M3G Chapter 1. Loading OBJ Models into M3G

This chapter describes how to display a Wavefront OBJ model in a MIDP 2.0 application called ModelDisplay, using the Mobile 3D Graphics API (M3G for short).

ModelDisplay displays a single OBJ model in the center of the screen, and allows it to be translated left, right, up, down, forward, and back, and rotated around the y-axis via the keypad (see Figures 1 and 2). Holding a key down causes its associated action to repeat.

Figure 1. Displaying a Hand.

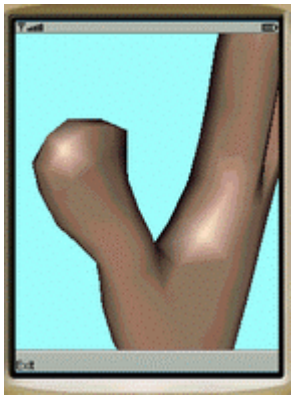

Figure 2. After Translations and Rotations.

Although ModelDisplay is designed to display a single model at a time, with some effort it can be modified to show multiple models at once. I'll go into the details later.

I won't be explaining MIDP 2.0, or even M3G; plentiful documentation on both of them comes with the latest version of the J2ME Wireless Toolkit, v.2.2 (available from http://java.sun.com/products/j2mewtoolkit/).

The motivation for this work is that 3D models can only be loaded into M3G if they're stored in the M3G file format. There are a few free or shareware utilities for exporting M3G files from 3D modeling software. For example, HI Corporation, developers of the Mascot Capsule engine, offers several tools at http://www.mascotcapsule.com/M3G/index_e.html.

Ben Hui maintains a great website listing M3G resources, including tutorials and development software (http://www.benhui.net/mobile3d).

Our approach doesn't use the M3G file format, instead the model is converted into Java methods.

The 3D modeling package must export the model as an OBJ file. This is processed by a separate Java 3D application, called ObjView, which outputs a text file of methods containing the model's vertices, normals, and so on. These methods are manually pasted into the ModelDisplay prior to its compilation. The general strategy is shown in Figure 3.

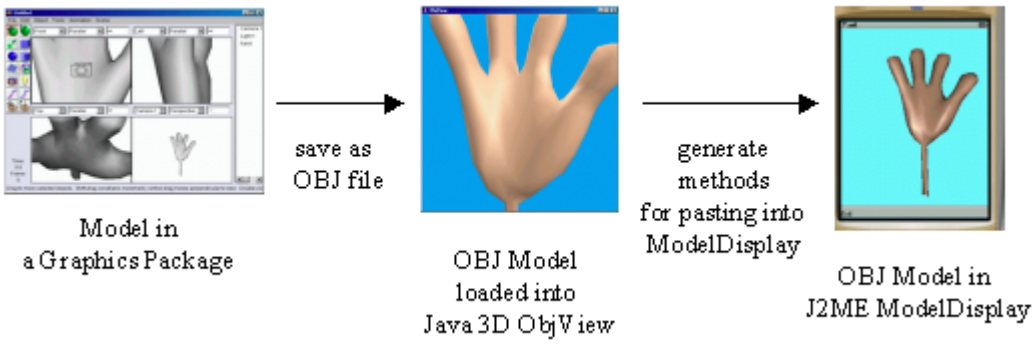

Figure 3. Porting an OBJ file to M3G.

The main advantage of this approach is the accessibility of the code representing the model. The methods can be pasted into *any* M3G-based application, not just ModelDisplay. The methods can be easily modified, and integrated with other M3G code in various ways. Later chapters will use ObjView-generated methods to create animated and morphing models.

1. Creating the Model

Most 3D graphics packages can manipulate OBJ files and export them. I use ArtOfIllusion (http://www.artofillusion.org/), which is easy to learn, and comes with a surprising number of advanced features, considering that it's open source.

MatCube.obj is shown in ArtOfIllusion in Figure 4.

