Generating Self-Affine Tiles and Their Boundaries

Mark McClure

A self-affine tile is a two-dimensional set satisfying an expansion identity that allows tiling images to be generated. In this article, we discuss the generation of such images paying particular attention to the boundary of the set, which frequently displays a fractal structure.

■ Introduction

A *tile* is a bounded subset of the plane, copies of which may be used to cover the whole plane without gaps or overlap. There are many sources (such as [1]) of beautiful images involving tiling, from medieval Islamic art, through Escher, to more modern work. Perhaps the simplest example of a tile, though, is a solid square, which may tile the plane in a familiar checkerboard pattern. The square is also an example of an important subclass of tiles called the *self-affine tiles*. A tile T is self-affine if there is an expanding matrix A and a collection of vectors $\mathcal{D}$ (called the *digit set*) such that

$$A \cdot (T) = T + \mathcal{D} \equiv \bigcup_{d \in \mathcal{D}} (T + d), \tag{1}$$

where the pieces in the union are assumed to intersect only in their boundaries. Here $A \cdot (T)$ is the image of T under multiplication by the matrix A. Note that if T is a self-affine tile with respect to A and $\mathcal{D}$, then $A \cdot (T)$ is a self-affine tile with respect to A and $A \cdot (\mathcal{D})$. Thus, iteration of equation (1) yields arbitrarily large tiling images. The unit square is an example of a self-affine tile where

$$A = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix} \text{ and } \mathcal{D} = \left\{ \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \end{pmatrix} \right\}. \tag{2}$$

Iteration of equation (2) yields the checkerboard pattern.

As we will see, self-affine tiles of surprising intricacy may be generated using the notion of an iterated function system (IFS) from fractal geometry. For example, the image in Figure 1 is a self-affine four-tile (i.e., it consists of four parts) corresponding to the matrix and digit set

$$A = \begin{pmatrix} 1 & -\sqrt{3} \\ \sqrt{3} & 1 \end{pmatrix} \text{ and } \mathcal{D} = \left\{ \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} -1/2 \\ \sqrt{3}/2 \end{pmatrix}, \begin{pmatrix} -1/2 \\ -\sqrt{3}/2 \end{pmatrix} \right\}. \tag{3}$$

This text is the heading of a closed group that sets up definitions.

```
In[25]:= Show[Graphics[{pic1, pic2, pic3, pic4}],
          AspectRatio → Automatic]
```

Out[25]=

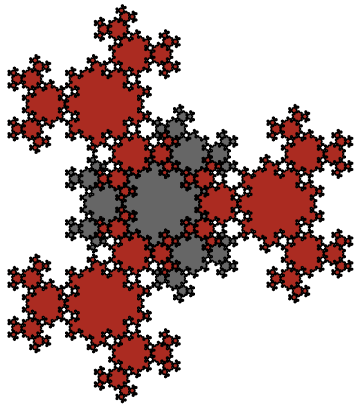


Figure 1. A self-affine four-tile.

Constructing interesting images is greatly simplified by the existence of fairly simple rules dictating possible choices for the matrix A and digit set $\mathcal{D}$. Also of interest is the boundary between the constituent parts. The boundary of a self-affine tile frequently has a fractal structure and may be generated and analyzed using a generalized notion of an IFS. The boundary of the four-tile, for example, may be shown to have a fractal dimension of $\log(3)/\log(2) \approx 1.585$.

■ Self-Affine Sets and Tiling

Self-similarity and iterated function systems are, by now, fairly well-known concepts. See, for example, [2, 3] for a general introduction or [4, 5], which describe implementations using *Mathematica*. Here, we briefly define our terms to establish notation and clarify important results.

Roughly speaking, a set is called *self-similar* if it is composed of two or more sets geometrically similar to the whole. Self-similarity is more rigorously defined and analyzed using an important tool called an *iterated function system*, or IFS. An IFS is simply any finite collection $\{f_i\}_{i=1}^{m}$ of contractive mappings of the plane. Associated with an IFS there is always a unique nonempty, closed, bounded set E satisfying

$$E = \bigcup_{i=1}^{m} f_i(E). \tag{4}$$

The set E defined in equation (4) is called the *invariant set* or *attractor* of the IFS. The functions in an IFS describe the exact relationship between the invariant set and its constituent parts. If the IFS consists entirely of contractive similarities, then E is called *self-similar*. If the IFS consists of affine functions, then E is called *self-affine*.

Self-affine sets have played an important role in the development of fractal geometry in part because they provide a dazzling class of images, even though affine functions are very easy to describe and implement on a computer. Thus, it takes a small amount of information to store very interesting images. In *Mathematica* (in particular, in the packages described here), an affine function may be represented as `{A,b}`, where `A` is a two-dimensional matrix and `b` is a shift vector. The following code represents an IFS for the unit square.

```
In[26]:= A = {{1/2, 0}, {0, 1/2}};
         f1 = {A, {0, 0}};
         f2 = {A, {1/2, 0}};
         f3 = {A, {1/2, 1/2}};
         f4 = {A, {0, 1/2}};
         squareIFS = {f1, f2, f3, f4};
```

In order to generate an image of the square, we use the `ShowIFS` command defined in the *IFS* package. The implementation of the `ShowIFS` command is similar to commands described in [4, 5].

```
In[32]:= Needs["FractalGeometry`IFS`"];

In[33]:= ShowIFS[squareIFS, 9, Color → True,
          Colors → {Maroon, Gray, Maroon, Gray}]
```

Out[33]=

Figure 2. A square generated from an IFS.

In the `ShowIFS` command, the second argument (9 in this case) indicates the depth of the approximation. Thus, the image consists of $4^9 = 262{,}144$ points distributed over the unit square. A large number of points is typically required, as we want to fill a two-dimensional region. The `Color` option is nice when in-

vestigating tiles to highlight the constituent parts. When `Color` is set to `True`, the `Colors` option may be set to `Automatic` (the default), in which case the `Hue` function is used to generate a spectrum of colors, or `Colors` may be set to a list of colors.

Now, a self-affine tile is also a self-affine set. If a self-affine tile T satisfies equation (1), then after applying A^{-1} to both sides we see that

$$T = \bigcup_{d \in \mathcal{D}} \left(A^{-1}\, T + A^{-1}\, d\right). \tag{5}$$

In fact, this is the exact relationship between the description of the square as a self-affine tile given by equations (2) and the IFS defined by `squareIFS`; it is easy to pass from one description to the other. The major question now is how to choose a matrix A and digit set $\mathcal{D}$ to generate interesting images. A beautiful theorem, published by Christoph Bandt [6], provides an answer. This theorem is also described in [7] at a more elementary level.

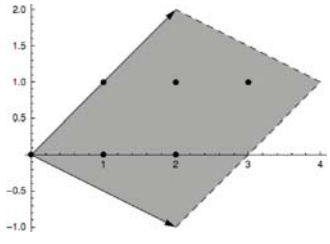
Theorem. *Let A be a two-dimensional expansive matrix with integer entries and let $\mathcal{D}$ form a residue system for A. Then, there is a unique self-affine tile T with matrix A and digit set $\mathcal{D}$. In fact, T is the invariant set of the IFS consisting of the affine functions defined by $\{A^{-1},\ A^{-1}\, d\}$ for $d \in \mathcal{D}$.*

An *expansive* matrix is simply a matrix whose eigenvalues are all larger than one in absolute value. The terminology residue system and digit set originates from work by Gilbert [8] describing certain self-similar sets in terms of number representation in the complex plane. By definition, a *residue system* for A is a complete set of coset representatives for the quotient group $\mathbb{Z}^2 \big/ A\,\mathbb{Z}^2$. While this definition is abstract, it is fairly easy to describe how to construct a residue system. Given the matrix A, denote its column vectors by v_1 and v_2. The simplest residue system for A consists of those points with integer coordinates lying inside the parallelogram determined by v_1 and v_2 and including only the two sides containing the origin. For example, the following figure illustrates this simple digit set for the matrix

$$\begin{pmatrix} 2 & 2 \\ -1 & 2 \end{pmatrix}.$$

```
In[34]:= ShowBaseDigitSet[Transpose@{{2, -1}, {2, 2}}]

Out[34]= ShowBaseDigitSet[{{2, 2}, {-1, 2}}]
```



We may construct other residue systems from this simple one as follows: two integer points are said to be equivalent if their difference is a linear combination of v_1 and v_2. Any vector from our simple residue system may be replaced by another from its equivalence class; that is, starting from our simplest residue system, we may simply shift some of its members by some linear combination of v_1 and v_2 to obtain another residue system. A digit set which forms a residue system for A is called a *standard digit set*. Note that the shift of a standard digit set by an integer vector is again a standard digit set; thus, we may suppose that the zero vector is one of the digits. Some of our package functions use this simplifying assumption, so it is best to use digit sets containing the origin.

