



Loading Models

This chapter continues using the 3D world of Chapter 16, but as a setting to demonstrate four new coding techniques:

- * The *loading and positioning* of 3D models created using the Wavefront OBJ file format.
- The selection (*picking*) of objects in the scene by clicking them with the mouse. OpenGL's selection mode is utilized to implement this feature.
- 3D *sound*, in this case the chirping of a penguin, which varies as the user moves around the scene (nearer and farther from a penguin model). It's implemented using JOAL and the JOALSoundMan class introduced in Chapters 13.
- *Fog* shrouding the scene, making it harder to find the models.

The vast majority of this chapter (nearly two-thirds of it) is concerned with the first technique, leading to the development of an OBJ model loader package called OBJLoader and an application called ModelLoaderGL that uses the loader. The loader *and* the other three topics (picking, sound, and fog) are required by the chapter's other example, TourModelsGL.

ModelLoaderGL and TourModelsGL are briefly explained in the next section.

Overviews of the Examples

ModelLoaderGL is a simple model viewer, shown in Figure 17-1, displaying a penguin.

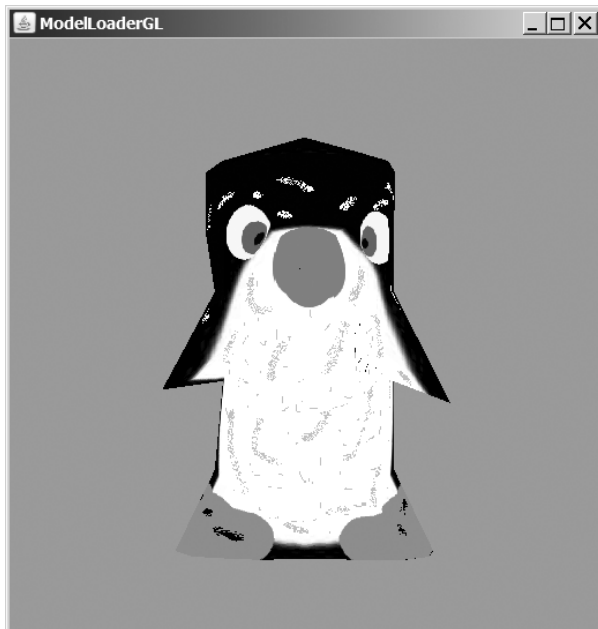


Figure 17-1. *Displaying a penguin*

The model's name is specified on the command line, along with the display size and whether the model should rotate. The model is loaded and positioned using the OBJLoader package developed in this chapter. ModelLoaderGL can be employed to check what an object looks like before it's loaded into a more complex application.

TourModelsGL is a simple game based around trying to find a penguin and a couch hidden in the fog. The search is made a bit harder by including other models in the scene, but the player is given some aural help because the penguin is constantly chirping. When the user clicks the mouse on the penguin or the couch, a message is printed to standard output. Needless to say, a lot could be done to make this a more exciting gaming experience. Figures 17-2 and 17-3 show TourModelsGL without the fog and with it. The four OBJ models are a penguin wrapped in a single texture, a red couch, a red rose in a blue vase, and a racing car decorated with several colors and textures.

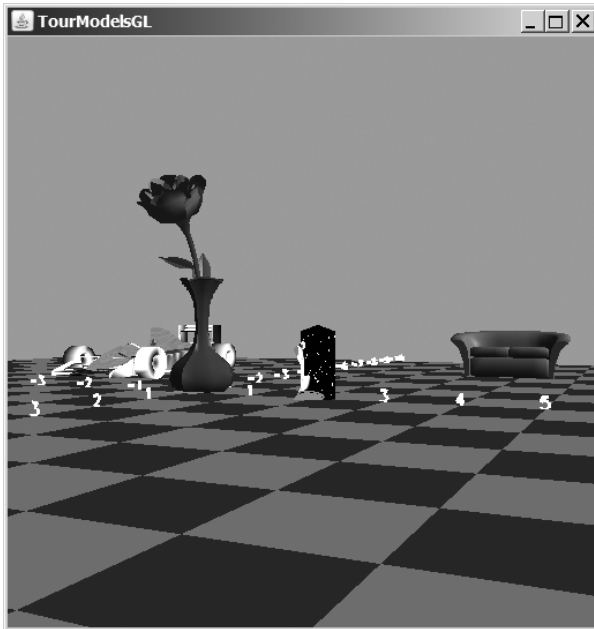


Figure 17-2. *TourModelsGL* with a clear blue sky

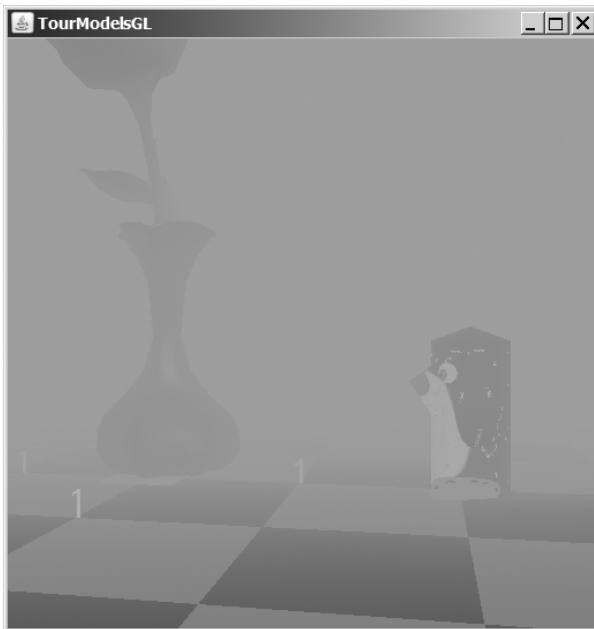


Figure 17-3. *TourModelsGL* after the fog has descended

TourModelsGL reuses the checkerboard floor from the previous chapter, but I've removed the skybox, the billboard trees, the rotating sphere, the splash screen, and the "Game Over" message in order to simplify the code.

The OBJ File Format

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 from Chapter 7) only support polygonal shapes. A polygon's face is defined using vertices, with the optional inclusion of normals and texture coordinates. 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>. The description in this chapter will focus on the core elements found in the Java 3D loader (accessed via the ObjectFile class). Some unnecessary details will be left out, but they can be found in the ObjectFile class documentation.

An OBJ file is a text file consisting of lines of statements, comments, and blank lines. Comments start with # and are ignored. Each statement begins with an operator name indicating how to process the data that follows it on the line. There are three types of basic OBJ statements: those that are shape-related, those for grouping, and those for materials. I'll briefly explain the format of each.

Shape Statements

The three floats in a v statement specify a vertex's position:

```
v float float float
```

The first vertex listed in an OBJ file (i.e., the first v statement) is assigned an index value of 1, and subsequent vertices are numbered sequentially.

The floats in a vn statement specify a normal:

```
vn float float float
```

The first normal in the file (i.e., the first vn statement) is assigned index 1, and subsequent normals are numbered sequentially.

A vt statement is used to define a 2D or 3D texture coordinate:

```
vt float float [float]
```

The square brackets mean that the third float argument is optional.

The first texture coordinate in a file (i.e., the first vt statement) is index 1, and subsequent textures are numbered sequentially.

A polygonal face is specified using an f statement, which can employ three different formats:

```
f int int int ...
```

or

```
f int/int int/int int/int ...
```

or

```
f int/int/int int/int/int int/int/int ...
```

A face can be defined as a sequence of vertex indices (the first format), or vertices and textures indices (the second format), or vertices, textures, and normal indices (the last format). I'll call each collection of indices (e.g., int/int/int) a *term*.

When a term has three elements, it's possible for the texture indices to be left out if they haven't been defined for the model, resulting in the f statement:

```
f int//int int//int int//int ...
```

The number of terms making up a face depends on its shape; often it's a triangle (which needs three terms to define it) or a quadrilateral (four terms).

The first face in the file (i.e., the first f statement) is assigned index 1, and subsequent faces are numbered sequentially.

Grouping Statements

A g statement specifies a group name:

```
g name
```

Faces defined after a g statement will be added to the named group. Named groups are a useful way of referring to a collection of faces; for example, Java 3D maps each named group to a Shape3D object at load time. This makes it easier to apply transformations or appearance changes to sub-components of the model:

The s statement defines a “smoothing” group:

```
s int
```

Subsequent face definitions will be members of the numbered group and have their normals calculated as if they form part of the same surface.

Material Use Statements

The MTL file named in the mtlib statement contains material definitions that can be used in the rest of the OBJ file:

```
mtlib filename
```

When a usemtl statement is encountered in the OBJ file, all subsequent faces will be rendered with the named material obtained from the MTL file:

```
usemtl name
```

The MTL File Format

An MTL file consists of a series of material definitions made up of statements, comments, and blank lines. Comments start with # and are ignored.