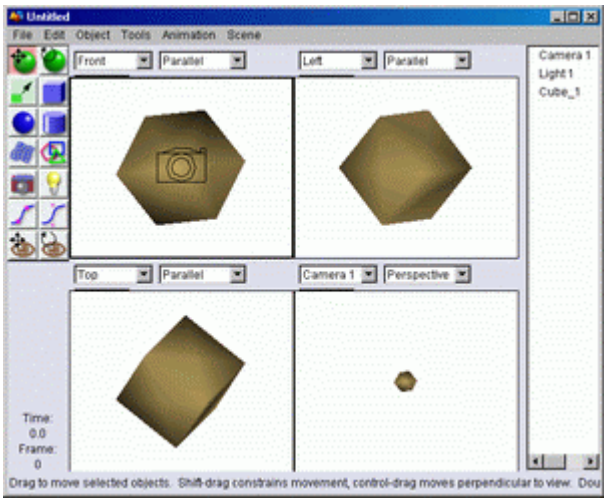

Figure 4. MatCube.Obj in ArtOfIllusion.

Only a single shape should be created, and it shouldn't be too complex. As explained later, if the vertex count reaches the high thousands, then there's a good chance that ModelDisplay will fail, by exceeding J2ME's memory limit for array sizes.

The shape should be centered at (0,0,0), and its y-axis vertically aligned. This will cause it to rotate tightly around the origin in ModelDisplay.

Unfortunately, there are some bugs in ArtOfIllusion, including some strange distortions to textures when they're wrapped around shapes. Since first writing this chapter, I've moved to using MilkShape 3D (http://www.swissquake.ch/chumbalum-soft/ms3d/) which has excellent features, including support for a very large range of file formats, with an emphasis on those used in gaming. There are many excellent tutorials available, numerous 3rd party plugins, and the software is actively supported. Milkshape 3D can be downloaded for a 30-day free test.

The OBJ export feature of MilkShape3D produces a separate MTL material file. One quirk of that file is the lack of an "illum" line for specifying lighting. Add the line:

```
illum 2
```

to switch on full lighting (ambient, diffuse, and specular).

Another minor issue is that the "map_Kd" line refers to the file using a "./" directory path. For instance:

```
map_Kd ./model.gif
```

Delete the path, leaving only the filename, model.gif.

2. Using ObjView

Before ObjView can be called, Java 3D must be installed – it's an extension to J2SE, available from http://java.sun.com/products/java-media/3D/. I recommend v1.3.1 using OpenGL.

Don't be (too) concerned if you don't know Java 3D. ObjView is a utility to generate the necessary ModelDisplay methods, so there's no real need to understand how it works.

However, if you *are* interested in Java 3D, then consider my games book, "*Java Graphics and Gaming*" at http://fivedots.coe.psu.ac.th/~ad/jg/. It contains over 15 chapters on Java 3D, and ObjView is derived from an example in chapter 9.

After installing Java 3D, you can test if everything is okay by running the standard HelloUniverse demo found in `<JAVA_HOME>\demo\java3d\HelloUniverse`:

```
> java HelloUniverse
```

When Java 3D is working, ObjView can be started by supplying it with a OBJ filename:

```
> java ObjView matCube.obj
```

matCube.obj and its material file, matCube.mtl, should be in the same directory as ObjView.java.

The OBJ file will be displayed on screen as shown in Figure 5.

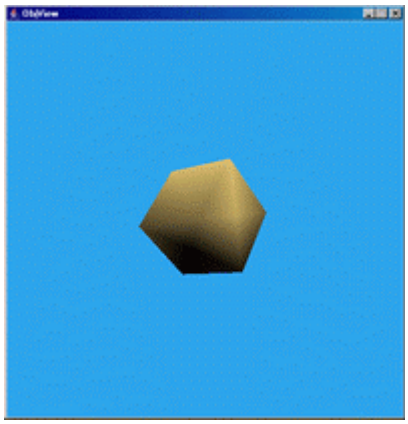

Figure 5. MatCube.obj in ObjView

The user's viewpoint can be moved via combinations of mouse drags and control keys. Dragging while pressing the main mouse button rotates the viewpoint. Dragging while pressing the second mouse button (or the ALT key and the main mouse button) zooms the viewpoint in or out. The third mouse button (or the shift key and the main mouse button) translates the view.

ObjView converts each model found in the OBJ file into a Java 3D Shape3D node. Below that node are Geometry and Appearance nodes. The loading process triangulates and stripifies the model, so the Geometry node will be a TriangleStripArray instance.

