## Spirograph Curves

The `spirofield` shader is an example of drawing a curve given its equation. A Spirograph curve can be described in terms of the container wheel radius `R`, revolving wheel radius `r`, and offset `b` of the pen/pencil from the center of the moving wheel. Given these values and an angle `a`, the radial location can be computed:

```
rho = (sqrt((R-r)*(R-r)+ b*b + 2*(R-r)*b*cos((1+R/r)*a)));
```

After a certain number of revolutions, the curve closes in on itself, creating a nice, symmetric, closed-loop pattern. The `LCM` (Least Common Multiple) between `R` and `r` is what determines the number of revolutions. Note that even though our function is called `LCM`, it is actually returning `LCM(R,r)/R`, which is the number of revolutions the smaller wheel has to make.

The usual way to draw a Spirograph curve on the computer is to vary the angle `a` in small increments using a `for()` loop, calculate `rho` each time, and convert `(rho,a)` into an `(x,y)` value using the standard polar-to-Cartesian conversion formula. But in a shader, we cannot do this because we do not have access to the entire surface being shaded all at once. We can only color the current point over which our shader gets evaluated. We can still plot our curve by testing if our current shading point lies on or near the curve and use the test result to color the point accordingly. Since the Spirograph curve loops back on itself in general, it is a bit tricky to detect if our current point lies on/near a curve point. We obtain the polar angle for our current point (via a Cartesian-to-polar conversion), calculate all curve points that lie along that angle, find the distance to each from our current shading point, and sort the distances.

The sorted distance values help us color the surface (see Figure 12.5). In the two leftmost images, the maximum of the distances is used to obtain a color. At the top right, the closest distance is used, and at the bottom right, the average of all the distances is used. Each of these three variations yields a distinct, identifiable result.

```
float LCM(float R,rv)
{
  float i=0,cm=1.0;

  if (mod(R,rv)==0)
    {
      cm = 1;
    }
  else
    {
      i=1;
      while(mod(R*i,rv)!=0)
```

```
      {
      i=i+1;
      }
      cm = i;
    }
  return cm;
}

float calcspiro(float R, rv, b, a)
{
  float rho;
  rho = (sqrt((R-rv)*(R-rv)+ b*b + 2*(R-rv)*b*cos((1+R/rv)*a)));
  return rho;
}// calcspiro()

surface spirofield(float R=10,rv=5,b=2.5,hoff=0.0,freq=1.0, calctype=0,f=1.0)
{
  float i, theta, rho, nrev, a, rsp, ss, tt;

  ss=s-0.5;
  tt=t-0.5;
  theta = atan(tt,ss);
  theta += 3.1415;
  rho = 2*sqrt(ss*ss+tt*tt);
  if((rho>((R-rv+b)/R))||(rho<((R-rv-b)/R)))
    {
      Ci = 0.25;
    }
  else
    {
      float deltad;
      color Ch, Cg;

      nrev = LCM(R,rv);
      if(0==calctype)
        {
          float maxdist;
          maxdist = -2;
          for(i=0;i<nrev;i+=1)
            {
              a = theta + (i)*2*PI;
              rsp = calcspiro(R,rv,b,a)/R;
              deltad = abs(rsp-rho);
              if(deltad>maxdist) maxdist=deltad;
```

```
            }
          maxdist *= (nrev*freq);
          maxdist = mod((maxdist+hoff),1.0);
          Ch = color "hsv" (maxdist,1,1);
          Cg = color (maxdist,maxdist,maxdist);
        }
      else if(1==calctype)
        {
          float mindist;
          mindist = 2;
          for(i=0;i<nrev;i+=1)
            {
              a = theta + (i)*2*PI;
              rsp = calcspiro(R,rv,b,a)/R;
              deltad = abs(rsp-rho);
              if(deltad<mindist)
                mindist=deltad;
            }
          mindist *= (nrev*freq);
          mindist = mod((mindist+hoff),1.0);
          Ch = color "hsv" (mindist,1,1);
          Cg = color (mindist,mindist,mindist);
        }
      else
        {
          float avdist=0;
          for(i=0;i<nrev;i+=1)
            {
              a = theta + (i)*2*PI;
              rsp = calcspiro(R,rv,b,a)/R;
              avdist += abs(rsp-rho);
            }
          avdist *= freq;
          avdist = mod((avdist+hoff),1.0);
          Ch = color "hsv" (avdist,1,1);
          Cg = color (avdist,avdist,avdist);
        }
      Ci = mix(Cg,Ch,f);
    }
}// spirofield()
```

