Best Practices: Sending & Receiving in LSL

A Ruth 2 White Paper

This an attempt by me to put together bits and pieces of information about best practices when it comes to sending and receiving in lsl code. I tried to be as accurate as possible and have included links to the source materials, but when it comes to coding there are lots of differing opinions. So don't take it as gospel, but, at the least it, brings together information from a number of divergent sources and provides some food for thought.

-- Sundance Haiku

Introduction

In a recent latest discussion on the Ruth Google + site, we were exploring the most efficient way of sending and receiving messages. In keeping with a basic purpose of the Ruth project, we want to do this in way which causes the least impact on the virtual world. In particular, we were looking at communication between HUD's and Ruth's body parts.

I spent some time researching this, and I thought I would put together what I've learned.

Some Background

First, a review of the two important parts the LSL code when it comes to receiving messages. To receive messages, the following comes first . . .

Function: integer llListen(integer **channel**, string **name**, key **id**, string **msg**);

An incoming message is processed first by the above function. Note that the parameters of "channel," "name," and "id" are also referred in the documentation to as "filters." So, using that terminology, in this first step, you are filtering for channel – or optionally you can also filter for a "name" or "id" in addition to channel.

If the channel of the incoming messages is the same of some designated channel that we've chosen, then it triggers the listen **event** (below).

Event: listen(integer **channel**, string **name**, key **id**, string **message**)

Once the above "listen event" is triggered, you can then have the script run the desired code. That code might include turning an Alpha on or off - or changing the color of the fingernail.

Note the two types of code above. The first (llListen) is a ***function****. The second (listen) is an* ***event****. It's helpful to keep that distinction mind as you read through the rest of this.*

In a HUD situation, it is good practice to check to make sure the message is coming from the owner of the prim. It's possible (but very unlikely in our case), that some other object owned by a different person is using the same channel. You can check to make sure it's coming from the owner (the avatar which owns the HUD) by using this in the *listen event* code:

```
    if (llGetOwnerKey(id) == llGetOwner())
```

I had a problem with understanding the "id." I kept thinking it was the owner of the prim. But if a prim (or more specifically, a HUD) is sending the message, "id" *is the key, the UUID of the HUD prim*. When an avatar is sending the message then "id" is the UUID of the avatar.

In the situation when a prim (a HUD) is sending the message, you can determine the owner, but it can't be done with the llListen *function*. You have to wait until the *listen event* is triggered and then use the code above.

*So, to re-state: the "****id"*** *is either the object sending the message — or the person (avatar) sending the message. Since the main orientation in this paper is about HUD communication, the "id" will be the key, the UUID of the HUD.*

A Way of Filtering at the Function Level

Let's look at another parameter of the llListen *function*: "Name." "Name" is the name of the object that is sending the message. ("Name" can be also be the name of an avatar if it's the avatar sending the message.)

What this means is that in an environment where you might have two or more senders using the same channel, you can use the "Name" filter to identify the message you want to process. But there's another use . . .

By experimenting, I found that you can temporarily change the name of a HUD before a message is sent. That's done by using the following code:

```
        llSetObjectName("NEW PRIM NAME");
```

That has relevance in our case because it becomes another way to filter the llListen function. The nice thing about this approach is that llSetObjectName is only a temporary name change. It does not change the name in the inventory, and when the object is detached the name returns to its original.

This method works particularly well in situations where you are using one channel – such as in Ruth. By using this method, you can filter out messages not intended for a particular body part. Moreover, you are able to do this very efficiently without having to trigger the *listen event*.

It's a perfect solution to my situation where I have one HUD. Depending upon the user's choice, the HUD sends messages to the toenails or to the fingernails. By using a different name for the fingernail messages, and a different one for the toenail messages, the listen *function* can filter out the unwanted messages before they even get to the listen event.

Since the listen event is not even triggered, you don't have to parse the message to determine whether the code should continue executing or not. Using the name parameter in this way didn't show up in any of the discussions, but I suspect that scripters have utilized this technique at one time or another.

What is the Best Procedure for Sending & Receiving

There's very, very little officially on best practices for sending and receiving. In fact, I could find only one quasi-official statement in regard to "listens." This is from the LSL Wiki:

> "Listens do impact sim performance. It's better to have a single listen processing commands coming in on a single channel then it is to have multiple listens open all on the same channel. — Strife Onizuka March, 2008.

Strife has a long history with SL scripting, and we can take his advice as reliable. The only other definite information that you can suss out from the LSL Wiki regarding listens is that there is a limit of 65 listens in one script. Other than these tidbits, there was nothing else official dealing with the narrow topic of the number of listeners and channels.

