M3G Chapter 3. Mighty Morphin' Penguin Flippers

The penguin that dizzily moved in a circle in the previous chapter gets a chance to stand still in this chapter, but the exercise regime continues, with him having to raise and lower his flippers for several minutes. Figure 1 shows two shots of the penguin, one with his flippers up, the other down. At execution time, there's a smooth transition between these extremes.

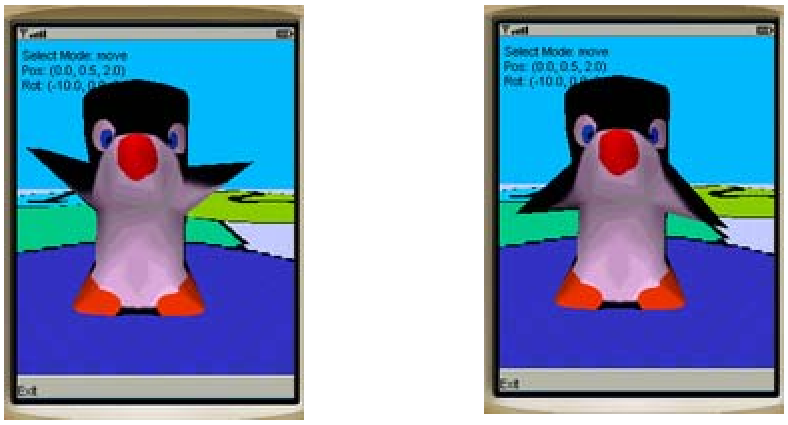
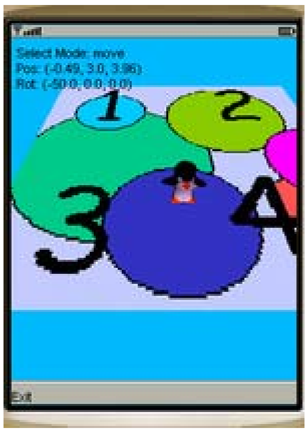


Figure 1. Get them Flippers Flapping: Up, Down, Up,...

The flipper movement is achieved through *mesh morphing*: a penguin *base model* (which has its flipper's extended horizontally) is gradually changed into an almost identical penguinUp *target model* (which has its flippers up). This is morphed back to the base shape, and then to a penguinDown *target model* (flippers down). From the down position, the penguin raises its flippers again by morphing back to the base case.

Although only one penguin is rendered to the screen, three penguin models are used in this application: the base version, a version with the flippers raised, and a version with the flippers lowered.

The other contribution of this example is a MobileCamera class, which allows the camera to be translated forwards, backwards, left, right, up, and down, and to be rotated around the y-axis (i.e. turned left or right), and about the x-axis (i.e. rotated forward or back). Figure 2 shows the camera moved up, left, and back, and rotated to look down towards the floor.



```
Select Mode: move
Pos: (-0.49, 3.0, 3.96)
Rot: (-50.0, 0.0, 0.0)
```

Figure 2. The Scene Viewed From on High.

Details about the mobile camera are written to the top-left of the screen: its current mode (move, rotate, or float), and its position and orientation.

The MobileCamera class can be easily added to other applications to offer camera mobility. A modified version of it is the basis of the First Person Shooter (FPS) example in M3G Chapter 5.

The morphing code in this chapter can be found in the /MorphM3G directory.

1. Class Diagrams for MorphM3G

Figure 3 shows the class diagrams for the MorphM3G application, including public and protected method names.

