4. Building 3D Models

In this part, I'll look at three ways to build models in Java 3D: by using built-in shapes such as Cone and Sphere, through coding with one of the Geometry subclasses, and by utilizing Wavefront OBJ models.

1. Using Java 3D Primitives

The four subclasses of Java 3D's Primitive class – Box, Cone, Cylinder, and Sphere – were used extensively in the previous two chapters. Techniques for combining these shapes to make more complicated models were demonstrated by Rocket.java and JointedArm.java in Part 3.

One shape that hasn't receive much exposure is Box, so I'll utilize it to create a die in Dice.java. In fact, the program contains four different ways to build the model; the first two are explained in this section, and the other two in section 2.

main() calls the functions using the different approaches:

```
  public static void main(String[] args)
  {
    Simple3D j3d = new Simple3D("A Dice", false, true);

    Box dice = Simple3D.texBox(1, 1, 1, "images/diceFaces.png"); // 1
    // Box dice = diceCubeMap();   // 2
/*
    Geometry geom = diceGeometryGI();   // 3
    // Geometry geom = diceGeometryQuad();   // 4

    Appearance app = Simple3D.texture("images/diceFaces.png", false);
                // texture() does not need to generate tex coords
    Shape3D dice = new Shape3D(geom, app);
*/
    j3d.rootBG.addChild( Simple3D.createBG(dice));
  }  // end of main()
```

Simple.texBox() creates a 2 x 2 x 2 cube centered on the origin, textured with "diceFace.png" shown in Figure 1.

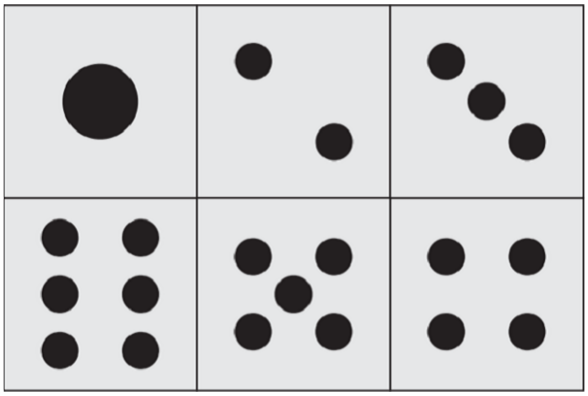

Figure 1. The diceFace.png Texture.

The rendering of the box in Figure 2 shows a limitation of using the default texturing: it duplicates the image on each face.

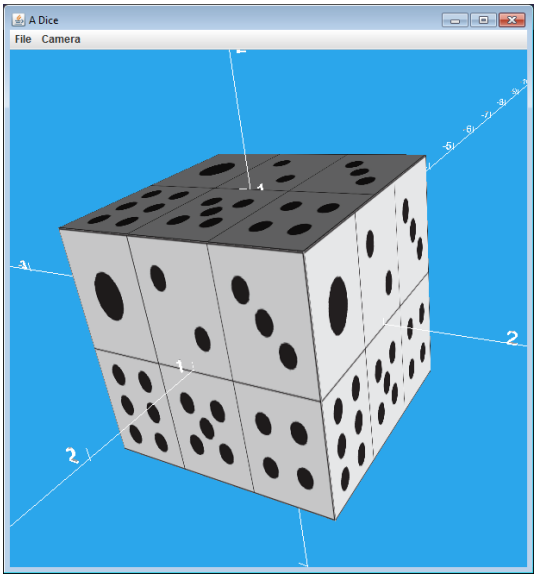

Figure 2. A Die with Poor Texturing.

This can be remedied by utilizing Box.getShape() to retrieve the Shape3D for a particular face, and then applying a texture solely to that shape (incidentally, this approach is also possible with Cone and Cylinder). diceCubeMap() implements this technique:

```
private static Box diceCubeMap()
{
  // build the textured box
  Box dice = new Box(0.5f, 0.5f, 0.5f,
                     Box.GENERATE_TEXTURE_COORDS,
                     Simple3D.color("green"));
```

```
    addPic(dice, Box.RIGHT, "images/pic1.jpg");
    addPic(dice, Box.LEFT, "images/pic6.jpg");

    // addPic(dice, Box.TOP, "images/pic2.jpg");
              // show color if no texture
    addPic(dice, Box.BOTTOM, "images/pic5.jpg");

    addPic(dice, Box.FRONT, "images/pic3.jpg");
    addPic(dice, Box.BACK, "images/pic4.jpg");

    return dice;
  } // end of diceCubeMap()


  private static void addPic(Box b, int partId, String texFnm)
  {
    Shape3D face = b.getShape(partId);
    face.setAppearance( Simple3D.texture(texFnm, false) );
     // "false" means that texture() will not generate texture coords
  }
```

The six Box faces are labeled BACK, BOTTOM, FRONT, LEFT, RIGHT, and TOP, and addPic() employs Simple3D.texture() to cover each with a different image. pic1.jpg through pic6.jpg, are pictures of the numbers 1 through 6, as shown in Figure 3.

Figure 3. pic1.jpg to pic6.jpg

When main() calls diceCubeMap(), the die is drawn as in Figure 4.

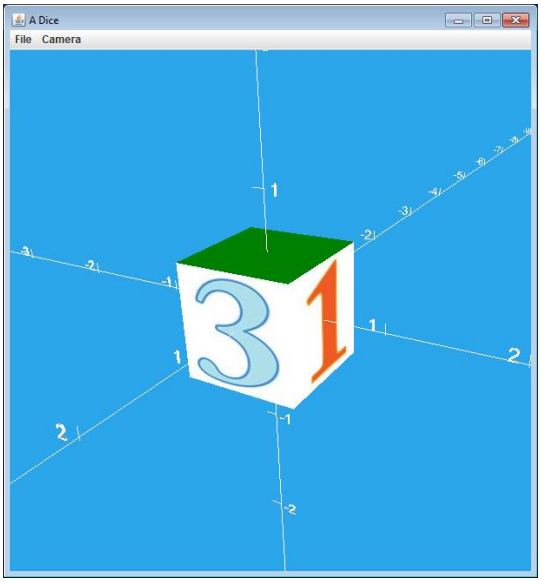

Figure 4. A Die with Different Textured Faces.

As expected, the "3" (pic3.jpg) and "1" (pic1.jpg) are drawn on the front and right of the cube. The texturing of the top face is commented out in the code which explains why that face is rendered in green, the color supplied in the Box() constructor.

The Box must have its Box.GENERATE_TEXTURE_COORDS flag set, which also means that Simple3D.texture() should be called with its needsTexCoords argument set to false.

The drawback of this approach is the need for six images. It would be preferable if diceFace.png could be utilized, but that requires texture coordinates, which are best applied in conjunction with a Geometry subclass, as detailed in section 2.

1.1. Simple3D.tetra()

A useful built-in shape missing from Java 3D's Primitive subclasses is a tetrahedron, probably because it can be coded quite simply using just four coordinates. Nevertheless, that can be problematic for beginners, so a version is included in Simple3D.

UseTetra.java creates two tetrahedrons:

```
  public static void main(String[] args)
  {
    Simple3D j3d = new Simple3D("Use Tetra", true, true);

    Shape3D tet1 = Simple3D.tetra(1,  1.5,  2, "blue");
                              // width, height, length

    Shape3D tet2 = Simple3D.texTetra(1, 4, 1, "images/marble.jpg");
    TransformGroup pos2 = Simple3D.setTranslation(3, 0, 0);
    pos2.addChild(tet2);
```

```
    BranchGroup sceneBG = new BranchGroup();
    sceneBG.addChild(tet1);
    sceneBG.addChild(pos2);

    j3d.rootBG.addChild(sceneBG);
  }  // end of main()
```

The resulting scene is shown in Figure 5.

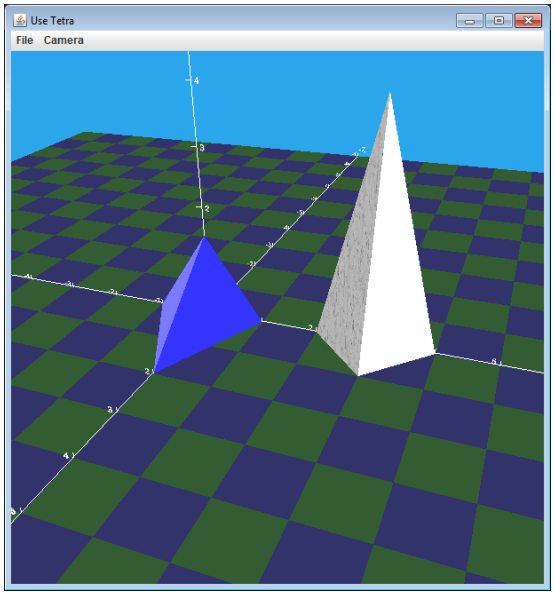

Figure 5. Two Tetrahedron.

