Chapter 9.5. Using a Lathe to Make Shapes

I've come up against the lack of useful shapes in Java 3D many times – there's only so much that can be done with the Box, Cone, Cylinder, and Sphere utilities, and then it's quite a large step to building with geometry arrays.

One solution is to follow the route used in chapter 9: create the shape in a 3D modeling package, and load it into Java 3D. The drawback is the need to learn the modeling software, and most are so fully featured that this endeavor is only for the stout-hearted.

The LatheShape3D class developed in this chapter aims at the middle ground: it allows a wide range of complicated looking shapes to be defined very easily, in terms of a few (x,y) coordinates, and colour or texture information. Internally it uses QuadArrays, Hermite curves, texture coordinates, etc., but these are all hidden.

Since LatheShape3D is a subclass of Shape3D, its objects can be readily integrated into the rest of a Java 3D application (e.g. moved, rotated).

Figure 1 shows a range of LatheShape3D objects. The text below the shapes is not part of their definitions; it was added to help distinguish between them

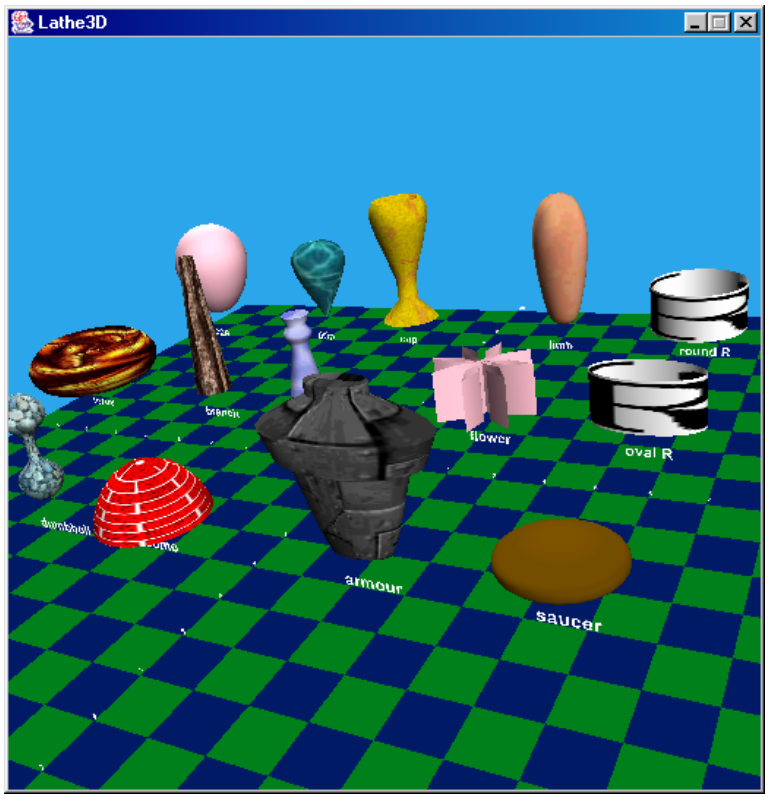

Figure 1. LatheShape3D objects.

A *lathe shape* is created in two stages: first the user defines a *lathe curve*, and this is automatically rotated around the y-axis to make the shape.

A lathe curve is a sequence of (x,y) points defining a series of curve segments and straight lines. When the curve is rotated, each point moves in a circular orbit back to its starting position. However, it is possible to modify the orbit to create an ellipse, or more complicated paths.

A lathe shape is pink by default, but this can be changed to a different colour or texture. In both cases, the shape will reflect light (i.e. a shape's faces always have normals).

A colour is defined using two Color3f objects: one for ambient illumination (typically a dark hue), the other for diffuse lighting.

A texture is wrapped round the shape starting from the middle of its back, continuing counter-clockwise round the front, and ending at its back again. The texture is also stretched in the y-direction so that it covers the shape vertically.

This chapter illustrates the following features:

- Hermite curves are used to represent the curve sequences inside a lathe curve;
- the QuadArray shape is made by revolving the lathe curve around the y-axis;
- the calculation of texture coordinates (s,t) is based on the shape's (x,y,z) coordinates, without using a TexCoordGeneration object (as was employed in chapter 9);
- normals are generated for the quads in the QuadArray with the aid of the GeometryInfo and NormalGenerator classes;
- LatheShape3D can be subclassed to modify the orbit of the surface revolution.

1. UML Diagrams for Lathe3D

Figure 2 shows the UML diagrams for all the classes in the Lathe3D application. The class names, and any public or protected methods are shown.

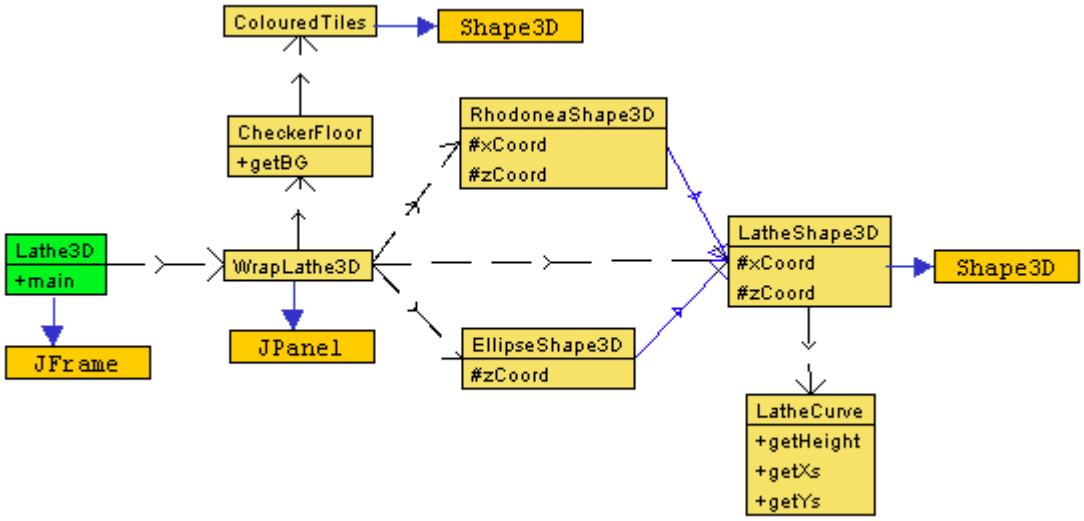

Figure 2. UML Class Diagrams for Lathe3D.

Lathe3D is the top-level JFrame, and essentially the same as in earlier examples.

