9

Computer Graphics Transformations

Why Study Computer Graphics Transformations?

- **They define how graphical objects move, rotate, and scale**. Transformations give a precise mathematical language for changing an object's position, size, and orientation, which is essential for animation and modelling. Projective transformations are necessary to convert 3D scenes into 2D images.
- **They allow consistent, efficient representations of complex scenes**. Using matrices to express transformations makes it possible to combine operations cleanly. Also, matrix and vector operations are fast on modern hardware.
- **They enable different coordinate systems to work together**. Camera coordinates, world coordinates, object coordinates, and screen coordinates are all linked by transformations.
- **They are the foundation for advanced techniques**. Texture mapping, skeletal animation, ray tracing, collision detection, and physics simulations all depend on transformation mathematics.

9.1 Introduction

This chapter looks at how to employ 2D and 3D geometrical transformations, such as translation, scaling, and rotation, in computer graphics.

For the most part, we'll rely on two Python classes, `HC2Mat.py` and `HC3Mat.py`, and render shapes using Matplotlib's polygons. We'll also use Matplotlib's Affine2D class, which covers the same ground as our `HC2Mat.py` but can manipulate images and SVG drawings.

Computer graphics rests atop linear algebra due to its use of vectors, matrices, and linear, affine, and projective transformations. A good introduction can be found in *Elementary Linear Algebra* by Stephen Andrilli and David Hecker [**AH22**], and some of my examples come from their discussion of computer graphics. A truly excellent set of short videos on the subject is *Essence of linear algebra* by Grant Sanderson (3Blue1Brown) at `https://www.youtube.com/playlist?list=PLZHQObOWTQDPD3MizzM2xVFitgF8hE_ab`. A good textbook focusing directly on graphics is *Computer Graphics: Principles and Practice*, by John Hughes *et al.* [**Hug14**].

9.2 2D Transformations

We can translate points in the (x, y) plane to new positions by adding translation amounts:

$$x' = x + d_x, \quad y' = y + d_y.$$

If we define the column vectors:

$$P = \begin{bmatrix} x \\ y \end{bmatrix}, \quad P' = \begin{bmatrix} x' \\ y' \end{bmatrix}, \quad T = \begin{bmatrix} d_x \\ d_y \end{bmatrix}$$

then the translation can be succinctly expressed as $P' = P + T$.

Points can be scaled by s_x along the x-axis, and by s_y along the y-axis, with the multiplications:

$$x' = s_x \cdot x, \quad y' = s_y \cdot y.$$

In matrix form:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \quad \text{or} \quad P' = S \cdot P.$$

where S is the scaling matrix. Note that this defines a scaling about the origin; techniques for scaling about other points are discussed below.

Points can be rotated through an angle θ about the origin with

$$x' = x \cdot \cos\theta - y \cdot \sin\theta, \quad y' = x \cdot \sin\theta + y \cdot \cos\theta.$$

Using matrices:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \quad \text{or} \quad P' = R \cdot P$$

where R is the rotation matrix. Positive angles are measured counterclockwise from the x-axis toward the y-axis. For negative (clockwise) angles, the identities $\cos(-\theta) = \cos\theta$ and $\sin(-\theta) = -\sin\theta$ can be used.

The rotation equation is easily derived from Fig. 9.1, in which a rotation by θ transforms $P(x, y)$ into $P'(x', y')$. Because the rotation is about the origin, the distances from the origin to P and to P', labeled r in the figure, are equal.

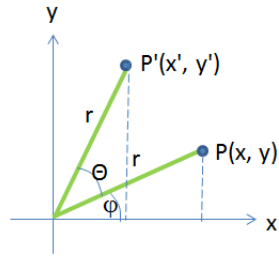

Figure 9.1. Derivation of the rotation equation

By trigonometry,

$$x = r \cdot \cos\phi, \quad y = r \cdot \sin\phi$$

With the angle sum rule, x' and y' are:

$$\begin{aligned} x' &= r \cdot \cos(\theta + \phi) = r \cdot \cos\phi \cdot \cos\theta - r \cdot \sin\phi \cdot \sin\theta, \\ y' &= r \cdot \sin(\theta + \phi) = r \cdot \cos\phi \cdot \sin\theta + r \sin\phi \cdot \cos\theta \end{aligned}$$

These equations can be rewritten in terms of x and y:

$$\begin{aligned} x' &= x \cdot \cos\theta - y \cdot \sin\theta, \\ y' &= x \cdot \sin\theta + y \cdot \cos\theta \end{aligned}$$

which can be expressed in the matrix form from above.

9.3 Homogeneous Coordinates

The matrix representations for translation, scaling, and rotation from the last section are:

$$P' = T + P, \quad P' = S \cdot P, \quad P' = R \cdot P.$$

Translation is treated differently from scaling and rotation, but we'd prefer to express all three in a consistent manner so that transformations can be combined more easily.

One answer is to define points using *homogeneous coordinates*, which enables every transformations to be defined using matrix multiplication alone. A point (x, y) becomes (x, y, W), and we assume that (x, y, W) and (x', y', W') represent the same point when one is a multiple of the other (e.g. $(2, 3, 6)$ and $(4, 6, 12)$ are the same). If W is nonzero, we divide through by its value, so (x, y, W) becomes $(x/W, y/W, 1)$, and $(x/W, y/W)$ denotes the point in 2D space. However,

if $W = 0$ then the point is deemed to be located at infinity, which is useful for projective transformations such as perspectives and warping.

Now that 2D points are triplets, our transformation matrices become 3×3. This allows translation to be encoded using a single matrix:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & d_x \\ 0 & 0 & d_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

This can be expressed as

$$P' = T(d_x, d_y) \cdot P$$

where

$$T(d_x, d_y) = \begin{bmatrix} 1 & 0 & d_x \\ 0 & 0 & d_y \\ 0 & 0 & 1 \end{bmatrix}$$

Multiplying out the matrix produces the desired translations:

$$x' = x + d_x, \quad y' = y + d_y.$$

What happens if a point P is translated by $T(d_{x_1}, d_{y_1})$ to P' and then by $T(d_{x_2}, d_{y_2})$ to P''? We expect a combined translation $T(d_{x_1} + d_{x_2}, d_{y_1} + d_{y_2})$. Initially:

$$\begin{aligned} P' &= T(d_{x_1}, d_{y_1},) \cdot P \\ P'' &= T(d_{x_2}, d_{y_2}) \cdot P' \end{aligned}$$

Substituting the first equation into the second, we obtain:

$$\begin{aligned} P'' &= T(d_{x_1}, d_{y_1}) \cdot (T(d_{x_2}, d_{y_2}) \cdot P) \\ &= (T(d_{x_2}, d_{y_2}) \cdot T(d_{x_1}, d_{y_1})) \cdot P \end{aligned}$$

The matrix product for $T(d_{x_2}, d_{y_2}) \cdot T(d_{x_1}, d_{y_1})$ is

$$\begin{bmatrix} 1 & 0 & d_{x_2} \\ 0 & 0 & d_{y_2} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & d_{x_1} \\ 0 & 0 & d_{y_1} \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & d_{x_1} + d_{x_2} \\ 0 & 0 & d_{y_1} + d_{y_2} \\ 0 & 0 & 1 \end{bmatrix}.$$

This shows that the net translation is indeed $T(d_{x_1} + d_{x_2}, d_{y_1} + d_{y_2})$.

Scaling is represented by

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

If we define

$$S(s_x, s_y) = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

then

$$P' = S(s_x, s_y) \cdot P.$$

Let's check that successive scalings are multiplicative. Given

$$\begin{aligned} P' &= S(s_{x_1}, s_{y_1}) \cdot P \\ P'' &= S(s_{x_2}, s_{y_2}) \cdot P' \end{aligned}$$

then substituting the first equation into the second, we get

$$\begin{aligned} P'' &= S(s_{x_2}, s_{y_2}) \cdot (S(s_{x_1}, s_{y_1}) \cdot P) \\ &= (S(s_{x_2}, s_{y_2}) \cdot S(s_{x_1}, s_{y_1})) \cdot P \end{aligned}$$

The matrix product $S(s_{x_2}, s_{y_2}) \cdot S(s_{x_1}, s_{y_1})$ is

$$\begin{bmatrix} s_{x_2} & 0 & 0 \\ 0 & s_{y_2} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s_{x_1} & 0 & 0 \\ 0 & s_{y_1} & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s_{x_2} \cdot s_{x_1} & 0 & 0 \\ 0 & s_{y_2} \cdot s_{y_1} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Thus, scalings are multiplicative.

Rotations become

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

If we define

$$R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

then

$$P' = R(\theta) \cdot P.$$

Consecutive rotations are encoded by matrix multiplication, but the combined result *adds* the rotations. This can be proved using similar angle sum rewrites as employed for the rotations in the previous subsection.

Shearing is another common transformation, either along the x- or y-axis. The x-axis operation is represented by

$$SH_x = \begin{bmatrix} 1 & a & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The product $SH_x(x, y)$ is $[x + ay, y, 1]^T$, demonstrating the proportional change in x as a function of y.

Similarly,

$$SH_y = \begin{bmatrix} 1 & 0 & 0 \\ b & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

shears along the y-axis.

Reflection about a line through the origin (i.e. by using $y = mx$) becomes the matrix

$$\frac{1}{1+m^2}\begin{bmatrix} 1-m^2 & 2m & 0 \\ 2m & m^2-1 & 0 \\ 0 & 0 & 1+m^2 \end{bmatrix}$$

This can be understood by assuming that the line forms an angle θ with the x-axis. Then the reflection can be divided into three steps:

(1) rotate the line by $-\theta$ to make it horizontal;

(2) invert the y coordinate;

(3) rotate the line back by θ.

