JOGL-ES Chapter 2. Loading OBJ Models

A model can only be loaded into JOGL-ES once it's been translated into its component coordinates arrays (for the shape's vertices, normal, colors, and texture coordinates), and other variables for the materials and/or texture images. Generating these manually is impossible for any reasonably complicated shape.

This chapter describes how to convert a Wavefront OBJ model into a JOGL-ES class which can load, position, scale, and render the model. This allows fairly complex models, involving textures and materials, to be utilized in JOGL-ES applications. The conversion process is illustrated in Figure 1.

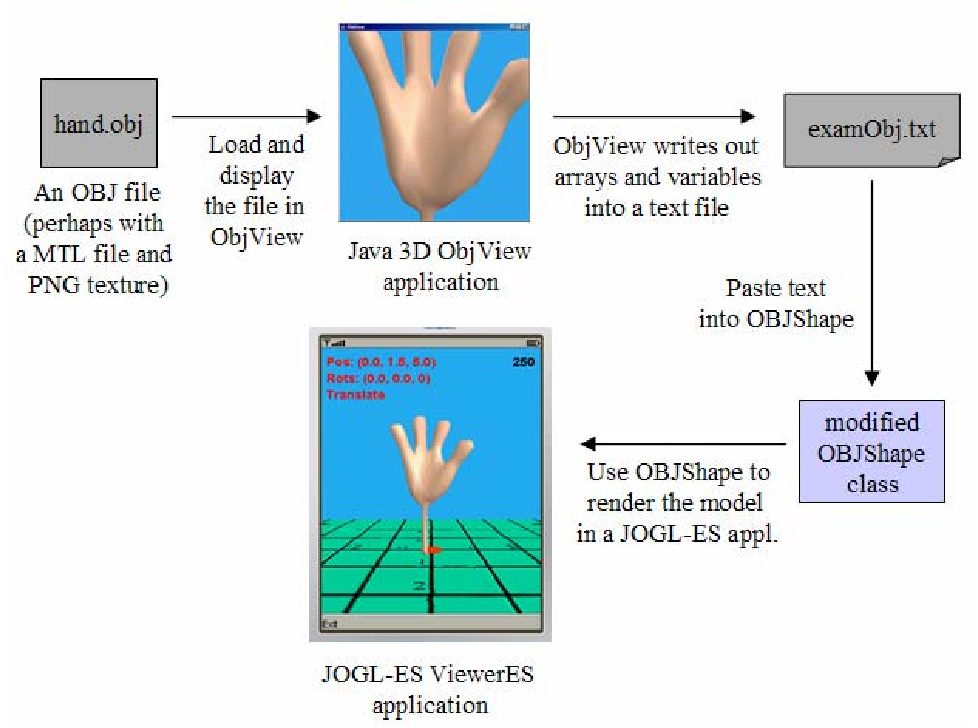
hand.obj
An OBJ file (perhaps with a MTL file and PNG texture)
Load and display the file in ObjView
Java 3D ObjView application
ObjView writes out arrays and variables into a text file
examObj.txt
Paste text into OBJShape
modified OBJShape class
Use OBJShape to render the model in a JOGL-ES appl.
JOGL-ES ViewerES application

Figure 1. Converting a OBJ Model into a Class.

The OBJ model is loaded and displayed by a Java 3D application called ObjView. More importantly, ObjView generates a text file (examObj.txt) of JOGL-ES arrays and variables which represent the shape. These data structures must be manually pasted into a JOGL-ES class called OBJShape, which can render the shape defined by the arrays and variables.

ObjView is used solely to generate the arrays and other data. It plays no part in the JOGL-ES application, which only uses OBJShape to render the shape.

The model rendering performed by OBJShape is shown in Figure 2, as utilized in the ViewerES JOGL-ES application.