A triangle strip is a series of triangles that share vertices: a triangle is built using two vertices from the previous triangle, plus one new vertex. Figure 6 shows the general idea.

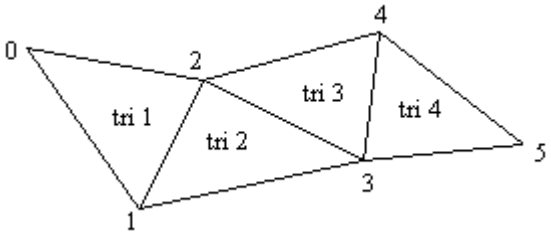

Figure 6. A Triangle Strip.

Triangle 1 (tri 1) is defined using the points (0,1,2), and triangle 2 (tri 2) reuses points 1 and 2, only requiring a new vertex for point 3. The *triangle strip* is collectively the points {0,1,2,3,4,5}. In general, a strip of n vertices defines n-2 triangles. This is a big improvement over the usual encoding of three points per triangle, which only allows n/3 triangles to be specified.

The points in a model represent multiple triangle strips, requiring a strips index, stating where a given strip begins and ends in the points data. Usually the strips index is implemented by storing the lengths of consecutive strips in an array.

Point data in Java 3D's TriangleStripArray is interleaved, with each 'point' represented by a group of values specifying its textures coordinates, colour coordinates, normals, and vertices. However, the OBJ file format doesn't support colour coordinates, so that information is absent from ObjView's data.

Some brief details on the OBJ format can be found at http://www.wes.hpc.mil/faq/tips/vis_tips/obj.htm. Another source is the Java 3D documentation for the OBJ file loader, the ObjectFile class, which is used by ObjView to load the model.

The contents of the points data is affected by the model's design. For instance, if the model doesn't use texturing, then there will be no texture coordinates in the data set. If the model only uses material colours (e.g. ambient and diffuse colours), that information will be present in the shape's Appearance node.

The points data and material information is extracted from the shape, and printed to the examObj.txt file, in the form of methods. Their names are: getVerts(), getNormals (), getTexCoords(), getColourCoords(), getStripLengths(), and setMatColours(). They hold the model's vertices, normals, texture coordinates, colour coordinates, strip lengths, and material data.

For example, the contents of examObj.txt generated for the MatCube.obj file is:

```
// Methods for model in MatCube.obj

  private short[] getVerts()
  // return an array holding Verts [96 values / 3 = 32 points]
  {
    short[] vals = {
     68,-97,49,    -9,1,127,     -27,-55,42,    -123,-14,35,
     -46,-112,-44,  -57,-7,-47,    9,-1,-128,     -57,-7,-47,
     -68,97,-49,    -123,-14,35,  -39,49,39,     -9,1,127,
     -39,49,39,     46,112,44,    -68,97,-49,    27,55,-42,
     9,-1,-128,     27,55,-42,    123,14,-35,    46,112,44,
     57,7,47,       -9,1,127,     57,7,47,       68,-97,49,
     123,14,-35,     39,-49,-39,  9,-1,-128,     39,-49,-39,
     -46,-112,-44,   68,-97,49,   -46,-112,-44, -27,-55,42
    };
    return vals;
  }  // end of getVerts()


  private byte[] getNormals()
  // return an array holding Normals [96 values / 3 = 32 points]
  {
    byte[] vals = {
     69,-97,49,    -9,2,127,     -47,-95,72,    -123,-13,35,
     -45,-111,-45, -99,-13,-81,  9,-2,-128,     -99,-13,-81,
     -69,97,-49,   -123,-13,35,  -67,85,69,     -9,2,127,
     -67,85,69,    45,111,45,    -69,97,-49,    47,95,-72,
     9,-2,-128,    47,95,-72,    123,13,-35,    45,111,45,
     99,13,81,     -9,2,127,     99,13,81,      69,-97,49,
     123,13,-35,   67,-85,-69,   9,-2,-128,     67,-85,-69,
     -45,-111,-45, 69,-97,49,    -45,-111,-45,  -47,-95,72
    };
    return vals;
  }  // end of getNormals()
```



```
private int[] getStripLengths()
// return an array holding the lengths of each triangle strip
{
  int[] lens = {
    32
  };
  return lens;
}  // end of getStripLengths()


private Material setMatColours()
// set the material's colour and shininess values
{
  Material mat = new Material();

  mat.setColor(Material.AMBIENT, 0x00000000);
  mat.setColor(Material.EMISSIVE, 0x00000000);
  mat.setColor(Material.DIFFUSE, 0xFFE3C472);
  mat.setColor(Material.SPECULAR, 0x00000000);
  mat.setShininess(1.0f);
  return mat;
} // end of setMatColours()
```