Simple3D.tetra() (and texTetra()) generates a tetrahedron with a vertical back face and y-axis symmetry. This allows the shape to be defined using three dimensions (width, height, and length) rather than four coordinates, as illustrated by Figure 6.

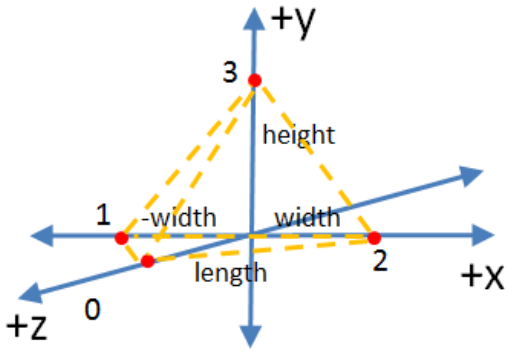

Figure 6. A tetra() Shape.

Simple3D.tetra() is implemented using Java 3D's GeometryInfo which is the subject of the next section.

2. Using Java 3D's Geometry Subclasses

Java 3D has a bewildering number of Geometry classes, as the class diagram in Figure 7 shows.

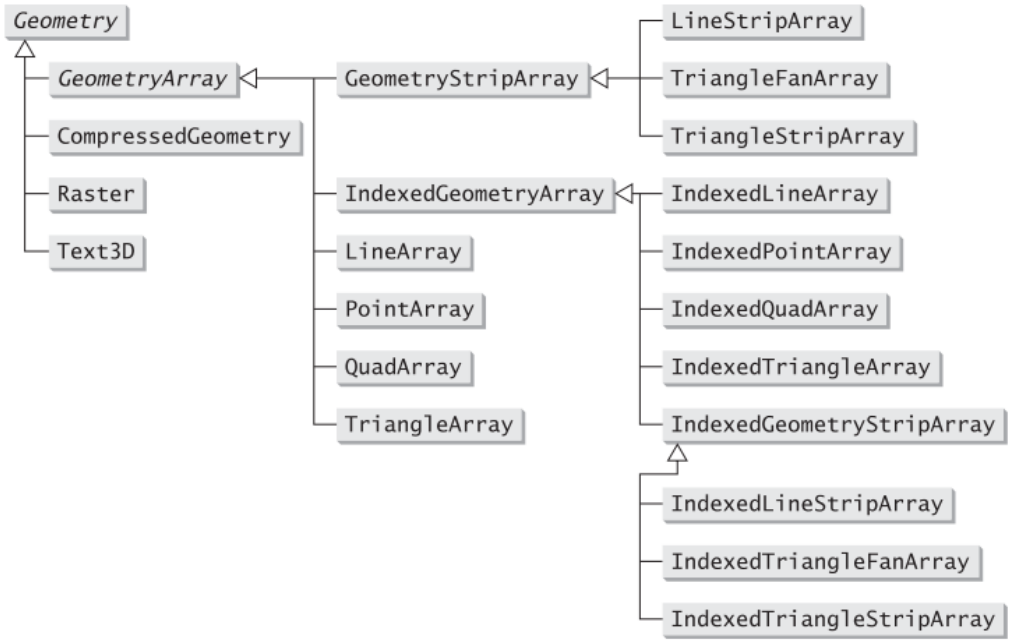

Figure 7. Class Hierarchy below the Geometry Class.

PointArray, LineArray, TriangleArray, and QuadArray are intended for shapes consisting of points, lines, triangles, and quadrilaterals. The "Strip" versions order faces so that vertices can be reused, but the most efficient models are built with the "Indexed" subclasses. These define faces using the indices of the vertex coordinates rather than the coordinates themselves. For example, a die created with IndexedQuadArray defines its six faces using just eight unique coordinates.

However, the simplest way to define a shape is by employing a class that isn't in the hierarchy of Figure 7 – GeometryInfo uses an indexed geometry very similar to IndexedGeometryArray *and* includes utility classes that make the generation of shape information simpler.

2.1. Implementing Simple3D.tetra()

The two tetrahedron in UseTetra.java are made by calls to Simple3D.tetra() and Simple3D.texTetra() which are defined below. They utilize Simple3D.color() and Simple3D.texture() to generate the shape's appearance, but the geometry is handled by a three-argument version of tetra() which employs the GeometryInfo class:

```
  public static Shape3D tetra(double width, double height,
                                    double length, String colorNm)
  { return new Shape3D( tetra(width, height, length),
                        Simple3D.color(colorNm) );
  }


  public static Shape3D texTetra(double width, double height,
                                    double length, String texFnm)
  { return new Shape3D( tetra(width, height, length),
                        Simple3D.texture(texFnm));
  }


  public static GeometryArray tetra(double width,
                                  double height, double length)
  /* a triangular tetrahedron of height, -width to width
     on the x-axis, and length along the z-axis
  */
  { GeometryInfo gi = new GeometryInfo(GeometryInfo.TRIANGLE_ARRAY);

    Point3d[] pts = new Point3d[4];
    pts[0] = new Point3d(0, 0, length);
    pts[1] = new Point3d(-width, 0, 0);
    pts[2] = new Point3d(width, 0, 0);
    pts[3] = new Point3d(0, height, 0);
    gi.setCoordinates(pts);

    int[] indices = {0,1,2, 0,3,1, 0,2,3, 2,1,3};
                  // base, left,  right,  back    in ccw order
    gi.setCoordinateIndices(indices);

    NormalGenerator ng = new NormalGenerator();
    ng.generateNormals(gi);

    return gi.getGeometryArray();
  }  // end of tetra()
```

Every shape definition must include vertices and the faces that use those coordinates. The GeometryInfo() constructor can simplify that process by stating that its shape is made from triangles or quadrilaterals; tetra() plumps for GeometryInfo.TRIANGLE_ARRAY to utilize faces made from triangles.

The tetrahedron's four vertices are given first, then its four triangular faces are defined using triplets of vertex *indices*:

```
int[] indices = {0,1,2, 0,3,1, 0,2,3, 2,1,3};
               // base, left,  right,  back    in ccw order
gi.setCoordinateIndices(indices);
```

The code is easier to understand by referring to Figure 6, while keeping in mind that indexing starts at 0. TRIANGLE_ARRAY requires that the faces be defined as

independent triangles, which means that the four triplets in indices[] could be given in any order (e.g. left, right, base, back). However, the indices within a triangle should follow a counter-clockwise order assuming that a face is being viewed from outside the model. This becomes important when texture coordinates are added, but it's good practice to employ this ordering in all situations.

If the shape's color will be affected by light in the rendered scene, then the model also requires normal vectors for all of its faces. Fortunately, one of the utilities supported by GeometryInfo is the NormalGenerator class which can generate these normals for you. The result can be seen in Figure 5 where the tetrahedron faces reflect different amounts of light.

2.2. The Die Again

The marble texture wrapped around the tetrahedron in Figure 5 was placed there by Simple3D.texture() which utilizes a rather simple mapping of the image. This is usually good enough, especially for general-purpose textures such as marble, sand, grass, and so on. However, if you need to precisely position a texture, then the shape will need to have texture coordinates.

The third version of the die rendered by Dice.java calls diceGeometryGI() which utilizes GeometryInfo to wrap the diceFaces.png image (see Figure 1) around the cube. The result is shown in Figure 8.