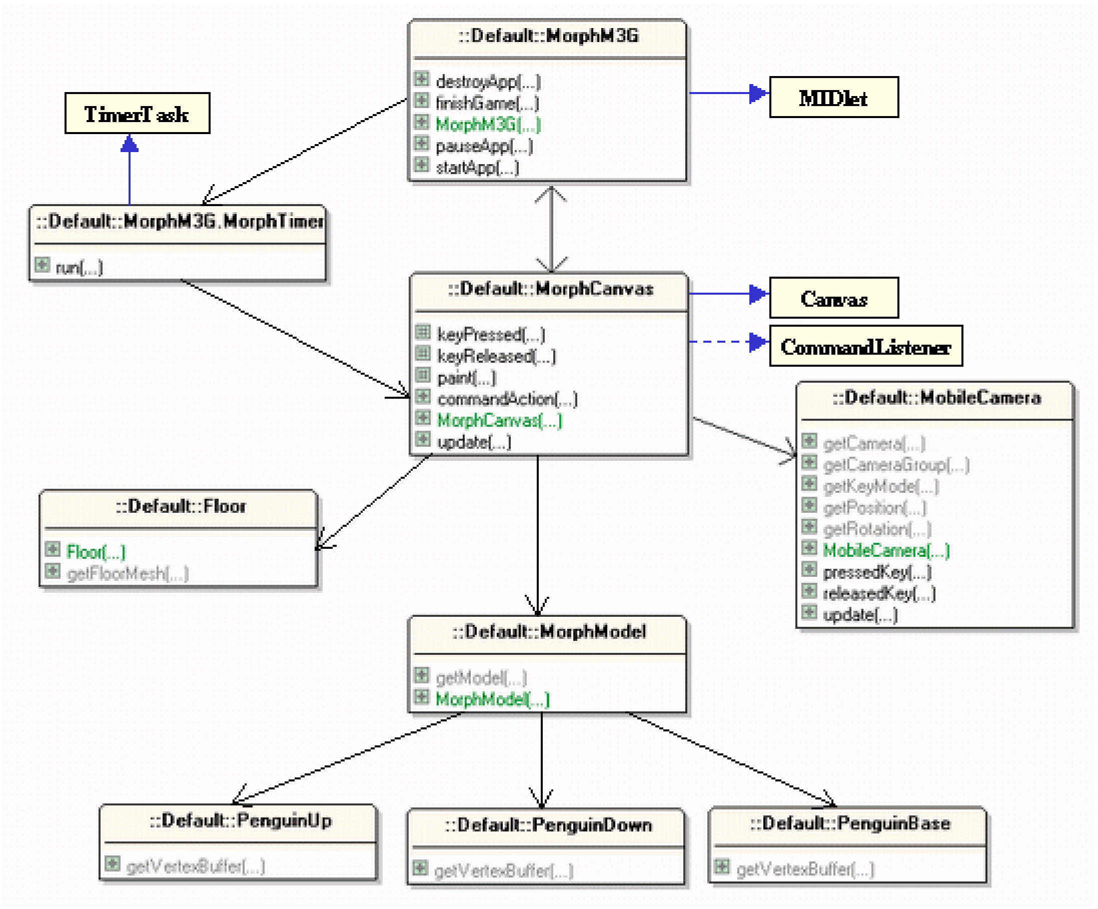


Figure 3. Class Diagrams for MorphM3G.

MorphM3G is the top-level MIDlet, and utilizes a TimerTask subclass to trigger periodic calls to update() in MorphCanvas. Structurally, MorphM3G and MorphTimer are the same as AnimM3G and AnimTimer in the previous chapter.

The Floor class is the same as the one used in the AnimM3G example: it creates a square aligned with the XZ plane, centered at (0,0), with an image wrapped over it as a texture.

MorphModel is in charge of the morphing penguin, utilizing three models stored in the PenguinUp, PenguinDown, and PenguinBase classes. As you'd expect, these represent the penguin with it's flippers up, down, and straight out. MorphModel uses M3G's MorphingMesh class to build the morphing penguin.

2. The MorphCanvas Class

MorphCanvas carries out three tasks: it build the scene graph, passes key presses and releases onto the MobileCamera object, and periodically updates and repaints the scene (in response to calls from MorphTimer). I'll explain each of these below.

2.1. Building the Scene Graph

MorphCanvas creates a 3D scene in retained mode (i.e. it builds a scene graph), in a similar way to AnimM3GCanvas in the previous chapter. The resulting scene graph contains a camera, a light, a blue background, a floor (made by the Floor class), and a morphing penguin constructed by MorphModel. The scene graph is shown graphically in Figure 4.

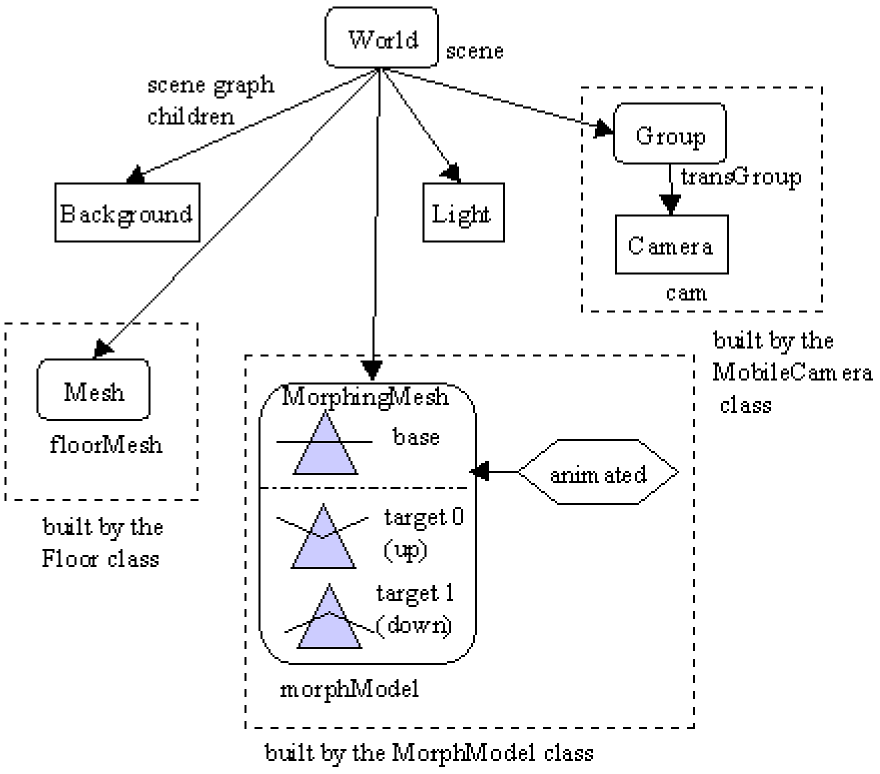


Figure 4. The MorphM3G Scene Graph

The new aspects of the graph are the animated MorphingMesh node built by MorphModel, and the Group and Camera chain built by MobileCamera.

The blue triangles in the MorphingMesh node are my attempt to draw a simplified penguin. The base penguin is the triangle with horizontal lines leaving its sides (these are its flippers). It can be morphed into a penguin with its flippers up (labeled as target 0 in Figure 4), or into a penguin with its flippers down (labeled as target 1). The 'animated' hexagon indicates that the morphing is controlled by an AnimationTrack, AnimationController, and KeyframeSequence triple. The animation varies the 'mix' between the base and the two targets, creating a single mesh where the flippers are moved up or down by some amount.

The camera is tied to the scene via a Group node, called transGroup, so it can be easily moved and rotated. The changes to transGroup and the camera are triggered by the user's key presses.