Let us demonstrate how easy it is to generate interesting self-affine tiles using this theorem. We first describe a simple modification of the square's IFS. We use the same matrix, a simple expansion by the factor 2, but we replace one of the digits by a shift. In particular, we shift the digit (1, 1) by $-(v_1 + v_2) = (-2, -2)$ to obtain $(-1, -1)$. We use the substitution operator to translate the digit set and matrix into the modified IFS.

```
In[35]:= A = {{2, 0}, {0, 2}};
         D = {{0, 0}, {1, 0}, {-1, -1}, {0, 1}};
         modifiedIFS = D /. {x_?NumericQ, y_} :>
             {Inverse[A], Inverse[A].{x, y}};

In[38]:= ShowIFS[modifiedIFS, 8, Color → True,
          Colors → {Gray, Maroon, Maroon, Maroon},
          Axes → True]
```

Out[38]=

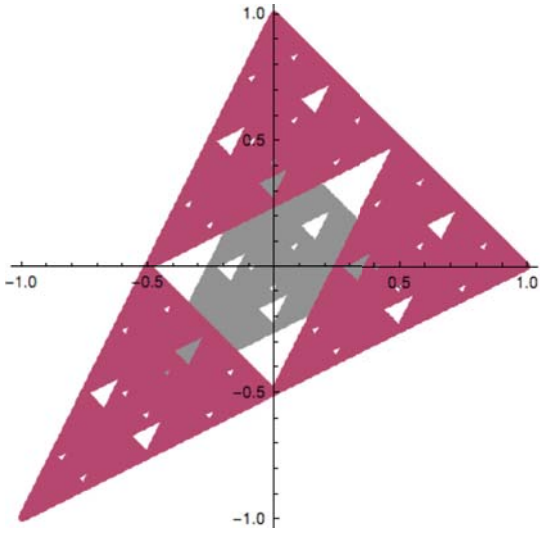
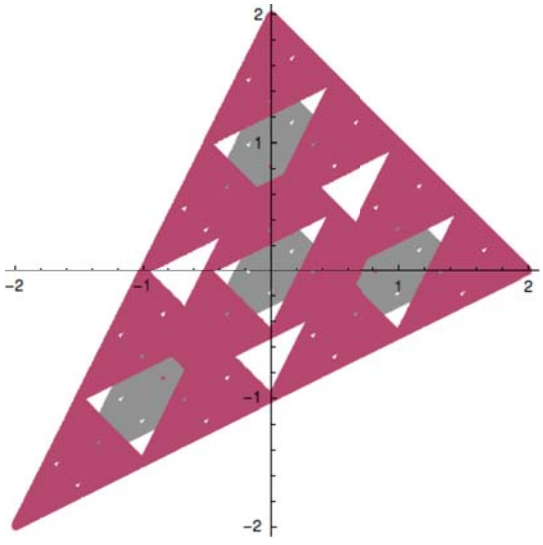


Figure 3. A "minor" modification of Figure 2.

This simple modification is already interesting and the result is difficult even to recognize as a tile. We can get an inkling of how it might tile by examining all shifts of the set by the digit set.

```
In[39]:= Show[% /. {x_?NumericQ, y_?NumericQ} :> {x, y} + # & /@ D,
          Axes → True]
```

Out[39]=



Our next example is called the "twin dragon" and is defined by the following matrix.

```
In[40]:= A = ( 1   1 )
             ( -1  1 );
```

Note that the determinant of this matrix is 2. In general, the absolute value of the determinant indicates the number of pieces constituting the tile. This is because the union on the right of equation (1) increases the area of T by the factor $\#(\mathcal{D})$, while the matrix on the left side of equation (1) increases the area of T by the factor $|\det(A)|$. Thus, in this case, our digit set will have two elements. Using the column vectors, it is easy to determine the simplest digit set.

```
In[41]:= D = {{0, 0}, {1, 0}};
```

Now we translate this matrix and digit set to an IFS, as in the previous example.

```
In[42]:= twindragonIFS = D /. {x_?NumericQ, y_} :>
             {Inverse[A], Inverse[A].{x, y}};
```

Here is the result.

```
In[43]:= twindragonPic = ShowIFS[twindragonIFS, 17,
           Color → True,
           Colors → {Maroon, Gray},
           Axes → True]
```

Out[43]=

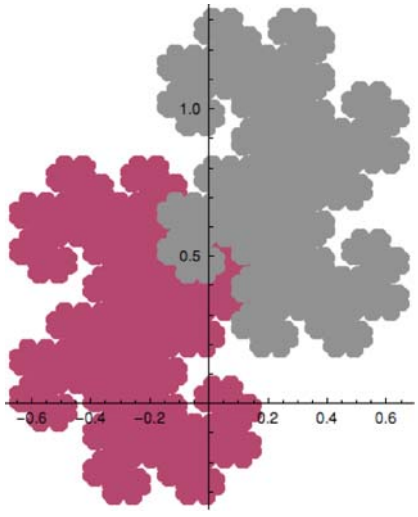


Figure 4. The twin dragon.

The rotation induced by the matrix A, and therefore by A^{-1}, makes it slightly more difficult to see how equation (1) is satisfied. In Figure 5, we see the image of Figure 4 under the mapping $x \to A\,x$. The red part of the twin dragon has clearly mapped onto the whole original twin dragon, while the gray part has mapped onto the original shifted to the right one unit.

```
In[44]:= Show[twindragonPic /. {x_?NumericQ, y_?NumericQ} :> A.{x, y}]
```

Out[44]=

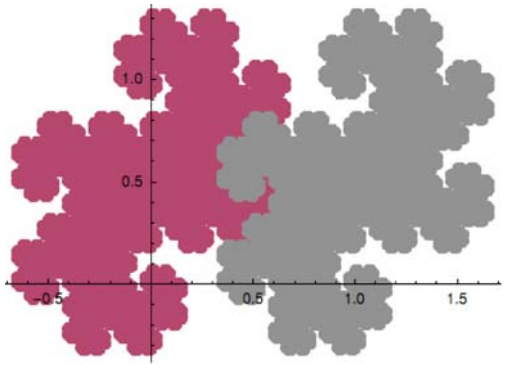


Figure 5. The image of Figure 4 under the mapping $x \to A\,x$.

