

Welcome to Linden Scripting Language: Introductory, or LSL. My name is Eric Graeb or Erich Hienrichs in Second Life. This tutorial will provide you with the basics of scripting in Second Life, and how to find more information if you are interested in becoming advanced. I will be providing interactive instruction by building a script with you based on the default script created in Second Life. Each portion of the video may have new code on the screen. If you are able, following along by coding the script yourself in Second Life as we progress. At the end of section 3 we will use what you learned up to that point to complete an example script.

In the first section we will go over some of the boring theory you need to know to script well. I will describe the various parts of the user interface. We will talk about what events and functions are and describe the use of variables and constants.

### **About LSL:**

If you've been looking around Second Life you probably have already seen objects containing LSL scripts to cause things to happen in the environment around you. Not all of it was LSL Script though.

LSL script is limited to manipulating prims and objects and using text based communications or chat.

If you've seen an avatar cross their legs while sitting, that's called an animation. An animation can be called by a script so it fires when the avatar sits on an object...but the manipulation is indirect. There are also advanced PHP programs that can be hacked into the interface and SL to perform more advanced functions.

LSL is an event driven script. Code is executed when something occurs, either to the prim or object it is in or an event in the environment it's been

monitoring for. LSL Scripts cannot run while in an avatar's inventory, only in a prim's inventory, while rezzed. Some examples of this are when a prim is taken out of inventory or when an avatar touches the prim.

Unlike the more advanced programming languages, LSL is Single-Threaded. This means it can only process one event at a time. While an event is running any new events are kept in a queue until the script is not busy. But there is a 64 event limit after which anything new is dropped into the ether. For this reason it is essential that anything you code finishes processing as fast as possible.

Each script you create has a limited amount of memory you can use. You are provided with 16K of memory, which is ample for basic scripts, but if the script is to store a list it is quickly used up.

### **The Interface:**

Now that I've bored you with the details, let's create a script so I can describe the interface. Open your avatar's inventory, go to the Create menu and select New Script. A script called New Script will appear in your inventory. I want you to call it LSL Intro Script v1.00. If the name New Script was already applied you could right click the file and select rename to change it.

Now double click the file to open it to see the contents of the script. Note the name of the file appears at the top of the screen, this is why it's a good practice to add the version number at the end of the scripts name. This way if you have multiple versions of the script and you need to have both open at the same time....it's a visual reminder of which one is the newest version.

The text area in the center that contains the multi-colored text is the code window. Each color represents a different type of LSL reserved word and black is unknown.

The menu options are similar to a basic notepad editor. Under the File Menu you have Save, to save your changes and Revert All Changes, which undoes everything back to the last save. There is no auto save.

Under Edit you have Undo and Redo to remove your last change or undo the undo. You also have Cut, Copy, and Paste, very useful for transferring bits of code from one script to the other. Select all highlights all of the code. Search and Replace is the same as Find in other programs. It looks for the word you want and optionally changes it with your replacement. Here we are going to search for the word "Touched" and replace it with "You touched me".

Under Help there is the help function which opens a web page on your pc that's installed with Second Life. This is a good reading resource but is not very searchable. There is another link to go to the LSL Wiki. This contains much of the same data but in a more searchable format. I prefer to use a different site, created and maintained by LSL coders. The site is [www.lslwiki.net](http://www.lslwiki.net).

The second text window is the compiler status window. This is where it will tell you if the script compiled correctly or if you have syntax errors.

The drop down at the bottom left is a list of all reserved LSL words. If you select from the list it will automatically insert that word at your cursor position on the screen. Here we are going to replace the 0 in the IISay function with the reserved word PUBLIC\_CHANNEL.

### **Events and Functions:**

Events are happening all the time in SL. People touch things, collide with objects, talk to other avatars, buy things, and so on. Many of these can have custom code attached to them by using LSL. \*\*\*\*\*Highlight Touched\*\*\*\*\* If you don't have an LSL script monitoring for the event...then the default occurs. We will discuss some of the more common events in detail in the course, but this is not an exhaustive list.

An LSL function is a way for you to request a specific piece or type of information from the server or for you to request the server to do something with the information you provide. When you call it, it may return a value that you can assign to a variable or use immediately in yet another function. It may or may not require you to provide information in the form of parameters in order to provide

you with the information. Here we are providing a channel and some text for it to chat.

