5. Animation

Many type of animation can be implemented using Java 3D's **Interpolator** classes or by subclassing **Behavior**, so they're the main topics of this chapter. As before, I'll introduce several Simple3D methods and classes that hide some of the confusing and/or boring details (e.g. the RotPosPath and TimeBehavior classes).

One kind of animation – the modification of a shape's geometry at run time – is usually not implemented using Behavior. The **Morph** class was a popular solution until it was (rather unfairly) deprecated in Java 3D 1.4. It can still be utilized of course, if you don't mind receiving `Warning` compiler messages. Another approach is to employ the **GeometryUpdater** interface, which is arguably better suited for carrying out small changes to a model.

1. Interpolators

Java 3D offers a bewildering range of Interpolator subclasses, as shown in Figure 1.

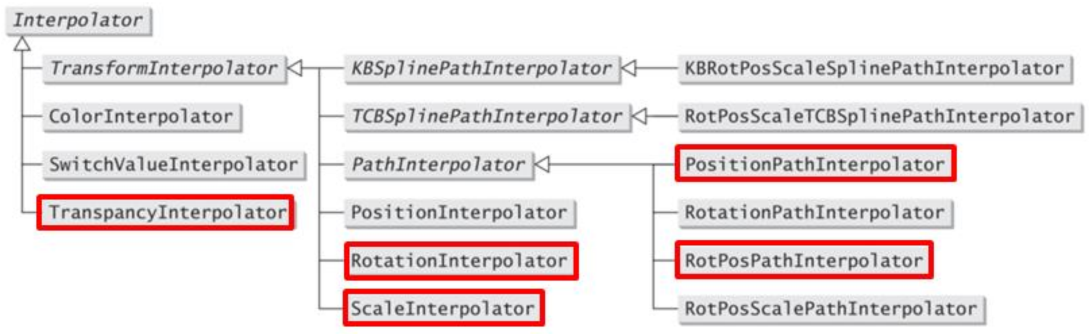

Figure 1. Subclasses of Interpolator.

In this chapter, I'll be using those outlined in red, and the most complicated (RotPosPathInterpolator) has a lot of its complexity hidden by Simple3D's RotPosPath class.

An Interpolator supports animation by generating intermediate values between two or more supplied attributes (e.g. two positions, colors, angles), with the timing of the changes from one value to the next controlled by an Alpha object. The Alpha class is capable of a wide range of timing actions, as hinted at by the number of attributes it supports, some of which are shown in Figure 2.

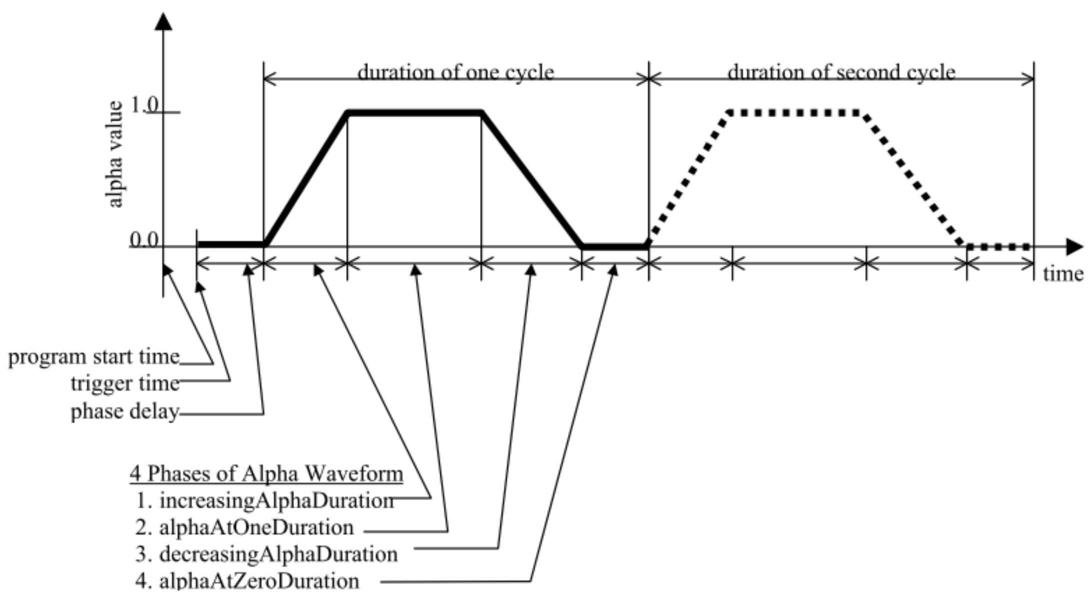

Figure 2. Alpha Attributes.

These attributes control how time is converted into values rangings between 0.0 and 1.0. The good news is that most common uses of Alpha only manipulate two of these variables:

- loopCount: the number of cycles of the timer; -1 indicates an unlimited number of loops.
- increasingAlphaDuration: the time in milliseconds for the alpha value to increase from 0.0 to 1.0.

All the other attributes have default values of 0, and so the above diagram becomes the somewhat less forbidding Figure 3.

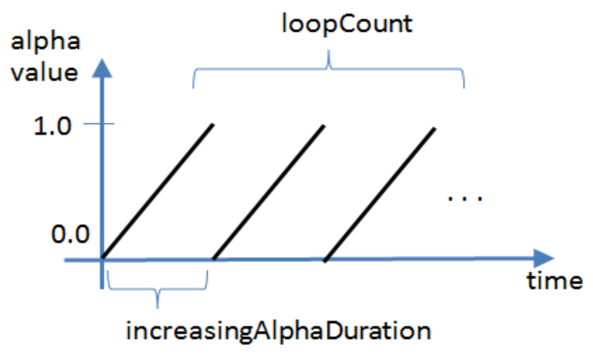

Figure 3. A Common Alpha Format.

An interpolator specifies how the changing alpha value causes its data to change. When the interpolator only manages two pieces of data, the usual mapping is to use the first datum when the alpha value is 0, then gradually change to the second datum, completing the transition by the time the alpha reaches 1.0.

Things become more fiddly when an interpolator possesses multiple data items, as in the case of the "Path" interpolators. Each value is paired with a particular alpha between 0.0 and 1.0, thereby allowing the changing Alpha object to control how the interpolation transitions through its data.

1.1. A Penguin Goes Walking

PenguinWalks.java loads a penguin model and calls one of four functions which moves it over the XZ plane, returning it to its starting point after a certain amount of time, and then repeating. Figure 4 shows three screenshots of the penguin out strolling.

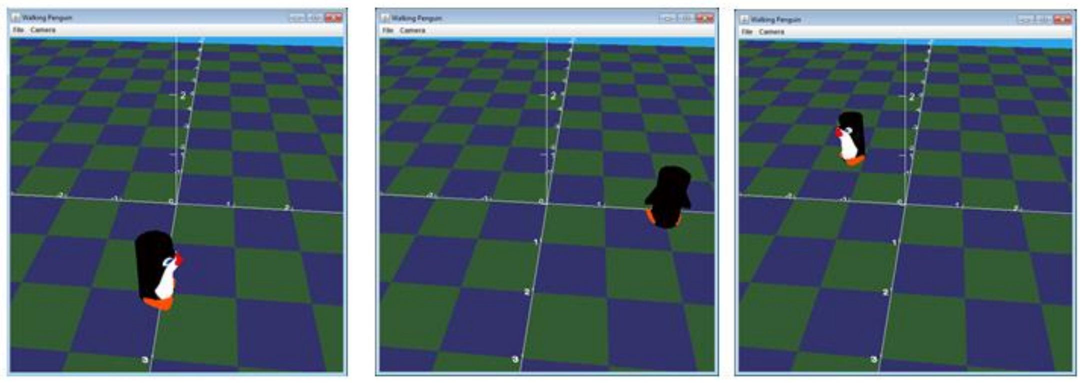

Figure 4. The Penguin's Perambulations.

