M3G Chapter 2. An Animated Model

This chapter is about the animation of a penguin model so it moves in a circle around the origin on the XZ plane. Figure 1 shows two screenshots of the penguin at different points in its journey.

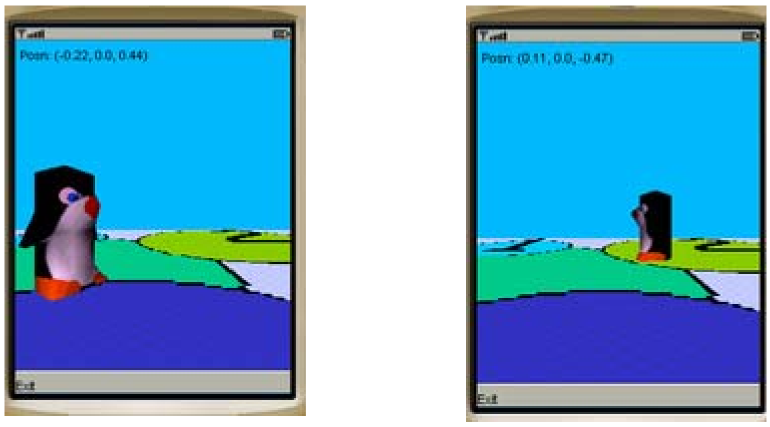

Figure 1. A Penguin Moving in a Circle.

Two animations are applied to the penguin: a *translation* to move the model around the circumference of a circle, and a *rotation* so the penguin stays pointing in its current direction of motion. The animations are implemented using M3G's AnimationTrack, KeyframeSequence, and AnimationController classes.

Other features of this application:

- we utilize M3G's *retained mode*, which means that the world is built using a *scene graph*;
- the floor is a textured square, implemented in its own Floor class;
- the display mixes 3D rendering and MIDP's drawString(), which adds the penguin's current coordinates to the top-left of the screen;
- the penguin model is represented by a Mesh object, constructed using coordinate data generated by the ObjView application described in the previous chapter. Model creation is wrapped up inside an AnimModel class;
- the scene uses a single directional light source, a light blue background, and a fixed camera. We also explain how to employ an image as a background.

The code is available in the AnimM3G/ subdirectory.

My thanks to Kari Pulli, Tomi Aarnio, and Kari J. Kangas of Nokia for helping me to understand animation using keyframe sequences.

1. UML Diagrams for AnimM3G

Figure 2 shows UML diagrams for the classes in the AnimM3G application. Only the class names and public/protected methods are shown.

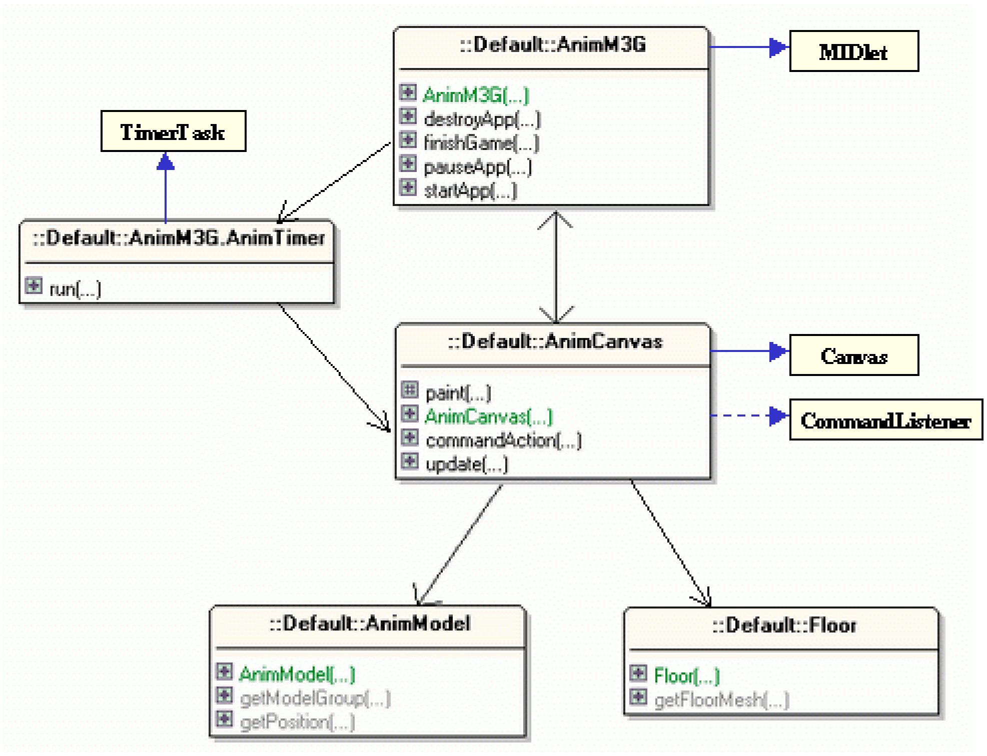

Figure 2. UML Class Diagrams for AnimM3G.

AnimM3G is the MIDlet, and creates an AnimTimer instance to periodically call update() in an AnimCanvas object.

AnimCanvas creates the scene graph, sets up the animations, and draws the scene. The floor mesh is created by a Floor object, while the penguin is created in an AnimModel instance.

2. Creating the Scene

The scene graph constructed by AnimCanvas is shown in Figure 3.

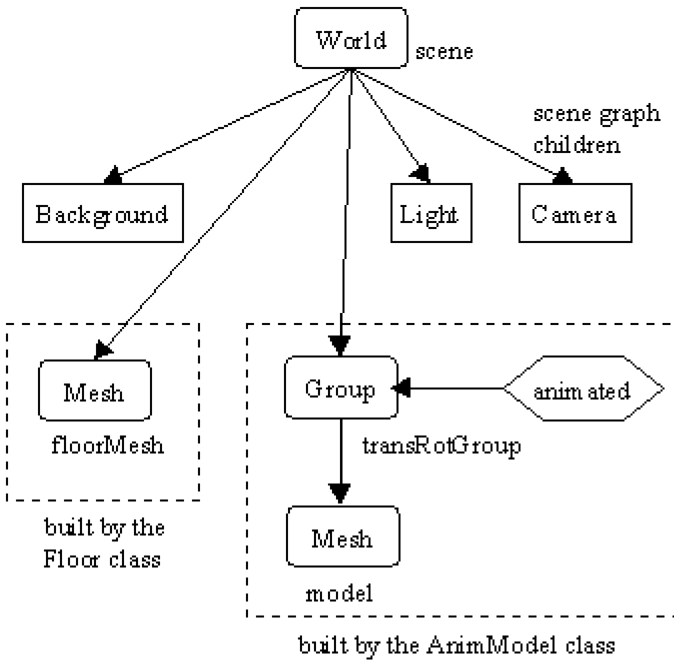

Figure 3. AnimCanvas Scene Graph.