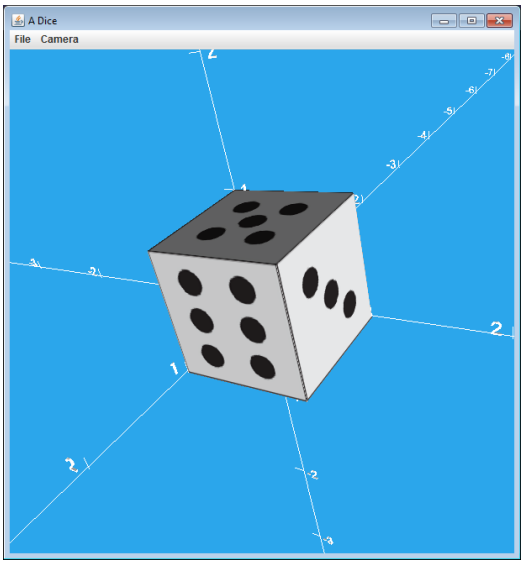

Figure 8. Using GeometryInfo and Texture Coordinates.

diceGeometryGI() specifies the vertices and faces as before, but also defines texture coordinates and their mapping onto each face:

```
private static GeometryArray diceGeometryGI()
// 6 square faces; 8 vertices
{
  GeometryInfo gi = new GeometryInfo(GeometryInfo.QUAD_ARRAY);
```

```
    // 8 vertices
    Point3f[] coords = {
          new Point3f(0,0,0), new Point3f(1,0,0),
          new Point3f(1,1,0), new Point3f(0,1,0),
          new Point3f(0,0,1), new Point3f(1,0,1),
          new Point3f(1,1,1), new Point3f(0,1,1)
    };

    // coordinates defining 6 faces
    int[] coordIndices = {
           0,1,2,3,  0,1,5,4,    // back, bottom
           1,2,6,5,  2,3,7,6,    // right, top
           3,0,4,7,  4,5,6,7     // left, front
    };

    gi.setCoordinates(coords);
    gi.setCoordinateIndices(coordIndices);

    // 12 coords of a 3 x 2 grid division of the texture
    TexCoord2f[] tex = {
          new TexCoord2f(0, 1), new TexCoord2f(1f/3, 1),
          new TexCoord2f(2f/3, 1), new TexCoord2f(1, 1),
          new TexCoord2f(0, 0.5f),new TexCoord2f(1f/3, 0.5f),
          new TexCoord2f(2f/3, 0.5f), new TexCoord2f(1, 0.5f),
          new TexCoord2f(0, 0), new TexCoord2f(1f/3, 0),
          new TexCoord2f(2f/3, 0), new TexCoord2f(1, 0)
    };

    // 6 faces from the 3 x 2 grid
    int[] texIndices = {
           0,4,5,1,   1,5,6,2,      // one, two
           2,6,7,3,   5,9,10,6,     // three, five
           6,10,11,7, 8,9,5,4       // four, six in diceFaces.png
    };

    gi.setTextureCoordinateParams(1, 2); // 1 set of 2d tex coords
    gi.setTextureCoordinates(0, tex);
    gi.setTextureCoordinateIndices(0, texIndices);

    NormalGenerator ng = new NormalGenerator();
    ng.generateNormals(gi);

    return gi.getGeometryArray();
  }  // end of diceGeometryGI()
```

It helps to draw a diagram (Figure 9) showing the die's vertices and their indices.

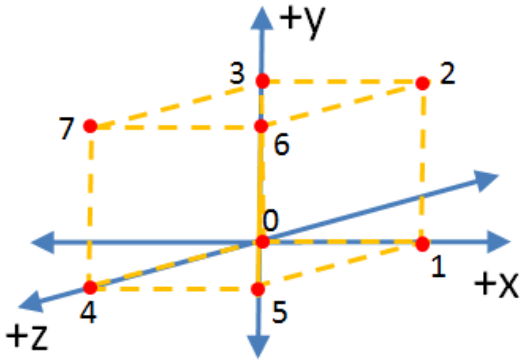

Figure 9. The Coordinates and Indices Used by the GeometryInfo Die.

The vertices are slightly different from the earlier versions of the die; in particular, the back, lower, left corner is located at the origin.

One important change between diceGeometryGI() and Simple3D.tetra() is that the GeometryInfo constructor in the former uses GeometryInfo.QUAD_ARRAY rather than TRIANGLE_ARRAY since this model consists of quadrilateral faces not triangles.

The texture coordinates are defined as a list of 2D texture objects utilizing the (s,t) coordinate system. The "s" axis corresponds to the x-axis of the image and runs between 0 and 1; the "t" axis maps to the image's y-axis and also runs between 0 and 1. The (s,t) origin is positioned at the lower left corner of the image, and the (s,t) unit square is stretched to match the dimensions of the image. For diceFaces.png (see Figure 1), the resulting (s,t) space looks like Figure 10.

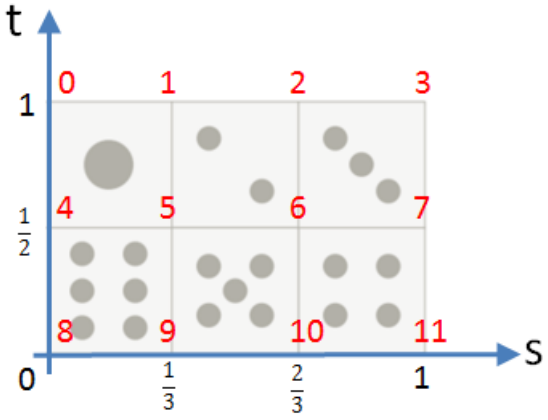

Figure 10. The (s,t) Texture Space for the Die Texture.

The red numbers in Figure 10 correspond to the indices of the Texture2f objects in diceGeometryGI()'s tex[] array.

The next step is to map the texture coordinates onto the die's faces, which is done by matching the texture coordinate indices (i.e. 0 through 11) to the vertex indices in coordIndices[] array. This is done by calls to setTextureCoordinateParams(), setTextureCoordinates(), and setTextureCoordinateIndices().

This is quite hard to visualize, so Figure 11 illustrates how the front face (which uses vertex indices (4,5,6,7)) is matched to the texture coordinates indices (8,9,5,4) for the "6" in dicefaces.png.

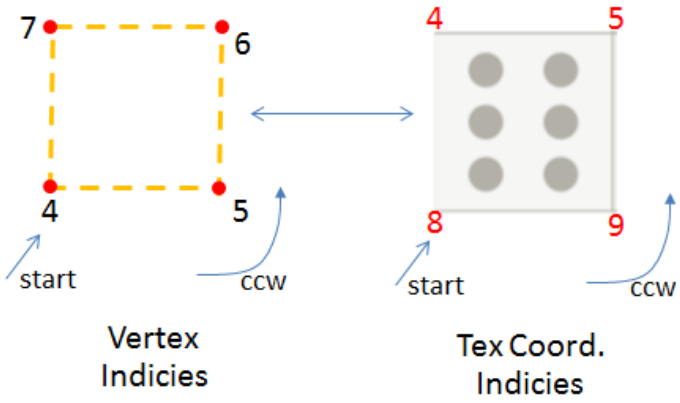

Figure 11. Mapping Vertices and Texture Coordinate Indices
for the Front Face of the Die.

The two index sequences must use the same counter-clockwise order, and the same starting position (i.e. the lower left corner). Actually, the starting index for the texture can be varied (e.g. we could change the texture sequence to (9,5,4,8)), but that would cause the texture to be rotated when mapped onto the face.

NormalGenerator is once again employed to create normals for the faces. Note, that this tool should only be called after all of the coordinates have been added to the GeometryInfo object.

GeometryInfo vs. IndexedGeometryArray

GeometryInfo can always be replaced by one of the subclasses of IndexedGeometryArray (see Figure 7), and the fourth function in Dice.java does just that. diceGeometryQuad() utilizes the IndexedQuadArray class to create a model based on indexed quadrilaterals, the same approach as in diceGeometryGI().

The result is shown in Figure 12.

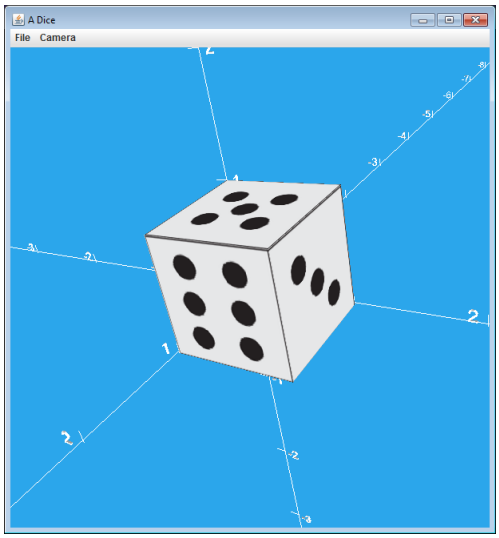

Figure 12. The Die Rendered with an IndexedQuadArray.

On first glance, the die in Figure 12 looks the same as the version created with GeometryInfo (Figure 8), but there is one difference – no light effects, as represented by the darker top and lighter right face of the die in Figure 8. The reason is that NormalGenerator only works on GeometryInfo objects, and I didn't define any normal vectors in diceGeometryQuad(). Apart from this, the creation of vertices and texture coordinates, and their indices, is very similar between the two functions. But from here on I'll restrict myself to using GeometryInfo.

2.3. The Platonic Solids

Platonics.java generates a scene containing one of the Platonic solids (https://en.wikipedia.org/wiki/Platonic_solid), 3D shapes whose faces all use the same regular polygon with equal angles and sides.

The five solids are shown in Figure 13.

