

# MOVE ME

## DETECTION AND MOVEMENT IN SECOND LIFE

To whom it may concern.

### 1 INTRODUCTION

#### 1.1 ABOUT THE MANUAL

This is a summary of my research into the Second-Life scripting language (LSL) during the first half of the spring semester 2007 at the School of Architecture, KTH, Stockholm.

It is by no means a complete manual to the world of LSL but deals foremost with the issues of detecting and moving objects.

My ambition was originally to investigate how to create a memory that could be used by different objects at different locations in Second Life, allowing them to store reactions and basically learn behavior.

The process of creating such a memory, setting up an of-world server (php-pages and sql-database), proved to be to cumbersome and my efforts was focused towards movement and detection: The behavior of objects, how that would be interpreted by a person in terms of hostility and friendliness and how the mechanics behind such behavior (code: functions, types, events, flow, math etc.) works.

The layout of this manual is (similar to the scripting language it self) object oriented. That means that the content is organized into the LSL-objects (functions, events, types) that it describes.

It is also a very personal documentation and I am not an experienced coder. The documents may contain errors, some due to lack of attention, others due to lack of knowledge.

The aim of this manual is foremost to describe the mathematical harshness that lies behind the creation of something that, in lack of better words, could perhaps be described as pretty.

peace.

Rutger Sjogrim 070315  
no rights reserved!

## TABLE OF CONTENTS

### 1 INTRODUCTION

- 1.1 About the manual
- 1.2 A brief introduction to Second Life
- 1.3 LSL, Linden Scripting Language

### 2 BASICS: FLOW CONTROL AND TYPES

- 2.1 if-else
- 2.2 while
- 2.3 for
  
- 2.4 integer
- 2.5 float
- 2.6 string
- 2.7 vector
- 2.8 rotation

### 3 SENSOR FUNCTIONS AND EVENTS

- 3.1 llSensor
- 3.2 llSensorRepeat
- 3.3 sensor
- 3.4 no\_sensor

### 4 DETECTION

- 4.1 llDetectedPos
- 4.2 llDetectedRot
- 4.3 llDetectedVel

### 5 DYNAMICS

- 5.1 llMoveToTarget
- 5.2 llRotLookAt
  
- 5.3 llGetPos
- 5.4 llGetRot
- 5.5 llGetVel
- 5.6 llGetOmega

### 6 MANIPULATING PRIMITIVES

- 6.1 llGetPrimitiveParams
- 6.2 llSetPrimitiveParams

## TABLE OF CONTENTS (continued)

### 7 EXAMPLES

- 7.1 Simple script that levitates an object
- 7.2 Node controller script (my main script)

### 8 REFERENCES



### 1.2 A BRIEF INTRODUCTION TO SECOND LIFE

Second Life is an online world inspired by the “Meta-verse” defined by Neal Stephenson in the 1995 science fiction novel “Snow Crash”.

It is owned and maintained by San Francisco based developer Linden Lab and has been online since 2002 (publicly available since 2003).

The world of Second Life consists of the “**main land**” and an archipelago of **private islands** surrounding it.

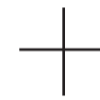
The main land is usually open for all people, although a lot of land owners put up barriers around their properties, and is part of a coherent (at least that seams to be the ambition) landscape.

On the other hand, as the owner of a private island you are free to design it however you like, although restricted to the fundamental two-dimensionality of a geographical world. You can also set up visitor restrictions, deciding who may enter. Some islands are closed to the public and some are not but they all require you to teleport in as users are not allowed to traverse the open sea, and thereby they are somewhat harder to discover. “The Office”, our online classroom, where I did most of my research, is such an island.

There are three ways a user can travel through Second Life; **on foot**, **flying** or by **teleporting** to different locations. Vehicles can be constructed or bought but are not very useful for travel. Instead they function mostly as toys to the residents of Second Life.