An easy way to tell if the function or event you are wanting to use requires parameters is with the popup description. If you move your mouse over the item in question, a popup will appear listing all the parameters, their types and a brief description.

### **Variables and Constants:**

The term variable describes a piece of memory you've allocated to hold a piece of information. It's called variable because it's prone to change. Each variable you define requires you to also define what kind of information it is allowed to hold, and a name so you can reference it. Some of the common variable types are integer, which holds a number, string, which holds a length of characters, a word, or a list, which holds more than one of any of the variable types. You can think of most variables as a cell in a Spreadsheet, and a list as a column of cells. The list has to contain all the same type of variables though. A variable can be defined as Global or Local. A global variable is defined and available to any event, procedure, or state within the script file. A Local variable is available only to the event or procedure you define it in. This determination is based only by where you type it. A global variable appears at the top of the script, outside of any events or states. A local variable appears inside the event you wish to use it. It is good programming practice to not only give a descriptive name but to provide the type and scope in the name as well. Using an integer as an example, I would precede the name with int for integer and if it was global I would add a g\_ in front of that. Thereby calling it g\_intVariable. This makes it easy to spot a global variable in your code making debugging easier. It also ensures you don't try to define a local and global variable with the same name which causes a syntax error. Variable names are also case sensitive, so must match their definition exactly.

Constants similar to variables but they cannot be changed. They are a predefined reference provided by LSL. They allow better code readability without

requiring you to define these commonly used pieces of information in every script.

In this section we will go over coding structures and operations. These definitions will allow you to control the flow of your code past the simple start and stop of an event. We will also show user defined functions and procedures which are very useful not only in allowing readability but greatly help with portability. I love Copy and Paste.

### **Operations, Typecasting and Parameters:**

Operations in SL are the same as elementary math. For now I'm talking about addition +, subtraction -, multiplication \*, and division /. Parentheses are also available to control the order of operation just as in math. That being said, some of the more advanced data types, like rotations and vectors, may react differently to the operation you've chosen than expected. Make sure you research these types before you use them. For this lecture the variables we are using will respond as expected.

Suppose we want to know how many times an object was touched. We create a global variable to maintain the count and add 1 to it every time the touch event was fired. Here I've commented out an lsl shortcut for adding one to a variable.

SL operations should only be performed if both variable types are the same. So if as an example an LSL function returns a string and I want to add the counter on the end, which for the sake of the previous addition to count it up is a number, how do I mash them together? It's called typecasting. LSL provides the ability to temporarily read a variable as a different type. Simply precede the variable in the operation with the data type you wish it to be enclosed in parenthesis.

## **Decisions and Looping:**

The If...Else code selection is by far the most common piece of code in LSL. It allows you to control the flow of code without looping. Why is that good you ask? Remember my description of LSL being single threaded and dropping requests if there are too many. The faster the event is over the better your script is. Looping takes time, so if you can use a decision instead of a loop, do it. It also allows you to handle an event multiple different ways without having to code an entirely different state. Take note of the double equal sign. This means it's a comparison not a variable assignment. If you put a single equal sign in an if statement it won't work properly. Also note the Open and close parenthesis, the same as the events. This is what tells the system which block of code is affected by this statement. One more thing about the if statement, notice there is no semicolon after it unlike functions. You can also use the other comparative tests, <, >, <=, or !=. The else is optional.

Do...While and While, though they are different commands perform essentially the same. Do the commands in the loop until your while comparison is true. Again remember LSL is single threaded. Never code a loop that is validated from a different event. It will never finish. The event the loop is in is blocking the event for the validation and your script is now stuck. Notice the semi-colon after the do...while command. This is the only loop that requires it.

For loops are, personally my favorite loop. They contain an integrated counter that you can use within the loop. They can also be set to count up or down depending how you want it to loop. It's hard to code an infinite for loop as you have to define the upper limit in the loop definition. You could increase that limit with each iteration, thus making it infinite...but you can't make the different event validation mistake.

## **User Defined Functions and Procedures:**

Truly good code is broken up into individual parts and these parts are coded separately from the events. This way in the event you can just call the functions you want to process with the small bit of code you need to process

them. It makes it easier for you to change your script as the area you want to change is not mixed in with the basic processing code. It makes it easier to find a problem as you can determine which function your script was in based on what it was doing when it failed. It also allows you to call the same piece of code in multiple events or states, so you will only have to maintain that code in one spot in the script.

