Turtle Geometry in Computer Graphics and Computer Aided Design

Ron Goldman, Scott Schaefer, Tao Ju
Department of Computer Science
Rice University
6100 Main Street
Houston, Texas 77005-1892
rng@cs.rice.edu, sschaefe@rice.edu, jutao@cs.rice.edu

Abstract: LOGO is a programming language incorporating turtle graphics, originally devised for teaching computing to young children in elementary and middle schools. Here we advocate the use of LOGO to help introduce some of the basic concepts of computer graphics and computer aided design to undergraduate and graduate students in colleges and universities. We shall show how to motivate affine coordinates and affine transformations, fractal curves and iterated function systems, relaxation methods and subdivision schemes from elementary notions in turtle geometry and turtle programming.

1. Introduction

Many different approaches have been suggested for explaining to students the basic mathematical concepts underlying computer graphics and computer aided design, ranging from baseball arithmetic [Goldman, 2001] to projective geometry [Riesenfeld, 1981], from Groebner bases [Hoffmann, 1989] to polar forms [Ramshaw, 1989], from tensors and tensor diagrams [Blinn, 2002] to geometric products and Clifford algebras [Dorst and Mann, 2002]. In this paper we advocate a very different approach, based on turtle programming (LOGO) and turtle geometry.

LOGO and turtle graphics have been popularized as a paradigm for teaching programming to young children in elementary and middle schools [Papert, 1980], but LOGO has also been used successfully as a first language to teach programming to freshmen at various colleges and universities [Harvey, 1985-1987]. Yet despite these achievements, turtle programming and turtle geometry have been largely ignored in traditional college and university courses on computer graphics and computer aided design.

The purpose of this paper is to promote the turtle as an effective way of introducing many of the fundamental concepts that underlie both of these disciplines. After briefly reviewing some of the basic notions of turtle programming and turtle graphics in Sections 2 and 4, we show how the turtle can be to used to motivate affine coordinates and affine transformations (Section 5), fractal curves and iterated function systems (Section 6), and fixed point methods and relaxation techniques (Section 7). In Section 8, we discuss a fractal algorithm for generating Bezier curves and surfaces

based on recursive subdivision, and we mention as well a way of extending the power of the turtle to render Bezier curves. We summarize our overall approach in Section 9, where we conclude with some possible future extensions of our turtle techniques.

2. Turtle Geometry and Turtle Programming

Turtle geometry is a local, coordinate free, version of computational geometry. Turtle geometry has been used to study many diverse subjects from simple polygons to complex fractals, from the Euler characteristic and the formula of Gauss-Bonnet to curved space-time and Einstein's general theory of relativity [Abelson and diSessa, 1986]. Turtle programs provide a graphical interpretation of L-systems, special grammars with specific kinds of production rules [Prusinkiewicz, 1986]. Turtle representations of planar polygons have been applied to morph polygonal shapes [Sederberg et al, 1993]. Here we are going to show how to use turtle geometry and turtle programming to help introduce college students to some standard concepts in computer graphics and computer aided design.

To study turtle geometry, we introduce a virtual turtle. The virtual turtle is a simple creature. It knows only where it is, in which direction it is facing, and its step size; it obeys solely simple commands to change either its location, or its heading, or its notion of scale.

Consider such a turtle living on a plane. Its location can be represented by a point P given by a pair of coordinates (p_1, p_2); similarly its heading can be represented by a vector w given by another pair of coordinates (w_1, w_2). The step size of the turtle is simply the length of the vector w. The pair (P, w) is called the *turtle's state*. Although internally the computer stores the coordinates (p_1, p_2) and (w_1, w_2), the turtle (and the turtle programmer) has no access to these global coordinates.

The turtle responds to four basic commands: FORWARD, MOVE, TURN, and RESIZE. These commands affect the turtle in the following ways:

- FORWARD *d*: The turtle moves forward *d* steps in the direction of its current heading, and draws a line from its initial position to its final position.
- MOVE *d*: Same as FORWARD *d* without drawing a line.
- TURN *a*: The turtle changes it heading by rotating in the plane counterclockwise from its current heading by the angle *a*.
- RESIZE *s*: Changes the turtle step size by the factor *s*.

Notice that all four commands are local, coordinate free instructions to the turtle.

The turtle state (P, w) is a complete description of what the turtle knows, and the four turtle commands FORWARD, MOVE, TURN, RESIZE are the only way that a programmer can communicate with the turtle. Yet from this simple setup much can be learned.

3. Affine Geometry

Affine geometry is the study of the geometric properties of shapes that are invariant under affine transformations. Affine transformations are precisely those maps that are combinations of translations, rotations, shearings, and scalings. Affine geometry is one of the foundations of computer graphics and computer aided design, since affine transformations are fundamental to repositioning and resizing objects in space.

Unfortunately, most undergraduates and even many graduate students are not so familiar with the fundamental concepts of affine geometry as one might suppose. Even the very notion of an affine space is often unfamiliar to many students. Linear spaces -- spaces of vectors -- are generally known to undergraduates from courses on linear algebra, but affine spaces contain points as well as vectors [Goldman, 2002b, 2003]. Points have a fixed position, but no direction or length; vectors have direction and length, but no fixed position. The distinction between points and vectors in affine space is fundamental, but this distinction is often obscured in the students' minds, where points and vectors in the plane are both represented by pairs of coordinates. The distinction between points and vectors though important in practice, appears to many students, whose perspective is grounded in coordinate geometry, as highly artificial.

The turtle can help students to grasp this distinction. The turtle knows three things: it's position, its heading, and its scale. The turtle's position is represented by a point in the affine plane; the turtles heading and scale are stored in a 2-dimensional vector. The point has a fixed position, but no direction or length; the vector has direction and length, but no fixed position. This distinction between points and vectors is natural in coordinate free turtle geometry, whereas it seems quite artificial to students familiar only with analytic geometry based on rectangular coordinates.

Implementing the turtle commands on a computer accentuates this distinction between points and vectors because each command affects points and vectors differently. Let (P, w) represent the current state of the turtle. The effect of each of the four turtle commands on the turtle's state is summarized in Table 1.

FORWARD d:	$P \rightarrow P + dw$	$w \rightarrow w$
MOVE d:	$P \rightarrow P + dw$	$w \rightarrow w$
TURN a:	$P \rightarrow P$	$w_1 \rightarrow w_1 \cos(a) - w_2 \sin(a)$ $w_2 \rightarrow w_1 \sin(a) + w_2 \cos(a)$
RESIZE s:	$P \rightarrow P$	$w \rightarrow sw$

Table 1: How the four turtle commands affect the turtle's state (P, w).

Thus we see that translations -- FORWARD and MOVE -- alter points but not vectors, reemphasizing that points have position, but vectors do not. Similarly, scaling -- RESIZE -- alters vectors, but not points, emphasizing that vectors have length, but points do not.

The TURN command is also important because this command is the first contact of many students with rotation. Since students typically are familiar with linear algebra and matrix multiplication, it is natural for them to implement this transformation using matrices:

$$(w_1 \;\; w_2) \rightarrow (w_1 \;\; w_2) * \begin{pmatrix} \cos(a) & \sin(a) \\ -\sin(a) & \cos(a) \end{pmatrix}.$$

This matrix will reappear when the students study 3-dimensional computer graphics; the matrix

$$\begin{pmatrix} \cos(a) & \sin(a) & 0 \\ -\sin(a) & \cos(a) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

represents rotation around the *z*-axis. Analogous matrices can be used to rotate objects around the *x* and *y* axes.

4. Turtle Graphics

Once the four turtle commands are implemented, students can start to write turtle programs to generate interesting shapes. The simplest programs just iterate various combinations of the FORWARD, TURN, and RESIZE commands. For example, by iterating the FORWARD and TURN commands, students can create polygons and stars (see Table 2). Circles can be generated by building polygons with lots of sides. Iterating FORWARD and RESIZE, the turtle walks along a straight line, and iterating TURN and RESIZE, the turtle simply spins in place. But by iterating FORWARD, TURN, and RESIZE, the turtle can generate spiral curves (see Figure 1).

```
POLYGON N                STAR N                   SPIRAL N, A, S
    REPEAT N TIMES           REPEAT N TIMES           REPEAT N TIMES
        FORWARD 1                FORWARD 1                FORWARD 1
        TURN 2π / N              TURN 4π / N              TURN A
                                                          RESIZE S
```

Table 2: Simple turtle programs for generating polygons, stars, and spirals by iterating the basic turtle commands. For the spiral program, *A* is a fixed angle and *S* is a fixed scalar.

