

Skeletal Animation in SketchUp

Animating an object along a path isn't difficult, but animating an assembly of moving objects is considerably more involved—especially when the objects are interconnected. But you don't have to figure everything out for yourself. Skeletal animation provides a systematic procedure for coordinating this animation, and it has been used in numerous game engines to animate complex structures.

This document presents a system of skeletal animation specific to SketchUp. It's not as full-featured as a professional offering, but it should be sufficient for a wide variety of animation tasks. In particular, this presentation focuses on designing and animating a robot. As shown in Figure 1, this is a dancing robot, and you can see it move by executing `robot.rb`.

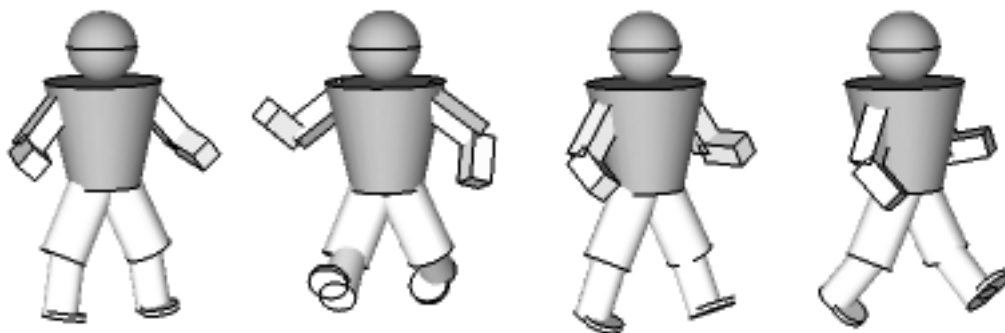


Figure 1: A Dancing Robot ($t=0, 4, 8, 12$ sec)

Before we create this robot in code, there's a lot of theoretical groundwork to cover. We'll look at how skeletal hierarchies are structured and see how the parts move over time. There's a lot of new terminology, but don't lose heart if the subject doesn't make sense at first—this document takes a gradual approach to presenting skeletal animation, and if the theory is hard to grasp, the code should make things clear.

Skeletal Structure: Bones, Joints, and Skeletons

Think about your right arm. You'll need at least two shapes to model it in SketchUp: one to represent the upper arm and one to represent the lower arm, commonly called the forearm. To animate this in a realistic fashion, the model needs to satisfy three requirements:

- The two shapes must always be connected.
- When the upper arm moves, the forearm must always follow.
- The forearm can move only by rotating around its connection to the upper arm.

You could further constrain the motion, but writing animation code that meets these criteria is a complex process. The complexity grows larger when you add more shapes, such as a wrist connected to the forearm, a hand connected to the wrist, and fingers connected to the hand.

To reduce the difficulty, this discussion presents skeletal animation—a system for creating and animating connected structures like this hypothetical arm. The parts that make up this system are defined using three anatomical terms:

1. Bone - A bone is a shape or set of vertices that moves in a unified manner. When a bone moves, each of its vertices move in the same way.
2. Joint - A joint connects one bone to another, and defines the point of rotation for the connected bone.
3. Skeleton - A skeleton is a hierarchy of bones and joints.

Going back to the arm model, it should be clear that when the upper arm moves, the forearm follows. The reverse isn't true—the forearm can rotate while the upper arm remains still. This relationship between the independent bone (upper arm) and the dependent bone (forearm) is referred to as a *parent-child* relationship.

The set of parent and child bones in a structure form a tree-like hierarchy called the skeleton. Figure 2 displays the tree structure of the robot depicted in Figure 1. Each child bone has one parent, but a parent bone may have multiple children. The Torso is the top-most parent, and it has five direct children: the Head, Upper Arm (Right), Upper Arm (Left), Upper Leg (Right), and Upper Leg (Left). The Torso has no parent, and for this reason, it's called the *root* bone of the skeleton. When the root bone moves, every other bone in the skeleton moves as well.