Each new material definition begins with a newmtl statement that specifies a material's name:

```
newmtl name
```

The name is used by the usemtl statement in an OBJ file to refer to the material.

A material's properties (e.g., its color, transparency, texture) are defined using material statements (e.g., Ka, Kd) given on the lines after the newmtl statement, as in this example:

```
newmtl penguinMat
# material properties for the penguin model
```

```

Ka 0.01 0.75 0.75
Kd 0.90 0.69 0.90
Ks 0.93 0.93 0.93
Ns 445
illum 2
map_Kd penguin.gif

```

The following are the basic material statements:

```
Ka r g b
```

The three floats (r, g, and b) define the ambient RGB color of the material.

```
Kd r g b
```

The three floats specify the diffuse RGB color of the material.

```
Ks r g b
```

The r, g, and b floats represent the specular color of the material.

```
d alpha
```

or

```
Tr alpha
```

The alpha value specifies the transparency of the material. Java 3D doesn't support either the d or Tr statement.

```
Ns s
```

The shininess of the material is set with the s float value. If no Ns statement is supplied for a material, the material's default shininess will be 0.0f (i.e., it will not be shiny).

```
illum n
```

The illum statement sets the illumination mode for a material. If n is 1, the material has no specular highlights and the Ks values are ignored. If n is 2, specular highlights are present and will utilize the Ks value. When n is 0, lighting is disabled.

```
map_Ka filename
```

The named file contains a texture for the material. The MTL specification states that this should be an ASCII dump of RGB values, but most tools (including the Java 3D loader) also support standard image files (GIF, JPG, PNG). The image must have a size that is a power of 2 (e.g., 64 by 64, 128 by 128).

The OBJ File Loader

My OBJLoader package can load models and materials from simple OBJ and MTL files. The shape statements (v, vt, vn, f) and material statements (mtllib and usemtl) are understood, but grouping operations are ignored (g and s). MTL statements are processed, except for transparency (d, Tr) and illumination (illum). Textures and colors can't be blended together; the presence of a texture for a material (map_Ka) disables any color settings (e.g., Ka, Kd, Ks).

Class diagrams for the OBJLoader package are shown in Figure 17-4.

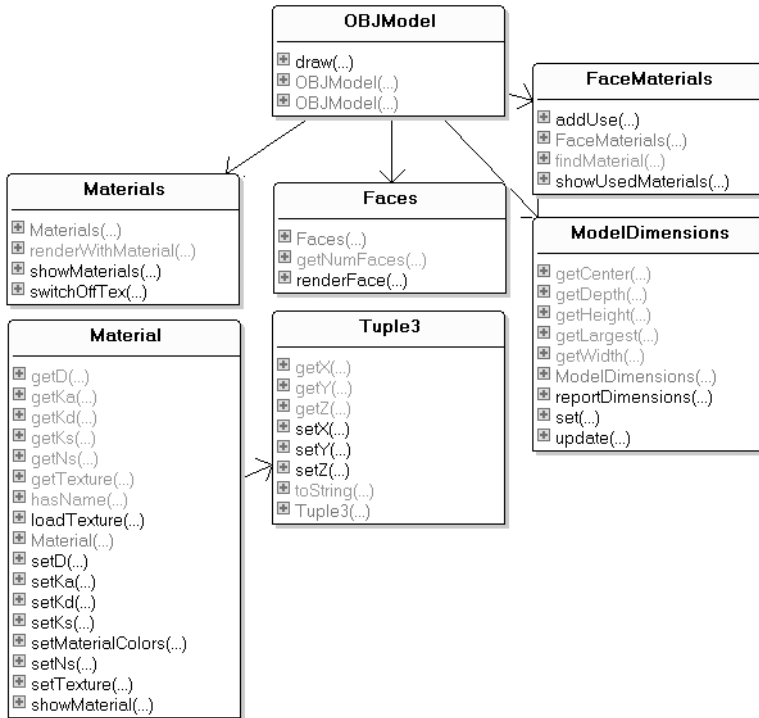


Figure 17-4. The class diagrams for the OBJLoader package

The OBJModel class loads the OBJ model, centers it at the origin and scales it to a size supplied in its constructor. The OpenGL commands for rendering the model are stored in a display list and are executed by calling OBJModel.draw().

The Faces class stores information about each face of the model. When OBJModel is constructing the display list, it calls Faces.renderFace() to render a face in terms of the loaded model's vertices, texture coordinates, and normals.

The Materials class loads material details from the MTL file, storing them as Material objects in an ArrayList.

FaceMaterials stores the face indices where materials are first used. This information is used to load the right material when a given face needs to be drawn.

ModelDimensions holds the smallest and largest coordinates for the model along its three dimensions. These are utilized to calculate the model's width, height, depth, its largest dimension, and its center point. ModelDimensions is employed by OBJModel to resize and center the loaded model.

Tuple3 is a general-purpose class for storing a three-element tuple. It's used in several places in the package to store vertices, normals, and texture coordinates as Tuple3 objects.

While writing the OBJLoader package, I got a lot of help and inspiration from examining the loaders written by Evangelos Pournaras in his *JautOGL* game (<http://today.java.net/pub/a/today/2006/10/10/development-of-3d-multiplayer-racing-game.html> and <https://jautogl.dev.java.net/>) and Kevin Glass's *Asteroids* tutorial (<http://www.cokeandcode.com/asteroidstutorial/>).

Reading in the OBJ File

OBJModel is responsible for reading in the OBJ file, line by line, and processing the statements it finds. The shape data (vertices, texture coordinates, and normals) is stored in ArrayLists of Tuple3 objects:

```
private ArrayList<Tuple3> verts;
private ArrayList<Tuple3> normals;
private ArrayList<Tuple3> texCoords;
```

OBJModel also initializes the other package objects:

```
private Faces faces;           // holds model faces
private FaceMaterials faceMats; // materials used by the faces
private Materials materials;   // materials defined in MTL file
private ModelDimensions modelDims; // model dimensions
```

The parsing of the OBJ file is carried out in readModel():

```
private void readModel(BufferedReader br)
{
    boolean isloaded = true; // hope things will go okay

    int lineNum = 0;
    String line;
    boolean isFirstCoord = true;
    boolean isFirstTC = true;
    int numFaces = 0;

    try {
        while (((line = br.readLine()) != null) && isloaded) {
            lineNum++;
            if (line.length() > 0) {
                line = line.trim();

                if (line.startsWith("v ")) { // vertex
                    isloaded = addVert(line, isFirstCoord);
                    if (isFirstCoord)
                        isFirstCoord = false;
                }
                else if (line.startsWith("vt")) { // tex coord
                    isloaded = addTexCoord(line, isFirstTC);
                    if (isFirstTC)
                        isFirstTC = false;
                }
                else if (line.startsWith("vn")) // normal
                    isloaded = addNormal(line);
                else if (line.startsWith("f ")) { // face
                    isloaded = faces.addFace(line);
                    numFaces++;
                }
                else if (line.startsWith("mtllib ")) // load material
                    materials = new Materials( line.substring(7) );
                else if (line.startsWith("usemtl ")) // use material
                    faceMats.addUse( numFaces, line.substring(7));
                else if (line.charAt(0) == 'g') { // group name
                    // not implemented
                }
            }
        }
    }
}
```



```

        else if (line.charAt(0) == 's') { // smoothing group
            // not implemented
        }
        else if (line.charAt(0) == '#') // comment line
            continue;
        else
            System.out.println("Ignoring line " + lineNum +
                               " : " + line);
    }
}
}
}
catch (IOException e) {
    System.out.println( e.getMessage() );
    System.exit(1);
}

if (!isLoading) {
    System.out.println("Error loading model");
    System.exit(1);
}
} // end of readModel()

```

The `v`, `vt`, and `vn` statements trigger code that adds a vertex, a texture coordinate, and a normal `Tuple3` object to the `verts`, `texCoords`, and `normals` `ArrayLists`. For example, `addVert()` adds a tuple to `verts` and updates the model dimension's information:

```

private boolean addVert(String line, boolean isFirstCoord)
{
    Tuple3 vert = readTuple3(line);
        // store (x,y,z) from "v x y z" in a tuple
    if (vert != null) {
        verts.add(vert);
        if (isFirstCoord)
            modelDims.set(vert); // add first coordinate
        else
            modelDims.update(vert); // add a later coordinate
        return true;
    }
    return false;
} // end of addVert()

```

In `readModel()`, an `f` statement is handled by `addFace()` in the `Faces` class, and `mtllib` triggers the creation of a `Materials` object that reads in the named MTL file. A `usemtl` statement causes the `FaceMaterials` object to record the current face index and the named material. The material will be utilized when that face and subsequent ones need to be rendered.

Reading a Face

The `Faces` object stores information about all the face statements in the OBJ file.

The data for a single face is stored in three arrays of vertex indices, texture coordinate indices, and normal indices; the indices come from the face's `f` statement.

