M3G Chapter 4. A Bending Block in a Landscape

This chapter explains how to build a bending coloured block, using the SkinnedMesh class. SkinnedMesh has a 'skin' and 'skeleton' – as the skeleton moves, the skin goes with it.

The other topic of this chapter is how to create an interesting landscape, by improving the texturing of the ground, adding trees, and animating the sky.

The Floor class, used in the preceding chapters, is replaced by TiledFloor, which *tiles* a texture over the surface, making the ground appear more detailed.

Trees are included in the scene, not implemented with 3D models, but as 2D images drawn on flat screens (billboards), which always stay oriented towards the camera. This orientation means that the tree images are visible from every angle, supporting the illusion of 3D solidity. I use M3G's Sprite3D class for some of the trees, but also utilizes my own Billboard class. Billboard employs a partially transparent mesh and camera alignment, techniques that we'll need (and extend) in M3G chapter 5.

The MobileCamera class from M3G chapter 3 (with some minor changes) moves the camera. As the camera changes position and orientation, the background (a blue sky with clouds) also moves. The background is controlled through a MobileBackground class.

Figure 1 shows two shots of the bendy block taken at different times, from different camera angles.

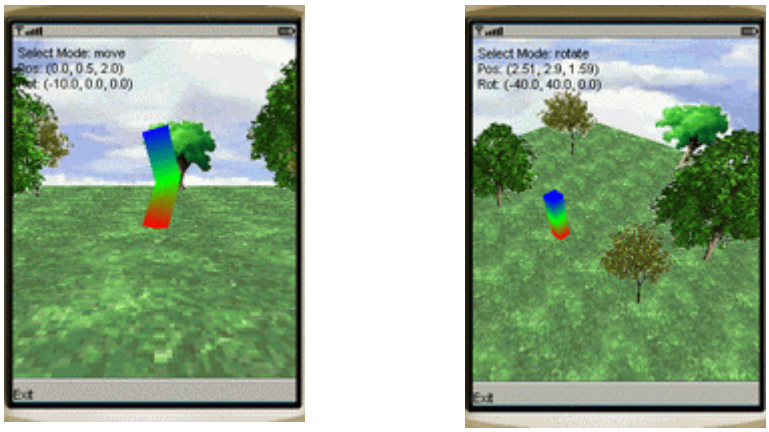

Figure 1. Two Views of the Bendy Block.

The block uses vertex colouring: the vertices in its base are red, the middle ones are green, and the ones in its top are blue. The bending is achieved through rotations around the block's z-axis; one rotation point is in the block's base, the other in its middle. I'll go into the details later.

The two screenshots in Figure 1 show that the five 2D tree images keep facing the camera, even after the camera has been moved. Also, the cloudy sky background is different between the two shots. The ground (a tiled 'grass' image) is more detailed than the grass surface used in M3G chapters 2 and 3.

The application is stored in the SkinM3G/ directory.

1. Class Diagrams for SkinM3G

Figure 2 gives the class diagrams for all the classes in the SkinM3G application. The public and protected methods are shown.

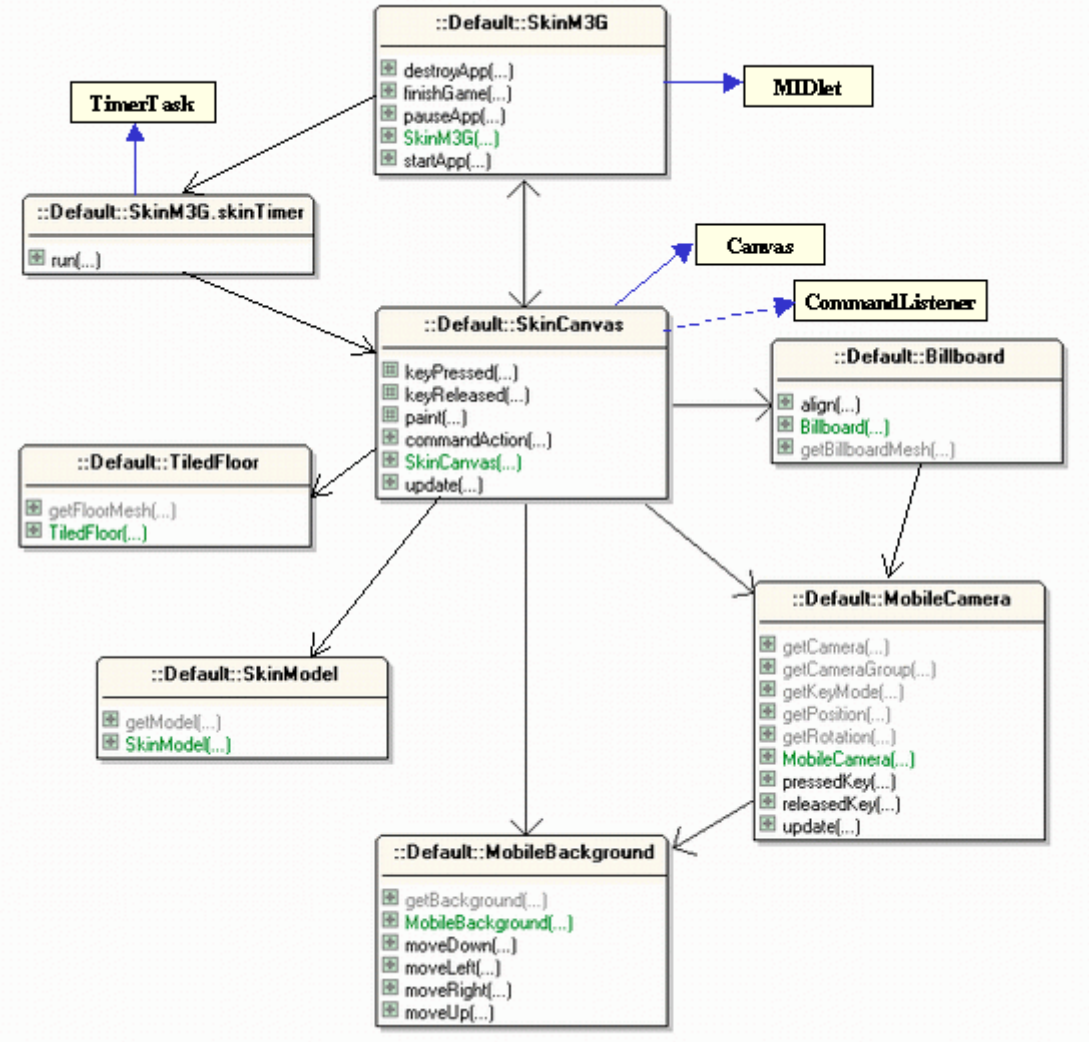

Figure 2. Class Diagrams for SkinM3G.

SkinM3G is the top-level MIDlet, and uses a TimerTask inner class to keep triggering the update() method in SkinCanvas, which updates the scene and redraws it.

SkinCanvas builds the scene: TiledFloor handles the floor, SkinModel is responsible for the bendy block and its animation. The scene's five trees are implemented in SkinCanvas using M3G's Sprite3D class for two of them, and the Billboard class for the others. The cloudy blue sky is moved by MobileBackground when the user moves or rotates the camera in MobileCamera.

2. Scene Creation

The scene is constructed by SkinCanvas in buildScene():

```
// global variable
private World scene;


private void buildScene()
{ addBackground();
  addCamera();
  addTrees();

  SkinModel sm = new SkinModel();
  scene.addChild( sm.getModel() );  // add the model

  addFloor();
}
```

The construction of the background, camera, and trees must be carried out in the order used in buildScene(). The MobileCamera object in addCamera() requires a reference to the MobileBackground object, and the Billboard objects, used for three of the trees in addTrees(), requires a reference to the MobileCamera.