WrapLathe3D sets up the 3D world like previous 'wrap' classes: it creates the check board, the lights, background, and mouse controls. The only change is a large method called addLatheShapes() which makes multiple calls to LatheShape3D (and its subclasses) to create the shapes shown in Figure 1.

CheckerFloor and ColouredTiles are unchanged from previous chapters.

LatheShape3D creates a shape, using a LatheCurve object to create the lathe curve.

The subclasses of LatheShape3D (EllipseShape3D and RhodoneaShape3D) are simple examples showing how the rotation path employed by LatheShape3D can be modified.

The code can be found in code/Lathe3D/.

2. The WrapLathe3D Class

The addLatheShapes() method contains a large number of lathe shapes examples. For instance:

```
TextureLoader texLd3 =
       new TextureLoader("textures/water.jpg", null);
Texture waterTex = texLd3.getTexture();

double xsIn15[] = {0, 0.1, 0.7, 0};
double ysIn15[] = {0, 0.1, 1.5, 2};
LatheShape3D ls2 = new LatheShape3D( xsIn15, ysIn15, waterTex);
displayLathe(ls2, -3.5f, -5.0f, "drip");
```

This produces the 'water globule', shown in close up in Figure 3.

Figure 3. A Drop of Water.

The coordinates for the lathe curve are supplies as two arrays, one for the x-values, one for the y's. Figure 4 shows the four coordinates plotted against the x- and y-axes.

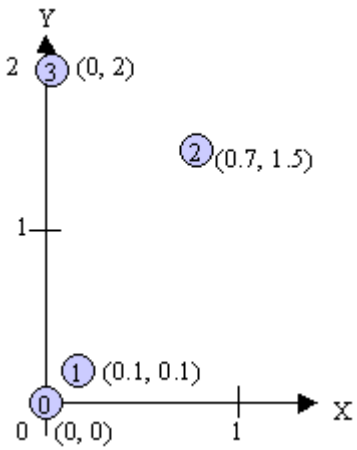

Figure 4. Coordinates for the Drop of Water.

The coordinates are ordered by increasing y-value, indicated by the numbered circles in the figure.

The x and y-values must be >= 0, and the first y-value must be 0. These restrictions simplify the calculation of the shape's height, which is used when mapping the texture over the shape's surface. A x-value may use a negative sign, but this has a special meaning (see below).

displayLathe() positions the shape at a given (x,z) location, 1.5 units above the XZ plane. The string is displayed as a Text2D object, a little below the shape.

```
private void displayLathe(LatheShape3D ls, float x, float z,
                                        String label)
{ // position the LatheShape3D object
  Transform3D t3d = new Transform3D();
  t3d.set( new Vector3f(x, 1.5f, z));
  TransformGroup tg1 = new TransformGroup(t3d);
  tg1.addChild( ls );
  sceneBG.addChild(tg1);

  // position the label for the shape
  Text2D message = new Text2D(label, white, "SansSerif",
                      72, Font.BOLD );
  t3d.set( new Vector3f(x-0.4f, 0.75f, z) );
  TransformGroup tg2 = new TransformGroup(t3d);
  tg2.addChild(message);
  sceneBG.addChild(tg2);
}
```

Due to the ordering of the coordinates, the base of a lathe shape is its 'origin' (most Java 3D utilities have their origins at their centers).

displayLathe() shows that a LatheShape3D instance can be used in the same way as a Shape3D object, due to LatheShape3D being a subclass of Shape3D.

2.1. Shapes with Curves and Lines

A LatheShape3D object can be built from a lathe curve made up of curve segments *and* straight lines, as illustrated by the cup example in Figure 5.

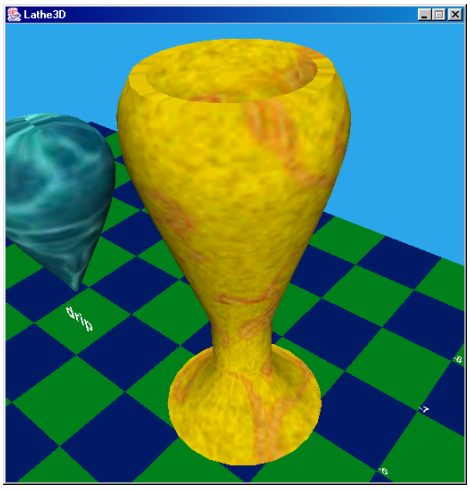

Figure 5. A Cup with Curves and Straights.

The code for this shape:


```
TextureLoader texLd10 =
          new TextureLoader("textures/swirled.jpg", null);
Texture swirledTex = texLd10.getTexture();

double xsIn2[] = {-0.001, -0.7, -0.25, 0.25, 0.7, -0.6, -0.5};
double ysIn2[] = { 0,      0,    0.5,  1,    2.5,  3,    3};
LatheShape3D ls3 = new LatheShape3D( xsIn2, ysIn2, swirledTex);
displayLathe(ls3, -1.0f, -5.0f, "cup");
```

A confusing aspect is the use of negative x-values, especially the starting value of –0.001. A plot of these points should ignore the negative signs, leading to Figure 6.

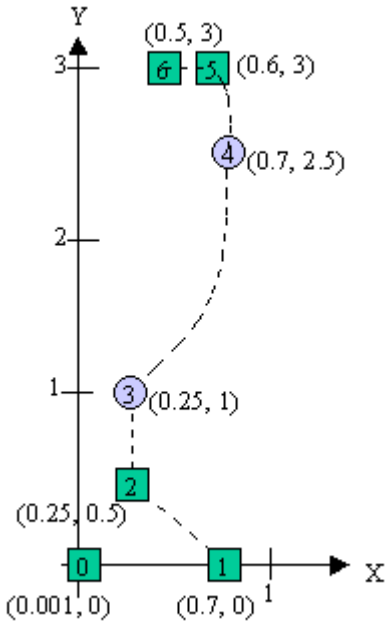

Figure 6. Coordinates for a Cup.

Points with negative x-values are represented by squares in the figure. Also, dotted lines have been added to indicate the curve segments or straight lines between the points.

The LatheCurve class (which LatheShape3D utilizes) can link points together with either curves or lines. If a coordinate has a negative x-value then a straight line is drawn from it to the next point in the sequence, otherwise a curve is created. Once this choice about the next segment has been made, any negative sign is discarded.

The negative sign labeling is admittedly a bit of a hack, but has the payoff that it keeps the specification of the shape's coordinates simple, without the need to introduce additional data structures/classes. The drawback is shown in Figure 6 – how to make a coordinate with an x-value equal to 0 become 'negative'? Our solution is to use a very small negative x-value (-0.001). This leaves a tiny hole in the base of the cup when the shape is rotated, but the hole is too little to be seen.