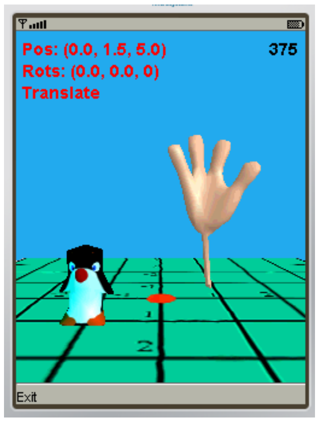

Figure 2. ViewerES with Penguin and Hand Models.

The version of ViewerES in Figure 2 employs two versions of OBJShape, renamed as PenguinModel and HandModel. PenguinModel contains arrays and variables representing a penguin, while HandModel holds the data for the hand model.

The 3D scene in the ViewerES application consists of a textured floor (first seen in the previous chapter), and one or more OBJ models. The camera can be moved around the scene via keyboard controls, utilizing the same technology as in the last chapter (i.e. the KeyCamera and Camera classes).

By default when OBJShape loads a model into a scene, it’s centered at the origin and its longest dimension is 1 unit. However, it's possible to adjust the shape’s position and size, as shown in Figure 2.

The manual pasting of the JOGL-ES arrays and variables into OBJShape is a bit low-tech, but it’s a only single copy-and-paste. It’s also good practice to rename the resulting class (e.g. I renamed the copy of OBJShape containing penguin data to PenguinModel).

1. Creating an OBJ Model

Almost every 3D graphics packages can import and export OBJ files. I've used Blender (http://www.blender.org/) and MilkShape 3D (http://chumbalum.swissquake.ch/) in the past: both are relatively easy to learn, are full of features, come with excellent tutorials, and are utilized by large communities of helpful users.

The complete Wavefront OBJ file format offers many advanced elements, such as free-form curves and surfaces, rendering interpolation, and shadowing. However, most OBJ exporters and loaders (including the Java 3D loader used in ObjView) only support polygonal shapes. A polygon's face is defined using vertices, with the optional inclusion of normals and texture coordinates. Color coordinates are not supported. Faces can be grouped together, and different groups can be assigned materials made from ambient, diffuse, and specular colors and textures. The material information is stored in a separate MTL text file.

A list of OBJ features can be found at http://www.csit.fsu.edu/~burkardt/data/obj/obj.html, and examples of MTL are at http://www.csit.fsu.edu/~burkardt/data/mtl/mtl.html.

OBJShape's design, and its use in Java ME, requires that a OBJ model consist of a single, relatively simple shape which utilizes a single material and/or texture. If the model's vertex count reaches the mid-thousands, then there's a good chance that OBJShape will fail, by exceeding Java ME's memory limit for array sizes.

The shape should have its y-axis vertically aligned, so it will appear upright in the Java 3D and JOGL-ES applications; some modeling packages orientate the y-axis to point out of the screen. Depending on the package, it may also be necessary to specify that the model's material settings be saved in a separate MTL file.

2. Using ObjView

Before ObjView can be called, Java 3D must be installed – it's an extension to Java SE, available from https://java3d.dev.java.net/.

Don't be (too) concerned if you don't know Java 3D. ObjView is a utility to generate the necessary JOGL-ES data structures, so there's no real need to understand how it works. However, if you *are* interested in Java 3D, then consider my website, "*Killer Game Programming in Java*" at http://fivedots.coe.psu.ac.th/~ad/jg/. ObjView is derived from an example in Chapter 16, "Loading and Manipulating External Models".

ObjView is started by supplying it with a OBJ filename:

```
java ObjView cube.obj
```

cube.obj and its MTL file, should be in the subdirectory "Models/" below the directory holding ObjView.java.

The OBJ file will be displayed on screen as shown in Figure 3.