Since the MatCube.obj model doesn't use colour or texture coordinates, the getTexCoords() and getColourCoords() methods are not generated.

The points methods (getVerts(), getNormals(), getTexCoords(), and getColourCoords ()) store their data as integers, as required by the M3G VertexArray class inside ModelDisplay. However, the data is manipulated as floats by Java 3D, so ObjView has to do a conversion before the model methods are generated.

The vertices and normals are scaled and rounded to be integers in the range -128 to 127, while the texture and colour coordinates become integers in the range 0 to 255.

3. The M3G ModelDisplay Application

The UML class diagrams for ModelDisplay are given in Figure 7. Only the public methods are shown.

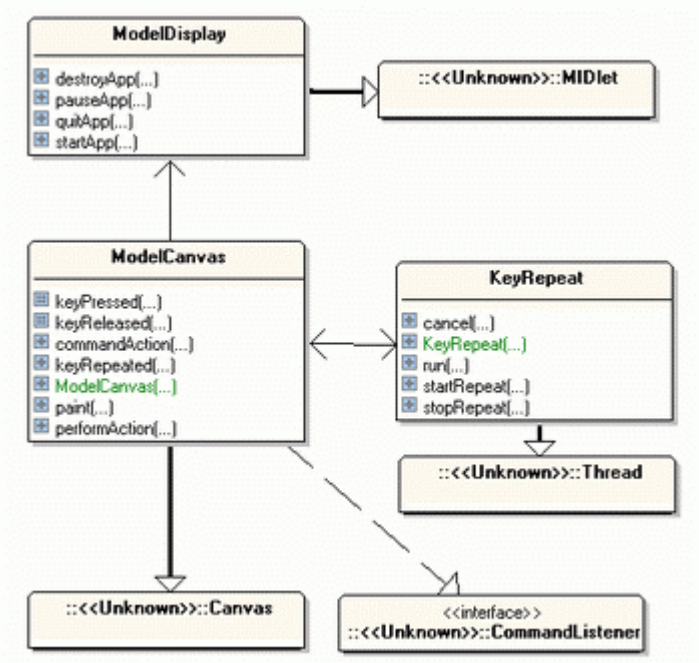

Figure 7. ModelDisplay Class Diagrams

ModelDisplay.java is a simple midlet that creates a ModelCanvas object for displaying the model.

Once the model methods have been pasted into the ModelCanvas class, the application is built and run inside the Wireless Toolkit v.2.2 (or any other development environment supporting M3G), resulting in Figure 8.

Figure 8. MatCube.obj in ModelDisplay.

The KeyRepeat class supports key repetition, which isn't offered by most J2ME implementations. The code is derived from the article:

"*Using the Low-level GUI - Repeating Keys*"
Roman Bialach, microdevnet, August 2001
`http://www.microjava.com/articles/techtalk/repeating`

The essential idea is to create a thread which periodically calls the key action method in ModelCanvas, sending it the keycode of the currently pressed key.

3.1. The ModelCanvas Class

ModelCanvas uses M3G's immediate mode, which requires the programmer to directly manage most aspects of the 3D scene, such as mesh rendering, the camera position, and lighting.