The World node, called scene, is the graph's 'root', with the other nodes hanging below it. For now, I've hidden the animation behaviour in Figure 3 as an "animated" hexagon; I'll explain it later (see Figure 6 if you're interested).

The advantages of using a scene graph aren't really apparent in this simple example. With a Group node it's easy to collect objects together; the 'parent' Group can have translations, rotations, and other transformations applied to it, affecting all of its children. This is the function of the transRotGroup node in Figure 3: translation and rotation animations update the node, thereby affecting the Mesh model (the penguin) beneath it.

AnimCanvas' constructor calls buildScene() to create the graph:

```
private World scene;   // global variable


public AnimCanvas(...)
{ // ... other code
  scene = new World();
  buildScene():
  // ... other code
}

private void buildScene()
// add nodes to the scene graph
{
  addCamera();
  addLight();
  addBackground();
```

```
    animModel = new AnimModel();
    scene.addChild( animModel.getModelGroup() );   // add the model

    addFloor();
  } // end of buildScene()
```

2.1. Adding the Camera

The camera is created in the standard way, and moved up and back with setTranslation(). The camera's default position is at the origin, pointing along the negative z-axis into the scene.

```
  private void addCamera()
  {
    Camera cam = new Camera();
    float aspectRatio = ((float) getWidth()) / ((float) getHeight());
    cam.setPerspective(70.0f, aspectRatio, 0.1f, 50.0f);

    cam.setTranslation(0.0f, 0.5f, 2.0f);   // up and back
    // cam.setOrientation(-10.0f, 1.0f, 0, 0);
                      // angle downwards slightly

    scene.addChild(cam);
    scene.setActiveCamera(cam);
  }
```

The commented out call to setOrientation() would rotate the camera around the x-axis, turning it downwards.

The addChild() call connects the camera to the scene graph, below the scene node. The camera must also be made active.

A useful position for the camera when a scene is being tested, is directly overhead, looking downwards. The necessary translation and orientation settings are:

```
  cam.setTranslation(0.0f, 5.0f, 0.0f);
  cam.setOrientation(-90.0f, 1.0f, 0, 0);
```

The result for AnimM3G is shown in Figure 4.

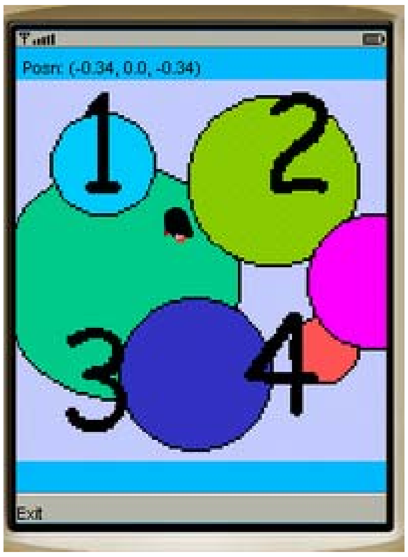

Figure 4. AnimM3G Viewed from Above.

All the nodes in a scene graph are subclasses of Node, which is a subclass of Transformable. This gives a node the ability to be translated, rotated, scaled, and transformed via a 4 by 4 matrix. The operations are applied to a node's position, p, according to the equation:

p' = translation * rotation * scaling * matrix_op * p

p' is the resulting new position.

The operations are applied to the node in a right-to-left order. For our camera, the rotation will be applied first, then it'll be translated. Due to the underlying use of homogenous coordinates, the rotation and translation components are independent of each other, so the translation won't be affected by the rotation. For our camera example in Figure 4, this means that the rotated camera will still be moved up the y-axis by 5 units, even though it's been turned face downwards.

All transformations can be encoded using only the 4 by 4 matrix component of a node, but the separation out of the translation, orientation and scale elements allows them to be animated independently of each other, a capability we're utilizing for the penguin model.

The matrix component cannot be animated directly, but it can be changed using setTransform(). We'll explore this coding style in the next chapter, when the penguin starts waving it's flippers.

2.2. Adding a Light

The code makes use of a default directional light, which emits in white. However, instead of using the default direction along the negative z-axis, the light is turned downwards by 45 degrees.

```
private void addLight()
{ Light light = new Light();  // default white, directional light
  light.setIntensity(1.25f);  // make it a bit brighter
  light.setOrientation(-45.0f, 1.0f, 0, 0); // down and into scene
  scene.addChild(light);
}
```

2.3. Adding the Background

A light blue background is added to the scene.

```
private void addBackground()
{ Background backGnd = new Background();
  backGnd.setColor(0x00bffe); // a light blue background
  scene.setBackground(backGnd);
}
```

An alternative is to draw an image in the background.

```
private void addBackground()
{
  Background backGnd = new Background();
  Image2D backIm = loadImage("/clouds.gif"); // cloudy blue sky
  if (backIm != null) {
    backGnd.setImage(backIm);
    backGnd.setImageMode(Background.REPEAT, Background.REPEAT);
  }
  else
    backGnd.setColor(0x00bffe); // a light blue background

  scene.setBackground(backGnd);
}
```

If the image file can't be found, then the background falls back to employing a light blue colour. The cloudy background is shown in Figure 5.

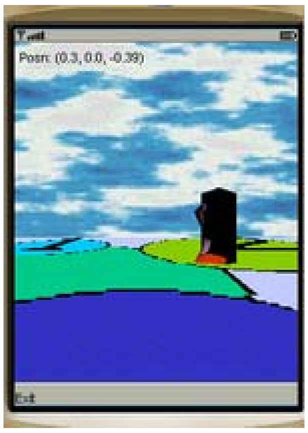

Figure 5. AnimM3G with Clouds in the Background.

The image is tiled over the background by using the REPEAT mode in both the horizontal and vertical directions.

loadImage() packages up the common task of loading an image and storing it as an Image2D object.

```
private Image2D loadImage(String fn)
{ Image2D im = null;
  try {
     im = (Image2D)Loader.load(fn)[0];
  }
  catch (Exception e)
  { System.out.println("Cannot make image from " + fn); }
  return im;
}  // end of loadImage()
```

The static method Loader.load() is an efficient choice here since it returns the required Image2D object (as the first element of an array of Object3Ds). An alternative would be to use MIDP's createImage(), but this returns an Image object, which must then be converted to Image2D in an extra step.

2.4. Adding the Penguin

The AnimModel constructor creates the Group and Mesh nodes for the penguin shown in Figure 3.

```
// global for translating and rotating the model
private Group transRotGroup;

public AnimModel()
{
  // other code ...

  Mesh model = makeModel();

  // reposition the model's start position and size
  model.setTranslation(0.25f, 0.25f, 0.25f); // so at center
  model.scale(0.5f, 0.5f, 0.5f);

  // translation/rotation group for the model
  transRotGroup = new Group();
  transRotGroup.addChild(model);

  // other code ...
}
```

Details of the mesh creation, implemented in makeModel(), will be explained later. The resulting Mesh object is a subclass of Transformable, so can be translated, rotated, and scaled. In this case, the model is moved up and scaled to make it rest on the XZ plane at the center, and be a reasonable size in relation to the floor.

