Beginner's Guide to Scripting with LSL

The place to start learning the Linden Scripting Language

Contents

Beginner's Guide to Scripting with LSL **1**
Contents 1
Part 1: Getting Started **2**
Introduction 2
Finding Scripts 4
Opening Scripts 4
Installing a Script Into an Object 4
Modifying Scripts—Part I 6
Modifying Scripts—Part II (Lightswitch Revisited) 9
Modifying Scripts—Part III (The Help Station) 14
Creating a New Script 15
Anatomy of a Script—Hello Avatar! 16
Analyzing Some Commented Scripts 17
Notes, Hints and Tips 21
Part 2: The LSL Language **22**
Brief Overview 22
The Process 22
Comments 22
Functions 23
States 23
Events 24
Handlers 24
Brackets 24
Global and Local 24
Program Order 24

Part 1: Getting Started

Introduction

What Is LSL?

LSL – Linden Scripting Language – is a simple yet powerful programming language designed to work within Second Life to add action, reaction, interaction and functionality to the world.

Why Is LSL?

LSL exists so both Linden staff and residents can build interactive content for the world, and continually customize and expand the world to make it more interesting, less static, and more "alive."

Who Can Use LSL?

LSL is available to all Second Life residents and Linden Lab staff. Yes, that means that with LSL, you have at your disposal the same tools that the Linden staff uses to create content, with all the power, as well as rights and responsibilities, that comes with it.

With LSL, it is possible to create scripts that cause problems, harass people, slow down the simulators and generally cause mischief. Always use LSL responsibly, and within the bounds of our posted community standards.

What Can You Do With It?

LSL lets you add "behaviors" to any object in Second Life.

You can make doors that open with a password, fires that burn, fireworks that sparkle, weapons that shoot projectiles, and just about anything else you can imagine.

If you learn the basics of this language, you'll expand your creative options within Second Life. If you become a skilled LSL programmer, you'll be able to write scripts for others as a business within Second Life.

What If You've Never Programmed Before?

Not a problem.

Of course, someone with programming and math skills will learn LSL more quickly, and will be able to create far more complex and flexible scripts. But non-programmers will be able to learn the basics and create their own simple-yet-effective scripts in a very short amount of time.

Even those who really don't want to invest any time at all into learning LSL will be able to modify and customize existing scripts to meet their needs. For example, it's a simple matter to change the password on a door script.

How Do You Learn It?

You're in the right place to start.

Read through Part 1 of this document, paying close attention to the sample scripts.

Next, scan (or read if you're motivated) Part 2, to get a better idea of LSL's structure and potential.

Then, try some scripts and see what happens.

Beyond this document, sources of scripting help/instruction are:

- The Complete Language Reference—available through the Help menu (Scripting Help), this document is a detailed reference to every function, call and keyword. (You can also access this HTML-format document outside of Second Life for printing. It's in C:\program files\second life\lsl2.)
- In-world scripting workshops—experienced scriptors regularly hold in-world workshops to teach scripting. Check the calendar for times and locations.
- The Forums—there is a Scripting area in the Forums where residents and Linden staff ask and answer questions, share hints and tips and give tutorials and challenges.
- The Liaisons—you can ask the Linden Liaisons for help with scripting.
- Residents—many of the residents who are skilled at scripting are very willing to help you with your scripting questions.

How This Document Will Help You

This document has been written to introduce the complete beginner to LSL. It will teach you the basics you need to create your own simple scripts without overloading you with technical details.

For those who want the technical details, we've included some more technical sidebars.

Finding Scripts

You can find scripts all over Second Life.

Objects all around you have scripts. To find out if an object has a script, press and hold the Alt key. Scripted objects will glow red.

You don't have access to viewing scripts in objects that are owned by other people (unless they grant you the rights). You do have access to scripts (usually) in objects that you own and public objects.

In addition, there are some scripts in the Scripts folder in your Inventory.

Another place to find scripts is in the Building Bazaar (Stillman 141,47). Here you'll find all sorts of free stuff, some with scripts. You may have to take a copy of something then place it in the world before you can access the script.