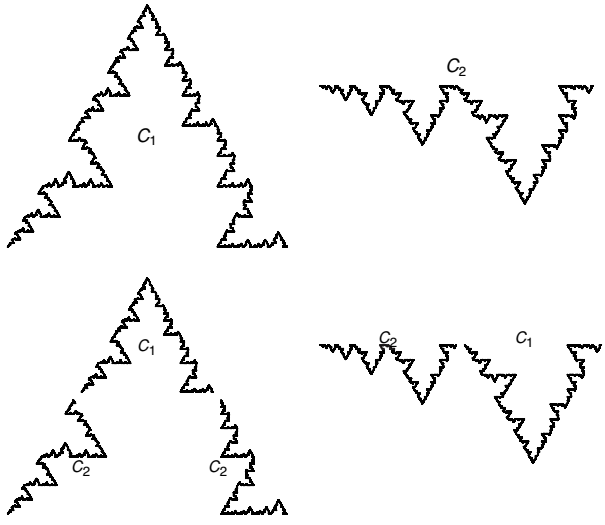
Digraph Iterated Function Systems

Before examining more examples, we turn to the question of how to highlight the boundary between the parts. It turns out that the boundary of a self-affine tile may be generated by a generalized type of IFS called a *digraph iterated function system*. To illustrate this concept, consider the two curves C_1 and C_2. The curve C_1 is composed of one copy of itself, scaled by the factor $1/2$, and two copies of C_2, rotated and scaled by the factor $1/2$. C_2 is composed of one copy of itself, scaled by the factor $1/2$, and one copy of C_1, reflected and scaled by the factor $1/2$.

This text is the heading of a closed group that sets up definitions.

```
In[62]:= Show[
          GraphicsGrid[{{labeledA, labeledB}, {decomposedA, decomposedB}}],
          ImageSize → {288, Automatic},
          BaseStyle → {Background → GrayLevel[1]}]
```

Out[62]=



In general, digraph self-similarity is exhibited by a family of sets $\{K_i\}$. Each set is composed of parts which are scaled images of sets chosen from the collection. A digraph IFS is a matrix M whose elements are lists of affine functions. The elements in row i indicate how the set K_i is composed. Thus, the element M_{ij} in row i and column j should be a list of affine functions mapping K_j into K_i. The analog of equation (4) for a digraph IFS is

$$K_i = \bigcup_j \bigcup_{f \in M_{ij}} f(K_j). \tag{6}$$

As with an IFS, the list of sets $\{K_i\}$ is uniquely determined by the digraph IFS. The curves C_1 and C_2 may be generated using a digraph IFS, which is represented as follows.

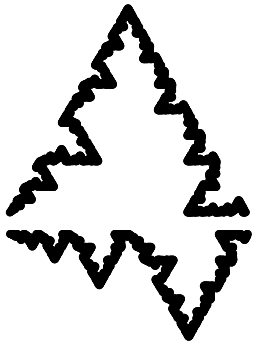
```
In[48]:= a11 = {{{1/2, 0}, {0, 1/2}}, {1/4, Sqrt[3]/4}};
         a12 = {1/2 RotationMatrix[π/3], {0, 0}};
         b12 = {1/2 RotationMatrix[-π/3], {3/4, Sqrt[3]/4}};
         a21 = {{{1/2, 0}, {0, -1/2}}, {1/2, 0}};
         a22 = {{{1/2, 0}, {0, 1/2}}, {0, 0}};
         curvesDigraph = ({a11}  {a12, b12}
                          {a21}    {a22});
```

The `RotationMatrix` function is defined for all of the *FractalGeometry* packages. The *DigraphFractals* package also defines the command `ShowDigraphFractals`, which may be used to generate the curves.

```
In[54]:= Needs["FractalGeometry`DigraphFractals`"];

In[55]:= Column[ShowDigraphFractals[curvesDigraph, 9]]
```

Out[55]=



The terminology digraph fractal arises from a description of the combinatorics involved using directed multigraphs. A *directed multigraph* consists of a finite set of vertices and a finite set of directed edges between vertices. We use the terminology multigraph because we allow more than one edge between any two vertices. Figure 6 depicts the digraph for the curves C_1 and C_2. There are two edges from node C_1 to node C_2 and one edge from node C_1 to itself, since C_1 consists of two copies of C_2 with one copy of itself. Similarly, there is one edge from node C_2 to node C_1 and one edge from C_2 to itself, since C_2 consists of one copy of C_1 together with one copy of itself.

This text is the heading of a closed group that sets up definitions.

```
In[87]:= x0y0Digraph
```

Out[87]=

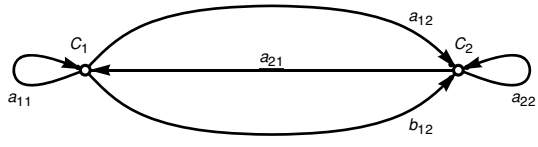


Figure 6. The digraph for the curves C_1 and C_2.

A *path* through a digraph is a finite sequence of edges such that the terminal vertex of any edge is the initial vertex of the subsequent edge. The digraph is called *strongly connected* if, for every pair of vertices u and v, there is a path from u to v. The concept of strong connectivity is important to understand for the following reason: as with a standard IFS, there are two common algorithms for generating images using a digraph IFS; one algorithm is stochastic and the other deterministic. The stochastic algorithm works only when the digraph is strongly connected, while the deterministic algorithm works whether the digraph is strongly connected or not. For the purposes of this article, the stochastic algorithm typically works better but, as we will see, it is not always applicable.

The *DigraphFractals* package is fully described in [9] along with more complete descriptions of the theory and implementation.

■ The Boundary of a Tile

Now we wish to use the digraph IFS scheme to describe the boundary of a self-affine tile. The following technique was published in [10]. Suppose that T is a self-affine tile and there is a lattice Γ of points in the plane so that the translates of T by the points of Γ form a tiling of the plane. The lattice should be invariant under the action of A in the sense that $A(\Gamma) \subset \Gamma$. (Note that the lattice condition is frequently, but not always, satisfied.) Given $\alpha \in \Gamma$, define $T_\alpha = T \bigcap (T + \alpha)$. The boundary of T is formed by the collection of sets T_α that are nonempty, excluding the case $\alpha = 0$. It turns out that these sets T_α are digraph self-affine; that is, if we let $\mathcal{F} = \{\alpha \in \Gamma : T_\alpha \neq \emptyset \text{ and } \alpha \neq 0\}$, then the collection $\{T_\alpha : \alpha \in \mathcal{F}\}$ forms the invariant list of a digraph IFS. This can be demonstrated by examining how the expansion matrix A affects each set T_α and then translating to a digraph IFS by applying A^{-1}:

$$A(T_\alpha) = A(T) \bigcap A(T + \alpha) = \left(\bigcup_{d \in \mathcal{D}} (T + d)\right) \bigcap \left(\bigcup_{d' \in \mathcal{D}} (T + d' + A\,\alpha)\right) =$$

$$\bigcup_{d,d' \in \mathcal{D}} ((T + d) \bigcap (T + d' + A\,\alpha)) = \tag{7}$$

$$\bigcup_{d,d' \in \mathcal{D}} [(T \bigcap (T - d + d' + A\,\alpha)) + d] = \bigcup_{d,d' \in \mathcal{D}} \left(\left(T_{A\,\alpha - d + d'}\right) + d\right).$$

We are only interested in the nonempty intersections, so, given α and β in $\mathcal{F}$, let $M(\alpha, \beta)$ denote the set of pairs of digits (d, d') so that $\beta = A\,\alpha - d + d'$. Then, applying A^{-1} to both sides of equation (7), we see that

$$T_\alpha = \bigcup_{\beta \in \mathcal{F}} \bigcup_{(d,d') \in M(\alpha,\beta)} \left(A^{-1}\, T_\beta + A^{-1}\, d\right). \tag{8}$$

Equation (8) defines a digraph IFS to generate the sets T_α. Given α and β in $\mathcal{F}$, the functions mapping T_β into T_α are precisely those affine functions defined by $\left\{A^{-1}, A^{-1}\, d\right\}$ for all digits d so that there is a digit d' satisfying $\beta = A\,\alpha - d + d'$.