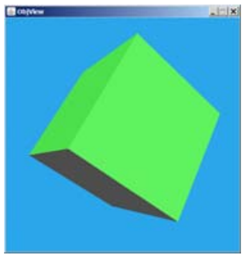

Figure 3. Cube.obj in ObjView

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 relies on the Java 3D library class ObjectFile to load the model. ObjectFile understands a small (but useful) subset of the OBJ and MTL formats (as detailed in ObjectFile’s class documentation). ObjView will raise an exception if supplied with a file using OBJ commands unknown to ObjectFile. For example:

```
> java ObjView cube.obj
Loading OBJ model from Models/cube.obj
Could not parse the contents of Models/cube.obj
com.sun.j3d.loaders.ParsingErrorException: Unrecognized token, line 4
>
```

Such errors can be fixed by editing the OBJ file (cube.obj in this case). Line 4 of that file is:

```
o Cube_Cube
```

“o” is an OBJ command for specifying a shape’s name that ObjectFile doesn’t recognize. The solution is to comment away the line:

```
# o Cube_Cube
```

Now ObjView will happily process cube.obj.

2.1. How ObjView Works

ObjView converts each model found in the OBJ file into a Java 3D Shape3D node. Below that node are Java 3D Geometry and Appearance nodes. The loading process triangulates and stripifies the model, so the Geometry node will be a Java 3D 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 4 shows the general idea.

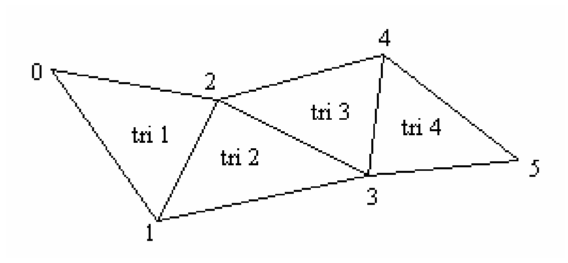

Figure 4. A Triangle Strip.

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. 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, color coordinates, normals, and vertices. However, since the OBJ file format doesn't support color coordinates, that information is absent from ObjView's data.

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 colors (e.g. ambient and diffuse colors), that information will be present in the shape's Appearance node.

2.2. The Generated JOGL-ES Data

ObjView extracts the points data and material information from the shape, and writes them to examObj.txt, in the form of several arrays and variables.

The data structures falls into three groups: model transformations, model coordinates, and material settings.

The model transformations data:

- `private static final float xCenter, yCenter, zCenter;`
  These three variables store the center point of the model, which is used by OBJShape to move the model to the origin at render time.
- `private static final float scaleFactor;`
  This variable is used by OBJShape to scale the model so its longest dimension is 1 unit in length.

The data related to the model’s coordinates:

- `private static final byte[] verts;`
  This array stores the vertices for the model, with each x, y, and z value scaled to be between -128 and 127 so it will fit into a byte. This saves space, but with some loss of positional accuracy. The vertices will form triangle strips when rendered.
- `private static final boolean hasNormals;`
  This boolean will be true or false depending on if the model uses normals.
- `private static final byte[] normals;`
  This array contains the normal values for the model, scaled to be between -128 and 127 so each one will fit into a byte. The array may be empty, meaning that the model doesn't have normals. In that case, hasNormals will be false.
- `private static final boolean hasTexture;`
  This boolean will be true or false depending on if the model uses texture coordinates.
- `private static final float[] texCoords;`
  This array holds the model's texture coordinates, in the form of floats between 0 and 1. This array may have no values, which means the model doesn't use texturing. In that case, hasTexture will be false.
- `private static final int[] strips;`
  This array contains the number of vertices in each triangle strip.

The data related to material settings:

- `private static final String TEX_FNM;`
  This string is assigned the name of the file containing the texture image. ObjView 'guesses' the name, since the Shape3D node for a model doesn't include filename information. The guess is the name of the OBJ file, with a ".png" extension.
- `private static final float[] ambientMat, emissiveMat, diffuseMat, specularMat;`
  These four arrays hold the RGBA values for the ambient, emissive, diffuse, and specular properties of the shape's material.
- `private static final float shininess;`
  This variable contains the material's shininess value, which can range from 0 to 128.