createScene() obtains a Graphics3D instance, and sets up the camera, lights, and the initial position and rotation values for the model. The model is loaded via calls to makeGeometry() and makeAppearance().

```
private void createScene() throws Exception
{
  addCommand( new Command("Exit", Command.EXIT, 1) );

  g3d = Graphics3D.getInstance();

  // create a camera
  camera = new Camera();
  float aspectRatio = ((float) getWidth()) / ((float) getHeight());
  camera.setPerspective(45.0f, aspectRatio, 0.1f, 50.0f);

  // set up the camera's position
  camTrans = new Transform();
  camTrans.postTranslate(X_CAMERA, Y_CAMERA, Z_CAMERA);

  // create a light
  light = new Light();
  light.setColor(0xffffff);  // white light
  light.setIntensity(1.25f); // over bright

  // initialise the model's global transform vars
  modelTrans = new Transform();
  xTrans = 0.0f; yTrans = 0.0f; zTrans = 0.0f;
  totalDegrees = 0;

  makeGeometry();
  makeAppearance();

  bg = new Background();
  bg.setColor(0x9EFEFE); // light blue
} // end of createScene()
```

3.2. The makeGeometry() Method

makeGeometry() calls the methods getVerts(), getNormals(), getTexCoords(), and getColourCoords() in order to initialise VertexArrays with positions, normals, texture coordinates, and colour coordinates. These arrays are then collected together in a VertexBuffer.

A M3G TriangleStripArray is created using getStripLengths(), to enable the renderer to index into the VertexBuffer to access a particular strip.

```
  private void makeGeometry()
  {
    // 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);
                          // this assumes (s,t) texture coordinates
    texArray.set(0, tcs.length/2, tcs);

/*
    // create colour coordinates
    short[] cols = getColourCoords();
    VertexArray colsArray = new VertexArray(cols.length/3, 3, 2);
                          // this assumes RGB colour coordinates
    colsArray.set(0, cols.length/3, cols);
*/

    // ------ create the VertexBuffer for the model -------

    vertBuf = new VertexBuffer();
    float[] pbias = {(1.0f/255.0f), (1.0f/255.0f), (1.0f/255.0f)};
    vertBuf.setPositions(va, (2.0f/255.0f), pbias); // scale, bias
       // fix the scale and bias to create points in range [-1 to 1]

    vertBuf.setNormals(normArray);

    vertBuf.setTexCoords(0, texArray, (1.0f/255.0f), null);
        // fix the scale to create texCoords in range [0 to 1]

    // vertBuf.setColors(colsArray);

    // create the index buffer for the model (this tells MIDP how to
    // create triangle strips from the vertex buffer).
    idxBuf = new TriangleStripArray(0, getStripLengths() );
  }  // end of makeGeometry()
```

There are many assumptions wired into makeGeometry().

The vertices and normals are assumed to be integers in the range -128 to 127, the texture and colour coordinates integers in the range 0 to 255.

The vertices are added to the VertexBuffer with scale and bias values that map them to floats in the range -1 to 1. The normals are similarly scaled, but this is done automatically by the KVM.

The texture coordinates are added with a scale that put them in the range 0 to 1. This is also done to the colour coordinates, but automatically.

We assume that (s,t) texture coordinates are supplied, which seems standard for OBJ models. However, 3 and 4 element components are possible.

As mentioned earlier, the Wavefront format doesn't support colour coordinates, and so it is fairly pointless to include the setColors() calls to VertexBuffer. The code is there just in case. I assume that a colour uses 3 components (RGB), but colours may have an additional alpha argument.

The texture coordinates are accessed indirectly via a call to getTexCoordsRev(). This applies the M3G non-standard (s,t) encoding: s increases from left to right as usual, but t increases from top to bottom. This means that the supplied t values must be reversed since Java 3D generates t values increasing from bottom to top.

```
private short[] getTexCoordsRev()
{
  short[] tcs = getTexCoords();

  // t' = 255 - t   (will later be scaled from 255 to 1)
  for(int i=1; i < tcs.length; i=i+2)
    tcs[i] = (short)(255 - tcs[i]);

  return tcs;
}
```

The texture related code in makeGeometry(), and all of getTexCoordsRev(), should be commented out if the supplied model doesn't use a texture.

