CHAPTER 2

MAKING YOUR AVATAR STAND UP AND STAND OUT

Uniqueness is everything in ***Second Life***. Appearance is part of that: clothing, gender, height, and species. So is action: nobody wants their avatar to look or act like anyone else's. Some of the major uses for ***SL*** scripting give avatars and content creators the mechanisms to control how avatars ***behave***, and also to make radical alterations to avatar appearance that blend seamlessly with the basics ***Second Life*** provides to every user.

This chapter addresses some of the basics of making avatars look good in the world—from getting them to sit on objects so they don't look silly (with a side trip into teleportation), to letting avatars express themselves with controllable labels, to making them more visible in the dark. It also introduces some basic ways to add objects to your avatar that act like body parts that act in concert with whatever else the avatar is doing. Finally it introduces the basics of overriding ***SL***'s standard avatar animations with those of your choosing.

MOVING YOUR AVATAR AROUND THE WORLD

There are numerous options for helping your avatar move. Chapter 6, "Land Design and Management," has an example of a security screen that stops avatars from going where they are not wanted. You'll find vehicles and jet packs in Chapter 7, "Physics and Vehicles." On the SYW website, you'll find an in-depth project on controlling animations for dance. There are many, many options, but it all starts when an avatar sits down.

SITTING

Sitting is one of several basic ways an avatar can interact with objects in ***Second Life***. Apart from the obvious "sitting on a chair" sorts of actions, sitting is also how avatars can be moved (translated and rotated) directly under script control.

Sitting on a Dumb Plywood Box

What should the first script in a book on ***Second Life*** scripting be? The simplest one. But what is the simplest script? No script at all, of course. It may not be very interesting, but it is a great starting point.

Make a box. Sit on it. Exciting, isn't it? Turns out there ***is*** something interesting to say about it: you will always sit on an unscripted object on the upper edge of what was originally the eastern side of the object, as shown in Figure 2.1. Rotate the object around the z-axis, and the sit position will also rotate.

Figure 2.1: An unscripted box causes the avatar to sit oddly.

With some care, you can build an unscripted chair you look reasonable sitting on. The trick is to remember that the root prim's original rotation is what determines where and how an avatar sits on the object. You could, for example, use a very small prim to make the avatar appear to be sitting correctly. This approach, however, eats up your prim budget pretty quickly; especially given how easy it is to write a simple script, like the one in Listing 2.1, that controls where and how an avatar sits on a prim.

BUILD NOTE

Make a cube. Right-click on the cube, and notice that the pie menu offers a Sit Here option. Sit down, and notice where your avatar sits relative to the cube. Stand up.

Drop the code from Listing 2.1 into the object. (The prim that holds the code—in this case the cube—is called the *containing prim*.) In the object's Edit tab, open the Content folder, click New Script, double-click the new script that is created, and you will get the script you saw in the previous chapter in Listing 1.1. Delete all the text you see and replace it with the script in Listing 2.1. Click Save and wait until the compiler reports that the save has completed. If there are compiler errors, double-check that your script is *exactly* what is shown in Listing 2.1.

Right-click on the cube, and notice that the pie menu has changed. Sit down again, and notice how your avatar's position has changed. (If it didn't change, that's probably because you forgot to stand up.)

Listing 2.1: Simply Sitting

```
default {
    state_entry() {
        llSitTarget(<0,0,1>, ZERO_ROTATION);
        llSetSitText("Stay awhile");
    }
}
```

This script defines where the avatar should sit by specifying the location relative to the center of the prim holding the script, and the rotation to apply to the avatar. With this script's settings, the avatar will be seated unrotated and 1 meter above the center of the cube. All settings—location, rotation, and dynamics—are relative to the cube, meaning that if the cube rotates, the avatar also rotates and stays sitting in the same position, with orientation ***relative to the cube***. Figure 2.2 illustrates this: Box 1 and the one behind it are plain unscripted boxes: a box like this looks fine if you are sitting squarely on it, but your avatar always sits somewhere on the top of the object. Box 2 is scripted with Listing 2.1. You can see that it gives you fine control of the avatar's location. Box 3 is spinning: since the position of the avatar stays constant relative to the orientation of the box, if the box changes then the avatar's position will change accordingly. Box 4's script specifies a different rotation for the sit target. Box 5 shows what happens if the sit position is really high: the avatar is about to hit the ground, having fallen 50 meters from box 5's sit target.

In addition, the script in Listing 2.1 changes the text displayed in the pie menu to be Stay Awhile instead of the default Sit Here.

Figure 2.2: A seated progression

DEFINITION

`llSitTarget(vector location, rotation rot)`

Sets the sit location relative to the containing prim. If the arguments passed are `ZERO_VECTOR` and `ZERO_ROTATION`, the sit target is canceled rather than set. A target must be set for the function `llAvatarOnSitTarget()` (discussed later in this chapter) to work correctly.

`location` — Offset of the sit target position from the center of the prim.

`rot` — Rotation of the avatar while seated at the location.

DEFINITION

`llSetSitText(string text)`

Sets the label on the sit action in the GUI pie chart.

`text` — Text used to label the sit action in the pie menu.

Both `llSitTarget()` and `llSetSitText()` are ***persistent*** features of the prim. That means that even if the script is deleted, the set text and target location remain in force until reset.

NOTE

You can use `llSetTouchText(string text)` in exactly the same way to change the label of the pie chart's touch action.

If an avatar is already sitting on the prim, the sit target doesn't do anything until the avatar stands up and sits again. The sit offset must be within 519m of the prim's center, with no axis more than 300m away. Also, an avatar must be close to the prim's location to sit directly; otherwise the avatar will attempt to walk or fly nearby before seating. When the avatar stands ("unsits"), they will do so from the current sit location, not from the prim center!

A small improvement to this script would be to use `llGetPrimitiveParams()` or `llGetScale()` to ask how big the prim is, and then calculate what the offset from the center needs to be, even if the prim is resized.

Sits and teleports are easy to build—you can pretty much drop these scripts into any sort of object you'd like. Sit scripts are very often put into "sit balls"—balls that are usually colored pink or blue to indicate that the seating is customized for female or male avatars.

TELEPORTING

Teleporting is the act of moving from one place to another very quickly. It is the basis for many types of movement in ***SL***, including movement of vehicles. Listing 2.2 is a simple teleport; you'll augment the script to go greater distances, improve the teleport speed, and, finally, turn one of ***SL***'s limitations into a benefit, allowing you to cheat a little on the math!

Short-Distance Teleporter

In the previous section you saw that an avatar sits down relative to the center of the prim holding the script, and stands up at the sit target. You can use this same sit-displacement trick to teleport an avatar up to 519m away: you set the target position to your teleport location, then forcibly unsit the avatar at the destination.