The ability to teleport in and out of locations together with your ability to fly greatly defines the urban organization of Second Life: It’s messy. Roads exists merely as decoration, a reference to the real world, and no one uses them.

Perhaps the most characteristic feature of Second Life is it’s **economy**. Everything in Second Life can be bought or sold. Especially land. Some people actually make their living (in real life) by buying and selling objects, scripts animations and land.



The way everything in Second Life is privately owned is an important part of the in-world culture. In short, people tend to be very protective of their possessions (objects, land, avatar). The creation and usage of these objects becomes the purpose of most Second Life activity (other popular activities include; sex, role-playing, sitting in chairs, conversations, sex and... Oh, did I mention sex?). The first impression of Second Life could roughly be described as **MTV gone wild**, with people looking like be Beyonce and living like Beyonce. Except they have bigger breasts and a much bigger TV.

For more information on Second Life and it’s history see:

[http://en.wikipedia.org/wiki/Second\\_life](http://en.wikipedia.org/wiki/Second_life)  
<http://www.secondlife.com/>  
<http://secondlife.reuters.com/stories/category/second-life/warren-ellis/>

### 1.3 LSL, LINDEN SCRIPTING LANGUAGE

The Linden Scripting Language, LSL in short, is an object oriented scripting language.

**Object orientation** means that a program (or in this case; a script) is comprised of smaller parts, objects, that can handle information and be called upon individually.

**Scripting** mainly differs from programming in that a scrip can be run directly (and is interpreted by a program) whereas the code describing a program needs to be compiled into binaries (1's and 0's) and can then be run on its own. In a Windows environment these compiled executable usually end in ".exe".

LSL has a syntax (syntax is to a programming language what grammar is to a spoken language) similar to that of JAVA.

This manual will deal only briefly with the general aspects of LSL.

Important to any kind of scripting (or programming) are flow-control and types. These will be explained later on in chapter 2. The following example explains the basic structure of the syntax:

```

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

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

This is the default **state**. This is where the script will start when it's run.

**Event**

**Function**

The "{}" defines the content of a function or event. If a function needs to be defined that is done outside of the main flow of the script.

Within the "()" following a function or event input data is defined. Some functions and events require no input and the "()" are left empty.

## 2 BASICS: FLOW-CONTROL AND TYPES

### 2.1 IF - ELSE

The **flow-controls** are the mechanisms that determines the progression through the script. It tells the script to stop, repeat, move on or move to a specific position in the code.

The **if - else** flow-control is possibly the most fundamental and important. It's function is to activate a piece of code if certain conditions are met, if not an other pice of code can be activated and an other and an other and so on.

If "statement" is true, then do this. If not (else) do that.

```

if (x > 10){
    llSay(0, "x is bigger than 10");
}
else{
    llSay(0, "x is not bigger than 10");
}
    
```

In this case "if" checks whether x is **bigger than 10**. If that is **true** then the object the script is attached to says "x is bigger than 10". If the condition is **not true** (x is smaller than or equal to 10) the object says "x is not bigger than 10".

Common operators are: bigger/smaller (</>), bigger/smaller or equal to (<=/>=), or equals (=). These operators are called boolean operators and are used to check if a condition is **true** or **false**.

### 2.2 WHILE

The **while** flow-control loops the code within it as long as the conditions are met.

```

integer some_number = 0;

while(some_number < 10){
    llSay(0, "some_number is still smaller than 10");
    some_number++;
}
    
```

This creates a loop that will repeat itself until "some\_number" is bigger than or equal to 10. For each loop the object the script is attached to will say "some\_number is still smaller than 10" and the value of "some\_number" will be increased by 1. When the value of some\_number reaches 10, the loop will release and the script will continue.

### 2.3 FOR

The **for** flow-control creates a loop i the same way "while" does. It is usually used to execute a pice of code a set number of times.

```

integer number_of_times = 10;

for(x = 0, x < number_of_times, x++){
    llSay(0, "Still looping");
}
    
```

This is a very simple for-loop that basically does the same thing as the if-loop in the example above. The important part is the code that is between the "()".

