```
// Detailed script for beginners to set vehicle type, camera position, vehicle flags, ...  //Basic
Motorcycle Script
//
// by Cory
// commented by Ben
// Last edited by Nada Epoch on 01-01-2004 at 08:51 PM
//
// Retrieved from Free SL Scripts on www.gendersquare.org/sl
//
//The new vehicle action allows us to make any physical object in Second
//Life a vehicle. This script is a good example of a
// very basic vehicle that is done very well.
default{
   //There are several things that we need to do to define vehicle,     // and how the user
interacts with it.  It makes sense to
   // do this right away, in state_entry.
   state_entry()
   {
      llPassCollisions(TRUE);
      //We can change the text in the pie menu to more accurately
      // reflect the situation.  The default text is "Sit" but in
      // some instances we want you to know you can drive or ride a
      // vehicle by sitting on it. The llSetSitText function will
      // do this.
      llSetSitText("Ride");
      //Since you want this to be ridden, we need to make sure that        // the avatar "rides" it in
a acceptable position                                                                                    //
and the camera allows the driver to see what is going on.
      //
      //llSitTarget is a new function that lets us define how an avatar will orient itself when sitting.
      // The vector is the offset that your avatar's center will be
      // from the parent object's center.  The
      // rotation is based off the positive x axis of the parent. For
      // this motorcycle, we need you to sit in a way
      // that looks right with the motorcycle sit animation, so we
      // have your avatar sit slightly offset from the seat.
      llSitTarget(<0.6, 0.03, 0.20>, ZERO_ROTATION);
      //To set the camera, we need to set where the camera is, and        // what it is looking at.
 By default, it will                                                                                  // be
looking at your avatar's torso, from a position above and
      // behind. It will also be free to rotate around your
      // avatar when "turning."
      //
      //For the motorcycle, we are going to set the camera to be
      // behind the cycle, looking at a point in front of it.
      // Due to the orientation of the parent object, this will appear to be looking down on the
```

avatar.
llSetCameraEyeOffset(<-5.0, -0.00, 2.0>);
llSetCameraAtOffset(<3.0, 0.0, 2.0>);

//To make an object a vehicle, we need to define it as a // vehicle. This is done by assigning it a vehicle type. // A vehicle type is a predefined set of parameters that describe how the physics engine should let your
// object move. If the type is set to VEHICLE_TYPE_NONE it will no longer be a vehicle.
//
//The motorcycle uses the car type on the assumption that this
// will be the closest to how a motorcycle should work.
// Any type could be used, and all the parameters redefined later.
llSetVehicleType(VEHICLE_TYPE_CAR);

//While the type defines all the parameters, a motorcycle is // not a car, so we need to change some parameters
// to make it behave correctly.
//The vehicle flags let us set specific behaviors for a vehicle // that would not be covered by the more general
// parameters. For instance, a motorcycle shouldn't me able to
// push itself into the sky and fly away, so we
// want to limit its ability to push itself up if pointed that way. There are several flags that help when
// making various vehicles.
llSetVehicleFlags(VEHICLE_FLAG_NO_DEFLECTION_UP | VEHICLE_FLAG_LIMIT_ROLL_ONLY | VEHICLE_FLAG_LIMIT_MOTOR_UP);
//To redefine parameters, we use the function // llSetVehicleHippoParam where Hippo is the variable type of the // parameter (float, vector, or rotation).
//
//Most parameters come in pairs, and efficiency and a timescale. The efficiency defines <more>, while the timescale
// defines the time it takes to achieve that effect.
//
//In a virtual world, a motorcycle is a motorcycle because it looks and moves like a motorcycle. The look is
// up to the artist who creates the model. We get to define
// how it moves. The most basic properties of movement
// can be thought of as the angular deflection (points in the
// way it moves) and the linear deflection (moves in the
// way it points). A dart would have a high angular deflection, and a low linear deflection. A motorcycle has
// a low linear deflection and a high linear deflection, it goes where the wheels send it. The timescales for these
// behaviors are kept pretty short.

```
        llSetVehicleFloatParam(VEHICLE_ANGULAR_DEFLECTION_EFFICIENCY, 0.2);
        llSetVehicleFloatParam(VEHICLE_LINEAR_DEFLECTION_EFFICIENCY, 0.80);
        llSetVehicleFloatParam(VEHICLE_ANGULAR_DEFLECTION_TIMESCALE, 0.10);
        llSetVehicleFloatParam(VEHICLE_LINEAR_DEFLECTION_TIMESCALE, 0.10);
       //A bobsled could get by without anything making it go or turn        // except for a icey hill.