There's no Light nodes in the scene, since none of the objects reflect light. In earlier examples, the scene's 3D model was textured and lit, but the bendy block in SkinM3G utilizes vertex colouring instead.

The resulting scene graph is shown in Figure 3.

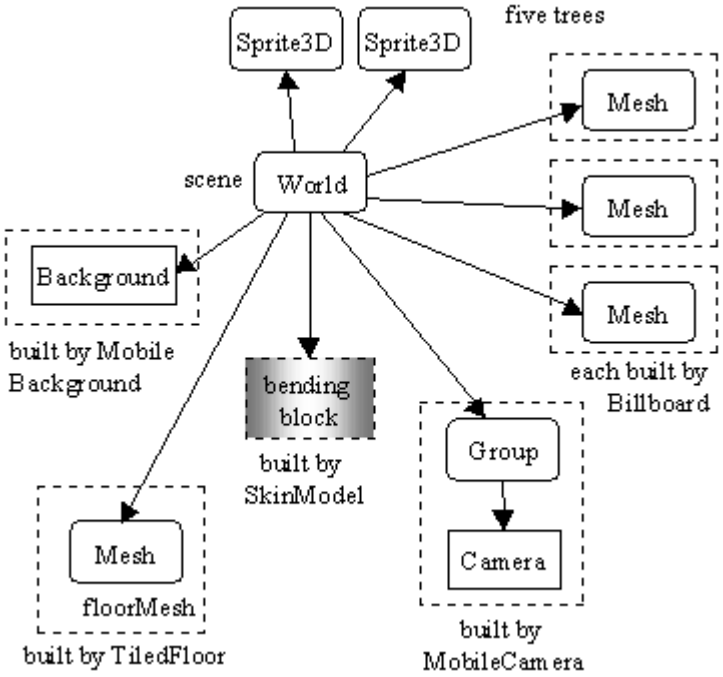

Figure 3. Scene Graph for SkinM3G.

Figure 3 doesn't include details for the bending block made by the SkinModel class; that part of the scene graph will be discussed later.

3. The Mobile Background

addBackground() creates a MobileBackground object, and obtains a reference to its Background object via getBackground().

```
// global
private MobileBackground mobileBackGnd;

private void addBackground()
{
  Image2D backIm = loadImage("/clouds.gif");
  mobileBackGnd = new MobileBackground(backIm,
                                  getWidth(), getHeight());
  scene.setBackground( mobileBackGnd.getBackground() );
}
```

The "clouds.gif" image is shown in Figure 4. Unlike textures, the width and height of a background image does not have to be a power of two.

Figure 4. The "clouds.gif" Image.

The MobileBackground constructor tiles the background image in the horizontal and vertical directions, and sets a cropping rectangle to the size of the screen.

```
// global variables in MobileBackground
private Background backGnd;
private boolean isMoveable = false;

private int width, height;   // of the screen


public MobileBackground(Image2D backIm, int w, int h)
{
  width = w;  height = h;
  backGnd = new Background();

  if (backIm != null) {
    backGnd.setImage( backIm );
    backGnd.setImageMode(Background.REPEAT, Background.REPEAT);
    backGnd.setCrop(backGnd.getCropX(), backGnd.getCropY(),
                                          width, height);
    isMoveable = true;
  }
  else
    backGnd.setColor(0x00bffe); // a light blue background
}
```


If the image isn't available for some reason, then the background defaults to light blue. The isMoveable boolean is set true only if the background is a tiled image.

MobileBackground offers four movement methods for shifting the background right, left, up, and down, which are utilized by MobileCamera. These methods work by adjusting the cropping rectangle's top-left coordinate.

```
private final static int STEP = 3;


public void moveRight(int factor)
{ if (isMoveable)
    backGnd.setCrop(backGnd.getCropX()-(STEP*factor),
                    backGnd.getCropY(), width, height);
}

public void moveLeft(int factor)
{ if (isMoveable)
    backGnd.setCrop(backGnd.getCropX()+(STEP*factor),
                    backGnd.getCropY(), width, height);
}

public void moveUp(int factor)
{ if (isMoveable)
    backGnd.setCrop(backGnd.getCropX(),
              backGnd.getCropY()+(STEP*factor), width, height);
}

public void moveDown(int factor)
{ if (isMoveable)
    backGnd.setCrop(backGnd.getCropX(),
             backGnd.getCropY()-(STEP*factor), width, height);
}
```

Consider moveRight(): the intention is to make the clouds shift right, and this is achieved by *subtracting* `STEP*factor` from the crop rectangle's x-coordinate. An example of this process is given in Figure 5.

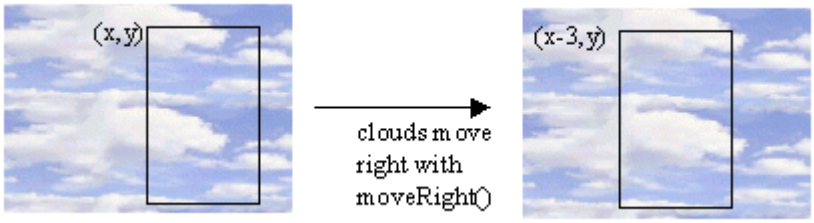

Figure 5. Moving the Clouds to the Right.

An examination of the crop rectangles in Figure 5 shows that the clouds have 'moved' right, even though it's really the crop area that has shifted left.

The other move methods use the same principle. For instance, moveUp() moves the crop rectangle *downwards* by incrementing its y-coordinate.

4. The Mobile Camera

A MobileCamera object is created by addCamera() in SkinCanvas:

```
// globals
private MobileCamera mobCam;
private Group cameraGroup;


private void addCamera()
{
  mobCam = new MobileCamera( getWidth(), getHeight(),
                                              mobileBackGnd);
   /* The camera needs a reference to the MobileBackground
      object, so it can tell the background to move. */

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

This version of the MobileCamera class is almost the same as the one in MorphM3G (in M3G chapter 3), except that it communicates with the MobileBackground object to move the background.

The coding additions are in MobileCamera's update methods: updateMove(), updateRotation(), and updateFloating(). The basic idea is that as the camera moves (or rotates) in one direction, then the background is translated in the other. For example, in updateMove() and updateRotation():

```
private void updateMove()
{
  // ...
  else if (leftPressed) {   // move to the left
    trans.postTranslate(-MOVE_INCR, 0, 0);
    mobileBackGnd.moveRight(1);    // move background to the right
  }
  // ...
}  // end of updateMove()


private void updateRotation()
{
  if (upPressed) {  // rotate up around x-axis
    cam.postRotate(ANGLE_INCR, 1.0f, 0, 0);
    xAngle += ANGLE_INCR;
    mobileBackGnd.moveDown(3);  // background moves down
  }
  else // ...
}  // end of updateRotation()
```

The background is translated further when the camera rotates simply because it looks better. The underlying aim of this code is not to simulate the real world in all its glory, but to emphasize camera movement in a fast (fairly believable) manner.

The background isn't adjusted when the camera moves forward or back, although it would be possible to enlarge or shrink the image by reducing or expanded the size of the crop rectangle.

5. Trees in the Orchard

Five trees are added to the scene by addTrees() in SkinCanvas:

```
// globals: tree billboards
private Billboard bb1, bb2, bb3;

private void addTrees()
{
  Sprite3D ts1 = makeSprite("/tree1.gif", 1.5f, 0, 1.0f);
  scene.addChild( ts1 );

  Sprite3D ts2 = makeSprite("/tree2.gif", -1.5f, 0, 1.5f);
  scene.addChild( ts2 );

  Image2D bbIm1 = loadImage("/tree2.gif");
  bb1 = new Billboard(bbIm1, cameraGroup, 2.0f, -1.0f, 2.0f);
                          // needs a ref. to cameraGroup
  scene.addChild( bb1.getBillboardMesh() );

