Lecture 5: Affine Graphics
A Connect the Dots Approach to Two-Dimensional Computer Graphics

The lines are fallen unto me in pleasant places; Psalms 16:6

1. Two Shortcomings of Turtle Graphics

Two points determine a line. In Turtle Graphics we use this simple fact to draw a line joining the two points at which the turtle is located before and after the execution of each FORWARD command. By programming the turtle to move about and to draw lines in this fashion, we are able to generate some remarkable figures in the plane.

Nevertheless, the turtle has two annoying idiosyncrasies. First, the turtle has no memory, so the order in which the turtle encounters points is crucial. Thus, even though the turtle leaves behind a trace of her path, there is no direct command in LOGO to return the turtle to an arbitrary previously encountered location. Second, the turtle is blissfully unaware of the outside universe. The turtle carries her own local coordinate system -- her state -- but the turtle does not know her position relative to any other point in the plane. Turtle geometry is a local, intrinsic geometry; the turtle knows nothing of the extrinsic, global geometry of the external world. The turtle can draw a circle, but the turtle has no idea where the center of the circle might be or even that there is such a concept as a center, a point outside her path around the circumference.

These two shortcomings -- no memory and no knowledge of the outside world -- often make the turtle cumbersome to program. For example, even though it is straightforward to program the turtle to generate a regular polygon, it is not so easy to program the turtle to draw a rosette. A rosette is simply the collection of all the lines joining all pairs of vertices in a polygon. Nevertheless, though we can easily program the turtle to visit all the vertices of a polygon, we cannot simply command the turtle to draw the lines joining each pair of vertices because the turtle does not remember where these vertices are located. It is for this reason that in Turtle Graphics to generate the rosette, we need to precompute the distances between each pair of vertices and the angles between adjacent diagonals. Two points determine a line, but Turtle Graphics gives us access to only one pair of points at a time.

2. Affine Graphics

In Affine Graphics, we will have simultaneous access to many different points in the plane, and we will be able to join any two of these points with a straight line. These two abilities distinguish affine geometry from turtle geometry, and these two properties make Affine Graphics an extremely powerful tool for Computer Graphics.

To implement Affine Graphics, we are going to introduce a new language: CODO (COnnect the DOts). In CODO, as in LOGO, points and vectors are stored internally in terms of coordinates, but the CODO programmer, just like the LOGO programmer, has no direct access to these coordinates. Instead, the main tools in CODO for generating new points and lines are the affine transformations discussed in the Lecture 4.

2.1 The CODO Language. In CODO there are four types of objects: points, vectors, line segments, and affine transformations. Points and vectors are stored internally using affine coordinates; line segments are represented by their end points; affine transformations are stored as 3×3 matrices.

There are only three types of commands in CODO: commands that create geometry, commands that generate affine transformations, and a command to display geometry. Below is a complete description of the CODO language.

Geometry Creation

1. VECTOR
   - $VECTOR(P,Q)$ -- creates the vector from P to Q, usually denoted by $Q-P$
2. LINE
   - $LINE(P,Q)$ -- creates the line segment PQ
   - $LINE(P_1,\ldots,P_n)$ -- creates the line segments $P_1P_2,\ldots,P_{n-1}P_n$
3. TRANSFORM
   - $TRANSFORM(X,M)$ -- applies the affine transformation M to the object X, which may be a point, a vector, or a line segment, to create a new point, vector, or line segment. Note that, by definition,

     $$TRANSFORM\big(LINE(P,Q),M\big) = LINE\big(TRANSFORM(P,M),TRANSFORM(Q,M)\big)$$
   - $TRANSFORM\big((X_1,\ldots,X_n),(M_1,\ldots,M_p)\big)$ -- applies each affine transformation M_j to each object X_i to create the new collection of objects $\big\{TRANSFORM(X_i,M_j)\big\}$.

Affine Transformations

1. Vectors
   a. $ROT(\theta)$ -- rotates vectors by the angle θ
   b. $SCALE(s)$ -- scales the length of vectors by the scale factor s
2. Points
   a. $TRANS(v)$ -- translates points by the vector v
   b. $ROT(Q,\theta)$ -- rotates points about the fixed point Q by the angle θ
   c. $SCALE(Q,s)$ -- scales the distance of points from the fixed point Q by the factor s
   d. $SCALE(Q,w,s)$ -- scales the distance of points from the fixed point Q in the

direction w by the scale factor s

e. $AFFINETRANS(P_1,P_2,P_3;\ Q_1,Q_2,Q_3)$ -- creates the unique affine transformation that maps $P_k \to Q_k$, $k = 1,2,3$.

f. $COMPOSE(M_1,M_2)$ -- creates the composite transformation of M_1 and M_2. We often write $M_1 * M_2$ instead of $Compose(M_1,M_2)$.

g. *INVERT*(*M*) -- creates the inverse of the transformation *M*. We often write M^{-1} instead of *INVERT*(*M*)

Rendering

$DISPLAY(X_1,\ldots,X_n)$ -- displays the objects $X_1,\ldots,X_n$. The objects displayed may be only either points or lines.