3.3. The makeAppearance() Method

makeAppearance() loads the texture image and converts it into a Texture2D object. The material information from setMatColours() is also stored.

```
private void makeAppearance() throws Exception
{

  // load the image for the texture
  Image im = Image.createImage(TEX_FNM);

  // create an Image2D for the Texture2D
  Image2D image2D = new Image2D(Image2D.RGB, im);

  // create the Texture2D and enable mip mapping
  // the texture color is modulated with the lit material color
  Texture2D tex = new Texture2D(image2D);
  tex.setFiltering(Texture2D.FILTER_NEAREST,
                              Texture2D.FILTER_NEAREST);
  tex.setWrapping(Texture2D.WRAP_CLAMP, Texture2D.WRAP_CLAMP);
  tex.setBlending(Texture2D.FUNC_MODULATE);

  // create the appearance
  app = new Appearance();
  app.setTexture(0, tex);
  app.setMaterial( setMatColours() );
}  // end of makeAppearance()
```


If the model doesn't use a texture, them most of makeAppearance() can be commented away.

3.4. Key Processing

The key repetition technique used in ModelCanvas is based on Roman Bialach *microdevnet* article (`http://www.microjava.com/articles/techtalk/repeating`). One change is to introduce a two-speed repetition rate. For the first few repaints, key duplication is slow, then jumps to a higher rate. This allow the user to press and release a key slowly without causing the model to move multiple times.

The key repetition thread calls ModelCanvas' performAction() with the gameAction value taken from the pressed key.

```
public void performAction(int gameAction)
{
  if (gameAction == UP)
    moveModel(0,MODEL_MOVE,0);    // y up
  else if (gameAction == DOWN)
    moveModel(0,-MODEL_MOVE,0);   // y down
  else if (gameAction == LEFT)
    moveModel(-MODEL_MOVE,0,0);   // x left
  else if (gameAction == RIGHT)
    moveModel(MODEL_MOVE,0,0);    // x right
  else if (gameAction == GAME_A)
    moveModel(0,0,MODEL_MOVE);    // z fwd
  else if (gameAction == GAME_B)
    moveModel(0,0,-MODEL_MOVE);   // z back
  else if (gameAction == GAME_C)
    rotYModel(-MODEL_ROT);        // rotate left
  else if (gameAction == GAME_D)
    rotYModel(MODEL_ROT);         // rotate right
}
```

The method is a multi-way branch that calls either moveModel() or rotYModel() to update the model's position or y-axis angle. These values are applied to the model by paint().

```
private void moveModel(float x, float y, float z)
// update the model's translation values
{ xTrans += x; yTrans += y; zTrans += z;
  repaint();
}

private void rotYModel(float degrees)
// update the model's y-axis rotation value
{ totalDegrees += degrees;
  repaint();
}
```


3.5. Painting

paint() is called whenever there's a change to the model's position and orientation. Due to the use of immediate mode, paint() must set up the lights and camera, and carry out low-level tasks like clearing the colour and depth buffers.

```
public void paint(Graphics g)
{
  // bind the canvas graphic to our Graphics3D object
  g3d.bindTarget(g, true, Graphics3D.DITHER |
                          Graphics3D.TRUE_COLOR);

  g3d.clear(bg);    // clear the colour and depth buffers

  g3d.setCamera(camera, camTrans);   // position the camera

  /* Set up a "headlight": a directional light shining
     from the direction of the camera. */
  g3d.resetLights();
  g3d.addLight(light, camTrans);

  updateModelTrans();

  /* Render the model. We provide the vertex and index buffers
     to specify the geometry; the appearance so we know what
     material and texture to use; and a transform to position
     the model. */
  g3d.render(vertBuf, idxBuf, app, modelTrans);

  g3d.releaseTarget();
} // end of paint()
```

The model's transform is set by updateModelTrans(); modelTrans is a global Transform instance.

```
private void updateModelTrans()
// the model's transform = a translation * a rotation
{
  modelTrans.setIdentity();   // reset
  modelTrans.postTranslate(xTrans, yTrans, zTrans);
  modelTrans.postRotate(totalDegrees, 0, 1, 0);  // around y-axis
}
```

render() requires that the VertexBuffer use triangle strips to represent the model, and employs the TriangleStripArray to determine which points belong to which strip.

4. SwordDisplay

We'll work through another complete example in this section: displaying a textured sword model (see Figures 9, 10, and 11).

Figure 9. sword.obj in ObjView.

In Figure 9, the sword's texture is most clearly visible as marking on its grip.

The model is represented by three files: sword.obj, sword.mtl, and swordTextures.gif (which holds the texture image).