2.3. Data Structures for the Cube

The data output by ObjView for cube.obj is shown below. The cube utilizes normals, a green material, and no texture coordinates.

```
// position and scaling info
private static final float xCenter = -0.5f;
private static final float yCenter = -0.5f;
private static final float zCenter = -0.5f;
private static final float scaleFactor = 0.003921569f;

// verts coords [72 values/3 = 24 points]
private static final byte[] verts = {
  -128,-128,-128,  -128,127,-128,  127,-128,-128,  127,127,-128,
   127,-128,-128,   127,127,-128,  127,-128,127,   127,127,127,
   127,127,127,     127,127,-128, -128,127,127,   -128,127,-128,
  -128,-128,127,   -128,127,127,  -128,-128,-128, -128,127,-128,
  -128,-128,127,   -128,-128,-128, 127,-128,127,   127,-128,-128,
   127,-128,127,    127,127,127,  -128,-128,127,  -128,127,127
};  // end of verts[]

private static final boolean hasNormals = true;

// normals coords [72 values/3 = 24 points]
private static final byte[] normals = {
  0,0,-128,  0,0,-128,  0,0,-128,  0,0,-128,
  127,0,0,   127,0,0,   127,0,0,   127,0,0,
  0,127,0,   0,127,0,   0,127,0,   0,127,0,
 -128,0,0,  -128,0,0,  -128,0,0,  -128,0,0,
  0,-128,0,  0,-128,0,  0,-128,0,  0,-128,0,
  0,0,127,   0,0,127,   0,0,127,   0,0,127
};  // end of normals[]

private static final boolean hasTexture = false;
private static final float[] texCoords = {};   // not used

// an array holding triangle strip lengths
private static final int[] strips = {
    4, 4, 4, 4, 4, 4
};  // (24 points)


// materials
private static final String TEX_FNM = "";  // not used

private static final float[] ambientMat = {0.3f, 0.3f, 0.3f, 1.0f};
private static final float[] emissiveMat = {0.0f, 0.0f, 0.0f, 1.0f};
private static final float[] diffuseMat = {0.0f, 1.0f, 0.0f, 1.0f};
private static final float[] specularMat = {0.5f, 0.5f, 0.5f, 1.0f};
private static final float shininess = 96.0f;
```

The original cube has sides of 1 unit, and is centered on the origin.

The generated vertices in verts[] are scaled to be between -128 and 127, which introduces some inaccuracy into the cube's position. This is shown by the values for the cube's center, (-0.5, -0.5, -0.5). The scale factor, 0.003921569f, will reduce the longest dimension of the cube (255 units) to 1 unit at render time, which will adjust the center point to be very close to the origin.

The strips[] array indicates that the 24 vertices in verts[] are combined into six faces, with each face drawn using a triangle strip of four points. This is the same encoding technique that I used in the previous chapter, except that the faces here are specified in a different order. For example, the first four coordinates in verts[] define the triangle strip for the back face of the cube, and the next four are for the right face.

The normals[] array is organized into faces as well, with the same normal applied to the four points of each face. The first four normals specify a negative z-axis direction for the back face, and the next four point along the positive x-axis for the right face.

The important material variables in this example are ambientMat and diffuseMat, which sets the cube's ambient color to be dark grey and its diffuse color to be green.

3. The OBJShape class

The basic OBJShape class isn't functioning code since it doesn't contain the data structures for the shape it's supposed to render. These need to be manually copied into it from examObj.txt before OBJShape will compile and run.

OBJShape is very similar to the TexCube class from the previous chapter since it performs much the same duties. The OBJShape constructor uses the pasted-in coordinates arrays (verts[], normals[], etc) to create the buffers employed by JOGL-ES. The public draw() method renders those buffers.