  Image2D bbIm2 = loadImage("/tree3.gif");
  bb2 = new Billboard(bbIm2, cameraGroup, -2.0f, -2.0f, 1.5f);
                          // needs a ref. to cameraGroup
  scene.addChild( bb2.getBillboardMesh() );

  Image2D bbIm3 = loadImage("/tree4.gif");
  bb3 = new Billboard(bbIm3, cameraGroup, 0.5f, -3.2f, 1.5f);
                          // needs a ref. to cameraGroup
  scene.addChild( bb3.getBillboardMesh() );
}
```

Two of the trees are drawn on the faces of Sprite3D objects, while the other three use my Billboard class. The Sprite3D class is sufficient for our needs in this application, so the main reason for introducing a Billboard class is to show how to implement a partially transparent mesh and camera alignment. We'll build on these techniques in M3G chapter 5.

5.1. Sprite3D Trees

SkinCanvas uses makeSprite() to create a Sprite3D object resting on the floor, centered at (x,z), with sides of length `size`. A texture is wrapped over the sprite's face, but any transparent parts of the image aren't displayed.

```
private Sprite3D makeSprite(String treeFnm,
                            float x, float z, float size)
{
  Image2D image2D = loadImage(treeFnm);

  // don't display the transparent parts of the image
  Appearance app = new Appearance();
  CompositingMode compMode = new CompositingMode();
```



```
    compMode.setBlending(CompositingMode.ALPHA);
    app.setCompositingMode(compMode);

    Sprite3D sprite = new Sprite3D(true, image2D, app);
       // create a unit-size sprite

    sprite.scale(size, size, size);  // fix size
    sprite.setTranslation(x, (0.5f * size), z);
             // move it up to rest on the floor

    return sprite;
  }  // end of makeSprite()
```

A Sprite3D object is a 2D screen parallel to the XY plane. It can be positioned anywhere in the scene, and will automatically rotate to align itself with the screen's viewpoint. The sprite will always face the camera, even if the camera moves around, as it does in the SkinM3G application.

The first argument of the Sprite3D constructor is a boolean which specifies whether is should be scaled to be a 1-by-1 unit square (when the argument is true), or be left unscaled.

In makeSprite(), the Sprite3D object is set to unit size, then scaled and positioned according to the method's input arguments. By default, a sprite's center is placed at (0,0) on the XZ plane, so our 'tree' sprites must be lifted by `0.5*size` units to make them rest on the floor.

The Sprite3D class only utilizes Appearance and Image2D nodes, the latter for the image draped over its face. Many Appearance attributes have no effect on Sprite3D objects, but CompositingMode is understood: it's blending functions allow the RGB colours and alpha of the image to be combined with those of the sprite. The `CompositingMode.ALPHA` mode ensures that the sprite is invisible wherever the source image is transparent.

makeSprite() can be seen in action, by considering the call in addTrees():

```
Sprite3D ts2 = makeSprite("/tree2.gif", -1.5f, 0, 1.5f);
```

The ts2 object is placed at (-1.5, 0) on the XZ plane, and appears as in Figure 6. It uses the image in "tree2.gif".

Figure 6. A Sprite3D Object using "tree2.gif".

The contents of "tree2.gif" are shown in Figure 7.

Figure 7. The Contents of "tree2.gif".

The alpha parts of the image aren't rendered onto the sprite.

A Sprite3D object will always stay aligned with the camera, which for our application, can produce some strange effects. Figure 8 shows the camera pointing straight down from above the floor.

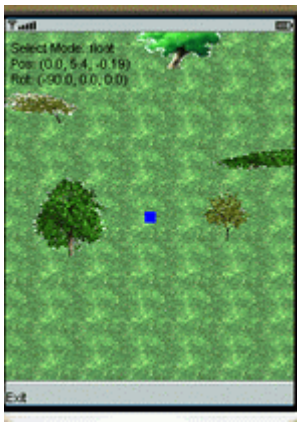

Figure 8. The Landscape Viewed from Above.

The two trees in the center of Figure 8 are Sprite3D objects, which are aligned with the camera, so appear to be lying on the ground. The three trees in the upper half of the image use my Billboard class, which doesn't rotate about the x-axis when the camera moves up or down.

5.2. Billboard Trees

addTrees() creates three Billboard trees. For example, the one at (2,-1) on the XZ plane:

```
Image2D bbIm1 = loadImage("/tree2.gif");
bb1 = new Billboard(bbIm1, cameraGroup, 2.0f, -1.0f, 2.0f);
                          // needs a ref. to cameraGroup
scene.addChild( bb1.getBillboardMesh() );
```

Billboard creates a square mesh of a specified length, resting on the XZ plane, centered at (x,z). Any transparent parts of the attached image aren't rendered. The Billboard will stay aligned with the camera along its z-axis.

The Billboard constructor builds a mesh, and sets its position, size, and alignment criteria. The MobileCamera Group node (cameraGroup) passed to the method is used at run time to align the billboard with the camera.

```
// globals
private Mesh bbMesh;
private Group cameraGroup;


public Billboard(Image2D bbImage, Group camGroup,
                             float x, float z, float size)
{ cameraGroup = camGroup;

  // build bbMesh
  VertexBuffer bbVertBuf = makeGeometry();

  int[] indicies = {1,2,0,3};  // the billboard is one quad
  int[] stripLens = {4};
  IndexBuffer bbIdxBuf =
            new TriangleStripArray(indicies, stripLens);

  Appearance bbApp = makeAppearance(bbImage);

  bbMesh = new Mesh(bbVertBuf, bbIdxBuf, bbApp);

  float size2 = size * 0.5f;
     /* The mesh is 2-by-2 in size, and so the extra 0.5 factor
        in the scaling reduces it to 1-by-1. */
  bbMesh.scale(size2, size2, size2);
  bbMesh.setTranslation(x, size2, z);

  bbMesh.setAlignment(cameraGroup, Node.Z_AXIS, null, Node.NONE);
       /* The billboard alignment will be along its z-axis only,
          no y-axis alignment is employed. */
}  // end of Billboard()
```


The mesh is made of four points placed around the origin on the XY plane. Figure 9 shows the points, and their numbering scheme, which starts at the bottom-left corner, and proceeds in a counter-clockwise direction.

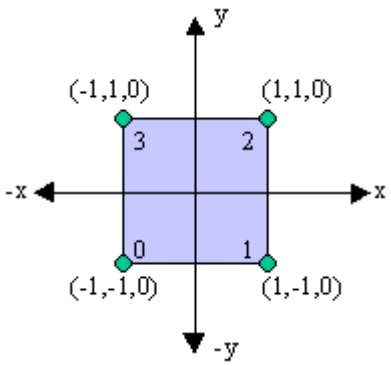

Figure 9. Billboard Object Coordinates.

This mesh is represented by two triangles in a triangle strip using the coordinate sequence {1,2,0,3}.

The call to setAlignment() in the constructor sets the mesh's alignment rules along its z- and y- axes:

```
bbMesh.setAlignment(cameraGroup, Node.Z_AXIS, null, Node.NONE);
```

The first two arguments of setAlignment() determine the mesh's z-axis alignment, the latter two its y-axis orientation.

The billboard's mesh has its z-axis aligned with the z-axis of the cameraGroup node. No node is given for the mesh's y-axis alignment, since `Node.NONE` switches it off.

The call to setAlignment() is *not* enough to keep the Billboard object aligned with the moving camera. The billboard's orientation will only be updated when the align() method is called; for example:

```
bbMesh.align(cameraGroup);
```

This will apply the alignment rules using the current position of the cameraGroup.

The align() method must be called whenever the camera is moved, or the alignment will not stay current. The call is carried out from the update() method of SkinCanvas, as shown later.

5.2.1. The Billboard's Appearance

An image is plastered over the face of the Billboard, but any transparent bits aren't displayed.

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

  // The alpha component of the texture will be used
```



```
    CompositingMode compMode = new CompositingMode();
    compMode.setBlending(CompositingMode.ALPHA);
    app.setCompositingMode(compMode);

    if (bbImage != null) {
      Texture2D tex = new Texture2D(bbImage);
      tex.setFiltering(Texture2D.FILTER_NEAREST,
                              Texture2D.FILTER_NEAREST);
      tex.setWrapping(Texture2D.WRAP_CLAMP, Texture2D.WRAP_CLAMP);
      tex.setBlending(Texture2D.FUNC_REPLACE);
                    // the texture with replace the surface

      app.setTexture(0, tex);
    }
    return app;
  }  // end of makeAppearance()
```

The CompositingMode object brings the alpha component of the texture into the rendering, in the same way as in makeSprite() for the Sprite3D object.