We now implement these ideas to generate the boundary of the twin dragon. We first define `A` and `D`.

```
In[88]:= A = {{1, 1}, {-1, 1}};
         D = {{0, 0}, {1, 0}};
```

We also need to know the set $\mathcal{F}$. In general, it can be difficult to determine $\mathcal{F}$. Fortunately, [10] describes an algorithm to automate the procedure. The algorithm is fairly difficult, however, and the technique seems rather far removed from the other techniques described here. Thus, we refer the interested reader to [10] and the code defining the `NonEmptyShifts` function in the *SelfAffineTiles* package. Examining Figure 4, it is not too difficult to see that the correct set of vectors $\mathcal{F}$ for the twin dragon is defined as follows.

```
In[90]:= F = {{-1, -1}, {0, -1}, {1, 0},
             {1, 1}, {0, 1}, {-1, 0}};
```

The following code illustrates the six translates of the twin dragon and colors them so that they are easy to distinguish.

```
In[91]:= points = Cases[twindragonPic, _Point, Infinity];
         points =
           points /. {x_?NumericQ, y_?NumericQ} :> {x, y} + # & /@ F;
         coloredPoints = Inner[Prepend, h /@ points,
             {Maroon, Gray, Maroon,
              Gray, Maroon, Gray}, List] /. h → List;
         coloredTranslates = Show[Graphics[coloredPoints],
           AspectRatio → Automatic, Axes → True,
           Prolog → AbsolutePointSize[.4]]
```

Out[94]=

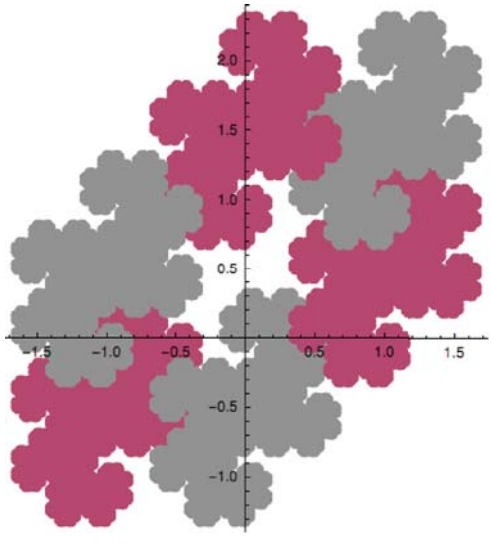


Figure 7. Translates of the twin dragon defining the boundary.

The original twin dragon from Figure 4 is the central white region in Figure 7. The six pieces of the boundary are the boundaries between the white region and the six colored shifted regions.

Now, for each pair (α, β) where α and β are chosen from $\mathcal{F}$, we want $M(\alpha, \beta)$ to denote the set of pairs of digits (d, d') so that $\beta = A\,\alpha - d + d'$. This can be accomplished as follows.

```
In[95]:= pairs[l_List] := (Flatten[Outer[h, l, l, 1]] /. h :> List);
         M[α_, β_] := Select[pairs[D],
            #[[1]] - #[[2]] == β - A.α &];
         digitPairsMatrix = Outer[M, F, F, 1];
```

In order to make sense of this, let us look at the length of each element of the matrix.

```
In[98]:= Map[Length, digitPairsMatrix, {2}] // MatrixForm
```

```
Out[98]//MatrixForm=
⎛0 0 0 0 0 1⎞
⎜2 1 0 0 0 0⎟
⎜0 1 0 0 0 0⎟
⎜0 0 1 0 0 0⎟
⎜0 0 0 2 1 0⎟
⎝0 0 0 0 1 0⎠
```

This matrix is called the *substitution matrix* of the tile and tells us simply the combinatorial information of how the pieces of the boundary fit together. Reading the rows, for example, we see that the first piece is composed of one copy of the last piece, the second piece is composed of one copy of itself and two copies of the first, and so on. Note also that the order of the rows and columns is dictated by the order of the set $\mathcal{F}$. Thus, the first piece refers to the boundary along the maroon image in the lower left of Figure 7, since $(-1, -1)$ is the first shift vector in the set $\mathcal{F}$. The subsequent pieces are numbered counterclockwise around the central tile, since that is the way that $\mathcal{F}$ is set up.

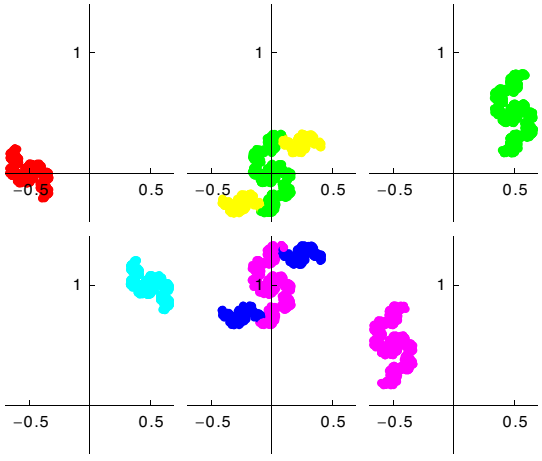
We can transform the `digitPairsMatrix` into a digraph IFS defining the boundary by simply replacing each pair (d, d') with the affine function $\left(A^{-1}, A^{-1}\,d\right)$.

```
In[99]:= boundaryDigraphIFS = digitPairsMatrix /.
           {_, {x_?NumericQ, y_}} :> {Inverse[A], Inverse[A].{x, y}};
```

To see how it worked, we will use the function `ShowDigraphFractalsStochastic` defined in the *DigraphFractals* package. This stochastic algorithm frequently seems to work better for this particular task than the deterministic version defined by `ShowDigraphFractals`. We will use color to distinguish the constituent parts.

```
In[100]:= boundaryParts = ShowDigraphFractalsStochastic[boundaryDigraphIFS,
            20 000, PlotRange → {{-0.7`, 0.7`}, {-0.4`, 1.4`}},
            Ticks → {{-0.5`, 0.5`}, {1}}, Color → True,
            Axes → True, DisplayFunction → Identity];
          GraphicsGrid[Partition[boundaryParts, 3],
           ImageSize → {288, Automatic}]
```

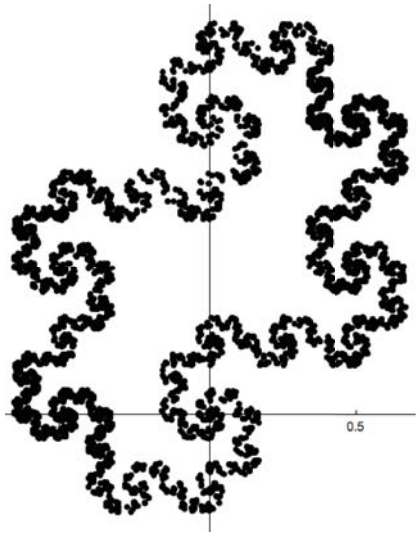
Out[101]=



We can collect all of the pieces to form the entire boundary.

```
In[102]:= boundary = Show[boundaryParts /. Hue[_] → GrayLevel[0]]
```

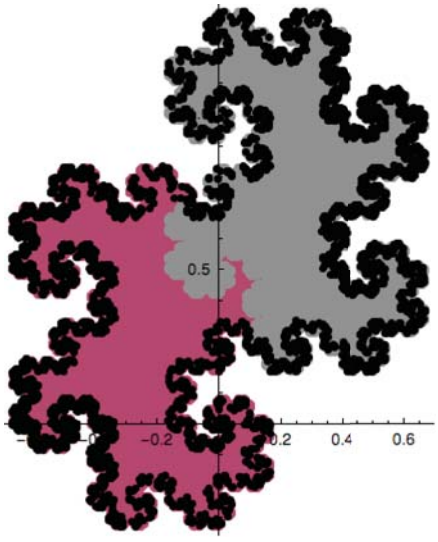
Out[102]=



We can display the boundary with the original image of the twin dragon.

```
In[103]:= Show[{twindragonPic, boundary}]
```