Hopefully, the math in the shader does not come across as intimidating. The idea we are trying to get across is that with a multiply self-intersecting curve, such as one from a Spirograph, we could generate interesting patterns by considering not just the closest curve point to an arbitrary point (our shading sample location), but also a set of them that lie along the radial line containing the point. Given a set of such curve points, we can use the closest of those to our shading location, the farthest, or the average of all of them to derive a color for our shading sample.

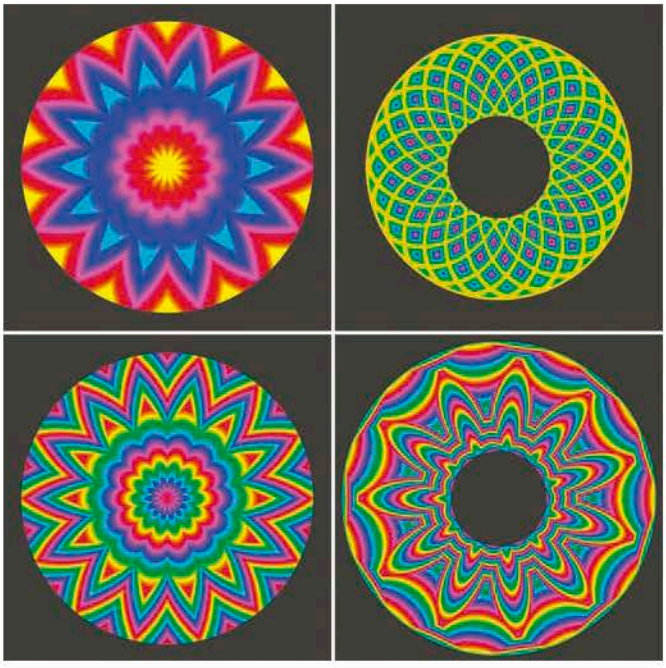

**Figure 12.5** Spirograph patterns.

Colorful animations can be created by simply varying `R`, `r`, or `b` in small increments. Try creating one of these as an exercise. Also, try "plotting" several other curves such as `sin()`, an ellipse, and so forth using this idea of deriving shading color from distance(s) to curve point(s).

## Mandelbrot Fractal

Fractal zooms create colorful, aesthetic imagery and are often coded as pixel routines. We can use `(s,t)` instead of pixel location `(x,y)` to compute fractal patterns over any surface.

The `mandel` shader contains the classic Mandelbrot fractal calculation. Starting with a location derived from the current shading point, we iteratively calculate new point locations. At the edges of the fractal, the successive new points tend to wander away from the starting location. After a certain number of iterations, a point is said to "escape" its neighborhood if it crosses a circle of preset radius centered at the starting location. We use the iteration count at the time of the escape to calculate a color for the starting point:

```
n=0; a=x; b=y;

  while(n<maxiter)
    {
      aa = a*a;
      bb = b*b;
      twoab = 2*a*b;
      if((aa+bb)>escape)
      {
        break;
      }
      n=n+1;
      a = aa-bb+x;
      b = twoab+y;
    }
  // convert n to a color value
  h = 0.5*(1+sin(huefreq*n/maxiter));
  Cmandel= color "hsv" (h,1,1);
```

Figure 12.6 shows representative images obtained from the shader. The first three images show three different "zoom" levels on a plane, while the fourth (bottom-right) image shows the fractal computed over a sphere. You can pan and zoom along the fractal's edges given a location `(xc,yc)` and zoom factor `sz`:

```
// convert s,t to (x,y)
xmin = xc-0.5*sz;
ymin = yc-0.5*sz;
x = xmin+sz*s;
y = ymin+sz*t;
```

(x,y) is now the starting point that gets iterated using the while() loop given above and in the shader below.