The World node, called scene, is created in the MorphCanvas constructor, which calls buildScene() to populate the scene graph.

```
private void buildScene()
// add elements to the scene
{
  addCamera();
  addLights();
  addBackground();

  MorphModel mm = new MorphModel();
  scene.addChild( mm.getModel() );  // add the model

  addFloor();
}
```

addLights(), addBackground(), and addFloor() are unchanged from AnimM3G. The MorphModel object will be described presently.

addCamera() is quite short since MobileCamera encapsulates most of its set-up.

```
// global variable
private MobileCamera mobCam;

private void addCamera()
{ mobCam = new MobileCamera( getWidth(), getHeight());

  scene.addChild( mobCam.getCameraGroup() );
  scene.setActiveCamera( mobCam.getCamera() );
}
```

The call to getCameraGroup() returns a reference to the Group node at the top of the camera's scene graph chain, which is attached to the main graph. The getCamera() call retrieves a reference to the Camera object, so it can be made active.

2.2. Key Processing

Since MorphCanvas is a subclass of MIDP's Canvas class, it handles key presses and releases by overriding keyPressed() and keyReleased():

```
protected void keyPressed(int keyCode)
{ int gameAction = getGameAction(keyCode);
  mobCam.pressedKey(gameAction);
}

protected void keyReleased(int keyCode)
{ int gameAction = getGameAction(keyCode);
  mobCam.releasedKey(gameAction);
}
```

The key codes are mapped to game action integers, then passed to the mobile camera for processing.

2.3. Updating and Repainting

MorphCanvas' update() method is called regularly by the MorphTimer object, as detailed in Figure 5.

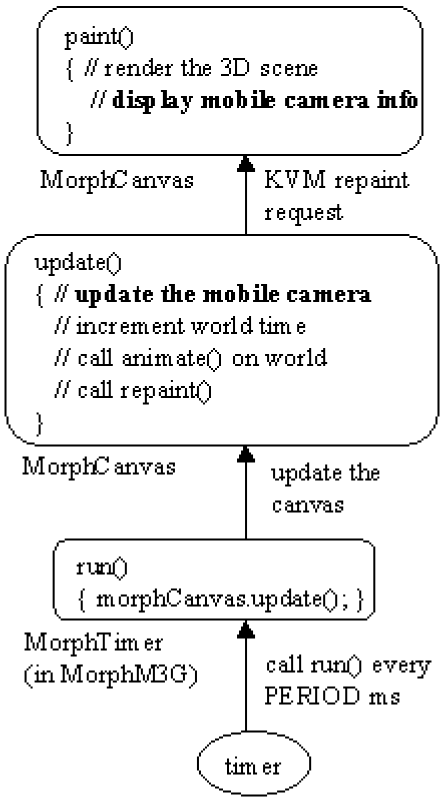


Figure 5. Updating and Repainting MorphM3G.

The update() method:

```
public void update()
{ mobCam.update();
  appTime++;
  if (appTime >= nextTimeToAnimate)
    nextTimeToAnimate = scene.animate(appTime) + appTime;
  repaint();
}
```

A new feature compared to update() in AnimCanvas is the call to MobileCamera's update(). The method updates the camera's position if a key is being held down. In this way, a pressed key will keep the camera moving or rotating.

Another change is that repainting is requested outside the if-block containing the animate() call. This is to deal with the situation when the camera is moved but the penguin isn't animated (morphed). If the repaint() call was inside the if-block, then a camera movement wouldn't necessarily trigger rendering. For example, if the model didn't need animating then the if-test would return false, and no repainting would occur.

paint() is little changed from the version in AnimCanvas, except for three drawString () calls to report on the mobile camera:

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

  // show the camera's details
  g.drawString( mobCam.getKeyMode(), 5,5,
                        Graphics.TOP|Graphics.LEFT);
  g.drawString( mobCam.getPosition(), 5,18,
                        Graphics.TOP|Graphics.LEFT);
  g.drawString( mobCam.getRotation(), 5,31,
                        Graphics.TOP|Graphics.LEFT);
} // end of paint()
```

The drawString() calls must be made after the Graphics3D object, iG3D, has released the graphics context. Only then is the context available to MIDP methods.

3. Creating the Penguin Meshes

MorphM3G uses three penguin models: one acting as the base model (a penguin with its flippers straight out), and two target models (flippers up, flippers down). How are these penguins made?

The base penguin was created first, using the MilkShape 3D modeler. A screenshot of the penguin under construction is shown in Figure 6.

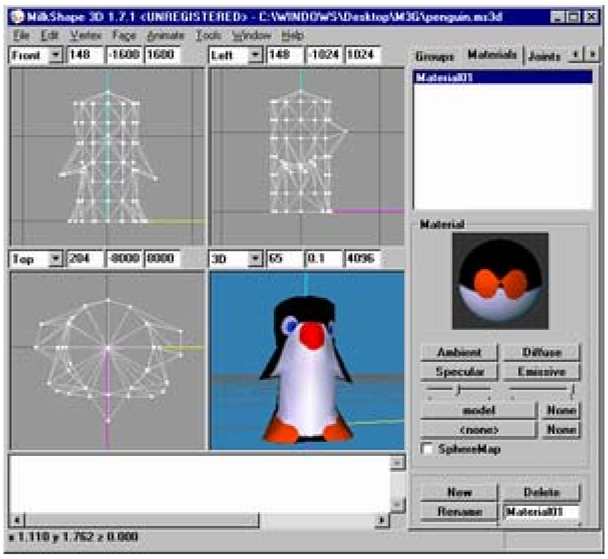


Figure 6. The Penguin in MilkShape 3D.

When finished, the model was exported as an OBJ file, then converted to Java methods using our ObjView utility (see M3G Chapter 1 for information on ObjView).

I returned to the model in MilkShape 3D, and moved the mesh vertices representing the flippers, taking care not to delete or create any vertices. The resulting mesh was saved as a new OBJ file, and converted to Java methods using ObjView. This process was carried out twice, once to create methods for the PenguinUp class, once for PenguinDown.

The overall strategy is illustrated by Figure 7.

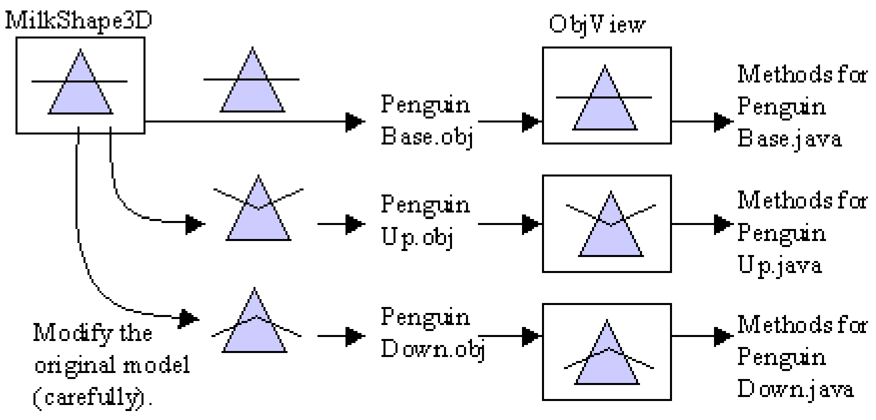

MilkShape3D
ObjView
Penguin Base.obj
Penguin Up.obj
Penguin Down.obj
Methods for Penguin Base.java
Methods for Penguin Up.java
Methods for Penguin Down.java
Modify the original model (carefully).

Figure 7. From Penguin Mesh to Methods.

3.1. The Penguin Model Classes

Three classes were created: PenguinBase, PenguinUp, and PenguinDown, all with the same essential structure aside from their ObjView generated methods. I'll examine PenguinBase; the other two are basically the same.

The only public method in PenguinBase is getVertexBuffer(), which returns a VertexBuffer wrapping up the vertices, normals, and texture coordinates for the penguin.