The model methods output from ObjView are stored in examObj.txt. Unlike in the matCube.obj example, the methods include getTexCoords().

To avoid confusion at later stages, the ModelDisplay and ModelCanvas files are copied and renamed to SwordDisplay.java and SwordCanvas.java. KeyRepeat.java is copied as well. All occurrences of the class names ModelDisplay and ModelCanvas in the files are changed to SwordDisplay and SwordCanvas.

The model methods from examObj.txt are pasted into SwordCanvas.java. Also, its TEX_FNM constant is set:

```
private static final String TEX_FNM = "/swordTextures.gif";
```

A new project called SwordDisplay is created in the Wireless Toolkit. It's important not to forget to select Mobile 3D Graphics (JSR 184) as an additional API at this stage.

SwordDisplay.java, SwordCanvas.java, and KeyRepeat.java are copied over to `<WIRELESS_TOOLKIT_HOME>\apps\SwordDisplay\src`. The swordTextures.gif texture image goes in `<WIRELESS_TOOLKIT_HOME>\apps\SwordDisplay\res`. A subtle catch is to ensure that the image has sides which are a power of 2 in length; swordTextures.gif is 512x512 pixels large.

The project is built and run. The initial viewpoint is shown in Figure 10. Figure 11 shows the sword after a few rotations and translations.

Figure 10. SwordDisplay Initially.

Figure 11. SwordDisplay Later.

One element that can be 'tweaked' is the initial camera position, which is defined by constants at the start of SwordDisplay:

```
private static final float X_CAMERA = 0.0f;
private static final float Y_CAMERA = 0.0f;
private static final float Z_CAMERA = 3.0f;
```

Each translation moves the model by MODEL_MOVE units:

```
private static final float MODEL_MOVE = 0.2f;
```

This allows a better camera position to be calculated by recording the number of x, y, and z translations needed to get the model to the required spot.

5. Displaying Multiple Models

The examples so far have involved the rendering of a single model at a time. It is possible to get ModelDisplay to display multiple models at once, but there are some significant problems.

When ObjView detects multiple model in the OBJ file, each one is converted to a separate Shape 3D node, with its own Geometry and Appearance nodes. Several copies of the model methods are then output, one set for each model. The sets are distinguished by having method names that end with a number.

The first model will generate getVerts1(), getNormals1(), getTexCoords1(), getColourCoords1(), getStripLengths1(), and setMatColours1(). The second model's methods will end in a '2', and so on.

The current implementation of ModelCanvas cannot make use of these multiple model methods. Several parts of the class will need to be changed.

The simplest approach is to create multiple copies of the makeGeometry() and makeAppearance() methods, each calling one set of model methods. For example, makeGeometry1() and makeAppearance1() can load the first model's data, makeGeometry2() and makeAppearance2() the second model, and so on.

These 'make' methods will create their own VertexBuffer, TriangleStripArray, and Appearance objects (e.g. vertBuf1, idxBuf1, app1, and vertBuf2, idxBuf2, app2). Each of these can be rendered separately in the paint() method:

```
g3d.render(vertBuf1, idxBuf1, app1, modelTrans);
g3d.render(vertBuf2, idxBuf2, app2, modelTrans);
                :
```

Note, that I've used the same modelTrans transform for all the models.

Figure 12 shows a dumbbell, made from two spheres and a long box, displayed in ObjView. Figure 13 shows it loaded into ModelDisplay using the approach just described.

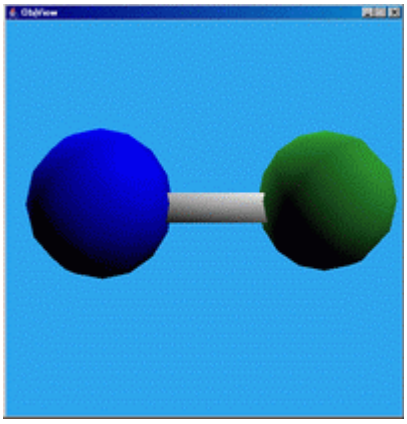

Figure 12. Dumbbell in ObjView.

Figure 13. Dumbbell in ModelDisplay.

There are two problems: scaling and positioning.