As the image's suggest, the first penguin odyssey is simply a rotation around the origin at a distance of 2 units in a counterclockwise direction.

The top-level code:

```
public static void main(String[] args)
{
  Simple3D j3d = new Simple3D("Walking Penguin", true, true);
  BranchGroup penguinBG =
            Simple3D.loadModel("models/penguin/penguin.obj");

  TransformGroup rot1TG = Simple3D.rotY(90);  // face right
  rot1TG.addChild(penguinBG);

  // four ways to use an interpolator
  TransformGroup pathTG = orbitY(rot1TG, 2);                // 1
  // TransformGroup pathTG = squarePath(rot1TG, 2);         // 2
  // TransformGroup pathTG = squareRotPath(rot1TG, 2);      // 3
  // TransformGroup pathTG = pathRotateY(rot1TG, 2);        // 4

  // build the scene
  BranchGroup bg = Simple3D.createBG(pathTG);
  j3d.rootBG.addChild(bg);
}  // end of main()
```

```
private static TransformGroup orbitY(TransformGroup obj,
                                     double radius)
{
  TransformGroup posTG = Simple3D.setTranslation(0,0,radius);
  posTG.addChild(obj);
  TransformGroup rotTG = Simple3D.orbit(posTG, 6000);
  return rotTG;
}  // end of orbitY()
```

The code loads the penguin model at the origin and rotates it to face along the +x axis. orbitY() translates the penguin to (0,0,2), and starts rotating it around the origin.

Simple3D.orbit() hides the creation of an Alpha timer, a RotationInterpolator, and their interconnection:

```
public static TransformGroup orbit(Node obj, int period)
{  return orbit(obj, period, Y_AXIS);   }


public static TransformGroup orbit(Node obj, int period,
                                   int axisType)
{
  Transform3D axisT3d = new Transform3D();
                                // along the +y-axis (default)
  if (axisType == Z_AXIS)
    axisT3d.rotX( Math.PI/2);   // along +z-axis
  else if (axisType == -Z_AXIS)
    axisT3d.rotX( Math.PI/2);   // along -z-axis

  else if (axisType == X_AXIS)
    axisT3d.rotZ( -Math.PI/2);  // along +x-axis
  else if (axisType == -X_AXIS)
    axisT3d.rotZ( Math.PI/2);   // along -x-axis

  else if (axisType == -Y_AXIS)
    axisT3d.rotX( Math.PI);     // along -y-axis

  TransformGroup tg = Simple3D.createTG();
  tg.addChild(obj);

  Alpha alpha = new Alpha(-1, period);
  alpha.setStartTime( System.currentTimeMillis()); // start now

  RotationInterpolator rotator =
                  new RotationInterpolator(alpha, tg);
  rotator.setTransformAxis(axisT3d);
       // no min or max angle is specified,
       // so the rotation is one complete turn
  rotator.setSchedulingBounds(Simple3D.INFINITE_BOUNDS);

  tg.addChild(rotator);
```

```
    return tg;
  }  // end of orbit()
```

The Alpha object's loopCount is set to -1 (i.e. it repeats forever), and increasingAlphaDuration is assigned the time delay so that one cycle will take that number of milliseconds to transition from 0.0 to 1.0.

A RotationInterpolator interpolates between two angles, which aren't supplied here, meaning that the object assumes they are 0 and 360 degrees. The two arguments that *are* passed to the RotationInterpolator are an Alpha reference and a TransformGroup representing the rotation axis.

A common error when coding with interpolators is to forget to call setSchedulingBounds() which links the interpolator's execution to the location of the camera. The idea is that if the camera is a long way away from the thing being animated then the animation can be paused until the camera gets closer. The simplest thing is to set the bounds to Simple3D.INFINITE_BOUNDS which causes the interpolator to keep running no matter where the camera is positioned.

Simple3D.orbit() includes a call to Alpha.setStartTime() which ensures that the timer (and therefore the animation) only begins when execution reaches this point in the code.

1.2. Walking in a Square

The second way that the penguin can walk around is coded in squarePath() with a PositionPathInterpolator class. It utilizes a sequence of positions that are mapped to Alpha values. The nine points employed by squarePath() are illustrated in Figure 5.

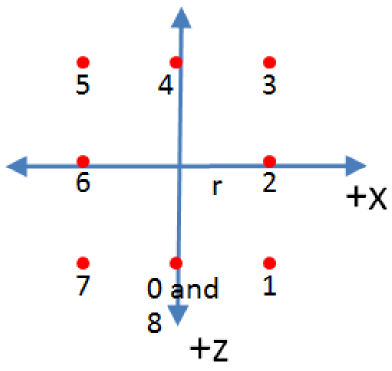

Figure 5. Journey around a Square.

The penguin starts at position 0, which is mapped to the 0.0 Alpha value, and each subsequent point is mapped to a later alpha value; Point 8 is assigned to the Alpha value 1.0, indicating the end of a single cycle.

The reuse of the start point at the end of the path ensures that the penguin is back in its starting position when the Alpha cycle begins repeating.

The code for squarePath():

```
private static TransformGroup squarePath(TransformGroup obj,
                                          double radius)
{
  TransformGroup pathTG = Simple3D.createTG();
  pathTG.addChild(obj);

  float r = (float)radius;
  Point3f[] positions = { new Point3f(0,0,r),    // point 0
                          new Point3f(r,0,r),
                          new Point3f(r,0,0),
                          new Point3f(r,0,-r),
                          new Point3f(0,0,-r),
                          new Point3f(-r,0,-r),
                          new Point3f(-r,0,0),
                          new Point3f(-r,0,r),
                          new Point3f(0,0,r)     // back at start
       };

  float[] knots = {  0.0f, 0.1f, 0.2f, 0.3f, 0.4f,
                     0.5f, 0.6f, 0.7f, 1.0f
                  };

  Alpha alpha = new Alpha(-1, 6000); // repeat every 6 secs
  alpha.setStartTime( System.currentTimeMillis()); // start now

 Transform3D axis = new Transform3D();    // + y-axis
  PositionPathInterpolator ppInterp =
     new PositionPathInterpolator(alpha, pathTG,
                                  axis, knots, positions);
  ppInterp.setSchedulingBounds(Simple3D.INFINITE_BOUNDS);

  pathTG.addChild(ppInterp);
  return pathTG;
}  // end of squarePath()
```

The sequence of Alpha values mapped to the positions are called "knots". A knot can be any quantity between 0.0 and 1.0, but the sequence must be in ascending order.

Figure 6 shows several screenshots of this penguin's journey.

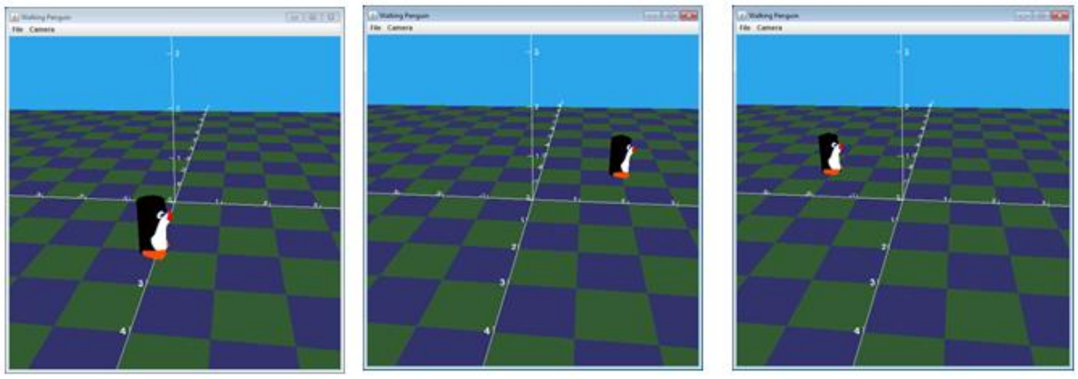

Figure 6. The Penguin's Square Dance (at Points 0, 3 and 5).

