

Game Scripting 101

By Steve Jones
October 4, 2006
Game Institute

What you will learn:

- What is a scripting language?
- Scripting and Game Programming
- Embedding the Script Engine

Introduction

I thought I would start this series of scripting tutorials with a basic understanding of using scripting in games. This first article is by no means able to even scratch the surface on understanding a particular script language. In fact, I'm not really going to talk about any particular script language in this article.

What is a Scripting Language?

Scripting languages are computer programming languages used primarily to reduce the edit-compile-link-run process. Scripting languages were first used as batch command tools. They were typically used to run a specific set of easily editable tasks or commands on a computer. Later, scripting languages started showing up in computer games. I will discuss in more detail why and how they are used for games later in this article. Some script languages have another important role and that is to extend some other programming language like C or C++. Some languages are stand-alone, general purpose languages like Perl, Smalltalk, and PHP but others like Python, Lua, AngelScript, and Perl, to name a few, can be embedded within the native code of the application and serve to extend its set of available functions and types. That's the type of scripting language I will be talking about.

One of the biggest downsides to script languages is the execution speed. Typically scripts run a little, or sometimes a lot slower than compiled code like C and C++ executables. The reason for this is the scripts are interpreted by a thing called a virtual machine (VM). A VM is a separate piece of software that isolates the main application from the computer the application is running on. The VM places itself between the application and the script. A script is interpreted by the VM at run-time and if you have embedded the VM in the main application or game engine, in our case, you now can have access to everything in the script, like functions and variables or other objects. And, likewise, the script can have access to the application's functions and objects. You have now just *extended* the native language because you can write new functions and such in the script file and use it in the application.

Another common name for embedding the scripting language into other languages is “gluing”. If you don’t see it yet, you will see the power in being able to do that for games.

Scripting and Game Programming

Why would you add a scripting engine to your game? Scripting gives you the ability to code your game’s major functionality without having to compile the game engine at all! Some of those game components are Artificial Intelligence (AI), User Interface (UI), game events, save and load game functionality and so on. Scripting also provides the vehicle for modders to change the way a game currently runs. The modder can change the look-and-feel and the behavior of commercial games that have scripting engines included. They do that sometimes so much that the game looks completely different, like a brand new game and they did all that without touching or having the ability to touch game engine source code.

Scripting can allow people of a game company of various disciplines to work on their piece of the pie, if you will, and not have to ask the C++ programmers to change one thing in the game engine. Because of the slower execution speed of the script compared to native code, you would not want to write any high performance code using it. C++ functions can be written in the engine so that the script can get access to it.

For example, say that you have a class for a tree mesh that will be placed somewhere in your scene. That class can be written in C++ and the rendering code is also in C++. You want the faster rendering possible. Now let’s say that you want the flexibility to put an object of that tree type anywhere in your scene in 3D space. Well, you don’t really want to “hardcode” the x,y,z coordinates of the world space position into the C++ code because in order to change it for any reason, you would have to recompile the whole or part of the application. That’s not so good! If you put the x,y,z values in a script you would be able to position the tree anywhere you want without even touching the C++ source code. In fact, let’s go one step further. The script engine could also have the ability to create an instance or object of that C++ class before it places it in the scene. I hope you see the power here. Your level designer can design away to their heart’s content without bugging the game engine programmer. That’s always a good thing!

The different programmers working with the script language could create all their own script files without interfering with other programmers. The UI programmers could use a copy of the current release of the game engine to create and place their UI objects on the screen. At the same time the AI people are developing paths and decision trees in script.

Here is an example of what a UI script writer might use:

```
// Script code
CreateText(1, 100, 200, “My Text Label”) // ID, X, Y, string
SetItemFont(1, “Verdana”, 12) // ID, font, pitch
SetItemColor(1, 255,255,255,255) // ID, Alpha, Red, Green, Blue
```

Each of those script functions would be “registered” C++ functions with script name aliases that when called, would call their C++ registered counterparts. The tutorial series will get into what a “registered” function is.

The tutorial series will also attempt to dive into the various ways a game can take advantage of scripting. Of course, if I’ve left anything out that you would like to see please contact me. I’m probably not going to think of everything.

Embedding the Script Engine

The script engine needs to be bound to the C++ world so that a two-way communication can take place. The following sketch illustrates in a simple diagram the communication pathways.

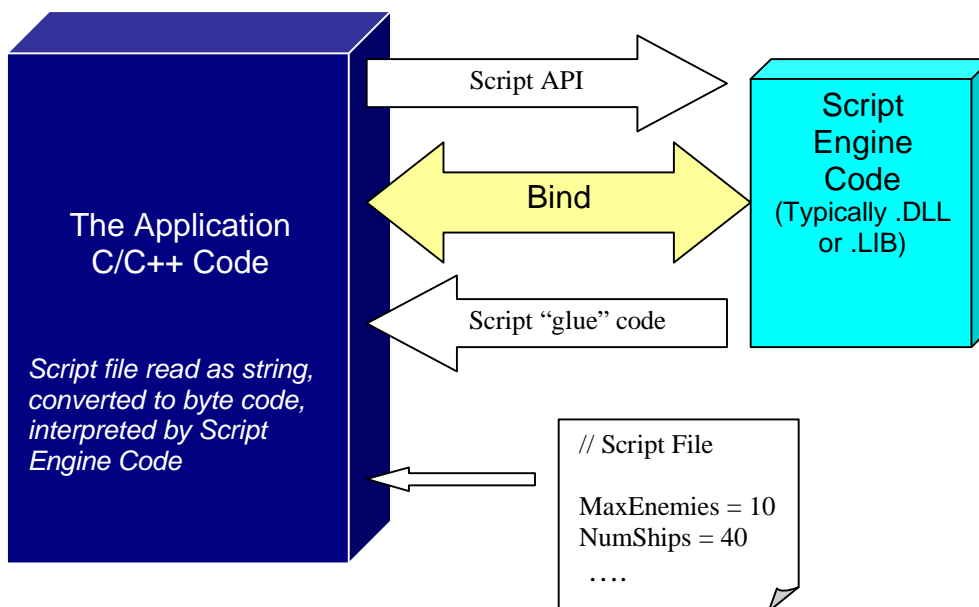


Figure 1. C++ and Script Communication

The script engine code is embedded in the C++ code typically by linking implicitly or explicitly to a DLL or static (.LIB) library. This is not the only way a game scripting engine is going to do it but it does represent the AngelScript and Lua embedding scenario.

The embedded script allows the script programmer to call C++ functions, access local, class and global C++ variables and even C++ classes and class objects as well. In the same manner, the C++ code will have access to local and global variables and functions of the script language. The way this series of tutorials will provide binding to the script code will be static linking the C++ application to the script library. Then a header file for the library will be included in the application. That’s it! Well, sort of. We will need to make certain calls to setup and make the

way to the script engine available to native code. But again, we will cover all of that in the tutorials.

Conclusion

We saw what scripting is and how it can be used for computer games. We saw how it can be embedded into your application on a high-level type of view. You should understand a little bit about what kinds of things we can generally do with scripting in a game engine. Don't worry if all this is still a blur. We will get deep into the details during the tutorial series. My hope is that you'll be able to take away something from these lessons.

References

Lutz, Mark and Ascher, David (2004). *Learning Python*. O'Reilly Media, Inc. ISBN 0-596-00281-5

Schuytema, Paul (2005). *Game Development with Lua*. Charles River Media, Inc. ISBN 1-58450-404-8