A good place to get scripts is through the Forums. Lindens and residents regularly post new scripts there for all to learn from and enjoy.

Finally, you may also find scripts at various stores or vending machines in-world, or get them (or buy them) from other people.

Opening Scripts

To open a script that's in your Inventory, just double-click on it.

To open a script that's in an object:

1. **Right-click on the object, and select Edit from the pie menu.**
2. **Click on the Content tab.**
3. **If no script shows, then double-click on the Content folder to open it.**
4. **Double-click on the script.**

Installing a Script Into an Object

To install a script in an object, just drag it from your Inventory and drop it onto the object (if you have rights to modify/edit the object). You can also drop the script into the object's Contents folder on the Content tab.

To see that the script is there, edit the object and look at the Content tab.

Let's try it:

1. **Create a box.**
2. **Open your Inventory, then the Scripts folder, then drag Rotation Script out of your Inventory and drop it onto the box.**
3. **Close the Tool palette or click on the ground near the box to release it from editing.**

The box will start rotating.

Let's look inside the box to see where the script is:

4. **Click on the box, and select Edit from the pie menu.**
5. **In the Tool palette, click on the Content tab. You should see the script there.**
6. **Keep the box handy for the next section.**

Modifying Scripts—Part I

Why modify a script?

It helps you learn how scripts work, and it's usually easier to modify one than start from scratch.

Preparing for a Very Simple Modification

Let's modify the script in the rotating box.

1. **Right-click on the box you made in the previous section, select Edit from the pie menu and open the Content tab in the Tool palette.**
2. **Double-click on the script.**

This is a very short script that does one thing, so it's a good one to play with.

There is only one function: `llTargetOmega`.

```
llTargetOmega(<0,0,1>, PI,1.0);
```

(There are other words in the script, and we'll get to them later.)

Notice that llTargetOmega has a number of numbers after it (yes, here, pi is a number). These are called "parameters" or "arguments" in programmer talk. They change how the function works and make it more flexible.

If you look up `llTargetOmega` in the LSL Reference, it'll say:

llTargetOmega llTargetOmega(vector axis, float spinrate, float gain); Attempt to spin at spinrate with strength gain. A gain of 0.0 cancels the spin.

If you're already a programmer, terms like gain and vector will make sense to you, but you don't really have to understand all the math to play with the script and make some changes.

Something to understand about the way the Reference displays information is that each parameter generally has two parts: the form/format of parameter and what it is/does.

In other words, the one that says *float spinrate* doesn't have anything to do with floating in the air. When the reference says *float spinrate,* it means that *spinrate* is a floating point, or real number.

When it says *vector,* it means three floating point numbers, which are often used to denote a direction (because directions are defined by three axes) or a color (because colors are defined as values of red, green and blue, or RGB). Here, the three vector numbers define the *axis* that the object will spin around.

So, with just a little guesswork, we can probably conclude that:

1. Changing the first 3 numbers (the ones inside < >) will change the direction of the rotation, and that the three numbers represent the x,y and z axes.

2. Changing the second number (currently PI) will change the speed of rotation, and
3. Changing the gain will ... that's not an easy one to guess at, but since the reference says that "a gain of 0.0 cancels the spin," we probably don't want to change that value to 0.

And if we're wrong in our conclusions, we'll soon find out once we start playing with the numbers.

Technical Sidebar

The gain is the force or torque that is applied to the object in order to spin it. It comes more into play when objects are made physical.

Playing It Safe

Because we're going to be messing around with a script—and as anyone who has ever been involved with software knows, things can go wrong—we'll take a simple precaution that will let us back up and start over.

We started with a script in your Inventory. We put *a copy* of that script into the box. The original is still in Inventory.

We will be playing only with *the copy* and not the original. That way, if we delete too much, we can always start over with a fresh, working copy of the original script.

So, for now, only edit the script in the object, not in Inventory.