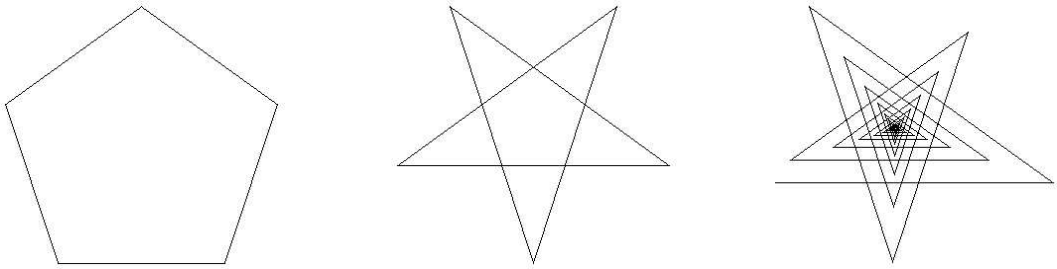

Figure 1: A pentagon, a five pointed star, and a spiral generated by the programs in Table 2. Here the spiral angle $A = 2\pi / 5$ and the scale factor for the spiral is $S = 9/10$.

With a bit more ingenuity (and with some help from the law of cosines), students can generate more complicated shapes such as the wheel and the rosette (see Figure 2).

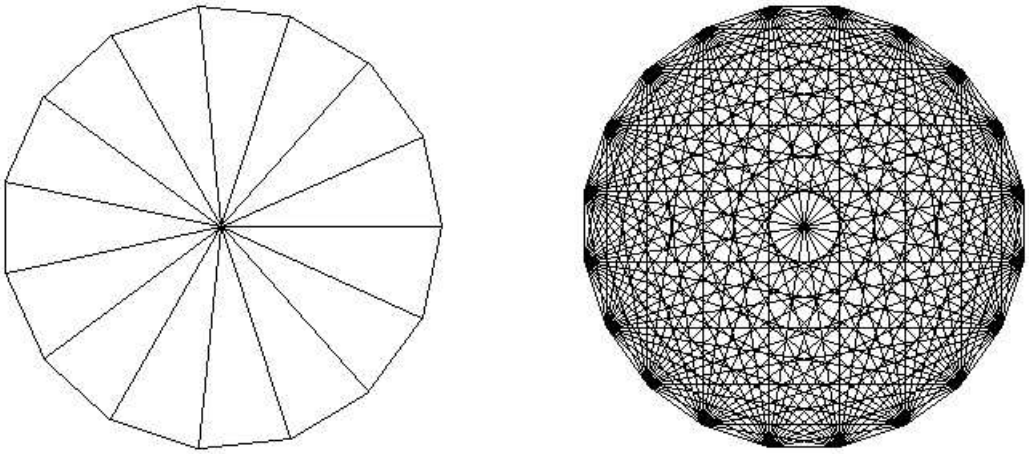

Figure 2: The wheel and the rosette. The wheel is simply a polygon together with the lines connecting its vertices to its center. The rosette is a polygon, together with all its diagonals -- that is, with the lines joining every vertex to every other vertex. The wheel displayed here has 15 sides and the rosette has 20 sides. We leave it as a challenge to the reader to develop simple turtle programs that generate these shapes.

But by far the most interesting and exciting shapes that students can generate using turtle graphics are fractals. Fractals require more than iteration; they require recursion. There is no single, commonly accepted, definition of the term *fractal*, but from the point of view of the turtle, *fractals are simply recursion made visible.* The study of fractals can help students to understand both computer graphics and recursive programs.

Let's try to write a turtle program SIERP to create the Sierpinski gasket displayed at the far right of Figure 3. How should we proceed? Notice that the Sierpinski gasket is made up of three smaller Sierpinski gaskets, each one half the size of the original gasket. This observation suggests that the turtle should adopt the following strategy to generate the Sierpinski gasket:

a. generate a Sierpinski gasket one half the size of the big gasket at one of the vertices of the outer triangle;
b. move to the next vertex of the outer triangle and turn to face the subsequent vertex;
c. repeat step a and b two more times.

This plan won't quite work because the recursion has no case at which to bottom out; to finish off the program, we need to include a base case. For the Sierpinski gasket, a natural base case is simply to draw a triangle. This approach leads to the recursive turtle program for the Sierpinski gasket coded in Table 3.

```
SIERP  LEVEL
    BASE CASE:  IF LEVEL = 0,  POLYGON 3
    RECURSION:
        REPEAT 3 TIMES
            RESIZE 0.5
            SIERP  LEVEL – 1
            RESIZE 2
            MOVE 1
            TURN 2π/3
```

Table 3: A recursive turtle program for generating the Sierpinski gasket. The base case draws a triangle, a polygon with three sides.

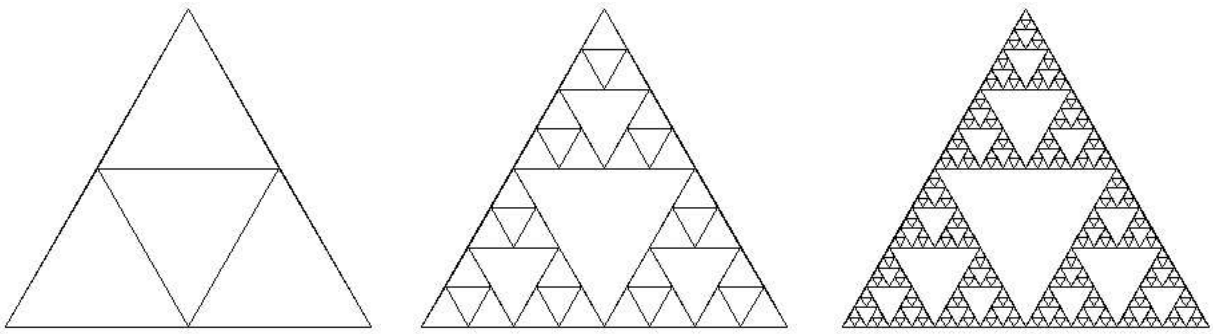

Figure 3: The Sierpinski gasket generated from the turtle program in Table 3. Levels 1, 3 and 5 of the recursion are illustrated here.

The program in Table 3 does indeed generate the Sierpinski gasket, and by a similar analysis one can write turtle programs to generate lots of other engaging fractals, including Koch snowflakes, fractal trees, and space filling curves [Abelson and diSessa, 1986].

How robust is this turtle program for the Sierpinski gasket? What happens if we make a simple mistake in the base case and type POLYGON 4 instead of POLYGON 3 -- that is, what happens if we start with a square in the base case instead of a triangle? Remarkably, the answer is that though the first few levels of recursion appear to generate a different figure, in the limit the turtle generates the same fractal gasket (see Figure 4)! Why does this happen? To answer this question as well as many other questions about fractals generated by recursive turtle programs, we first need to deepen our students' understanding of affine transformations.

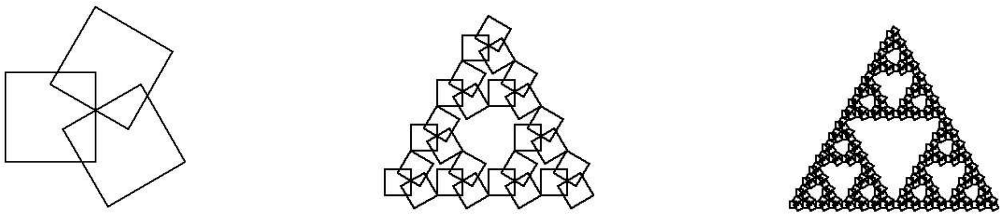

Figure 4: The Sierpinski gasket generated by drawing a square instead of a triangle in the base case: levels 1, 3 and 5 of the recursion are illustrated here. Compare to Figure 3.