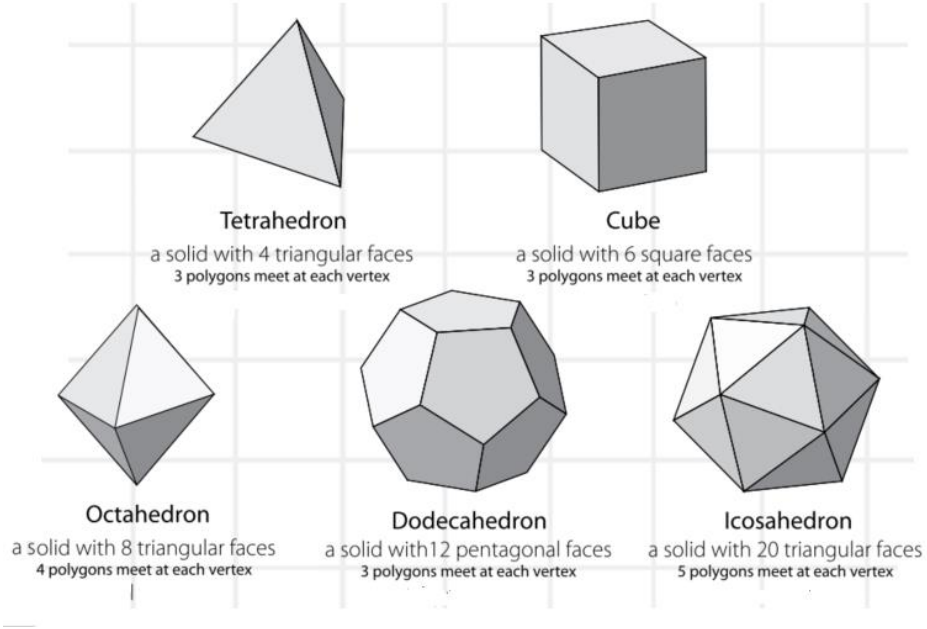

Figure 13. The Platonic Solids.

Figure 14 shows the results of calling Platonics.java in different ways.

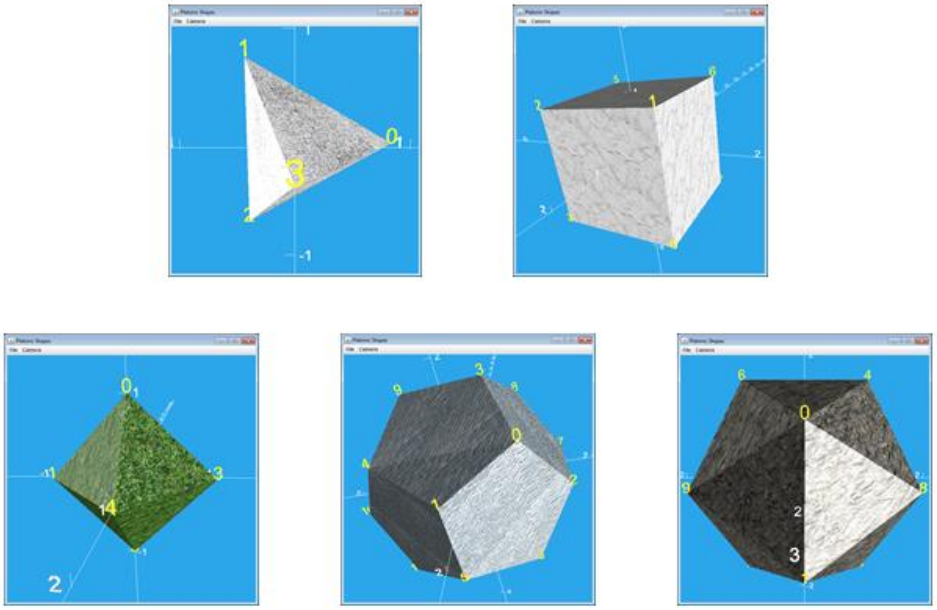

Figure 14. Invoking Platonics.java.

The main() function reads a letter from the command line to decide which shape to build, and enters a multi-way branch which calls labelCoords() and createTexShape() with the necessary data arrays. For example, the branch that generates the cube is:

```
    else if (shapeCh == 'c') {
      System.out.println("Shape: cube");
      coordsBG = labelCoords(cubeCoords());
      shape = createTexShape(cubeCoords(), cubeIndices(),
                             cubeStrips(), "images/marble.jpg");
    }
```

The coordsBG BranchGroup is assigned a tree of 3D labels which position vertex index values next to the shape's corners.

After the multi-way branch, the labels and shape are added to the scene:

```
    // build scene graph
    BranchGroup sceneBG = new BranchGroup();
    sceneBG.addChild(shape);
    sceneBG.addChild(coordsBG);

    j3d.rootBG.addChild(sceneBG);
```

createTexShape() is passed arrays of vertices and face indices which are processed by createGeom(), and the texture is wrapped around the shape by Simple3D.texture().

```
  private static Shape3D createTexShape(Point3d[] coords,
                                int[] indices, int[] stripCounts,
                                String texFnm)
  {
    GeometryArray geom = createGeom(coords, indices, stripCounts);
    return new Shape3D(geom, Simple3D.texture(texFnm));
  }
```

createGeom() utilizes GeometryInfo to create the vertices and the faces.

```
  private static GeometryArray createGeom(Point3d[] coords,
                                int[] indices, int[] stripCounts)
  {
    GeometryInfo gi = new GeometryInfo(GeometryInfo.POLYGON_ARRAY);
    gi.setCoordinates(coords);
    gi.setCoordinateIndices(indices);
    gi.setStripCounts(stripCounts);

    NormalGenerator ng = new NormalGenerator();
    ng.setCreaseAngle( Math.toRadians(35));  // for icosahedron
    // System.out.println("Crease angle: " +
    //          Math.toDegrees(ng.getCreaseAngle()));
```

```
    ng.generateNormals(gi);

    return gi.getGeometryArray();
  }  // end of createGeom()
```

createGeom() employs three features not seen before. The first is that the GeometryInfo() constructor utilizes GeometryInfo.POLYGON_ARRAY rather than TRIANGLE_ARRAY or QUAD_ARRAY. This means that the function is able to create all the Platonic solids including the dodecahedron, which uses pentagons for its faces (i.e. a 5-sided shape). However, a POLYGON_ARRAY-based GeometryInfo object must be assigned a stripCounts[] array to specify the number of sides in each face. For the dodecahedron, which consists of twelve pentagonal faces, the array will be {5,5,5, 5,5,5, 5,5,5, 5,5,5}.

Another new aspect of createGeom() is the modification of the crease angle used by NormalGenerator; it is the angular difference between two adjacent faces that must be exceeded before they're assigned different normals. The default value is 44 degrees, which is fine for all of the Platonic solids except the icosahedron whose faces are around 41.81 degrees apart.

If the crease angle is left unchanged, then the edges between the faces of the icosahedron appear rounded or even invisible, as in Figure 15.

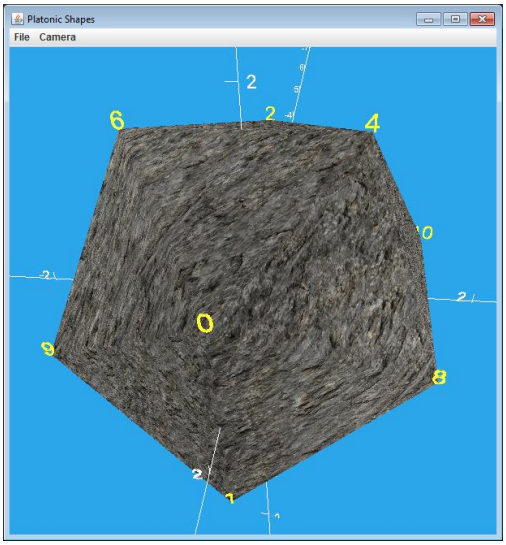

Figure 15. The Icosahedron with an Unmodified Crease Angle.

The Cube's Coordinates

In the case of the Platonic cube, createGeom() is passed arrays returned by three functions:

```
      shape = createTexShape(cubeCoords(), cubeIndices(),
                             cubeStrips(), "images/marble.jpg");
```

These functions are defined like so:

```
private static Point3d[] cubeCoords()
{
  Point3d[] coords = {   // 8 vertices
     new Point3d(1, -1, 1),          // coord 0
     new Point3d(1, 1, 1),
     new Point3d(-1, 1, 1),
     new Point3d(-1, -1, 1),

     new Point3d(-1, -1, -1),
     new Point3d(-1, 1, -1),
     new Point3d(1, 1, -1),
     new Point3d(1, -1, -1)
  };
  return coords;
}


private static int[] cubeIndices()  // 6 square faces
{
  int indices[] = {
         0,1,2,3,    // front (CCW)
         4,5,6,7,    // back
         0,7,6,1,    // right
         3,2,5,4,    // left
         1,6,5,2,    // top
         4,7,0,3     // bottom
  };
  return indices;
}


private static int[] cubeStrips()  // 6 square faces
{  return new int[]{4,4,4,4,4,4};  }
```

The vertices in cubeCoords() are defined in a different order from the cube in Figure 9, and the cube's center is at the origin, as confirmed by Figure 16.