Notice that CODO does not include a full suite of vector operations. In CODO we can perform scalar multiplication on vectors -- $c\,v = TRANSFORM\big(v, SCALE(c)\big)$ -- but there are no commands to compute $u \pm v$, $u \bullet v$, or $\det(u,v)$. The emphasis in CODO is on affine transformations, not on vector operations. Nevertheless, these vector operations could easily be added as a simple enhancement to the version of CODO presented here.

The commands to create geometry can be used to generate new points, lines, and vectors from existing points, lines, and vectors. But we still need some initial geometry to get started. We begin by fixing a single point *C* and a single vector *ivec*. In affine coordinates, we set $C = (0,0,1)$ and $ivec = (1,0,0)$. Thus *C* is located at the origin of the underlying coordinate system, and *ivec* is the unit vector pointing along the positive *x*-axis.

We can generate a unit vector along the *y*-axis by the command $TRANSFORM\big(ivec, ROT(\pi/2)\big)$, and we can generate a new point along the *x*-axis by the command $TRANSFORM\big(C, TRANS(ivec)\big)$. In the next section we will show how to generate lots of interesting shapes starting from this modest beginning.

2.2 Sample CODO Programs. We begin by writing a CODO program to generate the vertices of a polygon. To find the vertices of an *n*-sided polygon, we simply rotate one vertex around the center of the polygon *n* times through an angle of $2\pi / n$.

```
POLYVERTS (Center, Vertex, Number of Vertices)
    $\alpha = 2\pi / Number\ of\ Vertices$
    For $k = 0$, $Number\ of\ Vertices$
        $PolyVerts[k] = TRANSFORM\big(Vertex, ROT(Center, k\,\alpha)\big)$
```

The program STARVERTS to generate the vertices of a star is identical to the program POLYVERTS for generating the vertices of a polygon, except that the command $\alpha = 2\pi \,/\, Number\,of\,Vertices$ is replaced by the command $\alpha = 4\pi \,/\, Number\,of\,Vertices$.

The program *LINE(POLYVERTS(P, Q, n))* generates the edges of an *n*-sided polygon with center at *P* and a vertex at *Q*; the program DISPLAY(*LINE(POLYVERTS{P, Q, n)))* displays the edges of this polygon. To generate a circle, simply generate a polygon with a large number of sides; to construct an ellipse, apply non-uniform scaling to the circle (see Figure 1). Notice how in CODO non-uniform scaling facilitates the construction of the ellipse, a simple curve that is not so easy to generate in LOGO.

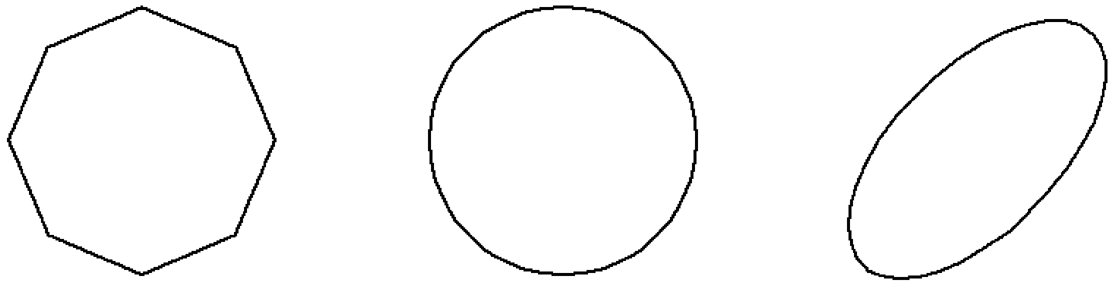

Figure 1: A regular octagon, a circle, and an ellipse. For the octagon and the circle, the center is at C and one vertex is located at $TRANSFORM\big(C, TRANS(ivec)\big)$. The ellipse is a generated by scaling the circle non-uniformly by 0.5 along the minor axis of the ellipse.

Rosettes and wheels are somewhat cumbersome to generate in LOGO, but rosettes and wheels are easy to create in CODO. The lines of the rosette (the diagonals of a polygon as well as the edges) are created by cycling through all pairs of vertices. Notice that in CODO, unlike LOGO, we do not need to know the lengths of the diagonals or the angles between adjacent diagonals to generate the diagonals of a polygon; all we need to know are the vertices of the polygon. Similarly, the spokes for a wheel are generated by cycling through the vertices of the polygon; we do not need to calculate the length of the circumradius. The CODO programs for the lines of a rosette and the spokes of a wheel are presented in Table 1.

```
$ROSETTE\ (V_1,\ldots,V_n)$
    $For\ i = 1,\ n-1$
        $For\ j = i+1,\ n$
            $Rosette[i,j] = LINE(V_i, V_j)$
```

```
$SPOKES\ (P, V_1,\ldots,V_n)$
    $FOR\ i = 1,\ n$
    $Spokes[i] = LINE(P, V_i)$
```

Table 1: The CODO programs for the lines of a rosette and the spokes of a wheel with vertices $V_1,\ldots,V_n$ and center at *P*.

