You now know pretty much everything needed to program great World of Warcraft addons, but you have learned more than just how to write addons for World of Warcraft. You’ve learned the programming language Lua, which is used by far more applications and games than just WoW. An incomplete list of applications that use Lua can be found on Wikipedia: <http://en.wikipedia.org/wiki/Lua_(programming_language)#Applications>.

There are many games, like Warhammer Online, Crysis, Far Cry, and S.T.A.L.K.E.R, that use Lua in a similar way to World of Warcraft. Besides WoW, the most interesting game that uses Lua for its interface is Warhammer Online, as it also supports addons that are very similar to WoW addons. I’ll show you a simple Hello, World addon for Warhammer Online here.

Another particularly interesting application is SciTE, the editor I presented at the beginning of the book. You can use Lua to customize it to your needs, as I will show how to do in this chapter. We are going to write a function that fixes the indention of Lua code in the editor, which can be useful when you copy and paste Lua code from somewhere like a WoWWiki article or forum post.

And there is yet another project I’m going to show you here in this chapter. Lugre is a Lua binding for the graphics engine OGRE 3D, a fully featured 3D graphics engine that can be used to write your very own game. Lugre allows you to use all of its features from Lua. I’ll show you a simple example of this here because I think it’s one of the coolest projects using Lua out there.

You are probably already using this editor as an IDE for your addons. You can find my customized version of it in the code download if you have been using another IDE or editor. The editor is based on Scintilla, an open-source editor engine used by many editors, most notably SciTE, Notepad++, and Code::Blocks. SciTE is highly customizable; you can change almost everything by modifying the config files, and almost the whole Scintilla API is exposed to userprovided Lua scripts.

Let’s start with a look at the config file. You can open this file by clicking Open Global Options File in the Options menu. I’ve already added two user-defined commands there: a simple Hello, World script and a script that inserts the current time. You can find the definitions of these two commands at the end of the config file

I’m not going into detail on the SciTE config commands here, but you can find the full documentation at http://www.scintilla.org/SciTEDoc.html. The code basically creates two commands that show up in the Tools menu. The second one uses Ctrl-1 as a shortcut, while the first has no shortcut defined. It will automatically use Ctrl-1, as that is the first custom command. These two commands then call the global Lua functions HelloWorld and Timestamp, which are created in the startup script startup.lua. Let’s start with a simple Hello, World script

The file startup.lua is in the SciTE installation directory, but you can also open it by clicking Open Lua Startup Script under Options. We can create the function HelloWorld there

You can now press Ctrl-1 (or select Hello, World from the Tools menu) to execute it; there is no need to restart SciTE. It automatically detects when you modify the startup script and reloads it the moment you save the file. The output we get is simply “Hello, World” in the default output area of SciTE

But printing to the standard output is trivial; it would be more interesting to insert text into the currently opened document. SciTE provides a few objects in the global namespace of this startup script. One of these objects is editor. It provides numerous methods that can be used to insert text, read text, or modify existing text. For example, we can use the method editor:AddText(text) to insert something at the current caret position

SciTE now inserts “Hello, World” into our file when we press Ctrl-1 or select the command from the menu. There are, of course, all the usual Lua functions available, so it is easy to write the command that inserts the current date and time into the document:

But you don’t have to click a command in a menu to execute a Lua function; you can also define event handlers that are called on events like file saves

You can create a function that acts as an event handler by naming a global function after the event. For example, the event OnDoubleClick occurs when you double-click in the editor or output pane. Add the following function to your startup script to display a message every time the user double-clicks

It prints OnDoubleClick every time you double-click, an effective demonstration but not a really useful application. Other event handlers may be more valuable, such as OnBeforeSave, which is executed before a file is saved and receives the name of the file being saved as an argument. You could use it to do a few checks or fixes before writing the file to disk