```

for(initializasion, test, update)
    
```

The first part defines x as equal to 0. Then the condition "if x is smaller than number\_of\_times" is tested (if statement is true the loop is activated). Then the value of x is increased by 1.

### 2.4 INTEGER

**Types** stores values. **Integer, float, string, vector** and **rotation** are the types I have used in my scripts. There are, of course, more types and they are all explained on the LSL web pages (see references). Different kinds of **values** can be stored in different kinds of types.

In order to be used, a type first has to be **declared**:

```
integer number;
```

The above code declares the variable "number" as an integer. It can now be used to store data. For example by stating:

```
number = 10;
```

The integer "number" now holds a value of 10. This can in turn be accomplished in one line, defining "number" as an integer and giving it a value of 10 all at once:

```
integer number = 10;
```

The **integer** type can hold a 32-bit, numerical value, between -2147483648 and 2147483647. It is a whole number. That means it has no decimals and is usually good for counting.

### 2.5 FLOAT

The **float** type also stores numbers. The definition of a float is a bit harder to understand than that of an integer (I don't really understand it myself so I'll just explain how it can be used).

Most importantly, it can be defined with decimals. Second it can also be defined as an exponent. Ex. 3\*E^10.

I use it for values that need decimals.

### 2.6 STRING

The **string** type can hold a string of characters. For example:

```

string i_would_like_to_say = "A string can hold any kinds of characters, like: 20,+ "###% and such";

llSay(0, i_would_like_to_say);
    
```

This would make an object say: "A string can hold any kinds of character, like: 20, + "###% and such".

### 2.7 VECTOR

The **vector** type in LSL holds three float values. It's usually used to describe a position in three dimensional space or a direction, the velocity of an object and similar values.

The vector type and how it can be used is the core of my script and my research so this part will be rather lengthy.

A vector is defined as follows:

```
vector some_vector = <x,y,z>;
```

Where x, y and z are any acceptable **float values** (for example <10.2, 3, 76.455>). They correspond to the x, y, z - axis coordinates in global or local three dimensional space, but can of course be used for other purposes.

Vectors can be added and subtracted.

```
<1, 2, 3> + <2, 4, 1> = <3, 6, 4> or
<1, 2, 3> - <2, 4, 1> = <-1, -2, 2>
```

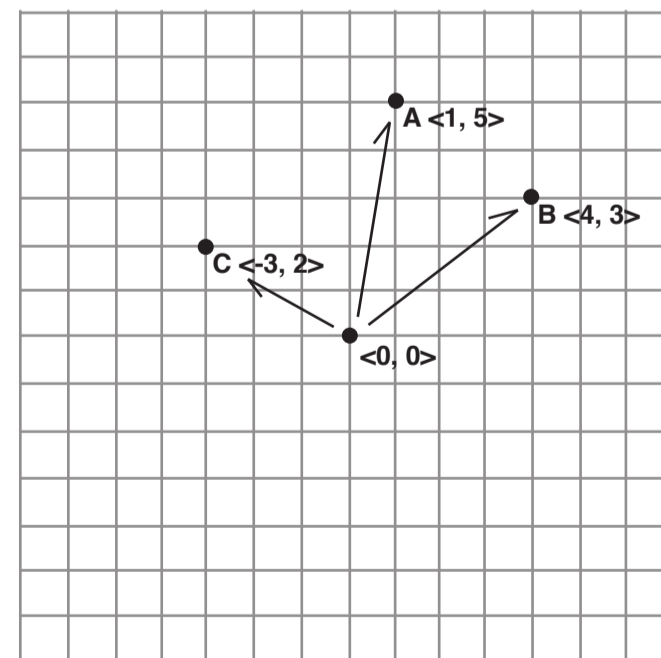
Vectors can also be multiplied together with any float value thus scaling the vector:

```
<1, 2, 3> * 5 = <5, 10, 15>
```

### 2.7 VECTOR (CONTINUED)

The addition and subtraction of vectors is useful when investigating positions in space. Because this paper is a two dimensional space, the next example will use two-dimensional vectors.

If the vectors A <1, 5> and B <4, 3> represents two positions on a x-y coordinate grid the graphical representation would look like.



By subtracting the vector B from the vector A the direction from B to A can be obtained.

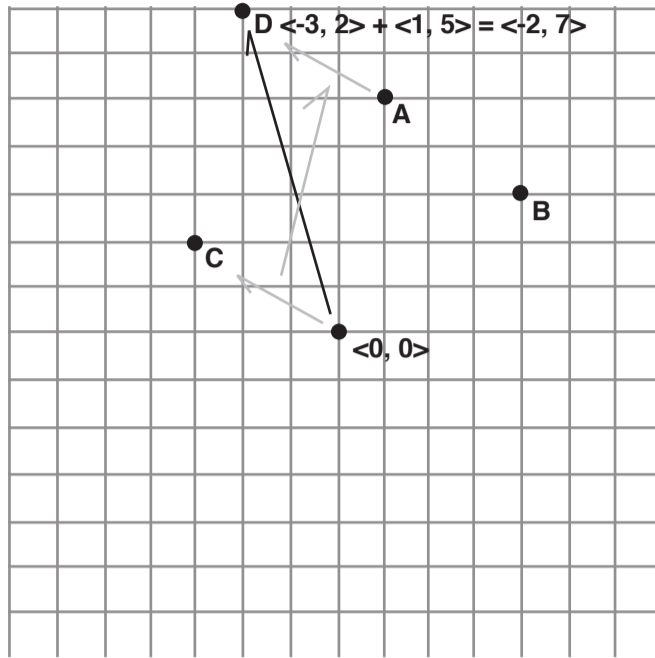
```
<1, 5> - <4, 3> = <-3, 2>
```

This new vector (**vector C**, in the diagram above) points away from origo (the point <0, 0>) in the same direction that B pongs to C.

## 2.7 VECTOR (CONTINUED)

The vector C can then be transferred to the vector A in order to obtain a point relating to A the way A relates to B. This point is the vector D.

A vector can be represented by a point as well as an arrow. In x, y, z coordinates a point is the tip of an arrow with its base in the <0, 0> coordinate. A vector can therefore describe a position as well as a direction at the same time.



In this example vector C (the direction from B to A) is added to vector A resulting in vector D, offset -3 steps on the x-axis and +2 steps on the y-axis from vector A. D can also be described as an arrow pointing from origo at <0, 0> to <-2, 7>

So in terms of Second Life geometry: The direction from your Avatar to any other object is obtain by subtracting your position from that of the object.

```
target - you = you_to_target_direction_rela-
tive_to_origo
```

## 2.2 IISensorRepeat

```
llSensorRepeat(string name, key id, integer
type, float range, float arc, float rate)
```

This function performs the same task as the IISensor function but repeats it every "rate" seconds.

This is the sensor function I use and its great!

## 2.3 SENSOR (event)

```
sensor(integer num_detected){
    any piece of code here...
}
```

This is the event that is activated whenever the IISensor or the IISensorRepeat detects a presences. The "num\_detected" integer can be used in code within the sensor event to access different objects if more than one are detected. I have used the integer value of "0" in my code which is the first avatar/object the sensor function detects.

## 2.4 NO\_SENSOR (event)