Once you get a nice modified script that you might like to use again, you can copy it to your Inventory (and give it a new name) so you can use it any time you want without starting over and editing the original one.

Trying Some Changes

The first thing we'll do is play with the axis numbers, to change the direction of rotation.

When the script is running with axis numbers, <0,0,1>, we can see that the box is rotating around the z axis.

Let's try a few:

1. **In the script editing window, change the numbers to <1,0,0>.**
2. **Click Save.**
3. **Click in the world, off of the box to release it from editing.**

The box will start to rotate around the x axis.

4. **Now reset the axis numbers to <0,1,0> and see what happens.**
5. **Now try <.5,2,1>.**

Something to keep in mind is that floating point numbers can be negative as well as positive.

6. **Now try <0,0,-1>.**

7. **To be clear about this, copy the box, so you have two. Set one at <0,0,-1> and the other at <0,0,1>.**

Now let's see what we can do with the rotation speed.

8. **Change the spinrate (currently PI) to 1 and see what happens.**
9. **Change it to 5.**

Practical note: You can make objects spin very fast—in the simulator. But, depending on your framerate, you'll eventually hit a point of diminishing returns, and it won't actually look like it's rotating any faster. In fact, there's a point where the framerate limitations cause the optical illusion that the box is spinning the opposite direction. For the fun of it, see if you can find that rate on your computer.

Breather and Reality Check

OK, so changing the rotation speed and direction of a box may not seem that impressive, but look at the big picture: you changed and reapplied a script quite a few times. What you did with `llTargetOmega`, you can also do with dozens of other functions, so you can customize and fine tune many, many scripts now.

Let's try another example.

Modifying Scripts—Part II (Lightswitch Revisited)

Preparing for the Next Modification

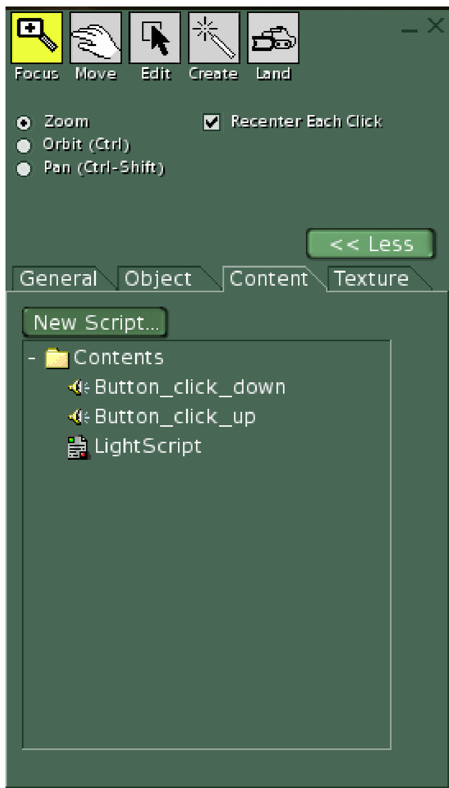
If you went through all the online Help, you've already played with this script. Here, we'll do it again, and take it a bit further, with a little more explanation along the way.

We'll be working with the "Floor Lamp from Help" from your Inventory's Objects folder. In order to keep the original copy of the script—and the lamp—intact, we'll pull out a copy by Shift-dragging it.

1. **Hold down the Shift key and pull a copy of the "Floor Lamp from Help" from your Inventory's Objects folder to the ground.**
2. **Right-click on the lamp and select Edit from the pie menu.**
3. **Open the Content tab. There is no script.**

Since the lamp is made of multiple parts and the script is only in the bulb, we'll have to select just the bulb in order to see its script.

4. **Click on "Select Individual" in the Tool palette, then click on the lamp's bulb.**



Now the script, "Lightscript," is visible.

Before we move on, a quick note: notice that there are also a couple of sounds in the lamp's contents. If you want a script to play a sound, that sound has to be in the object's contents along with the script.

5. **Double-click on Lightscript.**

This is a much longer and more complicated script than the Rotation script, but with a little patience, it'll make sense.