```
public VertexBuffer getVertexBuffer()
{
  // create vertices
  short[] verts = getVerts();
  VertexArray va = new VertexArray(verts.length / 3, 3, 2);
  va.set(0, verts.length / 3, verts);

  // create normals
  byte[] norms = getNormals();
  VertexArray normArray = new VertexArray(norms.length / 3, 3, 1);
  normArray.set(0, norms.length / 3, norms);

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

  float[] pbias = { (1.0f / 255.0f), (1.0f / 255.0f),
                    (1.0f / 255.0f)};

  VertexBuffer vb = new VertexBuffer();
  vb.setPositions(va, (2.0f / 255.0f), pbias); // scale, bias
  vb.setNormals(normArray);
  vb.setTexCoords(0, texArray, (1.0f / 255.0f), null);

  return vb;
}  // end of getVertexBuffer()
```

The vertices are obtained by calling getVerts(), the normals from getNormals(), and the texture coordinates from getTexCoordsRev() (which calls getTexCoords()). These methods are all generated by ObjView.

getVertexBuffer() also defines the scale and bias for the vertices and texture coordinates. These settings must be the same for all the penguin classes, but ObjView ensures this, by automatically scaling all the penguin models in the same way.

ObjView also generates the getStripLengths() and setMatColours() methods; these are utilized in the MorphModel class described next.

4. Creating the Morphing Penguin

As the class diagrams in Figure 3 indicate, the MorphModel class employs the PenguinBase, PenguinUp, and PenguinDown classes to create the morphing penguin. The resulting scene graph node, an instance of M3G's MorphingMesh class, is shown in Figure 4. MorphModel also sets up the animation mechanism which morphs the penguin model over time.

MorphModel's constructor builds the model and sets up the animation:

```
// global variable
private MorphingMesh morphModel;

public MorphModel()
{
  morphModel = makeMorph();
  // reposition the model's start position and size
  morphModel.setTranslation(0.15f, 0.15f, 0.1f);  // at origin
  morphModel.scale(0.5f, 0.5f, 0.5f);

  setUpAnimation(morphModel);
}
```

The translation and scaling applied to the MorphingMesh positions it at the origin on the XZ plane, and makes it a reasonable size.

4.1. Building a MorphingMesh

A MorphingMesh is a special kind of mesh, which may contain several VertexBuffer objects rather than just the one found in an ordinary Mesh object. The format of a MorphingMesh object is shown in Figure 8.

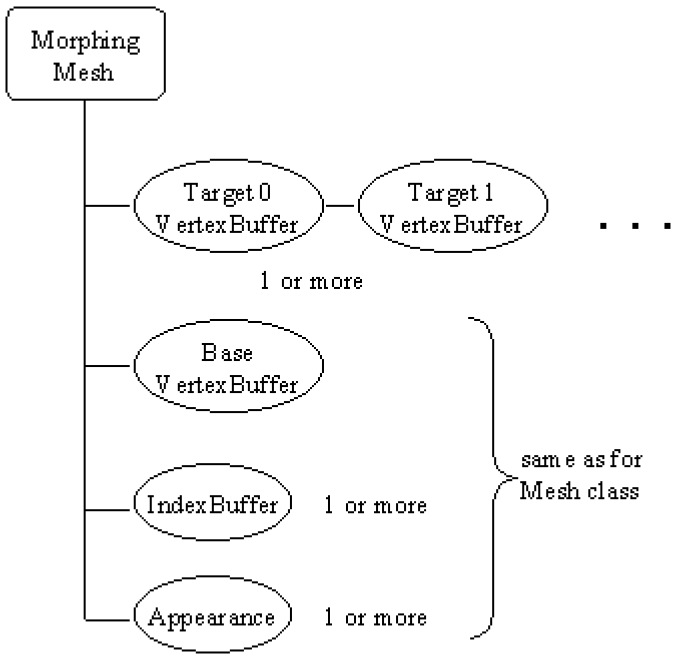


Figure 8. The Components of a MorphingMesh Object.

The additional VetexBuffers are for the morph targets, which hold the shapes that are 'mixed' with the base VertexBuffer to create the mesh that is actually rendered to the screen. The equation for generating the mesh is:

$$\text{resulting\_mesh} = \text{base} + \sum [\ \text{weight}_i * (\text{target}_i - \text{base})\ ]$$

The weights express how much of each target is used in the resulting mesh. Typically the weights are in the range 0 to 1: 0 meaning no contribution, and 1 denoting fully used. However, the weights can be anything, including negative values.

The programmer will usually fix the weight so their sum equals 1, which will stop the resulting mesh from growing larger than the largest VertexBuffer component. However, there's nothing stopping the weights from adding up to a smaller or larger value, but the resulting mesh may look strange.

There are more stringent requirements for the properties of the VertexBuffers. The targets must hold the same types of VertexArrays as the base, contain the same number of vertices in each array as those in the base, the same number of components per vertex, and use the same component sizes. These restrictions can be easily met if the various targets are all generated from the same model as the base, and only differ in the position of some of their mesh vertices.

All the VertexBuffers share the same IndexBuffer and Appearance nodes.

The MorphingMesh object for the penguin is created in makeMorph():