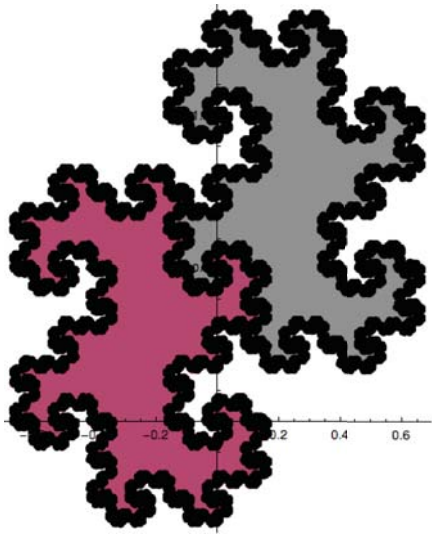
Out[103]=



Note that we have outlined the boundary of the entire set. If we would like to highlight the boundaries of the constituent parts, we need simply to feed the boundary to the `ShowIFS` command using the `boundaryPoints` as an option to `Initiator`.

```
In[104]:= boundaryPoints = Cases[boundary, _Point, Infinity];
          boundaries = ShowIFS[twindragonIFS, 1,
             Initiator → boundaryPoints,
             DisplayFunction → Identity];
          Show[{twindragonPic, boundaries}]
```

Out[106]=



More Examples

The algorithms described are encapsulated in the package *SelfAffineTiles*. We load the package and use it to look at many more examples.

```
In[93]:= Needs["FractalGeometry`SelfAffineTiles`"];
```

The main graphical command which ties all of the previous algorithms together is the `ShowTile` command. `ShowTile[A, depth]` accepts the matrix `A` and generates an approximation to the corresponding self-affine tile to level *depth*. The boundary is automatically generated and the parts are colored differently.

```
In[109]:= A = {{1, 2}, {-1, 1}};
          ShowTile[A, 10]
```

Out[109]=

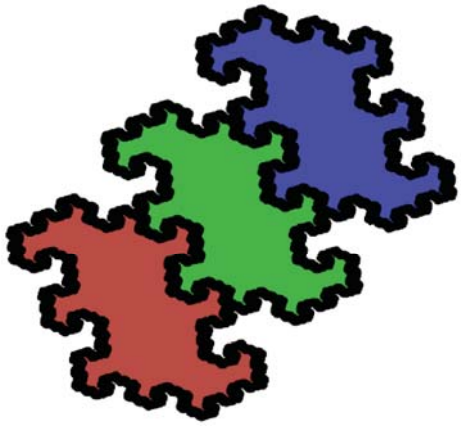
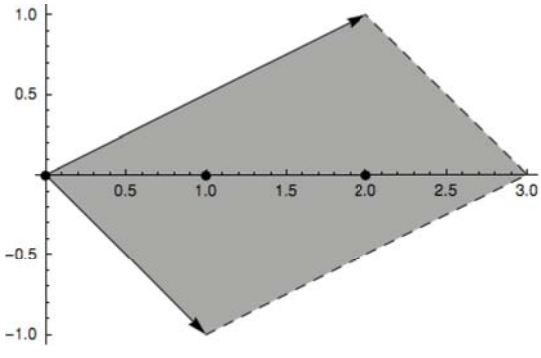


Figure 8. A self-affine three-tile.

The `ShowTile` command accepts the option `DigitSet`. When `DigitSet` is set to the default of `Automatic`, `ShowTile` calls the `BaseDigitSet` function to compute the simple digit set described previously. We can illustrate this simple digit set using the command `ShowBaseDigitSet`.

```
In[110]:= ShowBaseDigitSet[A]
```

Out[110]=



We can also look at the tiles generated by alternative digit sets using the `DigitSet` option. In the following, we subtract the first column vector of A, $\begin{pmatrix} 1 \\ -1 \end{pmatrix}$, from the digit $\begin{pmatrix} 2 \\ 0 \end{pmatrix}$ to obtain the shifted digit $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$.

```
In[111]:= D = {{0, 0}, {1, 0}, {1, 1}};
          ShowTile[A, 10, DigitSet → D]
```

Out[112]=

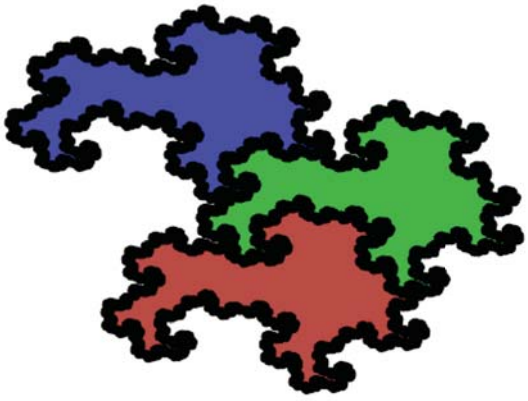


Figure 9. A self-affine three-tile using an alternative digit set.

The tiles in Figures 8 and 9 consist of three pieces, since $\det(A) = 3$. Three-tiles are more diverse than two-tiles, as we have more flexibility in choosing the matrix A and relative positions of the digits. Here is another three-tile using a different matrix.

```
In[113]:= A = {{2, -1}, {1, 1}};
          ShowTile[A, 10]
```

Out[114]=

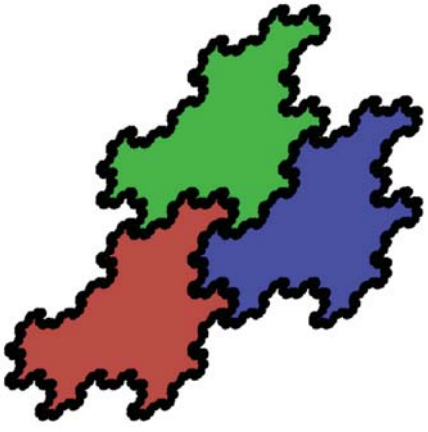


Figure 10. Another self-affine three-tile.

The examples in this section so far have been self-affine but not self-similar. Sometimes, such a set is affinely equivalent to a self-similar set. In this case, the self-similar set will correspond to the same matrix and digit set expressed in another basis. As explained in [7], if A has a pair of complex conjugate eigenvalues, then A is similar (i.e., conjugate) to a similarity matrix. In this case, we may find the change of basis matrix B as follows. Suppose that the vector

$$\begin{pmatrix} v_{11} + i v_{12} \\ v_{21} + i v_{22} \end{pmatrix}$$

is an eigenvector for A, and let B be the inverse of the matrix

$$\begin{pmatrix} v_{11} & v_{12} \\ v_{21} & v_{22} \end{pmatrix}.$$

Then, $B A B^{-1}$ will be a similarity transformation. The self-affine tile shown in Figure 10 falls into this case, as the following computation shows.

```
In[115]:= Eigenvalues[A]
```

$$\text{Out[115]=} \quad \left\{\frac{1}{2}\left(3 + i\sqrt{3}\right), \frac{1}{2}\left(3 - i\sqrt{3}\right)\right\}$$

We can now find one of the corresponding eigenvectors.

```
In[116]:= eigenvec = Eigenvectors[A][[1]] // Simplify
```

$$\text{Out[116]=} \quad \left\{\frac{1}{2}\left(1 + i\sqrt{3}\right), 1\right\}$$

And we can use this to find the change of basis matrix B.

```
In[117]:= B = Inverse[{{Re[eigenvec[[1]]], Im[eigenvec[[1]]]},
              {Re[eigenvec[[2]]], Im[eigenvec[[2]]]}}];
          B // MatrixForm
```

```
Out[118]//MatrixForm=
```

$$\begin{pmatrix} 0 & 1 \\ \frac{2}{\sqrt{3}} & -\frac{1}{\sqrt{3}} \end{pmatrix}$$