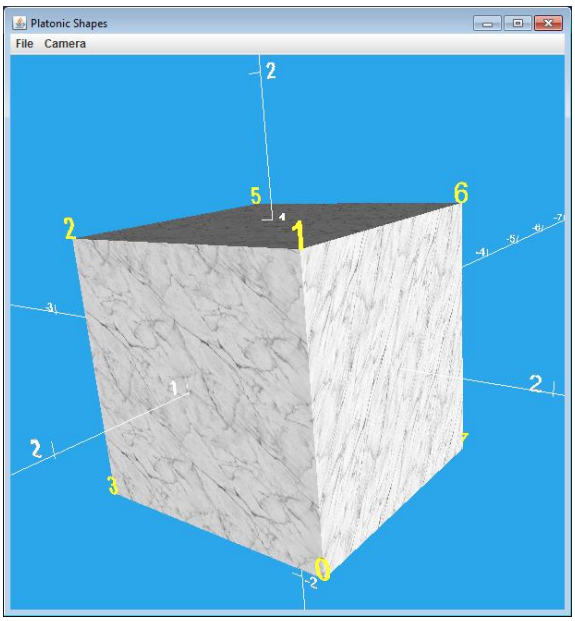

Figure 16. The Cube Drawn by Platonics.java.

The change in the ordering of the vertices affects the ordering of the indices for the faces in cubeIndicies(), but each is still defined in counter-clockwise order.

3. Using the Wavefront OBJ File Format

My reasons for championing the Wavefront OBJ format were given in chapter 1, section 4: it's simple, text-based, supported by Java 3D via the ObjectFile class and by almost every 3D modeling tool and viewer. The fact that it's almost as old as Methuselah should be seen as an advantage.

The OBJ format, and its accompanying MTL format for the model's appearance, are well documented (e.g. see https://en.wikipedia.org/wiki/Wavefront_.obj_file and https://cs.wellesley.edu/~cs307/readings/obj-ojects.html), but the Java 3D documentation for the ObjectFile class (https://download.java.net/media/java3d/javadoc/1.3.2/index.html?com/sun/j3d/loaders/objectfile/ObjectFile.html) is probably the best starting point since it only talks about the subset of OBJ and MTL elements supported in Java 3D.

I'll explain OBJ and MTL features by looking at several cube models, which also means that I can contrast the OBJ cubes with the one made using GeometryInfo in the previous section (i.e. Figure 16). The key observation is both approaches are very similar.

3.1. A Plain Cube

The simplest OBJ model only uses two elements: vertices defined using the "v" operator, and faces using the "f" operator. cube.obj is:

```
# cube.obj

# vertices
v  0.0  0.0  0.0
v  0.0  0.0  1.0
v  0.0  1.0  0.0
v  0.0  1.0  1.0
v  1.0  0.0  0.0
v  1.0  0.0  1.0
v  1.0  1.0  0.0
v  1.0  1.0  1.0

# faces (2 triangles per square side)
# back
f  1 7 5
f  1 3 7

# left
f  1 4 3
f  1 2 4

# top
f  3 8 7
f  3 4 8

# right
f  5 7 8
f  5 8 6

# bottom
f  1 5 6
f  1 6 2

# front
f  2 6 8
f  2 8 4
```

The vertices define a 1 x 1 x 1 cube whose lower, back, left corner is at the origin. In this respect it's similar to the die created by diceGeometryGI() in section 2.2, but without texturing.

The faces are defined in terms of the indices of the "v" coordinates, but starting from 1 (i.e. "v 0.0 0.0 0.0" is deemed to be vertex 1), not 0 as in GeometryInfo. Another difference is that OBJ faces are always triangles, and so each quadrilateral side of the cube must be represented by two triangles; I've tried to emphasize this point by adding newlines between the six pairs of "f" commands although blank lines are ignored in the OBJ format.

Figure 17 shows cube.obj loaded into MeshLab (https://www.meshlab.net/) on the left, and into Java 3D using LoadModel.java on the right.

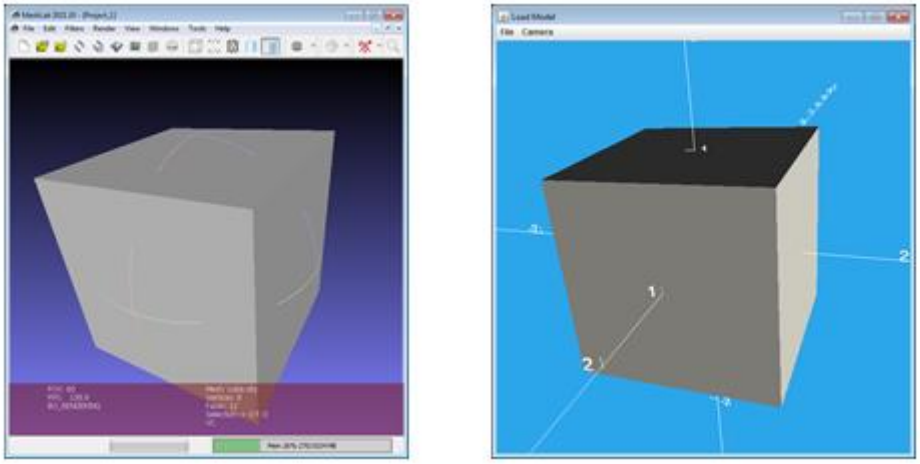

Figure 17. Cube.obj in MeshLab and Java 3D.

As usual, it helps to draw a diagram of the vertices and their indices, as in Figure 18.

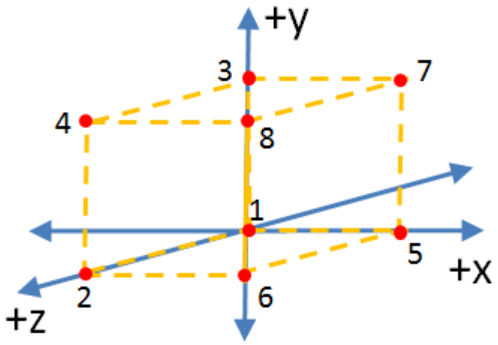

Figure 18. The vertices of cube.obj.

A close study of the screenshot of the Java 3D window in Figure 17 reveals that the cube is **not** positioned like the cube in Figure 18. The Java 3D version is centered at the origin while cube.obj vertices position its lower, back, left corner at the origin.

The code for LoadModel.java:

```
public class LoadModel
{
  public static void main(String[] args)
  {
    if (args.length != 1) {
      System.out.println("Usage: run LoadModel <filename>");
      System.exit(1);
    }

    Simple3D j3d = new Simple3D("Load Model", false, true);
```

```
    BranchGroup modelBG = Simple3D.loadModel(args[0]);
    // Simple3D.colorShape(modelBG, "blue");
    // Simple3D.textureShape(modelBG, "images/wood.jpg");

    if (modelBG != null)
      j3d.rootBG.addChild(modelBG);
  }

} // end of class LoadModel
```

The cube's repositioning is a *feature* of Simple3D.loadModel(), not a bug. The function calls the ObjectFile() constructor with the ObjectFile.RESIZE flag which changes the object's vertices so that the shape is centered at (0,0,0) and all of its coordinates are resized to be in the range (-1,-1,-1) to (1,1,1). This ensures that every rendered model appears in the scene at a known point (the origin), and is a standard size.

The definition of a face triangle uses a triplet of coordinate indices, as in Figure 19 which shows the two triangles for the front face of the cube.

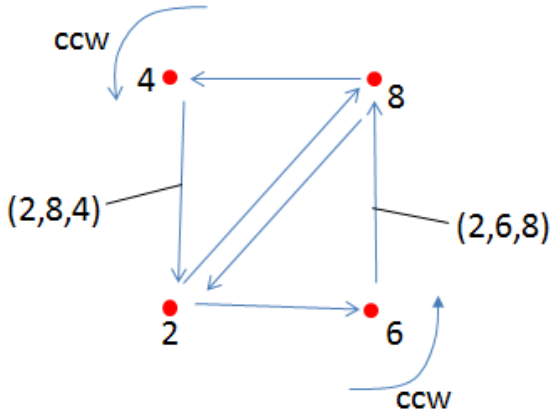

Figure 19. Two Triangles for a Cube Face.

The indices must be in counter clockwise order.

