Breif information about LibRocket/Lua on the PC and Zenterio Platform

Preface

Lua is a powerful, lightweight and high performance scripting language developed back in 1993 by the computer graphics group at the pontifical catholic university of Rio de Janeiro.
Due to it's light memory usage and high runtime performance it has gained a lot of traction as an embedded scripting language is various tools and games through the years.

libRocket is an graphical layout engine which combines a subset of HTML with a subset of CSS (referred to RML <Rocket Markup Language> and RCSS <Rocket Cascade Style Sheets>)
libRocket does not render anything on it's own but instead just layouts and sends the processed data to a implementation specific graphics backend.
Due to the simple nature of libRocket it has very limited builtin element types, but relies heavily on RCSS pattern matching to style and layout elements.
libRocket itself is written in C++, but has support for both Lua and Python bindings in the main repository.

Getting started (quick Lua recap)

Before you get started it's important to grasp the fundamental concepts of Lua (which are not covered in this tutorial), I'll make some comparisons to javascript to high light the similarities and differences.

Additionally it's essential to understand the metaprogramming aspect of Lua as the libRocket bindings relies heavily on this (and unfortunately the libRocket Lua manual is inaccurate; probably because it appears to be mostly a copy of the python manual with some syntactical elements modified)

Lua does not have true classes, but it supports encapsulation by various means and heritage through it's metatable, which is somewhat similar to the prototype chain in javascript.
In Lua the basic object is a table, the table has a few hidden entries; __index, __newindex etc; these are hooks for it's meta programming.

For example; to define a table you simply write

ATable = {}

You can then assign a function to this table using ex

ATable.myFunction = function () print("Hello World") end

and this function can be executed by calling

ATable.myFunction()

This is very similar to javascript

Now lets say you want to inherit this “table” in another table, that’s done like this

```
AnotherTable = {}
ATable.__index = ATable
setmetatable(AnotherTable, ATable)
```

Once the above call is done; you can now execute

AnotherTable.myFunction() and you’ll see “Hello World”

This is the metaprogramming aspect of Lua; and it simply works as following

1. AnotherTable.myFunction internally transforms to AnotherTable[“myFunction”] which will return nil as it’s simply not defined
2. Because we set ATable to be AnotherTable’s metatable the interpreter will in this circumstance internally “refer” to ATables metatable hooks
3. As we defined __index on ATable (which is the __index meta table hook) it will return ATable as the active object and this do ATable[“myFunction”]() and successfully execute

This is commonly exploited using a “constructor” on the base table

```
BaseTable = {}
BaseTable.__index = BaseTable

BaseTable.Create()
    local newInstance = {}
    setmetatable(newInstance, BaseTable)
    — Define any private values on the instance here
    — eg. newInstance.myVariable = 0 etc
    newInstance.baseValue = 10
    return newInstance
end
```

And then used as a “factory” object; to spawn instances such as

myInstance = BaseTable.Create()

Now typically you want to assign “methods” to the base table and this is where things can be a bit confusing (and unfortunately error prone) but is very important

Assume you have

BaseTable.myMethod = function (input)

```
    return input
end
```

You can call this as

myInstance.myMethod(1) as demonstrated above; however you have no way to access “baseValue” on your instance. As the execution context will be “BaseTable” due to the metatable trickery.
This has been rectified using some syntactical sugar in the Lua syntax,

so to start with you need to define your method such as

```
BaseTable.myMethod = function (self, input)
    return self.baseValue + input
end
```

And then you can call

myInstance.myMethod(myInstance, input) and refer to your instance as “self” (similar to “this” in javascript and other languages) however this is not particularly pretty so instead the Lua syntax has support for a “:” (colon) accessor which transparently sends the object itself as the first parameter to the target.

myInstance:myMethod(1)

So to recap, this is equivalent to doing ***myInstance.myMethod(myInstance, 1)***

Similarly if you define your method such as

function BaseTable:myMethod(input) instead of ***BaseTable.myMethod = function (self, input)***

Lua transparently inserts the “self” variable, personally I prefer the explicit case to avoid additional confusion but this is just to demonstrate that the language actually supports it.

Getting started (LibRocket - The PC/Reference environment)

When I’ve been developing in this environment I’ve used the sdl2 application as a host to do PC based development then test the results on the STB,
But the problem is that sdl2 does not by default support the Lua bindings so we need to modify the sdl2 sample somewhat, so lets go through the required steps to get the environment up and running.

To build the sdl2 version you will need to install a toolchain, lua, libsdl and cmake as a minimum (there might be others but that’s left as an exercise to the reader to figure)

Start with checking out ***https://github.com/libRocket/libRocket.git*** using your preferred git client.

Then go into the “Build” directory and modify CMakeLists.txt and change the line ***option(BUILD_SAMPLES "Build samples" OFF)*** to ***option(BUILD_SAMPLES "Build samples" ON)***
Additionally enable Lua bindings if disabled ***option(BUILD_LUA_BINDINGS "Build Lua bindings" OFF)*** to ***option(BUILD_LUA_BINDINGS "Build Lua bindings" ON)***
Also make sure the library_SAMPLES definition includes the Lua libraries, so it should look something like this

```
if(NOT BUILD_FRAMEWORK)
    set(sample_LIBRARIES
        shell
        RocketCore
        RocketControls
        RocketCoreLua
        RocketControlsLua
        RocketDebugger
    )
```