Figure 5 also shows that the texture is rendered on both sides of the lathe shape.

2.2. Shapes with Colours

If no texture is supplied (or the texture loading stage fails, returning null), then the shape is rendered in pink. This approach is seen in two examples in Figure 1: the 'egg' and the 'flower'.

It is possible to set the shape's colour by supplying *two* Color3f objects: one for ambient lighting, the other for diffuse illumination. Often, the ambient colour is a darker version of the diffuse one.

The 'saucer' example uses dark brown and brown, as seen in Figure 7.

Figure 7. The Brown Saucer is Landing!

The code that creates the saucer:

```
Color3f brown = new Color3f( 0.3f, 0.2f, 0.0f);
Color3f darkBrown = new Color3f(0.15f, 0.1f, 0.0f);

double xsIn10[] = {0, 0.75, 0.9, 0.75, 0};
double ysIn10[] = {0, 0.23, 0.38, 0.53, 0.75};
LatheShape3D ls14 = new LatheShape3D( xsIn10, ysIn10,
                                      darkBrown, brown);
displayLathe(ls14, 6.0f, 5.0f, "saucer");
```


2.3. Different Curve Rotations

LatheShape3D rotates each point in the lathe curve around the y-axis, marking out a circular path back to the point's starting position. However, it is possible to subclass LatheShape3D to modify the path.

EllipseShape3D sends the points in an elliptical orbit, while RhodoneaShape3D makes the points trace the outlines of petals. Details about how these are implemented will be given later.

Figure 8 shows two shapes. The LatheShape3D object at the back of the picture is a rotation of a single straight line in a circular orbit, covered with a texture of the letter "R". The object in the foreground is an EllipseShape3D object made with the same line, but forming an ellipse. The same "R" texture dresses the shape.

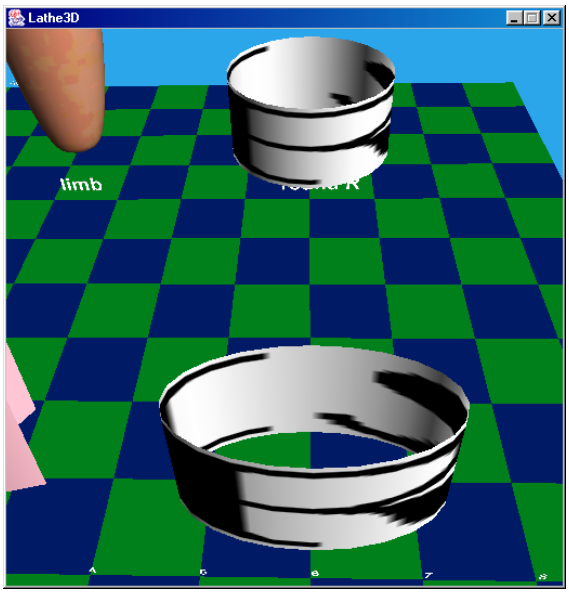

Figure 8. Circular and Elliptical R's.

The code that creates these shapes:

```
TextureLoader texLd1 = new TextureLoader("textures/r.gif", null);
Texture rTex = texLd1.getTexture();

double xsIn3[] = {-1, -1};   // straight line
double ysIn3[] = {0, 1};

LatheShape3D ls5 = new LatheShape3D( xsIn3, ysIn3, rTex);
displayLathe(ls5, 6.0f, -5.0f, "round R");

EllipseShape3D ls6 = new EllipseShape3D( xsIn3, ysIn3, rTex);
displayLathe(ls6, 6.0f, 0, "oval R");
```

These example show that the texture is stretched over the shape rather than being tiled. The left side of the texture is attached to the middle of the back of the shape, and wrapped around it in a counter-clockwise order. The texture is stretched in the vertical direction to cover the shape from its base (at y == 0) up to its maximum height.

A texture of a letter (such as "R") is a good test case when carrying out texture mapping since it allows issues to do with orientation, scaling, and texture repetition to be identified easily.

3. The LatheCurve Class

LatheCurve takes two arrays of x- and y-values as input, and creates two new arrays of x- and y- values representing the curve. The difference between the two pairs of arrays is the addition of interpolated points in the second group to represent curve segments. This change is illustrated by Figure 9, where the input arrays have three points, but the lathe curve arrays have 13.

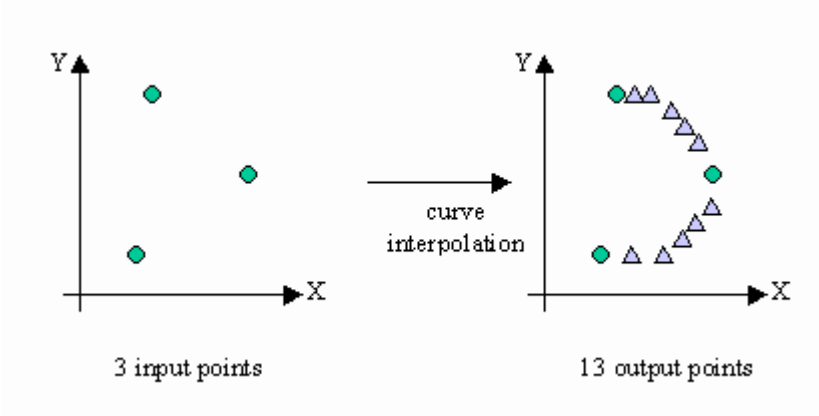

Figure 9. Interpolating Curves.

If all the input points became the starting and ending coordinates for curve segments, then the size of the output arrays would be (number of points-1)*(STEP+1) + 1, where STEP is the number of introduced interpolation points.

Unfortunately, the sizes of the output arrays is a more complicated matter, since points connected by straight lines don't require any additional points.

The size calculation is implemented in countVerts() which checks the sign of each x-value in the input array (xsIn[]) to decide on the number of output points.

```
private int countVerts(double xsIn[], int num)
{
  int numOutVerts = 1;
  for(int i=0; i < num-1; i++) {
    if (xsIn[i] < 0)   // straight line starts here
      numOutVerts++;
    else    // curve segment starts here
      numOutVerts += (STEP+1);
  }
  return numOutVerts;
}
```


3.1. Specifying Curve Segments

A crucial problem is how to interpolate the curve segment. There are many possible methods, including Bezier interpolation and B-splines. However, we use *Hermite curves*: a curve segment is derived from the positions and tangents of its two endpoints. Hermite curves are simple to calculate, and can be generated with minimal input from the user.