For example, we can insert a “last modified” time stamp before the file is saved. The following OnBerforeSave handler does this by iterating over all lines, using editor.LineCount to determine the number of lines and editor:GetLine(i) to retrieve a line. It looks for the pattern $Modified.-$ and replaces it with $Modified.timestamp$ by using the methods editor:SetSel(selStart, selEnd) and editor:ReplaceSel(newText)

The strings in this function must not use $Modified...$, because the event handler would otherwise modify itself while editing the file, which would be really annoying. The following code therefore uses \36 (the ASCII code for $) instead of $ to prevent this

The event handler uses the attribute editor.CurrentPos to save the caret position before doing the selection and the method editor:GotoPos(pos) to restore it after updating the file. This prevents the script from changing the caret position and scrolling to the occurrence of $Modified$

This simple event handler allows you to place $Modified$ anywhere in your files; it will automatically add and update a time stamp there every time you save the file

There is a lot of good documentation available for SciTE

A very good tutorial and documentatio

A lot of useful Lua scripts that can be used in SciTE

: Full documentation of all available objects and their methods/attributes

Our next topic is another MMORPG (Massively Multiplayer Online Role-Playing Game), Warhammer Online (WAR). Its interface is written in Lua and can be extended with addons

In this section you will see how a simple Hello, World addon works in Warhammer Online (http://www.warnammeronline.com). There are many similarities between the user interfaces of WoW and WAR; it is easy for a World of Warcraft addon programmer to get started in WAR. The WAR Interface API looks to me like a very unpolished version of the World of Warcraft API; many features are similar to World of Warcraft but more difficult to use or unnecessarily complicated

Addons are placed in the folder Interface\AddOns\<addon name>; you will have to create this folder by hand for your first addon. Create a folder called HelloWorld for our Hello, World addon there. If we were programming for WoW now, the next step would be creating a .toc file. Warhammer Online uses .mod files instead

The main difference between a WoW .toc file and a WAR .mod file is that the .mod file is written in XML. Create a file called HelloWorld.mod in the HelloWorld folder and place the following XML in it

The XML should be pretty self-explanatory. The version of the game the addon is created for is stored in the version attribute of the UiMod element. This attribute is equivalent to the Interface attribute of World of Warcraft. The game will complain that the addon is outdated if this version number is lower than the current version of the game (which is currently 1.2.1)

The XML also contains metadata like the description and author of the addon. It then loads the file HelloWorld.lua, which we are going to write next

It is not necessary to restart the game when you change the .mod file in WAR. It is sufficient to reload the UI by typing /reload

The simplest way to create a Hello, World addon is just to write the code that displays “Hello, World” in this file. You might be tempted to write print(“Hello,World”) there, but print is not available. We have to use the method EA\_ChatWindow.Print(msg) to print a message.

But this function doesn’t accept a normal Lua string; we have to convert it to a UCS-2 encoded string. This is the character encoding that is used by WAR, and it basically means that every character consists of two bytes. There is a simple function available that takes a normal string and converts it to UCS-2 so it can be displayed: L. Recall that you don’t need parentheses for a function call that only takes a single string or table. We can therefore simply write the following line into our Lua file and reload the interface with /reload

We now see “Hello, World” in the chat frame after reloading the UI

Use the command /debug and enable logging in the opened menu to see Lua errors in WAR

You might now expect the next step in the Hello, World addon to be adding a slash command. But slash commands are really complicated in Warhammer Online. The only way to add a slash command is by hooking a function provided by the game. There is a library available that does the hook and provides a function to register slash commands: LibSlash, which can be downloaded on war.curse.com. It is easy to use and it comes with its own documentation, so I’m skipping slash commands here. Event handlers are more interesting

There are two types of event handlers in Warhammer Online: events that are defined in the .mod file of an addon and game events. The former event type is the equivalent of script handlers in World of Warcraft, while the latter is similar to WoW game event handlers

Let’s start with the events in the .mod file. There are only three event handlers available

To test the OnInitialize handler, add the following element to the UiMod element of your .mod file

This simply calls the function stored in the global variable HelloWorld\_Initialize. Let’s create this function in our Lua file

This produces a message in your chat when you reload your interface

The second type of event handler can be registered with the function RegisterEventHandler(event, func). The event argument is not a string but a number that identifies the event. All events with their corresponding IDs are stored in the table SystemData.Events. Similarly, func is not a function but a string that holds the name of a global variable that contains a function, which will then be called when the event occurs. It is also possible to pass a string in the format “someTable.key” to call the function stored under key in the table in the global variable someTable.