5. Affine Coordinates and Affine Transformations

Let us consider for now only oriented conformal affine transformations -- that is, affine transformations that preserve angle and orientation. These transformations are precisely those maps that are composites of translations, rotations, and uniform scalings -- the transformations our students encounter in turtle graphics. In turtle geometry we apply translation (FORWARD AND MOVE) to points, but we apply rotation and scaling (TURN AND RESIZE) to vectors. In affine geometry, we shall apply all of these affine transformations directly to points.

Students know what it means to translate a point, but what does it mean to rotate or scale a point? Earlier, we insisted that scaling applies only to vectors, never to points because points have no fixed length. Similarly, points have no fixed direction, so what does it mean to rotate a point? If, however, we fix a point in the plane (not necessarily at the origin of the coordinate system), then we could rotate points about this fixed point. Similarly, we could scale the distance from any point to this fixed point. In effect, if we fix a point Q, then what we mean by rotating or scaling a point P about the point Q is to rotate or scale the vector $P - Q$, and then to add the resulting vector to Q.

Since students already know formulas for translation, rotation, and uniform scaling from their study of turtle geometry, they can easily summarize the effects of these transformations as in Table 4.

Translate(w,d):	$P \to P + dw$
Rotate(Q,a):	$p_1 \to q_1 + (p_1 - q_1)\cos(a) - (p_2 - q_2)\sin(a)$ $p_2 \to q_2 + (p_1 - q_1)\sin(a) + (p_2 - q_2)\cos(a)$
Scale(Q,s):	$P \to Q + s(P - Q) = sP + (1 - s)Q$

Table 4: How the three basic conformal affine transformations affect points. Here $Q = (q_1, q_2)$ is a fixed point, *w* is a vector, *d* is a scalar indicating distance, and *s* is a scale factor. To rotate or scale the point *P* about the point *Q*, we rotate or scale the vector $P - Q$ and add the result to *Q*.

In turtle graphics, students use matrices to simplify their implementation of the TURN command. Similarly, they could simplify the transformation *Rotate*(Q,a) by representing this map in matrix form. For uniformity, we encourage students to rewrite all three of these basic conformal affine transformations using matrix notation as in Table 5.

Translate(w,d):	$P \to \begin{pmatrix} p_1 & p_2 \end{pmatrix} * \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} + d\begin{pmatrix} w_1 & w_2 \end{pmatrix}$
Rotate(Q,a):	$P \to \begin{pmatrix} p_1 & p_2 \end{pmatrix} * \begin{pmatrix} \cos(a) & \sin(a) \\ -\sin(a) & \cos(a) \end{pmatrix} + \begin{pmatrix} q_1 & q_2 \end{pmatrix} * \begin{pmatrix} 1-\cos(a) & -\sin(a) \\ \sin(a) & 1-\cos(a) \end{pmatrix}$
Scale(Q,s):	$P \to \begin{pmatrix} p_1 & p_2 \end{pmatrix} * \begin{pmatrix} s & 0 \\ 0 & s \end{pmatrix} + (1-s)\begin{pmatrix} q_1 & q_2 \end{pmatrix}$

Table 5: The three basic conformal affine transformations expressed in matrix notation.

Now all three of the basic conformal affine transformations have the same form: $P \to P * M + R$ for different choices of *M* and *R*. Often students need to compose two such affine transformations. In linear algebra, students can compose linear transformations on vectors by representing the transformations as matrices and then applying matrix multiplication. But here the constant additive term $(+R)$ gets in the way of composing affine transformations by matrix multiplication.

Affine coordinates are a device for overcoming this shortcoming [Murray *et al*, 1994]. Although we have insisted that points and vectors are distinct types, there is still no way students can distinguish between points and vectors simply by looking at their rectangular coordinates.

Affine coordinates introduce a distinction and also allow us to represent affine transformations by matrix multiplication. In affine coordinates, in addition to the two rectangular coordinates, we affix a third affine coordinate: this third coordinate is 1 for points and 0 for vectors[1]. Thus we have:

$P = (p_1, p_2, 1)$ (points)

$w = (w_1, w_2, 0)$ (vectors).

Using this notation, we can rewrite translation, rotation, and scaling in terms of matrix multiplication (see Table 6).

Translate(w,d): $P \to (p_1 \;\; p_2 \;\; 1) * \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ dw_1 & dw_2 & 1 \end{pmatrix}$

Rotate(Q,a): $P \to (p_1 \;\; p_2 \;\; 1) * \begin{pmatrix} \cos(a) & \sin(a) & 0 \\ -\sin(a) & \cos(a) & 0 \\ q_1(1-\cos(a)) + q_2 \sin(a) & q_2(1-\cos(a)) - q_1 \sin(a) & 1 \end{pmatrix}$

Scale(Q,s): $P \to (p_1 \;\; p_2 \;\; 1) * \begin{pmatrix} s & 0 & 0 \\ 0 & s & 0 \\ (1-s)q_1 & (1-s)q_2 & 1 \end{pmatrix}$

Table 6: The three basic conformal affine transformations expressed as matrix multiplication using affine coordinates. These matrices are valid for any fixed point Q, independent of our choice of the origin for the coordinate system.

Using these matrices, composition of conformal affine transformations is given by matrix multiplication. More generally, using affine coordinates, arbitrary affine transformations can be represented by 3×3 matrices, where the third column is $(0 \;\; 0 \;\; 1)^T$. Notice too that if we replace points by vectors, then these transformation matrices are still valid, since the zero in the third coordinate of each vector cancels the translation in the third row of the transformation matrices. Thus students can apply these matrices to implement the four basic commands FORWARD, MOVE, TURN, and RESIZE of turtle graphics (see Table 7).

$FORWARD\ d \leftrightarrow Translate(w,d)$	$MOVE\ d \leftrightarrow Translate(w,d)$
$TURN\ a \leftrightarrow Rotate(P,a)$	$RESIZE\ s \leftrightarrow Scale(P,s)$

Table 7: The four basic turtle commands expressed as conformal affine transformations. The current turtle state is (P,w).

[1] Vectors in affine space should not be confused with the points at infinity of projective space, and affine coordinates should not be mistaken for homogeneous coordinates; see [Goldman 2002b, 2003].

6. Recursive Turtle Programs and Iterated Affine Transformations

Iterated functions systems are another approach to generating fractals [Barnsley, 1993]. We shall now show that by analyzing the geometry generated by recursive turtle programs, students are led naturally to the study of iterated affine transformations. First, however, we need to formalize precisely what we mean by a recursive turtle program. The simplest recursive turtle programs have the following form:

Recursive Turtle Program (RTP)

- *Base Case:*
  Turtle Program(T_0*)*
- *Recursion:*
  Turtle Program(T_1), *Turtle Recursion*
  ⋮
  Turtle Program(T_m), *Turtle Recursion*
  Turtle Program(T_{m+1})

Here $T_0, T_1, \ldots, T_{m+1}$ are simple turtle programs, consisting of finite sequences of FORWARD, MOVE, TURN, and RESIZE commands. For now, we shall also make two additional assumptions:

- the FORWARD command appears only in the base case;
- in both the base case and in the recursion, the turtle starts and ends in the same state.

The first assumption is there only for simplicity; later on, we shall dispense with this assumption. The second assumption (or some similar assumption) is required to assure that the turtle's state, and hence too the geometry generated by the recursive turtle program, converges.

The keys to understanding the fractal geometry generated by an RTP are the following three observations:

1. The only effect of the turtle programs $T_1, \ldots, T_{m+1}$ in the recursion is to change the state of the turtle.
2. For any two turtle states $S_1 = (P_1, w_1)$ and $S_2 = (P_2, w_2)$, there is an oriented conformal affine transformation *T* that maps S_1 to S_2.
3. Suppose that T_1, T_2 are two turtle programs that differ only by their initial states S_1, S_2, and let G_1, G_2 be the geometry generated by T_1, T_2 If $S_2 = T(S_1)$, then $G_2 = T(G_1)$.

The first observation follows because, by assumption, there are no FORWARD commands in the turtle programs $T_1, \ldots, T_{m+1}$. Therefore these programs affect only the state of the turtle, but not the geometry drawn by the turtle. The second observation can be proved by observing that to get from

$S_1 = (P_1, w_1)$ to $S_2 = (P_2, w_2)$, we can first translate P_1 to P_2, then rotate w_1 to align with the direction of w_2, and finally scale w_1 to the size of w_2. Each of these transformations is an oriented conformal affine map, so their composite is also an oriented conformal affine transformation. The third observation follows from the fact that the turtle commands are local commands. Thus, if we translate, rotate, or scale the turtle's initial state, then we will translate, rotate, or scale the geometry generated by the turtle program.

