3x3 and 4x4 matrix math. The following is a listing of the modules

vec2	2D vector functions
vec3	3D vector functions
vec4	4D vector functions
quat	quaternion functions
mat3	3x3 matrix math
mat4	4x4 matrix math

In Lua, vectors are defined as tables with a series of numberical values starting at index 1 since unlike C, the first value of an array in Lua is 1. The following adds two vec3 vectors together and returns the result

```
local sum = vec3.add({1., 1.2, -0.9}, {-9., 10., 11.})
```

Jitter Functions

Sometimes it may be necessary to send messages to Jitter objects in a patcher from a Lua script or to get more information on a Jitter object. There are a number of functions in jit.common.h for accessing and querying Jitter objects. These are available in the jit.table. The functions are

jit.attach

jit.attr_get

jit.class jit.classname

jit.findregistered

jit.findregisteredbyptr

jit.new

jit.notify

jit.register

jit.unregister

Matrix

As in Javscript, matrices in Lua scripts are created with a special constructor jit.matrix() which takes the same arguments.

Submatrix

Like Jitter matrices, **jit.submatrix** with the constructor jit.submatrix(). It takes no arguments, same as **jit.submatrix** in a patcher.

Listener

For processing output data from Jitter objects like mouse coordinates from jit.window there is a Listener object that can be created with jit.listener(). It works exactly like JitterListener in Javascript. It takes the name of a context and a callback as its two arguments.

Extensions

Lua is written in ANSI-C and is meant to be embedded in a software environment. As such it is expected that code written in C-type languages will be made accessible from the Lua environment and the authors of Lua have made wrapping such code in Lua convenient and straightforward to implement. Supplementary libraries can be added to the **jit.gl.lua** object's environment by importing them as Lua modules.

To import modules, the Lua.framework must be installed in */Library/Frameworks*. An Xcode project for build the Lua.framework is located in *Lua_dev/Framework*. Additionally, a sample Xcode project for building Lua modules is available in *Lua_dev/Package*. To build it, Lua.framwork must be in */Library/Frameworks*. After it is built, install it in *C74/lua/luamods*.

For more information on how to build your own modules, see the documentation at lua.org. A repository of available Lua modules that others have made is accessible at luaforge.net.