For a given curve segment, four vectors are required:

- P0: the starting point of the curve segment;
- T0: the tangent at P0, analogous to the direction and speed of the curve at that position;
- P1: the endpoint of the curve segment;
- T1: the tangent at P1.

Figure 10 illustrates the points and vectors for a typical curve.

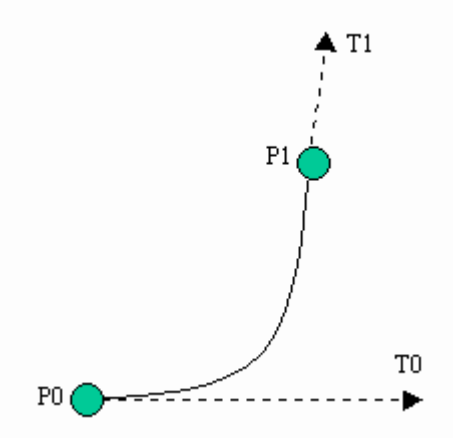

Figure 10. Hermite Curve Data.

The longer a tangent vector, the more the curve is 'pulled' in the direction of the vector before it begins to move towards the endpoint.

Four *blending functions* control the interpolations:

$$fh1(t) = 2t^3 - 3t^2 + 1$$

$$fh2(t) = -2t^3 + 3t^2$$

$$fh3(t) = t^3 - 2t^2 + t$$

$$fh4(t) = t^3 - t^2$$

As t varies from 0 to 1, fh1(t) and fh2(t) control the transition from P0 to P1, while fh3() and fh4() manage the transition from T0 to T1. The resulting x- and y- values are calculated like so:

```
x = fh1(t)*P0.x + fh2(t)*P1.x + fh3(t)*T0.x + fh4(t)*T1.x
y = fh1(t)*P0.y + fh2(t)*P1.y + fh3(t)*T0.y + fh4(t)*T1.y
```


The reader may be wondering where the blending functions come from. The maths is relatively straightforward, and can be found in any good computer graphics textbook, for example *Foley and Van Dam*. We will not discuss it here.

A good online explanation on Hermite curve interpolation can be found at http://www.cubic.org/~submissive/sourcerer/hermite.htm, written by Nils Pipenbrinck. A Java applet, written by Lee Holmes, which allows the user to play with natural splines, Bezier curves, and Hermite curves, is located at http://www.leeholmes.com/projects/grapher/.

3.2. Implementation

The Hermite curve interpolation points are calculated in makeHermite() in LatheCurve. The points are placed in xs[] and ys[], starting at index position startPosn. The P0 value is represented by x0 and y0, P1 by x1 and y1. The tangents are two Point2d objects, t0 and t1.

```
private void makeHermite(double[] xs, double[] ys, int startPosn,
                double x0, double y0, double x1, double y1,
                Point2d t0, Point2d t1)
{
  double xCoord, yCoord;
  double tStep = 1.0/(STEP+1);
  double t;

  if (x1 < 0)      // next point is negative to draw a line, make it
    x1 = -x1;      // +ve while making the curve

  for(int i=0; i < STEP; i++) {
    t = tStep * (i+1);
    xCoord = (fh1(t) * x0) + (fh2(t) * x1) +
             (fh3(t) * t0.x) + (fh4(t) * t1.x);
    xs[startPosn+i] = xCoord;

    yCoord = (fh1(t) * y0) + (fh2(t) * y1) +
             (fh3(t) * t0.y) + (fh4(t) * t1.y);
    ys[startPosn+i] = yCoord;
  }

  xs[startPosn+STEP] = x1;
  ys[startPosn+STEP] = y1;
}
```

The loop increments t in steps of 1/(STEP+1), where STEP is the number of interpolated points to be added between P0 and P1. The division is by (STEP+1) since the increment must include P1.

The loop does not add P0 to the arrays, since it will already have been added as the endpoint of the previous curve segment or straight line.

The blending functions are Java equivalents of the maths equations given above:

```
private double fh1(double t)
{  return (2.0)*Math.pow(t,3) - (3.0*t*t) + 1;  }

private double fh2(double t)
{  return (-2.0)*Math.pow(t,3) + (3.0*t*t); }
```



```
private double fh3(double t)
{  return Math.pow(t,3) - (2.0*t*t) + t; }

private double fh4(double t)
{  return Math.pow(t,3) - (t*t);  }
```

3.3. Calculating the Tangents

Another problem is how to specify the tangents: one is required for each point in the input sequence. Our aim is to reduce the burden on the user as much as possible, and so LatheCurve is capable of generating *all* the tangents by itself.

The first and last tangents of a curve are obtained by making some assumptions about its typical shape. Usually it will be convex, like the one in Figure 11, so the starting tangent will most likely point to the right, and the last tangent point to the left.

Both tangents should have a fairly large magnitude to ensure that the curve is suitably rounded at the bottom and top.

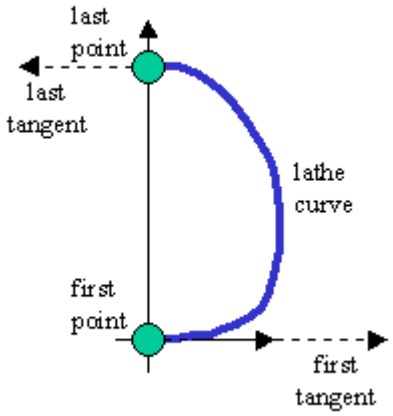

Figure 11. Typical Lathe Curve with Tangents.

The code which does this is located in LatheCurve's constructor:

```
Point2d startTangent =
    new Point2d((Math.abs(xsIn[1]) - Math.abs(xsIn[0]))*2, 0);

Point2d endTangent =
     new Point2d((Math.abs(xsIn[numVerts-1]) -
                  Math.abs(xsIn[numVerts-2]))*2, 0);
```

The xsIn[] array stores the user's x-values, and numVerts is the size of the array. The use of Math.abs() around the x-values is to ignore any negative signs due to the points being used to draw straight lines. The tangents are multiplied by 2 to make them stronger.