The point where a parent bone and child bone meet is called a *joint*. In Figure 2, each joint is represented by a black dot. For example, the right upper leg and right lower leg meet at a joint and the head and torso meet at a joint. Each child bone moves only by rotating around its joint.

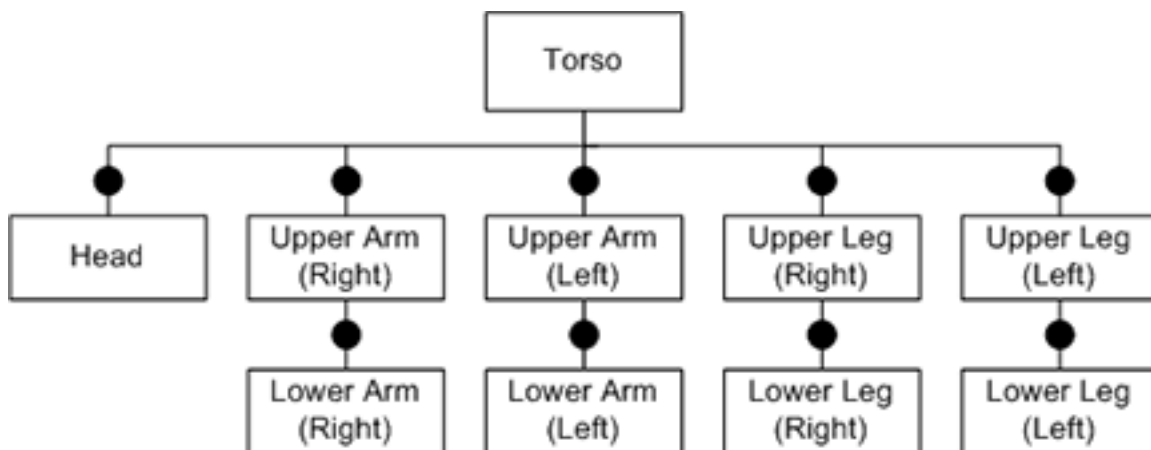


Figure 2: Bones and Joints in the Robot Skeleton

The Skel Module

Now that you understand the basics of skeletal structure, let's look the `Skel` module defined in the `skel_animation.rb` script. This defines two classes: `Bone` and `Skeleton`. Each `Skeleton` object contains a hierarchy of `Bones`, and each moving part in the model needs a separate `Bone` object. Each `Bone` stores the point around which it rotates. Therefore, there is no `Joint` class in the module.

To create a skeletal structure in code, you need to call three important methods:

1. `Skeleton.new` - Create a new `Skeleton` with a given name
2. `Skeleton.add_root` - Create a `Bone` to represent the root of the `Skeleton`
3. `Bone.add_child` - Create a `Bone` to be added as a child

The `Skeleton.add_root` and `Bone.add_child` methods create and return new `Bone` objects. Like `Bone.new`, both methods accept three arguments (one required, two optional):

- `comp` (required) - the `ComponentDefinition` that defines the `Bone`'s appearance
- `trans` (optional) - the `Transformation` needed to place the `Bone` in the model
- `joint` (optional) - the point around which the `Bone` rotates (`[0, 0, 0]` by default)

The first argument is the most important, so let's review how components work. SketchUp provides two component classes: `ComponentDefinition` and `ComponentInstance`. The `ComponentDefinition` sets the component's basic information, such as its shape, color, and texture. The `ComponentInstance` represents an instance of the component inside the model. A `ComponentInstance` is added to a design with code similar to the following:

```
ents = Sketchup.active_model.entities
comp_inst = ents.add_instance comp_def, t
```

The second command creates a `ComponentInstance` from the `ComponentDefinition` and applies the `Transformation` named `t` to it. This `Transformation` defines how and where the instance should be placed in the model. This is the same purpose served by the second argument in the `Bone` constructor. If the second argument is omitted, the identity transformation will be used and the instance will be placed at the origin. It's very common, however, to position child `Bones` using the parent's `Transformation`.

Each `Bone` object stores the location of its joint, and even the root `Bone` has a joint that it can rotate around. The third argument of the `Bone` constructor defines an initial location of this joint. If this is omitted, the origin is used. Note that this location corresponds to the pre-`Transformation` state of the `Bone`—when the instance is created, the `Transformation` is applied to both the `Bone` and its joint.