The most obvious problem with this approach is that the penguin isn't rotated as it moves around the square. For instance, this means that as it moves from points 3 to 5 in Figure 6, the penguin is facing the wrong way.

Another minor problem is the less than ideal mapping of knots (Alpha values) to positions. A single cycle is specified to take 6 seconds, and the time taken to move between adjacent points in the early part of the journey is 0.6 seconds (6 * 0.1 alpha). But the final transition from point 7 to point 8 takes longer – 1.8 seconds (6*0.3), which means that the penguin noticably slows down.

1.3. Walking and Rotating in a Square

A more useful interpolator for this kind of penguin pilgrimage is RotPosPathInterpolator, which permits a path to mix positions and rotations. In other words, we can extend the path from section 1.2 with rotations to ensure that the penguin is always facing forward.

The bad news is that these rotations must be specified as quaternions which define a rotation as an angular displacement around an axis written in a notation based on complex numbers. The big advantage of quaternions is its slerp operation (spherical linear interpolation) which smoothly transitions between two quaternions. This is preferable to rotations defined as Euler angles whose interpolation can be problematic.

Nevertheless, Euler angles are much easier to understand than complex numbers, and so Simple3D's RotPosPath class allows a path's rotations to be supplied as degree angles. To simplify matters, only rotations around the x-, y-, and z- axes can be utilized, and the same axis is used for all the rotations.

The path we wish to specify is shown in Figure 7, with the positions and rotations numbered to signify their execution order.

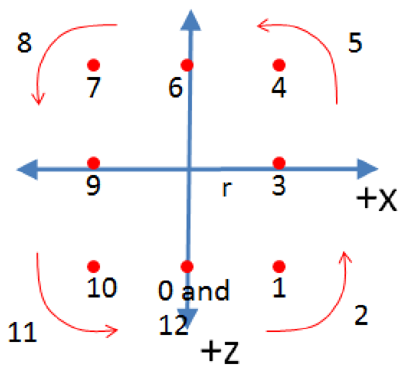

Figure 7. The Points and Rotations in the Penguin's Path.

Comparing Figure 7 with Figure 5, the same positions are used, but four rotations have been added. For example, when the penguin is at point 1, it rotates left by 90 degrees, and then moves to point 3.

The path is set up in squareRotPath():

```
private static TransformGroup squareRotPath(TransformGroup obj,
                                            double radius)
{
  float r = (float)radius;

  RotPosPath rpPath = new RotPosPath(Simple3D.Y_AXIS);
  rpPath.add(0,0,r, 0.0f);  // datum 0: (x,y,z), angle
  rpPath.add(r,0,r, 0.0f);  // unchanged rot, so only a move
  rpPath.add(r,0,r, 90.0f); // unchanged pos, so only a rotation

  rpPath.add(r,0,0,  90.0f);  // 3
  rpPath.add(r,0,-r, 90.0f);
  rpPath.add(r,0,-r, 180.0f);

  rpPath.add(0,0,-r,  180.0f);  // 6
  rpPath.add(-r,0,-r, 180.0f);
  rpPath.add(-r,0,-r, 270.0f);

  rpPath.add(-r,0,0, 270.0f);  // 9
  rpPath.add(-r,0,r, 270.0f);
  rpPath.add(-r,0,r, 0.0f);

  rpPath.add(0,0,r, 0.0f);   // 12

  return rpPath.getTG(obj, 6000);
}  // end of squareRotPath()
```

Each RotPosPath.add() creates a path datum consisting of both a coordinate and a rotation. The underlying RotPosPathInterpolator carries out both operations at once.

If a path datum should only carry out a translation, then the rotation argument must be left unchanged from the previous path data item. This can be seen by comparing data 3 and 4 (i.e. (r,0,0, 90) and (r,0,-r, 90)) which moves the penguin from (r,0,0) to (r,0,-r) but leaves its rotation unchanged.

In a similar way, if a path datum should only perform a rotation, then the translation part of the tuple should be left unchanged from the previous data item. This can be seen in action by comparing data 1 and 2 (i.e. (r,0,r,0) and (r,0,r,90)) which leaves the penguin at coordinate (r,0,r) but rotates it 90 degrees around the y-axis.

Note that all the rotations are absolute values (e.g. 0, 90, 180, 270), applied to the penguin's orientation at the start.

The axis used by the rotations is given in the RotPosPath() constructor; other possible values are Simple3D.X_AXIS and Simple3D.Z_AXIS, with the optional inclusion of a minus sign.

RotPosPath.getTG() creates the Alpha timer and RotPosPathInterpolator beneath a TransformGroup node. The Euler angles are converted into quaternions, and a knot sequence is generated which maps alpha values to the data items by equaling dividing the time between them. This means that each datum will take the same amount of time to be carried out.

The addition of the rotations to the path means that the penguin now stays facing forward as it moves around the square, as illustrated by the screenshots in Figure 8.

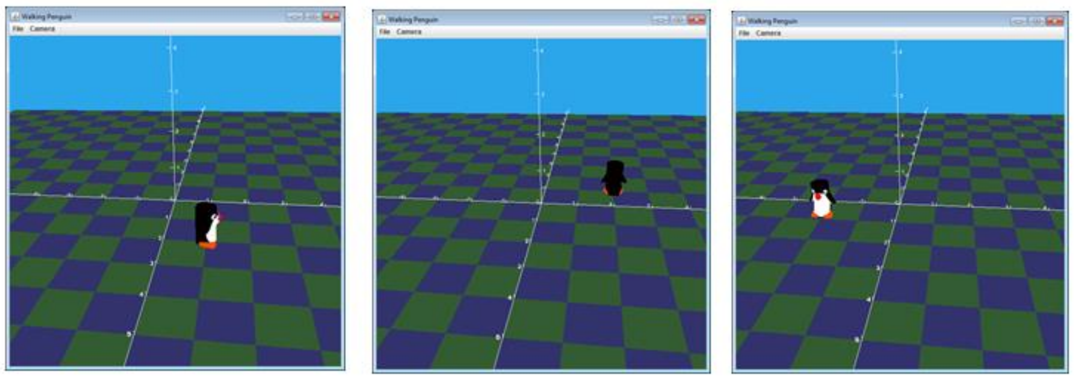

Figure 8. The Penguin's Rotating Square Dance.

1.4. Only Rotations

A somewhat frivolous use of RotPosPath is to approximate an orbital path as a series of small translations and rotations around a circle. It's frivolous since its simpler to just use Simple3D.orbit(), and it's an approximation since each translation is linear not curved. However, if the angular change at each step is sufficiently small, this will be hard to detect. In essence, we're modeling a circle as a multisided polygon.

The fourth function in PenguinWalks, pathRotateY(), implements this approach, and nicely shows how a path can be generated algorithmically:

```
 private static TransformGroup pathRotateY(TransformGroup obj,
                                          double radius)
// A path that closely matches a circle.
{
  RotPosPath rpPath = new RotPosPath();
  for (float i = 0.0f; i <= 360.0f; i = i + 10.0f) {
    float z = (float)(radius*Math.cos( Math.toRadians(i)));
    float x = (float)(radius*Math.sin( Math.toRadians(i)));
    rpPath.add(x,0.0f,z, i);   // (x,y,z), angle (both changing)
  }

  return rpPath.getTG(obj, 6000);
}  // end of pathRotateY()
```

The angles generated by the for-loop increase in steps of 10 degrees, and it ends by including the 360 degree value so that the penguin returns to its starting point.