For example, if the statement is

```
f 10/12/287 9/14/287 8/16/287
```

the vertex indices array will hold `{10, 9, 8}`; the texture coordinate indices array will contain `{12, 14, 16}`; and the normal indices array is `{287, 287, 287}`.

All the faces data is held in three ArrayLists called `facesVertIdxs`, `facesTexIdxs`, and `facesNormIdxs`. `facesVertIdxs` stores all the vertex indices arrays, `facesTexIdxs` all the texture coordinate indices arrays, and `facesNormIdxs` the normal indices arrays:

```
private ArrayList<int[]> facesVertIdxs; // for the vertices indices
private ArrayList<int[]> facesTexIdxs; // texture coords indices
private ArrayList<int[]> facesNormIdxs; // normal indices
```

The `Faces.addFace()` method (called from `OBJModel.readModel()`) pulls the terms out of an `f` line, builds arrays for the vertices, texture coordinates, and normals indices, and adds those arrays to the ArrayLists.

Things are complicated by the fact that terms may be missing texture and normal information:

```
public boolean addFace(String line)
{
    try {
        line = line.substring(2); // skip the "f "
        StringTokenizer st = new StringTokenizer(line, " ");
        int numTokens = st.countTokens(); // number of v/vt/vn tokens
        // create arrays to hold the v, vt, vn indices
        int v[] = new int[numTokens];
        int vt[] = new int[numTokens];
        int vn[] = new int[numTokens];

        for (int i = 0; i < numTokens; i++) {
            String faceToken = addFaceVals(st.nextToken());
            // get a v/vt/vn token

            StringTokenizer st2 = new StringTokenizer(faceToken, "/");
            int numSeps = st2.countTokens();
            // how many '/'s are there in the token

            v[i] = Integer.parseInt(st2.nextToken());
            vt[i] = (numSeps > 1) ? Integer.parseInt(st2.nextToken()) : 0;
            vn[i] = (numSeps > 2) ? Integer.parseInt(st2.nextToken()) : 0;
            // add 0s if the vt or vn index values are missing;
            // 0 is a good choice since real indices start at 1
        }
        // store the indices for this face
        facesVertIdxs.add(v);
        facesTexIdxs.add(vt);
        facesNormIdxs.add(vn);
    }
    catch (NumberFormatException e) {
        System.out.println("Incorrect face index");
        System.out.println(e.getMessage());
        return false;
    }
    return true;
} // end of addFace()
```

Reading in an MTL File

The processing of an MTL file is handled by a Materials object. `readMaterials()` parses the MTL file line by line, adding Material objects to a materials ArrayList:

```
// global
public ArrayList<Material> materials;
    // stores the Material objects built from the MTL file data

private void readMaterials(BufferedReader br)
{
    try {
        String line;
        Material currMaterial = null; // current material

        while ((line = br.readLine()) != null) {
            line = line.trim();
            if (line.length() == 0)
                continue;

            if (line.startsWith("newmtl ")) { // new material
                if (currMaterial != null) // save previous material
                    materials.add(currMaterial);

                // start collecting info for new material
                currMaterial = new Material(line.substring(7));
            }
            else if (line.startsWith("map_Kd ")) { // texture filename
                String fileName = MODEL_DIR + line.substring(7);
                currMaterial.loadTexture( fileName );
            }
            else if (line.startsWith("Ka ")) // ambient color
                currMaterial.setKa( readTuple3(line) );
            else if (line.startsWith("Kd ")) // diffuse color
                currMaterial.setKd( readTuple3(line) );
            else if (line.startsWith("Ks ")) // specular color
                currMaterial.setKs( readTuple3(line) );
            else if (line.startsWith("Ns ")) { // shininess
                float val = Float.valueOf(line.substring(3)).floatValue();
                currMaterial.setNs( val );
            }
            else if (line.charAt(0) == 'd') { // alpha
                float val = Float.valueOf(line.substring(2)).floatValue();
                currMaterial.setD( val );
            }
            else if (line.startsWith("illum ")) { // illumination model
                // not implemented
            }
            else if (line.charAt(0) == '#') // comment line
                continue;
            else
                System.out.println("Ignoring MTL line: " + line);
        }
        materials.add(currMaterial);
    }
}
```

```

    catch (IOException e)
    { System.out.println(e.getMessage()); }
} // end of readMaterials()

```

When a `newmtl` statement is encountered, the current `Material` object is added to the `materials` `ArrayList`, and a new object is created, ready to be filled with color and texture information read from subsequent statements.

The `Ka`, `Kd`, `Ks`, `Ns`, and `d` values are passed to the `Material` object via `set` methods. When `readMaterials()` sees a `map_Kd` statement, it calls `loadTexture()` in the current `Material` object:

```

// in the Material class
// global texture info
private String texFnm;
private Texture texture;

public void loadTexture(String fnm)
{
    try {
        texFnm = fnm;
        texture = TextureIO.newTexture( new File(texFnm), false);
        texture.setTextureParameter(GL.GL_TEXTURE_MAG_FILTER, GL.GL_NEAREST);
        texture.setTextureParameter(GL.GL_TEXTURE_MIN_FILTER, GL.GL_NEAREST);
    }
    catch(Exception e)
    { System.out.println("Error loading texture " + texFnm); }
} // end of loadTexture()

```

Recording Material Use

A subtle aspect of the `OBJ` format is how materials are linked to faces. After a material is named in a `usemtl` statement, all subsequent faces will use it for rendering until another `usemtl` line is encountered, as in the following example:

```

usemtl couch
f 10/10/287 9/9/287 8/8/287
f 10/10/287 8/8/287 7/7/287
f 10/10/287 7/7/287 6/6/287
f 10/10/287 6/6/287 5/5/287
// many more faces ...

```

All the faces defined after the `usemtl` line will use the `couch` material at render time.

When `OBJModel.readModel()` encounters a `usemtl` statement, it stores the information by passing the current face index and material name to a `FaceMaterials` object:

```

else if (line.startsWith("usemtl ")) // use materials
    faceMats.addUse( numFaces, line.substring(7));

```

`numFaces` contains the current index, and the substring is the material name.

A `HashMap` in the `FaceMaterials` object is employed to connect face indices to material names:

```

private HashMap<Integer, String>faceMats;

```

FaceMaterials.addUse() adds a new face index and material name to faceMats:

```
public void addUse(int faceIdx, String matName)
{
    // store the face index and the material it uses
    if (faceMats.containsKey(faceIdx)) // face index already present
        System.out.println("Face index " + faceIdx +
            " changed to use material " + matName);

    faceMats.put(faceIdx, matName);

    // other non-relevant code...
} // end of addUse()
```

Centering and Resizing a Model

After the OBJ and MTL files have been read in, OBJModel calls centerScale() to center the model at the origin and resize it. The size is either specified in OBJModel's constructor or defaults to 1 unit.

centerScale() relies on the ModelDimensions object, which stores the minimum and maximum coordinates for the model and includes methods for calculating the model's largest dimension and center point:

```
// global
private float maxSize; // for scaling the model

private void centerScale()
{
    // get the model's center point
    Tuple3 center = modelDims.getCenter();

    // calculate a scale factor
    float scaleFactor = 1.0f;
    float largest = modelDims.getLargest();
    if (largest != 0.0f)
        scaleFactor = (maxSize / largest);
    System.out.println("Scale factor: " + scaleFactor);

    // modify the model's vertices
    Tuple3 vert;
    float x, y, z;
    for (int i = 0; i < verts.size(); i++) {
        vert = (Tuple3) verts.get(i);
        x = (vert.getX() - center.getX()) * scaleFactor;
        vert.setX(x);
        y = (vert.getY() - center.getY()) * scaleFactor;
        vert.setY(y);
        z = (vert.getZ() - center.getZ()) * scaleFactor;
        vert.setZ(z);
    }
} // end of centerScale()
```

centerScale() directly modifies the model's vertices to modify its scale. An alternative approach, which may seem more efficient, is to apply translation and scaling transformations to the geometry. Unfortunately, a scaling transformation also affects the model's normals, so they're no longer guaranteed to be of unit length. This will cause the model's color to change at render time and textures to be positioned incorrectly.

Creating a Display List for the Model

Once OBJModel has centered and scaled the model, it can render it to a display list. Subsequent calls to OBJModel.draw() will execute the list, greatly improving the drawing speed.

OBJModel.drawToList() creates the display list:

```
// globals
private int modelDispList; // the model's display list
private boolean flipTexCoords = false;
    // whether the texture coords should be vertically flipped

private void drawToList(GL gl)
{
    modelDispList = gl.glGenLists(1);
    gl.glNewList(modelDispList, GL.GL_COMPILE);

    gl.glPushMatrix();
    // render the model face-by-face
    String faceMat;
    for (int i = 0; i < faces.getNumFaces(); i++) {
        faceMat = faceMats.findMaterial(i);
        // get material used by face i
        if (faceMat != null)
            flipTexCoords = materials.renderWithMaterial(faceMat, gl);
            // render using that material
        faces.renderFace(i, flipTexCoords, gl);    // draw face i
    }
    materials.switchOffTex(gl);
    gl.glPopMatrix();

    gl.glEndList();
} // end of drawToList()
```

drawToList() draws each face by calling Faces.renderFace() in a loop. Before rendering a face, it checks whether the face's index is associated with a material (with FaceMaterials.findMaterial()). If a material change is required, it is loaded into OpenGL by Materials.renderWithMaterial().