```
// global constants
private static final int NUM_TARGETS = 2;
private static final String MODEL_TEXTURE = "/model.gif";


private MorphingMesh makeMorph()
/* The model base is PenguinBase, and its two targets
   are PenguinUp (flippers up) and PenguinDown (flippers down).
*/
{ VertexBuffer base = (new PenguinBase()).getVertexBuffer();

  VertexBuffer[] targets = new VertexBuffer[NUM_TARGETS];
  targets[0] = (new PenguinUp()).getVertexBuffer();  // PenguinUp
  targets[1] = (new PenguinDown()).getVertexBuffer(); //PenguinDown

  IndexBuffer indBuf = new TriangleStripArray(0,getStripLengths());
  Appearance app = makeAppearance(MODEL_TEXTURE);

  MorphingMesh m = new MorphingMesh(base, targets, indBuf, app);
  return m;
}
```

The VertexBuffers for the base and two targets are obtained from the penguin classes.

The IndexBuffer is created with the help of getStripLength(), which is one of the methods generated by ObjView. There will be three copies of this method to choose from, output for the base and the two targets. If care was taken when the targets were created, the three methods should be the same. If they're not, then a vertex was created or deleted when the targets were formed, and the programmer will have to go back and reconstruct them.

The Appearance node is built with makeAppearance(), which is identical to the one used for the penguin mesh in AnimM3G. The node has a Material node initialized with the setMatColours() method output by ObjView, and modulates (combines) that colour with a texture.

Three copies of the setMatColours() method will be available to choose from, but they should all be the same. It doesn't really matter if they're different, but only one of them can be used in the MorphingMesh.

5. Animating the Penguin

The scene graph in Figure 4 shows the animation part of MorphingModel as a hexagon. Figure 9 replaces the hexagon with the standard triple of AnimationTrack, AnimationController and KeyframeSequence. Details about these classes were given in the previous chapter.

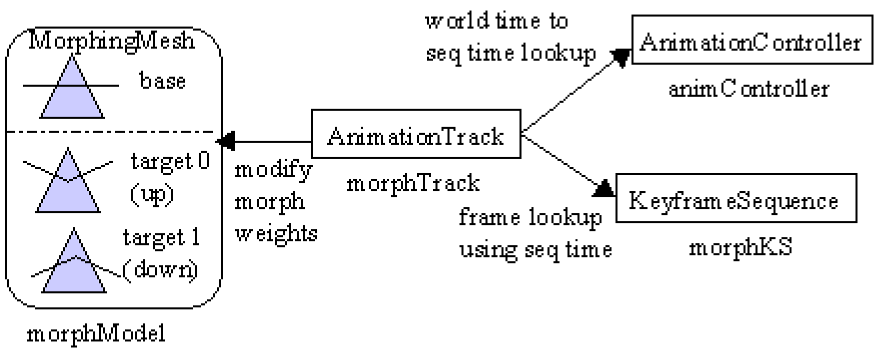


Figure 9. The Animation of the MorphingMesh Object.

The crucial element is the property being affected by the AnimationTrack object; in this example, it's the morph weights of the MorphingMesh (the $weight_i$ values in the equation given above). There are two targets (flippers up, flippers down), so two weights need to be controlled.

Figure 9 is implemented by setUpAnimation() in MorphModel.

```
private void setUpAnimation(MorphingMesh morphModel)
/* The animation adjusts the morph weights to switch between the
   base and the targets. */
{
  // creation animation controller
  AnimationController animController = new AnimationController();
  animController.setActiveInterval(0, 1500);
  animController.setSpeed(4.0f, 0);     // flap quickly