You can do a Web search for interesting coordinates (locations, zoom factors) on the Mandelbrot fractal that produce stunningly beautiful zooms. Like real terrain, such areas have been mapped out and even given names in some cases. Zooming into an interesting region produces nice animations. Be aware that after a certain magnification (small zoom factor sz), you start to reach the floating-point limit of your machine, at which point the zooms stop getting refined and instead start to become pixilated.

```
surface mandel(float xc=-.5, yc=0, sz=4, escape=256, maxiter=20,
               huefreq=1.0; float Ka=0, Kd=1, Ks=0,roughness=.1)
{
  float xmin,ymin,x,y,a,b,n,aa,bb,twoab,h;
  normal Nf;
  vector V;
  color Cmandel;

  // convert s,t to (x,y)
  xmin = xc-0.5*sz;
  ymin = yc-0.5*sz;
  x = xmin+sz*s;
  y = ymin+sz*t;


  n=0; a=x; b=y;

  while(n<maxiter)
    {
      aa = a*a;
      bb = b*b;
      twoab = 2*a*b;
      if((aa+bb)>escape)
      {
      break;
      }
      n=n+1;
      a = aa-bb+x;
      b = twoab+y;
    }

  // convert n to a color value
```

```
    h = 0.5*(1+sin(huefreq*n/maxiter));
    Cmandel= color "hsv" (h,1,1);

    Nf = faceforward( normalize(N), I );
    V = -normalize(I);
    Oi = Os;
    Ci = Os*Cs*Cmandel*(Ka*ambient() + Kd*diffuse(Nf)+ Ks*specular(Nf,V,
         roughness));
}// mandel()
```

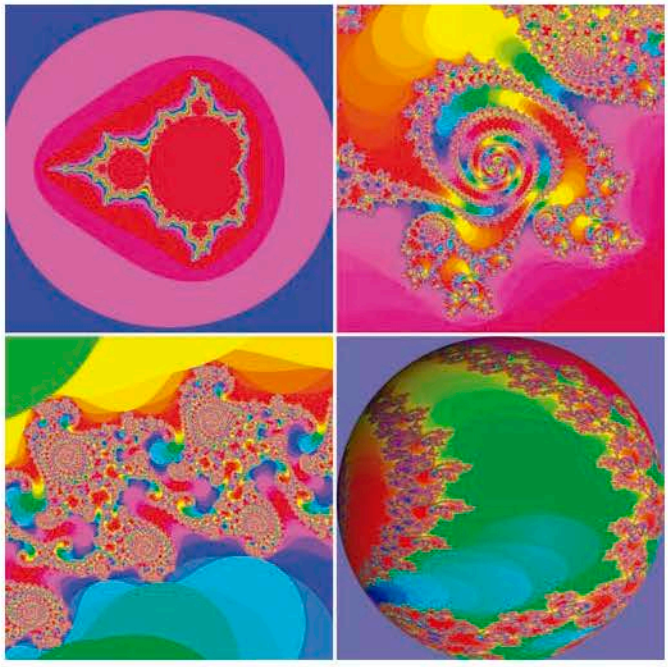

Figure 12.6 Mandelbrot fractals.

## noise() Function

The `noise()` function is one of the most useful building blocks for procedural pattern generation because it can be used to mimic the look of a multitude of naturally occurring patterns, both inorganic and organic.

`noise()` simply associates a unique value (scalar, vector, or color) with every point in 1D, 2D, 3D, or 4D space (in 4D we usually take the fourth axis to be time). Figure 12.7 shows a 1D scalar noise pattern, where distance varies along the X axis and the corresponding `noise()` value is shown both as a grayscale value and as a curve. The `noiseplot` shader that was used to generate the figure is shown after it.