Let's look at a simple example. Figure 1 presents the two bones that make up the robot's arm. The upper arm is the root and the forearm is its child. The following code shows how the `Skeleton` and two `Bones` can be created:

```
# Create two Transformations
upper_t = Geom::Transformation.translation [0,0,2]
lower_t = Geom::Transformation.rotation [0,0,0], [0,1,0], 17.degrees

# Create the Skeleton and two Bones
arm_skeleton = Skel::Skeleton.new "Arm"
upper_bone = arm_skeleton.set_root upper_comp, upper_t, [0,0,3]
lower_bone = upper_bone.add_bone lower_comp, upper_t * lower_t
```

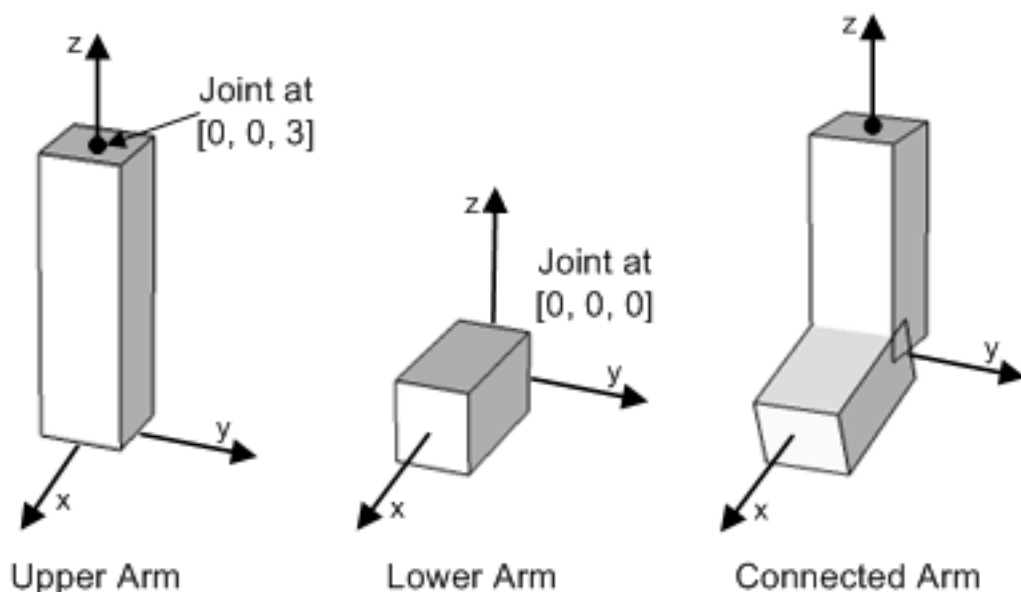


Figure 3: Constructing the Robot's Arm

In this case, `upper_bone` is translated by `[0, 0, 2]` when it's placed in the model. Its joint, originally placed at `[0, 0, 3]`, is also translated by `[0, 0, 2]` to `[0, 0, 5]`. When `lower_bone` is placed in the model, it is transformed by `upper_t * lower_t`. This means it is initially rotated 17 degrees around the y-axis (`lower_t`) and then translated by `[0, 0, 2]` (`upper_t`).

Listing 1 creates and positions the Bones depicted in Figure 3. The Bones in this example don't move yet, but we'll add animation shortly.