The matrix B should conjugate A to a similarity matrix.

```
In[119]:= B.A.Inverse[B] // MatrixForm
```

```
Out[119]//MatrixForm=
```

$$\begin{pmatrix} \frac{3}{2} & \frac{\sqrt{3}}{2} \\ -\frac{\sqrt{3}}{2} & \frac{3}{2} \end{pmatrix}$$

We can see that $B A B^{-1}$ does indeed induce a similarity transformation. In fact, it is simply a clockwise rotation throughout the angle $\pi/6$ together with an expansion of $\sqrt{3}$.

```
In[120]:= Sqrt[3] RotationMatrix[-π/6] // MatrixForm
```

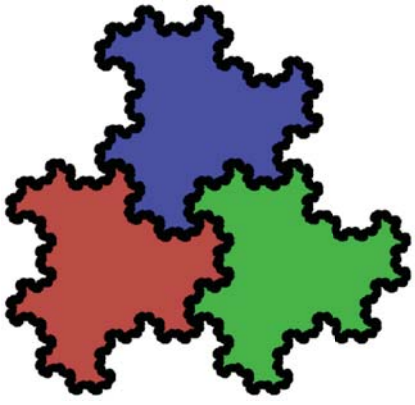
Out[120]//MatrixForm=

$$\begin{pmatrix} \frac{3}{2} & \frac{\sqrt{3}}{2} \\ -\frac{\sqrt{3}}{2} & \frac{3}{2} \end{pmatrix}$$

Now the point is that, while this last matrix does not have integer entries, so it does not seem to fall into the scheme outlined by Bandt's theorem, it may be expressed as a matrix with integer entries with respect to the correct choice of basis. In fact, if we choose our basis to be the column vectors of B, then this similarity is expressed as the matrix A. The statement and proof of Bandt's theorem are essentially algebraic, so the choice of basis does not affect the result. The `ShowTile` function accepts the option `Basis`, which assumes that the matrix is expressed with respect to the given basis. If we rerender the tile defined by A with respect to this new basis, we generate a self-similar set.

```
In[121]:= ShowTile[A, 10,
           Basis → Transpose[B]]
```

Out[121]=



When A is conjugate to a similarity, the fractal dimension of the boundary may be calculated by the formula $\frac{\log\lambda}{\log r}$, where λ is the spectral radius of the substitution matrix and r is the spectral radius of A. (The spectral radius is simply the largest of the absolute values of the eigenvalues.) This formula is encoded in the package function `BoundaryDimension`. For example, here is the dimension of the boundary of the previous tile.

```
In[122]:= BoundaryDimension[A]

Out[122]= BoundaryDimension[{{2, -1}, {1, 1}}]
```

A change of basis can be useful even if the matrix A is already a similarity matrix. For example, the self-similar tile of Figure 3 may be expressed in another basis to yield the tile in Figure 11, which has three-fold rotational symmetry. Note that the matrix and digit set have not changed; only the `Basis` option has been added. Also notice that the `ShowTile` command accepts a `Colors` option, which is similar to the `Colors` option for the `ShowIFS` command.

In[123]:=
```
A = {{2, 0}, {0, 2}};
ShowTile[A, 8,
 Colors → {Gray, Maroon, Maroon, Maroon},
 DigitSet → {{0, 0}, {1, 0}, {-1, -1}, {0, 1}},
 Basis → {{-Sqrt[3]/2, -1/2}, {0, 1}}]
```

Out[124]=

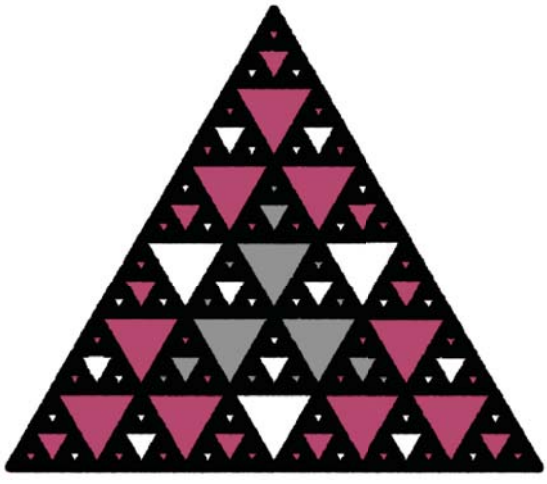


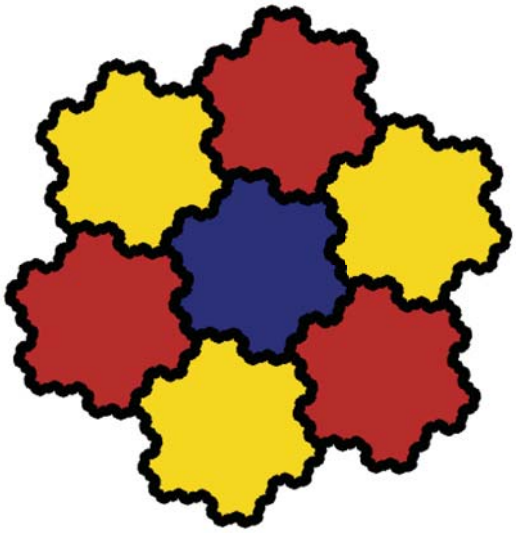
Figure 11. A self-similar four-tile with three-fold rotational symmetry.

We will explain the warning message in the next section, although it does not appear to have genuinely caused a problem in this case.

As a final example, we generate Gosper's famous snowflake.

```
In[125]:= A = {{1, -2}, {2, 3}};
          D = {{0, 0}, {0, 1}, {-1, 1},
             {-1, 0}, {0, -1}, {1, -1}, {1, 0}};
          basis = {{1, 0}, {1/2, Sqrt[3]/2}};
          ShowTile[A, 6,
           DigitSet → D,
           Basis → basis,
           Colors → {MidnightBlue,
             Gold, IndianRed,
             Gold, IndianRed,
             Gold, IndianRed}]
```

Out[128]=



Note that Gosper's flake was also generated using the change of basis technique, as was the tile shown in Figure 1.

■ Potential Problems and Tricks

There are subtle difficulties that may arise when generating images of self-affine tiles, particularly when dealing with the boundary. In this section, we outline some of the tricks that the *SelfAffineTiles* package provides to assist in dealing with these potential problems.

First, it should be understood that many tiling pictures are simply not very attractive. In fact, a randomly chosen digit set is not likely to generate a nice image. Those who play with the package are likely to find several such examples.

Even when the image is quite attractive, subtle issues can arise with the boundary. One of the most important issues is that the digraph describing the

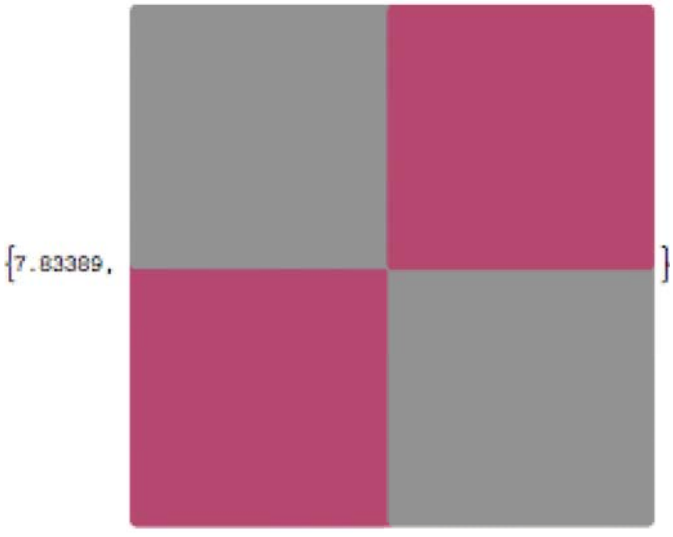
boundary may not be strongly connected. In this case, the stochastic algorithm to generate the boundary might not be effective. This situation arises in the simplest of examples, that of the unit square. Let us try to generate the unit square as simply as possible. For example, the following command will lead to trouble. (The cell is therefore given the setting Evaluatable → False.)

```
A = {{2, 0}, {0, 2}};
ShowTile[A, 9,
   Colors → {
     Maroon, Gray,
     Gray, Maroon}]; // Timing
```

The `ShowTile` command recognizes that the boundary digraph IFS is not strongly connected and suggests two possibilities. Let us follow the first suggestion.