On the sending end of things, the official recommendation found on the LSL Wiki is to use "llRegionSay" or "llRegionSayTo." The reason is that this form of sending messages goes through one less process. Nonetheless, there doesn't appear to be total agreement on this. More about it later.

After exhausting official sources, I turned to the Scripting Forum.

From reading the discussions in the Forum, I found a general agreement with the Strife's Wiki comment on limiting the number of listeners. I couldn't find much, however, on limiting channels. The only mention of limiting channels that I could find is Strife's comment above.

This is what Void Singer has to say about good listening code practices. (Void has, for many years, provided solid advice on the Scripting Forum and is a trusted source of information):

- Avoid them [having open listeners] when it's possible (Often not for command menus from huds)
- Never use PUBLIC_CHANNEL (zero), or DEBUG_CHANNEL (21474836487)
- Always filter as tightly as possible, in the llListen call if you can, or in the listen event for things you can't
- Try to pick an unused channel, using a mostly unique one based on the Object or Owner key may prevent collisions (may not be possible for commands that need to be enterable from user chat as well)

Lear Cale agrees with Void Singer's list of good practices. In particular, he points out that it is important to consider how many times the listen handler is triggered. By this, I assume that he means, the less you trigger the listen event, the better.

Lear goes on to say this:

> "[Good technique considers minimizing] how many chats match the llListen filters. Void made a point of this above, and correctly so. The most important one here is usually the channel, picking a good unique channel filters out nearly all "crosstalk" — objects hearing messages meant for other objects. Any other filters you can apply such as, for a HUD, the owner, also help and should be used."

In regard to llRegionSay, Void Singer says that there's some debate over whether is llRegionSay better than other sending methods. He does, however, say that llRegionSayTo is definitely more efficient.

Conclusions

What conclusions can we draw from this in relation to the Ruth project:

1. Channel Selection. We can tick off several from Void Singer's list of best practices. We do not use the public channel or the debug channel – and we are using recommended code to select a channel based on the Owner's key.

2. llRegionSayTo. The use of llRegionSayTo is a possibility for us, but since Ruth will be utilized across many different grids, we would need to know the UUID's of the receiving objects. That complicates things. For the time being, we're probably fine with our current system but it is something to keep in mind for later use.

3. Closing listeners. There was quite a bit of discussion on the Scripting Forum about closing open listeners. It can be done in lsl code by using llListenRemove(handle). However, unless I'm misreading things, in our case, we need to keep the listeners open. If anyone sees this differently, please pipe in.

4. Number of Listeners. My conclusion from all of this research is that we are well within acceptable levels — even when other avatars in the same area are taken into consideration. Since all the prims of the main part of the Ruth body are linked, we need only one listener in a small cube hidden in the neck which serves as the root. Other listeners are in the hands, feet, head and finger/toe nails. In total, we do not have any more listeners than what would be found in a typical mesh body. As a comparison, one individual posting on the Scripting Forum had created a mesh body with 200 prims and was placing a listener in each prim. (Responders on the Scripting Forum suggested that he change his approach.)

5. Number of Channels. The only comment I could find in regard to limiting channels is Stife's Wiki statement above. From his statement, however, it is reasonable to conclude that reducing the number of channels is a good idea – and it if can be kept to one channel, that's even better. The final take? If it is programmatically possible, then using one channel for all of Ruth's HUD functions is good practice.

How Best to Implement the One Channel Idea?

When using one channel, each of the listeners need to determine whether the message is meant for them. There are two ways of doing that:

- One is at the **function** level (llListen). Void Singer suggests filtering is best done at that level. Consequently, this is where the "Name" method, described above, can come in handy and will work quite well for my fingernail/toenail project. When the HUD sends a message, it will temporally change its name to reference either finger or toenails. The listeners in the fingernail or toenail mesh will then filter for the appropriate name at the function level, forgoing having to trigger the listen event.

- The other method is to filter at the listen **event** (listen). Several examples provided on the Scripting Forum utilized this method. Shin uses this method. His code is well tested, and it makes sense to stick with it. Additionally, Shin is dealing with two different processes within the same script — changes to textures and alphas — which would complicate the "Name" method.

The point of all this is to keep track of the considerations going into the creation of Ruth. But, most importantly, recording information like this serves as a learning tool for people working on their own mesh projects.

Oh, and there is one other thing . . . I need to cracking on my own code and make sure I implement all of these changes!