Suppose that G_0 is the geometry generated by the turtle program T_0 in the base case, and that G_k is the geometry generated after the *kth* level of recursion for $k \geq 1$. Let S_0 denote the initial state of the turtle, and let S_i denote the state to which the turtle program $T_1,\ldots,T_i$ maps S_0. If R_i is the oriented conformal affine transformation that maps S_0 to S_i, then by our third observation

$$G_k = R_1(G_{k-1}) \cup \cdots \cup R_m(G_{k-1}). \qquad (1)$$

By definition, the fractal generated by the recursive turtle program is

$$G = Lim_{k\to\infty} G_k. \qquad (2)$$

Thus Equations (1) and (2) encapsulate what we know about the geometry of the fractal *G* generated by the recursive turtle program.

Let $T = \{R_1,\ldots,R_m\}$, and let *H* be any subset of the plane. If we define

$$T(H) = R_1(H) \cup \cdots \cup R_m(H), \qquad (3)$$

then Equation (1) can be rewritten more compactly as

$$G_k = T(G_{k-1}).$$

Thus the fractal *G* can be generated in the limit by iterating the transformation *T* starting with the geometry G_0. The collection of affine transformations $T = \{R_1,\ldots,R_m\}$ together with the set G_0 is called an *iterated affine transformation (IAT)*, a special case of an iterated function system where all the transformations are affine transformations.

Fractals can be generated by iterating contractive affine transformations. A transformation *R* is called *contractive* if for any two points *P* and *Q*, there is a fixed constant *s* independent of *P* and *Q* such that

$$dist\big(R(P), R(Q)\big) \leq s * dist(P, Q) \qquad 0 < s < 1.$$

The simplest contractive maps are *Scale*(Q, s), where $0 < |s| < 1$. More generally, if *R* is an oriented conformal affine transformation, then *R* is contractive if and only if $0 < \det(R) < 1$.

Let $T = \{R_1,\ldots,R_m\}$ be a set of affine transformations, and let G_0 be a compact (i.e. closed and bounded) collection of points and lines. If the maps in *T* are contractive, then we will show in Section 7 that:

i. There exists a unique compact set G such that $T(G) = G$.
   The set G is called a *fixed point* of T.
ii. The sets $G_k = T(G_{k-1})$ converge to G (in the Hausdorff metric on compact sets).
   The convergence is independent of the choice of the initial set G_0!

To fully understand the second statement, we would need to consider the space of all compact subsets of the plane together with the Hausdorff metric. We shall not go into the details here; see [Barnsley, 1993] for a full discussion. It suffices to tell our students that there is a function that measures the distance between any two compact sets, and using this measure of distance $G_k \to G$. The fixed set G is called the *fractal* generated by the transformations T, and the algorithm that builds the sequence $G_0, G_1, \ldots$ is called the *deterministic algorithm* for generating G. (There is also a *randomized algorithm* for generating G -- see [Barnsley, 1993].)

Equations (1)-(3) provide the connection between fractals generated by recursive turtle programs and fractals generated by the deterministic algorithm for iterated affine transformations. If we apply the affine transformations $T = \{R_1, \ldots, R_m\}$ to the geometry G_0, then by Equations (1)-(3) this IAT will generate exactly the same geometry at the *kth* iteration of the deterministic algorithm that the RTP generates at the *kth* level of the recursion. Since for an IAT the convergence is independent of the choice of the initial set G_0, the same must be true for turtle programs -- that is, the geometry generated by an RTP is independent of the turtle program in the base case, provided that in the base case (and in the recursion), the turtle starts and ends in the same state. Notice that if the turtle changes state either in the base case or during the recursion, then Equation (1) may no longer hold. Thus recursive turtle programs motivate the study of iterated affine transformations, and iterated affine transformations shed light on the behavior of recursive turtle programs.

Given any specific recursive turtle program, the transformations $T = \{R_1, \ldots, R_m\}$ of the corresponding IAT are easy to find. Using Table 7, students can simply write down the transformation matrices for each turtle command in each of the turtle programs $T_1, \ldots, T_m$. If $R_{k,1}, \ldots, R_{k,n_k}$ are the matrices corresponding to the turtle commands appearing in the turtle program T_k, $1 \le k \le m$, then

$$R_1 = R_{1,1} \cdots R_{1,n_1}$$
$$R_k = R_{k-1} R_{k,1} \cdots R_{k,n_k}.$$

For example, consider the turtle program in Table 3 for the Sierpinski gasket. Suppose that the outer triangle of the gasket has vertices at the points P_1, P_2, P_3 and edges equal in length and parallel to the vectors w_1, w_2, w_3. If the turtle starts in state (P_1, w_1), then

$$R_1 = Scale\left(P_1, \frac{1}{2}\right)$$

$$R_2 = R_1 * Scale(P_1, 2) * Translate(w_1, 1) * Rotate\left(P_2, \frac{2\pi}{3}\right) * Scale\left(P_2, \frac{1}{2}\right)$$

$$= Translate(w_1, 1) * Rotate\left(P_2, \frac{2\pi}{3}\right) * Scale\left(P_2, \frac{1}{2}\right)$$

$$R_3 = R_2 * Scale(P_2, 2) * Translate(w_2, 1) * Rotate\left(P_3, \frac{2\pi}{3}\right) * Scale\left(P_3, \frac{1}{2}\right)$$

$$= Translate(w_1, 1) * Rotate\left(P_2, \frac{2\pi}{3}\right) * Translate(w_2, 1) * Rotate\left(P_3, \frac{2\pi}{3}\right) * Scale\left(P_3, \frac{1}{2}\right)$$

Notice that the transformation R_k maps the Sierpinski gasket S into one of the smaller Sierpinski gaskets at the vertex P_k, $k = 1,2,3$. Therefore if $T = \{R_1, R_2, R_3\}$, then $T(S) = S$, so the Sierpinski gasket S is indeed a fixed point of these transformations. Figure 5 illustrates the Sierpinski gasket generated by the deterministic algorithm for the transformations $T = \{R_1, R_2, R_3\}$ using two different base cases. An even easier general method for extracting the transformations $T = \{R_1, R_2, R_3\}$ for the IAT from the recursive turtle program is provided in [Ju *et al*, 2003].

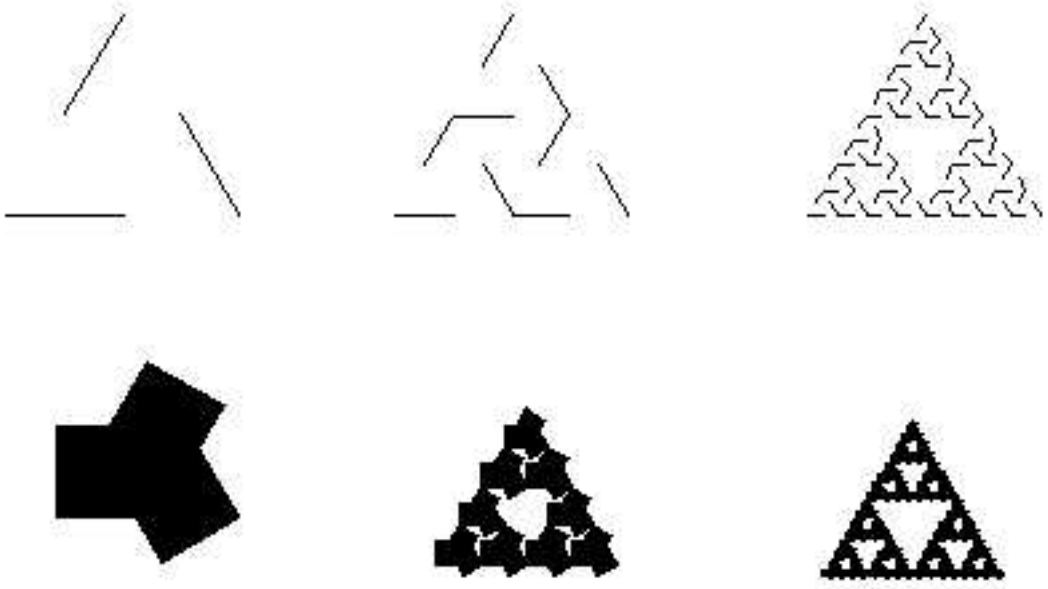

Figure 5: The Sierpinski gasket generated by the IAT $T = \{R_1, R_2, R_3\}$ constructed directly from our RTP for the Sierpinski gasket. On the top row the base case is a horizontal line, and we display from left to right 1 , 2, 4 levels of iteration. On the bottom row the base case is a solid square, and we display from left to right 1, 3, 5 levels of iteration.