This script does a few things: it toggles the lamp on and off when it's clicked—but only when the lamp's owner clicks it; and it listens for the words "on" and "off" and if those words are spoken by the owner, it turns the lamp on and off. It also triggers different sounds when the lamp turns on and off.

We'll look at this script in detail later, but for now, we'll just look for a couple things to change.

The easiest things to change here are the words that turn the light on and off, and the sounds that it plays when it turns on and off.

Conveniently enough, all the things we're going to change are surrounded by quotation marks, which will make them easier to find.

Changing the Switch Words

Again, we'll go through this script in detail later, but for now, we'll just jump around to the spots we want to change.

1. Scroll down the script until you see the lines:

```
llListen(0, "",llGetOwner(), "on");
llListen(0, "",llGetOwner(), "off");
```

What these two lines do is have the script listen to channel 0 (the chat channel) for the words "on" and "off." The llGetOwner part of the line is used by the script later to make sure that only the owner can turn the lamp on and off. We'll worry about that later. For now all we care about are the "on" and "off."

2. Change `"on"` to `"foo"` and change `"off"` to `"bar."` (You can use any words or phrases you want, but keep them short. You don't want to have to type a novel to turn your light on and off.) Make sure that the quotation marks are still there.

3. Now scroll down to the line:

```
if  (text == "on")
```

4. Change `"on"` to `"foo"` (or whatever word you used in the previous step).

5. Save the script. Hit Esc to release the lamp from editing.

6. Chat the words foo and bar (or whatever words you used).

The light should turn on and off.

If it doesn't, then make sure you didn't lose any quotation marks or delete anything else by accident.

Changing the Sounds

Now we'll change those boring clicks that sound when the lamp turns on and off to something more interesting.

1. Right-click on the lamp, select Edit from the Pie menu.

2. Click on Select Individual in the Tool palette and click on the bulb.

3. Click on the Content tab.

You should again see the contents of the bulb: two sounds and the script.

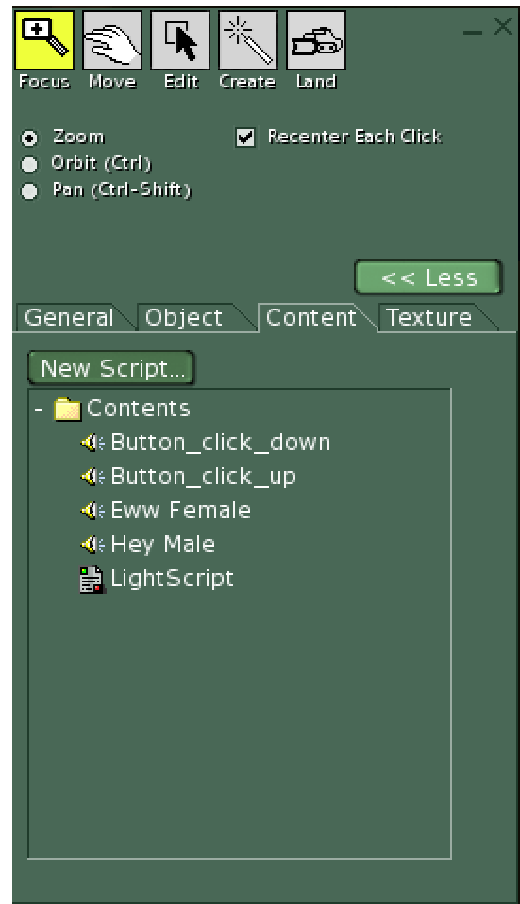
We'll add a couple of sounds.

4. In your Inventory's Sounds folder, there's a Gesture Sounds folder. Open it.

5. Drag the sounds "Hey Male" and "Eww Female" out of Inventory and into the Contents folder. (You have to drag it to the actual folder, not to the list of files.)

Note: When you drag something from Inventory into the world, as we did with the lamp, you have to Shift-drag it to make a copy (or it moves instead of copying). But when you drag something out of Inventory and *into something else,* like an object (like a lamp), it always copies, so you don't have to Shift-drag.