Then go to libRocket/Samples/basic/sdl2/src/ and modify main.cpp as following

```
diff --git a/Samples/basic/sdl2/src/main.cpp b/Samples/basic/sdl2/
src/main.cpp
index 0a1e3b6..710755b 100644
--- a/Samples/basic/sdl2/src/main.cpp
+++ b/Samples/basic/sdl2/src/main.cpp
@@ -98,6 +98,8 @@ int main(int argc, char **argv)
                return 1;

     Rocket::Core::Lua::Interpreter::Initialise();
+
Rocket::Controls::Lua::RegisterTypes(Rocket::Core::Lua::Interpreter:
:GetLuaState());
+
 /*
     Rocket::Core::FontDatabase::LoadFontFace("Delicious-Bold.otf");
        Rocket::Core::FontDatabase::LoadFontFace("Delicious-
BoldItalic.otf");
@@ -108,7 +110,7 @@ int main(int argc, char **argv)
                Rocket::Core::Vector2i(window_width,
window_height));

        Rocket::Debugger::Initialise(Context);
-
+    /*
        Rocket::Core::ElementDocument *Document = Context-
>LoadDocument("index.rml");

        if(Document)
@@ -121,6 +123,8 @@ int main(int argc, char **argv)
        {
```

```
                fprintf(stdout, "\nDocument is NULL");
        }
+*/
+
Rocket::Core::Lua::Interpreter::LoadFile(Rocket::Core::String("start
.lua"));

     bool done = false;
```

This will enable Lua in the sdl2 sample, once this done traverse back to libRocket/Build
and run ***cmake .***

It will generate the required Makefiles and resolve dependencies; look throughly at the list to make sure that Lua and SDL are found
(Sidenote: If building on Mac OS X, make sure you have the i386 versions of the required dependencies as there are Carbon dependencies and Carbon is not available for x64 so it will not build unless this is resolved, macports installs the FAT binary version (both i386 and x64) if you install the packages by postfixing **+universal** to the package name)

Finally build the project and the build directory should contain the sdl2 executable (or on Mac OS X the SDL2.app; you can simply copy SDL2.app/Contents/MacOS/sdl2 to your working directory and launch it via the terminal instead as you will need to keep the file (start.lua) in the same directory as the application which is a bit cumbersome if it remains in the application container.

To verify the installation create a simple start.lua file that just contains a single ***print("hello")*** and start the executable ./sdl2 the console prints "hello" the lua integration is successfully working and executing in the librocket context.

Basic libRocket usage

libRocket exports the "rocket" global which is the only entrance to libRocket you have when start.lua executes, it's important to notice libRocket uses the "class" system described above thus ":" is used for all method calls if you don't use ":" it simply will not work as you expect, so in case of errors that's the first thing to investigate.

A basic start.lua

looks something like

context = rocket.contexts["main"]
defaultDocument = contextt:LoadDocument("index.rml")
defaultDocument:Show()

Once you launch "./sdl2" this will load, parse and finally display the RML file named "index.rml" once performed the lua interpreter exits and further actions are event driven.

On the STB you will have to make a few changes to successfully start (read the known issues document to outline the required changes).

The actual command to launch the Rocket environment on the STB is "***/var/apps/Rocket/bin/bcm7241/RocketApp***"
eg. `/ # /var/apps/Rocket/bin/bcm7241/RocketApp <path to file>`

End notes:

Events on the PC you can use libRockets event system, unfortunately it's not integrated in zenterios build so you have to handle input events via a side channel call "onKey", this is of course not an issue once Zenterio has their PC based simulator ready but until then you will have to live with two input systems (or if they adapt to the libRocket integrated handling which of course would be preferable)

Further more another difficulty with the current zenterio handling is the fact it does not provide the origin document from which the event originated, which means you will have to expose all documents loaded globally to be able to access and operate on them as a reaction to events, this may not be an issue for a smaller project but can be an issue if you try to write something larger and try to keep a modular design as you are significantly more likely to eventually collide with other global variables named the same way- especially if written by different developers.

(See the provided code how my input handling works, even though I didn't fully finish the context routing the basic idea might be obvious even though the limitation is that only the last activated document will trigger input events)

The lua Rocket API is outlined here ***http://librocket.com/wiki/documentation/LuaManual/APIReference***

Regarding RML I won't cover it here as the supported elements (attributes on an element object ***http://www.librocket.com/wiki/documentation/LuaManual/Elements***) are well documented,
Same goes RCSS (***https://librocket.com/wiki/documentation/RCSS/PropertyIndex***) which lists all supported CSS tags (as well as variants of the listed which are not supported)

And to repeat whats been stated several times already, it's important to note that libRocket uses the class hierarchy and expects the self parameter or will silently fail, this can unfortunately be difficult to spot but it's been the most common pitfall in my experience and to make matters worse certain calls will work just fine without being colon addressed (they just don't use the "self" variable internally).

Finally it's probably worth mentioning the only RML tags libRocket actually has any knowledge of are <img> (display an image) and <handle> (used for mouse operations, primarily drag n drop so it's not interesting for our use case)
So the "magic" with libRocket is RCSS styling via custom tags, i.e. you define your own tags and use RCSS to style it- so libRocket doesn't have any knowledge of
 <h1> etc, it's totally up to you to define the tags and set their characteristics

using RCSS.

Also in the official repository there is currently no support for css transformations, to add to this matter Lua has no builtin “timer” source; thus by default transitions can’t even be emulated in pure Lua.

On the Zenterio platform there is an API to create timers but at the time I wrote this this API was not functioning as expected (See known issues document)

This should be enough to get you started and going with the basic environment