You need to make two important changes to the basic sit script. The first is to figure out where the target position is relative to the prim's center. The second is to figure out when the avatar has arrived at the sit location so that you can unsit them.

Before testing this script, replace the `DEST` (destination) coordinates with something near the object so that you don't lose your avatar!

Listing 2.2: Teleport1—A Local Teleporter

```
vector DEST = <128,128,200>; // replace with nearby coordinates
default
{
    state_entry() {
        rotation primCurrentRotation = llGetRot();
        vector primCurrentPosn = llGetPos();
        vector targetOffsetPosn = DEST-primCurrentPosn;
        vector targetPosition = targetOffsetPosn/primCurrentRotation;
        rotation targetRotation = ZERO_ROTATION/primCurrentRotation;
        llSitTarget(targetPosition,targetRotation);
        llSetSitText("Teleport!");
    }
    changed(integer changebits) {
        if (changebits & CHANGED_LINK) {
            key av = llAvatarOnSitTarget();
            if (av != NULL_KEY) {
                llSleep(0.1);
                llUnSit(av);
            }
        }
    }
}
```

`DEST` is a destination target location that is relative to the sim, but sit targets must be relative to the prim. The first five lines of the `state_entry()` method convert `DEST` from sim-local to prim-local. `llGetRot()` calculates the rotation of the prim relative to the sim, while `llGetPos()` calculates the prim's location. The next line calculates the location of the target relative to the prim's current location:

```
vector targetOffsetPosn = DEST-primCurrentPosn;
```

Remember that it must be within 519m, with no one axis more than 300m away! The next two lines adjust for the fact that the prim might be rotated:

```
vector targetPosn = (DEST-targetOffsetPosn)/primCurrentRotation;
rotation targetRotation = ZERO_ROTATION/primCurrentRotation;
```

If you just moved the prim to ***targetOffsetPosn*** then the avatar could end up on any point on the sphere of that radius! Setting the target rotation to be the same as the prim's rotation is polite, but not strictly necessary. The underlying math is natively handled by quaternions: see the section called "Using Quaternions" in Chapter 4, "Making and Moving Objects," for more details.

To figure out when the avatar has arrived at the sit location so that you can unsit them, the script relies on the `changed()` event. It is invoked under many conditions, including change of color, owner, or location. The argument ***changebits*** is a bit pattern that indicates exactly what has changed, but doesn't tell you ***how*** it changed; if you need to know the prior values, your script needs to get them from `llGetPrimitiveParams()` and cache those values. Because an avatar sitting on an object acts like an object being added to the linkset, we first check to see if the `CHANGED_LINK` bit is set in the pattern, using the bitwise "and" operator, `&`.

EVENT DEFINITION

`changed(integer changebits) { }`

This event is invoked whenever the prim changes in one of several ways, including whether links have changed or an avatar has sat on the prim. Additional documentation about event handlers can be found at the SYW website.

changebits — The bit pattern describing what has changed in the prim.

The bits can be tested using the bitwise arithmetic, as shown in the following snippet:

```
if (changebits & CHANGED_LINK) {
    llOwnerSay("Object changed number of links");
}
```

The script then checks to see whether an avatar is actually sitting on the object using the function `llAvatarOnSitTarget()`, which returns the key of the avatar sitting on the prim, or `NULL_KEY` if nobody is sitting. Note that it will also return `NULL_KEY` if a sit-target position is not specified. If an avatar is seated, the script infers that it is the one that just sat. To finish the teleport, the script pauses for a moment using `llSleep(float time)` to make sure the sim has repositioned the agent, and then the script ejects the seated avatar.

DEFINITION

`key llAvatarOnSitTarget()`

Returns a key that is the UUID of the user seated on the prim. If the prim lacks a sit target or there is no avatar sitting on the prim, then `NULL_KEY` is returned. If the sit target position has not been explicitly set or if the script called `llSitTarget(ZERO_VECTOR, ZERO_ROTATION)`, then `llAvatarOnSitTarget()` will return `NULL_KEY`.

DEFINITION

`llUnSit(key id)`

If the agent identified by *id* is sitting on the object the script is attached to, or is over land owned by the object's owner, the agent is forced to stand up.

Medium-Distance Teleporter

But what if you want to teleport the sitter farther than 519m, or, more likely, more than 300m along one axis (say, straight up to your skybox)? The simplest option is to move the object while the avatar is seated, unseat the avatar at the destination, and return to the original location. The function `llSetPos()` will move the prim.

DEFINITION

`llSetPos(vector newLoc)`

Sets the position of the prim, up to a maximum of 10m away.

newLoc — Target location. *newLoc* can be interpreted three ways, and is thus very confusing.

Essentially, *newLoc* is interpreted relative to whatever the prim is attached to:

- If the containing prim is attached to the sim (it is a root prim not attached to an avatar), *newLoc* is the region coordinates.
- If the containing prim is attached to an avatar (it is a root prim attached to an avatar), *newLoc* is relative to the attachment point on the avatar.
- If the prim is a child in a linked object, *newLoc* is relative to the root prim.

Unfortunately, it's not as straightforward as saying `llSetPos(`*destination*`)`: the complication is that an object may move a maximum of 10m at a time. Thus the script in Listing 2.3 takes the distance from the prim's original location to the target location, and divides it into 10m steps. However, repeated calls to `llSetPos()` reveal an unfortunate side effect: each call causes the script to pause (artificially) for 0.2 seconds, leading to a maximum effective "teleport speed" of 50 m/s. (Use `llSetPos()`, and a small value for *jumpdist*, if you want a slow teleport—for example, if you're making an elevator.)

The script in Listing 2.3 completes the teleport with a single 0.2-second delay by using the function `llSetPrimitiveParams()`. It takes advantage of the fact that you can supply a list of repeated `PRIM_POSITION` tuples (an ordered list of values); `llSetPrimitiveParams()` can move the prim to the destination in 0.2 seconds regardless of the distance.

Listing 2.3: Teleport2—Intrasim Teleports

```
// this script must be in the root prim (or the only prim) to work

vector DEST = <128,128,200>; // the teleport will be to this location
vector SITPOS = <0,0,0.5>;

moveTo(vector origin, vector destination, float jumpdist) {
    vector relativeDestination = destination–origin;
    float dist = llVecMag(relativeDestination);
    if (jumpdist > 10.0) jumpdist = 10.0;
    integer steps = llCeil(dist / jumpdist) + 1;
    vector distanceVector = relativeDestination / steps;
    integer i;
    vector currPosition = origin;
    list params = [];
    for (i=0; i<steps; i++) {
        currPosition += distanceVector;
        params += [PRIM_POSITION, currPos];
    }
    llSetPrimitiveParams(params); // actually move the prim
}
teleport(key av) {
    vector origin = llGetPos();
    moveTo(origin, DEST, 10.0);
    // no need to sleep -- llSetPrimParams has 0.2s delay
    llUnSit(av);
    moveTo(DEST, origin, 10.0);
}
default
{
    state_entry() {
        llSitTarget(SITPOS, ZERO_ROTATION);
        llSetSitText("Teleport!");
    }
    changed(integer changebits) {
        if (changebits & CHANGED_LINK) {
            key av = llAvatarOnSitTarget();
            if (av != NULL_KEY) {
                teleport(av);
            }
        }
    }
}
```

Quicker Intrasim Teleport

A popular optimization of this technique makes use of the fact that if you try to move an object by more than 10m, the effect will be to move it exactly 10m toward the target. This means you can avoid computing all of the intermediate steps, and instead use the same `[PRIM_POSITION, dest]` expression for every tuple in the list. It also allows you to build up the list of teleportation steps much more quickly by doubling the list each time instead of inserting one element at a time, as shown in Listing 2.4.