2. Do-It-Yourself (DIY) Animation

DIY animation employs a loop to change models by a small amount, pauses for a short time, and then repeats. This approach is perfectly adequate so long as the code doesn't need to do anything else, since it will probably have to stay in the animation loop for the duration of the program.

This section describes two examples of DIY animation: ChangeVisibility.java and MovingModels.java. Section 3 begins by recoding MovingModels to use Java 3D's Behavior, which is a better approach for all but the simplest animation.

2.1. The Ghostly Sphere

ChangeVisibility.java displays a floating red sphere, which gradually becomes invisible after the user presses `<enter>`. Pressing `<enter>` again returns it to visibility, and the process can be repeated ad infinitum, or until the excitement becomes too much to bear.

The code uses a specialized version of Alpha, called FireAlpha, which executes the timer for one cycle whenever FireAlpha.fire() is called. The associated interpolator is executed once, but can be executed again by another call to fire().

ChangeVisibility uses two FireAlpha objects, one to control a TransparencyInterpolator that transitions the shape to invisibility, and another controlling a TransparencyInterpolator that makes the shape opaque. ChangeVisibility's main() sets up the scene graph:

```
public static void main(String[] args)
{
  Simple3D j3d = new Simple3D("Change Color", true, true);

  /* bg --> posTG -->  sphereTG --> sphere
                        |
                        ----> invis interpolator
                        |
                        ----> vis interpolator
  */

  Appearance app = Simple3D.color("red");
  TransparencyAttributes tas = new TransparencyAttributes(
                  TransparencyAttributes.NICEST, 0);
  tas.setCapability(TransparencyAttributes.ALLOW_VALUE_WRITE);
  app.setTransparencyAttributes(tas);

  Sphere sphere = new Sphere(0.5f, app);
  boolean isVisible = true;

  TransformGroup sphereTG = new TransformGroup();
  sphereTG.addChild(sphere);

  // visibility changers (0 = opaque; 1 = invisible)
  FireAlpha invisAlpha = new FireAlpha(DELAY);
  TransparencyInterpolator invis =
            new TransparencyInterpolator(invisAlpha.getAlpha(),
                          app.getTransparencyAttributes(), 0, 1f);
  invis.setSchedulingBounds(Simple3D.INFINITE_BOUNDS);
  sphereTG.addChild(invis);

  FireAlpha visAlpha = new FireAlpha(DELAY);
  TransparencyInterpolator vis =
            new TransparencyInterpolator(visAlpha.getAlpha(),
                          app.getTransparencyAttributes(), 1f, 0);
  vis.setSchedulingBounds(Simple3D.INFINITE_BOUNDS);
  sphereTG.addChild(vis);

  // position the sphere
  TransformGroup posTG = Simple3D.setTranslation(0,1,0);
  posTG.addChild(sphereTG);

  // build the scene graph
  BranchGroup bg = Simple3D.createBG(posTG);
  j3d.rootBG.addChild(bg);

     : // more code
     :
```

The animation loop begins after the scene graph has been made live:

```
    Scanner s = new Scanner(System.in);
    System.out.println("Press <enter> to toggle
                        the sphere's visibility...");
    while (true) {
      s.nextLine();     // pause until user types enter

      if (isVisible) {    // && !invisAlpha.isRunning())
        invisAlpha.fire();
        System.out.println("--> invisible");
        isVisible = false;
      }
      else if (!isVisible) {  // && !visAlpha.isRunning())
        visAlpha.fire();
        System.out.println("--> visible");
        isVisible = true;
      }
      Simple3D.pause(DELAY);  // give interpolator time to finish
    }
```

The drawback of this approach, as mentioned above, is that main() is fully occupied with the animation loop, so it’s hard to extend this code to do other things.

The commented-out calls to FireAlpha.isRunning() are a way of checking if the currently running interpolation is still executing. These tests aren't needed here since the animation DELAY time (2 seconds) equals the execution time for both timers.

2.2. Animating Several Objects

MovingModels.java has three animated elements – a rotating sphere, a car model driving in a circle, and 3D text floating up and down. The scene is shown in Figure 9.

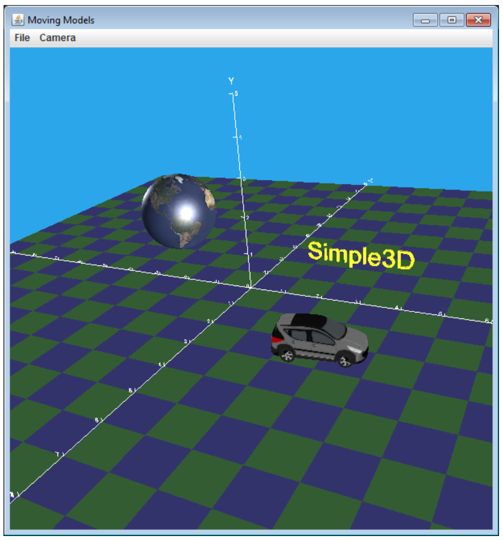

Figure 9. Three Models Moving.

The main() function of MovingModels has a similar structure to ChangeVisibility.java: the first part sets up the scene graph, and the second starts the animation loop. The scene graph utilizes four global TransformGroup variables:

```
private static TransformGroup posCarTG, rotCarTG,
                              rotSphereTG, posTextTG;
```

They are used in the scene graph like so:

```
       sceneBG --> posCarTG --> rotCarTG --> car model
            |
            -----> pos2 --> rotSphereTG --> sphere
            |
            -----> posTextTG --> scale --> text
```

My reason for using globals is to make it easier to compare this approach to the Behavior-based version in the next section.

The animation loop calls animate():

```
    //  in main()
    while (true) {
      animate();
      Simple3D.pause(DELAY);
    }
```

Although animate() is only animating three models, it updates four TransformGroups since the circling motion of the car is a combination of a translation applied via posCarTG and a rotation in rotCarTG:

```
  // globals
  private static final double CAR_RADIUS = 2.5;
  private static final double CAR_ANGLE = 5;
                           // angle step on each update
  private static final double EARTH_ANGLE = 5;
  private static final int MAX_LEV_STEPS = 20;

  private static double carAngle = 180;
  private static double levDist = 0.1;
  private static int levStep = 0;


  private static void animate()
  {
    // rotate the sphere
    Simple3D.rotYBy(rotSphereTG, EARTH_ANGLE);

    // drive the car in a circle
    carAngle = (carAngle + CAR_ANGLE)%360;
    double rads = Math.toRadians(carAngle);
    Simple3D.moveTo(posCarTG,
                  CAR_RADIUS+CAR_RADIUS*Math.cos(rads), 0,
                 -CAR_RADIUS*Math.sin(rads));
    Simple3D.rotYBy(rotCarTG, CAR_ANGLE);

    // move the text up or down
    Simple3D.moveBy(posTextTG, 0, levDist, 0);
    levStep++;
    if (levStep > MAX_LEV_STEPS) {  // toggle direction
      levDist = -levDist;
      levStep = 0;
    }
  }  // end of animate()
```

The car's position is modified with Simple3D.moveTo() which sets its absolute position, whereas the other transformations employ Simple3D's "By" methods which apply incremental changes.

3. Having Java 3D Behave

A Behavior object executes code in a separate thread when triggered by events, thereby freeing up the rest of the program for other tasks. In particular, animation updates, such as the work of animate() from the last section, can be performed by a Behavior rather than the main program.

Many types of events (called wakeup conditions in Java 3D) are supported, as indicated by the numerous WakeupCriterion subclasses listed in Figure 10. These can be combined using "and" and "or" operators so a Behavior can be woken in multiple ways.