The intermediate tangents can be interpolated from the data points, using the Catmull-Rom spline equation:

$T_i = 0.5 * (P_{i+1} - P_{i-1})$

The tangent at point i is derived from the data points on either side. The method that does this is setTangent():

```
private void setTangent(Point2d tangent, double xsIn[],
                                double ysIn[], int i)
{
  double xLen = Math.abs(xsIn[i+1]) - Math.abs(xsIn[i-1]);
  double yLen = ysIn[i+1] - ysIn[i-1];
  tangent.set(xLen/2, yLen/2);
}
```

3.4. Building the Entire Curve

The for-loop in makeCurve() iterates through the input points (stored in xsIn[] and ysIn[]) building new arrays (xs[] and ys[]) for the resulting curve.

```
private void makeCurve(double xsIn[], double ysIn[],
                Point2d startTangent, Point2d endTangent)
{
  int numInVerts = xsIn.length;
  int numOutVerts = countVerts(xsIn, numInVerts);
  xs = new double[numOutVerts];    // seq after adding extra pts
  ys = new double[numOutVerts];

  xs[0] = Math.abs(xsIn[0]);    // start of curve is initialised
  ys[0] = ysIn[0];
  int startPosn = 1;

  // tangents for the current curve segment between two points
  Point2d t0 = new Point2d();
  Point2d t1 = new Point2d();

  for (int i=0; i < numInVerts-1; i++) {
    if (i == 0)
      t0.set( startTangent.x, startTangent.y);
    else   // use previous t1 tangent
      t0.set(t1.x, t1.y);

    if (i == numInVerts-2)    // next point is the last one
      t1.set( endTangent.x, endTangent.y);
    else
      setTangent(t1, xsIn, ysIn, i+1);   // tangent at pt i+1

    // if xsIn[i] < 0 then use a line to link (x,y) to next pt
    if (xsIn[i] < 0) {
      xs[startPosn] = Math.abs(xsIn[i+1]);
      ys[startPosn] = ysIn[i+1];
      startPosn++;
    }
    else {   // make a Hermite curve
      makeHermite(xs, ys, startPosn, xsIn[i], ysIn[i],
                              xsIn[i+1], ysIn[i+1], t0, t1);
      startPosn += (STEP+1);
    }
  }
```



```
  }  // end of makeCurve()
```

The loop responds differently if the current x-value is positive or negative. If it is negative then the coordinates are copied over to the output arrays unchanged (to represent a straight line). If the x-value is positive then makeHermite() is called to generate a series of interpolated points for the curve.

The two tangents, t0 and t1, are set for each coordinate. Initially, t0 will be the starting tangent, but normally it will be the t1 value from the previous calculation. At the end, t1 will be assigned the endpoint tangent, but normally it will be calculated with setTangent().

The new arrays of points, and the maximum height (largest y-value), are made accessible through public methods:

```
public double[] getXs()
{  return xs;  }

public double[] getYs()
{  return ys;  }

public double getHeight()
{  return height;  }
```

4. The LatheShape3D Class

A LatheShape3D object first creates a lathe curve using the points supplied by the user, then decorates it with colour or a texture. The choice between colour or texture is represented by two constructors:

```
public LatheShape3D(double xsIn[], double ysIn[], Texture tex)
{ LatheCurve lc = new LatheCurve(xsIn, ysIn);
  buildShape(lc.getXs(), lc.getYs(), lc.getHeight(), tex);
}


public LatheShape3D(double xsIn[], double ysIn[],
                    Color3f darkCol, Color3f lightCol)
// two colours required: a dark and normal version of the colour
{ LatheCurve lc = new LatheCurve(xsIn, ysIn);
  buildShape(lc.getXs(), lc.getYs(), lc.getHeight(),
                                       darkCol, lightCol);
}
```

Both versions of buildShape() call createGeometry() to build a QuadArray for the shape, then either a createAppearance() method for adding colour or one for laying down a texture.

4.1. Creating the Geometry

createGeometry() passes the lathe curve coordinates to surfaceRevolve() which returns the coordinates of the resulting shape. The coordinates are used to initialise a QuadArray, complete with vertex normals (to reflect light), and texture coordinates if a texture is going to be wrapped around the shape.

```
private void createGeometry(double[] xs, double[] ys,
                                  boolean usingTexture)
{
  double verts[] = surfaceRevolve(xs, ys);

  // use GeometryInfo to compute normals
  GeometryInfo geom = new GeometryInfo(GeometryInfo.QUAD_ARRAY);
  geom.setCoordinates(verts);

  if (usingTexture) {
    geom.setTextureCoordinateParams(1, 2);    // set up tex coords
    TexCoord2f[] texCoords = initTexCoords(verts);
    correctTexCoords(texCoords);
    geom.setTextureCoordinates(0, texCoords);
  }

  NormalGenerator norms = new NormalGenerator();
  norms.generateNormals(geom);

  setGeometry( geom.getGeometryArray() );   // back to geo array
}
```

The calculation of the normals is carried out by a NormalGenerator object, which requires that the coordinates be stored in a GeometryInfo object.

setTextureCoordinatesParams() specifies how many texture coordinate sets will be used with the geometry, and their dimensionality (Java 3D offers 2D, 3D, and 4D texture coordinates). The actual texture coordinates are calculated by initTexCoords(), and added to the geometry with setTextureCoordinates().

We will encounter NormalGenerator again, and other geometry utilities, in chapter 17 when a fractal landscape requires normals, and other optimizations.

4.2. "You Say You Want a Revolution" (sorry)

surfaceRevolve() generates the shape's coordinates by revolving the lathe curve clockwise around the y-axis in angle increments specified by ANGLE_INCR. This results in NUM_SLICES columns of points around the y-axis.

```
private static final double ANGLE_INCR = 15.0;
    // the angle turned through to create a face of the solid
private static final int NUM_SLICES = (int)(360.0/ANGLE_INCR);
```

The coordinates in adjacent slices are organized into quadrilaterals (quads). Each quad is specified by four points, with a point represented by 3 floats (for the x-, y-, and z-values). The points are organized in counter-clockwise order so that the quad's normal is facing outwards.

Figure 12 shows how two quads are defined. Each point is stored as three floats.

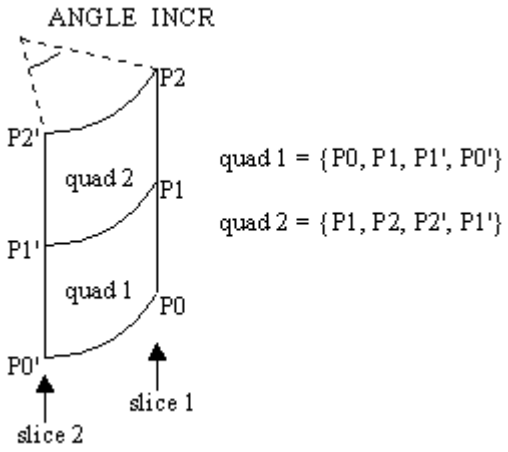

Figure 12. The Creation of Quads.