Listing 2.4: Teleport3—Optimized Intrasim Teleport

```
moveTo(vector origin, vector destination ) { // removed jumpdist
    float dist = llVecDist(origin, destination);
    integer passes = llCeil( llLog(dist/10.0) / llLog(2.0) );
    integer i;
    list params = [PRIM_POSITION, destination];
    for (i=0; i<passes; i++) {
        params = (params=[]) + params + params;
    }
    llSetPrimitiveParams(params);
}
```

We construct the ***params*** list by doubling it each time through the loop. The local variable ***passes*** is set to the smallest whole number exponent of 2 that can be used to divide the distance to be covered into chunks of less than 10m; i.e., $2^{passes} > (dist \div 10.0)$.

NOTE

The odd-looking extra `(params=[])` in Listing 2.4 is a trick to get the LSL compiler to save a significant amount of memory when manipulating large lists. The most robust version of this technique can be found on the LSL wiki as LibraryWarpPos.

Intersim Teleporter

As of the writing of this book, you cannot teleport across sim boundaries as easily as you can within a sim. One of your options is to have your object give the avatar a landmark to the destination point (as shown in Chapter 8, "Inventory"). While it isn't a teleport, it is simple to program. Another option is to expose a map of the destination point using `llMapDestination()` when your teleporter object is touched, as shown in Listing 2.5.

DEFINITION

`llMapDestination(string simname, vector position, vector lookat)`

Exposes a map showing the specified point. This works only within one of the three `touch()` events.

simname — The name of the destination sim.

position — The position of your destination in the named sim.

lookat — A point within the sim that the map should indicate is being looked at; currently ignored.

Listing 2.5: Teleport4—Intersim Teleport Assistant

```
string SIM = "Hennepin";
vector LOC = <46,144,107>;  // LM for Scripting Your World
default {
    touch_start(integer n) {
        llMapDestination(SIM, LOC, ZERO_VECTOR); // 3rd param ignored
    }
}
```

It is possible to use the prim-movement techniques to cross sim boundaries if enough care is taken. However, sim crossings are perilous, and thus we suggest you wait to try it until you are an expert scripter.

ATTACHMENTS

An attachment is an object made from one or more prims that you wear on your avatar. Many clothes are attachments, as are most accessories, such as handbags, hats, and earrings. Even some body parts, like hair, can be attached to your avatar. Wherever you go, your attachments go with you. There are many places you can attach something—about 30 different places on your avatar's body, and also eight heads-up display (HUD) locations. A HUD is like a controller board that appears only to you—other residents won't see it. (The SYW website has a section on bowling; it presents a HUD that controls the fine-grained direction of the bowling ball.)

Every object in your inventory can be attached. Most people try to make sure the attachment location makes sense aesthetically: a skirt on the pelvis and a hat on the head, for example. (Note, however, that a hat might actually be attached to the chin or the nose, allowing something else, like hair, to be attached to the head.) Attachments can even be made invisible—really useful for attachments that do something but don't need to be seen, such as flip tags (discussed in the "Flip Tag" section of this chapter).

Full avatar building is beyond the scope of this book but we can give you a taste of what is possible. Human avatars generally have their shapes changed with a combination of basic appearance controls, the wearing of layers of clothing (including the skin layer), and prim-based add-ons like clothing or prim hair. Human prim clothing and body parts are rarely heavily scripted; normally they're limited to customization (for instance, coloring hair). Sexual body parts are the notable exception. For an example of a scripted item of clothing, see the "Dialogs" section of Chapter 3, "Communications."

You can also extend prim clothing to seemingly change the shape of avatar bodies, surrounding the "natural" body parts with differently shaped parts and adding totally new parts, as shown with the Frog Prince in Figure 2.3. This costume is available for free at enkythings, Orbiuna <200, 96, 351>.

Figure 2.3: The Frog Prince is constructed by creating body parts that surround the standard avatar body.

The rest of this section describes attachments that are fun to add to your avatar: a descriptive tag, a face light, and a key for a wind-up toy. (Additionally, the SYW website has a discussion of auto-deploying wings.) This section also shows how to figure out whether an object is attached, and how to automatically attach it where it belongs.

FLIP TAG

Flip tags are very common avatar attachments that allow you to express yourself more persistently than chat, but more dynamically than group membership. They are essentially invisible (or hidden) objects with a script that translates chatted commands so that they set the object's "floating text" property, as shown in Figure 2.4. Listing 2.6 shows a simple flip tag.

Figure 2.4: This avatar is wearing an object that announces he is fully scripted!

Listing 2.6: Flip Tag—Label Yourself

```
default
{
    on_rez(integer start_param) {
        llResetScript();
    }
    state_entry() {
        llListen(9, "", llGetOwner(), "");
    }
    listen(integer channel, string name, key id, string message) {
        llSetText(message, <1,1,1>, 1.0);
    }
}
```

BUILD NOTE

There are several places you can add the script from Listing 2.6. You can add it to an object you are already wearing, such as your hair or glasses. You can create an object to hold it, and then wear the object on an attachment point, such as your nose or ear, and then move it into the place you want. You probably will want to make the prim invisible using a full alpha texture once you like the position, as wearing a plywood box on your head isn't the most aesthetic of fashion statements. (Use Ctrl+Alt+T to keep seeing it.) You can also wear it on a HUD point, but as with all HUD objects, you'll be the only one who can see it.

This script listens for commands typed by the owner and displays them as visible text. To use the script, chat ***/9Hello World!*** when wearing your flip tag. The ***9*** specifies which channel to chat on. `llListen()` sets up a listener event, `listen()`, that is called every time a chatted message passes the filters specified in `llListen()`. This script ensures that the listener pays attention to only channel number 9, and only when messages come from the owner of the containing object by specifying `llGetOwner()`. Chapter 3 describes the `llListen()` family of functions in more detail.

DEFINITION

```
key llGetOwner()
```

Returns the key of the owner of the scripted object.