3.1. Building the Shape

The constructor stores the position and scaling information used to move the shape from the origin and changes its default size. It also creates buffers for the shape's vertices, normals, and texture coordinates.

```
// globals
private GL10 gl;

// buffers for the model data arrays
private ByteBuffer vertsBuf;   // vertices
private ByteBuffer normsBuf;   // normal coords
private FloatBuffer tcsBuf;    // tex coords (as floats)

private int texNames[];   // for the texture name
private float xPos, yPos, zPos;  // position of model's center
private float scale;


public OBJShape(GL10 gl, float x, float y, float z, float sc)
// (x,y,z) is the model's position, and sc the scaling factor
{
  this.gl = gl;
  xPos = x; yPos = y; zPos = z;
  scale = sc;

  // create vertices buffer
  vertsBuf = ByteBuffer.allocateDirect(verts.length);
  vertsBuf.put(verts).rewind();

  if (hasNormals) {   // create normals buffer
    normsBuf = ByteBuffer.allocateDirect(normals.length);
    normsBuf.put(normals).rewind();
  }

  if (hasTexture) {  // create texture data
    ByteBuffer bb = ByteBuffer.allocateDirect(texCoords.length * 4);
    tcsBuf = bb.asFloatBuffer();
    tcsBuf.put(texCoords).rewind();

    loadTexture(TEX_FNM);

    // generate a texture name
    texNames = new int[1];
    gl.glGenTextures(1, texNames, 0);
  }
}  // end of OBJShape()
```

The code related to the normal and texture coordinates buffers is only executed if the shape has normals and texture coordinates.

The vertices and normals are stored in ByteBuffers, which is made possible by storing the original float values as integers between -128 and 127. However, there's no similar mapping possible for texture coordinates (which are floats between 0 and 1). Therefore, a FloatBuffer is employed to store the texture coordinates, created by mapping a float format over a ByteBuffer:

```
ByteBuffer bb = ByteBuffer.allocateDirect(texCoords.length * 4);
tcsBuf = bb.asFloatBuffer();
```

This approach ensures that the buffer is *direct*, which is required by the GL10.glDrawArrays() method. A direct buffer can be manipulated directly by the Java runtime system, without the use of buffer copying. ByteBuffer.allocateDirect() creates the direct byte buffer, and ByteBuffer.asFloatBuffer() retains this mode, while letting the buffer be treated as a container for floats.

The drawback of using a float buffer is its size – 4 bytes are needed for each value. There is one case when texture coordinates can be stored as bytes, which is employed by the cube in the previous chapter: if all the texture coordinate values are either 0 or 1, then they can be stored as integers in the byte buffer.

The loadTexture() method called from the constructor is unchanged from the last chapter. The texture is loaded as an Image object, and each pixel is stored as three bytes in a global ByteBuffer called texBuf.

3.2. Drawing the Shape

The draw() method begins by enabling the vertices, normals, and texture coordinates buffers, and setting the texture and material parameters. GL10.glDrawArrays() draws the shape, and the method finished by disabling the buffer capabilities and texturing.

```
public void draw()
{
  gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
  gl.glVertexPointer(3, GL10.GL_BYTE, 0, vertsBuf);  // use verts

  if (hasNormals) {  // use normals
    gl.glEnableClientState(GL10.GL_NORMAL_ARRAY);
    gl.glNormalPointer(GL10.GL_BYTE, 0, normsBuf);
  }

  if (hasTexture) {  // use texturing
    gl.glEnable(GL10.GL_TEXTURE_2D);
    gl.glEnableClientState(GL10.GL_TEXTURE_COORD_ARRAY);
    gl.glTexCoordPointer(2, GL10.GL_FLOAT, 0, tcsBuf);
    setTexture();
  }

  setMaterial();

  gl.glPushMatrix();
```

```
    // scale the model and move it using the user's settings
    gl.glTranslatef(xPos, yPos, zPos);   // move to (x,y,z) pos
    if (scale != 1.0f)
      gl.glScalef(scale, scale, scale);  // uniform scaling

    // center and scale the model
    gl.glScalef(scaleFactor, scaleFactor, scaleFactor);
    gl.glTranslatef(-xCenter, -yCenter, -zCenter);  // move to origin

    int pos = 0;
    int stripLen;
    for (int i = 0; i < strips.length; i++) {  // draw each strip
      gl.glDrawArrays(GL10.GL_TRIANGLE_STRIP, pos, strips[i]);
                // vertices, (normals,) (tex coords) for the strip
      pos += strips[i];
    }

  gl.glPopMatrix();

  gl.glDisableClientState(GL10.GL_VERTEX_ARRAY);
  gl.glDisableClientState(GL10.GL_NORMAL_ARRAY);
  gl.glDisableClientState(GL10.GL_TEXTURE_COORD_ARRAY);

  gl.glDisable(GL10.GL_TEXTURE_2D);
}  // end of draw()
```

The call to GL10.glTexCoordPointer() uses the data type GL10.GL_FLOAT since the buffer is being manipulated as a sequence of floats.