Materials.renderWithMaterial() assigns a value to the flipTexCoords boolean, which is passed to FaceMaterials.findMaterial(). The boolean indicates whether the texture's coordinates need to be vertically flipped when they're mapped to the shape's coordinates at render time.

Texturing may still be enabled at the end of the loop, so a call to Materials.switchOffTex() makes sure that it's switched off and that the lights are reenabled.

Finding a Material

The FaceMaterial instance, faceMats, stores a HashMap of face indices mapped to material names. When FaceMaterial.findMaterial() is called with a face index, the retrieval of the associated material name is a fast lookup:

```
// in the FaceMaterial class
private HashMap<Integer, String>faceMats;
    // the face index (integer) where a material is first used

public String findMaterial(int faceIdx)
{ return (String) faceMats.get(faceIdx); }
```

If the index isn't in the `HashMap`, the method returns null, which is tested for back in `OBJ-Model.drawToList()`.

Rendering with a Material

If the face that's about to be rendered has an associated material, it needs to be loaded first.

`Materials.renderWithMaterial()` has two types of material to deal with colors and textures. Also, before a new material can be loaded, any existing texturing must be disabled:

```
// in the Materials class
// globals
private String renderMatName = null;
// stores the material currently being used for rendering
private boolean flipTexCoords = false;
// whether the model's texture coords should be vertically flipped

public boolean renderWithMaterial(String faceMat, GL gl)
{
    if (!faceMat.equals(renderMatName)) { // is faceMat new?
        renderMatName = faceMat;
        switchOffTex(gl); // switch off any previous texturing

        // set up new rendering material
        Texture tex = getTexture(renderMatName);
        if (tex != null) { // use the material's texture
            switchOnTex(tex, gl);
            flipTexCoords = tex.getMustFlipVertically();
        }
        else // use the material's colors
            setMaterialColors(renderMatName, gl);
    }
    return flipTexCoords;
} // end of renderWithMaterial()
```

`renderWithMaterial()` checks the new material name (stored in `faceMat`) with the name of the currently loaded material (in `renderMatName`) and makes no changes if the names are the same.

The method returns a boolean indicating whether the texture's coordinates need to be vertically flipped when they're mapped to the shape's coordinates. The current value is stored in a global so it can be returned by future calls to `renderWithMaterial()` when the material name hasn't changed.

The method doesn't allow color and texturing to be mixed (i.e., blended). Any face color is ignored when a texture is applied.

`switchOffTex()` switches off 2D texturing (and enables lighting lighting). `switchOnTex()` turns texturing on (and disables lighting):

```
// global
private boolean usingTexture = false;

public void switchOffTex(GL gl)
{
    if (usingTexture) {
        gl.glDisable(GL.GL_TEXTURE_2D);
        usingTexture = false;
        gl.glEnable(GL.GL_LIGHTING);
    }
}
```

```

    }
} // end of switchOffTex()

private void switchOnTex(Texture tex, GL gl)
{
    gl.glDisable(GL.GL_LIGHTING);
    gl.glEnable(GL.GL_TEXTURE_2D);
    usingTexture = true;
    tex.bind();
} // end of switchOnTex()

```

getTexture() iterates through the materials ArrayList until it finds the named material and retrieves its texture:

```

// global
private ArrayList<Material> materials;
// stores the Material objects built from the MTL file data

```

```

private Texture getTexture(String matName)
{
    Material m;
    for (int i = 0; i < materials.size(); i++) {
        m = (Material) materials.get(i);
        if (m.hasName(matName))
            return m.getTexture();
    }
    return null;
} // end of getTexture()

```

setMaterialColors() performs a similar iteration through materials, but gets the Material object to turn on its own colors:

```

private void setMaterialColors(String matName, GL gl)
{
    Material m;
    for (int i = 0; i < materials.size(); i++) {
        m = (Material) materials.get(i);
        if (m.hasName(matName))
            m.setMaterialColors(gl);
    }
} // end of setMaterialColors()

```

Material.setMaterialColors() consists of several calls to GL.glMaterialfv() to switch on the ambient, diffuse, and specular colors for the material and its shininess:

```

// in the Material class
// global color info
private Tuple3 ka, kd, ks; // ambient, diffuse, specular colors
private float ns, d; // shininess and alpha

public void setMaterialColors(GL gl)
{
    if (ka != null) { // ambient color
        float[] colorKa = { ka.getX(), ka.getY(), ka.getZ(), 1.0f };
        gl.glMaterialfv(GL.GL_FRONT_AND_BACK, GL.GL_AMBIENT, colorKa, 0);
    }
    if (kd != null) { // diffuse color

```



```

    float[] colorKd = { kd.getX(), kd.getY(), kd.getZ(), 1.0f };
    gl.glMaterialfv(GL.GL_FRONT_AND_BACK, GL.GL_DIFFUSE, colorKd,0);
}
if (ks != null) { // specular color
    float[] colorKs = { ks.getX(), ks.getY(), ks.getZ(), 1.0f };
    gl.glMaterialfv(GL.GL_FRONT_AND_BACK, GL.GL_SPECULAR, colorKs,0);
}

if (ns != 0.0f) // shininess
    gl.glMaterialf(GL.GL_FRONT_AND_BACK, GL.GL_SHININESS, ns);

if (d != 1.0f) { // alpha
    // not implemented
}
} // end of setMaterialColors()

```

I haven't implemented transparency, although the Material object stores an alpha value (in the *d* variable). It would require the use of blending and depth testing and the inclusion of the *d* value in the three calls to `GL.glMaterialfv()`.

Rendering a Face

The code for rendering a face is complicated by the use of indices in the OBJ data. Each face is defined by a sequence of terms, with each term consisting of *indices* pointing to the actual vertex, texture coordinate, and normal data, as shown in the following example:

```

f 104/22/188 114/45/198 78/78/138
f 81/56/144 104/87/188 78/21/138
:

```

The numbers are indices for the vertices, texture coordinates, and normals data.

`Faces.renderFace()`'s task is to draw the *i*th face of the model. The *i* value is used to access the *i*th arrays in `facesVertIdxs`, `facesTexIdxs`, and `facesNormIdxs`:

```

private ArrayList<int[]> facesVertIdxs;
private ArrayList<int[]> facesTexIdxs;
private ArrayList<int[]> facesNormIdxs;

```

The array retrieved from `facesVertIdxs` contains vertex *indices* for the *i*th face. The array extracted from `facesTexIdxs` holds texture coordinate *indices*, and the array from `facesNormIdxs` has normal *indices*.

The actual data is stored in the `verts`, `normals`, or `texCoords` `ArrayLists`:

```

private ArrayList<Tuple3> verts;
private ArrayList<Tuple3> normals;
private ArrayList<Tuple3> texCoords;

```

When an index (e.g., index value *j*) is read from one of the indices arrays, such as `facesVertIdxs`, `renderFace()` uses it to access the *j*-1th tuple in `verts`. This tuple contains the model's vertex for index *j*.

I use *j*-1 since the OBJ format starts its indices at 1, while the tuples in the `verts`, `normals`, and `texCoords` `ArrayLists` start at position 0.

The following is `Faces.renderFace()`:

```

// global
private static final float DUMMY_Z_TC = -5.0f;

public void renderFace(int i, boolean flipTexCoords, GL gl)

```

```

{
    if (i >= facesVertIdxs.size()) // i out of bounds?
        return;

    int[] vertIdxs = (int[]) (facesVertIdxs.get(i));
        // get the vertex indices for face i

    int polytype; // the shape of the faces
    if (vertIdxs.length == 3)
        polytype = gl.GL_TRIANGLES;
    else if (vertIdxs.length == 4)
        polytype = gl.GL_QUADS;
    else
        polytype = gl.GL_POLYGON;

    gl.glBegin(polytype);

    // get the normal and tex coords indices for face i
    int[] normIdxs = (int[]) (facesNormIdxs.get(i));
    int[] texIdxs = (int[]) (facesTexIdxs.get(i));

    /* render the normals, tex coords, and vertices for face i
       by accessing them using their indices */
    Tuple3 vert, norm, texCoord;
    float yTC;
    for (int f = 0; f < vertIdxs.length; f++) {
        if (normIdxs[f] != 0) { // if there are normals, render them
            norm = (Tuple3) normals.get(normIdxs[f] - 1);
            gl.glNormal3f(norm.getX(), norm.getY(), norm.getZ());
        }

        if (texIdxs[f] != 0) { // if there are tex coords, render them
            texCoord = (Tuple3) texCoords.get(texIdxs[f] - 1);
            yTC = texCoord.getY();
            if (flipTexCoords) // flip tuple's y-value (the texture's t-value)
                yTC = 1.0f - yTC;

            if (texCoord.getZ() == DUMMY_Z_TC) // using 2D tex coords
                gl.glTexCoord2f(texCoord.getX(), yTC);
            else // 3D tex coords
                gl.glTexCoord3f(texCoord.getX(), yTC, texCoord.getZ());
        }

        vert = (Tuple3) verts.get(vertIdxs[f] - 1);
            // render the vertices
        gl.glVertex3f(vert.getX(), vert.getY(), vert.getZ());
    }

    gl.glEnd();
} // end of renderFace()

```

The vertex, texture coordinates, and normals data is rendered using the GL methods: `glVertex3f()`, `glTexCoord2f()`, and `glNormal3f()`.