The surfaceRevolve() method:

```
private double[] surfaceRevolve(double xs[], double ys[])
{
  checkCoords(xs);
  double[] coords = new double[(NUM_SLICES) * (xs.length-1) *4*3];
  int index=0;
  for (int i=0; i < xs.length-1; i++) {
    for (int slice=0; slice < NUM_SLICES; slice++) {
      addCorner( coords, xs[i], ys[i],slice,index); // bottom right
      index += 3;

      addCorner( coords, xs[i+1],ys[i+1],slice,index); // top right
      index += 3;

      addCorner( coords, xs[i+1],ys[i+1],slice+1,index); //top left
      index += 3;

      addCorner( coords, xs[i],ys[i],slice+1,index); // bottom left
      index += 3;
    }
  }
  return coords;
}
```

The generated coordinates for the shape are placed in the coords[] array. surfaceRevolve()'s outer loop iterates through the coordinates in the input arrays, which are stored in increasing order. The inner loop creates the corner points for all the quads in each slice clockwise around the y-axis.

This means that the quads are built a ring at a time, starting at the bottom of the shape, and working up to its top.

addCorner() rotates an (x,y) coordinate around to the specified slice, and stores its (x,y,z) position in the coords[] array.

```
private void addCorner(double[] coords, double xOrig, double yOrig,
```



```
                                    int slice, int index)
{ double angle = RADS_DEGREE * (slice*ANGLE_INCR);

  if (slice == NUM_SLICES)  // back at start
    coords[index] = xOrig;
  else
    coords[index] = xCoord(xOrig, angle);  // x

  coords[index+1] = yOrig;   // y

  if (slice == NUM_SLICES)
    coords[index+2] = 0;
  else
    coords[index+2] = zCoord(xOrig, angle);   // z
}
```

The x- and z- values are obtained by treating the original x-value (xOrig) as a hypotenuse at the given angle, and projecting it onto the x- and z- axes (see Figure 13).

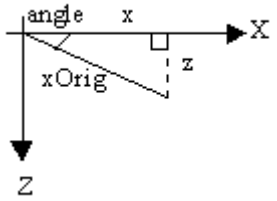

Figure 13. Obtaining new x- and z- Values.

The xCoord() and zCoord() methods:

```
protected double xCoord(double radius, double angle)
{  return radius * Math.cos(angle);  }

protected double zCoord(double radius, double angle)
{  return radius * Math.sin(angle);  }
```

These methods carry out a mapping from Polar coordinates (radius, angle) to Cartesian (x,y). Since the radius argument (xOrig) never changes, the resulting coordinates will always be a fixed distance from the origin, and so be laid out around a circle.

The methods are protected, so it is possible to override them to vary the effect of the radius and/or angle.

The algorithm in surfaceRevolve() and addCorner() comes from the SurfaceOfRevolution class by Chris Buckalew, which is part of his FreeFormDef.java example (see http://www.csc.calpoly.edu/~buckalew/474Lab6-W03.html).

4.3. Creating Texture Coordinates

In chapter 9, a TexCoordGeneration object mapped (s,t) values onto geometry coordinates (x,y,z). Unfortunately, the simplest TexCoordGeneration form only

supports planar equations for the translation of (x,y,z) into s and t. Often quadratic or cubic equations would be more useful. In those cases, it is arguably simpler to calculate s and t directly, without the TexCoordGeneration class, which is the approach utilized here.

The s value 0 is mapped to the middle of the shape's back face, and increased in value around the edge of the shape in a counter-clockwise direction until it reaches the back face again, when it equals 1. Also, t is given the value 0 at the base of the shape (where y equals 0), and increased to 1 until it reaches the maximum y-value. This has the effect of stretching the texture vertically.

Figure 14 shows the s mapping applied to a circle, from a viewpoint looking down on the XZ plane from above.

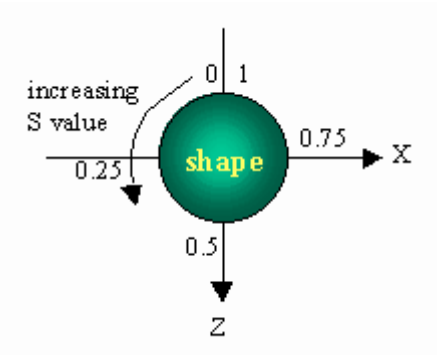

Figure 14. The s Mapping from Above.

Figure 14 gives a hint of how to calculate s simply: its value at a given (x,z) coordinate can be obtained from the angle that the point makes with the z-axis. This will range between π and –π (see Figure 15), which is easily converted into a value between 0 and 1.

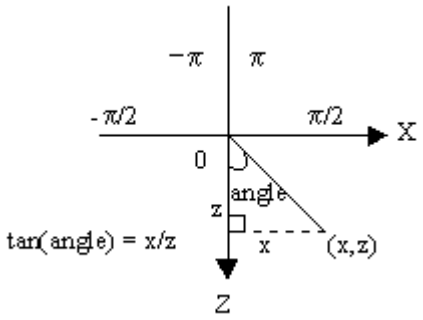

Figure 15. From Point to Angle.

The code for initTexCoords():

```
private TexCoord2f[] initTexCoords(double[] verts)
{
  int numVerts = verts.length;
  TexCoord2f[] tcoords = new TexCoord2f[numVerts/3];

  double x, y, z;
  float sVal, tVal;
  double angle, frac;

  int idx = 0;
  for(int i=0; i < numVerts/3; i++) {
    x = verts[idx];  y = verts[idx+1];  z = verts[idx+2];

    angle = Math.atan2(x,z);       // -PI to PI
    frac = angle/Math.PI;          // -1.0 to 1.0
    sVal = (float) (0.5 + frac/2);   // 0.0f to 1.0f

    tVal = (float) (y/height);  // 0.0f to 1.0f; uses height

    tcoords[i] = new TexCoord2f(sVal, tVal);
    idx += 3;
  }
  return tcoords;
}
```

The texture coordinates are stored in an array of TexCoord2f objects, each object holding a (s,t) pair.

The angle is obtained with a call to Math.atan2(), and the range of angles (π to $-\pi$) is scaled and translated to (0 to 1).

4.4. A Thin Problem

The mapping described in the last subsection has a flaw, which occurs in any quad spanning the middle of the shape's back face.

Figure 16 shows the "round R" example, with this problem visible as a thin "R" stretched down the middle of the shape's back. The letter is also reversed, when viewed from the back. Figure 16 should be compared with the "round R" example in Figure 8, rendered after the flaw was corrected.