A motorcycle, however, has
     // a motor and can be steered.  In LSL, these are linear and
       // angular motors.  The linear motor is a push, the angular
       // motor is a twist.  We apply these motors when we use the
       // controls, but there is some set up to do.  The motor
       // timescale controls how long it takes to get the full effect
       // of the motor, basically acceleration. The motor decay
       // timescale defines how long the motor stays at that strength
       // - how slowly you let off the gas pedal.
        llSetVehicleFloatParam(VEHICLE_LINEAR_MOTOR_TIMESCALE, 1.0);
        llSetVehicleFloatParam(VEHICLE_LINEAR_MOTOR_DECAY_TIMESCALE, 0.2);
        llSetVehicleFloatParam(VEHICLE_ANGULAR_MOTOR_TIMESCALE, 0.1);
        llSetVehicleFloatParam(VEHICLE_ANGULAR_MOTOR_DECAY_TIMESCALE, 0.5);
       //Real world vehicles are limited in velocity and slow to a        // stop due to friction.  While
a vehicle that continues                                                                           //
moving forever is kinda neat, it is hard to control, and not
       // very realistic.  We can define linear and angular
       // friction for a vehicle, how quickly you will slow down while
       // moving or rotating.
       //
       //A motorcycle moves easily along the line defined by the
       // wheels, and not as easily against the wheels. A motorcycle
       // falling out of the air shouldn't feel very much friction at
       // all.  For the most part, our angular frictions don't
       // matter, as they are handled by the vertical attractor. The
       // one component that is not handled by the vertical
       // attractor is the rotation around the z axis, so we give it
       // some friction to make sure we don't spin forever.
        llSetVehicleVectorParam(VEHICLE_LINEAR_FRICTION_TIMESCALE, <10.0, 0.5,
1000.0> );
        llSetVehicleVectorParam(VEHICLE_ANGULAR_FRICTION_TIMESCALE, <10.0, 10.0,
0.5> );
       //We are using a couple of tricks to make the motorcycle look        // like a real motorcycle.
 We use an animated texture to
       // spin the wheels.  The actual object can not rely on the real
       // world physics that lets a motorcycle stay upright.
       // We use the vertical attractor parameter to make the object
       // try to stay upright. The vertical attractor also allows
       // us to make the vehicle bank, or lean into turns.
       //
       //The vertical attraction efficiency is slightly misnamed, as
```

```
        // it should be "coefficient." Basically, it controls
        // if we critically damp to vertical, or "wobble" more. It also
        // has a secondary effect that it will limit the roll
        // of the vehicle. The timescale will control how fast we go
        // back to vertical, and
        // thus how strong the vertical attractor is.
        //
        //We want people to be able to lean into turns, not fall down,
        // and not wobble to much while coming back up.
        // A vertical attraction efficiency of .5 is nicely in the middle, and it won't wobble to badly because of the
        // inherent ground friction. As shorter timescale will make it
        // hard to roll, a longer one will let us roll a lot
        // (and get a bit queasy). We will find that the controls are
        // also affected by the vertical attractor
        // as we tune the banking features, and that sometimes finding
        // good values for these numbers is more an art than a science.
         llSetVehicleFloatParam(VEHICLE_VERTICAL_ATTRACTION_EFFICIENCY, 0.50);
         llSetVehicleFloatParam(VEHICLE_VERTICAL_ATTRACTION_TIMESCALE, 0.40);
        //Banking means that if we rotate on the roll axis, we will        // also rotate on the yaw axis, meaning that our motorcycle will lean to the
        // side as we turn. Not only is this one of the things that  it look like a real motorcycle, it makes it look really cool too. The
        // higher the banking efficiency, the more "turn" for your
        // "lean".  This motorcycle is made to be pretty responsive, so we have a high
        // efficiency and a very low timescale. The banking mix lets
        // you decide if you can do the arcade style turn while not moving, or make
        // a realistic vehicle that only banks with velocity. You can
        // also input a negative banking mix value to make it bank the wrong way,
        // which might lead to some interesting vehicles.
         llSetVehicleFloatParam(VEHICLE_BANKING_EFFICIENCY, 1.0);
        llSetVehicleFloatParam(VEHICLE_BANKING_TIMESCALE, 0.01);
        llSetVehicleFloatParam(VEHICLE_BANKING_MIX, 1.0);
        //Because the motorcycle is really just skidding along the        // ground, its colliding with every bump it can find, the default behavior
        // will have us making loud noises every bump, which isn't very
        // desirable, so we can just take those out.
        llCollisionSound("", 0.0);
    }

    //A sitting avatar is treated like a extra linked primitive, which     // means that we can capture when someone sits on the                    // vehicle by looking for the changed event, specifically, a link
    // change.
    changed(integer change)
    {
```

```
        //Make sure that the change is a link, so most likely to be a
        // sitting avatar.
        if (change & CHANGED_LINK)
        {
           //The llAvatarSitOnTarget function will let us find the key
           // of an avatar that sits on an object using llSitTarget
           // which we defined in the state_entry event. We can use
           // this to make sure that only the owner can drive our vehicle.