If 3D texture coordinates are detected, `glTexCoord3f()` is called, but only the 2D part will be drawn due to the use of 2D texture rendering in `switchOnTex()`.

`renderFace()` is passed a boolean called `flipTexCoords` that indicates whether the texture's coordinates need to be vertically flipped when they're mapped to the shape's coordinates. This is done by subtracting the texture's `t`-value from 1.0.

OBJ face data may leave out texture coordinate and normal indices. For example, a face without texture coordinates will have the following form:

```
f 104//188 114//198 78//138
f 81//144 104//188 78//138
:
```

If faces don't use normals or texture coordinates, the indices arrays will contain 0s. This is tested for in `renderFace()`, and the calls to `glTexCoord2f()` and `glNormal3f()` are skipped.

Drawing a Model

The lengthy code needed to create a display list has its payoff in the brevity and speed of the drawing operation, `OBJModel.draw()`:

```
// in the OBJModel class
private int modelDisplst; // the model's display list

public void draw(GL gl)
{ gl.glCallList(modelDisplst); }
```

`draw()` is the only public method in `OBJModel`, aside from its constructors.

When to Create the Display List

The display list technique employed in `draw()` has a drawback: the display list (in `modelDisplst`) must already exist. The list is created in `drawToList()` (described previously in the “Creating a Display List for the Model” section), which is called at the end of the loading phase.

This approach can be criticized since display list creation is arguably not part of loading. In practical terms, it means that the loader can only start once it has a valid reference to the OpenGL state. This makes it impossible for the loader to be used for offline batch processing tasks where there is no rendering phase and no OpenGL state.

`OBJLoader` could be rewritten to not finish its loading phase with a call to `drawToList()`. The call could be moved to the start of `draw()` instead, with the addition of testing code so that `drawToList()` was only called once, when `draw()` was executed for the first time. This would free the loading phase from its dependency on the OpenGL state, but with the small penalty of having `draw()` take a little longer to render the model initially.

Viewing a Model

Before moving on to `TourModelGL`, I'll demonstrate the `OBJLoader` package by using it inside a simple model display application, `ModelLoaderGL` (shown in action in Figure 17-1).

`ModelLoaderGL` utilizes the callback coding approach, described in Chapter 15 and illustrated by Figure 17-5.

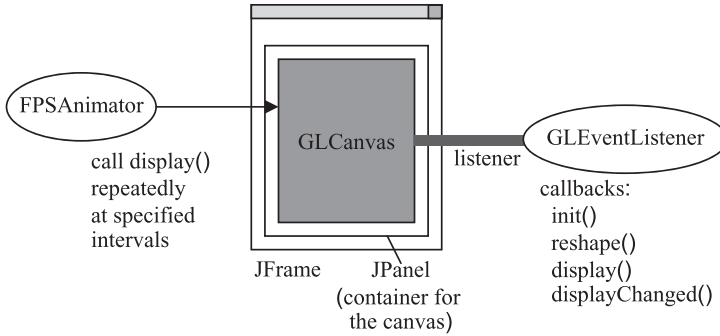


Figure 17-5. *The callback coding framework*

The ModelLoaderGL JFrame contains a JPanel that holds a GLCanvas. The GLCanvas displays the OBJ model, which may be rotating. The model is scaled and centered at the origin.

The canvas’s listener is ModelLoaderGLListener (a subclass of GLEventListener), and the canvas’s display is updated by an FPSAnimator instance using fixed-rate scheduling.

The simplicity of the application is reflected in the class diagrams for ModelLoaderGL in Figure 17-6 (only the public methods are listed).

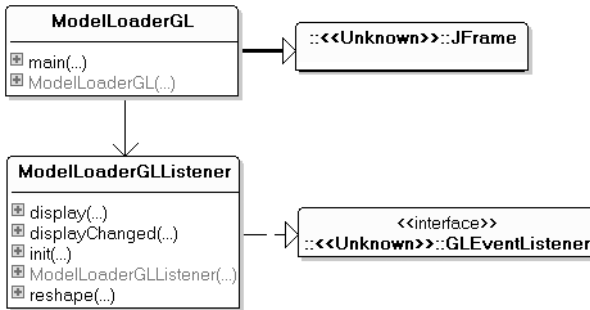


Figure 17-6. *Class diagrams for ModelLoaderGL*

Loading the Model

The name of the model is supplied on the command line and passed to the ModelLoaderGLListener constructor where it’s stored in the global string modelName.

When init() is called, the model is loaded using its name:

```
// in the ModelLoaderGLListener class
// globals
private String modelName;
private OBJModel model;

public void init(GLAutoDrawable drawable)
{
    GL gl = drawable.getGL();
    // other non-relevant lines...
```

```

    model = new OBJModel(modelName, maxSize, gl, true);
} // end of init()

```

The `maxSize` value in the `OBJModel` constructor specifies the maximum size of the model's largest dimension. The `true` argument switches on verbose reporting of the model's details, which includes the number of vertices, normals, and texture coordinates found and its dimensions and colors used. They're printed to standard output.

Drawing the Model

`OBJModel.draw()` is called in the `display()` callback method:

```

public void display(GLAutoDrawable drawable)
{
    GL gl = drawable.getGL();
    // other nonrelevant lines...

    model.draw(gl);
    gl.glFlush();
} // end of display()

```

Other JOGL Model Loaders

Pournaras's *JautOGL* is a 3D multiplayer racing game with many interesting features, such as use of the full-screen exclusive mode (FSEM), 3D sound through JOAL, multiple camera views, and a UDP client-server model employing nonblocking sockets (<http://today.java.net/pub/a/today/2006/10/10/development-of-3d-multiplayer-racing-game.html> and <https://jautogl.dev.java.net/>).

The loader part of the game consists of two classes, `GLModel` and `MtlLoader`. The former is responsible for parsing and displaying the OBJ file, the latter for loading the MTL file. Texturing isn't supported, and coloring is implemented using `GL.GL_COLOR_MATERIAL` and calls to `GL.glColor4f()`.

Glass's loader is part of his 3D asteroid game tutorial (<http://www.cokeandcode.com/asteroid-tutorial/>) built using LWJGL (which is quite similar to JOGL). He also develops a game framework, utilities for drawing the GUI (e.g., menus), a texture loader, classes for 3D sprites, a particle system, and sound based around LWJGL's binding of OpenAL and JOrbis for decoding OGG files.

His loader handles `v`, `vt`, `vn`, and `f` OBJ statements, but there's no MTL capability. Instead, a texture is loaded separately and wrapped around the entire model.

An OBJ loader is under development by Chris Brown at <https://jglmark.dev.java.net/>. As of March 2007, it didn't handle materials or textures.

A 3DS loader can be found at <http://joglutils.dev.java.net/>. The ThreeDS package by Greg Rodgers supports colors and textures, but 3DS features such as keyframe animation aren't in place yet. It is part of the full `joglutils` JAR, which is downloadable from the Documents & Files folder at the web site.

The NeHe site (<http://nehe.gamedev.net/>) is an excellent resource for OpenGL tutorials. Lesson 31 by Brett Porter explains how to build a MilkShape3D model loader. Color and texturing is available but not animation. The JOGL port by Nikolaj Ougaard can be found at http://pepijn.fab4.be/?page_id=34. Interestingly, it includes code for keyframe positioning of joints, but it was incomplete as of March 2007.

The need for model loaders in JOGL will undoubtedly drive development forward at a rapid pace, so it's a good idea to regularly search the JOGL forum at <http://www.javagaming.org/forums/index.php?board=25.0> for announcements about new and improved packages.

The TourModelsGL Application

Having developed the OBJLoader package and tested it with ModelLoaderGL, it's time to consider TourModelsGL. It reuses a lot of code from the TourGL example in Chapter 16. It implements the active rendering framework, and the 3D scene reuses TourGL's green and blue checkerboard floor with numbers along its x- and z- axes.

The class diagrams for TourModelsGL are shown in Figure 17-7; only public methods are shown.