In[129]:=
```
A = {{2, 0}, {0, 2}};
ShowTile[A, 9,
  Colors → {Maroon, Gray, Gray, Maroon},
  Boundary → False] // Timing
```

Out[130]=

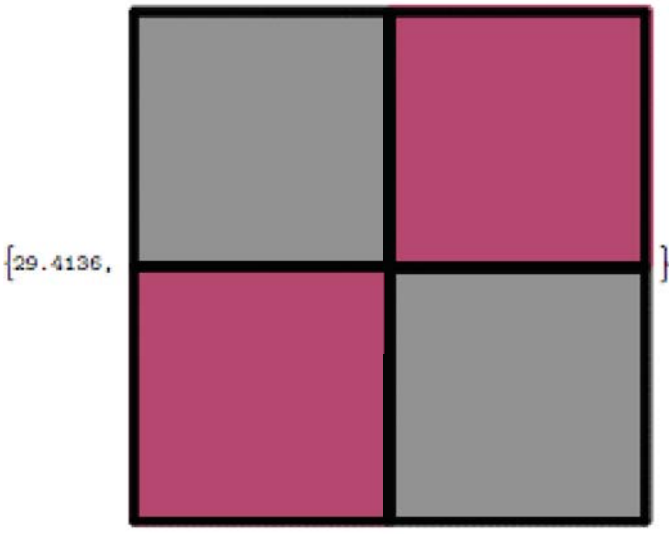


In fact, it is frequently a good idea to set `Boundary → False` when experimenting with `ShowTile` if you do not know what to expect.

Next, we try the second suggestion.

```
In[131]:= A = {{2, 0}, {0, 2}};
          ShowTile[A, 9,
            Colors → {Maroon, Gray, Gray, Maroon},
            BoundaryAlgorithm → Deterministic,
            BoundaryDepth → 6] // Timing
```

Out[132]=



Now an approximation to the boundary has been generated, but this took considerably longer than the previous command to yield a fairly poor image of the boundary. In this case, an understanding of the boundary digraph IFS allows us to refine it and improve the performance. The *SelfAffineTiles* package contains several functions to assist us. First, we look at the substitution matrix of the tile.

```
In[133]:= subsMatrix = SubstitutionMatrix[A];
          subsMatrix // MatrixForm
```

Out[134]//MatrixForm=

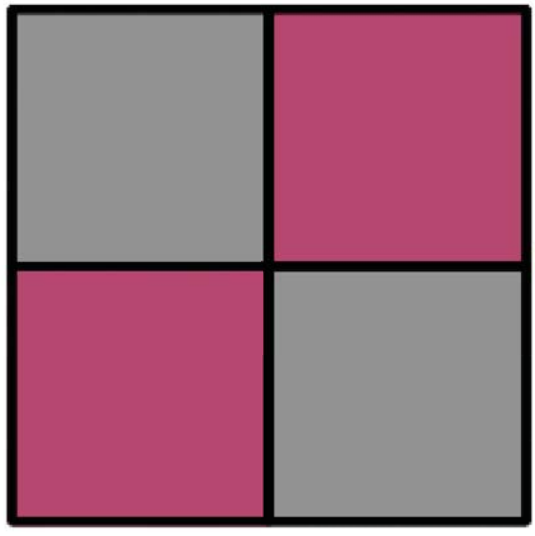
$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 2 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 2 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 2 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 2 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

This tells us that the boundary consists of eight pieces. Note that the first and last pieces each consist of one copy of themselves, while the third and fifth pieces each consist of one copy of the other. Such simple parts of the digraph IFS will generate single points and, in fact, these parts correspond to the vertices of the square. We can verify this by examining the shift set $\mathcal{F}$ used by the program.

```
In[135]:= NonEmptyShifts[A]
Out[135]= {{-1, -1}, {-1, 0}, {-1, 1}, {0, -1}, {0, 1}, {1, -1}, {1, 0}, {1, 1}}
```

Indeed, the first portion of the boundary is simply $T \cap (T - (1, 1))$, where T is the unit square. Of course, this intersection is simply the vertex at the origin. We may generate the entire boundary using only the shifts $\{1, 0\}$, $\{0, 1\}$, $\{-1, 0\}$, and $\{0, -1\}$. This will make the boundary digraph IFS much smaller and speed up the rendering of the boundary considerably. This approach can be implemented using the `Shifts` option.

```
In[136]:= ShowTile[A, 9,
            Colors → {Maroon, Gray, Gray, Maroon},
            BoundaryAlgorithm → Deterministic,
            BoundaryDepth → 8,
            Shifts → {{1, 0}, {0, 1}, {-1, 0}, {0, -1}}] // Timing
Out[136]= {12.5624,
```



Note how much faster this image was generated, even though the greater `Depth` has rendered the boundary in much more detail. We can use the `Substitution Matrix` command to look at the new substitution matrix for the boundary.

```
In[137]:= SubstitutionMatrix[A,
            Shifts → {{1, 0}, {0, 1}, {-1, 0}, {0, -1}}] // MatrixForm
Out[137]//MatrixForm=
```

$$\begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix}$$

It appears that the new boundary digraph IFS is not strongly connected either, so we still could not use the stochastic algorithm for the boundary. We can use the `StronglyConnectedBoundaryQ` command to verify this.

```
In[138]:= StronglyConnectedBoundaryQ[A,
           Shifts → {{1, 0}, {0, 1}, {-1, 0}, {0, -1}}]
Out[138]= False
```

Finally, we outline a technique to generate the boundary of what appears to be the most challenging type of situation (with the exception of tiles simply consisting of a very large number of pieces). Lagarias and Wang [11, 12] carry out a careful analysis of how self-affine tiles can tile the plane and prove that every self-affine tile does indeed tile using translates chosen from some lattice. However, that lattice need not be A invariant, meaning that the technique of [10], which we have implemented here, might not work. The work of Lagarias and Wang shows that frequently the lattice Γ can be chosen to be the lattice generated by $\mathcal{D} \bigcup A(\mathcal{D})$ and, if so, that lattice will be A invariant. In fact, that is exactly the lattice generated by the *SelfAffineTiles* package using the `LatticeReduce` command. They also outline a special case where this might not work and call such an example a *stretched tile* (since its area is too large to tile by the usual lattice). The basic example of a stretched tile is defined by the matrix A and digit set $\mathcal{D}$ given here.

```
In[139]:= A = ( 2  1 ); D = {{0, 0}, {3, 0}, {0, 1}, {3, 1}};
            ( 0  2 )
```

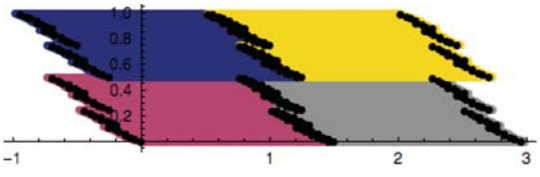
The lattice as described is the integer lattice in this example.

```
In[140]:= LatticeReduce[Join[D, A.# & /@ D]]
Out[140]= {{1, 0}, {0, 1}}
```