A nice feature of MeshLab and Java 3D's ObjectFile class is that a model's colors will be affected by light even when no vertex normals are included in the data (as in the case of cube.obj). ObjectFile employs NormalGenerator to calculate the necessary normals.

3.2. A Textured Cube

The cube in cube.obj has no appearance data, and so is rendered in gray. The simplest way to add color or texturing is by calling Simple3D.colorShape() or Simple3D.textureShape() in LoadModel.java; examples of those calls are commented out in that file. Figure 20 shows the results of executing one or other of those lines.

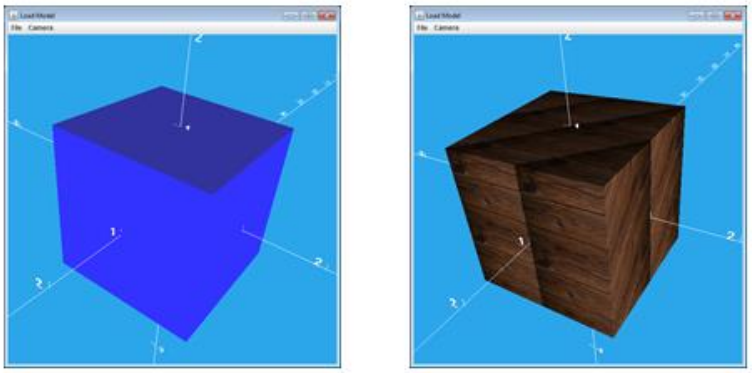

Figure 20. A Blue and a Wooden Cube.

If better texturing is required (e.g. as in the case of the die in section 2.2; Figure 8), then the OBJ file will need to utilize texture coordinates and a MTL file to hold the texture's appearance. The resulting cubeTex.obj file is:

```
# cubeTex.obj
# a cube with normals and a single texture

mtllib standard.mtl

# vertices
v  0.0  0.0  0.0
v  0.0  0.0  1.0
v  0.0  1.0  0.0
v  0.0  1.0  1.0
v  1.0  0.0  0.0
v  1.0  0.0  1.0
v  1.0  1.0  0.0
v  1.0  1.0  1.0

# normals
vn  0.0  0.0  1.0
vn  0.0  0.0 -1.0
vn  0.0  1.0  0.0
vn  0.0 -1.0  0.0
vn  1.0  0.0  0.0
vn -1.0  0.0  0.0

# texture coords
vt 0.0 0.0
vt 1.0 0.0
vt 0.0 1.0
vt 1.0 1.0

usemtl standard
```

```
# faces  (vertex / texture / normal indices)
# back
f  1/2/2  7/3/2  5/1/2
f  1/2/2  3/4/2  7/3/2
# left
f  1/1/6  4/4/6  3/3/6
f  1/1/6  2/2/6  4/4/6
# top
f  3/3/3  8/2/3  7/4/3
f  3/3/3  4/1/3  8/2/3
# right
f  5/2/5  7/4/5  8/3/5
f  5/2/5  8/3/5  6/1/5
# bottom
f  1/1/4  5/2/4  6/4/4
f  1/1/4  6/4/4  2/3/4
# front
f  2/1/1  6/2/1  8/4/1
f  2/1/1  8/4/1  4/3/1
```

Six normals are defined (the lines that start with "vn"), corresponding to the six axes arrows in Figure 21.

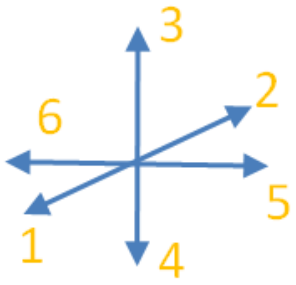

Figure 21. The Vertex Normals for the Cube.

The yellow numbers in Figure 21 are the indices of the normals, which are used in the "f" elements.

cubeTex.obj also includes four texture coordinates (the lines that start with "vt") which correspond to the four corners of the (s,t) texture space in Figure 22.

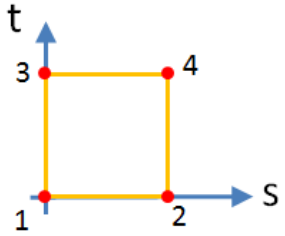

Figure 22. The Texture Coordinates in the (s,t) Space.

The black numbers in Figure 22 are the indices of the texture coordinates, which are used in the "f" elements.

The indices of the "vn" and "vt" elements (i.e. 1 to 6 for the normals, and 1 to 4 for the texture coordinates) are linked to their corresponding vertex coordinates by each face element. For example, the two "f" triangles for the front quadrilateral of the cube become:

```
# faces  (vertex / texture / normal indices)
# front
f  2/1/1  6/2/1  8/4/1
f  2/1/1  8/4/1  4/3/1
```

The three numbers separated by the "/" are vertex / texture / normal indices. Note that only the normal index "1" is used in these two "f" elements since both triangles are for the front quadrilateral whose normal points along the +z axis.

The texture coordinates in the "f" elements define two triangles (see Figure 23) which must have the same ordering as the two vertex triangles (see Figure 19).

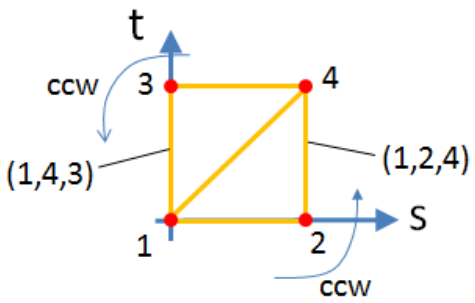

Figure 23. Two Texture Triangles for a Cube Face.

The MTL file (standard.mtl) used by the cube is named in the "mtllib" command at the start of the OBJ file, and the texture and other appearance elements are referred to by using the name "standard" in the "usemtl" command.

standard.mtl defines a set of lighting colors and one texture:

```
# standard.mtl

newmtl standard
Ka 0.1 0.1 0.1
Kd 0.6 0.6 0.6
Ks 0.2 0.2 0.2
Ns 16.0
map_Kd marbleDark.jpg
illum 2   # full lighting
```

"Ka", "Kd", and "Ks" define ambient, diffuse, and specular colors using red/green/blue values between 0 and 1. "Ns" is the amount of shininess, and "illum" enables full lighting. The "map_Kd" element gives a filename that contains the

texture used for the diffuse and ambient colors; as such it overrides the "Ka" and "Kd" elements, at least in Java 3D's implementation of the OBJ format.

The OBJ, MTL, and JPG files must all be in the same directory, and be correctly named (upper and lowercase letters are considered different, and spaces in the names confuse Java 3D's loader, and so should be avoided).

The end result is shown in Figure 24, loaded by MeshLab on the left, and by Java 3D on the right.

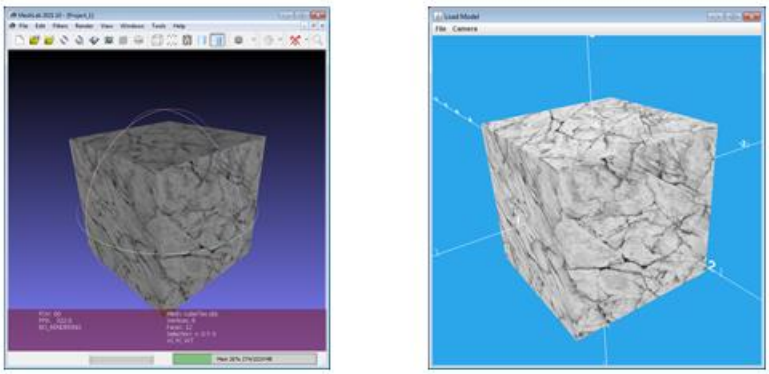

Figure 24. CubeTex.obj in MeshLab and Java 3D.

If texturing similar to the die in section 2.2 (Figure 8) is required, then the MTL file must utilize multiple "map_Kd" lines, each referring to a different "newmtl" name. For example, dice.mtl defines six materials:

```
# dice.mtl

newmtl one
Ka 0.1 0.1 0.1
Kd 0.6 0.6 0.6
Ks 0.2 0.2 0.2
Ns 16.0
map_Kd pic1.jpg
illum 2

newmtl two
Ka 0.1 0.1 0.1
Kd 0.6 0.6 0.6
Ks 0.2 0.2 0.2
Ns 16.0
map_Kd pic2.jpg
illum 2

newmtl three
Ka 0.1 0.1 0.1
Kd 0.6 0.6 0.6
Ks 0.2 0.2 0.2
Ns 16.0
map_Kd pic3.jpg
illum 2
```

```
newmtl four
Ka 0.1 0.1 0.1
Kd 0.6 0.6 0.6
Ks 0.2 0.2 0.2
Ns 16.0
map_Kd pic4.jpg
illum 2

newmtl five
Ka 0.1 0.1 0.1
Kd 0.6 0.6 0.6
Ks 0.2 0.2 0.2
Ns 16.0
map_Kd pic5.jpg
illum 2

newmtl six
Ka 0.1 0.1 0.1
Kd 0.6 0.6 0.6
Ks 0.2 0.2 0.2
Ns 16.0
map_Kd pic6.jpg
illum 2
```

The six JPG files used by the map_Kd lines in the materials contain the images in Figure 3.