Listing 1: `arm_static.rb`

```
require "skel_animation.rb"

model = Sketchup.active_model

# Create component for upper arm
upper_comp = model.definitions.add "Upper Arm"
```

```

face = upper_comp.entities.add_face [-0.5,-0.5,0],
    [0.5,-0.5,0], [0.5,0.5,0], [-0.5,0.5,0]
face.pushpull -3

# Create component for lower arm
lower_comp = model.definitions.add "Lower Arm"
face = lower_comp.entities.add_face [0,-0.5,-0.5],
    [2,-0.5,-0.5], [2,0.5,-0.5], [0,0.5,-0.5]
face.pushpull 1

# Create two Transformations
upper_t = Geom::Transformation.translation [0,0,2]
lower_t = Geom::Transformation.rotation [0,0,0], [0,1,0], 17.degrees

# Create the Skeleton and two Bones
arm_skeleton = Skel::Skeleton.new "Arm"
upper_bone = arm_skeleton.set_root upper_comp, upper_t, [0,0,3]
lower_bone = upper_bone.add_bone lower_comp, upper_t * lower_t
arm_skeleton.animate

```

The first part of this code creates the `ComponentDefinitions` for the upper arm and forearm. In both cases, the `ComponentDefinition` contains a `Face` which is extruded into a box-like figure.

The second part creates two `Transformations`. These are used to set the initial position of each `Bone` within the model. The first `Transformation` translates by `[0, 0, 2]` and the second rotates around the `y`-axis by `17°`.

The last four lines initialize the `Skeleton` and `Bones`. The `Skeleton` is created with `Skeleton.new`, which creates a top-level `ComponentDefinition` and adds it to the current model. Then, as each new `Bone` is created, the `Bone's ComponentDefinition` is added as a sub-component.

The `Skeleton.set_root` method makes `upper_bone` the root `Bone` of the model. Afterward, `upper_bone.add_bone` makes `lower_bone` a child. Both methods accept the same type of parameters, which set the initial position of the joint and the initial transformation.

The last line of code calls the `animate` method of the `Skeleton` class. This method generally performs three tasks:

1. Creates a `ComponentInstance` for each `Bone` and applies the `Bone`'s initial `Transformation`.
2. Determines the total time for the animation.
3. Starts the timer and calls `animate_kernel`. This animates the root `Bone` and recursively animates each of its children.

In Listing 1, the `animate` method only performs the first task, placing the two `Bone` instances in the design window. To actually animate these `Bones`, you need to configure keyframes. This important topic is discussed next.

Skeletal Animation: Keyframes and Tracks

The last section explained how `Bones` function in skeletal animation and how to set their initial position. Now we're going to examine how to make them move. The central method in this process is `add_keyframe` in the `Bone` class. But before we look at it closely, it's important to understand what a keyframe is.

The concept of a keyframe harkens back to the time of hand-drawn animation. Once the story was laid out, a primary animator would draw a few important frames. These frames delineate the plot, setting, and essential characters, and each frame might be separated by seconds or even tens of seconds. But no one wants to see frames updated every few seconds, so secondary animators are called in to draw in-between frames. The primary animator's scenes are called *keyframes*, and the secondary drawings are interpolations between the keyframes.

The skeletal animation process is similar. After you've created a `Bone`, you define how you want it to move and how much time is available for each movement. For `Bones`, this movement must be a rotation around a joint. The `Skeleton` can be translated with `Skeleton.add_keyframe`, but the transformations defined by `Bone.add_keyframe` can only perform rotations. Specifically, this method accepts three arguments:

- `axis` - the axis of rotation (according to the `Bone`'s local coordinate system)
- `angle` - the angle of rotation (in radians)
- `time` - the time in seconds for the rotation to be performed (cumulative)

You *do not* specify the origin of the rotation. A Bone always rotates around its joint, whose coordinates are set when the Bone is created. Further, each Bone has its own local coordinate system, and the keyframe's axis is relative to this local coordinate system. The coordinate axes of a child Bone aren't necessarily similar to the axes of its parent. In the `upper_bone-lower_bone` skeleton created earlier, the coordinate system of `lower_bone` is slightly different than the coordinate system of `upper_bone`.

Keyframes are crucial in skeletal animation, so let's look at an example. The following code tells `lower_bone` to rotate 30° around the x-axis for 5 seconds and then -30° for 5 seconds.

```
lower_bone.add_keyframe [1, 0, 0], 30.degrees, 5
lower_bone.add_keyframe [1, 0, 0], -30.degrees, 10
```

The last argument is *cumulative*—it represents the time in seconds since the start of the animation. In this case, `lower_bone` will rotate for a total of ten seconds: five seconds around the +x axis and five seconds around the -x axis. After ten seconds, it will stop moving.