A reference to the transRotGroup node is obtained by AnimCanvas calling AnimModel's getModelGroup():

```
public Group getModelGroup()
{  return transRotGroup;  }
```

2.5. Adding the Floor

The addFloor() method in AnimCanvas contains example code for making several different floors:

```
private void addFloor()
{
  Image2D floorIm = loadImage("/bigGrid.gif");
                // large, so slow to load
  Floor f = new Floor( floorIm, 8);   // 8 by 8 size

  // Image2D floorIm = loadImage("/grass.gif");
                   // or try "/floor.png"
  // Floor f = new Floor( floorIm, 6);   // 6 by 6 size

  scene.addChild( f.getFloorMesh() );
}
```

The Floor() constructor takes two arguments: an Image2D object representing the image, and the intended size of the floor. The floor will be centered at the origin of the XZ plane.

bigGrid.gif is a large green grid (512 by 512 pixels, 39Kb large), shown in Figure 6.

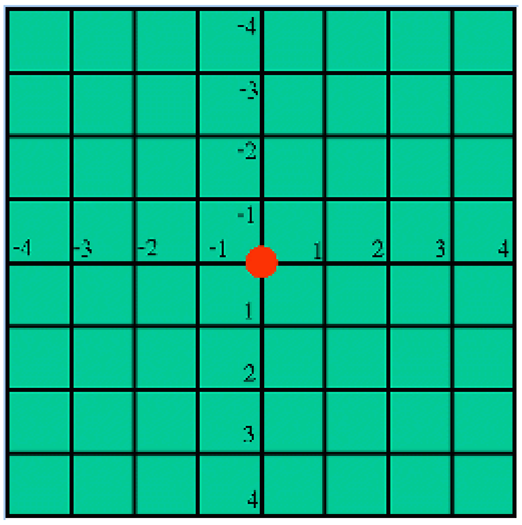

Figure 6. The bigGrid.gif Image.

The image is displayed with sides of 8 units in the scene, which will cause its red center to occur at the XZ origin.

bigGrid.gif is intended as a development aid, to check that the model (and any other objects) are positioned correctly in the scene. Figure 7 shows the penguin's original position in the scene with the animation code commented away, and bigGrid.gif used as the floor.

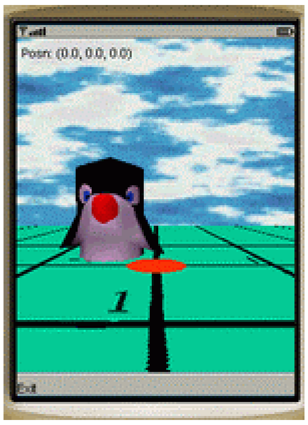

Figure 7. The Penguin's Initial Position in the Scene.

The penguin is offset 0.5 units along the negative x- and z- axes, and is too large. The start of AnimModel includes code that scales and translates the penguin so it will rest on the red circle at the center.

```
  Mesh model = makeModel();

  // reposition the model's start position and size
```

```
model.setTranslation(0.25f, 0.25f, 0.25f); // so at center
model.scale(0.5f, 0.5f, 0.5f);
```

The result is shown in Figure 8.

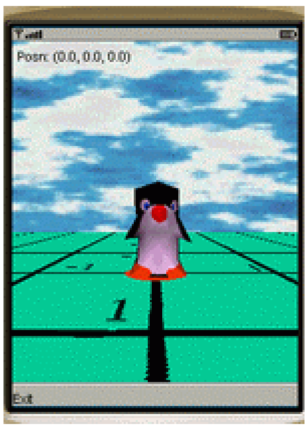

Figure 8. The Penguin Scaled and Translated.

This repositioning is important since the animation assumes that the penguin's starting position is at the origin.

grass.gif is a 64x64 pixel grass texture, shown as the floor in Figure 9.

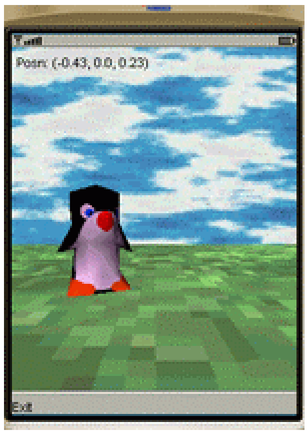

Figure 9. A Grass Floor.

A drawback of the Floor class is that it stretches the image to cover the required area (6 by 6 world units in this case), which can cause pixilation. We'll develop a TiledFloor class in M3G Chapter 4?? which tiles the floor image over the surface, producing more realistic texturing.

The constructor for the Floor class creates a floorMesh object (we'll examine how a bit later), which is accessed by calling Floor's getFloorMesh() method:

```
public Mesh getFloorMesh()
{  return floorMesh;  }
```

getFloorMesh(0 is called by addFloor() in AnimCanvas to get a reference to the floor's mesh, which is then connected to the scene.

3. Animating the Penguin

The "animated" hexagon in Figure 3 is expanded to reveal more detail in Figure 10.

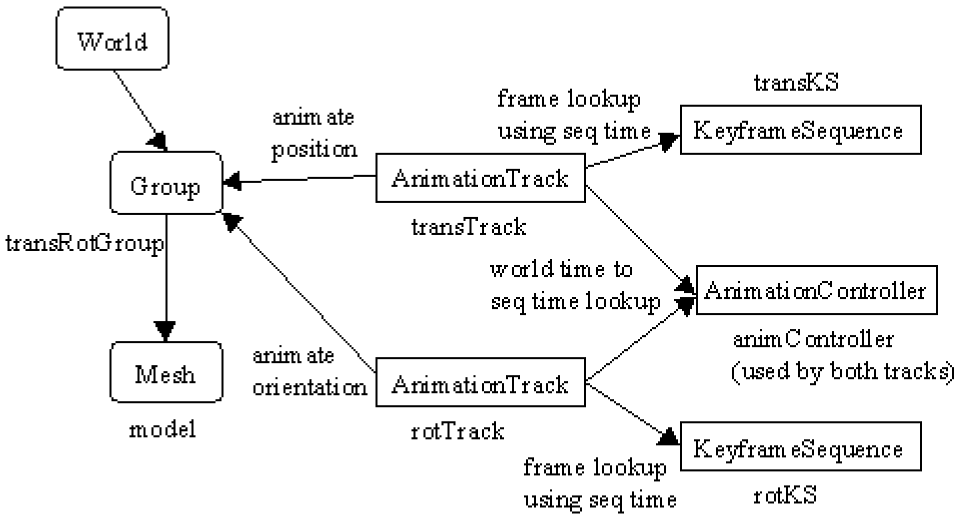

Figure 10. Animation in More Detail.