This result that every RTP corresponds to an IAT can be extended in many ways. First, we can remove the restriction that the FORWARD command does not appear in any of the turtle programs $T_1,\ldots,T_{m+1}$ in the recursion. If we remove this restriction, then the programs $T_1,\ldots,T_{m+1}$ will generate some geometry *C*, and Equation (1) will be replaced by

$$G_k = R_1(G_{k-1}) \cup \cdots \cup R_m(G_{k-1}) \cup C. \qquad (4)$$

The set C is called a *condensation set* [Barnsley, 1993] for the corresponding IAT. Hence each such RTP corresponds to an IAT with a condensation set.

We can also permit the turtle to change state in the base case and during the recursion, provided that in both cases the turtle starts in the same state S_0 and finishes in the same state S_f. If R'_k is the oriented conformal affine transformation that maps S_0 to S'_k, where S'_k is the state of the turtle after executing $T_1,\ldots,T_k$ interspersed with $k-1$ recursive calls, then Equation (1) will be replaced by

$$G_k = R'_1(G_{k-1}) \cup \cdots \cup R'_m(G_{k-1}). \qquad (5)$$

Hence once again the RTP corresponds to an IAT.

The converse is also true. Every IAT consisting of oriented conformal maps corresponds to an RTP. Students can construct this RTP in the following fashion. Let $T = \{R_1,\ldots,R_m\}$ be the transformations of the IAT. Every oriented conformal map can be decomposed into the product of a rotation, a scale, and a translation. These maps, in turn, correspond to turtle commands. In fact, suppose that R_k transforms $(P_{k-1}, w_{k-1}) \to (P_k, w_k)$. Then there is a turtle program T_k that maps $(P_{k-1}, w_{k-1}) \to (P_k, w_k)$. First TURN the turtle to face towards P_k, then MOVE the distance from P_{k-1} to P_k, then TURN w_{k-1} to align with the direction of w_k, and finally RESIZE w_{k-1} to the size of w_k. Now the geometry generated by an RTP with the turtle programs $T_1,\ldots,T_m$ in the recursion will satisfy the same recurrence relation as the geometry generated by the IAT with the transformations $T = \{R_1,\ldots,R_m\}$, provided that both in the base case and in the recursion the turtle starts and ends in the same state. We can easily enforce this constraint in the base case; in the recursion, we add the program T_{m+1}, whose sole purpose is to return the turtle to its initial state.

This result can be generalized even to an IAT consisting of arbitrary nonsingular affine transformations, but to do so we need to reconfigure the turtle. In addition to the heading vector *w*, let the turtle also carry a left hand vector *v*. Alter the TURN and RESIZE commands so that the turtle can rotate and scale the vectors *v* and *w* independently -- that is, the turtle now obeys the commands $TURN(a_1, a_2)$ and $RESIZE(s_1, s_2)$ that rotate and scale the vectors *v* and *w* independently. The same arguments that we just used for ordinary turtles and IATs consisting of oriented conformal maps can be applied to demonstrate the equivalence between an RTP for this reconfigured turtle and an IAT with arbitrary nonsingular affine transformations [Ju *et al*, 2003].

As a final generalization, students could consider mutually recursive turtle programs -- that is, two or more recursive turtle programs that call one another. There is a version of the IAT that also permits mutual iteration, and these more powerful versions of RTP and IAT are once again equivalent to one another [Ju *et al*, 2003].

7. The Trivial Fixed Point Theorem

The fractal generated by a recursive turtle program is independent of the geometry generated by the turtle program in the base case because every RTP is equivalent to an IAT, and the fractal generated by an IAT is known to be independent of the base case. But this result begs the question: why is the fractal generated by an IAT independent of the base case?

The reduction of fractals generated by recursive turtle programs to fractals generated by the deterministic algorithm for iterated affine transformations raises several more questions:

- Why does iteration necessarily lead to fixed points?
- Why are the fixed points unique?
- Why must the iteration converge?

We shall now answer each of these questions in turn by a sequence of three easy lemmas. These answers motivate the study of some simple analysis, leading students directly to this section's main result: the trivial fixed point theorem for complete metric spaces.

We begin by answering the first question by showing that for continuous functions, when iteration converges, iteration necessarily leads to a fixed point.

Lemma 1: *Let T be a continuous function, and for all* $n \geq 0$ *let* $P_{n+1} = T(P_n)$. *If* $P = Lim_{n\to\infty} P_n$ *exists, then P is a fixed point of T.*

Proof: Since *T* is a continuous map,

$$T(P) = T\left(Lim_{n\to\infty} P_n\right) = Lim_{n\to\infty} T(P_n) = Lim_{n\to\infty} P_{n+1} = P\,.$$

Since contractive maps are necessarily continuous, we conclude from Lemma 1 that iterating a contractive map will lead to a fixed point whenever the iteration converges. Next we show that for a contractive map, this fixed point is unique.

Lemma 2: *Let T be a contractive map. Then T can have at most one fixed point.*

Proof: If *P* and *Q* are both fixed points of *T*, then $T(P) = P$ and $T(Q) = Q$. Therefore

$$Dist\left(T(P), T(Q)\right) = Dist(P, Q)\,.$$

contradicting the assumption that *T* is a contractive map.

From Lemmas 1 and 2, we conclude that if we iterate a contractive map T and the iteration converges, then it must converge to the unique fixed point of T. But this result still leaves open the question of why iteration of contractive maps always converges. To establish this result, students need to understand the notion of a cauchy sequence.

A sequence $\{S_n\}$ is called a *cauchy sequence* if for any $\varepsilon > 0$ there is an integer N such that

$$m,n > N \Rightarrow Dist(S_n,S_m) < \varepsilon.$$

In simple words, a sequence is *cauchy* if the elements of the sequence get closer and closer as their indices get larger and larger. The following result shows that iterating a contractive map generates a cauchy sequence.

Lemma 3: *Suppose that T is a contractive map, and for all* $n \geq 0$ *let* $P_{n+1} = T(P_n)$. *Then* $\{P_n\}$ *is a cauchy sequence for any choice of* P_0.

Proof: Since T is a contractive map, there is a constant $0 < s < 1$ such that

$$\begin{aligned} Dist(P_{n+1},P_n) &= Dist(T(P_n),T(P_{n-1})) \\ &\leq sDist(P_n,P_{n-1}) = sDist(T(P_{n-1}),T(P_{n-2})) \\ &\quad \vdots \\ &\leq s^n Dist(P_1,P_0). \end{aligned}$$

Therefore for n sufficiently large,

$$\begin{aligned} Dist(P_{n+m+1},P_n) &\leq Dist(P_{n+m+1},P_{n+m}) + \cdots + Dist(P_{n+1},P_n) \\ &\leq (s^{n+m} + \cdots + s^n) Dist(P_1,P_0) \\ &\leq s^n \frac{Dist(P_1,P_0)}{1-s} \\ &< \varepsilon. \end{aligned}$$

In a complete metric space, cauchy sequences always converge. A metric space (X,d) is a space X together with a distance function d. The function d must satisfy the usual properties of distance; in particular, d must satisfy the triangular inequality:

$$d(x,z) \leq d(x,y) + d(y,z).$$

A metric space is said to be *complete* if every cauchy sequence converges. We can now state the main result of this section, which is well known in analysis [Barnsley, 1993].

Trivial Fixed Point Theorem

Suppose that T is a contractive map on a complete metric space, and for all $n \geq 0$ *let* $P_{n+1} = T(P_n)$. *Then* $\{P_n\}$ *converges to the unique fixed point of T for any choice of* P_0.