Unlike the Sprite3D case, a Texture2D object must also be configured so that its colour and alpha parts will replace any colour or alpha elements of the mesh. This means that the mesh will be transparent in the same places as the texture.

6. The Tiled Floor

The TiledFloor class divides the floor area into 1-by-1 unit tiles, and the texture is applied to each one individually. This makes the surface considerably more detailed than in the Floor class, which stretches a single texture over the entire area. Figure 10 presents the same view of the scene twice: the screenshot on the left shows TiledFloor in SkinM3G, while the picture on the right is when it employs Floor.

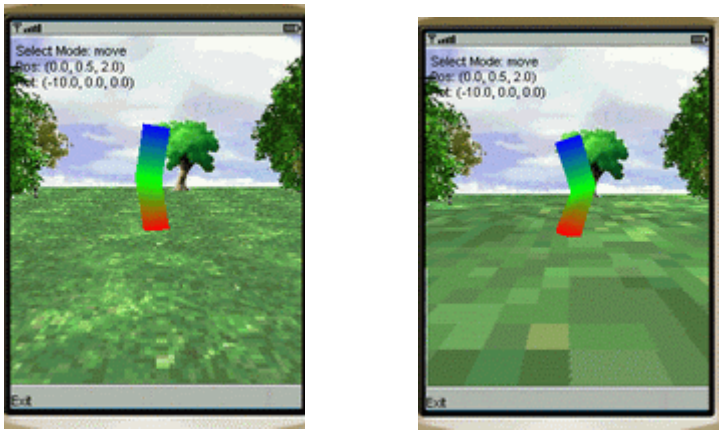

Figure 10. TiledFloor versus Floor.

SkinCanvas calls addFloor() to create a grass-covered floor whose total surface area is 8-by-8 units, which means that 64 tiles are utilized (8*8).

```
private void addFloor()
{ Image2D floorIm = loadImage("/grass.gif");
  TiledFloor f = new TiledFloor( floorIm, 8);
  scene.addChild( f.getFloorMesh() );    // add the floor
}
```

It's easy to switch to the Floor class, by replacing the TiledFloor line in addFloor() with:

```
Floor f = new Floor( floorIm, 8);
```

Now a single 8-by-8 square mesh is produced.

Figure 11 shows how TiledFloor divides up a size-by-size square into unit tiles.

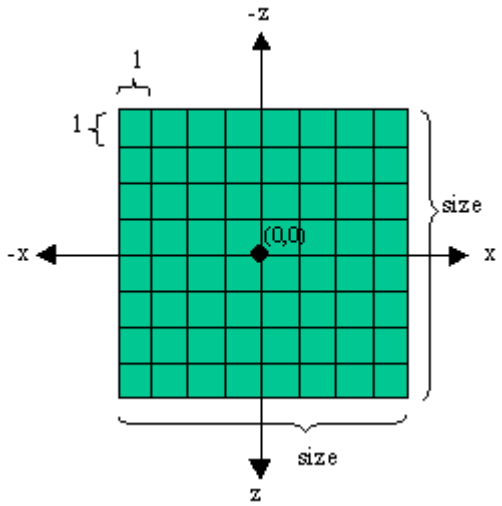

Figure 11. The TiledFloor View of a Surface.

The subdivision assumes that size is an even integer.

Figure 12 focuses on the top row of Figure 11 to indicate how the coordinates of the tiles are calculated, and ordered.

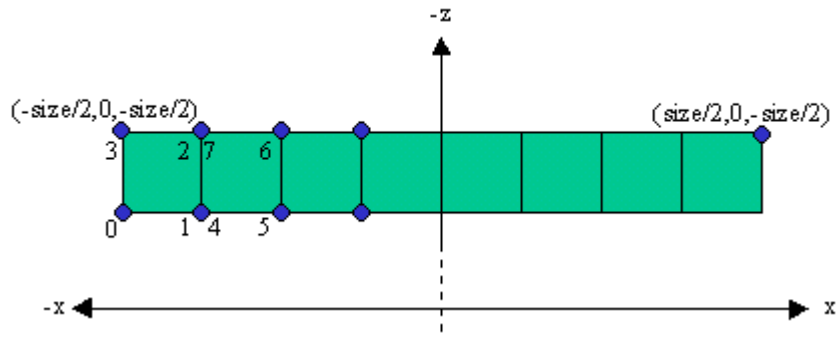

Figure 12. The Coordinates in the Top Tile Row in Figure 11.

Each tile is defined by four points, starting at the tile's bottom-left corner (when viewed from above), and ordered counter-clockwise. Point 3 of the first tile of the row is at (-size/2, 0, -size/2), which means that the z coordinate of point 0 is -size/2 + 1, and the x-coordinate of point 2 is -size/2 + 1.

Points 1 and 2 of the first tile are the same as points 4 and 7 in the second tile, which means that the complete mesh contains a large number of duplicated points (in fact, most points occur four times). The advantage of this repetition is that each tile can be easily assigned texture coordinates, and so assigned a texture.

Figure 13 shows the texture coordinates for the first two tiles of Figure 12.

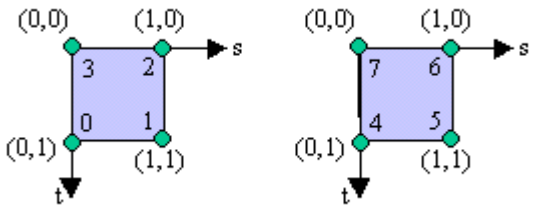

Figure 13. Texture Coordinates for Two Tiles.

The coordinates are specified in the same order for each tile, and so can be easily generated, as we'll see in makeTexCoords() below.

The TiledFloor constructor builds a single Mesh, called floorMesh.

```
private Mesh floorMesh;  // global


public TiledFloor(Image2D floorIm, int sz)
{
  /* Make sure the size value is divisible by 2, since
     we'll use size/2 later. */
  int size = (sz/2)*2;
  if (size != sz)
    System.out.println("Size set to multiple of 2: " + size);

  int numTiles = size*size;
    /* Each tile is a 1-by-1 quadrilateral (quad), so numTiles
       of them will cover a size-by-size floor. */

  // build the floor mesh
  VertexBuffer floorVertBuf = makeGeometry(size, numTiles);

  IndexBuffer floorIdxBuf =
       new TriangleStripArray( makeIndicies(numTiles),
                               makeStripLens(numTiles) );

  Appearance floorApp = makeAppearance(floorIm);

  floorMesh = new Mesh(floorVertBuf, floorIdxBuf, floorApp);
}
```


The hard work is done by the support methods makeGeometry(), makeIndicies(), and makeStripLens(). makeAppearance() is unchanged from the Floor class: it clamps the texture image (floorIm) to the texture coordinates for each triangle strip (i.e. for each tile).

makeGeometry() generates values for the mesh vertices and their texture coordinates.

```
private VertexBuffer makeGeometry(int size, int numTiles)
{
  // create vertices
  short[] verts = makeVerts(size, numTiles);
  VertexArray va = new VertexArray(verts.length/3, 3, 2);
  va.set(0, verts.length/3, verts);

  // create texture coordinates
  short[] tcs = makeTexCoords(numTiles);
  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 coordinates arrays returned by getVerts() and makeTexCoords() are used to build the VertexBuffer for the floor.

makeVerts() assigns four points to each tile. The first coordinate is in the bottom-left corner of the tile (when viewed from above), and the others are ordered counter-clockwise from there. The tiles are created starting from the top-left corner of the floor, working across, and going down row by row. This ordering is shown graphically in Figure 12.