There are separate animation behaviours for translating and rotating the penguin, both targeting the transRotGroup node. The transTrack AnimationTrack is aimed at the translation component of the node, while the rotTrack AnimationTrack updates its rotational component. The components being modified are called the node's *animation properties*.

There's a large range of node properties open to animation, including colour, lighting, scaling, and morphing weights, although the exact choice depends on the type of node being affected.

An AnimationTrack instance is always associated with one AnimationController and one KeyframeSequence object, and can modify a single kind of animation property. However, a scene graph node may have multiple animation tracks connected to it, with the tracks linked to different animation properties (as here). Multiple tracks may also be combined to change to a single property.

An AnimationTrack object may be linked to multiple scene graph nodes, allowing it to change the same property in several nodes at the same time. A given KeyframeSequence object or AnimationController can be utilized by several AnimationTrack objects at once.

At execution time, an AnimationTrack object is passed a *world* time which it sends onto its AnimationController instance, whose main role is to convert the world time into a *sequence* time. AnimationTrack uses the sequence time to do a lookup in the KeyframeSequence object to find a frame which holds the animation state at that time. AnimationTrack uses the frame information to update the animation property in the target node; any existing data stored in the property is replaced by the new information.

3.1. AnimationController's Time Equation

AnimationController is supplied with a world time (t_w) and converts it to a sequence time (t_s) using the equation:

$$t_s = speed * (t_w – tRef_w) + tRef_s$$

Values for speed, $tRef_w$, and $tRef_s$ can be set by calling methods in the animation controller:

setPosition($tRef_s$, $tRef_w$)

and setSpeed(speed, $speedStartTime_w$)

The $speedStartTime_w$ value states the world time at which the new speed should start being used; typically its value is time 0.

The default values for speed, $tRef_s$, and $tRef_w$ are 1, 0, and 0 respectively, making the sequence time the same as the world time ($t_s == t_w$)

The equation is further constrained to only being valid when t_w is between $startTime_w$ and $endTime_w$. If t_w is outside this range then AnimationTrack will be informed that the animation is inactive. The range is set in AnimationController using:

setActiveInterval($startTime_w$, $endTime_w$)

3.2. Using KeyframeSequence

Once AnimationTrack has a sequence time, it asks its KeyframeSequence object to return the frame information for that time instance.

KeyframeSequence maintains a sequence of frames, as illustrated by Figure 11.

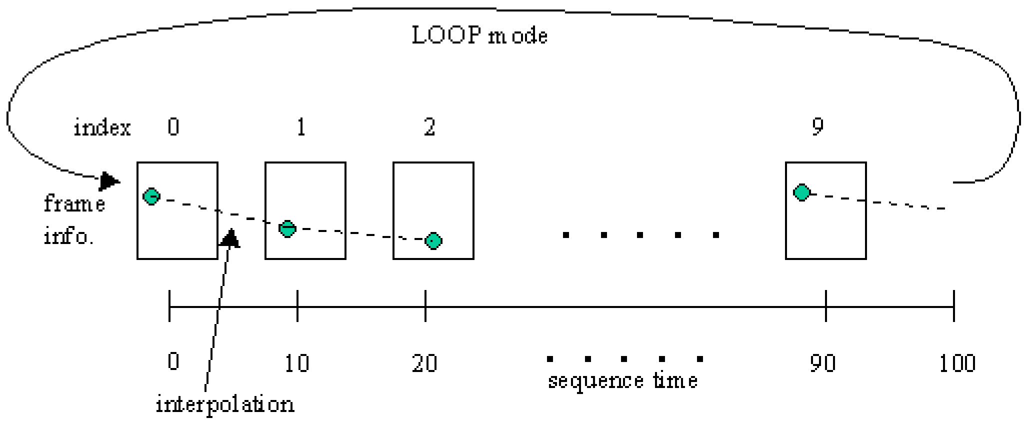

Figure 11. A Keyframe Sequence.

Each frame is actually a scalar or vector of data, used to update the animation property according to the type of that property.

A frame is associated with a time moment in the sequence, and so it's usually necessary to interpolate between two frames to get a frame value for the required time.

A sequence may loop, permitting a time occurring beyond the end of the sequence to be mapped to it, modulo the sequence's duration.

3.3. Initializing the Animation

The animation objects in Figure 10 are created by setUpAnimation() in AnimModel:

```
private void setUpAnimation()
{
  // creation animation controller
  AnimationController animController = new AnimationController();
  animController.setActiveInterval(0, 1500);
  // animController.setSpeed(2.0f, 0);

  // creation translation animation track
  KeyframeSequence transKS = translateFrames();
  AnimationTrack transTrack =
          new AnimationTrack(transKS, AnimationTrack.TRANSLATION);
  transTrack.setController(animController);
  transRotGroup.addAnimationTrack(transTrack);

  // creation rotation animation track
  KeyframeSequence rotKS = rotationFrames();
  AnimationTrack rotTrack =
         new AnimationTrack(rotKS, AnimationTrack.ORIENTATION);
  rotTrack.setController(animController);
```

```
    transRotGroup.addAnimationTrack(rotTrack);
  }  // end of setUpAnimation()
```

The time equation for animController in setUpAnimation() utilizes the default settings for speed and $tRef_w$, and $tRef_s$ (1, 0, and 0 respectively), so the sequence time is the same as the world time ($t_s == t_w$). The call to setActiveInterval() means that the animation will start immediately, and remain active until the world time reaches 1500 time units.

A simple way of speeding up a sluggish animation is with setSpeed(). For example,

```
  animController.setSpeed(2.0f, 0);
```

will change the time equation to:

$$t_s = 2 * t_w$$

This causes the sequence to be updated twice as fast, starting from when $t_w == 0$ (i.e. from the beginning of the animation).

The animation property modified by an animation track is specified in its constructor: AnimationTrack.TRANSLATION for transTrack, and AnimationTrack.ORIENTATION for rotTrack.

3.4. Translation Keyframes

The translation animation makes the penguin move around a circle of radius 0.5 units, centered at the origin on the XZ plane. Figure 12 shows this circle, as viewed from above, together with the frame indices where the information will be stored.

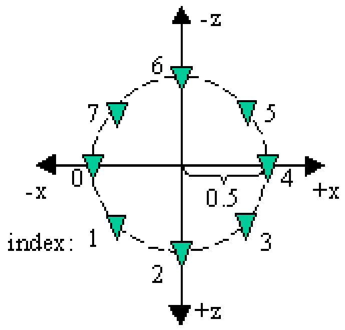

Figure 12. Translation Frame Information for the Penguin.

The first frame (index no. 0) will place the penguin on the negative x-axis at (-0.5, 0, 0). The triangles used in Figure 12 are meant to indicate that the penguin's orientation is unaffected by the translations.