These three operations become rotation, inversion, and rotation matrices composed using multiplication. Further simplifications are possible by noting that since $y = mx$, then $\tan\theta = m$, and so $1 + m^2 = 1/\cos^2\theta$.

The reflection transformation:

$$\begin{aligned} F &= \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} \cos\theta & \sin\theta & 0 \\ \sin\theta & -\cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ &= \cos^2\theta\begin{bmatrix} 1 & -\tan\theta & 0 \\ \tan\theta & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & \tan\theta & 0 \\ \tan\theta & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ &= \frac{1}{1+m^2}\begin{bmatrix} 1 & -m & 0 \\ m & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & m & 0 \\ m & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ &= \frac{1}{1+m^2}\begin{bmatrix} 1-m^2 & 2m & 0 \\ 2m & m^2-1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{aligned}$$

Note that the matrices are organized in a right-to-left order since we're mimicking function composition.

In the case when the line of reflection is the y-axis, the matrix reduces to

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Do you see why?

9.4 Composition of 2D Transformations

Consider the rotation of a shape about some arbitrary point $P_1(x_1, y_1)$. R assumes that any rotation is about the origin, so we must bracket the rotation with suitable translations:

(1) Translate P_1 to the origin (and move the shape by the same amount);

(2) Rotate the shape about the origin using R;

(3) Translate the point at the origin back to P_1 (and move the rotated shape by the same amount).

The translation to the origin will be by $(-x_1, -y_1)$, and the translation back (x_1, y_1). The net transformation, performed in right-to-left order, is:

$T(x_1, y_1) \cdot R(\theta) \cdot T(-x_1, -y_1)$

$$\begin{aligned} &= \begin{bmatrix} 1 & 0 & x_1 \\ 0 & 1 & y_1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -x_1 \\ 0 & 1 & -y_1 \\ 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} \cos\theta & -\sin\theta & x_1(1-\cos\theta) + y_1 \sin\theta \\ \sin\theta & \cos\theta & y_1(1-\cos\theta) - x_1 \sin\theta \\ 0 & 0 & 1 \end{bmatrix} \end{aligned}$$

A similar translation-bracketing approach is needed to scale an object about an arbitrary point: translate P_1 to the origin (along with the shape), then scale, then translate back to P_1 (along with the scaled shape). The net transformation is:

$T(x_1, y_1) \cdot S(s_x, s_y) \cdot T(-x_1, -y_1)$

$$= \begin{bmatrix} 1 & 0 & x_1 \\ 0 & 1 & y_1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -x_1 \\ 0 & 1 & -y_1 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & x_1(1-s_x) \\ 0 & s_y & y_1(1-s_y) \\ 0 & 0 & 1 \end{bmatrix}$$

Suppose we want to scale *then* rotate a shape about P_1? Crucially, we only need to utilize translation-bracketing once, so the composite matrix is made up of *four* operations performed in a right-to-left order:

$$T(x_1, y_1) \cdot R(\theta) \cdot S(s_x, s_y) \cdot T(-x_1, -y_1).$$

9.5 Transforming a Knee

Let's illustrate how to manipulate the two-dimensional figure (a "Knee") shown in Fig. 9.2 using transformations implemented in Python. (The examples in this section (but not the code) comes from *Elementary Linear Algebra* by Stephen Andrilli and David Hecker [**AH22**].)

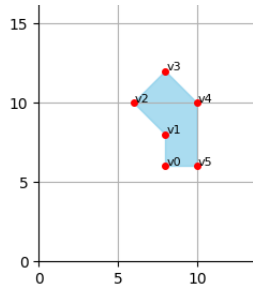

Figure 9.2. The "Knee" Polygon

The shape is encoded as a list of tuples:

```
coords = [(8,6),(8,8),(6,10),(8,12),(10,10),(10,6)]
```

Each tuple is a pair of x- and y-coordinates, but is converted into a column vector using homogeneous coordinate ($[x, y, 1]^T$) while the shape is being transformed, which means that the matrices are structured as in the previous section. The results are converted back to a list of tuples before they're rendered, which is done by converting them into a Matplotlib polygon.

Fig. 9.2 was generated using the following lines in `transformPoly.py`:

```
_, ax = plt.subplots()

coords = [(8,6),(8,8),(6,10),(8,12),(10,10),(10,6)]

m = identMat()  # the identity matrix; a 'do-nothing' op.
printMat(m)
nCoords = [applyMat(m, c) for c in coords]

plotPoly(ax, nCoords, 'skyblue')
plt.show()
```

identMat(), printMat(), applyMat(), and plotPoly() are imported from `HC2Mat.py` which collects functions for representing 2D homogeneous coordinates and their transformations. For example, identMat() returns an identity matrix:

```
def identMat():
  return [ [1, 0, 0],
```

```
          [0, 1, 0],
          [0, 0, 1] ]
```

plotPoly() plots a list of coordinates as a closed polygon and labels the vertices.

```
def plotPoly(ax, coords, color):
  poly = Polygon(coords, closed=True, color=color, alpha=0.7)
  ax.add_patch(poly)
  # Label each vertex
  for i, (x, y) in enumerate(coords):
    ax.plot(x, y, 'o', color='red', ms=4)  # small red dot
    ax.text(x + 0.05, y + 0.05, f'v{i}',
                    fontsize=8, color='black')
```

`Polygon()` is described at `https://matplotlib.org/stable/api/patches_api.html`.

applyMat() applies a transformation to a list of coordinates by multiplying the matrix to each coordinate after converting it to the vector $[x, y, 1]^T$:

```
def applyMat(matrix, coord):
  # apply the matrix to a (x,y) coord, returning
  # a new (x,y) coord
  vec = [coord[0], coord[1], 1]  # homogeneous
  nVec = [0, 0, 0]
  for i in range(3):
    for k in range(3):
      nVec[i] += matrix[i][k] * vec[k]
  return (nVec[0]/nVec[2], nVec[1]/nVec[2])
```

Rotation. Suppose we rotate "Knee" through an angle of 90° about the point (12, 6). We replace each (x, y) with its vector $[x, y, 1]^T$, then translate from (12, 6) to (0, 0) in order to center the point at the origin. We perform a rotation through 90° about the origin, then translate back to (12, 6). The net effect of these operations is

$$\underbrace{\begin{bmatrix} 1 & 0 & 12 \\ 0 & 1 & 6 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{translate from (0,0) back to (12,6)}} \underbrace{\begin{bmatrix} \cos 90° & -\sin 90° & 0 \\ \sin 90° & \cos 90° & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{rotate about (0,0) through angle 90°}} \underbrace{\begin{bmatrix} 1 & 0 & -12 \\ 0 & 1 & -6 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{translate from (12,6) to (0,0)}} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

This reduces to

$$\begin{bmatrix} 0 & -1 & 18 \\ 0 & 1 & -6 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

We apply this transformation to all the vertices of the "Knee" to obtain

$$\begin{bmatrix} 0 & -1 & 18 \\ 0 & 1 & -6 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 8 & 8 & 6 & 8 & 10 & 10 \\ 6 & 8 & 10 & 12 & 10 & 6 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 12 & 10 & 8 & 6 & 8 & 12 \\ 2 & 2 & 0 & 2 & 4 & 4 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

The (x, y) coordinates are saved as a list of tuples, and drawn as a polygon in Fig. 9.3.

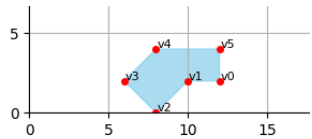

Figure 9.3. The "Knee" rotated through 90° about $(12, 6)$

In transformPoly.py, the transformation is encoded as:

```
m = multMatsList([ transMat(12,6), rotMat(90), transMat(-12,-6) ])
```

Note the right-to-left ordering of the list passed to multMatsList(). transMat() and rotMat() come from HC2Mat.py:

```
def transMat(a, b):
  # translate by 'a' along the x-axis, 'b' along y-axis
  return [ [1, 0, a],
           [0, 1, b],
           [0, 0, 1] ]

def rotMat(angle):
  # rotate by angle degrees counter-clockwise around origin
  theta = math.radians(angle)
  return [ [math.cos(theta), -math.sin(theta), 0],
           [math.sin(theta),  math.cos(theta), 0],
           [             0,                0,  1]  ]
```

Reflection. Let's reflect the original "Knee" about the line $y = -3x + 30$. In this case, $m = -3$, and the y intercept $b = 30$. As before, we replace (x, y) with its equivalent vector $[x, y, 1]^T$, then translate from $(0, 30)$ to $(0, 0)$ in order to lower the line 30 units vertically so its intercept passes through the origin. Now, we perform a reflection about the resulting line $y = -3x$, then translate back to $(0, 30)$. The net effect is to reflect each vertex of the "Knee"about the line $y = -3x + 30$ in three steps:

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 30 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{translate from (0,0) back to (0,30)}} \underbrace{\frac{1}{1+(-3)^2}\begin{bmatrix} 1-(-3)^2 & 2(-3) & 0 \\ 2(-3) & (-3)^2-1 & 0 \\ 0 & 0 & 1+(-3)^2 \end{bmatrix}}_{\text{reflect about the line } y=-3x} \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -30 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{translate from (0,30) to (0,0)}} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

This reduces to

$$\frac{1}{10}\begin{bmatrix} -8 & -6 & 180 \\ -6 & 8 & 60 \\ 0 & 0 & 10 \end{bmatrix}\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Applying this matrix to all the vertices of "Knee" produces

$$\frac{1}{10}\begin{bmatrix} -8 & -6 & 180 \\ -6 & 8 & 60 \\ 0 & 0 & 10 \end{bmatrix}\begin{bmatrix} 8 & 8 & 6 & 8 & 10 & 10 \\ 6 & 8 & 10 & 12 & 10 & 6 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 8 & 6.8 & 7.2 & 4.4 & 4 & 6.4 \\ 6 & 7.6 & 10.4 & 10.8 & 8 & 4.8 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

The (x, y) coordinates are drawn as a polygon in Fig. 9.4.