           // We can also use this to find if the avatar is sitting, or is getting up, because both will
be a link change.
           // If the avatar is sitting down, it will return its key, otherwise it will return a null key when
it stands up.
           key agent = llAvatarOnSitTarget();
           //If sitting down.           if (agent)           {              //We don't want random punks to
come stealing our                                                                // motorcycle! The
simple solution is to unsit them,
              // and for kicks, send um flying.
              if (agent != llGetOwner())
              {
                 llSay(0, "You aren't the owner");
                 llUnSit(agent);
                 llPushObject(agent, <0,0,100>, ZERO_VECTOR, FALSE);
              }
              // If you are the owner, lets ride!
              else
              {
                 //The vehicle works with the physics engine, so in
                 // order for a object to act like a vehicle, it must first be
                 // set physical.
                 llSetStatus(STATUS_PHYSICS, TRUE);
                 //There is an assumption that if you are going to
                 // choose to sit on a vehicle, you are implicitly giving
                 // permission to let that vehicle act on your controls, and to set your permissions, so
the end user
                 // is no longer asked for permission.  However, you
                 // still need to request these permissions, so all the
                 // paperwork is filed.
                 llRequestPermissions(agent, PERMISSION_TRIGGER_ANIMATION |
PERMISSION_TAKE_CONTROLS);
                 //We will play a little "startup" sound.
                 llPlaySound("SUZ_start (2).wav", 0.7);
                 // All the messageLinked calls are communicating
                 // with other scripts on the bike.  There is a script that controls
                 // particle systems, and one that controls sounds.
                 // This way we can make a simple "motorcycle" script that is modular
                 // and you can put in your own sounds/particles,
```

```
                // and still use the same base script.
                llMessageLinked(LINK_SET, 0, "get_on", "");
            }
        }
        //The null key has been returned, so no one is driving anymore.
        else
        {
            //Clean up everything.  Set things nonphysical so they
            // don't slow down the simulator.  Release controls so the
            // avatar move, and stop forcing the animations.
            llSetStatus(STATUS_PHYSICS, FALSE);
            llReleaseControls();
            llStopAnimation("motorcycle_sit");
            // Here we let the other scripts know the cycle is done.
            llMessageLinked(LINK_SET, 0, "idle", "");
        }
    }
  }
  //Because we still need to request permissions, the run_time_permissions event still occurs,
and is the perfect     // place to start     // the sitting animation and take controls.
 run_time_permissions(integer perm)
  {
    if (perm)
    {
      llStartAnimation("motorcycle_sit");
      llTakeControls(CONTROL_FWD | CONTROL_BACK | CONTROL_RIGHT |
CONTROL_LEFT | CONTROL_ROT_RIGHT | CONTROL_ROT_LEFT, TRUE, FALSE);
    }
  }
  //If we want to drive this motorcycle, we need to use the controls.    control(key id, integer
level, integer edge)                                                              {
 //We will apply motors according to what control is used.  For
    // forward and back, a linear motor is applied with a vector
    // parameter.
    vector angular_motor;
    if(level & CONTROL_FWD)       {          //The Maximum linear motor direction is 50, and
will try to                                           // get us up to 50 m/s - things like
friction and the                                                  // motor decay timescale
can limit that.
llSetVehicleVectorParam(VEHICLE_LINEAR_MOTOR_DIRECTION, <50,0,0> );
    }       if(level & CONTROL_BACK)       {
llSetVehicleVectorParam(VEHICLE_LINEAR_MOTOR_DIRECTION, <-20,0,0> );
    }
    if(level & (CONTROL_RIGHT|CONTROL_ROT_RIGHT))
    {
      //The Maximum angular motor direction is 4Pi radians/second.
```

```
            //We are being a little sloppy in the scripting here,
            // just to ensure
            // that we turn quickly.
            angular_motor.x += PI*4;
            angular_motor.z -= PI*4;
        }
        if(level & (CONTROL_LEFT|CONTROL_ROT_LEFT))
        {
            angular_motor.x -= PI*4;
            angular_motor.z += PI*4;
        }
        if(level & (CONTROL_UP))
        {
            angular_motor.y -= 50;
        }
        if((edge & CONTROL_FWD) && (level & CONTROL_FWD))        {            // We have a few
message links to communicate to the other
            // scritps when we start to accelerate and let off the gas.
            llMessageLinked(LINK_SET, 0, "burst", "");
        }
        if((edge & CONTROL_FWD) && !(level & CONTROL_FWD))
        {
            llMessageLinked(LINK_SET, 0, "stop", "");
        }
        //The angular motor is set last, just incase there is a sum of        // the right and left
controls (you have to swing the handlebars back to center)
       llSetVehicleVectorParam(VEHICLE_ANGULAR_MOTOR_DIRECTION,angular_motor);
    }
}
```