Using Lua as Script Language in Games Coded in Java

Gustavo Henrique Soares de Oliveira Lyrio
Roberto de Beauclair Seixas

Institute of Pure and Applied Mathematics – IMPA
Estrada Dona Castorina 110, Rio de Janeiro, RJ, Brazil 22460-320
e-mail: {glyrio,rbs}@impa.br

Computer Graphics Technology Group – TECGRAF
Catholic University of Rio de Janeiro – PUC-Rio
Rua Marqus de So Vicente 255, Rio de Janeiro, RJ, Brazil 22453-900
e-mail: {glyrio,rbs}@tecgraf.puc-rio.br

KEYWORDS
scripting language, Lua, Java, LuaJava, language binding.

ABSTRACT

Lua is a programming language that has been well accepted by the game development community as a script language. That is because Lua offers a series of methods to allow the use of C functions inside Lua code and vice-versa. When developers choose to code a game in Java, apparently Lua is not an option anymore.

The objective of this work is to show that Lua can also be used as script language for games coded in Java. For that developers just need to know the LuaJava library and a few tips.

INTRODUCTION

Through the years of game development the use of scripts has became more and more popular. Today almost all games use scripts in many different aspects. Describing attributes of different characters, objects, creatures, coding artificial intelligence, dialogs, events, history or even in configuration files. Scripts can even provide players a tool for build their own game modifications. Scripts are essential nowadays.

When Lua language was born the majority of game developers where coding in C/C++, and they loved Lua because with almost no effort they could exchange data with a script language that is easy to write, read and even extend. That's because Lua is open source and also coded in C. So that's how Lua became popular.

As a little example of the Lua's popularity between the game development communities, we can list some notable games that use Lua as scripting language and with other roles. Crysis, Far Cry, Grim Fandango and Escape from Monkey Island, Grand Theft Auto San Andreas, Ragnarok Online, SimCity 4, Star Wars Battlefront and Battlefront 2 also Empire at War, World of Warcraft and many others.

Nowadays we can see a lot of games and game engines that use Java as a programming language. The idea of this work is offer to developers that use Java a way to also use Lua in their game scripts.

The advantages of using a script language are well known. It provides rapid development and easy of deployment, because you don't have to recompile de code after every change [Cassino et al., 1999]. Also, script languages offer good integration with existing technologies such as programming languages and are easy to learn and use. And, we can also say that script languages provide dynamic coding because its code can be generated and executed in runtime.

But, as everything that comes with advantages, brings together disadvantages, scripting languages are not different. They assume a presence of a "real" programming language. They are not conductive to best practices in software engineering and code structure, such as object orientation. And also they are tuned toward a specific application, such as PHP for World Wide Web.

That's where Lua come in hand. The Lua language was not created just to be a scripting language, but a short, efficient and extensible programming language [Ierusalimschy, 2006]. So it brings the script languages advantages, but doesn't come with the disadvantages.

Lua offers extremely fast development and is also very easy to learn, write, and understand. It's interpretable

so, you don't have to compile. It also offers dynamic code and integrates with C programs. And with the LuaJava library, integrates with Java too.

About the disadvantages of the language we can say it's true that if the programmer doesn't use discipline, Lua code can become a mess. But the language syntax allows the implementation of object orientation or component based development [Ierusalimschy, 2006]. Also Lua is a general purpose language, so it can handle a lot of different applications using a lot of extension libraries such as LuaSQL, CGILua, and IupLua and, at least but not last, LuaJava.

THREE TIPS FOR GAME SCRIPTING USING LUAJAVA

LuaJava is a scripting tool for Java. The goal of this tool is to allow scripts written in Lua to manipulate components developed in Java [Cassino et al., 1999]. LuaJava allows integration between Lua and Java in both directions: manipulation of Java objects by Lua scripts and manipulation of Lua objects by Java programs [Cassino et al., 1999]. The access to Java components is made from Lua without any need for declarations or any kind of preprocessing.

In this paper we have choose to work using Java objects inside Lua scripts because in that way we can create generic objects and make scripts to change the value of its attributes producing a clean, and easy to understand, code.