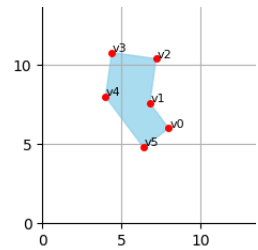

Figure 9.4. "Knee" reflection about line $y = -3x + 30$

The transformation as coded in `transformPoly.py` as:

```
m = multMatsList([ transMat(0,30), reflectMat(-3), transMat(0,-30) ])
```

reflectMat() in `HC2Mat.py` is:

```
def reflectMat(m):
  # reflect about line y = mx
  den = 1 + m*m
  return [ [(1-m*m)/den,     2*m/den,  0],
           [    2*m/den, (m*m-1)/den,  0],
           [          0,           0,  1] ]
```

Scaling. Let's scale "Knee" about the point $(6, 10)$ with a scaling factor of $c = 1/2$ in the x-direction and $d = 4$ in the y-direction. As before, the resizing must be bracketed by translations:

$$\underbrace{\begin{bmatrix} 1 & 0 & 6 \\ 0 & 1 & 10 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{translate from (0,0) back to (6,10)}} \underbrace{\begin{bmatrix} 1/2 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{scale about (0,0) using scale factors } \frac{1}{2} \text{ and 4, respectively}} \underbrace{\begin{bmatrix} 1 & 0 & -6 \\ 0 & 1 & -10 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{translate from (6,10) to (0,0)}} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} 1/2 & 0 & 3 \\ 0 & 4 & -30 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

The scaling of "Knee" produces

$$\begin{bmatrix} \frac{1}{2} & 0 & 3 \\ 0 & 4 & -30 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 8 & 8 & 6 & 8 & 10 & 10 \\ 6 & 8 & 10 & 12 & 10 & 6 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 7 & 7 & 6 & 7 & 8 & 8 \\ -6 & 2 & 10 & 18 & 10 & -6 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

The polygon built from the (x, y) coordinates is shown in Fig. 9.5. Two of the vertices have negative y-values, so aren't in the region displayed by Matplotlib.

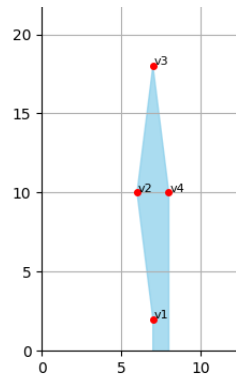

Figure 9.5. "Knee" scaling with c=1/2, d=4 about (6, 10)

The transformation in transformPoly.py is:

```
m = multMatsList([ transMat(6,10), scaleMat(0.5,4), transMat(-6,-10) ])
```

scaleMat() in HC2Mat.py:

```
def scaleMat(a, b):
  # scale by 'a' along the x-axis, 'b' along y-axis
  return [ [a, 0, 0],
           [0, b, 0],
           [0, 0, 1] ]
```

9.6 Multiple Transformations

As a grand finale, let's rotate "Knee" through 300° about (8, 10), then reflect it about the line $y = -(1/2)x + 20$. The key values are $\theta = 300°, m = -1/2$, and $b = 20$, and the composition utilizes six matrices:

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 20 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{translate from (0,0) back to (0,20)}} \underbrace{\left(\frac{1}{1+(-1/2)^2}\right)\begin{bmatrix} 1-(-1/2)^2 & 2(-1/2) & 0 \\ 2(-1/2) & (-1/2)^2-1 & 0 \\ 0 & 0 & 1+(-1/2)^2 \end{bmatrix}}_{\text{reflect about the line } y=(-1/2)x} \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -20 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{translate from (0,20) to (0,0)}}$$

$$\underbrace{\begin{bmatrix} 1 & 0 & 8 \\ 0 & 1 & 10 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{translate from (0,0) back to (8,10)}} \underbrace{\begin{bmatrix} \cos 300° & -\sin 300° & 0 \\ \sin 300° & \cos 300° & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{rotate about (0,0) through angle 300°}} \underbrace{\begin{bmatrix} 1 & 0 & -8 \\ 0 & 1 & -10 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{translate from (8,10) to (0,0)}}$$

The first three matrices deal with the second part of the transformation – the reflection in a line not through the origin, while the last three are for the rotation around a point. The composition reduces to (approximately)

$$\begin{bmatrix} 0.993 & 0.120 & 3.661 \\ 0.120 & -0.993 & 28.571 \\ 0 & 0 & 1 \end{bmatrix}$$

Applying this matrix to "Knee" produces

$$\begin{bmatrix} 0.993 & 0.120 & 3.661 \\ 0.120 & -0.993 & 28.571 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 8 & 8 & 6 & 8 & 10 & 10 \\ 6 & 8 & 10 & 12 & 10 & 6 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 12.32 & 12.56 & 10.81 & 13.04 & 14.79 & 14.31 \\ 23.57 & 21.59 & 19.36 & 17.61 & 19.84 & 23.81 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Its coordinates are shown in Fig. 9.6.

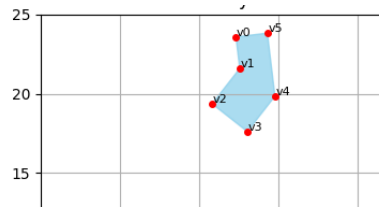

Figure 9.6. "Knee" rotation then reflection

The relevant line in `transformPoly.py`:

```
m = multMatsList( [
      transMat(0,20), reflectMat(-0.5), transMat(0,-20),
      transMat(8,10), rotMat(300), transMat(-8,-10) ])
```

9.7 Baravelle Spirals

A Baravelle spiral is a geometric shape constructed by subdividing a large regular polygon into triangles. The construction bisects each of the sides and joins the midpoints consecutively to create the triangles, leaving a smaller version of the polygon inside. The procedure is repeated, and the spirals are formed from adjacent nested triangles turning counter-clockwise, which shrink and converge upon the polygon's center. The National Curve Bank (`https://old.nationalcurvebank.org/baravelle/baravelle.htm`) has an online tool for generating Baravelle spirals for several types of regular polygon, as shown in Fig. 9.7.

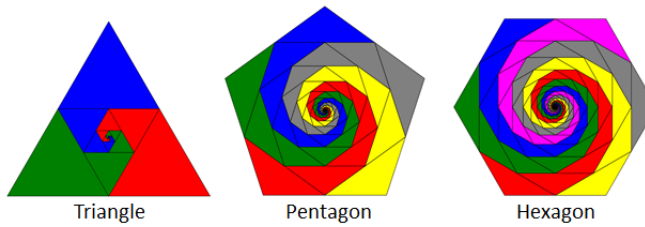

Figure 9.7. Baravelle Spirals

For further details, the paper by Suzanne Harper explains how these curves can be used to introduce infinite series in secondary mathematics [**Har01**].

baravelle.py generates baravelle spirals starting from a large square, as shown in Fig. 9.8.

Figure 9.8. Baravelle Spirals for a Square

The code begins by placing four copies of a right-angled triangle at the corners of a square, as in Fig. 9.9.

The code for this stage is in createColorCoords():

```
def createColorCoords():
  coords = [ (0,0), (SIZE,0), (SIZE,SIZE)] # basic triangle
  ''' set start positions of triangles at the
    four corners of a 2*SIZE square centered at origin
  '''
```

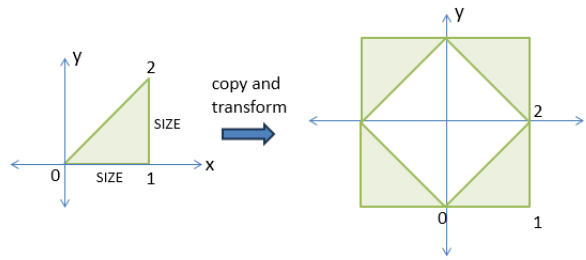

Figure 9.9. Initializing the Baravelle Square

```
m = multMatsList([ transMat(SIZE,0), rotMat(90) ]) # top right
coords0 = [applyMat(m, c) for c in coords]

m = multMatsList([ transMat(0,SIZE), rotMat(180) ]) # top left
coords1 = [applyMat(m, c) for c in coords]

m = multMatsList([ transMat(-SIZE,0), rotMat(270) ]) # bottom left
coords2 = [applyMat(m, c) for c in coords]

m = multMatsList([ transMat(0,-SIZE) ])   # bottom right
coords3 = [applyMat(m, c) for c in coords]

# specify a fill color for each triangle
return  [(coords0,'skyblue'), (coords1,'lightgreen'),
         (coords2, 'violet'), (coords3, 'salmon')]
```

Spiral generation is carried out by repeatedly shrinking, rotating, and moving a triangle to the midpoint of the diagonal of its parent (see Fig. 9.10).