Proof: The sequence $\{P_n\}$ is cauchy by Lemma 3. Therefore, $Lim_{n\to\infty} P_n$ exists because the metric space is complete. Since a contractive map is necessarily a continuous map, it follows by Lemma 1 that $Lim_{n\to\infty} P_n$ is a fixed point of *T*. Finally by Lemma 2, this fixed point is unique.

Two important examples of complete metric spaces are:

- $(\mathbf{R^n}, dist)$, where *dist* is the standard Euclidean metric on $\mathbf{R^n}$.
- $(H(\mathbf{R^n}), h)$, where *h* is the Hausdorff distance on $H(\mathbf{R^n})$, the space of compact subsets of $\mathbf{R^n}$. (For a rigorous proof, see [Barnsley, 1993].)

Let us focus for now on the second example.

Suppose that $\{R_1,\ldots,R_m\}$ is a collection of contractive maps on $\mathbf{R^n}$. Then $T = \{R_1,\ldots,R_m\}$ is a contractive map on $H(\mathbf{R^n})$ [Barnsley, 1993]. Therefore it follows by the trivial fixed point theorem that:

a. There exists a unique compact set *G* such that $T(G) = G$.

b. The sets $G_k = T(G_{k-1})$ converge to *G* and the convergence is independent of the choice of the initial set G_0.

In other words,

a′. a contractive IAT determines a unique fractal, the unique fixed point of the IAT.

b′. the deterministic algorithm always converges to the unique fractal determined by a contractive IAT independent of the base case.

In addition to the investigation of fractals, students should be shown several other important applications of the trivial fixed point theorem. Here we shall consider briefly two such applications -- solving transcendental equations and solving large systems of linear equations -- based on the completeness of the metric space $(\mathbf{R^n}, dist)$.

Suppose we need to solve the equation $F(x) = 0$. Let $G(x) = F(x) + x$. Then

$$F(a) = 0 \Leftrightarrow G(a) = a.$$

That is, *a* is a root of *F* if and only if *a* is a fixed point of *G*. If *F* is a transcendental function, it may be difficult to find a root of *F* directly, but if *G* happens to be a contractive map, then by the trivial fixed point theorem we can find a root of *G* by iteration.

For example, suppose we want to find a root of the transcendental function $F(x) = \cos(x) - x$. Let $G(x) = F(x) + x = \cos(x)$. Then a root of F is equivalent to a fixed point of G. But G is a contractive map on the interval $[-a,a]$ whenever $0 < a < \pi/2$ because by the Mean Value Theorem if $x, y \in [-a,a]$, then

$$|\cos(x) - \cos(y)| \leq \sin(a)\,|x - y|.$$

Therefore, since $[-a,a] \subset \mathbf{R}$ is a complete metric space, it follows by the trivial fixed point theorem that the sequence $x_0, G(x_0), G(G(x_0)), \ldots$ will converge to a fixed point of G, and hence to a root of F, for any choice of $x_0 \in [-a,a]$. We illustrate this convergence for two distinct choices of x_0 in Figure 6.

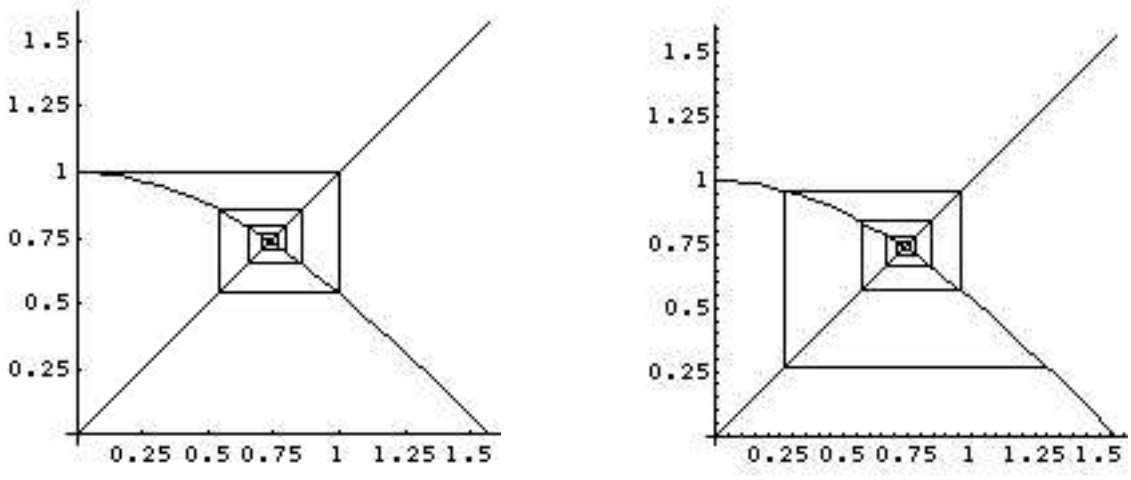

Figure 6: Finding a root of the function $F(x) = \cos(x) - x$ by computing a fixed point of the function $G(x) = F(x) + x = \cos(x)$ using iteration. Convergence to the root is illustrated for two distinct starting points x_0: on the left $x_0 = 0$, on the right $x_0 = 1.3$. In both cases 10 iterations are shown, and in both cases the sequence $x_0, G(x_0), G(G(x_0)), \ldots$ spirals rapidly into the root $x \approx .739$ at the intersection of the curves $y = \cos(x)$ and $y = x$.

Another important application of the trivial fixed point theorem in computer graphics is solving large systems of linear equations, for example the radiosity equations [Sillion and Puech, 1994]. Consider a system of n independent linear equations in n unknowns:

$$\begin{aligned} M_{11}X_1 + \cdots + M_{1n}X_n &= B_1 \\ \vdots \qquad\qquad & \vdots \quad \Leftrightarrow \quad M * B. \\ M_{n1}X_1 + \cdots + M_{nn}X_n &= B_n \end{aligned}$$

Students know that they can solve this system for the unknowns $X = (X_1, \ldots, X_n)$ using Gaussian elimination, or Cramer's rule, or simply by inverting the matrix M. But when n, the number of equations, is large, Cramer's rule and matrix inversion are numerically unstable, and Gaussian

elimination is slow; iterative methods often work better. Iterative methods for solving systems of linear equations are called *relaxation methods*. We shall look at two such relaxation methods: one due to Jacobi and the other to Gauss-Seidel. These relaxation methods are an important application of the trivial fixed point theorem.

In Jacobi relaxation we start with some initial guess, usually either $X^0 = \mathbf{0} = (0,\ldots,0)$ or $X^0 = \mathbf{B} = (B_1,\ldots,B_n)$. Then we iterate using the formula:

$$X_i^k = \frac{B_i}{M_{ii}} - \sum_{j \neq i} \frac{M_{ij}}{M_{ii}} X_j^{k-1} \qquad k \geq 1.$$

This formula is equivalent to

$$X^k = T(X^{k-1}) \equiv (I - Q^{-1}M)X^{k-1} + Q^{-1}B,$$

where Q is the diagonal part of M and I is the $n \times n$ identity matrix. Note that Q^{-1} is easy to compute, since to invert a diagonal matrix we simply invert the diagonal entries. If T is a contractive map, then the trivial fixed point theorem guarantees convergence. When the coefficient matrix is diagonally dominant -- that is, when

$$M_{ii} \geq \sum_{j \neq i} | M_{ij} |,$$

T is automatically a contractive map. This is exactly the case for the radiosity equations [Sillion and Puech, 1994].

In Gauss-Seidel relaxation we also start with some initial guess, again usually either $X^0 = \mathbf{0} = (0,\ldots,0)$ or $X^0 = \mathbf{B} = (B_1,\ldots,B_n)$. But now we iterate using the formula:

$$X_i^k = \frac{B_i}{M_{ii}} - \sum_{j<i} \frac{M_{ij}}{M_{ii}} X_j^k - \sum_{j>i} \frac{M_{ij}}{M_{ii}} X_j^{k-1} \qquad k \geq 1.$$

This formula is equivalent to

$$X^k = T(X^{k-1}) \equiv (I - Q^{-1}M)X^{k-1} + Q^{-1}B,$$

where Q is the lower triangular part of M and I is the $n \times n$ identity matrix. Notice again that Q^{-1} is much easier to compute than M^{-1}, since it is relatively easy to invert a lower triangular matrix. Again if T is a contractive map, then the trivial fixed point theorem guarantees convergence, and again when the coefficient matrix is diagonally dominant T is automatically a contractive map. Gauss-Seidel relaxation is typically faster than Jacobi relaxation because the method of Gauss-Seidel takes into account the values of X^k already computed during the current iteration.