The first thing we need to do when we want to handle Java objects inside Lua is to create a `LuaState`. The `LuaState` will control access to the Lua environment. We will be able to create a `LuaState` by doing this:

```
LuaState luaState;
luaState = LuaStateFactory.newLuaState();
```

After creating a `LuaState`, we now need to open the LuaJava libraries. This will be done by the following instruction:

```
luaState.openLibs();
```

Now that we have built our environment we are ready to start writing Lua code. This should be done in a Lua file (.lua). Now that we have an environment and Lua file, we just need to put all together. It should be done calling the /tt LdoFile method. This method tells the Lua environment to read the Lua file passed as a parameter. The instruction should be:

```
luaState.LdoFile(<luafile location>);
```

Once we have seen all the methods to create a Lua environment and use it inside a Java project, let's put all together in a classic Hello World example.

```
void Main() {
LuaState luaState;
luaState = LuaStateFactory.newLuaState();
luaState.openLibs();
luaState.LdoFile("helloworld.lua");
luaState.close();
}
```

`helloworld.lua file:`

```
print("Hello World")
```

LuaJava also offer many other features. We will take a closer look at two of these features which will be important to our scripting schema figuring in our second and third tips. The first feature is how you call a Lua function to be executed inside the Java scope. The second one is how do you use a Java object inside a Lua file.

To use a Lua function in Java you need to get the Lua global variable that stores the function, that will be done by calling the method `getGlobal` passing the function's name as a parameter:

```
luaState.getGlobal(<function name>);
```

After that we just use the `LuaState` call method. That method is particularly important because it first parameter is the number of parameters passed to the function. But how do you pass those parameters? That will be explained by our third tip.

To pass a Java object to Lua we will use the method `pushJavaObject` and pass it as a parameter of a function. The instruction sentence is:

```
luaState.pushJavaObject(<object>);
```

So, with these two features we will be able to pass a Java object as a parameter to Lua function. Use it inside that function and call the function inside the Java scope. That will allow us to create a class that will handle scripts for us as we will see in the next section.

Building a LoadScript class

Now that we have seen how LuaJava works, let's build a class that will handle all scripts in our game. That

class will have only one attribute, our `LuaState`. The constructor of our class will receive only one parameter that will be the path for our script file. Inside our constructor, we will create the Lua environment with `LuaStateFactory.newLuaState` and `openLibs` methods, and execute the Lua file received as a parameter by the constructor.

Our class will have two methods: `closeScript` and `runScriptFunction`.`closeScript` just call `luaState.close` to terminate the use of Lua environment.

`runScriptFunction` will get a Lua function received as parameter and call it passing a Java object, also received as parameter, to that function.

We have built a class to handle all our scripts. In the next section we will see how we use that class.

LOADSCRIPT CLASS

```
import org.keplerproject.luajava.LuaState;
import org.keplerproject.luajava.LuaStateFactory;

public class LoadScript {
  LuaState luaState;
  /**
   * Constructor
   * @param fileName File name with Lua script.
   */
  LoadScript(final String fileName) {
    this.luaState = LuaStateFactory.newLuaState();
    this.luaState.openLibs();
    this.luaState.LdoFile(filename);
  }
  /**
   * Ends the use of Lua environment.
   */
  void closeScript() {
    this.luaState.close();
  }
  /**
   * Call a Lua function inside the Lua script to insert
   * data into a Java object passed as parameter
   * @param functionName Name of Lua function.
   * @param obj A Java object.
   */
  void runScriptFunction(String functionName, Object obj) {
    this.luaState.getGlobal(functionName);
    this.luaState.pushJavaObject(obj);
    this.luaState.call(1, 0);
  }
}

}
```

Using Lua script files

For a short example of everything that we have saw until now, let's consider a game with a huge among of different monsters. It could be something like Blizzard's Diablo or World of Warcraft. Our monsters will have four different attributes: race, life, attack and defense. Let's suppose that the game has one hundred different kinds of monsters, it one with different attribute values. What would you do to model those monsters? Build a monster class and one particular class for each kind of monster? That would be very bad.

You should consider build the monster class with our previous made script class.

The monster class will receive a new attribute called script. The class constructor will receive the new monster race as a parameter and load its script calling the `LoadScript` class. After that, we just call the method `runScriptFunction` calling "create". Then each race will have a Lua script file that will load the monster instance with the race attributes.

Let's take a look in the code:

MONSTER CLASS

```
public class Monster extends Creature {
    /* Info */
    protected String race;
    protected int defense;
    protected int attack;
    protected int life;
    /* Script */
    private LoadScript script;
    public Monster(String race) {
    /* Loads Lua script for this race.*/
    this.script = new LoadScript(race+".lua");
    /*Call Lua create function.*/
    script.runScriptFunction("create", this);
  }
  public String getRace() {
    return race;
  }
  public int getDefense() {
      return this.defense;
  }
  public void setDefense(int defense) {
      this.defense = defense;
  }
  public int getLife() {
      return this.life;
  }
  public void setLife(int life) {
      this.life = life;
  }
  public void setAttack(int attack) {
      this.attack = attack;
  }
  public int getAttack() {
      return this.attack;
  }
}
```

Analyzing the code above us can see that the first line is the monster class declaration. Then the next four lines declare the class attributes relative to the information about monsters. Next we have an attribute that is our brand new class `LoadScript`. After the attribute declarations we can find the class constructor. As we saw above, the constructor receives a string with the monster's race and in its first line call the `LoadScript` constructor to store in the attribute script the Lua file that stores the values for the attributes of that monster race. The next line calls the Lua function create that will set the new monsters with the values set in the script. The next lines are just some gets

and sets methods to be used inside Java scope if needed.