```
sensor{
    any piece of code here...
}
```

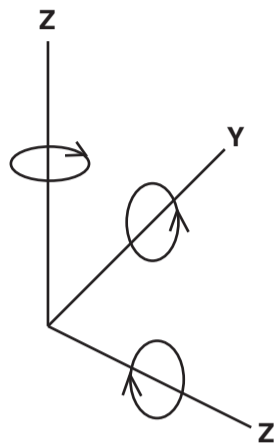
This event is activated when a sensor function (IISensor / IISensorRepeat) is active but not sensing anything.

## 2.8 ROTATION

Rotation in Second Life is defined as a **Quaternion**. A quaternion is a vector describing rotation in four dimensional space. Its very hard to visualize and, quite frankly, I don't understand how it works. The reason for four dimensional rotation is to avoid something called "Gimball lock". That, as well, is a hard to get concept and if this interests you I suggest you google it.

Luckily quaternions can be **converted into ordinary vectors** (and that makes us very happy).

A vector describing rotation describes it as three, one dimensional, rotations around each of the x, y, z-axis.



To further make things complicated rotation in Second Life is defined in **radians** instead of degrees. They, however, can also be converted as shown below:

```
vector eul = <0,0,45>;
    //45 degrees around the z-axis, in
    Euler form
eul *= DEG_TO_RAD;
    //convert to radians
rotation quat = llEuler2Rot(eul);
    //convert to quaternion
llSetRot(quat);
    //rotate the object
```

## 4 DETECTION

### 4.1 IIDetectedPos

```
vector llDetectedPos(integer number)
```

This function can be used inside the sensor event. It returns the position, in vector format (<x, y, z>), of any detected object. It is fed the "number" identifying the object. Use a value of "0" to return the position of the first object detected.

### 4.2 IIDetectedRot

```
rotation llDetectedRot(integer number)
```

Returns the rotation (in quaternion format) of the detected object.

### 4.3 IIDetectedVel

```
vector llDetectedVel(integer number)
```

Returns the velocity vector of the detected object.

Remember that vectors can be thought of as arrows as well as point. This vector defines such an arrow with its tail in the 0,0 coordinate. The length equals the speed of the object and the direction of it's trajectory.

## 3 SENSOR FUNCTIONS AND EVENTS

The part describing the LSL-functions will, regrettably, be somewhat quick and cryptic. For that I am sorry. Check out the different LSL-wiki's listed in the reference part for more detailed info.

These are the most of the functions and events i have used.

### 2.1 IISensor

```
llSensor(string name, key id, integer type,
float range, float arc)
```

**string name**

States a specific name of an avatar to scan for (can be left blank, just leave "" and that will do).

**key id**

States a specific object or avatar id to scan for (can be left blank)

**integer type**

Specifies what kind of object to scan for (1-AGENT, 2-ACTIVE, 4-PASSIVE and/or 8-SCRIPTED). The types are represented by integer. My script scans for 1-AGENT (avatars).

**float range**

Specifies the range (in in-world meters) of the scan.

**float arc**

Specifies the angle, in radians, (relative to the script's parent object) of the scan. By inserting "PI" instead of a number the scan is set to 360 degrees in all dimension.

## 5 DYNAMICS

Most of the functions in this chapter requires the scripts parent object to be set to "physical" in order to work

### 5.1 IIMoveToTarget

```
llMoveToTarget(vector target, float tau)
```

The IIMoveToTarget function is very simple and easy to work with. It tells the parent object to move towards the vector "target" with the dampening value (works like a brake) of "tau" seconds (something between 10 and 0.1 is usually a good start).

### 5.2 IIRotLookAt

```
llRotLookAt(rotation target, float strength,
float damping)
```

This function is similar to IIMoveToTarget but instead of moving it rotates the parent object towards the rotational angle of "target" with the rotational strength of "strength" (in Newtons) and the damping value of "damping" (in seconds).

It refers to the parent objects forward-facing vector.

### 5.3 llGetPos

**vector** llGetPos()

Returns the parent objects position (in global coordinates).

### 5.4 llGetRot

**rotation** llGetRot()

Returns the rotation of the parent object.

### 5.5 llGetVel

**vector** llGetVel()

Returns a vector describing the parent objects velocity in meters/second, relative to the global coordinates.

### 5.5 llGetOmega

**vector** llGetOmega()

This function returns the parent objects rotational velocity in radians/second in vector format.

## 6 MANIPULATING PRIMITIVES

### 6.1 llGetPrimitiveParams

**list** llGetPrimitiveParams(**list** params)

This is a very useful function because it can access nearly every property setting of an object: Color, texture, scale and so on).

### 6.2 llSetPrimitiveParams

llSetPrimitiveParams(**list** rule)

This function can set the properties for an object the same way llGetPrimitiveParams can access them. **This function is super!** Sadly, there is no time for me to describe it so you have to look it up yourself.

## 6 EXAMPLES

### 6.1 SIMPLE SCRIPT THAT LEVITATES AN OBJECT