  // creation morph weights animation track
  KeyframeSequence morphKS = morphFrames();
  AnimationTrack morphTrack = new AnimationTrack(morphKS,
                               AnimationTrack.MORPH_WEIGHTS);
  morphTrack.setController(animController);
  morphModel.addAnimationTrack(morphTrack);  // affect morphModel
  }  // end of setUpAnimation()
```

The morph weights are influenced by specifying the `AnimationTrack.MORPH_WEIGHTS` property in the AnimationTrack constructor.

The AnimationController is set to run for 1500 world time units, and the speed setting means that the sequence time runs 4 times faster than the world time, making the penguin flap its flippers quickly.

The most important element of the animation code is the key frame sequence, which is illustrated by Figure 10.

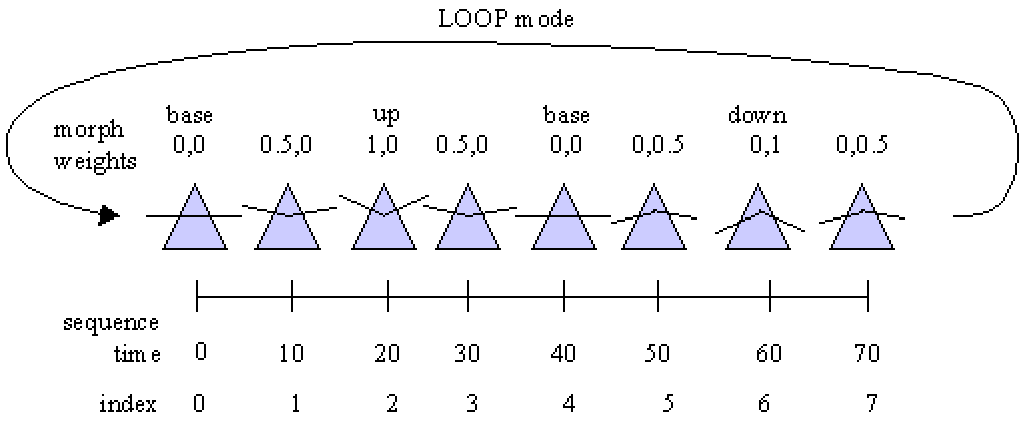


Figure 10. The Key Frame Sequence for the MorphingMesh Animation.

The changes to the two target's morph weights are shown above the triangles representing the penguin. $Weight_0$ is for $target_0$, the penguin with it's flippers up. $Weight_1$ is for $target_1$, the penguin with its flippers down.

As the sequence progresses, the weights are adjusted so the flippers move up from their horizontal position in the base, then down, then back up to be nearly horizontal, before the sequence repeats. When the target weights are both 0, only the base penguin is rendered.

Figure 10 is coded by morphFrames().

```
  private KeyframeSequence morphFrames()
  {
    KeyframeSequence ks =
            new KeyframeSequence(8, 2, KeyframeSequence.LINEAR);

    /* Morphing weights make the penguin's flippers flap up, then
       down, then back to the starting base position.
       First weight is for the up target, the second for the
       down target.
    */
    ks.setKeyframe(0, 0, new float[]  { 0.0f, 0.0f });  // at base
    ks.setKeyframe(1, 10, new float[] { 0.5f, 0.0f });
    ks.setKeyframe(2, 20, new float[] { 1.0f, 0.0f });  // totally up
    ks.setKeyframe(3, 30, new float[] { 0.5f, 0.0f });
    ks.setKeyframe(4, 40, new float[] { 0.0f, 0.0f }); //back at base
    ks.setKeyframe(5, 50, new float[] { 0.0f, 0.5f });
    ks.setKeyframe(6, 60, new float[] { 0.0f, 1.0f }); //totally down
    ks.setKeyframe(7, 70, new float[] { 0.0f, 0.5f });
    ks.setDuration(80);
    ks.setValidRange(0, 7);
    ks.setRepeatMode(KeyframeSequence.LOOP);

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

This method returns the KeyframeSequence object that is utilized by setUpAnimation ().

6. The MobileCamera Class

The mobile camera combines several design aims:

- The main application domain for the camera will be virtual world navigation.
- The camera can move forwards, backwards, left, right, up, and down, and rotate clockwise and anti-clockwise around the y- and x- axes. Rotations around the z-axis, which would allow the camera to swing to the left and right (like a ship rolling at sea), aren't supported; they don't seem necessary for most applications.
- A y-axis rotation affects subsequent translation directions. In other words, if the camera is turned left or right, then 'forward' means the new forward-facing direction, with corresponding changes to the meanings of left, right, and back.
- A z-axis rotation does *not* affect the translation directions. For example, if the camera is rotated towards the floor, then 'up' stills mean straight up the positive y-axis. The motivation for this design is that when the camera is rotated forwards or back, it's to look at something, not to head towards it.
- The camera supports ten operations: six forms of translation (forward, left, right, back, up, and down) and four rotations (y-axis turn left, y-axis turn right, x-axis roll forward, and x-axis roll backwards). We don't want to assign these operations to ten distinct keys; instead, we only utilize the four arrow keys. This is achieved by dividing the operations into three groups: move, rotate, and float. 'Move' contains four operations (left, right, forward, back), 'rotate' has four (the x- and y-axis rotations), and float has two (up and down). These groups (or *modes*, as we call them) are switched between by using the 'select' key on the phone.
- Holding a key down causes the current operation to be repeated.
- The current mode of operation, and position and rotation details for the camera, can be reported to the screen.
- The MobileCamera class should be easy to integrate into different applications.

6.1. Mobile Camera Creation

Figure 4 shows the complete scene graph for the MorphM3G application; the fragment related to the mobile camera is shown again in Figure 11.

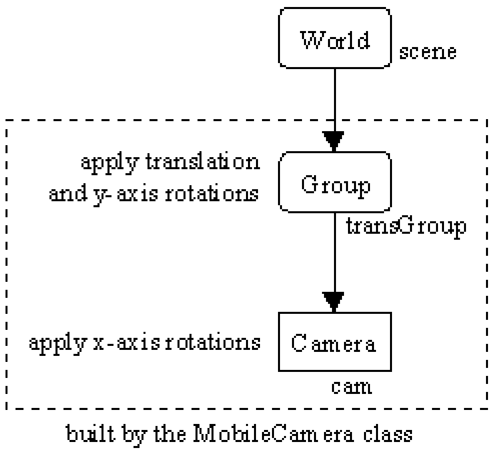

World
scene
apply translation and y-axis rotations
Group
transGroup
apply x-axis rotations
Camera
cam
built by the MobileCamera class

Figure 11. The Scene Graph Branch for the Mobile Camera.

The translation and y-axis rotations are applied to the transGroup Group node, and x-axis rotations operate upon the camera (Camera is a subclass of Transformable, so can be rotated). This separation means that x-axis changes do not affect the camera's translations (and its y-axis rotations), since they occur lower down, in a child node. However, we do want translations to be affected by y-axis rotations, so they're applied to the same node.