The next step is to map the frame information in Figure 12 into a frame sequence, complete with timing information. The result is Figure 13.

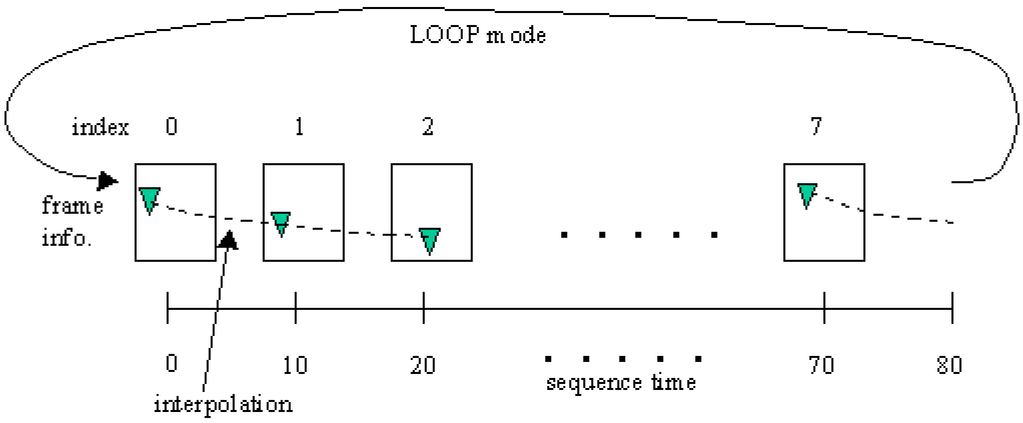

Figure 13. Translation Frame Sequence for the Penguin.

There's an interval of 10 time units between each frame, a total duration of 80 time units, and the sequence loops. The interpolation strategy defines a curved path between the coordinates in the frames.

The corresponding code is in translateFrames():

```
private KeyframeSequence translateFrames()
{
  KeyframeSequence ks =
      new KeyframeSequence(8, 3, KeyframeSequence.SPLINE);
      /* Use a spline to interpolated between the points, so the
         movement is curved */

  // move clockwise in a circle;
  // each frame is separated by 10 sequence time units
  ks.setKeyframe(0, 0, new float[] { -0.5f, 0.0f, 0.0f });
  ks.setKeyframe(1, 10, new float[] { -0.3536f, 0.0f, 0.3536f });
  ks.setKeyframe(2, 20, new float[] { 0.0f, 0.0f, 0.5f });
  ks.setKeyframe(3, 30, new float[] { 0.3536f, 0.0f, 0.3536f });
  ks.setKeyframe(4, 40, new float[] { 0.5f, 0.0f, 0.0f });
  ks.setKeyframe(5, 50, new float[] { 0.3536f, 0.0f, -0.3536f });
  ks.setKeyframe(6, 60, new float[] { 0.0f, 0.0f, -0.5f });
  ks.setKeyframe(7, 70, new float[] { -0.3536f, 0.0f, -0.3536f });

  ks.setDuration(80);   // one cycle takes 80 sequence time units
  ks.setValidRange(0, 7);
  ks.setRepeatMode(KeyframeSequence.LOOP);

  return ks;
}
```

Each frame is a coordinate for the penguin, so the information is encoded as an array of three floats, for the (x,y,z) position.

3.5. Rotation Keyframes

The rotation animation keeps turning the penguin so it always faces the current forward direction as it moves around the circle. This is illustrated in Figure 14.

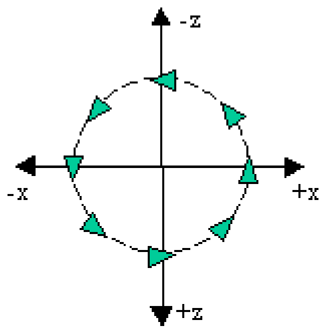

Figure 14. Facing Front as the Penguin Moves.

The rotations are clockwise around the positive y-axis, and the penguin will complete a 360 degree turn by the time it reaches its starting point again.

This behaviour becomes the series of rotations shown in Figure 15, which includes the frame indices where they will appear.

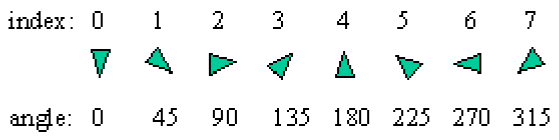

Figure 15. Rotations for Facing Front.

These become the rotation frame sequence in Figure 16.

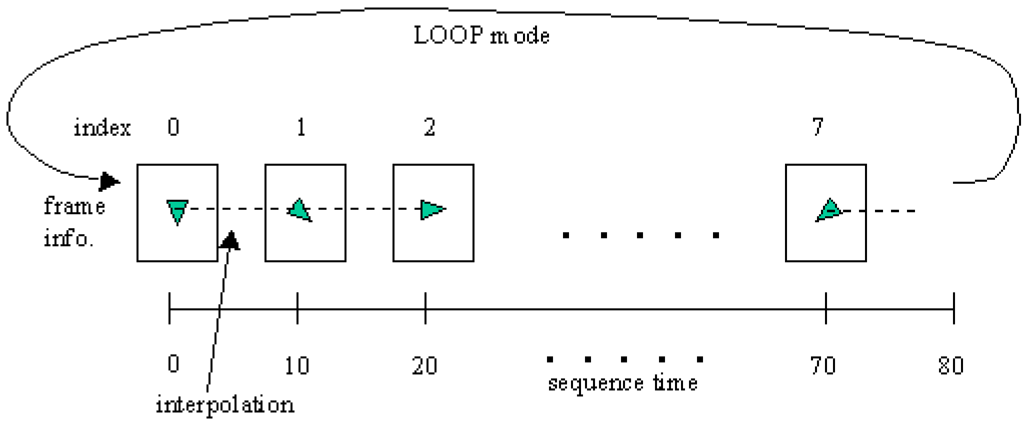

Figure 16. Rotation Frame Sequence for the Penguin.

The time period between each frame, the total duration, and the use of looping matches those for the translation keyframe sequence.

The corresponding code is in rotationFrames():

```
private KeyframeSequence rotationFrames()
{
  KeyframeSequence ks =
    new KeyframeSequence(8, 4, KeyframeSequence.SLERP);
    /* the interpolation is a constant speed rotation */

  // rotate clockwise in a complete loop;
  // each frame is separated by 10 sequence time units
  ks.setKeyframe(0, 0, rotYQuat(0));
  ks.setKeyframe(1, 10, rotYQuat(45));
  ks.setKeyframe(2, 20, rotYQuat(90));
  ks.setKeyframe(3, 30, rotYQuat(135));
  ks.setKeyframe(4, 40, rotYQuat(180));
  ks.setKeyframe(5, 50, rotYQuat(225));
  ks.setKeyframe(6, 60, rotYQuat(270));
  ks.setKeyframe(7, 70, rotYQuat(315));

  ks.setDuration(80);   // one cycle takes 80 sequence time units
  ks.setValidRange(0, 7);
  ks.setRepeatMode(KeyframeSequence.LOOP);

  return ks;
}  // end of rotationFrames()
```

3.6. Rotations Using Quaternions

Rotational animations use quaternions, which are generated by the rotYQuat() method in rotationFrames().