The only difference between functions and procedures is that a function returns a value that you assign to a variable. Both can receive parameters and process code in the same way otherwise. Functions have a Return line that returns the value of the variable beside it back to the calling statement.

### **Changing States:**

Every event we've worked with has been in the default running state. This is the main section of the program that starts up when the script begins running. All the events within this state are available for processing while this state is loaded. What if you don't want that event to run for a period of time? You can control the code using an if statement within the event, but the event still technically runs, uses processing time, and delays the rest of the queue for that period. There is another option. You can have more than one state. If the event you don't want to run is not declared in the state currently loaded, it doesn't run. This is also useful if you need an event to do 2 different things at different times, since you can only declare an event once per state.

Be careful with this functionality though. Adding too many states makes your code extremely hard to maintain. User defined functions helps with this. But you should avoid having more than a few states in a script.

In this section we will go over some of the more commonly coded events in SL. We will also discuss certain functions that directly affect these events. We

will be continuing to create a script on screen, but will go over all the necessary functionality for the script at the end of this section.

### **Rezzing, States, and Attachments:**

When you drag an item out of your inventory it fires the `on_rez` event. Notice there is a parameter on this event. This is not used if you rez it manually but can be used if another script is rezzing it using the `llRezObject` function as a simple way of initial communication.

`attach` fires right after `on_rez`, but only if it's worn by an avatar. You can use this to make sure items that are supposed to only be attachments, only work if it's attached. The parameter is used to give you the id of the wearer. Unlike `on_rez` though, `attach` also fires when the object is detached. This is signified when the parameter is null.

The `state_entry` event fires when you enter a new state. Because every script has to start up in the default state, you can use `state_entry` to tell you when your script has been reset. There is also a `state_exit` event that fires if you use the `state` call to go to a new state.

### **Touch, Dialog and Listen:**

The Listen event is used to monitor for chat messages on the channel it's set for. You use it to allow user input to your script. The event doesn't run unless you've started it by calling it with the `llListen` command. Listens produce a lot of lag so they should be turned off with the `llListenRemove` function as soon as possible.

Channel 0 is the open channel, it's what avatars use to communicate in chat. A listen on this channel is very problematic. In scripts you can have positive or negative channels and the number can be quite large. Negative channels are script based only but positive channels can still be referenced by an avatar on the chat line if they add a `'/` and the channel before what they type on the chat line.

A dialog menu is the easiest way for users to communicate with your script. It can use positive or negative channels but should never be used on the open channel as it will appear as chat on the screen. Using the `IIDialog` function you present the avatar with a list of options you want them to choose from and monitor for the response with a `listen` on the same channel. If they click ignore on the dialog, no communication comes back so you have to remember to allow for that.

A dialog is usually called from the `touch_start` event which you've already seen. I should explain the other 2 touch events though as well. The `touch_start` fires when the mouse button goes down. While the mouse button is held down the touch event fires, unlike the other touch events this one will fire multiple times per touch. The `touch_end` fires when the mouse button is released.

### **Linking:**

Scripts within the same object cannot access each others variables directly. For some projects though multiple scripts may be required and usually need to know what the other is doing. This could be because a modular design is wanted or that the script would require too much memory to run standalone. To resolve this, you can use an internal event called `link_message`. By having the `link_message` event coded in your script you can monitor for a message sent from another script using the `IIMessageLinked` function. It's faster and more secure then using the common chat functions. You use the `linknum` parameter to control which prims the link message will go to. The most common is `LINK_SET` which sends to all parts of the object.

### **Using a Timer:**

The timer event is used to run code after a delay in seconds that you provide. The script will continually repeat the event at that interval until you cancel it. You start the event by calling the `IISetTimerEvent` with the time delay you want in seconds. To cancel it you call the same function with a 0 second delay. You can compare this to the `IISleep` function, which also causes a delay in

seconds. But the `lSleep` function locks up the event it's in without releasing the event queue so your script cannot continue.

### **Change and the Dataserver:**

The changed event can be used to detect if something about the object has changed. If you look in your insert pick list, the options you can monitor for have constants that start with 'CHANGED\_'. When the event fires the parameter will contain the type of changes that have occurred. This event can receive more than one change at once though depending on how your object is set up. For this reason, rather than using an equality comparison you should use a bitwise comparison which essentially says, 'where this change is part of the parameter' rather than this change is the parameter. A bitwise comparison is processed using a single ampersand (&).