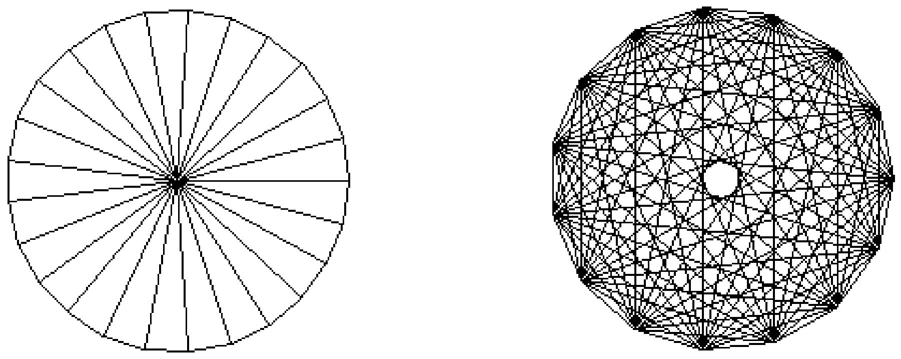

Figure 2: A wheel with 25 sides and a rosette with 15 sides. The CODO programs for the spokes of the wheel and the lines of the rosette are simple and are presented in Table 1.

We can also implement the operators SHIFT, SPIN, and SCALE inside of CODO.

```
SHIFT (X, w, n)
    For $i = 0, n$
        $Shift[i] = TRANSFORM\big(X, TRANS\big(TRANSFORM(w, SCALE(i))\big)\big)$
```

(Here n is the number of times we shift the object X in the direction w.)

```
SPIN (X, Q, α, n)
    For $i = 0, n$
        $Spin[i] = TRANSFORM\big(X, ROT(Q, i\alpha)\big)$
```

(Here n is the number of times we spin the object X around the point Q by the angle α.)

```
SCALE (X, Q, s, n)
    For $i = 0, n$
        $Scale[i] = TRANSFORM\left(X, SCALE\left(Q, s^i\right)\right)$
```

(Here n is the number of times we scale the object X about the point Q by the scale factor s.)

Examples of the SPIN operator applied to an ellipse are illustrated in Figure 3.

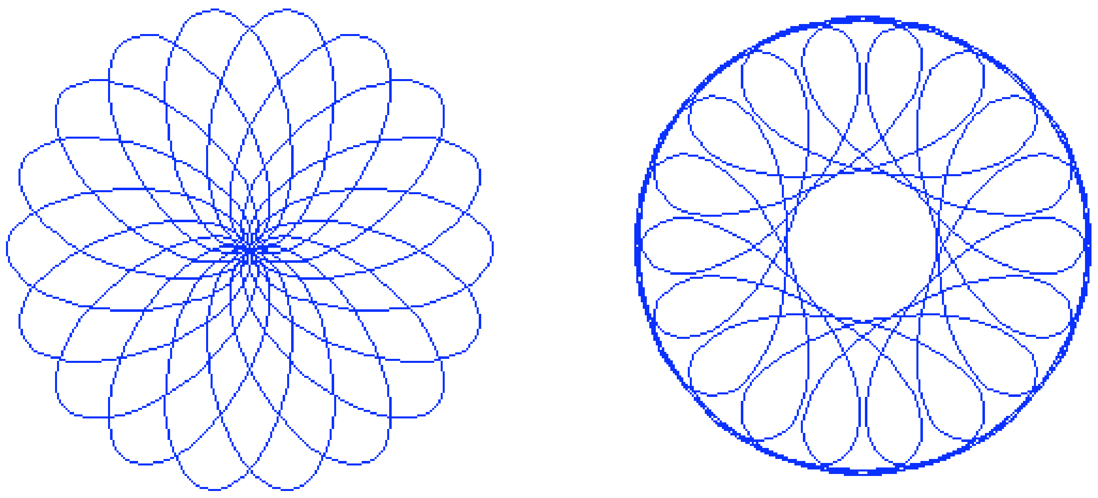

Figure 3: The spin operator applied to the ellipse: spinning the ellipse around an end point of the major axis (left), and spinning the ellipse around a point outside the ellipse along the minor axis (right).