The vertices of every model are scaled to between -128 and 127. This means that models which are different sizes in the OBJ file will be rendered at about the same size in ModelDisplay. The solution is to calculate a vertex scaling factor which is the same across all the getVerts() methods. Currently, ObjView does not do that.

The other problem is that the relative positions of the models are wrong; this is due to ModelDisplay using the same transform to position and orientate all the models.

The best solution is probably to use a graphics package to combine multiple models into a single mesh before passing the OBJ file over to ObjView. For example, ArtOfIllusion allows a union operator to be applied between models to create a single entity.

6. Large Models

A serious concern is that the ObjView translation strategy generates methods containing very long arrays. For example, I had hoped to view a human figure in ModelDisplay (Figure 14).

Figure 14. humanoid.obj in ObjView.

ObjView generated the model methods, with getVerts() and getNormals() returning arrays holding 9,600 integers. These arrays were too big for the Wireless Toolkit's compiler which reported "code too large" error messages.

The standard way to tackle this kind of problem is to split the offending data structure into parts, distributed over several methods, then rebuild it at run time.

For example, instead of having a single large getVerts() method, the array can be split into two, spread across getVs1() and getVs2(), and be glued back together when the program is executed:

```
private short[] getVerts()
{
  short[] vs1 = getVs1();   // part 1 of the data
  short[] vs2 = getVs2();   // part 2 of the data
  short[] vs = new short[vs1.length + vs2.length];
  System.arraycopy(vs1, 0, vs, 0, vs1.length);
  System.arraycopy(vs2, 0, vs, vs1.length, vs2.length);
  return vs;   // the complete data
}
```

This avoids the compile time errors. Unfortunately, the emulator only allows an array to be at most 32 Kb large, so ModelDisplay crashes when it tries to render the VertexBuffer holding the large arrays. There doesn't seem any way around this restriction.

Currently, when ObjView outputs an array with 5000 elements or more, it prints a warning message to the screen, and inserts a comment into the offending model method. The generated array is not split into parts.

7. Other File Formats

ObjView uses Java 3D's ObjectFile loader to load, triangulate, and stripify the model. There are numerous other Java 3D loaders for different file formats: a list can be found at http://www.j3d.org/utilities/loaders.html. It includes loaders for MD2 (Quake), 3DS, MilkShape 3D, and AC3D.

ObjView's triangulation and stripification is achieved by setting flags in the ObjectFile constructor. The relevant code is in loadModel() in WrapObjView.java – it takes the model's filename, fnm, as input, and returns a Java 3D BranchGroup node (which leads down to the model's Shape3D node).

```
private BranchGroup loadModel(String fnm)
{
  int flags = ObjectFile.TRIANGULATE | ObjectFile.STRIPIFY;
  ObjectFile f = new ObjectFile(flags);
  Scene scene = null;
  try {
    scene = f.load(fnm);
  }
  // various catch blocks ...
  return scene.getSceneGroup();
}
```

These flags are a non-standard feature, which other loaders may not support. However, Java 3D includes a Stripifier class in its geometry utilities, which can turn any mesh into triangle strips.

The basic idea is to initialise a GeometryInfo object with the triangles of the loaded model, then use Stripifier to create the strips:

```
GeometryInfo gi = new GeometryInfo(GeometryInfo.TRIANGLE_ARRAY);

// fill in the vertices and indicies for the model's triangles
gi.setCoordinates( vertices );
gi.setCoordinateIndices( vertexIndicies );

// stripify the triangles
Stripifier stripper = new Stripifier();
stripper.stripify(gi);

// extract the triangle strip array
TriangleStripArray tsa =
         (TriangleStripArray) gi.getGeometryArray();
```

More details can be found in the Java 3D documentation, and in the third chapter of the Java 3D tutorial, "Easier Content Creation", downloadable from http://java.sun.com/products/java-media/3D/collateral/.

There's no need for the input data to be indexed triangles. Stripifier will convert any mesh (e.g. a quad array) to indexed triangles automatically, before stripification begins.

These capabilities mean that Java 3D is an excellent tool for building "triangle strip extractors" for different 3D file formats. The resulting triangle strips can be pasted into ModelDisplay to be viewed.