8. Bezier Curves and Surfaces

So far the turtle has steered students towards the study of affine coordinates and affine transformations, iterated affine transformations and fractal curves, as well as to the trivial fixed point theorem and relaxation techniques for solving large systems of linear equations. These topics all have relevance to computer graphics; we turn now to applications in computer aided design. The main idea of this section, that Bezier curves and surfaces are fractals, was first disclosed to one of us (Ron Goldman) by Joe Warren [Warren, 1993].

Bezier curves and surfaces are some of the simplest, most common, and most important curves and surfaces in computer aided design. Bezier curves are typically defined in terms of the Bernstein basis functions

$$B_k^n(t) = \binom{n}{k} t^k (1-t)^{n-k} \qquad k = 0,\ldots,n.$$

Let $P_0,\ldots,P_n$ be a sequence of points in $\mathbf{R}^{\mathbf{m}}$. Then the Bezier curve with control points $P_0,\ldots,P_n$ is the parametric polynomial curve defined by

$$P(t) = \sum_{k=0}^{n} B_k^n(t) P_k \qquad 0 \le t \le 1.$$

Subdivision is the standard approach in computer aided design to analyzing Bezier curves. A *subdivision algorithm* is a technique for splitting a Bezier curve into two Bezier curves at some parameter value $t \in [0,1]$, typically at $t = 1/2$. Let $Q_0,\ldots,Q_n$ and $R_0,\ldots,R_n$ be the control points of the Bezier curve $P(t)$ restricted to the intervals $[0,1/2]$ and $[1/2,1]$. A subdivision algorithm is a procedure for computing the Bezier control points $Q_0,\ldots,Q_n$ and $R_0,\ldots,R_n$ from the original Bezier control points $P_0,\ldots,P_n$. Subdivision algorithms are important in computer aided design because subdivision can be applied to render and intersect Bezier curves [Goldman, 2002a]. We shall now show that recursive subdivision leads to a fractal algorithm for generating Bezier curves.

There is a well known subdivision algorithm for Bezier curves due to de Casteljau [de Casteljau, 1985]. The de Casteljau subdivision algorithm can be used to compute either explicit or recursive formulas for $Q = (Q_0,\ldots,Q_n)^T$ and $R = (R_0,\ldots,R_n)^T$ from the original control points $P = (P_0,\ldots,P_n)^T$. Here we shall focus on the explicit formulas. From the de Casteljau algorithm we find that

$$Q_k = \sum_{j=0}^{k} B_j^k(1/2) P_j \qquad k = 0,\ldots,n$$

$$R_k = \sum_{j=0}^{n-k} B_j^{n-k}(1/2) P_{j+k} \qquad k = 0,\ldots,n.$$

We can rewrite these equations more compactly in matrix form. Let

$$L = \begin{pmatrix} B_0^0(1/2) & 0 & \cdots & 0 \\ B_0^1(1/2) & B_1^1(1/2) & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ B_0^n(1/2) & B_1^n(1/2) & \cdots & B_n^n(1/2) \end{pmatrix} = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ \frac{1}{2} & \frac{1}{2} & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ \frac{1}{2^n} & \frac{n}{2^n} & \cdots & \frac{1}{2^n} \end{pmatrix} = \begin{pmatrix} \frac{\binom{j}{k}}{2^j} \end{pmatrix}$$

$$M = \begin{pmatrix} B_0^n(1/2) & B_1^n(1/2) & \cdots & B_n^n(1/2) \\ 0 & B_0^{n-1}(1/2) & \cdots & B_{n-1}^{n-1}(1/2) \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & B_0^0(1/2) \end{pmatrix} = \begin{pmatrix} \frac{1}{2^n} & \frac{n}{2^n} & \cdots & \frac{1}{2^n} \\ 0 & \frac{1}{2^{n-1}} & \cdots & \frac{1}{2^{n-1}} \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix} = \begin{pmatrix} \frac{\binom{n-j}{n-k}}{2^{n-j}} \end{pmatrix}.$$

Then

$$Q = L * P$$

$$R = M * P.$$

If P is invertible -- that is, if the control points of $P(t)$ are affinely independent -- then using these matrices, we can represent the Bezier curve $P(t)$ as the fixed point of an IAT. Let

$$L_P = P^{-1} * L * P$$

$$M_P = P^{-1} * M * P.$$

and consider the IAT given by the two maps $\{L_P, M_P\}$. It is easy to check that L_P and M_P are contractive maps. In addition,

$$P * L_P = P * (P^{-1} * L * P) = L * P = Q$$

$$P * M_P = P * (P^{-1} * M * P) = M * P = R.$$

More generally, if *N* is any finite product whose factors consist only of the matrices *L* and *M*, then

$$(N * P) * L_P = (N * P) * (P^{-1} * L * P) = (N * L) * P$$

$$(N * P) * M_P = (N * P) * (P^{-1} * M * P) = (N * M) * P.$$

Thus $\{L_P, M_P\}$ applied repeatedly to the control points P generates a recursive subdivision algorithm for the Bezier curve $P(t)$.

Recursive subdivision applied to the control points is known to converge to the original Bezier curve [Lane and Riesenfeld, 1980]. Therefore, if we iterate the two maps $\{L_P, M_P\}$ on the control polygon for the initial set of control points *P*, the resulting set of control polygons will converge to the original Bezier curve $P(t)$. But $\{L_P, M_P\}$ are contractive maps, so by the trivial fixed point

theorem this convergence is independent of the initial set! We illustrate this convergence for two different initial sets in Figure 7. In summary, the fractal generated by the IAT consisting of the two subdivision matrices $\{L_P, M_P\}$ is precisely the Bezier curve $P(t)$ whose control points are P.

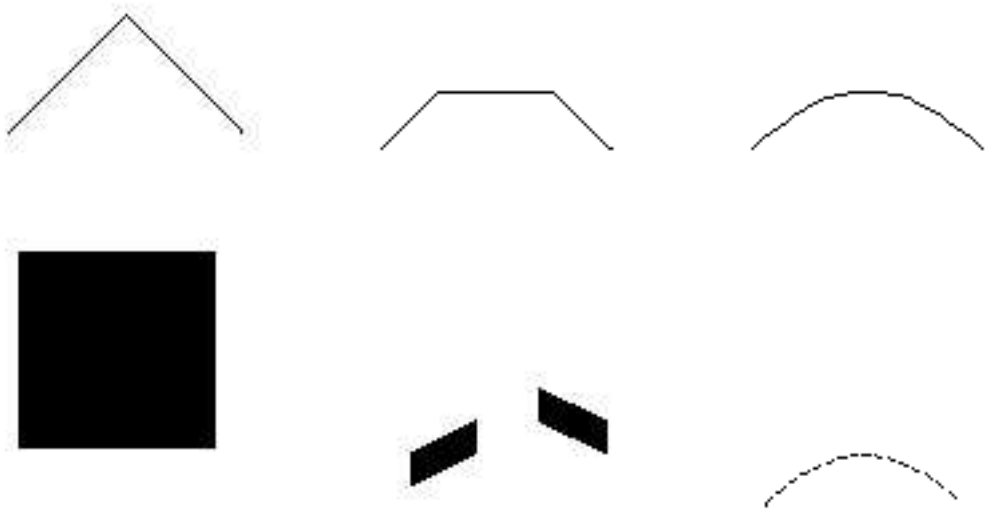

Figure 7: A quadratic Bezier curve generated by iterating the subdivision matrices $\{L_P, M_P\}$ on two different initial sets. On the top level, we illustrate the control polygon $P = ((0,0),(1,1),(2,0))$ together with the output of the subdivision algorithm after 1 and 3 levels of subdivision. On the bottom level, we start with a filled in square, and we illustrate the output of the deterministic algorithm after 1 and 4 iterations. In both cases, the sets converge to the quadratic Bezier curve whose control points are P.