Thus all the simple shapes we can draw using Turtle Graphics, we can easily recreate using Affine Graphics. In Figure 4, we illustrate some additional shapes that are easy to construct in CODO, but are less straightforward to generate in LOGO. In the next lecture we shall show how to generate fractals using the affine transformations in CODO.

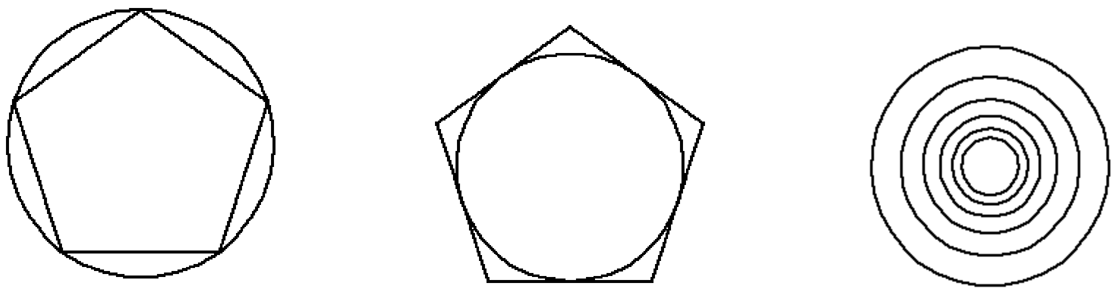

Figure 4. Three shapes involving circles that are easy to construct in CODO, but not so straightforward to generate in LOGO: a pentagon inscribed in a circle (left), a pentagon circumscribed about a circle (center), and a collection of concentric circles (right). We leave it as a simple exercise for the reader to write CODO programs to generate each of these simple shapes.

3. Summary

Turtle Graphics is a simple, but powerful, tool for rendering curves. Nevertheless, the lack of memory in LOGO and the absence in the turtle of any knowledge about the outside world make LOGO somewhat cumbersome for generating certain routine geometric shapes.

Affine Graphics provides an alternative to Turtle Graphics. CODO -- the connect the dots language embodying Affine Graphics -- has both memory and knowledge of how different geometric objects are related in the plane. These features facilitate simple CODO programs for generating shapes that often require more complicated LOGO programs.

Turtle geometry is a local, intrinsic geometry; affine geometry a global, extrinsic geometry. Turtle geometry is based on conformal maps; affine geometry on general affine transformations. Turtle geometry is implemented in rectangular coordinates; affine geometry in affine coordinates. These properties as well as other contrasting features of turtle geometry and affine geometry are summarized in Table 2.

Though coordinates are used to implement both turtle geometry and affine geometry, coordinates are not available to LOGO or CODO programmers. Geometry is generated in LOGO by manipulating the motion of the turtle using the commands FORWARD, MOVE, TURN and RESIZE. These commands represent the conformal transformations translation, rotation, and uniform scaling. Geometry is built in CODO by applying directly to already existing objects the affine transformations translation, rotation, and scaling (uniform and non-uniform), as well as general affine transformations generated from the image of three non-collinear points. The complete LOGO and CODO languages are listed in Table 3.

Turtle Geometry	*Affine Geometry*
Local	Global
Intrinsic	Extrinsic
Order Dependent	Order Independent
No Memory	Memory
State	Coordinate System
Rectangular Coordinates	Affine Coordinates
Conformal Transformations	Affine Transformations
Conformal Geometry	Affine Geometry

Table 2: Comparison of turtle geometry and affine geometry.

LOGO	*CODO*
FORWARD/MOVE	TRANS
TURN	ROT
RESIZE	SCALE
	AFFINETRANS
	COMPOSE
	INVERT
	VECTOR
	LINE
	TRANSFORM
	DISPLAY

Table 3: Comparison of the LOGO and CODO languages.

Exercises:

1. Write a CODO program SPIRALVERTS to generate the vertices of a spiral, where
   A = the angle between adjacent spiral arms
   S = the scale factor between adjacent spiral arms
   Q = the center of the spiral
   n = the number of spiral arms.

2. Write a CODO program to generate each of the shapes illustrated in Figure 3.

3. Write a CODO program to generate each of the shapes illustrated in Figure 4.

4. Write a CODO program to generate the vertices of the same star as the TRISTAR turtle program in Lecture 1, Exercise 5.

5. Write a CODO program to generate each of the shapes illustrated in Lecture 1, Figures 9.

6. Write a CODO program to generate each of the shapes illustrated in Lecture 1, Figure 10.

7. What are the rectangular coordinates of the point generated by the CODO command $TRANSFORM(C, M_1 * M_2)$, where

$$M_1 = TRANS\big(TRANSFORM\big(ivec, SCALE(x)\big)\big)$$

$$M_2 = TRANS\big(TRANSFORM\big(ivec,\ ROT(\pi/2) * SCALE(y)\big)\big).$$