In cubeMultiTex.obj, a face (or faces) requiring a particular texture must be preceded by a "usemtl" line that refers to the corresponding "newmtl" name in dice.mtl. **Also**, each group of faces must be assigned a group name with the "g" command:

```
# cubeMultiTex.obj
# a cube with normals and many textures

mtllib dice.mtl

# vertices
v  0.0  0.0  0.0
v  0.0  0.0  1.0
v  0.0  1.0  0.0
v  0.0  1.0  1.0
v  1.0  0.0  0.0
v  1.0  0.0  1.0
v  1.0  1.0  0.0
v  1.0  1.0  1.0

# normals
vn  0.0  0.0  1.0
vn  0.0  0.0 -1.0
vn  0.0  1.0  0.0
vn  0.0 -1.0  0.0
vn  1.0  0.0  0.0
vn -1.0  0.0  0.0

# texture coords
vt 0.0 0.0
vt 1.0 0.0
vt 0.0 1.0
vt 1.0 1.0
```

```
# faces  (vertex / texture / normal indices)
g back
usemtl one
f  1/2/2  7/3/2  5/1/2
f  1/2/2  3/4/2  7/3/2

g left
usemtl four
f  1/1/6  4/4/6  3/3/6
f  1/1/6  2/2/6  4/4/6

g top
usemtl five
f  3/3/3  8/2/3  7/4/3
f  3/3/3  4/1/3  8/2/3

g right
usemtl three
f  5/2/5  7/4/5  8/3/5
f  5/2/5  8/3/5  6/1/5

g bottom
usemtl two
f  1/1/4  5/2/4  6/4/4
f  1/1/4  6/4/4  2/3/4

g front
usemtl six
f  2/1/1  6/2/1  8/4/1
f  2/1/1  8/4/1  4/3/1
```

The result is shown in Figure 24, loaded by MeshLab on the left, and by Java 3D on the right.

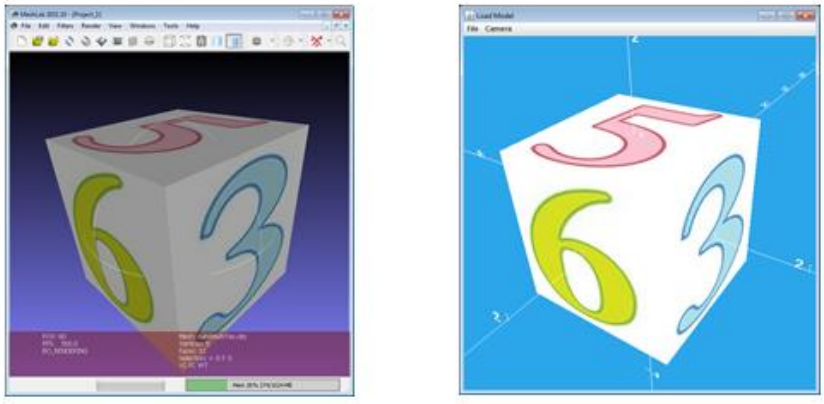

Figure 24. CubeMultiTex.obj in MeshLab and Java 3D.

3.3 OBJ Resources

Almost all the OBJ models included in the Simple3D download weren't created by me, but downloaded from various Web sites. One of the best places for free models is Free3D (https://free3d.com/) which has a subsection dedicated to OBJ files (https://free3d.com/3d-models/obj). But, you should have a look at the other formats as well since it's fairly easy to convert most of them to OBJ. MeshLab (https://www.meshlab.net/) has quite a versatile export capability, but another tool which supports more types is the Spin3D converter (https://www.nchsoftware.com/3dconverter/index.html).

By far the best tool for creating 3D models is blender (https://www.blender.org/), although it is rather complex. On the plus side, there are many online tutorials and books about how to use it.

Most of the Platonic solids in models/shapes/, including a few Archimedean and Kepler-Poinsot polyhedra, came from George W. Hart's informative "The Encyclopedia of Polyhedra" website (http://www.georgehart.com/virtual-polyhedra/vp.html). However, due to its great age (from the turn of the century) most of its models are in the venerable VRML 1.0 format. Even Spin3D doesn't support VRML 1.0, but fortunately a tool called vr1tovr2 (http://www.interocitors.com/polyhedra/vr1tovr2/) can convert v1.0 of VRML to version 2. It's possible to import that type into MeshLab and export it in OBJ form. The most reliable export setting is to disable the creation of normals and vertex colors. This generates the simplest kind of OBJ file, consisting of just vertices ("v" lines) and faces ("f" lines).

Problems Converting to OBJ

Very few of the OBJ files that I downloaded, and especially those files converted from other formats, could be loaded into Java 3D without some 'tweaking'. Fortunately, this is usually straightforward since the OBJ and MTL files are plain text.

The most common problem was that an OBJ model would refer to a missing MTL file. This can be fixed by edited the OBJ file to refer to standard.mtl, which is part of the Simple3D download, and was used back in section 3.2 (Figure 24).

Another problem was incorrect filenames – a name might include a defunct pathname or spaces. Java 3D requires the OBJ, MTL, and texture files to be all in the same directory.

Many files use the OBJ "o" command, which isn't supported by Java 3D's ObjectFile. Indeed, the OBJ and MTL formats contain a bewildering mix of additional commands (a good source for information about them is http://paulbourke.net/dataformats/obj/ and http://paulbourke.net/dataformats/mtl/). The simplest solution is to comment out the offending commands, and hope for the best!

To help identify troublesome OBJ commands, the Simple3D download includes ExamineObj.java in the models/ folder. It reads an OBJ file and reports on what it finds. For example:

```
java ExamineObj hexSphere.obj
  Line 3: o hexagon_sphere_Icosphere
   -- warning: the Java 3D loader does not recognize this; comment it
```

```
out
Total No. of lines: 7045
 No. of "v" lines: 1600
 No. of "f" lines: 1920
 No. of "vt" lines: 1600
 No. of "vn" lines: 1920
 No. of "s" lines: 1
```

Opening hexSphere.obj in a text editor shows that line 3 does indeed contain an "o" command, which should be commented out by inserting a "#" at the start of the line.

Incidentally, although I haven't used the OBJ "s" command in my examples, it is understood by Java 3D's ObjectFile. It's used to switch on/off smoothing between adjacent faces which have similar normals, and corresponds to the crease angle feature in Java 3D's NormalGenerator. For shapes, such as the icosahedron, where the default OBJ smoothing is too severe, the effect can be switched off by changing the command to "s off".

If ExampleObj.java discovers that the OBJ file refers to a MTL file, it also analyzes it, as seen in the following example:

```
java ExamineObj bird/Hummingbird.obj
  Line 4: mtllib 10027_Hummingbird_v1_iterations-2.mtl
  Line 21271: usemtl 01___Default
Total No. of lines: 28210
 No. of "v" lines: 6938
 No. of "f" lines: 6936
 No. of "vt" lines: 7379
 No. of "vn" lines: 6937
 No. of "s" lines: 1
 No. of "g" lines: 1
  Group names: [Hummingbird]
--------
Examining MTL file
  Line 15: map_Ka 10027_Hummingbird_v1_Diffuse.jpg
  Line 16: map_Kd 10027_Hummingbird_v1_Diffuse.jpg
Materials defined: [01___Default]
```

In this case, ExamineOBJ.java finds that Hummingbird.obj refers to 10027_Hummingbird_v1_iterations-2.mtl, which uses the texture in 10027_Hummingbird_v1_Diffuse.jpg. No errors were reported, so these files must all be present in the bird/ folder.

4. Manipulating OBJ Models in Java 3D

The main way of utilizing an OBJ model is by calling Simple3D.loadModel(), as shown in Part 2, section 4, and in section 3.1.

As I mentioned in section 3.1, Simple3D.loadModel() employs Java 3D's ObjectFile with the ObjectFile.RESIZE flag to ensure that the model is centered at the origin, and resized to occupy space between (-1,-1,-1) to (1,1,1). This may not be quite right, as we saw in section 3.1, and there are two ways to fix the problem. I'll describe them by loading home.obj from models/house/ using the following code in PosHome.java:

```
  public static void main(String[] args)
  {
    Simple3D j3d = new Simple3D("Position Home", true, true);

    BranchGroup modelBG = Simple3D.loadBasicModel(
                                        "models/house/home.obj");
    j3d.rootBG.addChild( modelBG );
  }
```

PosHome.java utilizes Simple3D.loadBasicModel(), a less fancy version of Simple3D.loadModel(), which causes the house to be rendered as in Figure 25. (What I mean by 'fancy' will be apparent by the end of this section.)