```
private short[] makeVerts(int size, int numTiles)
{
  short[] verts = new short[12*numTiles];  // 3*4 pts for each tile
  int i=0;
  for(int z = (-size/2)+1; z <= size/2; z++)
    for(int x = -size/2; x <= (size/2)-1; x++) {
      verts[i] = (short) x; verts[i+1]=0; verts[i+2] = (short) z;
      verts[i+3]=(short)(x+1); verts[i+4]=0; verts[i+5]=(short) z;
      verts[i+6]=(short)(x+1);verts[i+7]=0;verts[i+8]=(short)(z-1);
      verts[i+9]=(short) x; verts[i+10]=0;verts[i+11]=(short)(z-1);
      i += 12;
    }
  return verts;
}
```

makeTexCoords follows the ordering shown in Figure 13. Each tile uses four (s,t) coordinates, in the order{0,1, 1,1, 1,0, 0,0}. This is a counter-clockwise direction starting from the bottom-left corner of a tile.

```
private short[] makeTexCoords(int numTiles)
{ short[] tcs = new short[8*numTiles];   // 2 * 4 pts for each tile
```



```
    for(int i = 0; i < 8*numTiles; i += 8) {  // {0,1, 1,1, 1,0, 0,0}
      tcs[i] = 0; tcs[i+1] = 1;               // for each tile
      tcs[i+2] = 1; tcs[i+3] = 1;
      tcs[i+4] = 1; tcs[i+5] = 0;
      tcs[i+6] = 0; tcs[i+7] = 0;
    }
    return tcs;
  }
```

The indicies for each tile are the ordering of its four points to collect two triangles into a triangle strip. The first tile in Figure 12 is ordered {1,2,0,3}, the second tile {5,6,4,7}, and so on. The pattern is regular enough to be readily generated by makeIndicies().

```
  private int[] makeIndicies(int numTiles)
  { // positions for first tile: {1,2,0,3}
    int pos1 = 1; int pos2 = 2;
    int pos3 = 0; int pos4 = 3;

    int[] idxs = new int[4*numTiles];   // 4 points for each tile
    for(int i = 0; i < 4*numTiles; i += 4) {
      idxs[i] = pos1; pos1 += 4;   // increment the positions by 4
      idxs[i+1] = pos2; pos2 += 4;
      idxs[i+2] = pos3; pos3 +=4;
      idxs[i+3] = pos4; pos4 += 4;
    }
    return idxs;
  }
```

The floor mesh is a sequence of tiles, each tile made from a triangle strip of four points. This means that the strip lengths array is a long sequence of 4's.

```
  private int[] makeStripLens(int numTiles)
  { int[] lens = new int[numTiles];
    for(int i = 0; i < numTiles; i++)
      lens[i] = 4;
    return lens;
  }
```


7. The Bendy Block

The bendy block is a SkinnedMesh object with two parts: a mesh 'skin', and a 'skeleton' made of Group nodes. As the Group nodes are transformed (e.g. moved, rotated), the skin 'attached' to the nodes is affected as well.

Figure 3 represents the block's scene graph as a single rectangle, Figure 14 supplies more information.

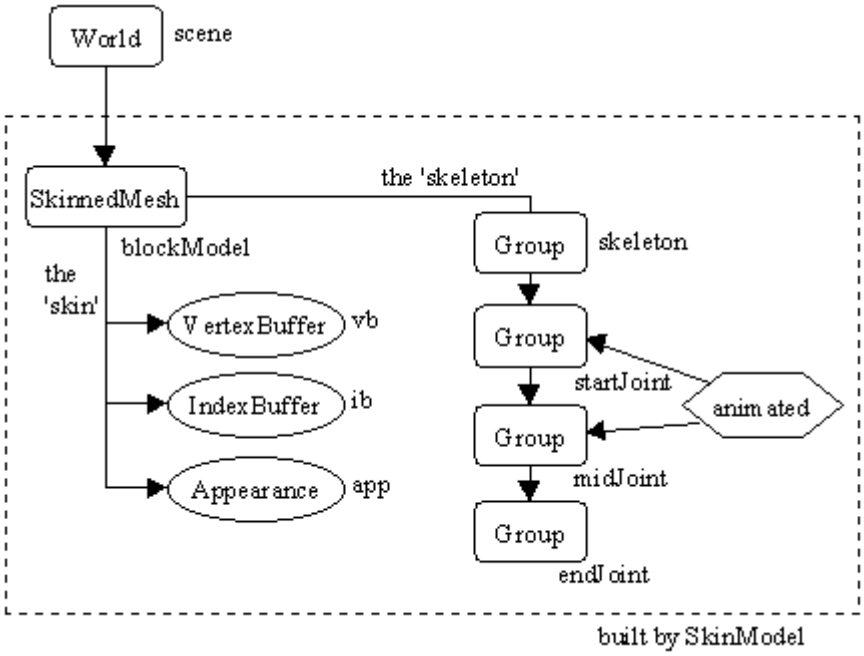

Figure 14. The SkinnedMesh Scene Graph for the Bendy Block.

The 'skin' part of the SkinnedMesh are the VertexBuffer, IndexBuffer, and Appearance components on the left of Figure 14. They define the skin mesh.

The skeleton is on the right hand side, and here the bendy block shows its simplicity, since only a chain of four Group nodes are employed. In general, a skeleton can be a complex arrangement of Group nodes forming a tree, and other types of nodes can be utilized, such as Mesh objects.

Figure 14 hides the details of the bendy block's animation inside a hexagon. The animation affects the startJoint and midJoint Group nodes. I'll describe it later.

Also missing from Figure 14 are the connections between the skin and skeleton. This is a matter of specifying which skin mesh vertices are associated with which Group nodes in the skeleton.

The SkinModel constructor highlights the main functional division in the class:

```
public SkinModel()
{ makeBlock();
  setUpAnimation();
}
```


The SkinnedMesh object is created using makeBlock(), and the animation is set up through setUpAnimation(). I'll explain each of these in turn.

7.1. Creating the Block

The SkinnedMesh object is a skin and skeleton. The skin mesh comprises 12 points, which define the block shown in Figure 15.

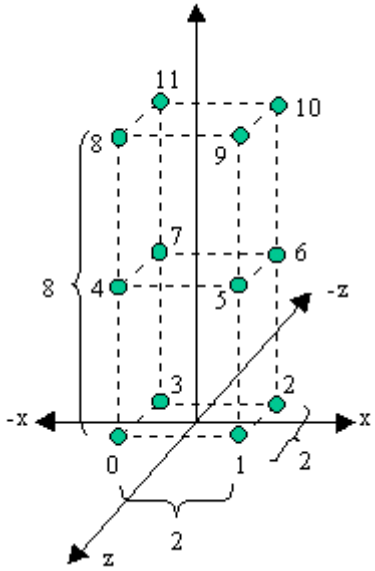

Figure 15. The Skin of the Block.

The block rests on the XZ plane, has a square 2-by-2 base, and is 8 units high. The block is divided into three levels, comprising three groups of points {0,1,2,3}, {4,5,6,7}, and {8,9,10,11}. There are ten faces, and each face is a quadrilateral.

The skeleton consists of a chain of four Group nodes, called skeleton, startJoint, midJoint, and endJoint (shown in Figure 14). skeleton isn't assigned any mesh points, but each 'joint' node is tied to a particular level in the mesh. The positions of the joint nodes relative to the mesh are indicated by red boxes in Figure 16, and labeled with abbreviations of their names.

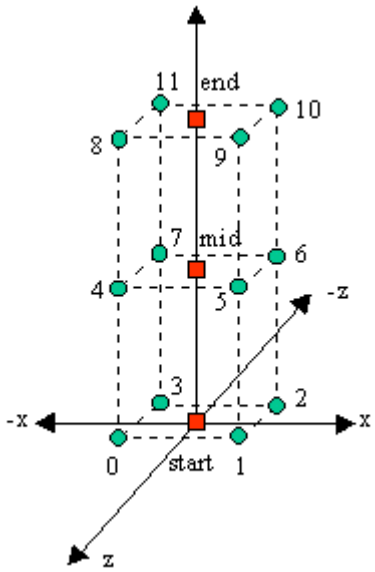

Figure 16. The Skin and Skeleton of the Block.

startJoint is linked to the coordinates {0,1,2,3}, midJoint to {4,5,6,7}, and endJoint to {8,9,10,11}. The nodes are positioned up the y-axis, equidistant from their skin vertices.

Since midJoint and endJoint are children of the startJoint, they will be translated when startJoint is rotated, which will cause their skin vertices to move as well. Similarly, when midJoint rotates, endJoint will be repositioned, affecting its skin vertices. This situation is illustrated by Figure 17, when midJoint is rotated 30 degrees clockwise around the z-axis. Only the vertices for the front faces are shown.

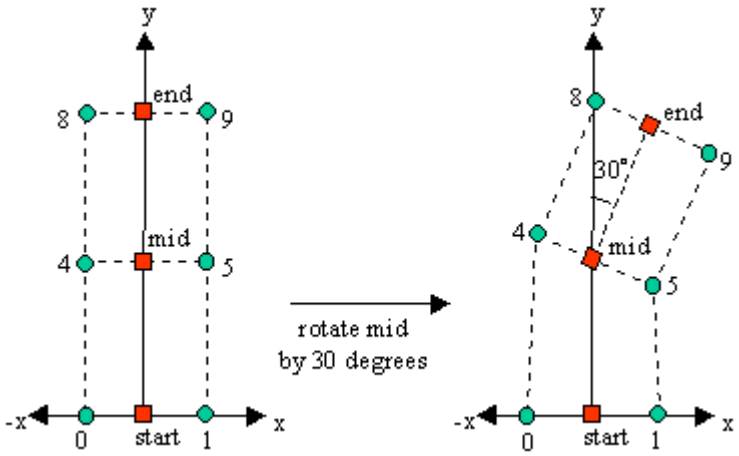

Figure 17. Rotating midJoint in the Block.

This behaviour shows why the Group node hierarchy is called a skeleton. The movement of a Group node affects all of its node descendents, *and* all the skin vertices attached to those nodes. The same principal applies to any jointed mechanism, such as your arm. When your shoulder rotates, it takes your elbow, wrist, and other joints with it, *and* the skin attached to those joints (just as well, really).

It's possible to link a mesh vertex to several Group nodes (in the same way that your smile is influenced by multiple muscles). The links between vertices and nodes can be weighted, to vary the strength of the attachments; this is useful if a vertex is affected by multiple Group nodes. In our example, a vertex is only attached to one node, and the weights are always set to 1.

7.2. Colouring the Block

The block doesn't include normals or texture coordinates, which makes the building task much easier. The block's vertices are coloured instead: the four coordinates in the base are red, the middle points are green, and those at the top are blue. When the shape is rendered, the colours on the block's faces are interpolated from the points. Figure 18 shows a close-up of the block.

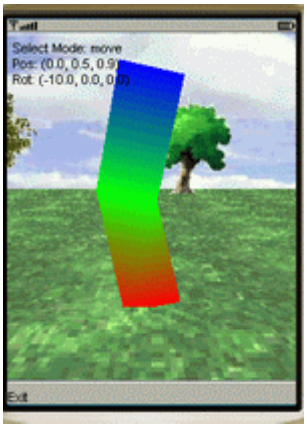

Figure 18. The Bendy Block's Colours.

The lack of normals means that the camera cannot reflect light, and so the block's Appearance component has no need for Material information.

7.3. Making the Skin

The skin of the SkinnedMesh object is created in makeBlock() in the SkinModel class.

```
private SkinnedMesh blockModel;  // global


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



```
    // create vertex colours
    byte[] cols = getColours();
    VertexArray colsArray = new VertexArray(cols.length/3, 3, 1);
    colsArray.set(0, cols.length/3, cols);

    VertexBuffer vb = new VertexBuffer();
    vb.setPositions(va, 1.0f, null); // no scale, bias
    vb.setColors(colsArray);

    IndexBuffer ib =
          new TriangleStripArray( getStripIndicies(),
                                  getStripLengths());

    Appearance app =  new Appearance();   // no material

    // create skinned mesh
    blockModel = new SkinnedMesh(vb, ib, app, makeSkeleton());
    blockModel.scale(0.1f, 0.1f, 0.1f);

    // code for linking the skin and skeleton, explained below

  }  // end of makeBlock()
```

As Figure 14 indicates, the skin consists of three components: a VertexBuffer of mesh coordinates, an IndexBuffer for the mesh, and an Appearance node. The Appearance node is essentially empty since the block is non-reflecting.

The VertexBuffer in makeBlock() utilizes two VertexArrays, an array of 12 vertices, and an array of colours. The vertices correspond to the 12 points shown in Figure 15.

```
private short[] getVerts()
{ short[] vals = { -1,0,1, 1,0,1, 1,0,-1, -1,0,-1,  //bottom coords
                   -1,4,1, 1,4,1, 1,4,-1, -1,4,-1,     // middle
                   -1,8,1, 1,8,1, 1,8,-1, -1,8,-1   }; // top
  return vals;
}
```

The colour coordinates are created by getColours():

```
private byte[] getColours()
{ byte[] cols = {  -1,0,0,  -1,0,0,  -1,0,0,   -1,0,0,    // red
                   0,-1,0,  0,-1,0,   0,-1,0,  0,-1,0,    // green
                   0,0,-1,  0,0,-1,   0,0,-1,  0,0,-1  }; // blue
  return cols;
}
```

Each colour is a triplet of byte values for red, green, and blue (RGB). The coding trick is that since a colour value is a single byte, then -1 will be converted to the hexadecimal 0xFF, which means 'full on' for that colour.

The IndexBuffer for the mesh must define the strip indicies for each triangle strip, including their lengths. The block is made from ten quadrilaterals, with each quad coded as a triangle strip of two triangles.

For example, the lower quad on the front of the block is shown in triangle strip form in Figure 19: the sequence of points for the strip is {1, 5, 0, 4}.

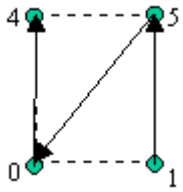

Figure 19. The Triangle Strip for the Lower Front Face of the Block.