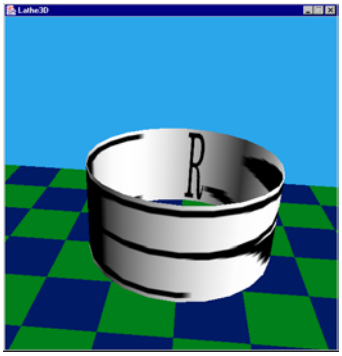

Figure 16. An Extra "R".

The same effect is apparent in all the other textured-wrapped shapes (although some shapes and textures make it harder to see).

The problem is that the quads which span the middle of the back face have coordinates at angles on either side of the –z axis. An example shows the problem in Figure 17.

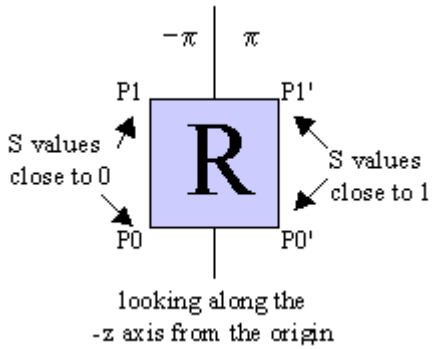

Figure 17. The Incorrect s Mapping.

P0 and P1 have angles near to –π, and so s values close to 0, while P0' and P1' have angles near to π, and so s values close to 1. Consequently, the s component of the texture will be drawn in its entirety in that one quad, as seen in Figure 16.

The solution is a bit of a hack. Each quad generates four TexCoord2f objects corresponding to the order of the coordinates of the quad (P0, P1, P1', P0'). In correctly textured quads, the s value for P0 is greater than P0' and P1 is greater than P1'. This is due to the surfaceRevolve() method rotating points clockwise around the y-axis.

In incorrectly textured quads, the reverse is true: P0 is less than P0' and P1 is less than P1'.

In correctTexCoords(), every group of four TexCoord2f objects is examined for this condition, and the offending textured coordinates for P0 and P1 are adjusted to be greater than those for P0' and P1'. The code:

```
private void correctTexCoords(TexCoord2f[] tcoords)
{
  for(int i=0; i < tcoords.length; i=i+4) {
    if( (tcoords[i].x < tcoords[i+3].x) &&
        (tcoords[i+1].x < tcoords[i+2].x)) { // should not increase
      tcoords[i].x = (1.0f + tcoords[i+3].x)/2 ; // between x & 1.0
      tcoords[i+1].x = (1.0f + tcoords[i+2].x)/2 ;
    }
  }
}
```

4.5. Making an Appearance

There are two createAppearance() method. One of them colours the shape with two colours (one for the light's ambient component, the other for diffuse illumination). This is achieved with a Material object:

```
Appearance app = new Appearance():
        :
Material mat = new Material(darkCol, black, lightCol, black, 1.0f);
       // sets ambient, emissive, diffuse, specular, shininess
mat.setLightingEnable(true);    // lighting switched on
app.setMaterial(mat);
setAppearance(app);
```

The other createAppearance() method sets the texture and a white Material object. The texture is combined with the colour using the MODULATE mode (see chapter 9 for some more details), which allows lighting and shading effects to be blended with the texture.

```
Appearance app = new Appearance();
      :
// mix the texture and the material colour
TextureAttributes ta = new TextureAttributes();
ta.setTextureMode(TextureAttributes.MODULATE);
app.setTextureAttributes(ta);

Material mat = new Material();   // set a default white material
mat.setSpecularColor(black);     // no specular color
mat.setLightingEnable(true);
app.setMaterial(mat);

app.setTexture( tex );

setAppearance(app);
```


5. Subclassing LatheShape3D

Figure 2 shows that LatheShape3D can be subclassed. This aim is to override its xCoord() and zCoord() methods, which control the shape of the path made by the lathe curve when it is rotated. These methods in LatheShape3D:

```
protected double xCoord(double radius, double angle)
{  return radius * Math.cos(angle);  }

protected double zCoord(double radius, double angle)
{  return radius * Math.sin(angle);  }
```

radius is the x-value of the point being rotated around the y-axis, and angle is the angle of rotation currently being applied. xCoord() and zCoord() return the new x-and z- values after the rotation has been applied.

5.1. The EllipseShape3D Class

An ellipse resembles a circle stretched in one direction. Another (more formal) way of characterizing the ellipse, is that its points all have the same sum of distances from two fixed points (called the foci).

The line which passes through the foci is called the *major axis*, and is the longest line through the ellipse. The *minor axis* is the line which passes through the center of the ellipse, at right angles to the major axis. See Figure 18 for examples of these.

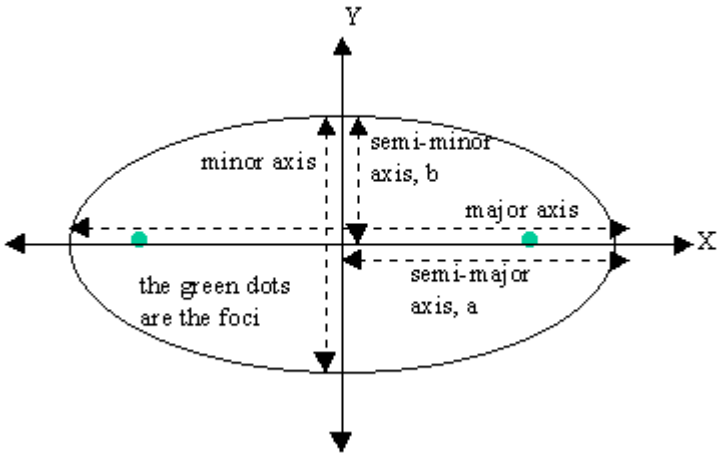

Figure 18. Elements of an Ellipse.

The *semi-major axis* (which we will call a) is half the length of the major axis: it runs from the center of the ellipse to its edge. There's also a *semi-minor axis* (half of the minor axis), which we will call b.

Figure 19 shows an ellipse with a semi-major axis of 2, semi-minor axis of length 3.

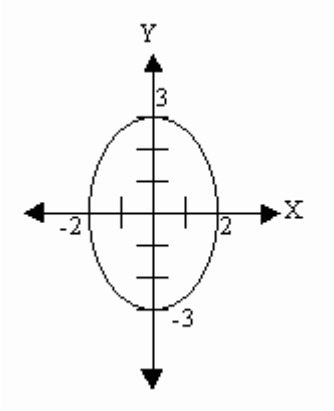

Figure 19. Another Ellipse.