The MobileCamera() constructor sets up the scene graph branch, and initializes the position and orientation of the camera.

```
// global constants and variables
// initial x-axis rotation and position
private static final float X_ANGLE = -10.0f;  // degrees
private static final float X_POS = 0.0f;
private static final float Y_POS = 0.5f;
private static final float Z_POS = 2.0f;

// for storing the camera's current x- and y- rotations
private float xAngle, yAngle;

// scene graph elements for the camera
private Transform trans = new Transform();
private Group transGroup;
private Camera cam;


public MobileCamera(int width, int height)
{
  cam = new Camera();
  float aspectRatio = ((float) width) / ((float) height);
  cam.setPerspective(70.0f, aspectRatio, 0.1f, 50.0f);
  cam.postRotate(X_ANGLE, 1.0f, 0, 0);  // apply x-axis rots

  // translation an y-axis rotation group for the model
  transGroup = new Group();
```

```
    trans.postTranslate(X_POS, Y_POS, Z_POS);
    transGroup.setTransform(trans);
    transGroup.addChild(cam);

    // store initial rotations
    xAngle = X_ANGLE; yAngle = 0.0f;
  }  // end of MobileCamera()
```

The camera will face along the negative z- axis by default, but is moved up and back from its default position at the origin, and rotated slightly downwards around the x-axis.

The camera is rotated by calling postRotate() on the orientation component of the node. You may recall from the last chapter that a Transformable node has four components: one for translation, one for orientation, one for scaling, and a 4x4 matrix for general-purpose transforms.

The Group node is translated by having setTransform() set its matrix component. Why use the matrix rather than the node's translation component? The reason is that later we must combine translations and y-axis rotations, which is easier if they're applied through the matrix component.

The Group node and Camera are globals, and can be accessed from MorphCanvas with getCameraGroup() and getCamera().

6.2. Key Presses and Releases

Key presses and releases are caught in MorphCanvas, and their corresponding game action codes are passed to MobileCamera's pressedKey() and releasedKey() methods.

MobileCamera records when keys are pressed and released, by using a set of global booleans. The current operation mode (move, rotate, or float) is also stored.

```
  // global constants and variables
  // key mode constants
  private static final int MOVE = 0;   // move left, right, fwd, back
  private static final int ROTATE = 1;  // turn left, right, up, down
  private static final int FLOAT = 2;   // move up, down
  private static final int NUM_MODES = 3;

  // booleans for remembering key presses
  private boolean upPressed = false;
  private boolean downPressed = false;
  private boolean leftPressed = false;
  private boolean rightPressed = false;

  private int keyMode = MOVE;   // current key mode


  public void pressedKey(int gameAction)
  { switch(gameAction) {
      case Canvas.UP:    upPressed = true;    break;
      case Canvas.DOWN:  downPressed = true;  break;
      case Canvas.LEFT:  leftPressed = true;  break;
      case Canvas.RIGHT: rightPressed = true; break;
      case Canvas.FIRE: keyMode = (keyMode + 1) % NUM_MODES;  break;
      default: break;
```

```
    }
  }

  public void releasedKey(int gameAction)
  { switch(gameAction) {
      case Canvas.UP:    upPressed = false;    break;
      case Canvas.DOWN:  downPressed = false;  break;
      case Canvas.LEFT:  leftPressed = false;  break;
      case Canvas.RIGHT: rightPressed = false; break;
      default: break;
    }
  }
```

The four booleans, upPressed, downPressed, leftPressed, and rightPressed, correspond to the four arrow keys, and are set true when the key is pressed, and changed back to false when the key is released.

The keyMode variable can cycle through three integer values representing the possible operation modes for the camera.

6.3. Updating the Mobile Camera

The reason for remembering key presses, is that the camera's update() method is periodically called, and uses the current key mode and the boolean settings to decide how the camera should be moved or rotated.

Figure 5 shows how MorphTimer triggers update() in MorphCanvas, which then calls update() in the MobileCamera.

The update() method calls a specialized update method depending on the key mode.

```
  public void update()
  { switch(keyMode) {
      case MOVE: updateMove();  break;
      case ROTATE: updateRotation();  break;
      case FLOAT: updateFloating();  break;
      default: break;
    }
  }
```

6.3.1. Updating the Camera's Position

A translation is applied to the matrix component of the transGroup node.

```
  // translation increments
  private static final float MOVE_INCR = 0.1f;


  private void updateMove()
  {
    transGroup.getTransform(trans);
    if (upPressed)            // move forward
      trans.postTranslate(0, 0, -MOVE_INCR);
    else if (downPressed)     // move backwards
      trans.postTranslate(0, 0, MOVE_INCR);
    else if (leftPressed)     // move to the left
```

```
      trans.postTranslate(-MOVE_INCR, 0, 0);
    else if (rightPressed)    // move to the right
      trans.postTranslate(MOVE_INCR, 0, 0);
    transGroup.setTransform(trans);
  }  // end of updateMove()
```

The current transform for the Group node is retrieved by the getTransform() call, and stored in trans. The postTranslate() method 'adds' the translation represented by the key press boolean to the transform. The updated transform is stored back in the Group node with setTransform().

The same coding approach is used in updateFloating(), which uses postTranslate() to move the camera up or down.

6.3.2. Updating the Camera's Orientation

When a camera is rotated around the y-axis, its current location will affect the final position. Figure 12 shows how a triangle's position is changed as it is rotated clockwise around the y-axis from it's starting (x,z) position.