All of the ten quads in the block are stripified in the same way:

```
private int[] getStripIndicies()
/* Return an array holding the indicies of each triangle strip.
   One quad = one triangle strip made up of 4 points.
*/
{ int[] ids = {   // the block is made from 10 quads
                1,5,0,4,  5,9,4,8,    // front 2 quads
                2,6,1,5,  6,10,5,9,   // right 2 quads
                0,4,3,7,  4,8,7,11,   // left 2 quads
                3,7,2,6,  7,11,6,10,  // back 2 quads
                2,1,3,0,     // bottom quad
                9,10,8,11    // top quad
              };
  return ids;
}

private int[] getStripLengths()
// return an array holding the lengths of each triangle strip
{ int[] lens = {4,4,4,4,4,4,4,4,4,4};  // 10 quads, each of 4 pts
  return lens;
}
```

7.4. Making the Skeleton

The skeleton is a chain of four Group nodes (see Figure 14), with the 'joint' nodes positioned up the y-axis (as in Figure 16). The chain is created by makeSkeleton():

```
// globals
private Group startJoint, midJoint, endJoint;


private Group makeSkeleton()
{
  Group skeleton = new Group();

  startJoint = new Group();
  skeleton.addChild(startJoint);
```



```
    midJoint = new Group();
    midJoint.setTranslation(0.0f, 4.0f, 0.0f);
    startJoint.addChild(midJoint);

    endJoint= new Group();
    endJoint.setTranslation(0.0f, 8.0f, 0.0f);
    midJoint.addChild(endJoint);

    return skeleton;
  } // end of makeSkeleton()
```

The skeleton and startJoint nodes are both positioned at (0,0,0) by default. There seems little point to having a skeleton Group node in this example. The reason is for extensibility: if extra nodes are added to the skeleton later, they should be attached to a Group node which is not affected by transformation. skeleton isn't changed, but startJoint is rotated.

7.5. Attaching the Skin to the Skeleton

The last part of makeBlock() creates the SkinnedMesh and specifies which Group nodes are linked to which vertices in the skin:

```
private SkinnedMesh blockModel;  // global


private Group makeSkeleton()
{
  // build the skin elements (see above for details)

  // create skinned mesh
  blockModel = new SkinnedMesh(vb, ib, app, makeSkeleton());
  blockModel.scale(0.1f, 0.1f, 0.1f);

  /* Link joints to the block's vertices. Each joint is linked to
     the four vertices at that level in the block. */
  blockModel.addTransform(startJoint, 1, 0, 4);
  blockModel.addTransform(midJoint, 1, 4, 4);
  blockModel.addTransform(endJoint, 1, 8, 4);
}
```

The SkinnedMesh addTransform() method links a Group node (a joint) to a sequence of vertices:

```
void addTransform(Node joint, int weight,
                  int firstVertex, int numVertices);
```

The weight is used to combine the effects of multiple links to a vertex (or vertices).

The addTransform() calls in makeSkeleton() links startJoint to the points {0,1,2,3}, midJoint to {4,5,6,7}, and endJoint to {8,9,10,11}.

7.6. Is this a Useful Approach?

Can a useful skinned mesh (e.g. a humanoid or animal) be built using the techniques just discussed? I have some doubts.

The block was designed on paper, and I took careful note of the vertex locations and their ordering. This made the creation of the VertexBuffer and IndexBuffer objects relatively painless. Complex shapes can be designed in this way, but with increased difficulty.

A more serious restriction is the shape's overly simple appearance: the block doesn't have normals or texture coordinates. These are essential for a realistic-looking articulated figure: coloured vertices are probably not enough. Unfortunately, the manual calculation of normals and texture coordinates is difficult.

Our solution since chapter 1 has been to create a model outside of J2ME, then import it using the ObjView converter. The problem is that a nicely designed model is stripified as part of the import process, usually creating new points, reordering others, and perhaps even combining vertices. All of this will cause the modeler's fine-tuned vertex ordering to be lost, and there's no way to link Group nodes to vertices without knowing their position in the VertexBuffer.

However, since ObjView outputs the vertex information as a Java array, it might be possible to hack together support code that can help us find vertices. For instance, we could look for all the points within a certain distance of a Group node.

The best solution is to utilize 3D modeling software which can generate M3G files containing SkinnedMesh objects. The 3DS Studio Max v.7 M3G exporter can do this, but two people have told me that the process seems a little flakey for large meshes.

8. Animating the Block

Figure 14 emphasizes the skin and skeleton parts of the block, hiding the animation in a hexagon. Figure 20 reveals it to contain two AnimationTracks, two KeyframeSequences, and an AnimationController.

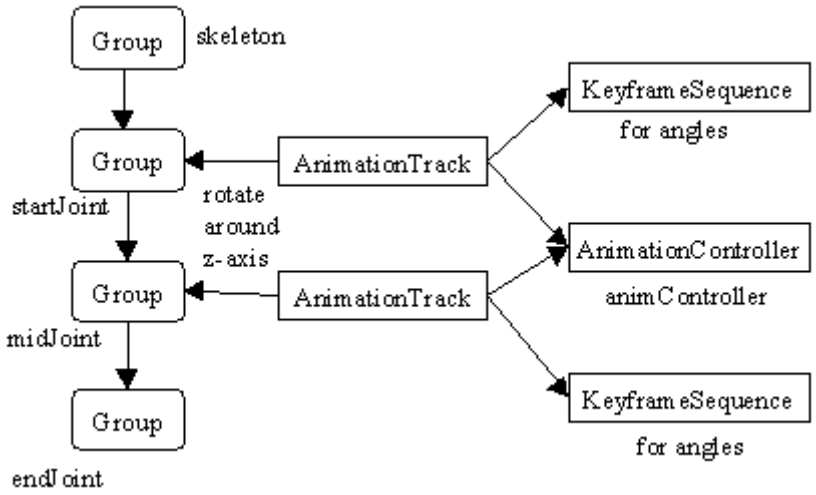

Figure 20. Animation Details for the Block

The startJoint and midJoint nodes are rotated around the z-axis by separate AnimationTracks. The choice of the z-axis is not significant, the nodes could be translated or rotated in other ways. Both nodes are transformed in the same manner simply to reduce the amount of coding for this part of SkinModel.

The keyframe sequence applied to the startJoint is shown in Figure 21.

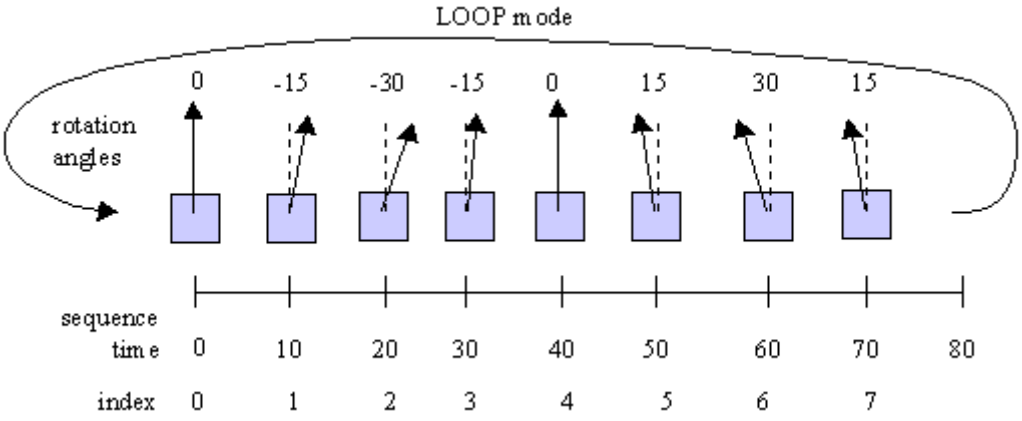

Figure 21. Keyframe Sequence for startJoint.

The blue boxes denote the Group node, with the positive z-axis pointing out of the paper. The node is rotated to the right, then back through the vertical to the left, and then back to the vertical again, all in the space of 80 sequence time units.

Figure 22 shows a similar behaviour for midJoint, except it starts moving left, and swings through larger angles.