The semi-major and semi-minor axes can be used to calculate the (x,y) coordinates of the ellipse:

x = semiMajorAxis * cos(angle)
y = semiMinorAxis * sin(angle)

In the case of the ellipse in Figure 19:

x = 2 * cos(angle) y = 3 * sin(angle)

The y equation can be rephrased by writing the semi-minor axis value as a scale factor applied to the semi-major axis number:

x = 2 * cos(angle) y = 1.5 * 2 * sin(angle)

The scale factor is 1.5.

Our ellipse is drawn over the XZ plane, and so the y equation will change to use z. Also, x initially equals the radius value, which occurs when the angle is 0. Thus:

radius = semiMajorAxis * cos(0), so radius = semiMajorAxis.

The ellipse equations now become:

x = radius * cos(angle)
z = scaleFactor * radius * sin(angle)

The x equation is the same as the xCoord() method in LatheShape3D, so does not need to be overridden. The zCoord() method does need changing, and becomes the following in EllipseShape3D:

```
protected double zCoord(double radius, double angle)
{  return  0.5 * radius * Math.sin(angle);  }
```

The scale factor is set to 0.5, which makes the semi-minor axis half the semi-major, which can be confirmed by examining the "oval R" example in Figure 8.

A weakness of this approach is that the user can not set the scale factor via a parameter of EllipseShape3D's constructor. The reason is that the xCoord() and zCoord() methods are called (indirectly) by the LatheShape3D constructor, and so must be fully specified before any code in the EllipseShape3D constructor is

executed. In other words, the scale factor (e.g. 0.5) must be hardwired into the EllipseShape3D class as a constant in zCoord().

The "armour" example also uses EllipseShape3D:

```
double xsIn9[] = {-0.01, 0.5, -1, -1.2, 1.4, -0.5, -0.5, 0};
double ysIn9[] = {0, 0, 1.5, 1.5, 2, 2.5, 2.7, 2.7};

EllipseShape3D ls13 = new EllipseShape3D( xsIn9, ysIn9, plateTex);
displayLathe(ls13, 3.0f, 5.0f, "armour");
```

Figure 20 shows the rendering of "armour".

Figure 20. The "armour" Ellipse.

5.2. RhodoneaShape3D

A rhodonea curve resembles the petals of a rose. The simplest way to define one is with an equation using polar coordinates:

r = a * cos(k*angle)

or

r = a * sin(k*angle)

These will lay down a curve with k or 2k petals depending on if k is an odd or even integer. a is the amplitude, affecting the length of the petals.

Some examples of rhodonea curves with different k values are given in Figure 20. They use the sine formulation.

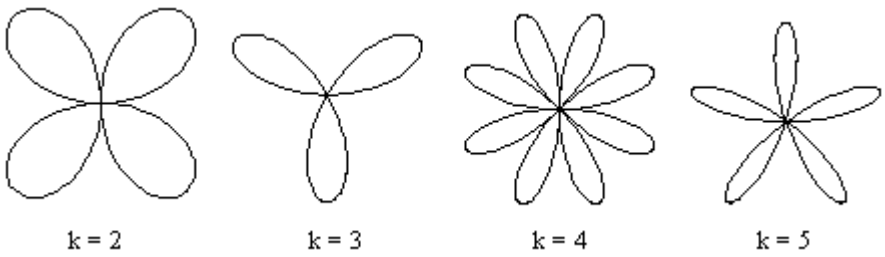

Figure 20. Some Rhodonea Curves.

We will use the cosine version of the rhodonea curve:

r = a * cos(k*angle)

This causes the first petal in the curve to be its largest when the angle equals 0.

Once r is obtained for a given angle, it can be translated to Cartesian coordinates with:

x = r * cos(angle)
y = r * sin(angle)

We store the initial x value in radius, which is the length of the first petal when the angle is 0, so:

radius = a * cos(0), so radius = a

The rhodonea equation then becomes:

r = radius * cos(k*angle)

In RhodoneaShape3D, k is set to be 4, and both xCoord() and yCoord() must be overridden:

```
protected double xCoord(double radius, double angle)
{ double r = radius * Math.cos(4 * angle);   // 8 petals
  return  r * Math.cos(angle);
}

protected double zCoord(double radius, double angle)
{ double r = radius * Math.cos(4 * angle);
  return  r * Math.sin(angle);
}
```

RhodoneaShape3D is used in the "flower" example, which is defined as:

```
double xsIn3[] = {-1, -1};
double ysIn3[] = {0, 1};

RhodoneaShape3D ls7 = new RhodoneaShape3D(xsIn3, ysIn3, null);
displayLathe(ls7, 3.0f, 0, "flower");
```

A vertical straight line of unit length is rotated, then coloured pink.

The resulting curve is shown in Figure 21.

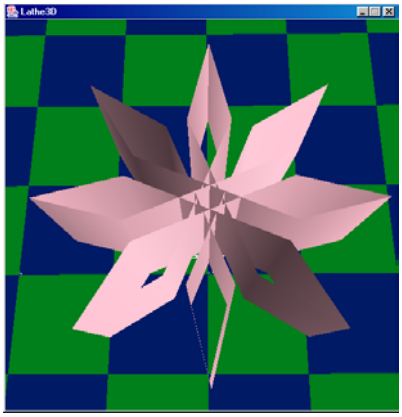

Figure 21. A Rough RhodoneaShape3D shape.

The curve is rather rough, due to the ANGLE_INCR setting in LatheShape3D (15 degrees between each slice). If this is reduced to 5 degrees, the result is more pleasing (see Figure 22).

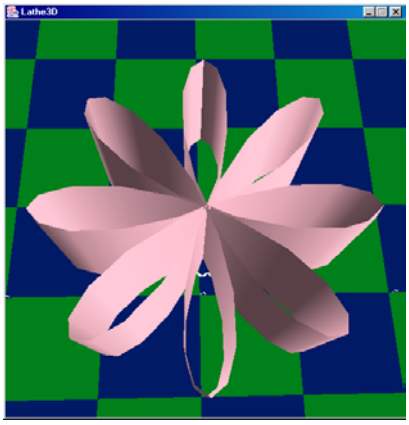

Figure 22. A Smoother RhodoneaShape3D shape.

The drawback of reducing the ANGLE_INCR value is the increase in the number of vertices generated for each shape.

5.3. Other Curves

A very nice resource for 2D curves is http://www.2dcurves.com/, which may suggest other ways of overriding LatheShape3D.

Some of the curves require a small ANGLE_INCR setting in order for fine details to become apparent.