Let’s use the event CHAT\_TEXT\_ARRIVED for testing; it is fired every time you receive a chat message. A full list of events is available at http://www.thewarwiki.com/wiki/Event\_List. We can’t pass the event’s name as a string, so we need to get the ID of the event from the table SystemData.Events, Our function call then looks like this:

We can now create the function HelloWorld\_OnChat, which is called every time the event occurs. One might expect this function to receive arguments that contain the sender of the chat message and the text, but this is not the case. The event arguments are stored in GameData.ChatData. The name argument stores the player (or NPC) who sent the message, and text stores the actual message. Let’s write a function that simply displays a message in your chat saying <sender> message

This sounds easy, but the name and text fields from this table are not strings, they are wstring. A wstring is a data type introduced by WAR; it is one of the multi-byte UCS-2 encoded strings that we also needed for the function Print. The problem is that you can’t mix normal strings with wstring in concatenations. A possible solution to this seems to be using string.format, but this does not work with wstring at all

The game never automatically converts between wstring and normal strings; you will always have to do this by hand. You can use wStringToString to convert a wstring to a normal string, but you risk losing information if the string contains special characters

Those are the basics of using Lua for Warhammer Online addons. It is slightly more complicated than the World of Warcraft API; in particular, having two different data types for strings can be extremely confusing and annoying. But it’s still Lua and therefore not too hard to understand and learn the API

There are many very good web sites out there if you want to learn more about addons in Warhammer Online

This wiki is the equivalent to WoWWiki in Warhammer Online

: Curse is not only for World of Warcraft addons; you can also find many Warhammer Online addons there

You now know how you can use Lua in two different games, but wouldn’t it be even cooler to write your very own game? You can do this with Lua by using the engine Lugre

Lugre (http://lugre.schattenkind.net) provides a Lua API for OGRE 3D and can be used to write a whole game just with Lua. This section is a little more advanced and requires basic knowledge of 3D graphics. I’m not explaining every single term here, because the whole topic is very complex and can literally fill books. There is in fact a good book about OGRE 3D available: Pro OGRE 3D Programming, by Gregory Junker (Apress, 2006). It doesn’t cover Lugre, but you need to know how OGRE 3D works in order to use Lugre, which simply forwards most of your function and method calls to the corresponding OGRE functions and methods

The simplest way to install Lugre is to get the precompiled binary version with a small example project from the SVN repository: svn://zwischenwelt.org/lugre/trunk/example. In case you don’t know what Subversion (SVN) is for, it is a version control system, managing the source code of a project. It allows you to switch between different versions of a project and have multiple programmers working on the same project as it synchronizes the files

What this means for you is that you will need a client that uses the Subversion command checkout to get all the latest versions of all the code that is stored in this repository. A very good client for Windows is TortoiseSVN, which can be obtained on http://tortoisesvn.tigris.org/. You will have a few new commands in your context menu in your Explorer; one of them is SVN Checkout. Create a new folder for Lugre and run this command. Enter svn://zwischenwelt.org/lugre/trunk/example when it asks you for a repositor

Subversion is not only useful for relatively big projects like Lugre. I use it for every addon that is longer than a Hello, World example, as it is an incredibly useful tool to have. Curse.com offers free Subversion repositories for addons. There are also tutorials available on setting it up, but this is fairly easy with TortoiseSVN

Lugre comes with a small example project: you can run the file example.exe in the folder bin to start it. It shows a tree in a skybox, nothing really impressive. But consider that the whole code that is responsible for creating and managing this scene is written in Lua. The next example shows how this works

The code responsible for this application can be found in the file main.lua in the folder Lua. We will not delete the whole file, as it also contains a lot of boring initialization code that we are going to reuse for our example.