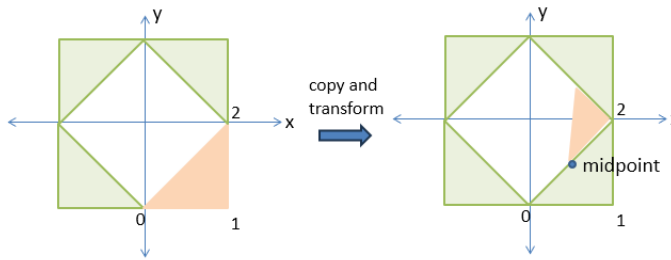

Figure 9.10. Shrink, Rotate, and Move Triangle to the Midpoint

This could be expressed recursively, but is just as simple to write iteratively:

```
cCoords = createColorCoords()
for coords, color in cCoords:
  drawPoly(ax, coords, color)
  for i  in range(STEPS):
    coords = moveIn(coords)
    drawPoly(ax, coords, color)
```

moveIn() scales, rotates, and moves a triangle:

```
def moveIn(coords):
  pt0 = coords[0]
  nPt0 = tuple((a+b)/2 for a, b in zip(coords[0], coords[2]))
         # midpoint of diagonal
  m = multMatsList([ transMat(nPt0[0], nPt0[1]),
         rotMat(ANGLE), scaleMat(SCALE, SCALE),
         transMat(-pt0[0], -pt0[1])  ])
  return [applyMat(m, c) for c in coords]
```

The bracketed translation in moveIn() is a little different from before. The triangle is moved back to the origin by using its first vertex (pt0), but is positioned after scaling and rotation at the midpoint of its parent's diagonal (nPt0).

9.8 Hilbert Curves Again

We've already met the Hilbert curve family in Section 7.8 on fractals. The approach used there was based around drawing a line using turtle operations such as Turtle.forward() and Turtle.setHeading(), and by calling four inter-linked recursive functions for drawing the U_i, D_i, R_i, and L_i move sequences. However, a more intuitive way to view these moves is as rotations applied to a shape represent H_0 (Fig. 9.11). (Note that H_0 is a "U" shape; the arrowhead was added by me to indicate the drawing order – the drawing begins at the top of the right arm then moves clockwise to the top of the left arm.)

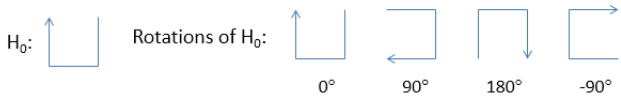

Figure 9.11. Four Rotations of H_0

As in the fractals section, H_i is created by utilizing four versions of H_{i-1} composed together. However, they're modified using transformations rather than with turtle functions. The necessary steps to convert four copies of H_0 into H_1 are shown in Fig. 9.12.

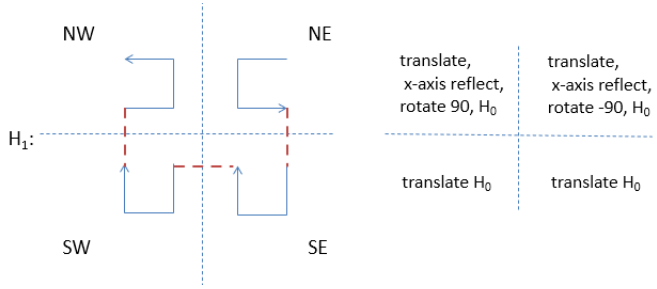

Figure 9.12. Four H_0s Transformed into H_1

The transformations in the four quadrants of Fig. 9.12 are applied to H_0 in the usual right-to-left order, producing four rotated and translated copies. They are linked by the dashed lines in the figure to produce H_1.

You should try out these same four transformations on H_1 to confirm that they generate H_2. Indeed, these transformations remain the same for any H_i produced from H_{i-1}, and so are conveniently collected together in transformCurve() in hilbert.py:

```
L = 50   # length of one side of H0

def transformCurve(h):
  # Use hilbert curve h to generate next one

  # build parts in 4 quadrants
  m = multMatsList([ transMat(L,L),
                reflectXMat(), rotMat(-90) ])
  ne = [applyMat(m, c) for c in h]  # north east quadrant

  se = [applyMat(transMat(L,-L), c) for c in h]

  sw = [applyMat(transMat(-L,-L), c) for c in h]

  m = multMatsList([ transMat(-L,L),
                reflectXMat(), rotMat(90) ])
  nw = [applyMat(m, c) for c in h]

  hNew = ne + se + sw + nw # concatenate parts
  return [applyMat(scaleMat(0.5,0.5), c) for c in hNew]
```

The composition of the four shapes, which is denoted by the three dashed lines in Fig. 9.12, requires nothing more than list concatenation since the lines are stored as lists of tuples. However, care must be taken that the lists are in the correct drawing order, which requires the NW and NE shapes to be reflected in the x-axis.

The top-level of the program could use recursion, but (as with the baravelle spirals) iteration is just as simple:

```
h = [ (L/2,L/2), (L/2,-L/2), (-L/2,-L/2), (-L/2,L/2)]
      # H0 centered on origin (a U-shape)
for i in range(hOrder-1):
  h = transformCurve(h)
```

9.9 Matplotlib's Affine2D Classes

The drawing of a shape by Matplotlib is achieved by converting its list of coordinates into a polygon (`https://matplotlib.org/stable/api/patches_api.html`). The relevant lines in plotPoly() in HC2Mat.py are:

```
def plotPoly(ax, coords, color):
  poly = Polygon(coords, closed=True, color=color, alpha=0.7)
  ax.add_patch(poly)

  # Label each vertex
  #  :
```

It's not surprising that Matplotlib also offers geometric transformations that can be applied to shapes via subclassses of matplotlib.transform (see `https://matplotlib.org/stable/api/transformations.html`). To keep things simple, we'll focus on the Affine2D class (`https://matplotlib.org/stable/api/transformations.html#matplotlib.transforms.Affine2D`) which defines its transformations in terms of 2D homogeneous coordinates (i.e. as 3×3 matrices). The matrices are implemented using Numpy arrays, but it's possible to ignore this complication when using the class, which includes methods such as translate(), rotate(), scale(), and skew() (but not reflection).

The earlier "Knee" examples were implemented in transformPoly.py. To aid comparison, we've reimplemented them using Affine2D transformations in affinePoly.py. The "Knee"'s vertices are created in the same way as before, but transformations are applied to its polygon:

```
coords = [(8,6),(8,8),(6,10),(8,12),(10,10),(10,6)]
poly = Polygon(coords, closed=True, color='skyblue', alpha=0.7)
```

Note that the coordinates don't require W homogeneous values, but they are added by Affine2D when transformations are applied.

The last example in transformPoly.py rotates "Knee" 300° around $(8, 10)$, then reflects it around the line $y = -0.5x + 20$. The resulting matrix is:

```
m = multMatsList( [
      transMat(0,20), reflectMat(-0.5), transMat(0,-20),
      transMat(8,10), rotMat(300), transMat(-8,-10) ])
```

The corresponding transformation using Affine2D in affinePoly.py is:

```
trfm = Affine2D().translate(-8,-10). \
            rotate_deg(300).translate(8,10) + \
       Affine2D().translate(0,-20) + \
       reflect(-0.5) + \
       Affine2D().translate(0,20)
```

A key difference is that Affine2D transformations are performed in left-to-right order. Also, although Affine2D allows the chaining of operations using ".", this is not possible when using reflection. Affine2D doesn't include a reflection method, but one can be defined manually, as in our reflect():

```
def reflect(m):
  ''' Build reflection Affine2D instance using the matrix
        a c e
        b d f
        0 0 1
      with from_values(a,b,c,d,e,f)
  '''
  den = 1 + m*m
  return Affine2D.from_values(
          (1-m*m)/den, 2*m/den, 2*m/den, (m*m-1)/den, 0, 0)
```

Sadly, the matrix returned by reflect() can not be chained, but Affine2D operations can also be composed using "+", which we used to create `trfm` above.

Drawing the polygon in affinePoly.py is similar to transformPoly.py:

```
poly.set_transform(trfm + ax.transData) # add axis offset
ax.add_patch(poly)
```

One new thing is to explicitly add a transformation so the shape is correctly positioned relative to the axes. The output is shown in Fig. 9.13, and should be compared to the image produced by transformPoly.py in Fig. 9.6.

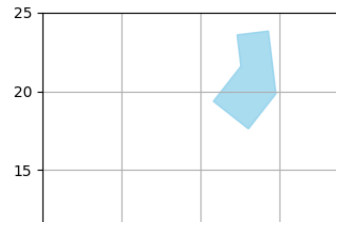

Figure 9.13. "Knee" rotation then reflection Using Affine2D

One reason for preferring Affine2D over our HC2Mat.py is efficiency: Affine2D utilizes Numpy arrays for its matrices rather than 2D lists, and so will be faster

when transforming shapes with many coordinates. Two other reasons for preferring Affine2D is that it also works with images and SVG (Scalable Vector Graphics) polylines, as detailed in the next three subsections.

9.10 Affine2D Images

imPlot.py loads a picture of a cat, displays it and a copy that is reflected, rotated, and translated (see Fig. 9.14).

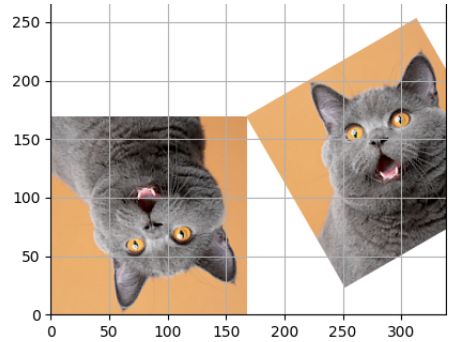

Figure 9.14. Two Cats, one Transformed

The cat on the left is the original image, which is shown with its origin at the *top-left* corner and its y-axis increasing downwards – the standard way that PNG images are encoded. The relevant code snippets in implot.py are:

```
_, ax = plt.subplots()

im = mpimg.imread('cat.png')  # load image
height, width = im.shape[0], im.shape[1]
    :  # more code
ax.imshow(im)
```

The image on the right of Fig. 9.14 has been reflected in the x-axis, rotated, and translated. The relevant lines are:

```
show2 = ax.imshow(im.copy())   # show first, then transform
trans = Affine2D().scale(1, -1). \
                   rotate_deg(30). \
                   translate(width, height)
show2.set_transform(trans + ax.transData)
```

There are two things of note: the first is that we didn't use our Affine2D reflect() function to reflect the image but got by with a 'tricky' use of scaling. A second point is that it's necessary to call imshow() on the image *before* applying the transformation.