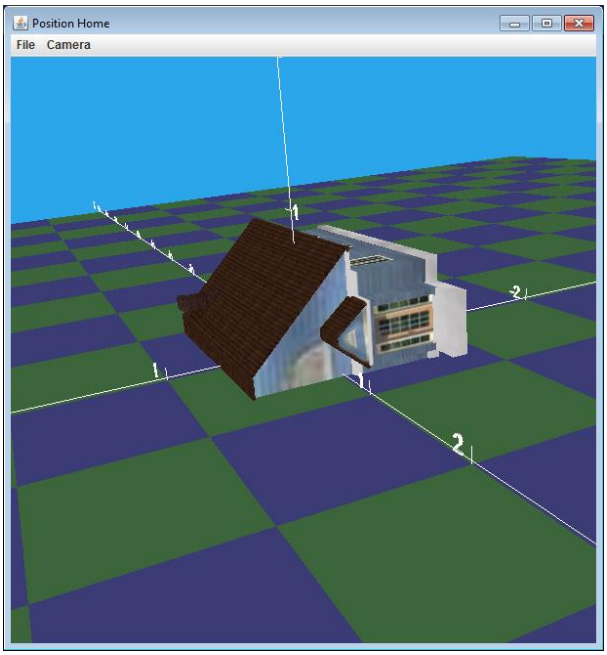

Figure 25. A Poorly Positioned House.

Note that ObjectFile has worked correctly, centering the shape at the origin, and resizing it to be between (-1,-1,-1) and (1,1,1). Nevertheless, the orientation could be improved.

One solution is to extend the code in PosHome.java so the scene graph includes TransformGroups to rotate, scale, and position the house so that its base is resting on the floor grid. This means building a scene graph represented by the diagram:

```
positionTG --> scaleTG --> rotateTG --> "home model"
```

This is quite easy to code up, although deciding on the actual numbers that go into the three TransformGroups requires several test renderings.

The final program is:

```
  public static void main(String[] args)
  {
```

```
    Simple3D j3d = new Simple3D("Position Home", true, true);

    BranchGroup modelBG = Simple3D.loadBasicModel(
                                        "models/house/home.obj");

    TransformGroup rot1TG = Simple3D.rotX(-90);  // ccw around x-axis
    rot1TG.addChild(modelBG);   // rot1TG --> "home"

    TransformGroup scale1TG = Simple3D.scale(1.5);
    scale1TG.addChild(rot1TG);   // scale1TG --> rot1TG --> "home"

    TransformGroup pos1TG = Simple3D.setTranslation(0,1.05,0);
                // up the y axis
    pos1TG.addChild(scale1TG);
              // pos1TG --> scale1TG --> rot1TG --> "home"

    j3d.rootBG.addChild( Simple3D.createBG(pos1TG));
  }
```

Figure 26 shows the repositioning.

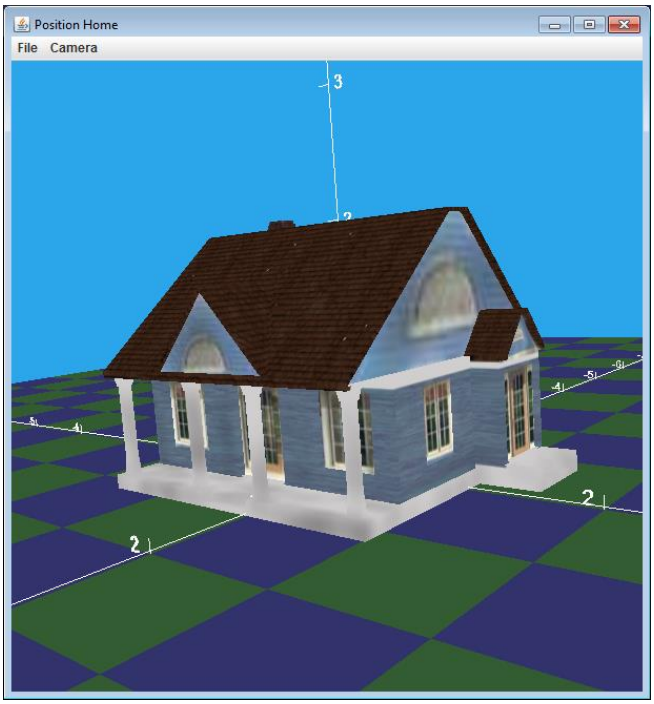

Figure 26. A Finely Appointed Home.

Once the 'magic' numbers for the repositioning have been worked out, it's possible to store them in a "Zero" file in the same folder as the OBJ file. For example, along with home.obj in models/house/, there's also a homeZero.txt file containing:

```
// House positioning at (0,0,0)
pos: 0 1.05 0
rots: -90 0 0
```

```
scale:  1.5
```

The "pos", "rots", and "scale" lines can appear in any order, but they're processed in the order used by PosHome.java: first the rotation, then scaling, then positioning.

Simple3D.loadModel() is 'fancier' than loadBasicModel() in that it also looks for an accompanying "Zero" file, and carries out its operations before rendering the model in the scene. This means that PosHome.java can be simplified to become:

```
  public static void main(String[] args)
  {
    Simple3D j3d = new Simple3D("Position Home", true, true);

    BranchGroup modelBG = Simple3D.loadModel(
                                    "models/house/home.obj");
    j3d.rootBG.addChild( modelBG );
  }
```

It produces the same result as in Figure 26.

5. Saving a Scene

Simple3D makes use of the OBJWriter class developed by Emmanuel Puybaret to save shapes (and their associated transformations) as OBJ models.

One minor limitation is that the shapes must be instances of GeometryArray or its subclasses, but that's the case for all the approaches described in this chapter.

Another tricky aspect is that the node being saved should have the necessary capabilities to examine its children where the geometries and Appearance nodes may be encoded. This criteria is met if the top-level BranchGroup is created using Simple3D.createBG().

SaveEarth.java is a version of the rotating Earth example from the start of Part 2, which also writes out the scene to the file savedEarth.obj after a few seconds.

```
  public static void main(String[] args)
  {
    Simple3D j3d = new Simple3D("The Rotating SaveEarth");

    Sphere sphere = Simple3D.texSphere(0.4f, "images/earth1.jpg");
    TransformGroup rotTG = Simple3D.orbit(sphere, 4000);
    BranchGroup bg = Simple3D.createBG(rotTG);

    j3d.rootBG.addChild(bg);
    Simple3D.pause(3000); // 3 sec delay

    try {
      OBJWriter ow = new OBJWriter("models/earth/savedEarth.obj");
                          // models/earth/ should already exist
      ow.writeNode(bg);
```

```
      ow.close();
      System.out.println("Write completed");
    }
    catch(IOException e)
    {  System.out.println(e);  }

    // j3d.exit();
  }  // end of main()
```

OBJWriter writes three files to the models/earth folder: savedEarth.obj (as requested), and MTL and JPG files to hold material information and the Earth texture.

savedEarth.obj is quite large since it stores vertices, faces, normals, and texture coordinates for a sphere, but ExamineObj.java can be employed to summarize it:

```
java ExamineObj earth/savedEarth.obj
  Line 1: mtllib savedEarth.mtl
  Line 3: usemtl 1
Total No. of lines: 5604
 No. of "v" lines: 730
 No. of "f" lines: 2704
 No. of "vt" lines: 709
 No. of "vn" lines: 1458
 No. of "g" lines: 1
  Group names: [1]
--------
Examining MTL file
  Line 8: map_Kd savedEarth_1.png
Materials defined: [1]
```

savedEarth.mtl is very short:

```
newmtl 1
illum 2
Ka 1.0 1.0 1.0
Kd 1.0 1.0 1.0
Ks 1.0 1.0 1.0
Ns 25.0
map_Kd savedEarth_1.png
```

SaveEarth.java pauses for three seconds before saving the model, but that's not absolutely necessary, and it's also not required that the bg BranchGroup be made live by being connected to Simple3D's j3d node. In other words, the following lines could be commented out:

```
    // j3d.rootBG.addChild(bg);
    // Simple3D.pause(3000);
```

This would mean that the scene displayed in the Java 3D window would be empty, and so it may be useful to uncomment the call to Simple3D.exit() at the end of main():

```
    j3d.exit();
```

This cleanly shuts down Java 3D and terminates the program after saving the OBJ file.