Your Contents should now look like the picture to the right.



6. Open the script (it may still be open) and, find, near the top, the following line:

```
llTriggerSound("button_click_down", 1.0)
```

And a few lines below it:

```
llTriggerSound("button_click_up", 1.0)
```

These lines play the sounds at turn on and turn off. The name of the sound file to be played is inside the quotation marks, and the number at the end is the volume. Let's change the sounds that are played.

7. Change `"button_click_down"` **to** `"Hey Male"` **and change** `"button_click_up"` **to** `"Eww Female"`**.**

8. Save your script, release the lamp from editing and turn it on and off.

You should hear the new sounds. If not, make sure you didn't delete anything extra by accident, and that the sound file names are inside quotation marks.

Adding a New Twist to the Script

Now we'll take another step, and add a couple of new lines to the script.

We'll use the llSay keyword to have the lamp talk to us when it's turned on and off.

1. Find the places in the script where you changed the sounds.

They look like this:

```
llTriggerSound("Hey Male", 1.0);

llSetColor(COLOR_ON, ALL_SIDES);
```

And a few lines below:

```
llTriggerSound("Eww Female", 1.0);

llSetColor(COLOR_ON, ALL_SIDES);
```

2. Put your cursor at the end of the top llSetColor line and hit Enter to create a new line (it automatically keeps the same indentation).

3. On this line, type: `llSay(0, "I'm turned on!");`

4. Make a new line below the second llSetColor line and on it, type: `llSay(0, "You turned me off!");`

Note: the 0 is the number zero, not a capital letter "O," and don't forget the semi-colon at the end of each line.

The two parts of the script should now look like this:

```
llTriggerSound("Hey Male", 1.0);

llSetColor(COLOR_ON, ALL_SIDES);

llSay(0, "I'm turned on!");
```

And a few lines below:

```
llTriggerSound("Ew Female", 1.0);

llSetColor(COLOR_ON, ALL_SIDES);

llSay(0, "You turned me off!");
```

Be sure all the quotation marks are there and each line ends with a semi-colon.

5. Save the script, and try it out.

Now, whenever the lamp is turned on or off, it sends out a message to chat.

Saving Your Work

You can save your work in a couple of ways:

1. **Acquire/Take the whole lamp into your Inventory and then rename it, or**
2. **Rename the script in the lamp's contents (right-click on it then select Rename), and drag a copy of it into the Script folder in your Inventory.**

There are times and reasons to use either or both. The main thing is to make sure you don't lose any of your work, and that you give your modified scripts and objects names that really tell you what's new and different about them.

Another Breather and Reality Check

These last simple rounds of changes showed you that you can very easily make some powerful changes to existing scripts.

You can change text—text for controls or passwords, and text for messages. You also changed the sounds that a script plays. You even added a whole new command to the script (in two places)!

What Can You Change?

Things you can generally change are:

- The parameters that follow the functions, and
- Words in quotation marks.

Words in quotation marks are strings or text words or phrases or file names (like sound files). These may include messages that the script may send as well as passwords to activate parts of the script.

What Shouldn't You Change?

Things you shouldn't change (until you really know what you're doing) are:

- Keywords,
- Any punctuation, and
- Words that aren't in quotation marks.

Modifying Scripts—Part III (The Help Station)

After the lightswitch, this will be simple.

You'll see Help stations (a.k.a., info stations, help kiosks or info kiosks) all over Second Life. You may want to use them yourself to pass information to others about something you built, arranged. Basically, what a help station does is offer a note card to anyone who clicks on it. What you put on the note card is up to you.

Info stations are free for the copying, but to make them fully work for you, you have to make a couple of modifications.

Let's look at the contents. You'll see three things:

1. A Script (Help Station),
2. A Note Card (Read Me or other), and
3. A Sound.

To make the help station do your bidding, you'll need to:

1. Rewrite—and rename—the note card to provide the information you want to provide.
2. Change the script to call the new note card.