The GL10.glDrawArrays() calls are preceded by two groups of translations and scalings. The first applies the values supplied in the constructor, while the second group places the shape at the origin and scales it to be at most 1 unit long in its longest dimension. The ordering of these calls is important.

Conceptually, the user sees the shape transformations (translations, rotations, and scalings) in *reverse* order to their execution ordering in the code. From the user's point-of-view, first the shape is moved to the origin and given unit length, then scaled to the desired size and moved to the user-supplied position (at (xPos, yPos, zPos)). The latter scaling is applied when the object is centered at the origin, so doesn't affect the user-specified translation which is applied relative to the object's center.

The transformations are nested between calls to GL10.glPushMatrix() and GL10.glPopMatrix() so they only apply to the shape.

Textures and Materials

The setTexture() method is unchanged from previously:

```
private void setTexture()
{
  gl.glBindTexture(GL10.GL_TEXTURE_2D, texNames[0]);  // use tex name

  // specify the texture for the currently bound tex name
  gl.glTexImage2D(GL10.GL_TEXTURE_2D, 0, GL10.GL_RGB,
                  imWidth, imHeight, 0,
                  GL10.GL_RGB, GL10.GL_UNSIGNED_BYTE, texBuf);
  // set the minification/magnification techniques
  gl.glTexParameterx(GL10.GL_TEXTURE_2D,
               GL10.GL_TEXTURE_MIN_FILTER, GL10.GL_LINEAR);
  gl.glTexParameterx(GL10.GL_TEXTURE_2D,
               GL10.GL_TEXTURE_MAG_FILTER, GL10.GL_LINEAR);
} // end of setTexture()
```

It binds the shape's texture ID to the GL10 state, then links the shape's texture buffer to that ID. This means that the texture will be used when the shape's texture coordinates are processed by GL10.glDrawArrays().

setMaterial() applies the material property arrays generated by ObjView for ambient, diffuse, specular, and emissive lighting, and sets the shininess value.

```
private void setMaterial()
{
  gl.glMaterialfv(GL10.GL_FRONT_AND_BACK,
                          GL10.GL_AMBIENT, ambientMat, 0);
  gl.glMaterialfv(GL10.GL_FRONT_AND_BACK,
                          GL10.GL_DIFFUSE, diffuseMat, 0);
  gl.glMaterialfv(GL10.GL_FRONT_AND_BACK,
                          GL10.GL_SPECULAR, specularMat, 0);
  gl.glMaterialf(GL10.GL_FRONT_AND_BACK,
                          GL10.GL_SHININESS, shininess);
  gl.glMaterialfv(GL10.GL_FRONT_AND_BACK,
                          GL10.GL_EMISSION, emissiveMat, 0);
}  // end of setMaterial()
```

4. The ViewerES Application

ViewerES shows how OBJShape variants can be used in a JOGL-ES application to load, position, scale, and display OBJ models. Figure 1 shows a single model loaded into ViewerES, and Figure 2 has two models (a penguin and a hand), More models can be easily added to ViewerES (at the price of slower rendering times).

The class diagrams for the version of ViewerES with a penguin and hand are given in Figure 5. Only the public methods are shown.