As we shall see by simply generating the tile, however, its area is too large to tile via shifts by the integer lattice. In fact, the area of this tile is 3, while any set which tiles via the integer lattice must have an area of only 1.

```
In[141]:= ShowTile[A, 8, DigitSet → D,
           Colors → {Maroon, Gray, MidnightBlue, Gold},
           Axes → True, BoundaryAlgorithm → Deterministic,
           BoundaryDepth → 5]
```

Out[141]=



Note that the boundary is not complete. Of course, we did not really expect this to work. We can, however, use the `Shifts` option to specify the set of all vectors

α from the integer lattice so that $T \bigcap (T + \alpha)$ is a portion of the boundary. We may neglect single point intersections such as $\alpha = \begin{pmatrix} 2 \\ 1 \end{pmatrix}$.

```
In[142]:= ShowTile[A, 8, DigitSet → D, Axes → True,
           Colors → {Maroon, Gray, MidnightBlue, Gold},
           BoundaryAlgorithm → Deterministic, BoundaryDepth → 5,
           Shifts → {{-1, -1}, {0, -1}, {2, -1}, {3, -1}, {3, 0},
             {1, 1}, {0, 1}, {-2, 1}, {-3, 1}, {-3, 0}}]
```

Out[142]=

Our technique essentially works, but the boundary is still not very well approximated. In the next section, we outline a technique to generate very high quality images, which works quite well with this example.

■ Polygonal Initiators

Many of the examples we have seen are tiles which are topological disks. When this is the case, we might try to approximate the boundary with a polygon and feed this result to the `ShowIFS` command. Let us illustrate this technique using the stretched tile of the previous section. We choose to work with this tile for three reasons: the previous techniques proved unsatisfactory; the structure of the tile makes it easy to set up the polygonal approximation; and it is an important theoretical example. We start by taking another look at the boundary.

```
In[143]:= A = {{2, 1}, {0, 2}}; D = {{0, 0}, {3, 0}, {0, 1}, {3, 1}};
          pic = ShowTile[A, 8, DigitSet → D,
            Colors → {Gray, Gray, Gray, Gray},
            OutlineParts → False,
            BoundaryAlgorithm → Deterministic,
            BoundaryDepth → 8]
```

Out[144]=

Once again, we are warned that there might be problems. But notice that the defining points in the boundary have been generated. If we can get them in the correct order, we could simply pass a line through them to generate the boundary. Here is one way to do this. We first grab the points corresponding to

the left half of the boundary, and then sort them according to the y coordinate. The other half of the boundary is simply a reverse-order translate.

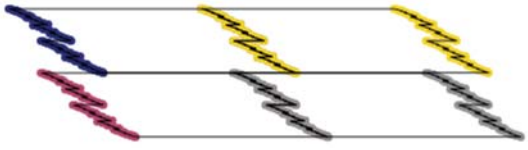
```
In[145]:= points = First /@ Flatten[Cases[pic, {_Point}, Infinity]];
          halfPoints = Select[points, #[[1]] < .01 &] // Union;
          orderedHalf = Sort[halfPoints, #1[[2]] ≤ #2[[2]] &];
          boundary = Join[orderedHalf,
             Reverse[orderedHalf] /. {x_, y_} :> {x + 3, y},
             {First[orderedHalf]}];
          goodPic = Show[Graphics[Polygon[boundary]],
            AspectRatio → Automatic]
```

Out[149]=

Now let us feed the result to the `ShowIFS` command to see how the set decomposes.

```
In[150]:= ifs = TileIFS[A, DigitSet → D];
          init1 = Polygon[boundary];
          init2 = Line[boundary];
          Block[polys = ShowIFS[ifs, 1, Color → True,
              Colors → {Maroon, Gray, MidnightBlue, Gold},
              Initiator → init1];
            boundaries = ShowIFS[ifs, 1, Initiator → init2];
          Show[{polys, boundaries}]
```

Out[154]=



This technique can be extended to many of the other tiles we have looked at in this article. For example, this is how Figure 1 was generated. Unfortunately, most situations require a careful refinement of the digraph IFS algorithms themselves, which is outside the scope of this paper. Furthermore, there is no way to expect that the technique could work in general, since not all self-affine tiles are even connected. Our final example illustrates exactly this point.

```
In[155]:= A = {{3, 0}, {0, 3}};
          D = {
             {0, 0}, {0, 1}, {0, 2},
             {1, 0}, {7, 1}, {1, 2},
             {2, 0}, {2, 1}, {2, 2}
           };
          ShowTile[A, 5, DigitSet → D, PlotRange → All,
           BoundaryAlgorithm → Deterministic, BoundaryDepth → 4,
           Shifts → {{-3, 0}, {-2, 0}, {-1, -1}, {-1, 0}, {-1, 1},
             {0, -1}, {0, 1}, {1, -1}, {1, 0}, {1, 1}, {2, 0}, {3, 0}}]
```

Out[157]=

■ Additional Material

McClure.zip

Available at www.mathematica-journal.com/data/uploads/2009/01/McClure.zip.

■ References

[1] B. Grünbaum and G. C. Shephard, *Tilings and Patterns*, New York: W. H. Freeman & Co., 1986.

[2] M. F. Barnsley, *Fractals Everywhere*, 2nd ed., San Francisco, CA: Morgan Kaufmann Pubs, 2000.

[3] K. J. Falconer, *Fractal Geometry: Mathematical Foundations and Applications*, West Sussex, England: John Wiley & Sons Ltd, 1990.

[4] J. M. Gutiérrez, A. Iglesias, M. A. Rodriguez, and V. J. Rodriguez, "Generating and Rendering Fractal Images," *The Mathematica Journal*, **7**(1), 1997 pp. 6-13.

[5] S. Wagon, *Mathematica in Action*, 2nd ed., New York: Springer-Verlag, 1999.

[6] C. Bandt, "Self-Similar Sets 5. Integer Matrices and Fractal Tilings of $\mathbb{R}^n$," *Proceedings of the American Mathematical Society*, **112**(2), 1991 pp. 549-562.

[7] R. Darst, J. Palagallo, and T. Price, "Fractal Tilings in the Plane," *Mathematics Magazine*, **71**(1), 1998 pp. 12-23.

[8] W. J. Gilbert, "Fractal Geometry Derived from Complex Bases," *The Mathematical Intelligencer*, **4**(2), 1982 pp. 78-86.

[9] M. McClure, "Directed-Graph Iterated Function Systems," *Mathematica in Education and Research*, **9**(2), 2000 pp. 15-26.

[10] R. S. Strichartz and Y. Wang, "Geometry of Self-Affine Tiles I," *Indiana University Mathematics Journal*, **7**(1), 1999 pp. 1-23.

[11] J. C. Lagarias and Y. Wang, "Integral Self-Affine Tiles in $\mathbb{R}^n$ I: Standard and Nonstandard Digit Sets," *Journal of the London Mathematical Society*, **54**(2), 1996 pp. 161–179.

[12] J. C. Lagarias and Y. Wang, "Integral Self-Affine Tiles in $\mathbb{R}^n$ II: Lattice Tilings," *The Journal of Fourier Analysis and Applications*, **3**, 1997 pp. 84–102.

M. McClure, "Generating Self-Affine Tiles and Their Boundaries," *The Mathematica Journal*, 2011. dx.doi.org/doi:10.3888/tmj.11.1–1.

About the Author

Mark McClure, associate professor of mathematics at the University of North Carolina at Asheville, was introduced to *Mathematica* in 1990 while teaching in the Calculus&*Mathematica* program as a graduate student at Ohio State University. His primary research is in fractal geometry, particularly as it arises in real analysis. The Mathematical Graphics column grew out of his desire to make high-level mathematics accessible via the use of computer graphics. In addition to mathematics, he enjoys hiking and biking in the mountains of western North Carolina.