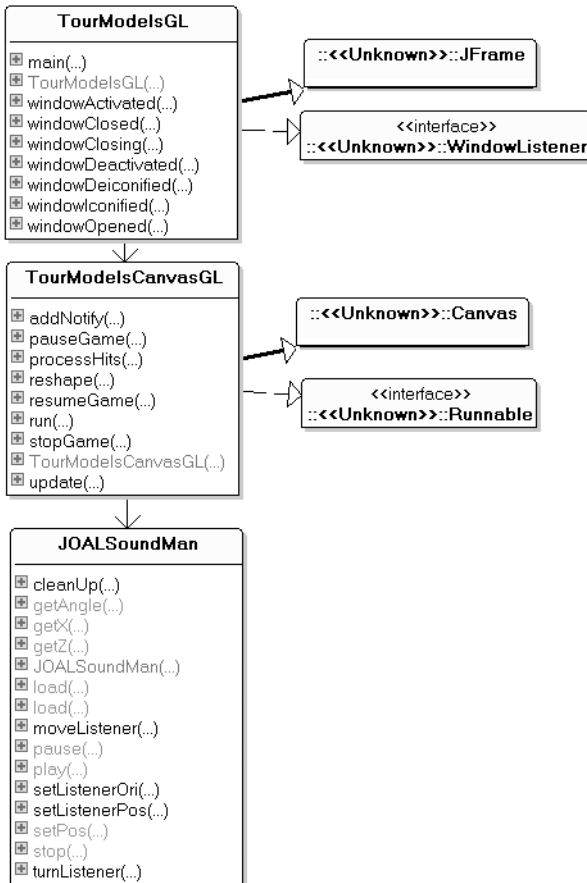


Figure 17-7. Class diagrams for TourModelsGL

TourModelsGL creates the JFrame and JPanel around the heavy-weight TourModelsCanvasGL canvas and handles window events such as resizing and iconification.

TourModelsCanvasGL spawns a single thread that initializes rendering, then loops, carrying out an update/render/sleep cycle with a fixed period.

Aside from the checkerboard, TourModelsCanvasGL makes use of TourCanvasGL's user navigation code, which converts key presses into camera movement forward, backward, left, and right. One change is that the user can't move vertically. This simplifies the 3D audio requirements for the game and lets me reuse JOALSoundMan from Chapter 13 (it assumes that a listener stays on the XZ plane).

TourModelsCanvasGL doesn't display a skybox, billboard trees, a rotating sphere, or the splash screen and "Game Over" message. Instead, the scene contains four OBJ models, shown in Figure 17-8 (and also in Figure 17-2).



Figure 17-8. *The models in TourModelsGL*

The other new elements in TourModelsCanvasGL are the following:

- The ability to select (pick) the penguin or couch with the mouse. OpenGL's selection mode is utilized to implement this feature.
- The penguin singing with the help of JOALSoundMan.
- The spooky fog (which is switched off in Figures 17-2 and 17-8 but can be seen in Figure 17-3).

Each of these is explained in detail in the rest of this chapter.

Adding Models

The four models (penguin, rose and vase, racing car, and couch) were chosen to illustrate the features (and limitations) of the OBJLoader package.

The penguin is a mesh wrapped with a single texture. The use of texturing means that the color lighting values defined in the penguin's MTL file are ignored.

The couch employs a single diffuse color, but the model's normals allow it to be affected by the scene's light source.

The rose and vase model has several colors using various ambient, diffuse, and specular settings.

I borrowed the racing car model from Pournaras's *JautOGL* game and modified its MTL file so the car uses different colors and textures on different faces.

Loading the Models

The models are loaded during the initialization phase in `TourModelsCanvasGL.initRender()`:

```
// globals for the four OBJ models
private OBJModel couchModel, carModel, penguinModel, roseVaseModel;

// loading done in initRender()
couchModel = new OBJModel("couch", 2.0f, gl, false);
carModel = new OBJModel("formula", 4.0f, gl, false);
penguinModel = new OBJModel("penguin", gl);
roseVaseModel = new OBJModel("rose+vase", 3.2f, gl, false);
```

`OBJModel` attempts to load an OBJ file with the specified name. The four-argument version of the constructor includes a maximum size, a reference to the GL state, and a boolean that determines whether verbose model details are printed to standard output.

It's important to set the model's size using a constructor argument rather than a later call to `GL.glScalef()`, since a scaling transformation will affect the model's normals and so will modify the model's coloring and/or texturing.

The two-argument version of the `OBJModel` constructor assumes that the maximum size of the model will be 1.0f and that its details shouldn't be output.

The previous code fragments show a potential drawback of this loader: the need for a reference to the OpenGL state (the `gl` argument) in the call to the `OBJModel` constructor. As explained previously in the "When to Create the Display List" section, the reference is required so a display list can be created for the model at the end of the loading phase.

Drawing the Models

`renderScene()` calls `drawModels()` to render the models. Each model is drawn after being translated and rotated:

```
private void drawModels()
{
    drawCouch();

    // the racing car
    gl.glPushMatrix();
    gl.glTranslatef(-3.0f, 0.5f, -3.0f); // move left, up, back
    carModel.draw(gl);
    gl.glPopMatrix();

    drawPenguin();

    // the rose vase
    gl.glPushMatrix();
    gl.glTranslatef(0f, 1.6f, 0f); // move up
    roseVaseModel.draw(gl);
}
```



```

    gl.glPopMatrix();
} // end of drawModels()

private void drawCouch()
{
    gl.glPushMatrix();
    gl.glTranslatef(4.0f, 0.5f, -4.0f); // move right, up, back
    gl.glRotatef(-90.0f, 1.0f, 0.0f, 0.0f);
                                   // rotate clockwise around x-axis
    couchModel.draw(gl);
    gl.glPopMatrix();
} // end of drawCouch()

private void drawPenguin()
{
    gl.glPushMatrix();
    gl.glTranslatef(2.0f, 0.5f, 0f); // move right, up
    gl.glRotatef(-90.0f, 0.0f, 1.0f, 0.0f); // rotate clockwise
    penguinModel.draw(gl);
    gl.glPopMatrix();
} // end of drawPenguin()

```

The couch and penguin are drawn by separate methods so these functions can be reused by the picking code described in the next section.

The calls to `GL.glPushMatrix()` and `GL.glPopMatrix()` stop the translation and rotation operations from affecting other elements in the scene. If a model isn't moved from its default position at the origin, stack pushing and popping isn't needed.

The rotation of a model around the x-axis (e.g., for the couch) is a fairly common requirement since many drawing packages use the XY plane as a “floor” rather than XZ.

Let's Be Picky

OpenGL supports a selection (or *picking*) mode that makes it fairly straightforward to click an object inside a scene with the mouse and retrieve details about it, such as its ID and distance from the camera.

Picking is enabled for the penguin and the couch in `TourModelsCanvasGL`. For example, I can click the penguin's eye when the camera is orientated as in Figure 17-9.

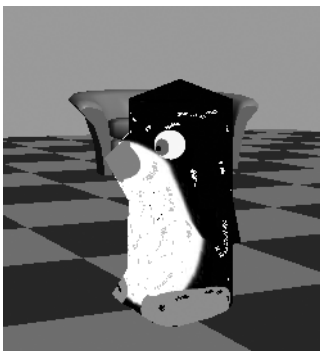


Figure 17-9. *The penguin in front of the couch*

The application then prints the following:

```
No. of hits: 2
Hit: 1
  minZ: 0.7478; maxZ: 0.769
  Name(s): couch
Hit: 2
  minZ: 0.3818; maxZ: 0.4625
  Name(s): penguin
Picked the penguin
```

The positioning of the penguin in front of the couch means that both models are selected when the user clicks the penguin's eye. Their depth information (stored in *minZ* and *maxZ*) allows the application to determine that the penguin is nearest to the camera, so it is chosen from the two possibilities.

If the camera is moved so the models don't overlap, picking will only return details for the one clicked upon.

The picking code has four main stages:

1. The cursor coordinates of a mouse press are recorded.
2. *Selection mode* is entered when it's time to render the scene, and the viewing volume is reduced to a small area around the cursor location.
3. The scene is *rendered*, which means that details about named objects inside the viewing volume are stored in *hit records* in a *selection buffer*. *Rendering* is a misleading word since nothing is drawn to the frame buffer.
4. Once the selection mode has been exited, name and depth information can be extracted from the hit records.

An object is named with an integer (not a string), which is pushed onto the *name stack* prior to the object's "rendering" in selection mode, and popped afterward. The names stored in the hit records are copied from the name stack when the viewing volume is examined in stage 3.