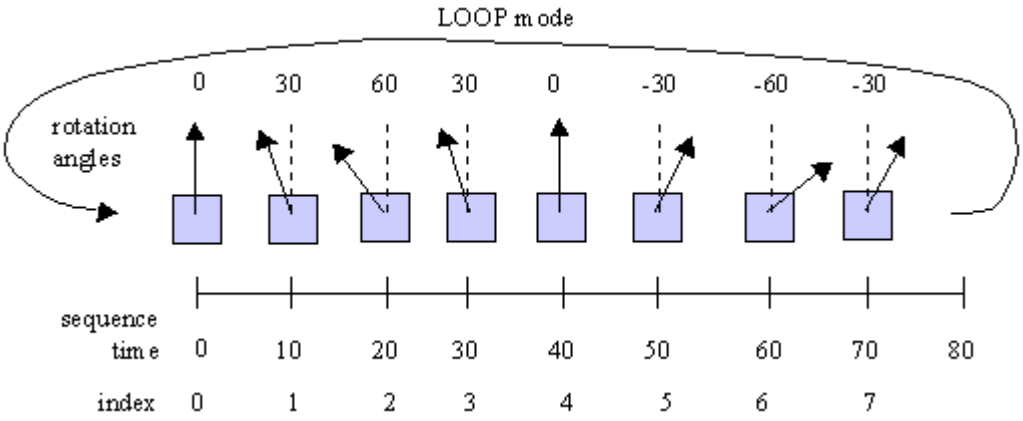

Figure 22. Keyframe Sequence for midJoint.

The combined effect of these rotations on the block can be hard to visualize. Figure 23 shows the first three stages of the rotations, corresponding to index positions 0, 1, and 2 in Figures 21 and 22.

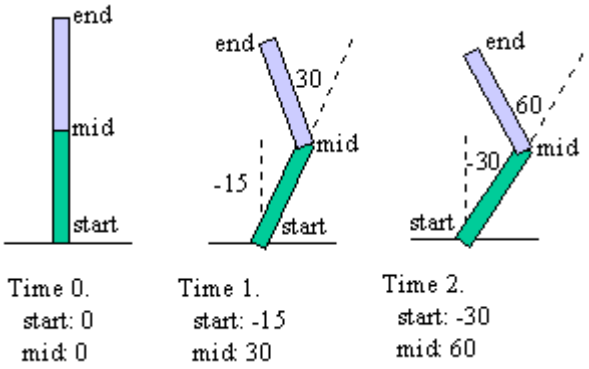

Figure 23. Rotations Applied to the Block.

'start', 'mid', and 'end' refer to startJoint, midJoint, and endJoint respectively.

The choice of rotations ensures that the top of the block stays over the origin, even as the base and middle oscillate underneath it.

The animation is created in setUpAnimation() in SkinModel.

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

  // create rotation animation track for start joint
  int[] startAngles = {0, -15, -30, -15, 0, 15, 30, 15};
  setJointRotation(startJoint, startAngles);
```



```
    // create rotation animation track for mid joint
    int[] midAngles = {0, 30, 60, 30, 0, -30, -60, -30};
    setJointRotation(midJoint, midAngles);
  }  // end of setUpAnimation()
```

The AnimationController is instantiated in setUpAnimation(), but it's left to setJointRotation() to create the AnimationTrack and KeyframeSequence for each rotation.

```
private void setJointRotation(Group joint, int[] angles)
// rotate the joint around the z-axis using the angles supplied
{
  KeyframeSequence rotKS = rotationFrames(angles);
  AnimationTrack rotTrack =
         new AnimationTrack(rotKS, AnimationTrack.ORIENTATION);
  rotTrack.setController(animController);
  joint.addAnimationTrack(rotTrack);   // affect joint
}
```

The rotation affects the `AnimationTrack.ORIENTATION` property of the Group node.

The array of z-axis angles passed into setJointRotation() is converted into a looping KeyframeSequence in rotationFrames().

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

  // each frame is separated by 10 sequence time units
  ks.setKeyframe(0,  0, rotZQuat(angles[0]) );
  ks.setKeyframe(1, 10, rotZQuat(angles[1]) );
  ks.setKeyframe(2, 20, rotZQuat(angles[2]) );
  ks.setKeyframe(3, 30, rotZQuat(angles[3]) );
  ks.setKeyframe(4, 40, rotZQuat(angles[4]) );
  ks.setKeyframe(5, 50, rotZQuat(angles[5]) );
  ks.setKeyframe(6, 60, rotZQuat(angles[6]) );
  ks.setKeyframe(7, 70, rotZQuat(angles[7]) );

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

  return ks;
}
```

rotationFrames() builds a rotation sequence spread over 8 frames and 80 time units, which repeats. This behaviour is illustrated by Figures 21 and 22. The method assumes that the angles[] array contains at least 8 values.

The z-axis rotation angle is converted to a quaternion by rotZQuat(), which bears a striking similarity to rotYQuat() developed in M3G chapter 2.

```
private float[] rotZQuat(double angle)
/* Calculate the quaternion for a clockwise rotation of
```

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

The z-axis rotation means that the k coefficient of the quaternion is affected in rotZQuat() rather that the j coefficient used in rotYQuat(). Information about quaternions can be found in M3G chapter 2.

9. Updating and Painting the Scene

SkinCanvas updates and repaints the scene in response to periodic calls to its update() method by a TimerTask object created in the SkinM3G class. Figure 24 shows the stages in each update and repaint.

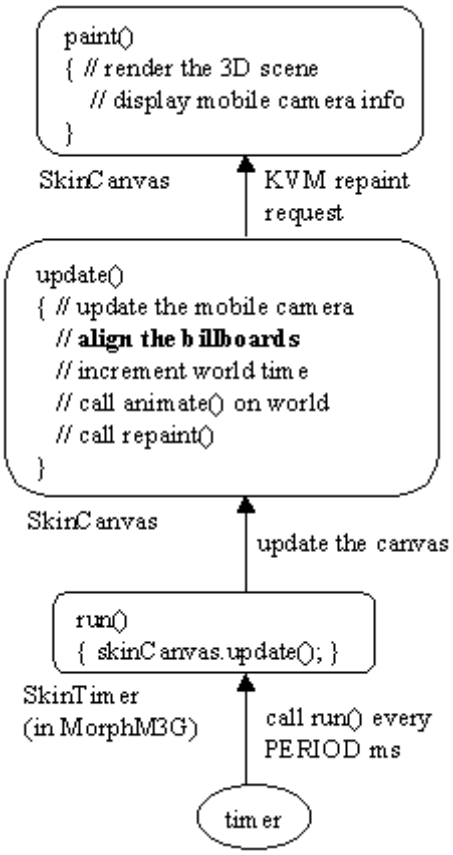

Figure 24. Updating and Repainting the SkinM3G Application.

The update() method:

```
// global scene elements
```



```
  private MobileCamera mobCam;
  private Billboard bb1, bb2, bb3;   // tree billboards

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


  public void update()
  {
    mobCam.update();
    bb1.align();  // align the billboards with the camera position
    bb2.align();
    bb3.align();

    appTime++;
    if (appTime >= nextTimeToAnimate)
      nextTimeToAnimate = scene.animate(appTime) + appTime;
    repaint();
  }
```

The mobile camera's update may result in it being moved or rotated, which necessitates a realignment of the three Billboard objects. The align() method in Billboard is:

```
  public void align()
  { bbMesh.align(cameraGroup);  }
```

The z-axis of the Billboard mesh (bbMesh) is aimed at the camera's new position, which is stored in cameraGroup.

The paint() method in SkinCanvas is unchanged from previous examples: it renders the scene, then draws the mobile camera's details into the top-left corner of the screen.