9.11 Affine2D Paths

A good visual overview of Matplotlib shapes can be found at `https://matplotlib.org/stable/api/patches_api.html`. A particularly powerful type is PathPatch (`https://matplotlib.org/stable/api/_as_gen/matplotlib.patches.PathPatch.html#matplotlib.patches.PathPatch`) which converts a Path into a renderable shape (a patch). A path is more general than just a list of coordinates; it can be a series of possibly disconnected, possibly closed, line and/or curve segments. An introduction to creating path-based shapes can be found at `https://matplotlib.org/stable/users/explain/artists/paths.html`.

Another approach, better suited to drawing complicated shapes, is to use a SVG editor (e.g. the "Free SVG Editor" at `https://freesvgeditor.com/en/svg-editor-online`). The result is an SVG file (a text file using XML) which typically encodes multiple SVG paths. These are similar to paths in Matplotlib, but may be considerably more complex. Fortunately, there are parsing tools that can convert SVG paths into Matplotlib ones; we've used the third-party svgpath2mpl module which offers the parse_path() function (`https://github.com/nvictus/svgpath2mpl`). The result of parsing an SVG file will usually be multiple Matplotlib paths, which are converted into a single shape using the Matplotlib PathCollection class.

This approach is admittedly a bit complicated, so we've included sample code in ShowSVG.py. Fig. 9.15 shows two windows: the one on the left is a SVG viewer's display of house.svg which we quickly knocked together with the "Free SVG Editor". The window on the right shows the plot generated by ShowSVG.py when passed that file.

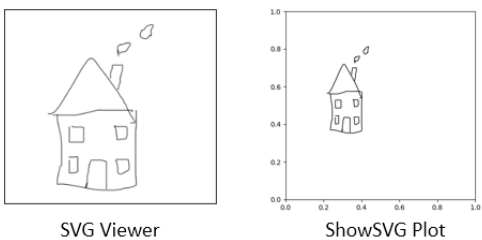

Figure 9.15. SVG and Matplotlib Houses

It's reasonable if you feel somewhat reluctant to follow this approach since SVG is a *very* sophisticated format, including features such as image compression, interactivity, and animation. The good news is that svgpath2mpl's parse_path()

will accept much simpler input, including a list of operations that specifies a single SVG "d" path. For example, in the text file f.path you'll find:

```
M .1, .1
L .9, .1
L .9, .2
L .2, .2
L .2, .4
L .6, .4
L .6, .5
L .2, .5
L .2, .9
L .1, .9
L .1, .1
```

Mozilla offers a good tutorial on SVG paths at `https://developer.mozilla.org/en-US/docs/Web/SVG/Tutorials/SVG_from_scratch/Paths`, but if you only know that 'M' and 'L' are short for "Move to" and "Line to", then you can probably already draw this 'F' shape on some graph paper.

We've put together sample code in ShowPath.py that loads a 'path' file, parses it, and plots it. The result for f.path is shown in Fig. 9.16.

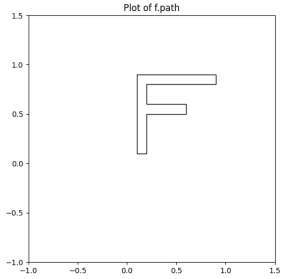

Figure 9.16. Plotting f.path

The relevant snippets of code in ShowPath.py are:

```
pathStr = open(fnm+'.path', 'r', encoding='utf-8').read()
path = parse_path(pathStr)
path = path.transformed(  # flip and move to first quadrant
                Affine2D().scale(1,-1).translate(0,1))

_, ax=plt.subplots(figsize=(6, 6))
ax.add_patch(PathPatch(path, fill=False))
```

The transformations are applied to the Matplotlib path and then PathPatch() convert it into a shape (a Matplotlib patch).

Paths generated using SVG tools utilize the same conventions for their coordinates as PNG images: the y-axis origin is at the top-left, and y-values increase downwards. This is remedied using similar transformations to those used on the 'cat' image. Scaling flips the path in the x-axis so its origin is at the bottom-left, and translation moves the path up from below the x-axis.

9.12 Henderson's Functional Geometry

Peter Henderson's functional geometry is an elegantly simple algebra for transforming and composing pictures based around operations such as rot(), flip(), above(), and beside(). Nevertheless, this small set of primitives are sufficiently powerful to draw the Escher woodcut, Square Limit (`https://www.wikiart.org/en/m-c-escher/square-limit`).

His ideas are presented in a classic paper (from 1982, revised in 2002) that beautifully illustrates the power of functional programming for data abstraction (`https://eprints.soton.ac.uk/257577/1/funcgeo2.pdf`. Another source is section 2.2.4 of Abelson and Sussman's magnificent *Structure and Interpretation of Computer Programs* [**ASS96**]. The second edition is freely available at `https://mitp-content-server.mit.edu/books/content/sectbyfn/books_pres_0/6515/sicp.zip/`.

Massimo Santini has implemented a version of Henderson's geometry in Python, using Matplotlib's patches, paths, and Affine2D (`https://mapio.github.io/programming-with-escher/`). He defines a Tile class which uses code similar to our ShowPath.py to load and convert a SVG-style path into a Matplotlib path, which is displayed as a PathPatch shape.

Henderson's transformations, such as rot() and flip(), are encoded using Affine2D, while compositional operations, such as above() and beside(), rely on Matplotlib's Path.make_compound_path(). As the name suggests, this takes two separate paths and packages them as a single compound path.

We've prepared a (slightly simplified) version of Santini's code in Tile.py. When run standalone, it generates a 2×4 grid of plots of various manipulations of the f.path shape (see Fig. 9.17).

Fig. 9.17 highlights that a tile always fits into a unit square even after several tiles have been combined.

squareLimit.py follows the steps outlined by Henderson to define Escher's 'Square Limit' woodcut. It requires the definition of two new functions, side() and corner(), which draw the sides and corners of the picture. It also employs nonet() which creates a 3×3 tile grid. With these, drawing Escher's image is a (long) one-liner:

```
def squarelimit(n):
  return nonet(   # 3 x 3 grid made of
```

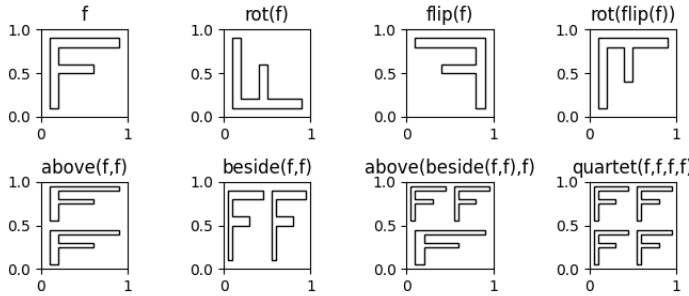

Figure 9.17. Tile Operations Upon "f"

```
    corner(n),              side(n),  rot(rot(rot(corner(n)))),
    rot(side(n)),               f4,     rot(rot(rot(side(n)))),
    rot(corner(n)), rot(rot(side(n))), rot(rot(corner(n)))
)
```

The resulting image is Fig. 9.18.

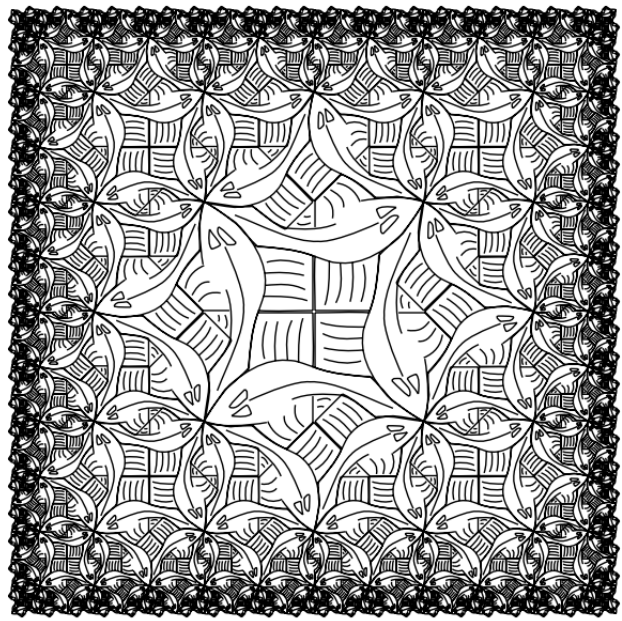

Figure 9.18. Tile Version of Escher's Square Limit

You can utilize Matplotlib's window controls to zoom in on a corner or side to see the tiling in more detail.

9.13 Hierarchy of Transformations

The mix of transformations supported by HC2Mat.py can be organized into the hierarchy shown in Fig. 9.19.

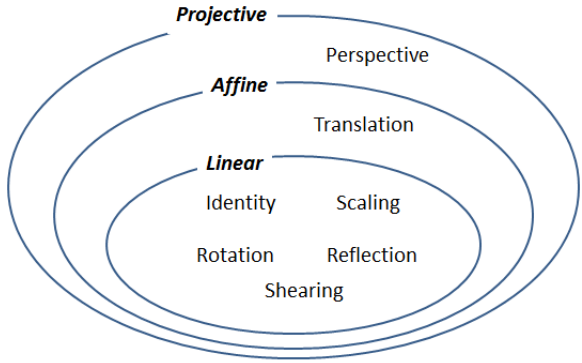

Figure 9.19. Transformation Hierarchy

The figure is a bit misleading since HC2Mat.py doesn't include any projective transformations, such as perspectives or warping. Even so, HC2Mat.py's use of homogeneous coordinates makes it simple to add such operations.