Capturing Mouse Presses

A mouse listener is set up in *TourModelsCanvasGL*'s constructor:

```
// in TourModelsCanvasGL()
addMouseListener( new MouseAdapter() { // used for picking
    public void mousePressed(MouseEvent e)
    { mousePress(e); }
});
```

`mousePress()` stores the cursor coordinates and switches on the `inSelectionMode` boolean:

```
// globals for picking
private boolean inSelectionMode = false;
private int xCursor, yCursor;

private void mousePress(MouseEvent e)
{
    xCursor = e.getX();
    yCursor = e.getY();
    inSelectionMode = true;
}
```

Switching to Selection Mode

In `renderScene()`, the `inSelectionMode` boolean is used to distinguish between normal rendering and selection mode:

```
// global
private GLDrawable drawable; // the rendering 'surface'

// in renderScene()
if (inSelectionMode)
    pickModels();
else { // normal rendering
    drawFloor();
    drawModels();
    drawable.swapBuffers(); // put the scene onto the canvas
    // swap front and back buffers, making the new rendering visible
}
```

All the normal scene rendering (e.g., of the floor and models) should be moved to the else part of the if-test since there's no point drawing objects unrelated to picking when selection mode is enabled.

In previous active rendering examples (e.g., `TourCanvasGL` in the previous chapter), the call to `GLDrawable.swapBuffer()` occurs after `renderScene()` has returned, back in `renderLoop()`. The call has been moved so it only occurs after the scene has really been rendered. Selection mode rendering only affects the selection buffer, so there's no need to swap the front and back buffers.

If the `swapBuffers()` call is left in `renderLoop()` in `TourModelsCanvasGL`, it triggers a nasty flicker since the back buffer is empty after picking but filled after normal rendering. This means the user will see a white screen for a moment after each selection.

Model Picking

`pickModels()` illustrates the picking code stages:

```
// global names (IDs) for pickable models
private static final int COUCH_ID = 1;
private static final int PENGUIN_ID = 2;

private void pickModels()
// draw the couch and penguin models in selection mode
{
    startPicking();

    gl.glPushName(COUCH_ID);
    drawCouch();
    gl.glPopName();

    gl.glPushName(PENGUIN_ID);
    drawPenguin();
    gl.glPopName();

    endPicking();
} // end of pickModels()
```

Picking initialization (stage 2) is carried out in `startPicking()`, then the objects are rendered (stage 3) and picking is terminated by `endPicking()` (stage 4), which also processes the hit records in the selection buffer.

The `drawCouch()` and `drawPenguin()` methods are reused without change, but their calls are bracketed by the pushing and popping of their names onto OpenGL's name stack.

A common mistake is to forget to pop a name after its object has been rendered. Also, `GL.glPushName()` and `GL.glPopName()` only work after the selection mode has been enabled (which is done in `startPicking()`).

The Start of Picking

`startPicking()` switches to the selection mode, initializes the selection buffer and name stack, and creates a reduced-size viewing volume around the cursor:

```
// globals
private static final int BUFSIZE = 512; // size of buffer
private IntBuffer selectBuffer;

private void startPicking()
{
    // initialize the selection buffer
    int selectBuf[] = new int[BUFSIZE];
    selectBuffer = BufferUtil.newIntBuffer(BUFSIZE);
    gl.glSelectBuffer(BUFSIZE, selectBuffer);

    gl.glRenderMode(GL.GL_SELECT); // switch to selection mode

    gl.glInitNames(); // make an empty name stack

    // save the original projection matrix
    gl.glMatrixMode(GL.GL_PROJECTION);
    gl.glPushMatrix();
    gl.glLoadIdentity();

    // get the current viewport
    int viewport[] = new int[4];
    gl.glGetIntegerv(GL.GL_VIEWPORT, viewport, 0);

    // create a 5x5 pixel picking volume near the cursor location
    glu.gluPickMatrix((double) xCursor,
                     (double) (viewport[3] - yCursor),
                     5.0, 5.0, viewport, 0);

    /* set projection (perspective or orthogonal) exactly as it is in
       normal rendering (i.e. duplicate the gluPerspective() call
       in resizeView()) */
    glu.gluPerspective(45.0,
                      (float)panelWidth/(float)panelHeight, 1, 100);

    gl.glMatrixMode(GL.GL_MODELVIEW); // restore model view
} // end of startPicking()
```

JOGL's `BufferUtil` utility class was utilized to create an integer buffer (`BufferUtil.newIntBuffer()`). The selection buffer in OpenGL is an array of *unsigned* integers, a slightly different thing, which impacts how depth values are extracted later.

The first two arguments of `GLU.gluPickMatrix()` are the cursor's (x, y) location, but it needs to be converted from Java coordinate's scheme (x and y starting at the top left) to OpenGL's scheme (x and y starting at the bottom left). This is done by subtracting the cursor's y-value from the viewport's height: `(viewport[3] - yCursor)`.

A common problem is forgetting to set the selection mode's projection (perspective or orthogonal) to be the same as in normal rendering. In the active rendering framework, this is done with a call to `GLU.gluPerspective()` in `resizeView()`, which is duplicated in `startPicking()`.

The End of Picking

`endPicking()` switches rendering back to normal, which has the side-effect of making the selection buffer available:

```
private void endPicking()
{
    // restore original projection matrix
    gl.glMatrixMode(GL.GL_PROJECTION);
    gl.glPopMatrix();
    gl.glMatrixMode(GL.GL_MODELVIEW);
    gl.glFlush();

    // return to normal rendering mode, and process hits
    int numHits = gl.glRenderMode(GL.GL_RENDER);
    processHits(numHits);

    inSelectionMode = false;
} // end of endPicking()
```

The buffer is examined in `processHits()`.

Processing the Hit Records

`processHits()` simply lists all the hit records in the selection buffer and reports the name of the object that was picked closest to the viewport.

Each hit record contains the following:

- The number of names assigned to the hit object (usually there's only one, but it's possible to assign more)
- The minimum and maximum depths of the hit
- The names assigned to the hit object (which come from the name stack)

One source of confusion is that the depth values are for the part of an object that intersects with the viewing volume; they do not correspond to the object's z-axis dimensions.

Also, although the OpenGL specification talks about names on the name stack and in the hit records, it's more accurate to think of them as integer name IDs:

```
public void processHits(int numHits)
{
    if (numHits == 0)
        return; // no hits to process

    System.out.println("No. of hits: " + numHits);

    // storage for the name ID closest to the viewport
    int selectedNameID = -1; // dummy initial values
    float smallestZ = -1.0f;
```

```

boolean isFirstLoop = true;
int offset = 0;

/* iterate through the hit records, saving the smallest z value
   and the name ID associated with it */
for (int i=0; i < numHits; i++) {
    System.out.println("Hit: " + (i + 1));

    int numNames = selectBuffer.get(offset);
    offset++;

    // minZ and maxZ are taken from the Z buffer
    float minZ = getDepth(offset);
    offset++;

    // store the smallest z value
    if (isFirstLoop) {
        smallestZ = minZ;
        isFirstLoop = false;
    }
    else {
        if (minZ < smallestZ)
            smallestZ = minZ;
    }

    float maxZ = getDepth(offset);
    offset++;

    System.out.println(" minZ: " + df4.format(minZ) +
        " ; maxZ: " + df4.format(maxZ));

    // print name IDs stored on the name stack
    System.out.print(" Name(s): ");
    int nameID;
    for (int j=0; j < numNames; j++){
        nameID = selectBuffer.get(offset);
        System.out.print( idToString(nameID) );
        if (j == (numNames-1)) {
            // if the last one (the top element on the stack)
            if (smallestZ == minZ) // is this the smallest min z?
                selectedNameID = nameID; // then store it's name ID
        }
        System.out.print(" ");
        offset++;
    }
    System.out.println();
}

System.out.println("Picked the " + idToString(selectedNameID));
System.out.println("-----");
} // end of processHits()

```

Typical output from `processHits()` was shown earlier. Here's another example, when only the couch is picked:

```
No. of hits: 1
Hit: 1
  minZ: 0.6352; maxZ: 0.6669
  Name(s): couch
Picked the couch
```

A depth is in the range 0 to 1 but is stored after being multiplied by $2^{32} - 1$ and rounded to the nearest unsigned integer. The number will be negative due to the multiplication and being cast to a signed integer in the buffer.

The conversion of the integer back to a float is done by `getDepth()`:

```
private float getDepth(int offset)
{
    long depth = (long) selectBuffer.get(offset); // large -ve number
    return (1.0f + ((float) depth / 0x7fffffff));
           // return as a float between 0 and 1
} // end of getDepth()
```

The depths aren't linearly proportional to the distance to the viewpoint due to the nonlinear nature of the Z buffer, but different depths can be compared to find the one closest to the camera.

The mapping from a name ID to a string is carried out by `idToString()`:

```
private String idToString(int nameID)
{
    if (nameID == COUCH_ID)
        return "couch";
    else if (nameID == PENGUIN_ID)
        return "penguin";

    // we should not reach this point
    return "nameID " + nameID;
} // end of idToString()
```

Gleem: A Different Way of Picking

The picking described in this chapter relies on OpenGL's selection mode, which is simple to use but has a reputation for being slow. A more advanced solution is to utilize ray-to-triangle intersection tests, as found in `gleem` (OpenGL Extremely Easy-to-Use Manipulators), a library used in many of the JOGL demos (<http://jogl-demos.dev.java.net/>).