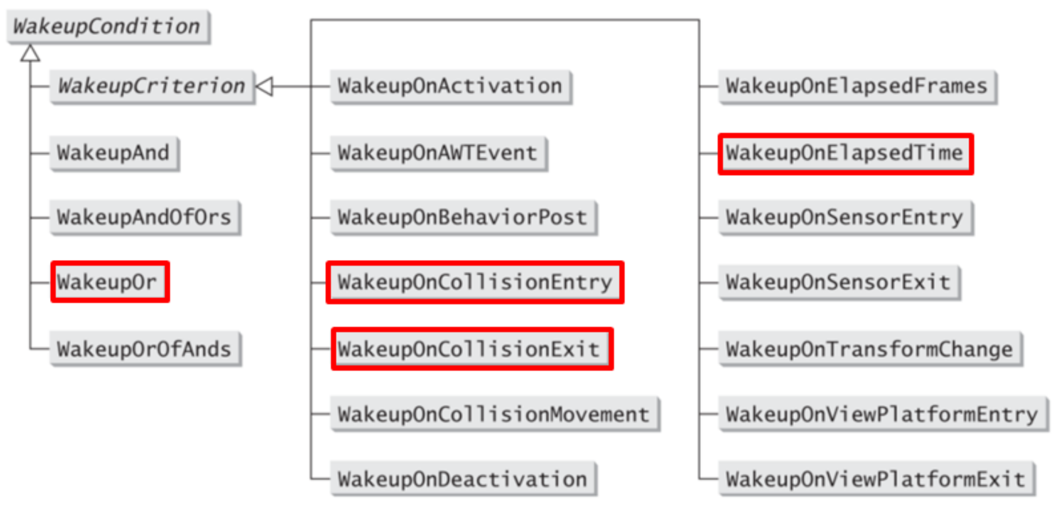

Figure 10. Subclasses of WakeupCondition.

The red rectangles in Figure 10 are the wakeup conditions used in this chapter. Almost all my examples use wakeupOnElapsedTime in order to implement time-based triggers for animation actions. The two collision wakeups are employed in section 7 on collision detection.

The format of a Behavior class is fairly standard, as typified by TimeBehavior below, which is part of Simple3D.

```
public class TimeBehavior extends Behavior
{
  private WakeupCondition wakes;
  private int delay;
  private TimeUpdater tu;


  public TimeBehavior(int delay, TimeUpdater tu)
  {
    this.delay = delay;
    this.tu = tu;
    setSchedulingBounds(Simple3D.INFINITE_BOUNDS);
  }


  public void initialize()
  {
    WakeupCriterion crits[] = new WakeupCriterion[1];
    crits[0] = new WakeupOnElapsedTime(delay);  // add a condition

    wakes = new WakeupOr(crits);  // wake on any condition
    wakeupOn(wakes);  // make conditions live
  }


  public void processStimulus(Iterator<WakeupCriterion> wCrits)
  {
    WakeupCriterion wakeUp;

    while (wCrits.hasNext()) {  // check each condition
      wakeUp = wCrits.next();
      if (wakeUp instanceof WakeupOnElapsedTime) {
        tu.tick(delay);
      }
    }
    wakeupOn(wakes);   // reset wake conditions
  } // end of processStimulus()

}  // end of TimeBehavior class
```

The constructor stores various globals, and also calls setSchedulingBounds(), which you may recall from the Interpolator examples. It specifies when the Behavior is active relative to the current camera position, and the simplest thing is to make it always live by using Simple3D.INFINITE_BOUNDS.

initialize() sets up the conditions that will trigger the behavior. Multiple wakeups are added to a WakeupCriterion[] array, and the behavior will be executed when any of them are detected by placing them in an or-combination.

TimeBehavior only needs to wait for a single time condition, but I've retained the array and or-condition to show those features. A crucial final step of initialize() is to call wakeupOn() which makes the conditions live.

processStimulus() is called when a wakeup condition is triggered. Typically, it uses a loop to examines all the pending events. The type of each one is examined, and corresponding code is executed. TimeBehavior does something if it finds a WakeupOnElapsedTime event.

As with initialize(), an important final step is to call wakeupOn() which makes the conditions live once again. This allows future events to trigger the behavior.

3.1. Using TimeBehavior

TimeBehavior is used in conjunction with the TimeUpdater interface:

```
interface TimeUpdater
{
  public void tick(int delay);
}
```

An object that wants to be affected by TimeBehavior implements TimeUpdater, and the behavior will call its tick() each time that processStimulus() is called.

This approach can be used to recode the MovingModels program of the last section, now cleverly renamed MovingModels**2**. Its main() function does two things – create a MovingModels2 object which builds the scene graph and implements TimeUpdater, and instantiates a TimeBehavior object that fires MovingModels2.tick():

```
   public static void main(String[] args)
  /* of MovingModels2

       rootBG --> sceneBG
                 |
                 ----> TimeBehavior
  */
  {
    Simple3D j3d = new Simple3D("Moving Models 2", 0,1,10, true,
true);

    MovingModels2 mm = new MovingModels2();
    TimeBehavior tb = new TimeBehavior(DELAY, mm);

    // build top-level scene
    BranchGroup bg = Simple3D.createBG();
    bg.addChild(mm.getScene());
    bg.addChild(tb);

    j3d.rootBG.addChild(bg);
  }  // end of main()
```

MovingModels2's constructor builds a scene graph identical to the one in MovingModels except that it's stored in a global BranchGroup variable accessible via MovingModels2.getScene():

```
  private BranchGroup sceneBG;

  // globals for animation
  private TransformGroup posCarTG, rotCarTG,
                         rotSphereTG, posTextTG;

    :  // many other globals

  public MovingModels2()
  { /*
       sceneBG --> posCarTG --> rotCarTG --> car model
                 |
                 ------> pos2 --> rotSphereTG --> sphere
                 |
                 ------> posTextTG --> scale --> text
    */

    // a moveable car
       : // same code as in MovingModels

    // a rotating, positioned sphere
       :

    // scaled, positioned text
       :

    // build top-level scene
    sceneBG = Simple3D.createBG();
    sceneBG.addChild(posCarTG);
    sceneBG.addChild(pos2TG);
    sceneBG.addChild(posTextTG);
  }  // end of movingModels2()


  public BranchGroup getScene()
  {  return sceneBG;  }
```

Unlike MovingModels there's no animation loop which calls animate(); that's replaced by TimeBehavior which calls MovingModels2.tick():

```
  public void tick(int delay)
  // animation code; equivalent to animate() in MovingModels
  {
    // rotate the sphere
    Simple3D.rotYBy(rotSphereTG, EARTH_ANGLE);

    // drive the car in a circle
    carAngle = (carAngle + CAR_ANGLE)%360;
    double rads = Math.toRadians(carAngle);
    Simple3D.moveTo(posCarTG,
                  CAR_RADIUS+CAR_RADIUS*Math.cos(rads), 0,
                 -CAR_RADIUS*Math.sin(rads));
    Simple3D.rotYBy(rotCarTG, CAR_ANGLE);

    // move the text up or down
    Simple3D.moveBy(posTextTG, 0, levDist, 0);
    levStep++;
    if (levStep > MAX_LEV_STEPS) {  // toggle direction
      levDist = -levDist;
      levStep = 0;
    }
  }  // end of tick()
```

Since tick() is called by the behavior, all the variables that it manipulates must be defined as globals, including the posCarTG, rotCarTG, rotSphereTG, and posTextTG TransformGroups.