You can do more if you want, such as change the sound, or even add more functions, but all you really need to do is rewrite and rename the note card and then change the script to call the newly named note card.

Let's say you just rewrote the note card, and renamed it to "Read Yourself".

Let's change the script.

1. Open the script (double-click on it).

2. Find the line:

```
String note_name = "Read Me";
```

Note: yours may not say "Read Me" in the quotes, but the rest of the line will be the same.

3. Inside the quotes, type in "Read Yourself" or whatever you named your modified note.

4. Save the script, close the Tool palette and test the help station.

Creating a New Script

You can create a script in two basic ways:

1. In Inventory—select New Script from either the Create menu or the context-sensitive menu for any folder in the Inventory.
2. In an object—open an object's Content tab and click on the New Script ... button.

Your choice of which way/place to create a script will depend on the project as well as your preferred way of working.

Anatomy of a Script—Hello Avatar!

When you create a new script, by default, you get the Hello Avatar script. Let's look at this simple script and see what it does.

Hello Avatar is a very simple, one-state script.

```
default
{
	state_entry()
	{
		llSay(0, "Hello, Avatar!");
	}

	touch_start(integer total_number)
	{
		llSay(0, "Touched.");
	}
}
```

The script starts with:

```
default
```

"Default" sets the default state, the starting point for changes. All scripts start with this keyword, even simple ones with only one state. We'll explain states a bit more when we get to the next script, which has more than one state.

The next line,

```
	state_entry()
```

is an event handler. It basically says when the script enters the current state (in this case, default), do something. Do what? Do what's inside the two curly brackets listed below:

```
	state_entry()
	{
		llSay(0, "Hello, Avatar!");
	}
```

In this case, what happens is the script chats: the words "Hello, Avatar!"

If we look up llSay in the script reference, we'll see:

```
llSay(integer channel, string text)
```

The integer channel, in this case "0," defines the channel that the script uses to communicate. Channel 0 is a public channel: anything sent to that channel shows up in chat. Channels 2 to 2,147,483,648 are private channels. You can use these for sending information from one script to another without it showing up on chat.

The "string text" part is the actual text that the llSay function sends to channel 0. You can change the string text to anything you want, but it must be inside quote marks to work.

Now we've covered everything inside the curly brackets under llSay, so we're done with that particular handler.

The next handler and what the handler does (inside the curly brackets) is:

```
touch_start(integer total_number)
{
      llSay(0, "Touched.");
}
```

`touch_start` senses if the object containing the script is touched—or clicked on. When it is clicked on, it does whatever's in the following curly brackets. In this case, it chats the word "Touched."