If the control points $P_0,\ldots,P_n$ lie in the plane and $n \geq 3$, then the points $P_0,\ldots,P_n$ cannot be affinely independent, so the matrix P^{-1} does not exist. Nevertheless, we can still construct the matrices $\{L_P, M_P\}$ for the IAT by lifting the control points $P_0,\ldots,P_n$ to a higher dimensional space. For quadratics, we use affine coordinates and set

$$P = \begin{pmatrix} P_0 & 1 \\ P_1 & 1 \\ P_2 & 1 \end{pmatrix} = \begin{pmatrix} x_0 & y_0 & 1 \\ x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \end{pmatrix};$$

For higher degrees, we simply generalize these affine coordinates to higher dimensions. For cubics, set

$$P = \begin{pmatrix} P_0 & 1 & 0 \\ P_1 & 1 & 0 \\ P_2 & 1 & 0 \\ P_3 & 1 & 1 \end{pmatrix} = \begin{pmatrix} x_0 & y_0 & 1 & 0 \\ x_1 & y_1 & 1 & 0 \\ x_2 & y_2 & 1 & 0 \\ x_3 & y_3 & 1 & 1 \end{pmatrix};$$

for quartics,

$$P = \begin{pmatrix} P_0 & 1 & 0 & 0 \\ P_1 & 1 & 0 & 0 \\ P_2 & 1 & 0 & 0 \\ P_3 & 1 & 1 & 0 \\ P_4 & 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} x_0 & y_0 & 1 & 0 & 0 \\ x_1 & y_1 & 1 & 0 & 0 \\ x_2 & y_2 & 1 & 0 & 0 \\ x_3 & y_3 & 1 & 1 & 0 \\ x_4 & y_4 & 1 & 1 & 1 \end{pmatrix},$$

and so on for higher and higher degree. By construction

$$\det(P) = \det \begin{pmatrix} P_0 & 1 \\ P_1 & 1 \\ P_2 & 1 \end{pmatrix},$$

so if P_0, P_1, P_2 are not collinear, then $\det(P) \neq 0$ and P^{-1} exists. For a Bezier curve of degree *n*, the fractal generated by the IAT consisting of the two subdivision matrices $\{L_P, M_P\}$ is an *n*-dimensional curve. Projecting this high dimensional curve orthogonally into the *xy*-plane reproduces the original Bezier curve $P(t)$.

Since every IAT corresponds to a recursive turtle program, every Bezier curve can be generated by a recursive turtle program. Since, however, the matrices $\{L_P, M_P\}$ typically are not oriented conformal affine transformations, we would need to use a turtle that carries along a left hand vector and responds to the more powerful $TURN(a_1, a_2)$ and $RESIZE(s_1, s_2)$ commands. For an explicit example of a turtle program that generates a Bezier curve, see [Ju *et al*, 2003].

Bezier surfaces also have subdivision procedures; therefore Bezier surfaces can also be generated by fractal algorithms. For triangular Bezier patches, the IAT corresponding to subdivision contains three matrices; for tensor product Bezier patches the IAT corresponding to subdivision contains four matrices. The remaining analysis is much the same as for Bezier curves.

9. Conclusions

Turtles are a simple virtual paradigm for investigating complicated geometric topics. Thus turtles lend themselves quite naturally to education; indeed education was the primary purpose for

which turtle geometry was originally invented.

We have traced only a few of the many possible ways that turtle geometry and turtle programming can be used for introducing and investigating standard concepts in computer graphics and computer aided design. Below is a schematic summary of the paths we have taken.

Turtle Commands $\rightarrow$ Translation, Rotation, and Scaling $\rightarrow$ Affine Coordinates and Affine Transformations

Recursive Turtle Programs $\rightarrow$ Fractals $\rightarrow$ IATs $\rightarrow$ Trivial Fixed Point Theorem

$\downarrow$ $\downarrow$

Bezier Curves (Subdivision) Relaxation Methods (Jacobi, Gauss-Seidel)

Other path are certainly possible. To the turtle's state (P,w), we could add a scalar mass m. The new turtle state could be denoted by (mP,m,w), where (mP,m) represents a mass-point. Mass-points (often mislabeled as homogeneous coordinates) provide the mathematical foundation for much of computer graphics [Goldman, 2002b, 2003], and they supply as well the control structures for rational Bezier and rational B-spline curves and surfaces [Goldman, 2002a].

To study the effects of mass (inertia), we would need to update the FORWARD and MOVE commands so that

FORWARD d: $(mP,m) \rightarrow (mP + dw, m)$ $w \rightarrow w$

MOVE d: $(mP,m) \rightarrow (mP + dw, m)$ $w \rightarrow w$

The turtle's new location after executing these commands is

$$\frac{mP + dw}{m} = P + \frac{dw}{m}.$$

Hence the higher the turtle's mass, the less the turtle's location changes when we command the turtle to advance.

Extensions of the turtle to 3-dimensions could also be introduced [Abelson and diSessa, 1986], and with the natural extensions of the turtle commands to 3-dimensions, students could proceed to the study of Grassmann coordinates and the full range of projective transformations much as we studied affine coordinates and affine transformations.

Thus the humble turtle can provide an easy initial entry for students into many of the most important and most vital topics in computer graphics and computer aided design.

Bibliography

1. Abelson, H. and diSessa, A. (1986), *Turtle Geometry: The Computer as a Medium for Exploring Mathematics*, MIT Press, Cambridge, Mass.

2. Barnsley, M. (1993), *Fractals Everywhere*, (Second Edition), Academic Press, Boston, Mass.

3. Blinn, J. (2002), Using tensor diagrams to represent and solve geometric problems, *Siggraph 2002, Course 25*.

4. de Casteljau, P. (1985), *Formes a Poles*, Hermes, Paris.

5. Dorst, L. and Mann, S. (2002), Geometric algebra: A computational framework for geometric applications, Parts I and II, *IEEE Computer Graphics and Applications*, Vol. 22, no. 3, pp. 24-31 and Vol. 22, no. 4, pp. 58-67.

6. Goldman, R. (2001), Baseball arithmetic and the laws of pseudoperspective, *IEEE Computer Graphics and Applications*, Vol. 21, pp. 70-78.

7. Goldman, R. (2002a), *Pyramid Algorithms: A Dynamic Programming Approach to Curves and Surfaces for Geometric Modeling*, Morgan Kaufmann, San Francisco.

8. Goldman, R. (2002b), On the algebraic and geometric foundations of computer graphics, *Transactions on Graphics* (2002), Vol. 21, pp. 1-35.

9. Goldman, R. (2003), Computer graphics in its fifth decade: Ferment at the foundations, (Invited Paper), *Pacific Graphics 2003*, Canmore, Canada, October 2003, pp. 4-21.

10. Harvey, B. (1985-1987), *Computer Science LOGO Style, Vols. 1-3*, MIT Press, Cambridge, Mass.

11. Hoffmann, C. (1989), *Geometric and Solid Modeling: An Introduction*, Morgan Kaufmann, San Mateo, California.

12. Ju, T., Schaefer, S., and Goldman, R. (2003), Recursive turtle programs and iterated affine transformations, in preparation.

13. Lane, J. and Riesenfeld, R. (1980), A theoretical development for the computer generation and display of piecewise polynomial surfaces, *IEEE Trans. on Pattern Anal. and Mach.*

Intell., Vol. 2, pp. 35-46.

14. Murray, R., Li, Z., and Sastry, S. (1994), *A Mathematical Introduction to Robotic Manipulation*, CRC Press, Boca Raton, Florida.

15. Papert, S. (1980), *Mindstorms: Children, Computers and Powerful Ideas*, Basic Books, New York.

16. Prusinkiewicz, P. (1986), Graphical applications of L-systems, *Proceedings of Graphical Interface 86 and Vision Interface 86*, pp. 247-253.

17. Ramshaw, L. (1989), Blossoms are polar forms, *Computer Aided Geometric Design*, Vol. 6, pp. 323-358.

18. Riesenfeld, R. (1981), Homogeneous coordinates and the projective plane in computer graphics, *IEEE Computer Graphics and Applications*, Vol. 1, pp. 50-55.

19. Sederberg, T., Gao, P., Wang, G., and Mu, H. (1993), An intrinsic solution to the vertex path problem, *Computer Graphics (Proceedings Siggraph '93)*, pp. 15-18.

20. Sillion, F. and Puech, C. (1994), *Radiosity and Global Illumination*, Morgan Kaufmann, San Francisco.

21. Warren, J. (1993), Private communication.