3.2. A Swirling Gas Giant

GasGiant.java utilizes TimeBehavior to perform animation in a similar way to MovingModels2: it implements TimeUpdater.tick() which is called periodically to update the scene. However, this time, the update is to the position of a texture wrapped around a sphere, causing it to rotate clockwise over the surface. This makes the pattern look a bit like clouds circulating on a planet like Jupiter, which explains the program's name. Figure 11 shows two screenshots of the scene.

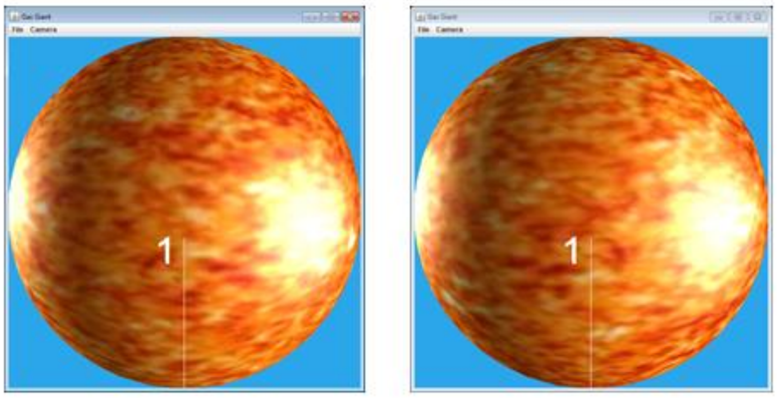
Figure 11. Jupiter's a Real Gas.

GasGiant's main() works in a similar way to MovingModels2, creating the GasGiant object, a TimeBehavior object, and building the scene:

```
public static void main(String[] args)
/* // in GasGiant
     rootBG --> sceneBG
          |
          ----> TimeBehavior
*/
{
  Simple3D j3d = new Simple3D("Gas Giant", false, true);

  GasGiant gg = new GasGiant();
  TimeBehavior tb = new TimeBehavior(DELAY, gg);

  // build top-level scene
  BranchGroup bg = Simple3D.createBG();
  bg.addChild(gg.getScene());
  bg.addChild(tb);
  j3d.rootBG.addChild(bg);
}  // end of main()
```

The GasGiant constructor creates a textured sphere and changes the texture's attributes so its position can be adjusted at run time. It also initializes two global transforms which will be utilized by tick():

```
private BranchGroup sceneBG;

// globals for animation
private TextureAttributes texAttrs;
private Transform3D toCenter, toOrigin;


public GasGiant()
/*   sceneBG -->  sphere
```

```
  */
  {
    Sphere shape = Simple3D.texSphere(1, "images/fire.jpg");

    texAttrs = shape.getAppearance().getTextureAttributes();
    texAttrs.setCapability( TextureAttributes.ALLOW_TRANSFORM_WRITE);

    // create transforms for moving the texture so its center will be
    // the center of rotation
    toCenter = new Transform3D();
    toCenter.setTranslation(new Vector3d(0.5, 0.5, 0));

    toOrigin = new Transform3D();
    toOrigin.setTranslation(new Vector3d(-0.5, -0.5, 0));

    // build top-level scene
    sceneBG = Simple3D.createBG(shape);
  }  // end of GasGiant()


  public BranchGroup getScene()
  {  return sceneBG;  }
```

GasGiant.tick() rotates the texture around its center in three steps. First, the center of the texture is moved to the origin, then a rotation is applied to the origin, and then the texture is moved back to the center. The coordinates of the texture center are (0.5, 0.5), since 2D texture coordinates (s,t) go from 0 to 1.

tick() is:

```
  public void tick(int delay)
  // animation code: rotate the texture by TURN_ANGLE
  {
    Transform3D attrT3D = new Transform3D();
    Transform3D t3d = new Transform3D();

    t3d.rotZ(Math.toRadians(TURN_ANGLE));
    texAttrs.getTextureTransform(attrT3D);

    // rotate around center of texture
    attrT3D.mul(toCenter);   // move origin back to center
    attrT3D.mul(t3d);        // rotate
    attrT3D.mul(toOrigin);   // move center to origin

    texAttrs.setTextureTransform(attrT3D);
  }  // end of tick()
```

While I was developing the code, I found it useful to have GasGiant modify a textured *cube*, which made it easier to observe if the texture is being moved correctly since each face of the cube is updated separately.

4. Morphing

A fairly common form of animation which isn't best handled by a subclass of Behavior is the changing of shape geometry. Java 3D offers two ways to gradually change the look of a shape over time – by using the Morph class, and by implementing the GeometryUpdater interface. This section will look at the former and GeometryUpdater is covered in section 5.

For reasons that are unclear to me, the Morph class was deprecated in Java 3D 1.4. Arguably, Morph is less flexible than GeometryUpdater, and I did encounter a scene refreshing bug when coding the MorphCube.java example.

The Morph class is supplied with an array of similar GeometryArrays (similar in the sense of having the same number of points, texture coordinates, and indices). It's also assigned an array of weights which specify how much of each shape should be used to render a 'combined' single shape.

Animation enters the fray because it's possible to change the values in the weights array at run time, and so cause the 'combined' shape to change.

MorphingCube.java utilizes an array of four shapes consisting of a cube, a truncated pyramid, another cube, and an inverted truncated pyramid. Figure 12 shows the coordinate indices for the three different shapes.

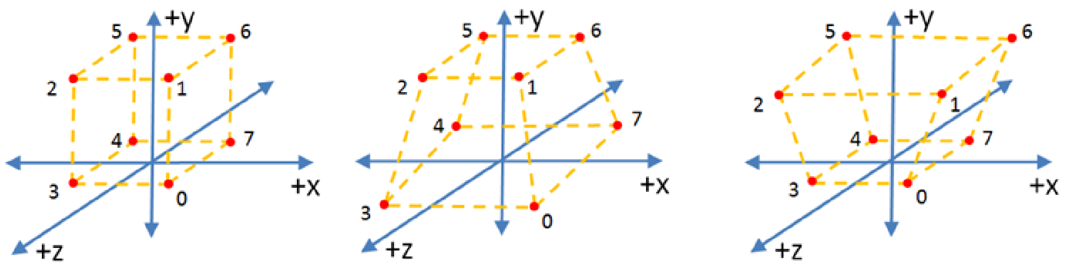

Figure 12. A Cube, a Truncated Pyramid, and an Inverted Truncated Pyramid.

Each shape has the same number of coordinates, indexed in the same order, but with no texture coordinates (in order to reduce the complexity of the code a little).

The shapes array includes the cube twice in order to simplify the code that changes the weights array. The intention is that the cube will morph into the truncated pyramid, then into the second cube, then into the inverted truncated pyramid, and thereafter repeat by morphing back into the first cube. The overall effect is of a cube that's oscillates in and out at its base and top. Figure 13 shows a few screenshots of the animation.

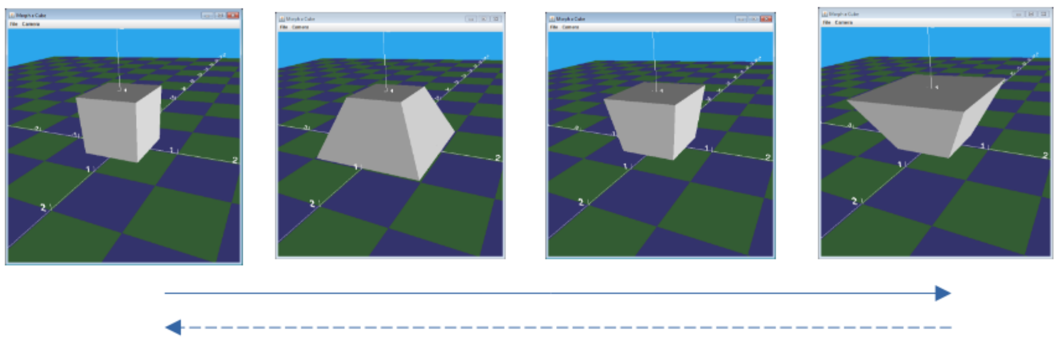

Figure 13. The Oscillating Cube.