Analyzing Some Commented Scripts

Here are some nicely commented sample scripts that will teach you more about scripting. Read them through, play with them, have fun.

```
// hello world

// all scripts need a default state
default
{
      // the state_entry event handler is called when the
      //  state (in this case, the default state) is entered
      state_entry()
      {
            // llSay is a library function call that says the string
            //  on channel 0 (the channel avatars chat and listen on
            llSay(0, "Hello, world!");
      }
}
```

```
// hello world on rez from inventory

default
{
      // the on_rez event handler is called whenever an object is
      // rez-ed out of inventory or by a script call
      //  start_param is 0 is rez-ed from inventory, but can be set
      //  when rez-ed by a script call
      on_rez(integer start_param)
      {
            llSay(0, "Hello, world!");
      }
}
```

```
// hello world when clicked on by an avatar

default
{
      state_entry()
      {
            llSay(0, "Hello, world!");
      }

      // the touch_start handler is called when an avatar clicks on the
```

```
        //  object total_number is the total_number of avatars that
        //  started clicking on  the object since the last touch_start
        //  handler was called
        touch_start(integer total_number)
        {
              llSay(0, "I've been touched");
        }
}
```

```
// change the color of an object when clicked on

// global variables
//  global variables are used to store data that is available inside
//  any function or event handler

// initialize this variable to false
integer COLOR_ON = FALSE;

// global functions
//  global functions can be called by other global functions or by
//  event handlers

// change the color of the object based on the COLOR_ON global variable
set_color(integer color)
{
        // if statements use the same syntax as C
        if (color == TRUE)
        {
              // colors are set via vectors where the first value
corresponds to
              //  red, the second green, the third blue
              //  this call sets all the sides of the object to white
              llSetColor(<1,1,1>, ALL_SIDES);
        }
        else
        {
              // otherwise, set the color to black
              llSetColor(<0,0,0>, ALL_SIDES);
        }
}

default
{
        touch_start(integer total_number)
        {
              // lsl supports C style Boolean and bit operations,
              // in this case
              //  logical NOT
              COLOR_ON = !COLOR_ON;
              //  tell the world what the setting is, using a type cast
              //  convert the integer to a string and using the +
              //  operator to concatentate
              //  the string
              llSay(0, "COLOR_ON set to " + (string)COLOR_ON);
              // actually change the color
              set_color(COLOR_ON);
```

```
	}
}
```

```
// change the color of an object when the owner tells it "on" or "off"

// use the same global variable and global function
integer COLOR_ON = FALSE;

set_color(integer color)
{
	if (color == TRUE)
	{
		llSetColor(<1,1,1>, ALL_SIDES);
	}
	else
	{
		llSetColor(<0,0,0>, ALL_SIDES);
	}
}

default
{
	state_entry()
	{
		//  llListen library function call to cause the script to
		//  listen for chat from its owner
		//  the first empty string is an optional name argument,
		//  allowing the listen to be set to listen to objects and
		//  avatars a certain name
		//  the second empty string specifies the text to listen
		//  for but since we want to get both on
		//  and off we leave that blank as well
		//  if the llGetOwner() was changed to an empty string then
		//  all chat on channel 0 would be heard
		llListen(0, "", llGetOwner(), "");
	}

	// the listen event is called when the object hears chat that
	//  meets the condition specified
	//  by the llListen library function
	//  scripts can have more than one listen active at a time
	listen(integer channel, string name, key id, string message)
	{
		// change COLOR_ON based in the message text
		if (message == "on")
		{
			COLOR_ON = TRUE;
		}
		else if (message == "off")
		{
			COLOR_ON = FALSE;
		}
		llSay(0, "COLOR_ON set to " + (string)COLOR_ON);
		set_color(COLOR_ON);
```

```
        }
}
```

```
// a door that moves up and down when clicked on

// this script causes an object to move up and down when clicked on
// it is set into position by the owner chatting "ready" at it

// we need a global to store whether the door is open or closed
integer CLOSED = TRUE;


default
{
        state_entry()
        {
                // we only need to listen for the command "ready"
                llListen(0, "", llGetOwner(), "ready");
        }
        listen(integer channel, string name, key id, string message)
        {
                // when the door is told ready, assume that it is in the
                //  proper closed position
                CLOSED = TRUE;
        }
        touch_start(integer total_number)
        {
                // we need to know where we are so that we can decide where
                //  to move to
                //  llGetPos() is a library function that returns our
                //  to move to current position
                vector our_position = llGetPos();
                // we also need to know how big we are so we know how much
                //  to move
                vector our_scale = llGetScale();
                // we'll move along the up (z) axis, so let's store that
                //  vectors access x,y,and z values via .x, .y, and .z
                float z_scale = our_scale.z;
                // if we're closed, move, otherwise move down
                //  also, set CLOSED correctly
                if (CLOSED)
                {
                        our_position.z += z_scale;
                        CLOSED = FALSE;
                }
                else
                {
                        our_position.z -= z_scale;
                        CLOSED = TRUE;
                }
                // now set the object to the new position
                llSetPos(our_position);
                // set the closed variable to the new setting
        }
}
```

Notes, Hints and Tips

- Save your work often!
- Objects can contain more than one script. If you want an object to do a number of things, it may be easier to write and debug a few small, simple scripts than one large, complicated one.
- Save your work often!
- And, of course:
- Save your work often!