Before we consider quaternions, it's helpful to look at the more intuitive axis-angle method for defining a rotation. The rotation is specified in terms of an angle to turn through about a given axis vector. The axis can be any vector, not just the usual x-, y-, or z- axes. The axis-angle is specified with 4 variables: a unit vector V made up of x, y, and z components, and the angle:

axis-angle = [Vx, Vy, Vz, angle]

There's a simple translation from an axis-angle to a corresponding quaternion q, which also has four elements:

q = [sin(angle/2)*Vx, sin(angle/2)*Vy, sin(angle/2)*Vz, cos(angle/2)]

These four values are often called the i, j, and k coefficients and the scalar component, reading left-to-right.

As an example, consider a rotation of 90 degrees around the y-axis. The axis-angle for this rotation is:

axis-angle = [0, 1, 0, 90]

The equivalent quaternion is:

$$q = [\ 0,\ \frac{\sqrt{2}}{2},\ 0,\ \frac{\sqrt{2}}{2}\ ]$$

since $\cos(45) = \sin(45) = \frac{\sqrt{2}}{2}$.

One question is why bother with quaternions when axis-angles seem to be equivalent, and easier to understand? One reason is the ease of interpolating between rotations expressed as quaternions.

A quaternion can be viewed as a point resting on a 4D sphere (if that seems a tad mind-boggling, then just image a 3D sphere instead). An interpolation between two quaternions follows the path of the *great circle* linking the two points. A great circle in 3D space is the plane cutting through the two points on the sphere's surface and the center of the sphere. More prosaically, the great circle is the shortest distance between the points when we travel over the sphere's surface.

This form of interpolation is known as *slerping* (spherical interpolation), and is utilized by the sequence defined in rotationFrames() above.

An advantage of slerping is that it maintains a constant angular velocity as the quaternion is adjusted during interpolation. This makes the rotation smooth, without sudden changes.

Another benefit of quaternions is that performing successive rotations is simply a matter of multiplying the quaternions together.

A nice introduction to quaternions, with a gaming slant, can be found at:

```
http://www.gamasutra.com/features/19980703/quaternions_01.htm
```

The penguin is only going to rotate around the y-axis, so rotYQuat() can be specialized. In particular, the quaternion's i and k coefficients (the sin(angle)/2*Vx and sin(angle/2)*Vz values) will always be 0. The resulting method:

```
private float[] rotYQuat(double angle)
/* Calculate the quaternion for a clockwise rotation of
   angle degrees about the y-axis. */
{
  double radianAngle = Math.toRadians(angle)/2.0;
  float[] quat = new float[4];
  quat[0] = 0.0f;    // i coefficient
  quat[1] = (float) Math.sin(radianAngle);   // j coef
  quat[2] = 0.0f;    // k coef
  quat[3] = (float) Math.cos(radianAngle);   // scalar component
  return quat;
}
```

rotYQuat() returns a 4-element float array, which becomes the frame information for the given rotation angle. The four elements correspond to the quaternion's i, j, and k coefficients and its scalar component.

3.7. Why Use a transRotGroup Group Node?

The animation tracks target the transRotGroup node rather than the penguin Mesh node beneath it. Why did I do this since a mesh is a Transformable object, and so could be the animation target? The reason is to separate the two forms of translation applied to the mesh. The mesh is being repeatedly translated via animation, but it's also initially translated and scaled when first placed in the scene:

```
  Mesh model = makeModel();
  model.setTranslation(0.25f, 0.25f, 0.25f); // so at center
  model.scale(0.5f, 0.5f, 0.5f);
```

If the translation animation was applied directly to the Mesh object, it would overwrite the existing translation, and the penguin would rotate with half it's body invisible, and offset from the center.

The ability to create hierarchies of nodes is an important benefit of the scene graph mechanism. Transformations carried out on a node apply to all of its descendents as well, but not to its ancestors higher in the graph.

This can be understood by considering your own arm, with the shoulder, elbow, wrist and finger joints taking the roles of transformable nodes linked by bone and sinew into a hierarchy. If you rotate your wrist, it affects your hand and fingers, but not your elbow and shoulder.

It's no surprise that animated articulated 3D figures use node hierarchies extensively, as we'll see in the context of M3G when we examine a skinned mesh example in M3G Chapter 4??.

4. Updating the Application

The animation progresses due to a TimerTask set up in the AnimM3G object. A timer ticks every PERIOD (50) ms and calls the AnimTimer object (which is a subclass of TimerTask). It calls the update() method in the AnimCanvas object, which updates the scene graph's animations and requests a repaint. This sequence is shown in Figure 17.

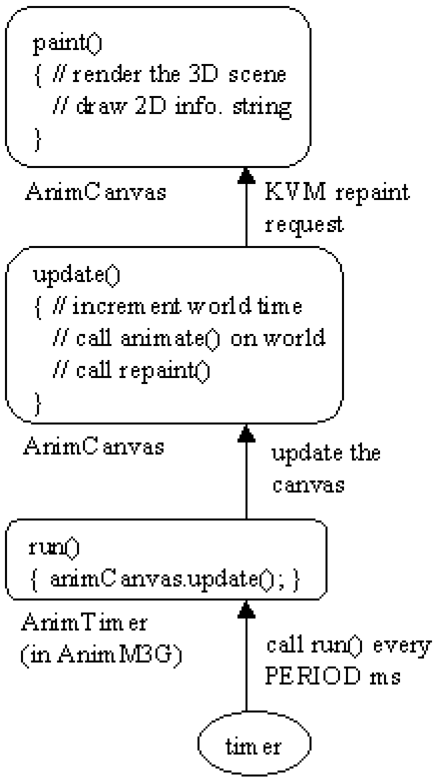

Figure 17. Updating the AnimM3G Application.

The timer and the AnimTimer inner class are defined in the AnimM3G class.