The main() function creates a MorphCube object (which implements TimeUpdater) and a TimeBehavior object. Each time that MorphCube.tick() is called, the morph's weights array is modified, thereby causing the rendered shape to change.

```
public static void main(String[] args)
/* // in MorphCube.java
     rootBG --> sceneBG
           |
           ----> TimeBehavior
*/
{
  Simple3D j3d = new Simple3D("Morph a Cube", true, true);

  MorphCube mc = new MorphCube(j3d);
  TimeBehavior tb = new TimeBehavior(DELAY, mc);

  //assemble scene graph
  BranchGroup bg = Simple3D.createBG();
  bg.addChild(mc.getScene());
  bg.addChild(tb);
  j3d.rootBG.addChild(bg);
}  // end of main()
```

MorphCube() creates the array of shapes:

```
private BranchGroup sceneBG;

// globals for animation
private Morph morph;
private Alpha alpha; // used by tick()
private Simple3D j3d;


public MorphCube(Simple3D j3d)
{
  this.j3d = j3d;    // so the morph shape can be reliably redrawn

  /* The morph contains 4 'frames':
     morph --> cube
```

```
              --> truncated pyramid
              --> cube
              --> inverted truncated pyramid
    */
    GeometryArray[] geoms = new GeometryArray[NUM_FRAMES];
    geoms[0] = createCuboid(cubeCoords());
    geoms[1] = createCuboid(pyraCoords());
    geoms[2] = createCuboid(cubeCoords());
    geoms[3] = createCuboid(ipyraCoords());

    morph = new Morph(geoms, Simple3D.color("gray"));
    morph.setCapability(Morph.ALLOW_WEIGHTS_WRITE);

    alpha = new Alpha(-1, 4000); // 4 sec period before repeating

    sceneBG = Simple3D.createBG(morph);
  }  // end of MorphCube()


  public BranchGroup getScene()
  {  return sceneBG;  }
```

createCuboid() uses an array of Point3d coordinates to build a GeometryArray with the GeometryInfo helper class discussed in the previous chapter:

```
  private static GeometryArray createCuboid(Point3d[] coords)
  /* Use supplied coordinates to generate faces, and
     also normals, for the geometry
  */
  {
    GeometryInfo gi = new GeometryInfo(GeometryInfo.QUAD_ARRAY);
    gi.setCoordinates(coords);

    int indices[] = {
           0,1,2,3,    // front (CCW)
           4,5,6,7,    // back
           0,7,6,1,    // right
           3,2,5,4,    // left
           1,6,5,2,    // top
           4,7,0,3     // bottom
    };
    gi.setCoordinateIndices(indices);

    NormalGenerator ng = new NormalGenerator();
    ng.generateNormals(gi);

    return gi.getGeometryArray();
  }  // end of createCuboid()
```

createCuboid() expects its array argument to contain eight coordinates organized in the order shown in Figure 12. For example, the function that supplies the cube's coordinates is:

```
private static Point3d[] cubeCoords()
/* coordinates for a 1 x 1 x 1 cube,
   centered at origin, on XZ floor  */
{
  Point3d[] coords = {   // 8 vertices
     new Point3d(0.5, 0, 0.5),    // front; point 0
     new Point3d(0.5, 1, 0.5),
     new Point3d(-0.5, 1, 0.5),
     new Point3d(-0.5, 0, 0.5),

     new Point3d(-0.5, 0, -0.5),  // back; point 4
     new Point3d(-0.5, 1, -0.5),
     new Point3d(0.5, 1, -0.5),
     new Point3d(0.5, 0, -0.5)
  };
  return coords;
}
```

Generating the Weights Array

The currently rendered shape is a mix of the current shape and the next shape in the array, modulo the number of shapes. The modulo means that when the current shape is the inverted truncated pyramid (i.e. the last shape in the array), then the 'next' shape will be the first cube.

This brings up the issue of how tick() determines what the 'current' shape is when it's called. I solved this by using an Alpha which cycles every 4 seconds. The Alpha value, which ranges between 0.0 and 1.0, is mapped to the index of the shapes array, which ranges from 0 to 3. The cycle time means that the sequence depicted in Figure 12 will take 4 seconds to repeat.

```
public void tick(int delay)
/* animation code. The morphing is expressed in the weights of
   adjacent frames which add up to a single 'shape' being
   drawn for this tick() call
 */
{
  double[] weights = new double[NUM_FRAMES];  // all values are 0

  float scaledAlpha = (float) (NUM_FRAMES * alpha.value());
                           // a float between 0 and NUM_FRAMES
  int idx = (int) scaledAlpha;
        // truncate to convert into an index for weights[]

  // increase weight stored at next index; decrease current one;
  double nextWeight = (double)scaledAlpha - (double)idx;
  double currWeight = 1.0 - nextWeight;
               // total of all weights must == 1

  /* the rendered shape is these weights applied to adjacent shapes
     in the Morph object */
  weights[idx] = currWeight;
```

```
    weights[(idx+1)%NUM_FRAMES] = nextWeight;
                 // use modulo to wraparound in array
/*
    // VERY useful for debugging purposes
    System.out.println("scaledAlpha: " + scaledAlpha +
                       "; index: " + idx);
    System.out.println("weights: " + Arrays.toString(weights));
*/
    morph.setWeights(weights);

    j3d.canvas3D.getView().repaint();
        // necessary to force morphed shape to redraw
  }  // end of tick()
```

Probably the most essential piece of code in tick() is the commented-out print statements which report the calculated weights. In this case, the printed values should combine the current shape and the next one. Also, over time, the weight for the current shape should decrease while the weight applied to the next one increase.

tick() also contains my Morph bug fix. The last line of tick() calls repaint() for the 3D canvas:

```
    j3d.canvas3D.getView().repaint();
```

Without this line, the rendered shape stays unchanged in its initial cube form.

5. The GeometryUpdater Interface

In the current version of Java 3D, it seems that the preferred way to update geometries is by using the GeometryUpdater interface. It's advantage over Morph, aside from not being deprecated, is that it doesn't require arrays of shapes and weights. Instead, shape coordinates are modified directly.

One tricky aspect of this approach is that the shape being modified must refer to its coordinates by reference rather than by copying them into a Shape3D object. This makes sense since the coordinates are changed at run time, not the shape itself.

Another difference is that GeometryUpdater cannot be manipulated by my TimeBehavior class. As we'll see, a new Behavior subclass is needed.

The execution of PulseCube.java is shown in the screenshots in Figure 14.

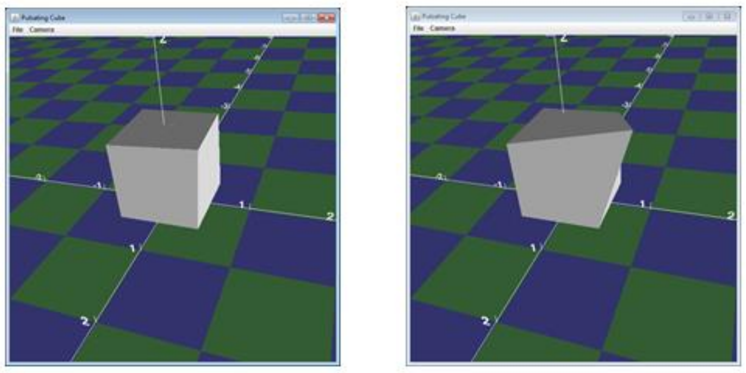

Figure 14. A Pulsing Cube.