`Gleem` supports several forms of object selection and dragging based on the manipulators idea first introduced in Silicon Graphics's Open Inventor 3D graphics API (<http://oss.sgi.com/projects/inventor/>). `Gleem` includes manipulators for translating a selected object along a line and across a plane and for rotating it about various axes and for scaling.

The manipulator functionality employs ray casting to find intersections with the triangles in the scene's objects. A *ray* is the path followed by a light beam from the camera to the object.

`Gleem` includes useful camera navigation controls in its `ExaminerViewer` class, such as trackball-style rotation, translation, and zooming.

`Gleem` was developed by Ken Russell, who is also one of the main developers of JOGL. The source is included with the JOGL demos source code, downloadable from <http://jogl-demos.dev.java.net>. Some older background information can be found at <http://www.media.mit.edu/~kbrussel/gleem/>.

A Singing Penguin

JOALSoundMan (developed in Chapter 13) is employed to set up a 3D sound for the penguin model and to attach an audio listener to the camera.

A JOALSoundMan instance is created in TourModelsCanvasGL's constructor:

```
// global
private JOALSoundMan soundMan;

// in TourModelsCanvasGL()
soundMan = new JOALSoundMan();
```

Locating the Penguin Sound

The penguin sound is positioned at (2, 0, 0) in initRender(), and set to play repeatedly:

```
// in initRender()
if (!soundMan.load("penguin", 2, 0, 0, true))
    System.out.println("Penguin sound not found");
else
    soundMan.play("penguin");
```

Although the penguin model is also loaded in initRender(), it isn't positioned until drawPenguin() is called at rendering time:

```
private void drawPenguin()
{
    gl.glPushMatrix();
    gl.glTranslatef(2.0f, 0.5f, 0f); // move up, right, to (2,0.5,0)
    gl.glRotatef(-90.0f, 0.0f, 1.0f, 0.0f);
        // rotate the model to face left
    penguinModel.draw(gl);
    gl.glPopMatrix();
} // end of drawPenguin()
```

There's no direct link between the audio source and the penguin model, so it's up to the programmer to ensure they stay colocated. That's easy here since the penguin doesn't move.

Connecting the Camera and the Listener

As the camera moves and rotates about the scene, so should the listener. The connection is made by updating the listener's position and y-axis orientation to match those of the camera.

Obtaining the positional data is straightforward since the camera details are stored in three globals, xPlayer, yPlayer, and zPlayer, updated by processKey(). The listener moves by using xPlayer and zPlayer (yPlayer isn't utilized since JOALSoundMan assumes the listener always stays on the floor).

Linking the rotation of the camera to the listener is a bit trickier. The camera's rotation angle is stored in the viewAngle global, which initially has the value -90 degrees to point it along the negative z-axis. When the camera rotates clockwise around the y-axis, a positive amount is added to viewAngle (see Figure 17-10). However, JOAL initializes its listener to point down the negative z-axis, so it starts at 0 degrees. Also, a clockwise rotation reduces the angle rather than increases it (as shown in Figure 17-10).

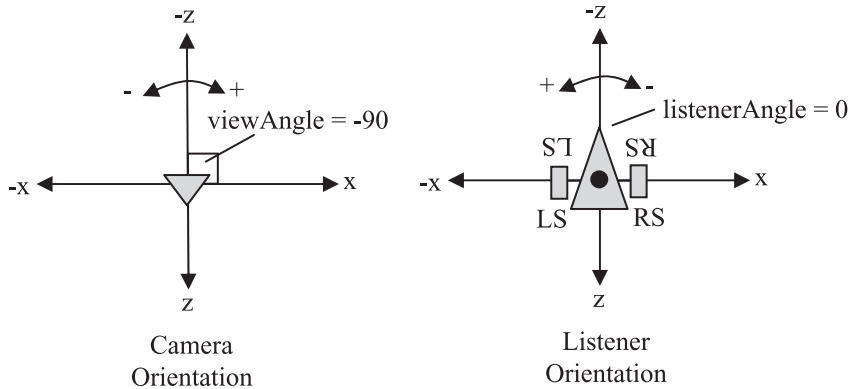


Figure 17-10. Rotating the camera and listener

TourModelsCanvasGL includes a new global, `listenerAngle`, which stores the current rotation angle of the listener around the y-axis. It starts with the value 0, which corresponds to it pointing down the negative z-axis. Both `viewAngle` and `listenerAngle` are initialized in `initViewerPosn()`:

```
// globals
private double viewAngle, listenerAngle;

// in initViewerPosn()
viewAngle = -90.0; // along negative z-axis
listenerAngle = 0;
```

When `processKey()` adjusts the camera's rotation value (in `viewAngle`) it also changes the listener's rotation (in `listenerAngle`) but with the opposite operation (e.g., addition instead of subtraction). For instance, the following code fragment deals with the camera turning left:

```
// globals
private final static double ANGLE_INCR = 5.0; // degrees

// turning left in processKey()
viewAngle -= ANGLE_INCR; // subtract
listenerAngle += ANGLE_INCR; // add
```

The positional and rotational data are employed in `renderScene()` to move the listener:

```
// in renderScene()
soundMan.setListenerPos( (float)xPlayer, (float)zPlayer );
soundMan.setListenerOri( (int) listenerAngle );
```

The casting of `listenerAngle` to an integer is a requirement of the `JOALSoundMan.setListenerOri()` method, and perhaps the code should be rewritten to accept doubles (or floats).

The Fog Descends

The fog shown in Figure 17-3 makes it much harder to find the models, which could be used as the basis of a time-constrained search game. Also, the fog reduces the amount of geometry that needs to be rendered, thereby improving the application's speed.

Almost all the fog-related code is in one method, `addFog()`, which is called from `initRender()`:

```
private void addFog()
{
    gl.glEnable(GL.GL_FOG);

    gl.glFogi(GL.GL_FOG_MODE, GL.GL_EXP2);
    // possible modes are: GL.GL_LINEAR, GL.GL_EXP, GL.GL_EXP2

    float[] fogColor = {0.7f, 0.6f, 0.6f, 1.0f};
    // same color as background
    gl.glFogfv(GL.GL_FOG_COLOR, fogColor, 0);

    gl.glFogf(GL.GL_FOG_DENSITY, 0.35f);

    gl.glFogf(GL.GL_FOG_START, 1.0f); // start depth
    gl.glFogf(GL.GL_FOG_END, 5.0f); // end depth

    gl.glHint(GL.GL_FOG_HINT, GL.GL_DONT_CARE);
    /* possible hints are: GL.GL_DONT_CARE, GL.GL_NICEST,
       GL.GL_FASTEST */
} // end of addFog()
```

The fog is enabled and its various characteristics are set. OpenGL implements fog by blending each pixel with the fog's color, depending on the distance from the camera, the fog density, and the fog mode.

Possible fog modes are `GL.GL_LINEAR`, `GL.GL_EXP`, and `GL.GL_EXP2`, with `GL.GL_EXP2` looking the most realistic but also being the most computationally expensive. If the linear blend is chosen, start and end depths for the fog must be defined using the `GL_FOG_START` and `GL.GL_FOG_END` attributes. If `GL.GL_EXP` or `GL.GL_EXP2` is employed, the `GL_FOG_DENSITY` attribute needs to be set.

I've used the `GL.GL_EXP2` mode in `addFog()`, so the `GL_FOG_START` and `GL.GL_FOG_END` values aren't really needed; I've included them to show how they're used.

The fog color is set with the `GL.GL_FOG_COLOR` argument, and the scene generally looks better if its background is the same color as well. In `initRender()`, I set the background to be the following:

```
gl.glClearColor(0.7f, 0.6f, 0.6f, 1.0f); // same color as the fog
```

In the clear sky screenshots (Figures 17-2, 17-8, and 17-9), the blue background was generated with the following:

```
gl.glClearColor(0.17f, 0.65f, 0.92f, 1.0f); // sky blue
```

The `GL.GL_FOG_HINT` argument *may* be utilized by OpenGL to switch to faster or higher-quality blending; its default value is `GL.GL_DONT_CARE`.

Fog can be switched off with `GL.glDisable()`, so it's possible to have the fog only selectively affect objects in the scene.

Summary

This chapter was primarily about the development of a loader for Wavefront OBJ models, resulting in the OBJLoader package that can load polygonal shapes utilizing multiple colors and textures defined using the Wavefront MTL format.

The main example, TourModelsGL, is the beginning of a search-type game, which utilizes OBJLoader to load and position models. The needs of the application also led to the discussion of three other programming techniques: picking with OpenGL's selection mode, 3D sound, and fog.

The 3D sound (a chirping penguin) employs JOAL via the JOALSoundMan class introduced in Chapter 13.