The update() method in AnimCanvas requires some global variables, and the animation needs to be initialized:

```
// global timing information
private int appTime = 0;
private int nextTimeToAnimate;


public AnimCanvas(AnimM3G top)
{ // other code...
  // start the animation
  nextTimeToAnimate = scene.animate(appTime);
}

public void update()
// called by the TimerTask every PERIOD (50) ms
{ appTime++;
  if (appTime >= nextTimeToAnimate) {
    nextTimeToAnimate = scene.animate(appTime) + appTime;
```

```
      repaint();
    }
  }
```

The only change carried out on the scene graph is to trigger an update of the translation and rotation animations. This is achieved by calling animate() on the top-level node, scene. The animation request percolates down through the graph, eventually reaching the transRotGroup node. The animate() argument is passed to the AnimationTrack objects linked to the node as the current world time, and the frame information is updated accordingly.

animate() returns a *validity interval*, the amount of time before another call to animate () will have an effect. We use this to reduce the frequency of the animate() calls, avoiding needless processing.

animate()'s argument acts as the world time, being incremented each time update() is called, every 50ms. This shows that a world time needn't be a time at all; here it's really a counter of the number of timer ticks.

5. Rendering the Scene

The repaint request sent to the KVM from update() will eventually trigger a paint() call. paint() renders the 3D scene, then draws the penguin's coordinates.

```
  private Graphics3D iG3D;  // global

  public AnimCanvas(AnimM3G top)
  { // other code...
    iG3D = Graphics3D.getInstance();
  }


  protected void paint(Graphics g)
  {
    iG3D.bindTarget(g);
    try {
      iG3D.render(scene);
    }
    catch(Exception e)
    { e.printStackTrace(); }
    finally {
      iG3D.releaseTarget();
    }

    // show the model's coordinates
    g.drawString( animModel.getPosition(), 5,5,
                             Graphics.TOP|Graphics.LEFT);
  }
```

The graphics context, g, must be released by the Graphics3D object before MIDP Canvas operations, such as drawString(), can be utilized.

The call to getPosition() in AnimModel returns a string holding the model's transRotGroup coordinates, rounded to 2 decimal places.

```
// globals for examining the model's position
private float transMat[] = new float[16];
private Transform modelTransform = new Transform();
private float xCoord, yCoord, zCoord;


public String getPosition()
{
  transRotGroup.getCompositeTransform(modelTransform);
  modelTransform.get(transMat);

  // store coords rounded to 2 dp
  xCoord = ((int)((transMat[3]+0.005)*100.0f))/100.0f;
  yCoord = ((int)((transMat[7]+0.005)*100.0f))/100.0f;
  zCoord = ((int)((transMat[11]+0.005)*100.0f))/100.0f;

  return "Posn: (" + xCoord + ", " + yCoord + ", " + zCoord + ")";
}
```

The call to getCompositeTransform() combines the node's separate Transformable components – the translation, rotation, scaling and matrix elements – into a single 4 by 4 matrix, which is copied into a one-dimensional transMat[] array. The translation component is stored in the last column of the composite matrix, as the top three values:

$$\begin{bmatrix} r11 & r12 & r13 & tx \\ r21 & r22 & r23 & ty \\ r31 & r32 & r33 & tz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

These correspond to the 4th, 8th, and 12th elements of the transMat[] array.

The translation information is only for the transRotGroup node; it doesn't include any translations carried out by children of the node. For this example, it means that the 0.5 unit translation of the penguin up the y-axis, applied to the Mesh object, is not included.

6. Making the Penguin

The penguin is a Mesh object, which can be represented graphically as Figure 18.

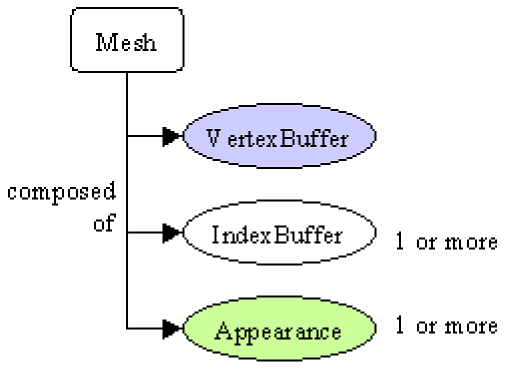

Figure 18. Parts of a Mesh.

VertexBuffer stores various kinds of coordinate information, and the IndexBuffer explains how that information should be divided into triangle strips. There may be multiple IndexBuffers, indicating that the coordinates represent separate submeshes. Each submesh will have its own Appearance node.

The penguin mesh doesn't use submeshs, employing a single VertexBuffer, IndexBuffer and Appearance node, as shown in makeModel() from AnimModel.

```
private Mesh makeModel()
{
  VertexBuffer modelVertBuf = makeGeometry();
  IndexBuffer modelIdxBuf =
         new TriangleStripArray(0, getStripLengths());
  Appearance modelApp = makeAppearance(MODEL_TEXTURE);

  Mesh m = new Mesh(modelVertBuf, modelIdxBuf, modelApp);
  return m;
}
```

The getStripLengths() method is generated by the ObjView application described in the previous chapter. ObjView reads in a Wavefront OBJ file and outputs at most six methods:

- private short[] getVerts()
  return an array holding position vertices;
- private byte[] getNormals()
  return an array holding normals;
- private short[] getTexCoords()
  return an array holding texture coordinates;
- private int[] getStripLengths()
  return an array holding the lengths of each triangle strip;
- private Material setMatColours()
  sets the material colours and shininess;

- private short[] getColourCoords()
  return an array holding colour coordinates.

OBJ files do not support colour coordinates, and so getColourCoords() will never be generated. The functionality to create the method was included in case ObjView is extended in the future.

ObjView outputs the methods to a text file, and then the programmer must cut-and-paste them into his code.

6.1. VertexBuffer Creation

The different forms of information that may be present in a VertexBuffer are illustrated in Figure 19.

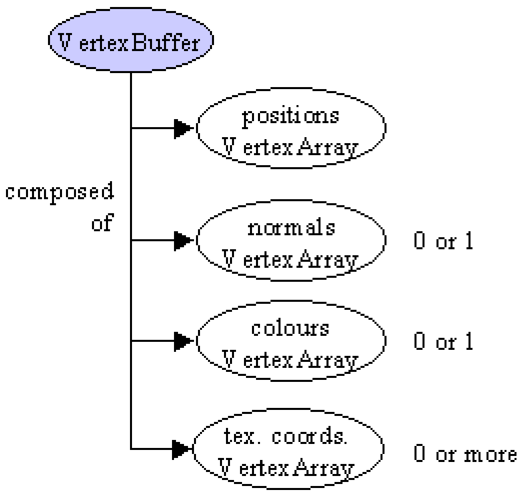

Figure 19. VertexBuffer Elements.

The normals, colours, and texture coordinates are optional, but if they're present then their arrays must represent the same number of coordinates as are in the positions array.

Our makeGeometry() method builds a VertexBuffer using positions, normals, and one array of texture coordinates, obtained from the ObjView-generated getVerts(), getNormals(), and getTexCoords() methods. makeGeometry() is coded in almost the same way as the makeGeometry() method described in the last chapter.

6.2. Appearance Creation

The surface appearance of a model can be composed from many components, as Figure 20 shows.