One of the most common checks are the `CHANGED_LINK` which detects if a prim has been added or removed. This is also how you detect if an avatar is sitting on an object. When an avatar sits on an object they become linked. The one shown here is also quite common. The `CHANGED_INVENTORY` parameter will be sent whenever the contents of the objects inventory change.

The dataserver event fires when you've sent a special function request that cannot be answered directly and must be queried off the server. Some functions that request information about avatars use this event, but we are going to use it for it's most common use, reading notecards from inventory. You can also use this type of setup to have configuration files for your scripts. This way if someone needs to be able to change the variables but you don't want to give them access to the entire script, they can maintain it in the note. If you are making multiple calls, you can use the query parameter to differentiate between the responses coming in. When the data parameter reads EOF, you're done.

## **Exercise1:**

You now have enough background to complete a touch based script. The code I have shown you up to this point has given you a glance at what will be built, we will now go over each required part in detail.

The code requirement for this script is to open all the individual drawers attached to a dresser. So, what do we need to do?

1. We need a Touch Event
  - a. To get the part touched
  - b. To call the function to open the drawer
2. We need a State Entry event
  - a. To send the touch from all areas to the root where the script is
3. We need to manipulate a child prim, a link\_message event will work
  - a. We are changing the position of multiple prims. This can't be done from the root.
4. We'll create a Function call for the work
  - a. Is it a drawer
  - b. Is it open or closed
  - c. Open or close it

1. Here's what the dresser portion should look like.
  - a. In the state\_entry remove the chat function and add the lIPassTouches function. This will send all touch information to the root prim, the dresser.
  - b. In the touch\_start event, remove the chat function and add the lIDetectedLinkNumber and the lIGetLinkName functions as this will give us which prim was touched and it's name. Add the lIDetectedKey function in case we want to know who is opening it. Add the 'if it's a drawer' condition because we don't want the whole dresser to move and add a function call.

- c. This function needs to notify the script in the prim touched that it's to open or close. Code the `llMessageLinked` function to send to the detected link that it's been used. I'm sending the key of the person who touched it as well.
2. Now open a new script for the drawer
    - a. Add 2 new global vector variables to record the position of the object open and closed. A vector is a three part variable, `<x,y,z>` and is used for positioning among other things. Whether you use `x,y,`or `z` is completely dependant on your dresser's root rotation and the number you place in it can be positive or negative depending on which way it needs to go. My rotation required it to move in the positive `z` direction. This may be trial and error until you've used these functions much more.
    - b. When the script starts up there is some information we are going to need to know. Where is open and closed? I'm going to code this in an init procedure in case I can think of somewhere else to put it.
    - c. You can use `llGetPrimitiveParams` to get the size of the drawer. Use this to tell you how far out you want the drawer to come.
    - d. The size gets returned as a list, and even though in this case there is only one item, we have to pull the size vector out of the list to use it.
    - e. Notice the `.z` after the variable name. This is a special item with vector variables. It allows me to pull out only the axis I want to use instead of manipulating the 3 dimensional variable. I could have used `.x` or `.y`. Here I'm placing the size of the `z` axis less 20 per cent into a float variable, this is like an integer variable but it allows decimal places.
    - f. I'm assuming that when the script starts that the drawer is closed. I'm using `llGetLocalPos` to get the drawers position relative to the root of the dresser. This is not the same as the drawer's position in

world which you can get with `llGetPos`, but the difference between the root position and the drawer position.

- g. In this calculation, I'm setting the drawer's open position to the same as the closed position, except I'm adding the size calculation from before to the z axis.
- h. Now that were initialized. Let's code the firing pin. We code the `link_message` event and monitor for the message we are sending in the `llMessageLinked` function in the other script.
- i. If the message is received, we call the drawer function with whatever the current drawer status is and the id of the person touching it.
- j. You'll notice I'm using boolean constants, `True` and `False`, to determine whether the drawer is open or closed. There is no Boolean variable type in LSL, it's simply another integer. But using the variable this way is easy to read, yet uses less storage than a string.
- k. We will use `llSetPrimitiveParams` to move the drawer. Because we figured out what open and close was in the init, we just need to provide those variable names to the function call at the right times.
- l. Now how many of you hate it when people leave the cupboards open? In SL they can close themselves, so why not add that. Here will set a timer event to call the drawer event again to close it. But what if they did remember to close it? Remember to disable the timer if the drawer close fires.

And here's how it will work....