The following sections briefly describe the hierarchy, which helps to explain some of the choices when implementing and using HC2Mat.py transformations.

Linear Transformations. A linear transformation L is a mapping $V \rightarrow W$ between two vector spaces that preserves the operations of vector addition and scalar multiplication. In other words, for any two vectors $u, v \in V$ and any scalar c the following conditions are satisfied:

$$\begin{aligned} L(u+v) &= L(u)+L(v) \\ L(cu) &= cL(u) \end{aligned}$$

From these it follows that $L(0) = 0$, which means that linear transformations leave the vector space's origin unaffected.

A set B is a *basis* for a vector space V if its elements are independent, and can be linearly combined to represent every vector in V. For example, when V is the 2D space $\mathbb{R}^2$, it's natural to define B as the unit vectors along the axes: $\{[1,0]^T, [0,1]^T\}$. A suitable combination of these, with scaling, allows any vector in $\mathbb{R}^2$ to be defined. Similarly, a natural basis set for $\mathbb{R}^3$ are the unit vectors for its axes: $\{[1,0,0]^T, [0,1,0]^T, [0,0,1]^T\}$.

The basis for a space can be employed to define how any linear function affects the space. For instance, for $L : \mathbb{R}^3 \rightarrow \mathbb{R}^3$ we can define its behavior by

focusing on how the basis is changed:

$$\begin{aligned} L([1,0,0]^T) &= [2,-1,4]^T \\ L([0,1,0]^T) &= [1,5,-2]^T \\ L([0,0,1]^T) &= [0,3,1]^T \end{aligned}$$

Linearity makes it trivial to determine how L affects other vectors given this information. For example,

$$[2,-3,-2]^T = 2[1,0,0]^T + 3[0,1,0]^T - 2[0,0,1]^T$$

and so

$$\begin{aligned} L([2,3,-2]^T) &= 2L([1,0,0]^T) + 3L([0,1,0]^T) - 2L([0,0,1]^T) \\ &= 2[2,-1,4]^T + 3[1,5,-2]^T - 2[0,3,1]^T \\ &= [7,7,0]^T \end{aligned}$$

Another advantage of linearity is that any transformation can be encoded as a *standard* matrix that stores the transformations of its basis as column vectors. So,

$$L(\mathbf{v}) = A\mathbf{v} = \begin{bmatrix} 2 & 1 & 0 \\ -1 & 5 & 3 \\ 4 & -2 & 1 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

Now $L([2,3,-2]^T)$ can be expressed using the A matrix as:

$$L(\mathbf{v}) = A\mathbf{v} = \begin{bmatrix} 2 & 1 & 0 \\ -1 & 5 & 3 \\ 4 & -2 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 3 \\ -2 \end{bmatrix} = \begin{bmatrix} 7 \\ 7 \\ 0 \end{bmatrix}$$

Determining the standard matrix for a given transformation is simply a matter of applying the transformation to each of the basis vectors, and collecting the results. For example, if $L([x,y,z]^T) = [2x+y, 0, x+z]^T$, then the standard matrix is obtained by applying L to $[1,0,0]^T$, $[0,1,0]^T$, and $[0,0,1]^T$. For the x-axis:

$$L([1,0,0]^T) = [2,0,1]^T$$

and the complete matrix for L is:

$$A = \begin{bmatrix} 2 & 1 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

More generally, linearity makes it possible to manipulate an infinite n-dimensional space in terms of a basis with a finite number of elements (n). The matrix correspondence also lets us tap into a vast assortment of tools such as Gauss-Jordan elimination and the simplex algorithm.

Linear geometric operations include rotation, scaling, reflection, and shearing, but not translation which takes us to the next level in the hierarchy.

Affine Transformations. Although affine transformations, such as translation, don't leave the origin untouched, they do preserve other geometry properties, such as parallelism.

It can be shown that any affine transformation $A : U \to V$ can be written as $A(x) = L(x) + v_0$, where v_0 is some vector from V, and $L : U \to V$ is a linear transformation. In other words, any affine function is the composition of a linear function and a translation.

A very useful feature of linearity is that the composition of transformations (e.g. a scaling followed by a rotation) can be encoded using only matrix multiplication, but this fails if we define an affine operation as $A(x) = L(x) + v_0$. As we've seen above, it's much more convenient to switch to matrices that utilize homogeneous coordinates, and restrict $W = 1$:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

However, care must still be taken when combining translation with linear operations such as rotation and scaling since they assume that their centers of rotation/scaling are the origin. This explains the frequent appearance of translations bracketed around linear operations – to move a shape's center to and from the origin.

Projective Transformations. A projective transformation doesn't necessarily preserve parallelism, but collinearity and incidence are maintained (since lines are mapped to lines). Fig. 9.20 illustrates the effect.

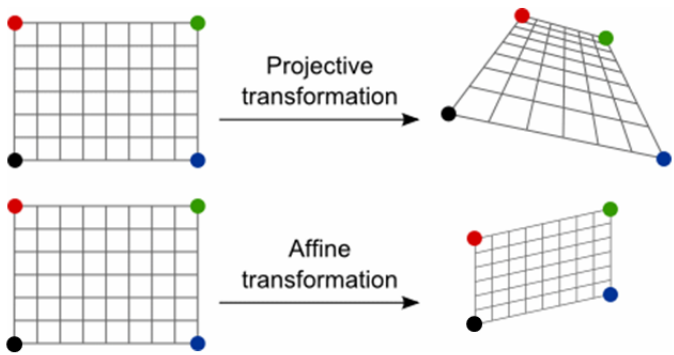

Figure 9.20. Projective vs. Affine

Homogeneous coordinates support projective transformations by allowing the last row of the matrix to contain any values:

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

9.14 3D Transformations as Matrices

Just as 2D transformations can be represented by 3×3 matrices using homogeneous coordinates, so 3D transformations can utilize 4×4 matrices. Instead of representing a point as (x, y, z), it becomes (x, y, z, W), with two quadruples denoting the same point if one is a nonzero multiple of the other. As in 2D, the standard representation of (x, y, z, W) with $W \neq 0$ is $(x/W, y/W, z/W, 1)$, and points whose W coordinate is zero are considered to be at infinity.

Translation in 3D:

$$T(d_x, d_y, d_z) = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

That is, $T(d_x, d_y, d_z) \cdot [x, y, z, 1]^T = [x + d_x, y + d_y, z + d_z, 1]^T$.

Scaling:

$$S(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

So $S(s_x, s_y, s_z) \cdot [x, y, z, 1]^T = [s_x \cdot x, s_y \cdot y, s_z \cdot z, 1]^T$.

Our earlier 2D rotations can be treated as 3D rotations about the z-axis:

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This is easily verified: a 90° rotation of the x-axis unit vector, $[1, 0, 0, 1]^T$, should produce the y-axis unit vector, $[0, 1, 0, 1]^T$. Evaluating

$$\begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

gives the predicted result.

X-axis rotation is

$$R_x(0) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Y-axis rotation:

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Any number of rotation, scaling, and translation matrices can be multiplied together, but the result always has the form

$$M = \begin{bmatrix} r11 & r12 & r13 & tx \\ r21 & r22 & r23 & ty \\ r31 & r32 & r33 & tz \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

The 3×3 upper-left submatrix contains the aggregate rotation and scaling, while the right-hand column holds the translation.

There are three 3D shear matrices, corresponding to the planes they work upon. The (x, y) shear along the z-axis is

$$SH_{xy}(sh_x, sh_y) = \begin{bmatrix} 1 & 0 & sh_x & 0 \\ 0 & 0 & sh_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Applying SH_{xy} to the point $[x, y, z, 1]^T$ produces $[x + sh_x \cdot z, y + sh_y \cdot z, z, 1]^T$. Shears along the x- and y-axes have a similar form.

Just as we packaged the functions for 2D transformations in HC2Mat.py, the 3D transformations are located in HC3Mat.py. For example, a rotation around the z-axis is defined by:

```
def rotZMat(angle):
  theta = math.radians(angle)
  return [ [math.cos(theta), -math.sin(theta), 0, 0],
           [math.sin(theta),  math.cos(theta), 0, 0],
           [             0,                0,  1, 0],
           [             0,                0,  0, 1]  ]
```

There are also support functions for 4×4 matrix multiplication, printing, and applying a matrix to a coordinate. For instance, multiplication:

```
def multMats(matrixA, matrixB):
  result = [[0 for _ in range(4)] for _ in range(4)]
```

```
  for i in range(4):
    for j in range(4):
      for k in range(4):
        result[i][j] += matrixA[i][k] * matrixB[k][j]
  return result
```

9.15 Composition of 3D Transformations

A desired transformation is obtained by composing primitive transformations such as T, R_x, R_y, and R_z. However, the linear transformations (rotation, scaling, shearing, and reflection) assume that the origin is at $(0, 0)$, and so, as in the 2D case, the composition of these operations will be frequently bracketed by translations. The first moves the shape's center or origin at P to the 3D space's origin. The final translation moves the transformed shape's center or origin back to P.

letter3D.py illustrates 3D transformation composition. An "R" shape (actually a box and a triangle) initially rests on its back on the XY plane, with the bottom, left corner at the origin (see Fig. 9.21).