The following code shows how a Lua script should be:

SAMPLE SCRIPT FILE

```
function create(monster)
   monster:setRace("Sample Monster")
   monster:setDefense(10)
   monster:setAttack(10)
   monster:setLife(100)
end
```

The script file consists only in a Lua function that call the set methods defined inside monster Java class, setting the values for that specific monster race. To create a new monster the developer just needs to copy the file, change its name to the new monster race name and the values passed to the methods inside it.

RESULTS AND CONCLUSIONS

The presented technique has been used to build both monster and players scripts for an experimental 2D MMORPG that's still under construction.

The Lua scripts made the insertion of new characters (monster or players) pretty easy, because produced a very clean and organized script file and made us able to copy an already made script, just replacing the values with the data from the new character. That made possible to produce new monsters for the game as fast as we could generate new combinations of attribute values.

For a quick development of a test platform we made use of Golden T Game Engine (GTGE) that is freeware and offers an advanced cross-platform game programming library written in Java language. GTGE is developed by Golden T Studios. Also we have used some graphics of RPG Maker XP for testing purposes only. RPG Maker XP is developed by Enterbrain, Inc. (Figure 1).

FUTURE WORKS

We intend to continue to work in the development of the 2D MMORPG using the Lua scripts. As project next goals we can detach: replacing the RPG Maker XP graphics and make use of LuaJava library to bind other Lua libraries such as LuaSocket (for network communication) and LuaSQL (for database access), into Java code.

REFERENCES

K. Arnold J. Gosling, 1997, *The Java Programming Language. 2nd Edition*, Addison-Wesley.

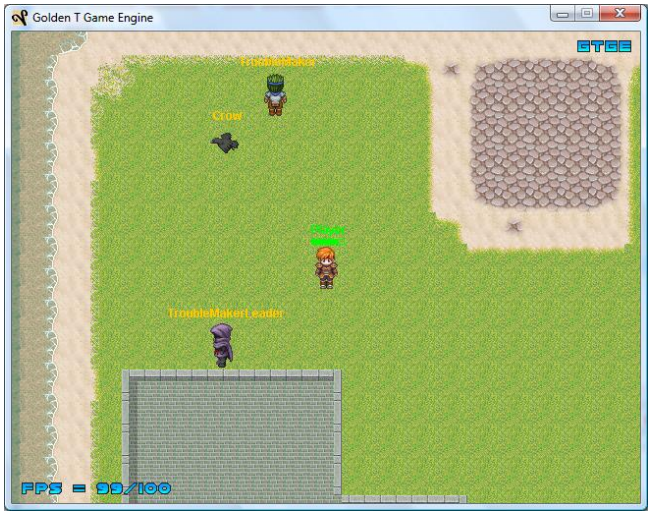

Figure 1: Screenshot of the test environment builded using GTGE and RPG Maker XP graphics. You can see the player character in the center, and three different monsters (Crow, TroubleMaker, TroubleMaker Leader) created using Lua script files.

R. Ierusalimschy, 2006, *Programming in Lua. Second Edition*, Lua.Org.

R. Ierusalimschy et. al, 2006, *Lua 5.1 Reference Manual*, Lua.Org,

C. Cassino et al., 1999, *LuaJava - A Scripting Tool for Java*, PUC-RioInf.MCC02/99, February.

BIOGRAPHY

ROBERTO DE BEAUCLAIR SEIXAS works with Research and Development at Institute of Pure and Applied Mathematics - IMPA, as member of the Vision and Computer Graphics Laboratory - Visgraf. He got his Ph.D. degree in Computer Science at Pontifical Catholic University of Rio de Janeiro - PUC-Rio, where he works with the Computer Graphics Technology Group - TeCGraf. His research interests include Scientific Visualization, Computer Graphics, High Performance Computing, GIS, Simulation Systems and Warfare Training Games. Currently he is the advisor of the Warfare Games Center of the Brazilian Navy Marines Corps.

GUSTAVO HENRIQUE SOARES DE OLIVEIRA LYRIO works with the Computer Graphics Technology Group - Tecgraf. He got his B.Sc. in Computer Engineering at Pontifical Catholic University of Rio de Janeiro - Puc-Rio. His interests include Computer Graphics and Warfare Training Games. Currently he is developer of the Warfare Games Center of the Brazilian Navy Marines Corps.