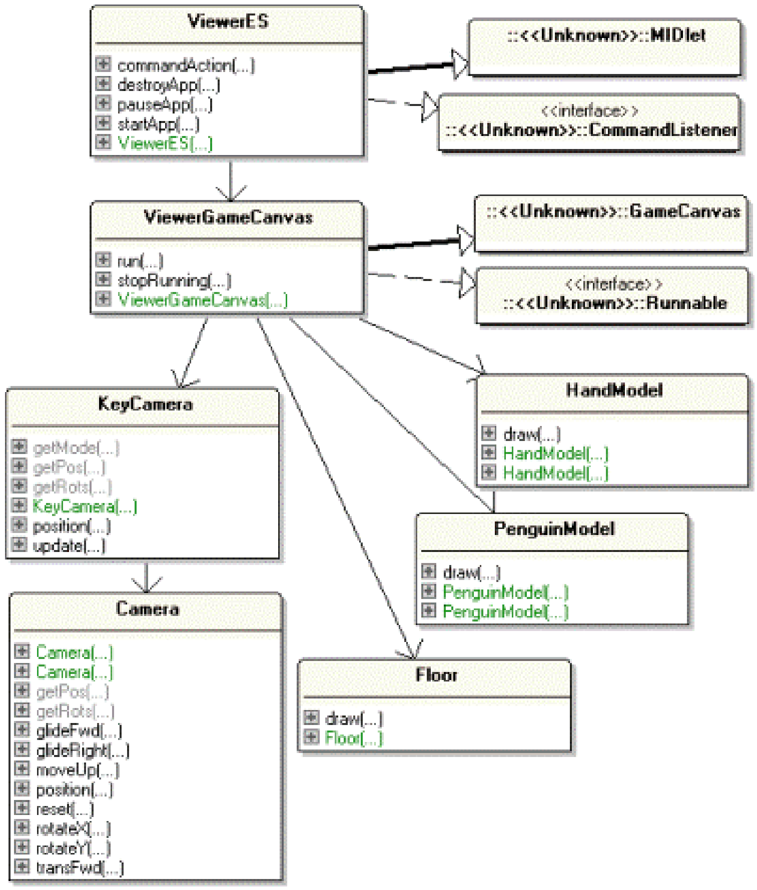

Figure 5. ViewerES Class Diagrams

The ViewerES MIDlet starts the ViewerGameCanvas thread and places it on-screen. ViewerGameCanvas employs the same animation algorithm as the RotBoxES example from the last chapter. It also reuses KeyCamera and Camera, and a very slightly modified version of the Floor class.

HandModel and PenguinModel are renamed copies of the OBJShape class. ObjView was run twice to generate the data for these classes, once with a hand model, and then with a penguin. The data structures output to examObj.txt were pasted into renamed copies of OBJShape.

4.1. Animating the Scene

The run() method in ViewerGameCanvas is unchanged from the previous chapter:

```
public void run()
/* Contains three stages: initialization, animation loop, shutdown.
   The animation loop has three parts: update, draw, maybe sleep.
*/
{
  if (!initGraphics())
    return;   // give up if there's an error during initialization

  initScene();

  long startTime;
  while (isRunning) {
    startTime = System.currentTimeMillis();

    keyCamera.update( getKeyStates() );
    drawScene();

    frameDuration = System.currentTimeMillis() - startTime;
    try { // sleep a bit maybe, so one iteration takes PERIOD ms
      if (frameDuration < PERIOD)
        Thread.sleep(PERIOD - (int)frameDuration);
    }
    catch (InterruptedException e){}
  }

  shutdown();
} // end of run()
```

The changes required to display the models are located deep inside initScene() and drawScene().

4.2. Scene Initialization

The creation of the hand and penguin shape instances is carried out in createScenery() (which is called from initScene()).

```
// globals
private Floor floor;
private PenguinModel penguin;   // the penguin model
private HandModel hand;         // the hand model


private void createScenery()
// create the floor and the OBJ models
{
  floor = new Floor(gl, "/bigGrid.png", 8);   // 8 by 8 size

  hand = new HandModel(gl, 1.0f, 1.5f, -1.0f, 3.0f);
                          // (x, y, z) and scale
  penguin = new PenguinModel(gl, -1.0f, 0.5f, 1.0f, 1.0f);
} // end of createScenery()
```

HandModel and PenguinModel are renamed versions of OBJShape after I had added the data structures for the hand and penguin model to them.