This script's `listen()` event handler sets the prim's floating text using `llSetText()`. Note that since the `llListen()` call listens only to the owner, there is no need to check whether the source of the chat is the right person. If the `llListen()` call specified `NULL_KEY` instead of `llGetOwner()`, the `listen()` event handler would accept changes from anyone chatting nearby on channel 9. Floating text is a persistent feature of the prim, in that it will stick even if the script is deleted or ownership is transferred. Just as for the sit target and sit text, it needs to be explicitly removed from the object.

DEFINITION

```
llSetText(string text, vector color, float alpha)
```

Sets the floating text associated with the enclosing prim. This is a persistent feature of the prim.

text — The text for the prim, or `""` to turn it off. New lines are OK, but lines must contain at least one character (e.g., a space) and the total must be less than 256 characters.

color — The color with which to display the text.

alpha — The alpha for the displayed text (0.0 = transparent, 1.0 = opaque).

A common enhancement to the flip-text script is to allow the owner to change the text display color. You will find this augmented listing at SYW HQ; Listing 2.7 uses a similar mechanism to parse the user's command.

FACE LIGHT

Second Life environments can get very dark, and while darkness can be beautiful, scary, or romantic, it's often annoying when it is so dark that avatars cannot recognize each other's faces, as shown in Figure 2.5. A favorite remedy is a face light—a gadget you can have your avatar wear that illuminates their face with light from an invisible source. The required code is shown in Listing 2.7.

Figure 2.5: Face lighting makes it much easier to see avatars in the dark.

Listing 2.7: Face Light—Illuminate Me

```
vector gColor = <1,1,1>;
integer gLightOn = FALSE;

lightControl()
{
    llSetPrimitiveParams([PRIM_COLOR, ALL_SIDES, <0,0,0>, 0.0,
                          PRIM_POINT_LIGHT, gLightOn, gColor, 1.0, 10.0, 0.75]);
}

vector parseColor(string text) {
    list rgb = llParseString2List(text, [","," ","<",">"], []);
    vector color256 = <llList2Integer(rgb,0),
                       llList2Integer(rgb,1),
                       llList2Integer(rgb,2)>;
    return color256/255.0;
}

default
{
    on_rez(integer start_param) {
        llResetScript();
    }
    state_entry() {
        llListen(9, "", llGetOwner(), "");
        gLightOn = FALSE;
        lightControl();
    }
    listen(integer channel, string name, key id, string message) {
        if (llGetSubString(message, 0, 0)=="=") {
            gColor = parseColor(llGetSubString(message,1,-1));
        } else {
            if (message == "on") {
                gLightOn = TRUE;
            } else if (message == "off") {
                gLightOn = FALSE;
            }
        }
        lightControl();
    }
}
```

All the `listen()` components and color parsing are the same as in the flip-tag code. The script interprets the message as either a color (this time, to be used as the color of the projected light), or as an `"on"` or `"off"` command. If it sees a color specification it parses the color as in the flip tag; otherwise it updates the state. Then the script updates the light effect by invoking the new function `lightControl()`.

The `lightControl()` function invokes a new mechanism of the `llSetPrimitiveParams()` function, `PRIM_POINT_LIGHT`:

```
llSetPrimitiveParams([PRIM_POINT_LIGHT,
                      gLightOn, // Boolean, TRUE means turn light on
                      gColor,   // light color vector
                      1.0,      // intensity (0.0-1.0)
                      10.0,     // radius (0.1 - 10.0)
                      0.75]);   // falloff (0.01 - 2.0)
```

`PRIM_POINT_LIGHT` is described in more detail in the "Lights" section of Chapter 9, "Special Effects." Appendix A, "Setting Primitive Parameters," describes all the prim properties you can set using `llSetPrimitiveParams()`. You will want to play with the intensity, radius, and falloff values to find the most pleasing effect for your situation.

BUILD NOTE

The simplest way to build a face light is to rez a plain box and drop in the code from Listing 2.7. Wear the box on the attachment point you'd like (such as the nose), then edit the box while you are wearing it, moving it to about a meter in front of your face. Change the size to be a 1cm cube. The script automatically makes the box transparent in `lightControl()` by setting the `PRIM_COLOR` attribute to a 0.0 alpha.

WIND ME UP!

Listing 2.8 animates a key that sticks out of the back of an avatar like a wind-up toy, as shown in Figure 2.6. This script has two pieces: it spins the key according to simple rules, and it allows other avatars to wind you up, altering the parameters of the spinning key to make it spin faster.

Figure 2.6: Wind me up and let me go.

Listing 2.8: Wind Me Up!

```
// rotate around (original) z axis (this is a direction!)
vector SPINAXIS = <0,0,1>;

float MAXRATE = 10.0;     // maximum spin rate—radians/sec
float DECAYTIME = 10.0;   // seconds to wake between slowing down
float DECAYLEVEL = 0.8;   // each decay cuts the speed by this much
float WINDLEVEL = 2.0;    // how much does each wind add

float gSpin = MAXRATE;    // how fast are we currently spinning?
float gMinSpin = 0.1;     // keep spinning a little when wound down

spin() {
    llTargetOmega(SPINAXIS, gSpin, 1.0);
}

default {
    // no on_rez—we want to maintain spins level across rezzes
    state_entry() {
        spin();
        llSetTimerEvent(DECAYTIME);
    }
    timer() {
        gSpin *= DECAYLEVEL;
        if (gSpin<gMinSpin) gSpin = gMinSpin;
        spin();
    }
    touch_end(integer count) {
        integer i;
        for (i=0; i<count; i++) {
            key winderKey = llDetectedKey(i);
            if (winderKey != llGetOwner()) {
                string winder = llDetectedName(i);
                llWhisper(0, winder+" has wound you");
                gSpin += WINDLEVEL;
            }
        }
        if (gSpin>MAXRATE) gSpin = MAXRATE;
        spin();
    }
}
```

BUILD NOTE

To use the script in Listing 2.8, the simplest thing to do is create a box sized x=0.01, y=1.0, z=1.0. Turn it into a triangle with a lot of *y* taper. Drop the script into the box's inventory, and then wear it on your spine. (You can attach items to the spine using the GUI, but, as noted later, you can't do it via scripting because there's no `ATTACH_SPINE` constant.) You can adjust the size and position while you're wearing it. If you've got a convenient key texture, make the (square) box transparent using a full-alpha texture (i.e., completely transparent), and then apply a texture that looks like a clock key to the large faces—the point that would enter the clock should be down and the handle should be up.