Open the file and navigate to the function Main, the main function that is called by Lugre when the program is started. Find the following two lines at the beginning of the function

Now delete everything between these lines (keep the call to Bind so that you can take a screenshot by pressing V) and the following lines

It is important that you do not delete these two lines, as this is the main loop of the program. It calls the function MainStep repeatedly until you close the program; MainStep can be found right below the function Main. This function calls other functions, which handle user input and draw frames. We will not change anything in this function; we will only replace the code we deleted earlier

We will now create a simple scene that is similar to the original example, showing a few trees with different parameters and a simple skybox

A tree is created from a CaduneTreeParameters object, an object that stores various attributes of our tree. Multiple trees can use the same CaduneTreeParameters object to create multiple trees that look alike. These objects are not directly part of OGRE 3D; they are provided by an addon for OGRE, Cadune Tree. But Lugre already includes a lot of extensions and addons for OGRE to make your life easier

The tree itself consists of two different GFX objects, one for the stem and another one for its leaves. These two objects need to be placed at a specific position within our scene. We will place the tree-creation in a small auxiliary function so we can quickly create different trees without having to copy and paste a lot of code. Create the following function above the main function, as it needs to be visible there:

The function takes three parameters that determine the look of the leaves, followed by three parameters that define the position of the tree in our small scene. We can now create our first tree by placing the following line into the main function at the point where we previously deleted the old example code

This creates a nice-looking tree in the left area of our scene. You can play around with the parameters to modify how the tree looks. Add the following line to create a second tree in the right half of the screen

This creates a tree with a different type of leaves and different parameters. We now need a sky with a sun and moon for our scene

The object we are going to use here is, like the tree object, not directly a part of OGRE 3D but of an additional library included in Lugre, in this case Caelum. It does more than simply display a skybox with a sun and a few clouds; it provides an entire universe for our small scene. It displays a sun and a few clouds or a moon and stars depending on the time. The time can be set and it automatically moves forward with a specified multiplier, so we can easily achieve a day and night effect without writing a lot of code

Place the following lines right under the code that creates the two trees

We now have a simple sky in our scene, but our trees have lost their color: they are gray now. We need to set a few parameters that define how the Caelum object can control the ambient light and fog in our scene. Add the following code to get realistic colors for our trees

The scene still looks static, because we haven’t defined a time scale yet. This means that the time runs in real-time, so you’d need to wait a few hours until it turns night in our little universe. But we can speed this up by calling SetTimeScale on the UniversalClock object that is tied to the Caelum

You can play around with the parameters to achieve different effects. We have created a simple scene, but you can now see that it is possible to create a whole game with this engine

There are many very good web sites out there if you want to learn more about Lugre and OGRE

The official Lugre wiki with a tutorial that covers creating a simple Pong game with Lugre

The web site of the OGRE 3D project, the 3D engine used by Lugre. There are many exciting tutorials, along with documentation for the API and its addons

In this chapter you saw that Lua can be used for more than just World of Warcraft. It is embedded as a scripting language in many games and applications. The language is the same in all of the platforms; only the provided API that consists of functions, variables, and objects differs

But Lua can not only be used inside another application; there are frameworks available that allow you to write a full stand-alone application in Lua. We saw the framework Lugre here, which allows you to write a complete game with just Lua. But there are hundreds of other frameworks out there that add exciting functionality to a Lua program. There are many possible ways to use Lua, if writing World of Warcraft addons ever becomes boring. A list with numerous libraries and bindings that allow you to use other libraries or frameworks can be found at <http://lua-users.org/wiki/LibrarieAndBindings>.