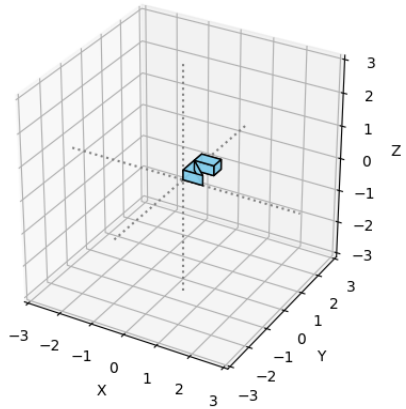

Figure 9.21. The Initial "R" Shape

The shape is defined using a list of vertices *and* a list of face indices which indicate how the coordinates are grouped to form the shape's faces:

```
# vertices for R shape
verts = [
  # Bottom (z = 0)
  (0.0, 0.0, 0.0),  # 0
  (0.0, 1.0, 0.0),  # 1
  (0.6, 1.0, 0.0),  # 2
```

```
  (0.6, 0.6, 0.0),  # 3
  (0.0, 0.6, 0.0),  # 4
  (0.6, 0.0, 0.0),  # 5

  # Top (z = 0.3)
  (0.0, 0.0, 0.3),  # 6
  (0.0, 1.0, 0.3),  # 7
  (0.6, 1.0, 0.3),  # 8
  (0.6, 0.6, 0.3),  # 9
  (0.0, 0.6, 0.3),  # 10
  (0.6, 0.0, 0.3),  # 11
]

# "R" faces using vertex indices
faceIndices = [
  [0, 1, 2, 3, 4, 5],      # bottom face
  [6, 7, 8, 9, 10, 11],    # top face
  [0, 1, 7, 6],            # side 1
  [1, 2, 8, 7],            # side 2
  [2, 3, 9, 8],            # side 3
  [3, 4, 10, 9],           # side 4
  [4, 5, 11, 10],          # side 5
  [5, 0, 6, 11],           # side 6
]
```

Fig. 9.22 shows two views of this data. The left hand image indicates the position of the first six vertices, on the XY plane ($z = 0$). The right-hand image shows all the vertices numbered with the indices used by the faces.

Rendering utilizes Matplotlib's Poly3DCollection class (`https://matplotlib.org/stable/api/_as_gen/mpl_toolkits.mplot3d.art3d.Poly3DCollection.html`) which requires a list of coordinates grouped into sublists for each face. The relevant code in letter3D.py is:

```
# no transformation applied
nVerts = [applyMat(identMat(), vert) for vert in verts]

# build "R" poly, and show
faces = [[nVerts[i] for i in face]
                    for face in faceIndices]
poly = Poly3DCollection(faces, facecolors='skyblue',
           linewidths=1, edgecolors='black', alpha=0.95)
ax.add_collection3d(poly)
```

The first line applies the identity matrix to every coordinate, leaving the shape unchanged.

Moving the "R": Our transformation will do four things:

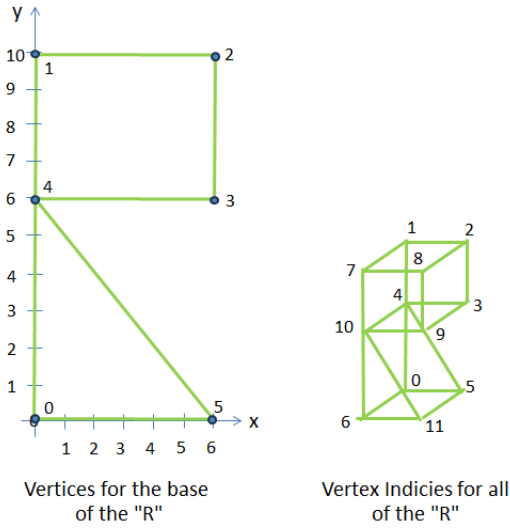

Figure 9.22. Defining the "R" Shape

(1) an x-axis rotation by 90° (to make the "R" stand on the XY plane);

(2) a z-axis rotation by 30° (to turn the "R" counterclockwise over the XY plane;

(3) scaling by 4 (i.e. "R" will now be 4 units high);

(4) translate the shape's origin to $(-1, 1, -3)$, moving the rest of the shape in the process.

Thankfully, the shape's origin (its vertex 0) is already at $(0, 0, 0)$, so there's no need to bracket these operations in translations. The resulting image is shown in Fig. 9.23.

You can confirm the transformation's result by rotating the viewpoint of the Matplotlib window. For instance, vertex 0 is at $(-1, 1, -3)$ and the shape is four units tall.

The transformation code:

```
# two rotations, scale, translate
mats = [transMat(-1,1,-3), scaleMat(4,4,4), rotZMat(30), rotXMat(90)]
mat = multMatsList(mats)
nVerts = [applyMat(mat, vert) for vert in verts]
```

As in the 2D case, the transformations are performed in right-to-left order.

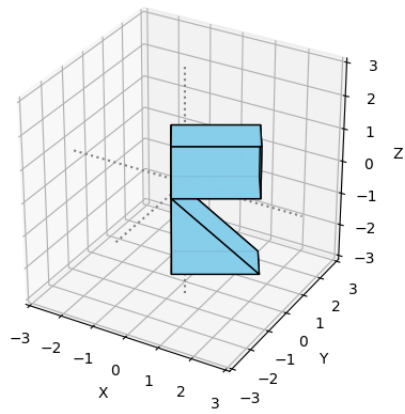

Figure 9.23. The Final Position of the "R" Shape

9.16 Recursive Cube Drawing

Fig. 9.24 shows how a single cube can be redrawn multiple times using recursive translation and scaling.

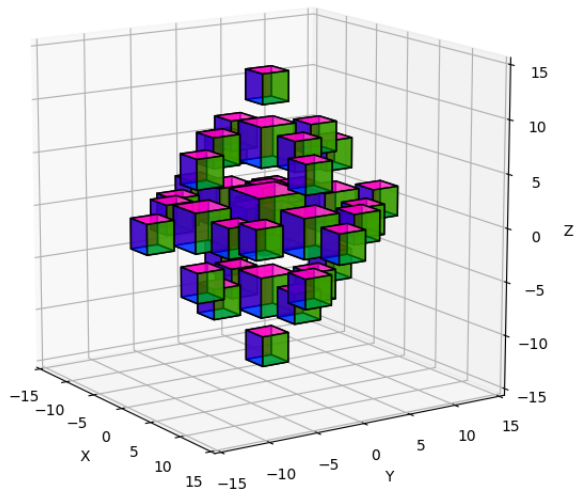

Figure 9.24. A Grid of Shrinking Cubes

cubes.py begins by defining the vertices and face indices for a unit cube with its front lower corner positioned at the origin, and its base resting on the XY plane. Its vertex indices are shown in Fig. 9.25.

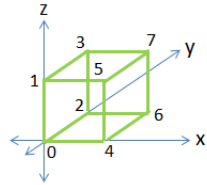

Figure 9.25. A Cube's Vertex Indices

The six faces are defined in cubes.py:

```
# unit cube vertices
verts = [
  [0, 0, 0],  # 0
  [0, 0, 1],  # 1
  [0, 1, 0],  # 2
  [0, 1, 1],  # 3
  [1, 0, 0],  # 4
  [1, 0, 1],  # 5
  [1, 1, 0],  # 6
  [1, 1, 1]   # 7
]

# cube faces using vertex indices
faceIndices = [
  [0, 1, 3, 2],  # Left
  [4, 5, 7, 6],  # Right
  [0, 1, 5, 4],  # Front
  [2, 3, 7, 6],  # Back
  [0, 2, 6, 4],  # Bottom
  [1, 3, 7, 5],  # Top
]

faceColors = ['red', 'green', 'blue', 'yellow',
              'cyan', 'magenta']
```

The ordering of the `faceColors` list corresponds to the ordering of the faces in `faceIndicies`. This will mean, for example, that the top, front, and right sides of the cube will be magenta, blue, and green at run time, as in Fig. 9.24.

Initially the cube is centered at the origin, and enlarged:

```
# center and resize the cube
side = verts[1][2] - verts[0][2]   # x-axis length
m = multMatsList([ scaleMat(SIZE,SIZE,SIZE),
                   transMat(-side/2, -side/2, -side/2)
                 ])
```

```
verts = [applyMat(m, v) for v in verts]

drawCubes(ax, verts, 3)  # use 2, 3 or 4
```

The resursive drawCubes() generates a grid of smaller cubes; its numerical argument is the depth of the recursion.

There are six recursive calls inside drawCubes(), one for each x-, y-, and z-direction. They're encoded using:

```
dirs = [(1,0,0), (-1,0,0), (0,1,0), (0,-1,0), (0,0,1), (0,0,-1)]

def drawCubes(ax, verts, depth):
  # draw a cube and then recurse in every direction
  if depth > 0:
    drawCube(ax, verts)
    for dir in dirs:
      drawCubes(ax, transformCube(verts, dir), depth-1)
```

drawCube() calls Matplotlib's Poly3DCollection to draw the cube in the same way as the letter-drawing in letter3D.py.

transformCube() scales and translates a copy of the cube relative to its center. This requires that the transformation be bracketed by a translation of the current cube's center at P (for example) to the origin, and back to P after the transformation:

```
def transformCube(verts, dir):
  P = calcCenter(verts)  # the cube's midpoint
  side = verts[4][0] - verts[0][0]   # x-axis length
  step = side*1.4
  m = multMatsList([
         transMat(P[0], P[1], P[2]),
         transMat(step*dir[0], step*dir[1], step*dir[2]),
               # translate by step*dir
         scaleMat(0.75, 0.75, 0.75),
         transMat(-P[0], -P[1], -P[2])
       ])
  return [applyMat(m, vert) for vert in verts]

def calcCenter(verts):
  # center coordinate of cube
  xc = (verts[0][0] + verts[4][0])/2
  yc = (verts[0][1] + verts[2][1])/2
  zc = (verts[0][2] + verts[1][2])/2
  return (xc, yc, zc)
```

The new cube is a copy of the current one scaled by 0.75 and translated by 1.4× the direction vector. Only the cube's vertices have to be computed since its face indices never change.