To actually spin the object, the script uses the `llTargetOmega()` function described in Chapter 4. It spins an object around the axis vector that runs through its center. This script specifies a vector that points straight up (`<0, 0, 1>`). Why, you ask, since it is going to be spinning around and pointing behind you (not upward)? Because when it is attached to your avatar's spine, its frame of reference will have been

rotated so that its "local up" is sticking straight behind you. Perhaps a little confusing, but the important thing is consistency. This script never has to worry about which direction the avatar is facing—all it ever has to do is spin around the z-axis. Making sure the axis is normalized (that is, its length is 1.0) is good practice. If it is normalized, the rate parameter is how quickly the object will turn in radians per second. For now the important thing about the third parameter is that it not be 0; otherwise the rate at which the key spins won't ever change.

The script sets a maximum and minimum spin rate, and sets a timer to make the spin rate decay slowly. Each time the `timer()` event fires, the script updates the spin rate to the slow decay.

The last part of this script makes the attachment more fun and social: only a different avatar can wind your key. It uses the `touch_end()` event handler to capture when avatars finish touching the wind-up key. Note that `touch()` and `touch_start()` are similar event handlers.

EVENT DEFINITION

```
touch_start(integer numDetected) { }
touch(integer numDetected) { }
touch_end(integer numDetected) { }
```

These events are invoked when the user touches an object. `touch_start()` is used once at the start of a touch, `touch()` is called repeatedly during a touch, and `touch_end()` is called when the touch is over. The SYW website provides additional documentation on event handlers.

numDetected — The number of detected touch events.

The `touch_end()` event handler follows all sensor-event patterns: ***any number*** of similar events can be bundled into a single invocation of an event handler. This is indicated when the parameter ***`count`*** is greater than 1. To find the different event sources (in this case, who touched our key), you can call the `llDetectedKey()` function with the zero-based index of the object you are interested in. Table 2.1 shows a list of similar functions and their behaviors. Since this script cares both about ***how many times*** the key is wound and ***who*** is winding the key, it loops over the keys of all detected touchers, each time increasing the spin rate and telling anyone nearby about it. Finally, the script makes certain it hasn't been overwound and updates the spin rate.

TABLE 2.1: FUNCTIONS THAT DESCRIBE DETECTED OBJECTS

Function	Behavior
`key llDetectedKey(integer number)`	Returns the UUID of the detected object.
`string llDetectedName(integer number)`	Returns the name of the detected object.
`vector llDetectedPos(integer number)`	Returns the position of the detected object.
`rotation llDetectedRot(integer number)`	Returns the rotation of the detected object.
`vector llDetectedVel(integer number)`	Returns the velocity of the detected object.
`key llDetectedOwner(integer number)`	Returns the object owner's UUID.
`integer llDetectedGroup(integer number)`	Returns a Boolean value representing if the detected object or avatar is in the same group that the prim containing the script is set to.
`key llDetectedCreator(integer number)`	Returns the object creator's UUID.

Function	Behavior
`vector llDetectedGrab(integer number)`	Returns the grab offset of the user touching the object; that is, where the user is compared to the object.
`integer llDetectedType(integer number)`	Returns a bitmask that indicates the types of detected object: `AGENT` for an avatar, `ACTIVE`, `PASSIVE`, and/or `SCRIPTED` for objects. (See Chapter 10, "Scripting the Environment," for more details.)
`integer llDetectedLinkNumber(integer number)`	Returns the link number of the prim that triggered a touch or collision event. (See Chapters 3 and 10 for more details.)

DISCOVERING WHETHER OBJECTS ARE ATTACHED

You can find out whether an object is attached by using the function `llGetAttached()`. This function is very useful for changing the behavior of an object—for example, if the face light is attached the object becomes invisible, but when it is rezzed to the world it makes itself visible. The bowling project on the SYW website shows several additional examples.

DEFINITION

`integer llGetAttached()`

Returns an integer identifying where the object is attached, or 0 if it is not attached. There are 36 possible values, including all the avatar body parts and HUD locations. (Values 29 and 30 correspond to the left and right pectorals, respectively, but their constants should not be used; see JIRA bug SVC-580. The is also no `ATTACH_SPINE` constant.) LSL is still evolving, and it's peppered with such small oddities.) The following table shows the constants and their associated values.

Constant	Value
ATTACH_CHEST	1
ATTACH_HEAD	2
ATTACH_LSHOULDER	3
ATTACH_RSHOULDER	4
ATTACH_LHAND	5
ATTACH_RHAND	6
ATTACH_LFOOT	7
ATTACH_RFOOT	8
ATTACH_BACK	9
ATTACH_PELVIS	10
ATTACH_MOUTH	11
ATTACH_CHIN	12
ATTACH_LEAR	13
ATTACH_REAR	14
ATTACH_LEYE	15
ATTACH_REYE	16
ATTACH_NOSE	17
ATTACH_RUARM	18

Constant	Value
ATTACH_RLARM	19
ATTACH_LUARM	20
ATTACH_LLARM	21
ATTACH_RHIP	22
ATTACH_RULEG	23
ATTACH_RLLEG	24
ATTACH_LHIP	25
ATTACH_LULEG	26
ATTACH_LLLEG	27
ATTACH_BELLY	28
ATTACH_HUD_CENTER_2	31
ATTACH_HUD_TOP_RIGHT	32
ATTACH_HUD_TOP_CENTER	33
ATTACH_HUD_TOP_LEFT	34
ATTACH_HUD_CENTER_1	35
ATTACH_HUD_BOTTOM_LEFT	36
ATTACH_HUD_BOTTOM	37
ATTACH_HUD_BOTTOM_RIGHT	38

Listing 2.9 uses `llGetAttached()` to detect whether the face light is attached, and if not, to attach the face light to the avatar's chin using `llAttachToAvatar()`.

Listing 2.9: Self-Attaching Face Light

```
// This script extends Listing 2.7 with the following code:
default
{
    state_entry() {
        llListen(9, "", llGetOwner(), "");
        gLightOn = FALSE;
        lightControl();
        if (llGetAttached() == 0) {
            llRequestPermissions(llGetOwner(), PERMISSION_ATTACH);
        }
    }
    run_time_permissions( integer perms ) {
        if (perms & PERMISSION_ATTACH) {
            llAttachToAvatar(ATTACH_CHIN);
        }
    }
}
```

DEFINITION

`llAttachToAvatar(integer attachPoint)`

Attaches the object to the owner after the owner has granted `PERMISSION_ATTACH` to the script. The object is taken into the user's inventory and attached to the specified attach point.

`attachPoint` — Indicates the attachment point using one of the `ATTACH_*` constants or a valid integer value.

DEFINITION

`llDetachFromAvatar()`

Detaches the object from the avatar and places the object in inventory. There is no way to place the object down in-world, for example on the ground.

Knowing the value of a named constant can be useful. (The values for all LSL constants are shown on the SYW website.) For example, you can use the following test to have your script determine whether the object is attached as a HUD:

```
if (llGetAttached() >= 31) {
    // then the object is attached as a HUD
}
```

However, it is best not to rely heavily on the actual values because LSL may change—what might your code do if Linden Lab were to add another `ATTACH` constant, say `ATTACH_SPINE`, with a value of 39? Another consideration is that code using the ***names*** of constants is far easier to read than code using their ***values***.