HandModel's input arguments specify that it be positioned at (1, 1.5, -1) and enlarged by a factor of three, while the penguin is moved to (-1, 0.5, 1) and not scaled. The models' (X, Z) positions can be confirmed by looking at the floor grid in Figure 2. The y-axis values were arrived at by a process of trial-and-error so the base of the models would appear to be resting on the floor's surface.

4.3. Scene Rendering

Only the 3D drawing code needs adjusting inside drawSceneGL() (which is called from drawScene()).

```
private void drawSceneGL()
// draw the scene (the camera, the light source, floor, the models)
{
  // wait until OpenGL ES is available before starting to draw
  egl.eglWaitNative(EGL10.EGL_CORE_NATIVE_ENGINE, g2d);

  // clear color and depth buffers
  gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);

  // set modeling and viewing transformations
  gl.glMatrixMode(GL10.GL_MODELVIEW);
  gl.glLoadIdentity();

  keyCamera.position();   // position the camera

  // set light direction
  gl.glLightfv(GL10.GL_LIGHT0, GL10.GL_POSITION, LIGHT_DIR, 0);

  floor.draw();
  hand.draw();
  penguin.draw();

  // wait until all Open GL ES tasks are finished
  gl.glFinish();
  egl.eglWaitGL();
  } // end of drawSceneGL()
```

The only change is to call draw() for the HandModel and PenguinModel instances:

```
  hand.draw();
  penguin.draw();
```

The draw() method for Floor has been changed slightly. The method now starts by disabling the scene's light, and switches it back on at the end:

```
public void draw()  // for Floor
{
  gl.glDisable(GL10.GL_LIGHTING);    // disable lighting

  // draw the floor...

  gl.glEnable(GL10.GL_LIGHTING);      // switch lighting back on
}  // end of draw()
```

When the floor isn't affected by the lighting, its texture is rendered more clearly. The absence of light also means that the floor's data structures don't need to include normal values.

5. Displaying Multiple Shapes

When ObjView detects multiple shapes in a single OBJ file, each one is converted to a separate Shape3D node with its own Geometry and Appearance nodes. Several copies of the JOGL-ES data structures are output, one set for each shape.

OBJShape is designed to only process the data structures for one shape. This means that multiple shapes output by ObjView require multiple copies of OBJShape, one for each shape. The issues of scaling and positioning must also be considered.

The vertices of every shape are scaled to between -128 and 127. This means that shapes which are different sizes in the OBJ file will be rendered at about the same size in ViewerES. To overcome this, the user must supply suitable scaling factors when the shape's constructor is called from ViewerES.

The other problem is that the relative positions of the shapes will be wrong in ViewerES. This is due to each OBJShape instance placing its shape at the origin by default. This can be fixed by supplying position coordinates when the OBJShape versions are created in ViewerES.

6. Large Models

A serious concern is that the ObjView generates very long arrays when supplied with medium-to-large models. For example, I had hoped to load a human figure into ViewerES (Figure 6 shows it displayed by ObjView).

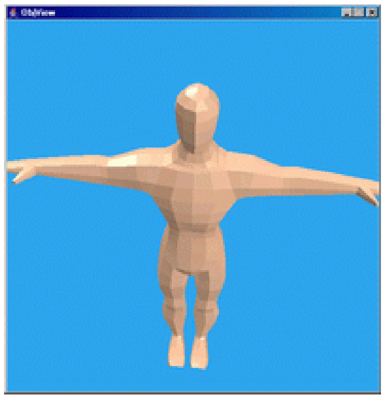

Figure 6. A Figure in ObjView.

When supplied with the model, ObjView generates verts[] and normals[] arrays holding 9600 integers. These arrays are too big for the WTK, which reports a "code too large" error message when it tries to compile verts[]. Unfortunately, the emulator only allows an array to be at most 32 KB large.

Currently, when ObjView outputs an array with 4000 elements or more, it prints a warning message to the screen, and inserts a comment next to the offending array written into examObj.txt.