Exercises

(1) Verify each of the following:

(a) A 2D rotation about (r, s) through angle θ is represented by the matrix

$$\begin{bmatrix} \cos\theta & -\sin\theta & r(1-\cos\theta)+s(\sin\theta) \\ \sin\theta & \cos\theta & s(1-\cos\theta)-r(\sin\theta) \\ 0 & 0 & 1 \end{bmatrix}$$

(b) A 2D reflection about the line $y = mx + b$ is represented by the matrix

$$\frac{1}{1+m^2}\begin{bmatrix} 1-m^2 & 2m & -2mb \\ 2m & m^2-1 & 2b \\ 0 & 0 & 1+m^2 \end{bmatrix}$$

(c) A 2D scaling about (r, s) with scale factors c in the x-direction and d in the y-direction is represented by the matrix

$$\begin{bmatrix} c & 0 & r(1-c) \\ 0 & d & s(1-d) \\ 0 & 0 & 1 \end{bmatrix}$$

(2) Modify the code in `transformPoly.py` (or `affinePoly.py`) to carry out the following transformations of the "Knee":

(a) Translate by $(3, 1)$, and then rotate 45° about the origin.
(b) Translate by $(-2, 3)$, and then scale the x-coordinates by 0.8 and the y-coordinates by 1.2.
(c) Rotate through 60° about the point $(6, 8)$.
(d) Reflect through the x-axis, and then rotate 30° about the origin.
(e) Rotate by 30°, and then reflect the shape through the x-axis.

(3) This exercise investigates whether certain transformations are commutative.

(a) Let L_1 be a scaling about the point (r, s) with equal scale factors in the x- and y-directions, and let L_2 be a rotation about the point (r, s) through angle θ. Show that L_1 and L_2 commute. (That is, show $L_1 \cdot L_2 = L_2 \cdot L_1$.)
(b) Give a counterexample to show that, in general, a reflection and a rotation do not commute.
(c) Give a counterexample to show that, in general, a scaling and a reflection do not commute.

(4) A 3×200 matrix D contains the 2D homogeneous coordinates of 200 points. Calculate the number of multiplications required to transform these points using two arbitrary 3×3 matrices A and B. Consider the two possibilities $A(BD)$ and $(AB)D$. Discuss the implications of your results for graphics calculations.

(5) A rotation is sometimes implemented as the product of two shear-and-scale transformations, which can speed up the calculations. The first transformation A_1, shears vertically and then compresses each column; the second transformation A_2 shears horizontally and then stretches each row. Let

$$A_1 = \begin{bmatrix} 1 & 0 & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \text{and} \quad A_2 = \begin{bmatrix} \sec\theta & -\tan\theta & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Show that the composition of the two transformations is a rotation in $\mathbb{R}^2$.

(6) A rotation in $\mathbb{R}^2$ usually requires four multiplications. Compute the product below, and show that the matrix for a rotation can be factored into three shears (each of which requires only one multiplication).

$$\begin{bmatrix} 1 & -\tan\theta/2 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ \sin\theta & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & -\tan\theta/2 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

(7) Write a recursive function called drawTriangles() which draws an equilateral triangle and then recursively calls itself three times to draw smaller triangles offset from its three sides. For example, if the function recurses three times, the image in Fig. 9.26 is drawn.

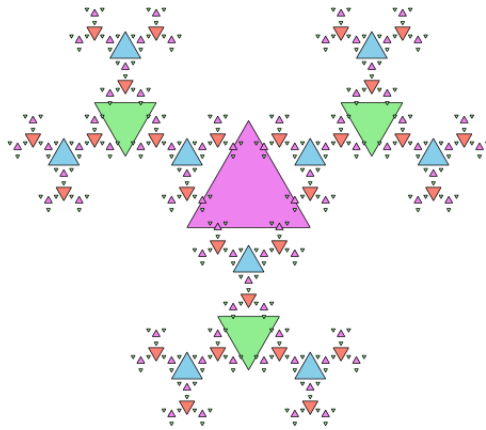

Figure 9.26. Recursive Triangles

The initial triangle could be defined like so:

```
base = math.sqrt(3)/6
coords = [ (-0.5,-base), (0.5,-base), (0, 2*base)]  # unit lengths
```

which would place its center at the origin.

The transformation applied to the triangle is composed from a scaling, translation, and a rotation, each performed three times as in Fig. 9.27. The diagram suggests that the translation should be twice the distance to the base, and the scaling factor is a 1/2.

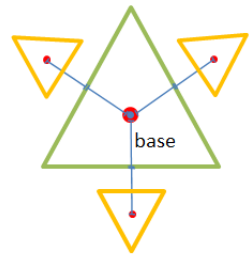

Figure 9.27. Generating Three Smaller Triangles

(8) Generate the tiling pattern shown in Fig. 9.28.

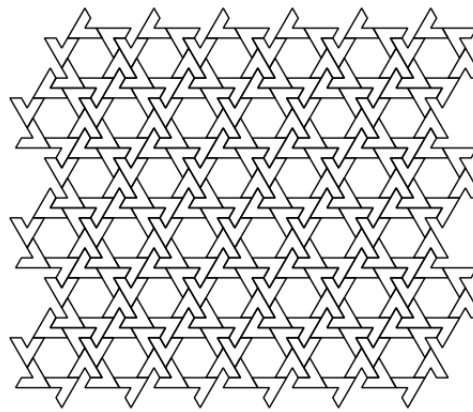

Figure 9.28. Tiling using Hexagons and ”V”s

One way to implement this is to start with a hexagon and "V" shape, as in Fig. 9.29.

A function could rotate and translate copies of "V" around the hexagon. The tiling then becomes a matter of translating copies of the hexagon before the "V"'s are drawn around them.

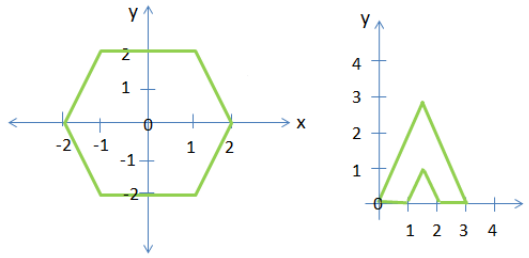

Figure 9.29. A Hexagon and a "V"

The easiest way of determining suitable translations for the hexagons is by sketching out Fig. 9.28 on some graph paper.

(9) Baravelle spirals (Sec. 9.7) can be created from any regular polygon. Modify the code in baravelle.py to start from a different shape than a square. You can check your results against the equilateral triangle, pentagon, and hexagon utilized by the National Curve Bank (`https://old.nationalcurvebank.org/baravelle/baravelle.htm`).

(10) Truchet tiles are decorated with patterns that are *not* rotationally symmetric, causing them to generate intriguing patterns when tiled over the plane with random 90° rotations (`https://en.wikipedia.org/wiki/Truchet_tiles`). Two such tiles are available on online in tri.png and arc.py, shown in Fig. 9.30.

Figure 9.30. Two Truchet Tiles

Write code using Matplotlib's Affine2D class to generate a 12 × 12 grid of randomly rotated Truchet tiles using one of the PNG images as the basic tile. For example, the code might generate the patterns shown in Fig. 9.31.

(11) The transformational approach to generating the Hilbert curve family in section 9.8 can of course be applied to other fractal curves. Modify the code in hilbert.py to generate a Koch curve, Dragon curve, or another of your choosing (see Fig. 9.32).

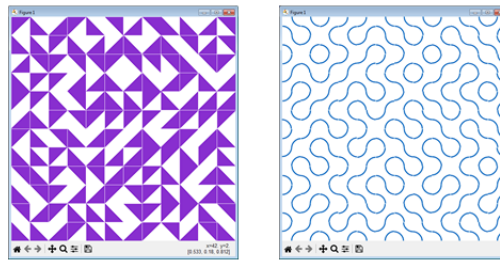

Figure 9.31. Truchet Patterns using tri.png and arc.png

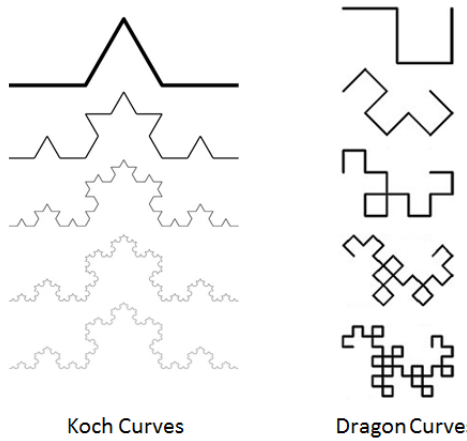

Figure 9.32. Two Fractal Curve Families

(12) The *Functional Geometry* paper by Henderson (sec. 9.12) is well worth a read. It can be downloaded from `https://eprints.soton.ac.uk/257577/1/funcgeo2.pdf`.

You can experiment with his approach using several path and SVG files available online: `edge.path`, `triangle.path`, `house.svg`, and `wave.svg`.

Remember that if you employ SVG files containing multiple 'd' paths (as in the two SVG examples) then you'll need to modify `Tile.py` to manipulate Matplotlib PathCollection objects rather than paths and PathPatch. Have a look at `ShowSVG.py` for coding details.