When an object attaches or detaches, the `attach()` event handler is triggered. The script will have very little time to execute after the user has detached the object. You can augment the script in Listing 2.9 with the following snippet:

```
attach( key id ) {
    if (id == NULL_KEY) {
        llOwnerSay("Detached; Returning to inventory.");
    } else {
        llOwnerSay("Successfully attached.");
    }
}
```

EVENT DEFINITION

`attach(key id) { }`

This event is invoked when an object attaches or detaches from an avatar. The SYW website provides additional documentation.

`id` — The UUID of the avatar the object was attached to. If ***`id`*** is `NULL_KEY`, the object was detached and the script will exit shortly because the object is being returned to inventory. ***`id`*** will be forgotten if `llResetScript()` is called.

Attaching and detaching objects requires the owner to grant `PERMISSION_ATTACH`. You request these permissions using the `llRequestPermissions()` function. The event `run_time_permissions()` is triggered when permissions are granted.

Once granted, a script retains permissions until they are released (when different permissions are requested or when the object is returned to inventory, for instance). If you ask for permissions that have already been granted to the script, the agent is not asked again and the `run_time_permissions()` event is triggered immediately. Also note that script permissions are not additive (additional requests replace any existing permissions granted) and a script may have permissions granted to only one avatar at a time.

NOTE

Some permissions are implicitly granted (that is, without a dialog) when the avatar sits down or wears an attachment. If you know that permissions are implicit, you can take advantage of that fact to make your code slightly more efficient. Chapter 7 shows an example of this idea in Listing 7.6.

DEFINITION

`llRequestPermissions(key avatarId, integer permissionsRequested)`

Asynchronously requests permissions. A `run_time_permissions()` event will be triggered when the permissions are granted. Nothing is generated if the user refuses to grant permissions.

`avatarId` — The avatar to get permissions from.

`permissionsRequested` — A bit pattern (also called a bitfield) indicating the permissions being requested. The following table shows the permissions that can be requested.

Constant	Value	Behavior	Granted By
`PERMISSION_DEBIT`	0x002	Take money from agent's account.	Owner
`PERMISSION_TAKE_CONTROLS`	0x004	Take avatar's controls.	Anyone
`PERMISSION_TRIGGER_ANIMATION`	0x010	Trigger animation on avatar.	Anyone
`PERMISSION_ATTACH`	0x020	Attach/detach from avatar.	Owner
`PERMISSION_CHANGE_LINKS`	0x080	Change links.	Owner
`PERMISSION_TRACK_CAMERA`	0x400	Track avatar's camera position and rotation.	Anyone
`PERMISSION_CONTROL_CAMERA`	0x800	Control avatar's camera.	Anyone

EVENT DEFINITION

`run_time_permissions(integer permissionsGranted)`

This event is invoked when the script is granted permissions. There is no event for when the avatar refuses to grant permissions. The SYW website provides additional documentation.

`permissionsGranted` — A bitmask of the permissions that were granted.

Two other useful permissions functions are `llGetPermissions()`, which tells the script which permissions have been granted—identical to the parameter for `run_time_permissions()`—and `llGetPermissionsKey()`, which indicates who granted the permissions.

DEFINITION

```
integer llGetPermissions()
```

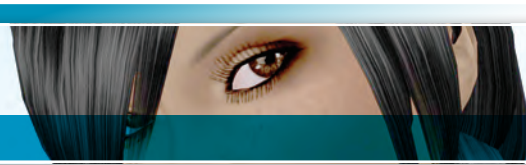
Returns an integer bit pattern with the permissions the script has been granted. Permissions can be tested with a bitwise "and" operator (&), as shown in the following snippet:

```
integer grantedPerms = llGetPermissions();
if (grantedPerms & PERMISSION_TAKE_CONTROLS) {
    llOwnerSay("Script has permission to take controls.");
}
```

DEFINITION

```
key llGetPermissionsKey()
```

Returns the key of the avatar that granted permissions.



ANIMATION

An animation is a set of instructions that cause an avatar to engage in a sequence of motions. An animation describes the way your avatar walks, dances, waves hello, or even sits down. One of the first things that people notice when they start out in ***Second Life*** is how awkward and artificial the default avatar animations look. So early in an avatar's life, people want to upgrade by adding an animation override, or AO, that replaces the default settings with higher-quality, more realistic, or special-purpose animations.

NOTE

You can create your own custom animations with tools such as Poser (`http://graphics.smithmicro.com/go/poser`) and Blender (`www.blender.org`), and then upload them into *SL*. Additionally, many high-quality animations are available in-world, many of them for free. Search for Animation in the All tab of the search window. Many people purchase a set of animators that have been packaged in a wearable attachment with an AO script, ready to go.

ANIMATION OVERRIDES

The various AO scripts differ in relatively minor ways, but the basic idea is always the same: to watch what your avatar is doing, and override the default animation with one of your choice. Listing 2.10 replaces the default walking animation with a floating crossed-leg pose; this is possibly the simplest AO script. It may look pretty complicated, but we'll walk through the pieces slowly. Place the script in any attachment, such as a bracelet or a transparent hat. Most of the complexity comes from setting up permissions to animate the avatar—extending the methods shown in Listing 2.10—and that is functionality you will use in a lot of places.

Listing 2.10: WalkAO—Why Walk When You Can Float?

```
// the animation we're going to override
string AN_TRIGGER = "Walking";
// which animation are we overriding walking with?
string AN_OVERRIDE = "yoga_float";
integer gOverriding = FALSE;  // are we running the override right now?
integer gHasPermission = FALSE; // did we get permission yet?
integer gEnabled = TRUE;        // is the AO on?

animOverride() {
    if (gHasPermission) {
        string  curAnimState = llGetAnimation(llGetOwner());
        if (curAnimState == AN_TRIGGER && gEnabled) {
            if (!gOverriding) { // already overriding? skip
                gOverriding = TRUE;
                llStartAnimation(AN_OVERRIDE);
            }
        } else {
            if (gOverriding) {
                llStopAnimation(AN_OVERRIDE);
                gOverriding = FALSE;
            }
        }
    }
}

default {
    on_rez( integer _code ) {
        llResetScript();
    }
    state_entry() {
        gOverriding = FALSE;
        gHasPermission = FALSE;
        gEnabled = TRUE;
        if ( llGetAttached() ) {
            llRequestPermissions(llGetOwner(),
                                 PERMISSION_TRIGGER_ANIMATION|
                                 PERMISSION_TAKE_CONTROLS);
        }
        llSetTimerEvent(1.0);
```

```
    }
    run_time_permissions(integer parm) {
        if( parm == (PERMISSION_TRIGGER_ANIMATION|
                     PERMISSION_TAKE_CONTROLS) ) {
            llTakeControls(CONTROL_DOWN|CONTROL_UP|
                           CONTROL_FWD|CONTROL_BACK|
                           CONTROL_LEFT|CONTROL_RIGHT|
                           CONTROL_ROT_LEFT|CONTROL_ROT_RIGHT,
                       TRUE, TRUE);
            gHasPermission = TRUE;
        }
    }
    attach( key k ) {
        if ( k != NULL_KEY ) {
            llRequestPermissions(llGetOwner(),
                PERMISSION_TRIGGER_ANIMATION|PERMISSION_TAKE_CONTROLS);
        }
    }
    control( key _id, integer _level, integer _edge ) {
        animOverride();
    }

    timer() {
        animOverride();
    }
}
```

NOTE

Several of the scripts in this book have function or event handler parameters that begin with an underscore (_). This naming convention is sometimes used to indicate parameters that will not be used by the function.