The code in Listing 2 is almost exactly similar to that in Listing 1. The difference is that `add_keyframe` is called to animate both Bones.

Listing 2: arm_motion.rb

```
require "skel_animation.rb"

model = Sketchup.active_model

# Create component for upper arm
upper_comp = model.definitions.add "Upper Arm"
face = upper_comp.entities.add_face [-0.5,-0.5,0],
  [0.5,-0.5,0], [0.5,0.5,0], [-0.5,0.5,0]
face.pushpull -3

# Create component for lower arm
lower_comp = model.definitions.add "Lower Arm"
face = lower_comp.entities.add_face [0,-0.5,-0.5],
```



```

    [2,-0.5,-0.5], [2,0.5,-0.5], [0,0.5,-0.5]
face.pushpull 1

# Create two Transformations
upper_t = Geom::Transformation.translation [0,0,2]
lower_t = Geom::Transformation.rotation [0,0,0], [0,1,0], 17.degrees

# Create the Skeleton and two Bones
arm_skeleton = Skel::Skeleton.new "Arm"
upper_bone = arm_skeleton.set_root upper_comp, upper_t, [0,0,3]
lower_bone = upper_bone.add_bone lower_comp, upper_t * lower_t

# Configure animation of upper arm
upper_bone.add_keyframe [1, 0, 0], -20.degrees, 5
upper_bone.add_keyframe [1, 0, 0], 20.degrees, 10

# Configure animation of lower arm
lower_bone.add_keyframe [0, 1, 0], -30.degrees, 5
lower_bone.add_keyframe [0, 1, 0], 30.degrees, 10

# Display the skeleton
arm_skeleton.animate

```

The code in this script can be divided into three main parts. The first part creates a `ComponentDefinition` and a `Transformation` for each `Bone`. The second part creates the `Skeleton`, sets the root `Bone`, and adds child `Bones`. The final part calls `add_keyframe` to configure the model's animation.

The `add_keyframe` commands configure the upper arm to rotate around the x-axis while the lower arm rotates around the y-axis. But because the lower arm is the child of the upper arm, it performs *both* rotations. This is why the two `Bones` remain connected during the animation.

Let's look briefly at the `animate_kernel` method in the `Skeleton` class. This method recursively processes each `Bone` and its keyframes. Every interval (0.1 seconds by default), this method animates `upper_arm` and `lower_arm` using four steps:

1. Rotate `upper_arm` around its joint according to `upper_arm`'s keyframe.
2. Rotate `lower_arm`'s joint around `upper_arm`'s joint according to `upper_arm`'s keyframe.
3. Rotate `lower_arm` around `upper_arm`'s joint according to `upper_arm`'s keyframe.
4. Rotate `lower_arm` around its joint according to `lower_arm`'s keyframe.

You don't need to understand precisely how the animation is performed, but if your application fails to work properly, it helps to know how the `Bones` are rotating. Also, when you're dealing with combinations of `Transformations` like `upper_t * lower_t`, it helps to look at the `Transformation` matrix to make sure the combination is performed in proper order.

Listing 3 presents the complete script for the dancing robot. Most of this code should look familiar: the first part creates components, the second part creates the skeletal structure, and the third part configures the animation.