Part 2: The LSL Language

Brief Overview

Linden Scripting Language (LSL) is similar in syntax to C and Java.

It uses an "event execution model," meaning that events (more on these later) cause specific pieces of code to execute.

Event handlers are commands that set, reset or react to events.

Library Function Calls (any words beginning with "ll") call and enact pre-compiled routines or modules (functions) that do any number of useful things within your program.

Following, you'll find some brief explanations of the basic elements and concepts of LSL for those without programming experience.

For more details, see the Complete Language Reference (available through the Help menu, or in C:\program files\second life\lsl2).

The Process

The basic method of writing a script is:

1. Write.
2. Save.
3. Test.
4. Repeat as necessary.

When you save a script, it's automatically compiled, so it's ready to go. If there's an error that prevents compiling, you'll know as soon as you save.

Comments

Comments are text messages within the script code. They don't "do" anything programwise, but, if they're well written, will explain how the script works, so anyone reading the code will be able to understand and modify it. They're especially important when you need to update or modify or expand a script you wrote some weeks or months before. If you comment it well, you won't spend extra hours trying to remember what you were thinking way back when.

Comments always begin with two forward slashes: //

Comments are only one line long, so if you need to make a long comment, be sure to put the double slashes at the beginning of every comment line.

If you leave off the double slashes, the compiler will get confused and the program won't run.

Examples:

```
// This is a comment.
```

```
// This is a really, really, really, really, really, really, really,
// really, really long comment, but works, because it has the double
// slashes on each and every line.

// This starts out as a comment, but, since the slashes are missing on
the second line, it actually stops the program from working.
```

Functions

There are around 200 functions built into LSL, and ready-to use. These functions exist to save you time and energy—they're already coded and tested, so you can just call them instead of writing a lot of extra code.

A complete list of these functions, along with their definitions and enough instructions to use them is in the Linden Scripting Language Reference (select Scripting Help ... from the Help menu).

Library function calls begin with "ll".

You can define your own functions as long as the name doesn't conflict with a reserved word, built-in constant or built-in function.

Functions that have been described in earlier parts of this document include:

```
llTargetOmega(vector axis, float spinrate, float gain)
```

```
llTriggerSound(key sound, float volume)
```

and

```
llSay(integer channel, string text)
```

The information in the parentheses contains parameters for the function, giving it much more power and flexibility.

For instance, llTriggerSound can trigger any number of different sounds, so where it says key sound, you'd put "soundfilename"—the name of the sound file that you want this function to trigger.

File names and text strings are always in quotation marks.

In the parameters, you'll see words like vector, float, string and integer. These define what format that particular parameter is. Float means it's a floating point or real number. Integer means it's an integer (positive whole numbers). String means it's a text word or phrase. Vector means it's a set of three floating point numbers.

States

LSL is organized into "states."

A state is a particular condition. For instance, a door may have two states, open and closed.

Every script has to have a default state, so it knows where to begin.

Simple scripts have only one state, the default state.

Other scripts have more than one state—as many as you need.

Events

Events are what they sound like: things that happen. Among many others, events can include:

- Entering a default or other state,
- An avatar approaching
- A mouse click
- A keyboard key being pressed
- Collisions between things

Handlers

Handlers, or Event Handlers detect specific events, then execute the code that follows it (that's contained in the curly brackets).

Handlers don't just activate once—they stay active, and will react each time the event they're looking for happens.

Brackets

The curly brackets {} are used to group chunks of code together. Every opening bracket needs a corresponding closing bracket and vice versa.

Global and Local

Variables and functions can be global—are active and accessible throughout the whole script, or local—only active and accessible in one part of the script.

For very short scripts, global and local don't really come into play. But in a longer, more complex script with many interacting functions, making things global can simplify things.

Program Order

There are some rules about what goes where in the script.

Comments can go anywhere—and should go just about everywhere.

Global variables and global functions go at the top of the script.

Below that, are the different states (and the events you want to handle while in each state), always beginning with `Default`.