`AN_TRIGGER` names the posture the script is going to animate: there are 21 standard *SL* postures that describe the sort of basic activity an avatar can be performing at a given time—examples include `"Walking"` (used in this script), `"Sitting"`, and `"Flying"`. The wiki page on llGetAnimation has a complete, current list.

`AN_OVERRIDE` names the animation the script will play when the avatar is detected to be in the chosen posture. This may be the name of a built-in animation or the name of an animation in the prim's inventory. For simplicity's sake, this script uses the built-in `"yoga_float"` animation (Figure 2.7). Note that some animations stop after they run and others loop and have to be stopped explicitly. For AO use, you probably want to use looping animations so that you don't have to guess when they naturally end.

Before you can control an avatar's animation, you need to request permissions to do so using `llRequestPermissions()`. This AO script begins by requesting two permissions: `PERMISSION_TRIGGER_ANIMATION` to let us control the animation of the owning avatar and `PERMISSION_TAKE_CONTROLS` to watch the keyboard controls of the user.

When the avatar responds to the permissions request, the script makes sure the required set was granted and then "takes controls" to watch the user interface (keys), using the `llTakeControls()` function. For each control the script is watching, a corresponding `control()` event is triggered every time the key is touched. Listing 7.6 in Chapter 7 describes `llTakeControls()` and `control()` in more detail; for now this script uses this mechanism to watch the "standard" six directions (up, down, left, right, forward and backward) brought together with bitwise "or" operators, indicated by the pipe symbol (|).

This script doesn't actually use the information about which controls were pressed, but rather uses the ***activation*** of the controls to react more rapidly to avatar changes. However, avatars often do not change animations immediately on `control()` events, so the script also uses a relatively slow timer to poll for whether the avatar has changed animations.

When the script has either a `timer()` or a `control()` event, it calls the `animOverride()` function. There it asks the avatar what posture it is in using the `llGetAnimation()` function, which returns a string name of the current locomotion animation. If you want more options, use `llGetAnimationList()`, which returns a list of all animations that the avatar is currently playing.

Figure 2.7: Yoga float

If the avatar is in the posture the script wants to animate, the script will start the chosen animation with `llStartAnimation()` (if the avatar wasn't already doing that animation). If the avatar is no longer moving, the script will stop the animation using `llStopAnimation()`. The `llGetAnimation()`, `llGetAnimationList()`, `llStartAnimation()`, and `llStopAnimation()` functions are defined in Table 2.2.

TABLE 2.2: ANIMATION OVERRIDE FUNCTIONS

Function	Behavior
`string llGetAnimation(key avatarId)`	Returns the name of the locomotion animation posture the avatar is using.
`list llGetAnimationList(key avatarId)`	Returns a list of keys of all animations the avatar is currently running.
`llStartAnimation(string animation)`	Starts playing the animation on the avatar when permission `PERMISSION_TRIGGER_ANIMATION` has been granted.
`llStopAnimation(string animation)`	Stops playing the named animation.

AOs are complicated mainly because they need to get a number of permissions and controls to do their job. You could write an AO that associates an animation with each posture, or even a set of animations with each posture. Most of the AO scripts you will find in-world are able to cycle between multiple animations, even randomly, within the same posture to make the avatar look more natural. Add in a user interface with a HUD, and you have a complete AO solution for your avatar. See, for instance, the ZHAO-II scripts that are available at no charge in-world from many places, including SYW HQ.

TYPING ANIMATOR

In your travels in-world, you may have already had the experience of chatting to a stranger and being surprised to see a keyboard (or something even more elaborate) appear as if by magic in front of the resident while they are typing, and then disappear just as quickly. Figure 2.8 shows a simple version of such a keyboard.

Figure 2.8: A basic keyboard animation.

In either case, how does this simple but effective effect work? Listing 2.11 shows a slight modification of a well-known implementation, attributed to Max Case (`www.maxcase.info`). It's simple, tight code but quite effective in carrying the illusion forward. A timer elapses every half second and tests whether the resident is typing. It's a narrow enough time frame to determine whether the keyboard user is typing, but not so narrow as to affect sim performance. Humans don't type at such a blindingly fast rate that a tiny bit of lag in initiating the animation at the onset of typing or halting it later will be noticed by others in the vicinity.

Listing 2.11: Max Case's Simple Keyboard Script

```
//Derived from Max Case's Free Keyboard Script 1.0 04/18/2005
key     owner;
integer status;
key kbtextureYellow = "eef62334-e85e-b8f5-0717-19056bdbafde";
initialize() {
        llSetTexture(kbtextureYellow, 2);
        llSetTextureAnim(ANIM_ON | LOOP, // mode
                         2,              // side
                         4, 1,           // rows, columns
                         0, 0,           // start and end frame
                         4);             // frame rate
        llSetAlpha(0.0, ALL_SIDES); // make completely transparent
        llSetTimerEvent(.5);
        owner = llGetOwner();
        status = 0;
}
```


```
default {
    state_entry() {
        if (llGetOwner() != owner) {
            initialize();
        }
    }
    on_rez(integer total_number) {
        if(llGetOwner() != owner){
            initialize();
        }
    }
    timer() {
        integer temp = llGetAgentInfo(owner) & AGENT_TYPING;
        if (temp != status) { //status changed since last checked?
            llSetAlpha(!status, 2);  //flip the status only on face 2
            status = temp;  //save the current status.
        }
    }
}
```

The cleverness in the script and the illusion of animation comes from using a texture that is actually multiple rows and columns with small pictures in them, similar to multiple frames in a little movie. Figure 2.9 shows an object that has a one-row-by-four-column keyboard texture applied to it. Each frame has a slightly different set of keys that are brighter than the rest, thus making the keyboard appear to move. The number of rows and columns become arguments to the `llSetTextureAnim()` function. The animation moves through the animation cells one by one, like frames through a movie. The function is described in detail in Chapter 9.