```
default
{
    state_entry()
    {
        float mass = llGetMass();
        // mass of this object
        float gravity = 9.8;
        // gravity constant
        llSetForce(mass * <0,0,gravity>,
FALSE);
        // in global orientation
    }
}
```

This is a simple but very useful script that I found on one of the LSL wiki's. When you make an object, you have put in mid air, physical it will fall to the ground. This script counters the force of gravity on that object making it levitate.

### 6.2 NODE CONTROLLER SCRIPT (this is my project)

```
key name;
vector start_pos;
rotation start_rot;
integer start_pos_true = 0;
integer stop_dist = 2;
vector my_pos;
rotation my_rot;

default
{
    state_entry()
    {
        if(start_pos_true == 0){
            //checks whether start position
            is locked or not
            start_rot = llGetRot();
            //sets start rotation
            start_pos = llGetPos();
            //sets start position
        }
        llSensorRepeat("", "", AGENT, 15.0, PI, 0
.25);
    }
}

sensor(integer total_number)
{
    start_pos_true = 1;
    //locks the starting position
    my_pos = llGetPos();

    integer j;
    integer count = total_number;

    vector target_pos = llDetected-
Pos(0);
    vector target_vel = llDetected-
Vel(0);
    rotation target_rot = llDetected-
Rot(0);
    vector object_to_target =
llVecNorm(target_pos - my_pos);
    //Calculates the direction to
    the target
    vector target_to_object =
llVecNorm(my_pos - target_pos);
```



//Calculates the direction away from the target

```
//debug functions:
//llSay(0, (string)llVecMag(target_
vel));
//llSay(0, (string)my_rot);
//llSay(0, (string)target_pos);

if (llVecMag(target_vel) > 4) {
    llSetPrimitiveParams([PRIM_
POINT_LIGHT, TRUE, <1,1,1>, 0,
10.0, 0.75]);
    llMoveToTarget(my_pos + target_
to_object,0.1); //gets scared
}
else {
    llSetPrimitiveParams([PRIM_
POINT_LIGHT, TRUE, <1,1,1>,
1.0, 10.0, 0.75]);
    llMoveToTarget(target_pos, 10);
    //gets curious
    llRotLookAt(target_rot,0.2,10);
}

no_sensor()
{
    llRotLookAt(start_rot,0.5,10);
    //resets rotation
    llMoveToTarget(start_pos,0.5);
    //resets position
    llSetPrimitiveParams([PRIM_POINT_
LIGHT, TRUE, <1,1,1>, 0, 10.0,
0.75]);
}
}
```



## 6 REFERENCES

### LSL WIKI'S

- [http://rpgstats.com/wiki/index.php?title=Main\\_Page](http://rpgstats.com/wiki/index.php?title=Main_Page)  
(this one is the best)
- [http://wiki.secondlife.com/wiki/LSL\\_Portal](http://wiki.secondlife.com/wiki/LSL_Portal)
- <http://www.lslwiki.net/ls/wiki/wakka.php?wakka=HomePage>
- <http://slhomepage.com/sl/index.htm>

Ok, that's it. I won't try to explain it. Hopefully the preceding text will have done some of that but unfortunately my time is up. I hope this has been of some interest. If so; feel free to check out the links in the reference list.