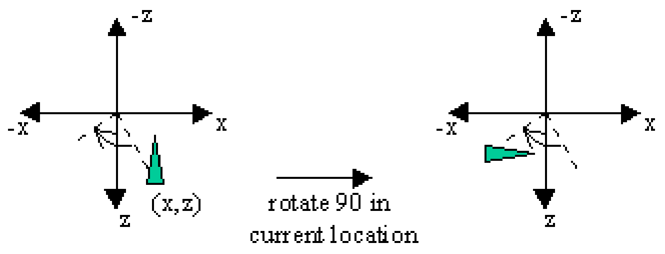


Figure 12. Rotating a Triangle.

This is *not* the behaviour we want for the mobile camera. It's current position should not change when it is rotated.

For the triangle example, it's position is maintained if the rotation is applied in three steps. First, move the triangle to the origin, then rotate it, then move it back to its old position. These operations are illustrated by Figure 13.

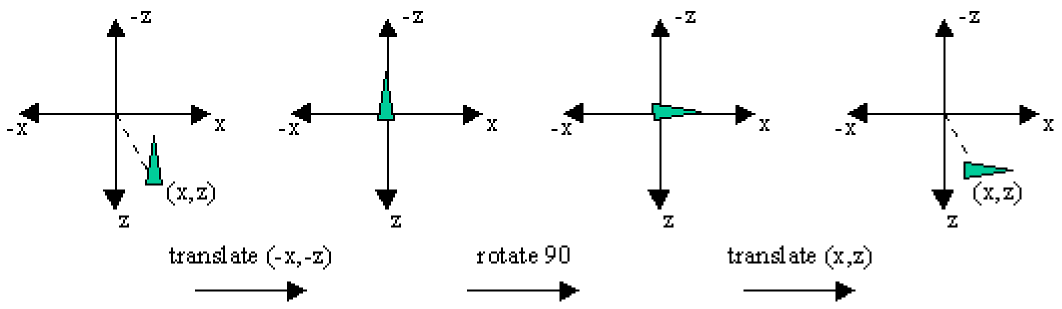


Figure 13. Rotating a Triangle with Extra Translations.

This approach leaves the triangle in the same location, which *is* the behaviour we want for the camera.

One of the difficulties of following this strategy is knowing the current position of the camera (the equivalent of the (x,z) coordinate for the triangle in Figure 13). It's not enough to record the key presses made by the user, since the location of the camera depends on the exact sequence of translations and rotations. For instance, three forward moves by the camera will end at very different spots if there's a rotation of 30 degrees at the start, middle, or end of the sequence.

We obtain the current position of the camera by examining the matrix component of the Group node.

```
  // globals for examining the camera's (x,y,z) position
  private Transform trans = new Transform();
  private float transMat[] = new float[16];
  private float xCoord, yCoord, zCoord;


  private void storePosition()
  // extract the current (x,y,z) position from transGroup
  {
    transGroup.getTransform(trans);
    trans.get(transMat);
    xCoord = transMat[3];
    yCoord = transMat[7];
    zCoord = transMat[11];
  }
```

getTransform() returns the matrix component of the node, which has the form:

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

The camera's current (x,y,z) position are in the 4th, 8th, and 12th elements of the transMat[] array.

storePosition() is called as part of the processing in updateRotation():

```
  private void updateRotation()
  {
    if (upPressed) {          // rotate up around x-axis
      cam.postRotate(ANGLE_INCR, 1.0f, 0, 0);
      xAngle += ANGLE_INCR;
    }
    else if (downPressed) {   // rotate down around x-axis
      cam.postRotate(-ANGLE_INCR, 1.0f, 0, 0);
      xAngle -= ANGLE_INCR;
    }
    else if (leftPressed)     // rotate left around the y-axis
      yAngle += ANGLE_INCR;
    else if (rightPressed)    // rotate right around the y-axis
      yAngle -= ANGLE_INCR;

    // angle values are modulo 360 degrees
    if (xAngle >= 360.0f)
      xAngle -= 360.0f;
    else if (xAngle <= -360.0f)
      xAngle += 360.0f;

    if (yAngle >= 360.0f)
      yAngle -= 360.0f;
    else if (yAngle <= -360.0f)
      yAngle += 360.0f;

    // apply the y-axis rotation to transGroup
    if (leftPressed || rightPressed) {
      storePosition();
      trans.setIdentity();
      trans.postTranslate(xCoord, yCoord, zCoord);
      trans.postRotate(yAngle, 0, 1.0f, 0);
      transGroup.setTransform(trans);
    }
  }  // end of updateRotation()
```

updateRotation() has to deal with rotations around the x-axis as well as those around the y-axis. The x-axis rotations are straightforward to handle: the Camera node is adjusted by calling postRotate().

If the left or right keys are pressed then the yAngle value is updated, and the transGroup node is changed at the end of the method. The trans Transform is set to identity, multiplied by the current camera translation, and then its rotation. When this is applied to transGroup, its new position p' will be:

p' = Trans * Rot

Matrix operations are applied to the node right-to-left, so the node will be rotated by yAngle, then translated by (xCoord, yCoord, zCoord). This corresponds to the last two steps shown in Figure 13. We bypass the need to translate the node to the origin by overwriting the node's old transform (it's old position) with the setTransform() call.

6.4. Reporting

MorphCanvas can call getKeyMode(), getPosition() and getRotation() to obtain the key mode, position, and rotation of the camera. getPosition() is of interest:

```
  public String getPosition()
  {
    storePosition();
    // round coords to 2 dp
    float x = ((int)((xCoord+0.005)*100.0f))/100.0f;
    float y = ((int)((yCoord+0.005)*100.0f))/100.0f;
    float z = ((int)((zCoord+0.005)*100.0f))/100.0f;
    return "Pos: (" + x + ", " + y + ", " + z + ")";
  }
```

The call to storePosition() populates the xCoord, yCoord, and zCoord global variables with the latest position for the camera. A string is constructed which rounds these numbers to two decimal places, and is then sent back to the caller.