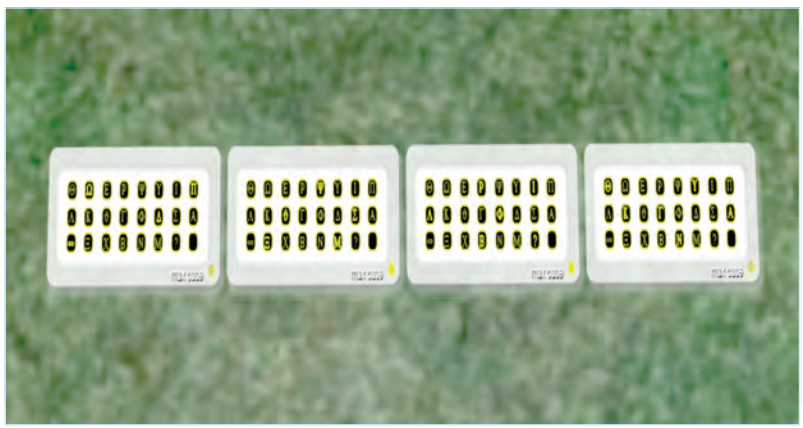


Figure 2.9: A texture containing four frames, laid out using one row of four columns. Different keys are highlighted in each frame.

DEFINITION

```
llSetTextureAnim(integer mode, integer side,
                 integer x_frames, integer y_frames,
                 float start, float length, float rate)
```

Animates the texture on one side of a prim. Only one animation can be run per prim, although all sides of a prim can be animated at the same time by setting side to `ALL_SIDES`. The "Texture Animation" section of Chapter 9 provides more details.

Note that the animation is ***always*** running—it's simply hidden because the keyboard is initially transparent from the call to `llSetAlpha(0.0, ALL_SIDES)`.

When the `timer()` event fires, it examines the avatar's state with

```
integer temp = llGetAgentInfo(owner) & AGENT_TYPING;
```

`llGetAgentInfo()` returns a bit pattern of 13 status items, including `AGENT_TYPING`, listed in Table 2.3. (If avatars are flying, both `AGENT_IN_AIR` and `AGENT_FLYING` will be true. Agents can be `AGENT_IN_AIR` but not `AGENT_FLYING` when they are jumping, falling, or kept away from the ground by means of a script.)

TABLE 2.3: CONSTANTS THAT DESCRIBE AGENT (AVATAR) STATUS

Constant	Value	Agent Status
AGENT_FLYING	0x0001	Is flying
AGENT_ATTACHMENTS	0x0002	Has attachments
AGENT_SCRIPTED	0x0004	Has scripted attachments
AGENT_MOUSELOOK	0x0008	Is in mouselook
AGENT_SITTING	0x0010	Is sitting
AGENT_ON_OBJECT	0x0020	Is sitting on an object
AGENT_AWAY	0x0040	Is in "away" mode
AGENT_WALKING	0x0080	Is walking
AGENT_IN_AIR	0x0100	Is in the air (hovering)
AGENT_TYPING	0x0200	Is typing
AGENT_CROUCHING	0x0400	Is crouching
AGENT_BUSY	0x0800	Is in "busy" mode
AGENT_ALWAYS_RUN	0x1000	Has Always Run enabled

DEFINITION

`integer llGetAgentInfo(key id)`

Returns a bitfield of information about the agent.

`id` — The agent's UUID.

Agent status can be tested using a bitwise "and" operator (&), as shown in the following snippet:

```
integer agentStatus = llGetAgentInfo( llGetOwner() );
if (agentStatus & AGENT_FLYING) {
    llOwnerSay("Agent is flying");
}
```

For this typing script, usually the ***`temp`*** variable will be `FALSE` because the avatar is not typing. That may seem like a waste of resources, but the test is short and yields a discrete result. When the result is `TRUE` the script changes the alpha (transparency) of the keyboard to make it visible.

CONTROLLING YOUR ATTACHMENTS

This chapter is full of neat toys...that could get extremely annoying in the wrong situations. Therefore, most good-quality builds provide a way to turn the effects on and off. One way to do this is with a simple HUD. HUDs are scripted objects that attach to "HUD points"—essentially the user's screen—instead of points on the avatar that are visible to others, as shown in Figure 2.10.

Figure 2.10: A floating avatar with a green HUD button showing that the `"yoga_float"` animation is enabled.

As you learned earlier, attachments are any objects your avatar wears, whether on the body or as a HUD and animation overrides (AOs) are sequences of internal commands that change the way your avatar moves. When you run an AO, the script asks your avatar permission to trigger the animation. While AOs do not need to be placed in attachments, they often are. In this case, the permissions are granted implicitly.

If you want to be able to control details of the attachment—for example to turn on the face light, to allow friends to wind up your key, or to float instead of walk—you can either create a chat interface for commands (as shown in Chapter 3) or create a HUD.

If you want to control an object that is not part of the HUD, you will probably want to add communication between the controller on the HUD (invisible to others) and the object you want to wear or otherwise have visible in the world. Chapter 3 talks about how to make this communication happen.

Listing 2.12 shows a modification to the WalkAO script from Listing 2.10. To make it work, put it in a box that you've sized to <0.2, 0.2, 0.2>. Instead of "wearing" it, attach it to your HUD at the bottom-left point. You'll see it on the lower-left corner of your ***SL*** screen, just above the communication buttons—other users will not be able to see it. It turns red when not enabled, and green when enabled. The bowling project on the SYW website demonstrates another example of a HUD.

Listing 2.12: WalkAO HUD—Adding a UI to Your AO

```
// This script is exactly the same as Listing 2.10 WalkAO except for
colorhud() { // new user-defined function
    if (gEnabled) {
        llSetColor(<0,1,0>, ALL_SIDES);
    } else {
        llSetColor(<1,0,0>, ALL_SIDES);
    }
}

default {
    // same on_rez, state_entry, timer and control event handlers
    attach( key k ) {
        if ( k != NULL_KEY ) {
            llRequestPermissions(llGetOwner(),
                PERMISSION_TRIGGER_ANIMATION|PERMISSION_TAKE_CONTROLS);
            colorhud();  // NEW LINE
        }
    }
    touch_end(integer count) {
        gEnabled = !gEnabled;
        colorhud();
        animOverride();
    }
}
```

This chapter has dealt mostly with getting avatars to look and act interesting by themselves. You can combine many of these techniques—for instance, mix the seating script with an animation override to have the avatar sit in a particular pose. We will revisit a few of these ideas later on—seating turns into vehicles in Chapter 7, animation overrides are the basis for coordinated animations in the dance project on the SYW website, and so on. The next chapter switches gears and looks into communications: how avatars and objects talk to each other (and to themselves!).