The upper, front, right vertex of the cube gradually stretches outwards, then just as gradually reverts to its original position, and then repeats.

The main() function creates a PulseCube object which implements the GeometryUpdater interface, which means that it must implement:

```
void updateData(Geometry geo)
```

This will be called periodically by my GeometryUpdaterBeh behavior in order to modify the cube's corner.

```
public static void main(String[] args)
/*
     rootBG --> pulsating cube
          |
          ----> GeometryUpdaterBeh
*/
{ Simple3D j3d = new Simple3D("Pulsating Cube", true, true);

  PulseCube pc = new PulseCube();
  GeometryUpdaterBeh gub =
          new GeometryUpdaterBeh(pc.getGeom(), pc);

  // assemble scene graph
  BranchGroup bg = Simple3D.createBG();
  bg.addChild(pc.getShape());
  bg.addChild(gub);
  j3d.rootBG.addChild(bg);
}  // end of main()
```

5.1. The PulseCube Class

PulseCube creates a cube from an IndexedQuadArray which must permit BY_REFERENCE access to its coordinates. In addition, each cordinate must be defined as three doubles. All of this is done by createCube():

```
// global
private double[] coords;  // so available to updateData()


private Geometry createCube()
// 1 x 1 x 1 cube, centered at origin, on XZ floor
// with normals applied to the faces
{
  IndexedQuadArray qa = new IndexedQuadArray(NUM_COORDS,
                           IndexedQuadArray.COORDINATES|
                           IndexedQuadArray.NORMALS |
                                 IndexedQuadArray.BY_REFERENCE,
                                 NUM_COORDS*3);
       /* geometry updating requires that the code be stored by
          reference so that changes to the coords[] array affect the
          shape */

    // the referenced data can be read and written
    qa.setCapability(IndexedQuadArray.ALLOW_REF_DATA_READ);
    qa.setCapability(IndexedQuadArray.ALLOW_REF_DATA_WRITE);

    coords = new double[] {
               0.5, 0, 0.5,    // 8 vertices; 3 doubles per coord
               0.5, 1, 0.5,    // 4 coords at the front
              -0.5, 1, 0.5,
              -0.5, 0, 0.5,

              -0.5, 0, -0.5,   // 4 coords at the back
              -0.5, 1, -0.5,
               0.5, 1, -0.5,
               0.5, 0, -0.5
    };
    qa.setCoordRefDouble(coords);

    int indices[] = {
           0,1,2,3,     // front (CCW)
           4,5,6,7,     // back
           0,7,6,1,     // right
           3,2,5,4,     // left
           1,6,5,2,     // top
           4,7,0,3      // bottom
    };
    qa.setCoordinateIndices(0, indices);


    float[] normals = {
            0,  0,  1f,     // +z
            0,  0, -1f,
           1f,  0,  0,      // +x
          -1f,  0,  0,
            0, 1f,  0,      // +y
            0,-1f,  0
       };
    qa.setNormalRefFloat(normals);

    // indices for normals array
    int normalIndices[] = { 0, 0, 0, 0,
                            1, 1, 1, 1,
                            2, 2, 2, 2,
                            3, 3, 3, 3,
                            4, 4, 4, 4,
                            5, 5, 5, 5   };
    //Set the data
    qa.setNormalIndices(0, normalIndices);

    return qa;
```

```
  }  // end of createCube()
```

The indices of the coordinates and the normals defined by createCube() are shown in Figure 15.

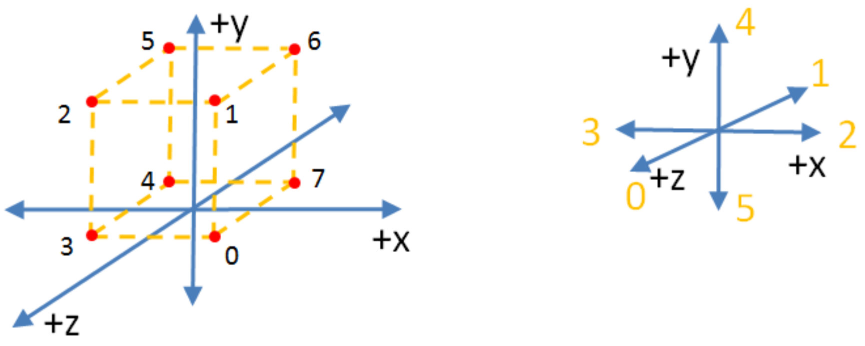

Figure 15. The Pulse Cube and its Normals.

According to Figure 15's labeling, the cube's index 1 coordinate will be adjusted at run time.

Aside from the IndexedQuadArray being set to BY_REFERENCE, it's also necessary to assign the coordinates and the normal data to the quad using "Ref" methods. This is true even in this example where the normals data won't be changed at run time.

Although my decision to leave the normals alone simplifies my code, it means that the lighting on the faces adjacent to the modified point won't change even though the angles of those faces to the lights is changing.

PulseCube also implements GeometryUpdater.updateData():

```
  // globals
  private int currStep = 0;
  private boolean isGrowing = true;


  public void updateData(Geometry geo)
  /* This method is trigged by GeometryUpdaterBeh.processStimulus().
     Note: only a single coordinate (corner 1) is changed.
  */
  {
  //  IndexedQuadArray qa = (IndexedQuadArray)geo;   // not used
  //  double coords[] = qa.getCoordRefDouble();

    // is corner 1 of the cube growing or shrinking?
    if (isGrowing) {
      coords[ coordIdx(1,X_COORD) ] += CHG_DIST;  // add change
      coords[ coordIdx(1,Y_COORD) ] += CHG_DIST;
      coords[ coordIdx(1,Z_COORD) ] += CHG_DIST;
      if (currStep == MAX_STEPS)
        isGrowing = false;
      else
        currStep++;
    }
    else { // shrinking
      coords[ coordIdx(1,X_COORD) ] -= CHG_DIST;  // remove change
      coords[ coordIdx(1,Y_COORD) ] -= CHG_DIST;
      coords[ coordIdx(1,Z_COORD) ] -= CHG_DIST;
      if (currStep == 0)
        isGrowing = true;
      else
        currStep--;
    }
  }  // end of updateData()
```

The Geometry argument passed to updataData() can be used to access the shape's data in the way shown by the two commented-out lines at the start of the method. However, since PulseCube's coords array is a global, it's just as easy to modify it directly.

coordIdx() converts the supplied coordinate number and axis value into a coords[] index.

5.2. The GeometryUpdate Behavior

GeometryUpdaterBeh employs the same wakeup condition as Simple3D's TimeBehavior:

```
  // global
  private WakeupCondition wakes;

  public void initialize()
  // in GeometryUpdaterBeh
  {
    WakeupCriterion crits[] = new WakeupCriterion[1];
    crits[0] = new WakeupOnElapsedTime(DELAY);

    wakes = new WakeupOr(crits);
    wakeupOn(wakes);
  }  // end of initialize()
```

However, I can't use TimeBehavior in this example since processStimulus() must call GeometryArray.updateData(GeometryUpdater gu):

```
  // globals
  private GeometryArray ga;
  private GeometryUpdater gu;

  public void processStimulus(Iterator<WakeupCriterion> wCrits)
  {
    WakeupCriterion wakeUp;
    while(wCrits.hasNext()) {
      wakeUp = wCrits.next();
      if (wakeUp instanceof WakeupOnElapsedTime)
        ga.updateData(gu);    // request an update of the geometry
    }
    wakeupOn(wakes);
  }  // end of processStimulus()
```

I've always found the two uses of the name "updateData" by Java 3D confusing. To be clear, processStimulus()'s updateData(**GeometryUpdater** gu) will eventually call updateData(**Geometry** geo) in PulseCube.