Listing 3: robot.rb

```
require "skel_animation.rb"

model = Sketchup.active_model

# Constants
head_radius = 1.5
body_radius = 1.5
body_height = 4.5

# Create torso component
torso_comp = model.definitions.add "Torso"
path = torso_comp.entities.add_circle [0,0,0], [0,0,1], body_radius
body = torso_comp.entities.add_face [0,0,0], [0,0,body_height],
[0,2.5,body_height], [0,1.5,0]
body.followme path

# Create head component
head_comp = model.definitions.add "Head"
```

```

path = head_comp.entities.add_circle [0,0,0], [0,0,1], body_radius
circle = head_comp.entities.add_circle [0,0,0], [0,1,0], head_radius
circle_face = head_comp.entities.add_face circle
circle_face.followme path

# Create upper arm component
upper_arm_comp = model.definitions.add "Upper Arm"
face = upper_arm_comp.entities.add_face [-0.5,-0.5,0],
    [0.5,-0.5,0], [0.5,0.5,0], [-0.5,0.5,0]
face.pushpull -3

# Create lower arm component
lower_arm_comp = model.definitions.add "Lower Arm"
face = lower_arm_comp.entities.add_face [0,-0.5,-0.5],
    [2,-0.5,-0.5], [2,0.5,-0.5], [0,0.5,-0.5]
face.pushpull 1

# Create upper leg component
upper_leg_comp = model.definitions.add "Upper Leg"
circle = upper_leg_comp.entities.add_circle [0,0,0], [0,0,1], 1
circle_face = upper_leg_comp.entities.add_face circle
circle_face.pushpull 3.5

# Create lower leg component
lower_leg_comp = model.definitions.add "Lower Leg"
circle = lower_leg_comp.entities.add_circle [0,0,0], [0,0,1], 0.75
circle_face = lower_leg_comp.entities.add_face circle
circle_face.pushpull 3
foot = lower_leg_comp.entities.add_circle [0.75,0,-2.75], [0,0,1], 0.8
foot_face = lower_leg_comp.entities.add_face foot
foot_face.pushpull -0.25

# Create skeleton, root, and head bones
robot_skeleton = Skel::Skeleton.new "Robot"
torso = robot_skeleton.set_root torso_comp, [0,0,0]
head = torso.add_bone head_comp, [0, 0, body_height + head_radius]

```

```

# Create upper arm bones
upper_arm_trans_r = Geom::Transformation.translation([0.0, body_
radius+2, 1.75]) *
    Geom::Transformation.rotation([0,0,0], [1,0,0], 40.degrees)
upper_arm_trans_l = Geom::Transformation.translation([0.0, -(body_
radius+2), 1.75]) *
    Geom::Transformation.rotation([0,0,0], [1,0,0], -40.degrees)
upper_arm_r = torso.add_bone upper_arm_comp, upper_arm_trans_r, [0,0,3]
upper_arm_l = torso.add_bone upper_arm_comp, upper_arm_trans_l, [0,0,3]

# Create lower arm bones
lower_arm_trans_r = upper_arm_trans_r *
    Geom::Transformation.rotation([0,0,0], [0, 1, 0], 17.degrees)
lower_arm_trans_l = upper_arm_trans_l *
    Geom::Transformation.rotation([0,0,0], [0, 1, 0], 17.degrees)
lower_arm_r = upper_arm_r.add_bone lower_arm_comp, lower_arm_trans_r
lower_arm_l = upper_arm_l.add_bone lower_arm_comp, lower_arm_trans_l

# Create upper leg bones
upper_leg_trans_r = Geom::Transformation.translation([0,0.5,0.25]) *
    Geom::Transformation.rotation([0,0,0], [1,0,0], 25.degrees)
upper_leg_trans_l = Geom::Transformation.translation([0,-0.5,0.25]) *
    Geom::Transformation.rotation([0,0,0], [1,0,0], -25.degrees)
upper_leg_r = torso.add_bone upper_leg_comp, upper_leg_trans_r
upper_leg_l = torso.add_bone upper_leg_comp, upper_leg_trans_l

# Create lower leg bones
lower_leg_trans_r = Geom::Transformation.rotation([0,0,0], [1,0,0],
25.degrees) *
    Geom::Transformation.translation([0,0.5,-3]) *
    Geom::Transformation.rotation([0,0,0], [1,0,0], -15.degrees)
lower_leg_trans_l = Geom::Transformation.rotation([0,0,0], [1,0,0], -25.
degrees) *
    Geom::Transformation.translation([0,-0.5,-3]) *
    Geom::Transformation.rotation([0,0,0], [1,0,0], 15.degrees)