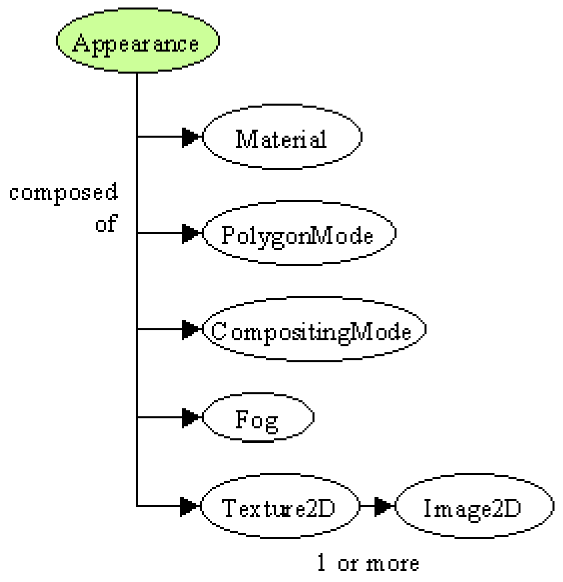

Figure 20. Appearance Elements.

The two most commonly used components are Material and Texture2D: the Material node dictates colour and shininess, while a model's texture (or textures) are handled by Texture2D. Our getAppearance() method uses both, modulating the texture so that the material's colour and shininess will be present. The code is very similar to the getAppearance() method of the previous chapter.

7. Making the Floor

The only purpose of the Floor class is to create a Mesh object representing the floor. The mesh is a square resting on the XZ plane, centered at the origin. The lengths of its sides are specified by a size variable input via the Floor's constructor.

The simplicity of the geometry means that its VertexBuffer can be constructed directly, without the need for ObjView to generate methods from an OBJ model. The floor's vertices are shown in Figure 21, as viewed from above.

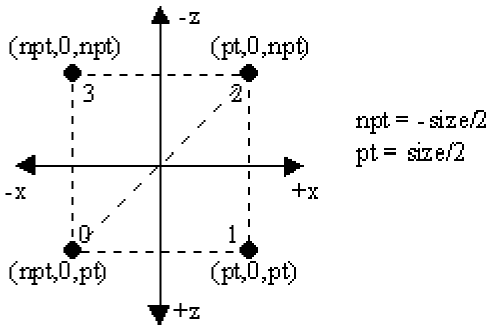

Figure 21. Floor Vertices.

The position coordinates must be stored in their VertexArray in a counter-clockwise order, so the 'front' face of the floor is pointing upwards. The first coordinate can be anything, but the usual convention is to use the bottom-left one. It's labeled as point 0 in the figure.

The IndexBuffer for the mesh must encode the shape as triangle strips. We can represent a square by a single triangle strip made of 4 points: {1,2,0,3}. The initial triangle in the strip, {1,2,0}, is given in anti-clockwise order, so the entire strip will face upwards.

The floor will not reflect light, so will not require normals (or Material information).

The texture coordinates as shown in Figure 22.

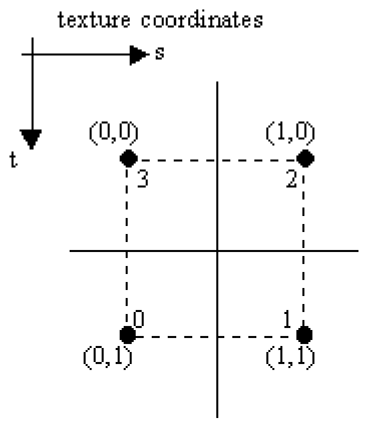

Figure 22. Texture Coordinates for the Floor.

Figure 22 gives the (s,t) texture coordinates for the four points of the floor. In a rather non-standard way, M3G places the (s,t) origin at the top-left rather than the bottom-left, so t increases going down the page.

The coordinates should be stored in their VertexArray in counter-clockwise order, so the texture will face upwards. The first coordinate in the array will be mapped to point 0 in the mesh, and so should be the (0,1) texture coordinate; this corresponds to the bottom-left of the texture image.

The Floor() constructor builds the Mesh using a makeGeometry() method to create the VertexBuffer, and makeAppearance() for the appearance.

```
private Mesh floorMesh;   // global

public Floor(Image2D floorIm, int size)
{
  VertexBuffer floorVertBuf = makeGeometry(size);

  int[] indices = {1,2,0,3};  // one quad
  int[] stripLens = {4};
  IndexBuffer floorIdxBuf =
           new TriangleStripArray(indices, stripLens);

  Appearance floorApp = makeAppearance(floorIm);

  floorMesh = new Mesh(floorVertBuf, floorIdxBuf, floorApp);
}  // end of Floor()
```

makeGeometry() adds positions and texture coordinates to a VertexBuffer:

```
private VertexBuffer makeGeometry(int size)
{
  // create vertices
  short pt = (short)(size/2);
  short npt = (short) (-size/2);   // negative pt
  short[] verts = {npt,0,pt,  pt,0,pt,  pt,0,npt,  npt,0,npt};
  VertexArray va = new VertexArray(verts.length/3, 3, 2);
```

```
    va.set(0, verts.length/3, verts);

    // create texture coordinates
    short[] tcs = {0,1,  1,1,  1,0,  0,0};
    VertexArray texArray = new VertexArray(tcs.length/2, 2, 2);
    texArray.set(0, tcs.length/2, tcs);

    VertexBuffer vb = new VertexBuffer();
    vb.setPositions(va, 1.0f, null); // no scale, bias
    vb.setTexCoords(0, texArray, 1.0f, null);

    return vb;
  }
```

The vertices and texture coordinates are defined according to Figures 21 and 22.

makeAppearance() doesn't need to create a Material node since the floor won't be reflecting light. However, a PolygonMode object is employed for perspective correction and switching off culling.

```
  private Appearance makeAppearance(Image2D floorIm)
  {
    Appearance app = new Appearance();

    if (floorIm != null) {
      Texture2D tex = new Texture2D(floorIm);
      tex.setFiltering(Texture2D.FILTER_NEAREST,
                                      Texture2D.FILTER_NEAREST);
      tex.setWrapping(Texture2D.WRAP_CLAMP, Texture2D.WRAP_CLAMP);

      app.setTexture(0, tex);
    }

    // add perspective correction, and switch off culling
    PolygonMode floorPolygonMode = new PolygonMode();
    floorPolygonMode.setPerspectiveCorrectionEnable(true);
    floorPolygonMode.setCulling(PolygonMode.CULL_NONE);

    app.setPolygonMode(floorPolygonMode);

    return app;
  }  // end of makeAppearance()
```

The perspective correction request may mean that the texture will be more accurately drawn as the floor recedes into the distance, but it depends on the underlying hardware.

The absence of culling will cause the underside of the floor to be displayed (complete with a reversed floor texture). This adds some inefficiency to the rendering, but is useful during game development so the position of objects can be judged from different viewpoints.