```

```

lower_leg_r = upper_leg_r.add_bone lower_leg_comp, lower_leg_trans_r
lower_leg_l = upper_leg_l.add_bone lower_leg_comp, lower_leg_trans_l

# Animate skeleton
forward = Geom::Transformation.translation [5,0,0]
backward = Geom::Transformation.translation [-5,0,0]
robot_skeleton.add_keyframe forward, 5
robot_skeleton.add_keyframe backward, 10
robot_skeleton.add_keyframe forward, 15

# Animate torso
torso.add_keyframe [0, 0, 1], -60.degrees, 5
torso.add_keyframe [0, 0, 1], 120.degrees, 10
torso.add_keyframe [0, 0, 1], -60.degrees, 15

# Animate upper arms
upper_arm_r.add_keyframe [1, 1, 0], 30.degrees, 5
upper_arm_r.add_keyframe [1, 1, 0], -60.degrees, 10
upper_arm_r.add_keyframe [1, 1, 0], 30.degrees, 15
upper_arm_l.add_keyframe [1, 1, 0], -30.degrees, 5
upper_arm_l.add_keyframe [1, 1, 0], 60.degrees, 10
upper_arm_l.add_keyframe [1, 1, 0], -30.degrees, 15

# Animate lower arms
lower_arm_r.add_keyframe [0, 1, 0], 40.degrees, 5
lower_arm_r.add_keyframe [0, 1, 0], -80.degrees, 10
lower_arm_r.add_keyframe [0, 1, 0], 40.degrees, 15
lower_arm_l.add_keyframe [0, 1, 0], -40.degrees, 5
lower_arm_l.add_keyframe [0, 1, 0], 80.degrees, 10
lower_arm_l.add_keyframe [0, 1, 0], -40.degrees, 15

# Animate upper legs
upper_leg_r.add_keyframe [0, 1, 0], 30.degrees, 5
upper_leg_r.add_keyframe [0, 1, 0], -60.degrees, 10
upper_leg_r.add_keyframe [0, 1, 0], 30.degrees, 15
upper_leg_l.add_keyframe [0, 1, 0], -30.degrees, 5

```

```

upper_leg_l.add_keyframe [0, 1, 0], 60.degrees, 10
upper_leg_l.add_keyframe [0, 1, 0], -30.degrees, 15

# Animate lower legs
lower_leg_r.add_keyframe [0, 1, 0], 50.degrees, 5
lower_leg_r.add_keyframe [0, 1, 0], -100.degrees, 10
lower_leg_r.add_keyframe [0, 1, 0], 50.degrees, 15
lower_leg_l.add_keyframe [0, 1, 0], -50.degrees, 5
lower_leg_l.add_keyframe [0, 1, 0], 100.degrees, 10
lower_leg_l.add_keyframe [0, 1, 0], -50.degrees, 15

# Animate
robot_skeleton.animate

```

As you watch the robot dance, think about how hard it would be to keep track of its joints and rotations in global coordinates. The great benefit of skeletal animation is that you specify each bone's movement in terms of the bone's local coordinates. This is why all the calls to `Bone.add_keyframe` look simple but the overall robot's movement is very complex.

Besides the size, the significant difference between `robot.rb` and `arm_motion.rb` is that the robot application transforms the `Skeleton` as well as its `Bones`. The important lines are as follows:

```

forward = Geom::Transformation.translation [5,0,0]
backward = Geom::Transformation.translation [-5,0,0]
robot_skeleton.add_keyframe forward, 5
robot_skeleton.add_keyframe backward, 10
robot_skeleton.add_keyframe forward, 15

```

As shown above, the `Skeleton.add_keyframe` method accepts a `Transformation` object and time. This is markedly different from the `Bone.add_keyframe` method, which accepts an axis, an angle, and a time. This difference is important—you can translate, rotate, or scale the overall `Skeleton`, but the only way to move one of its `Bones` is through rotation.

If you extend the animation time far beyond fifteen seconds, the robot will move strangely. This isn't the fault of the code, but of the way SketchUp generates rotation matrices. A rotation matrix contains approximations of sine and cosine values, and the error in each approximation is propagated to succeeding rotation matrices. After enough rotations, the error will be significant enough to be noticeable. It is faster and less error-prone to rely on quaternions.