

## jFAT: A Fairly Accurate Tiling Library

Andrew Davison

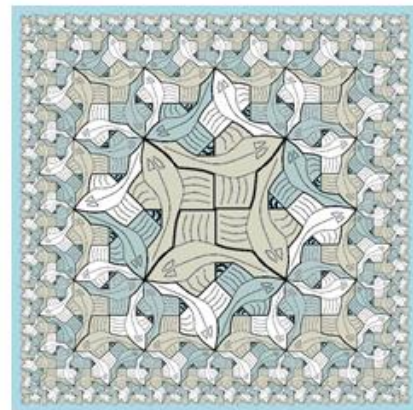
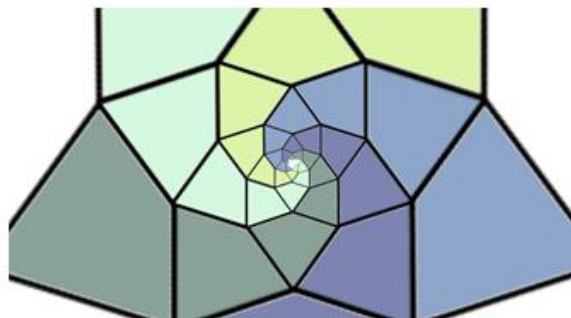
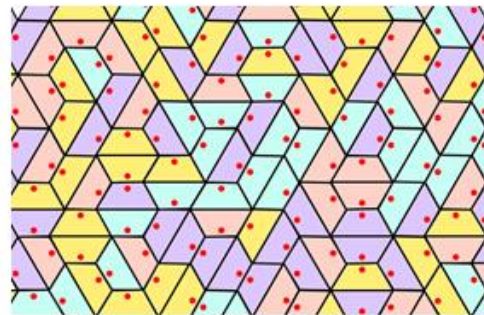
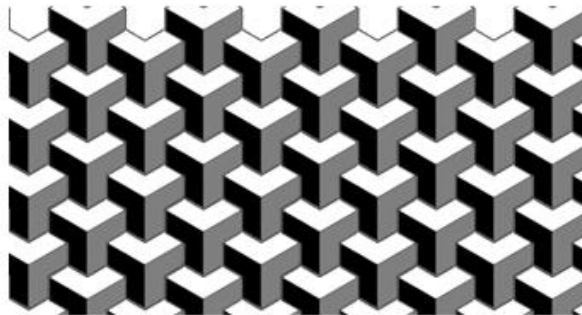
Dept. of Computer Eng., Prince of Songkla Univ.

Hat Yai, Songkhla 90110, Thailand

Email: [ad@fivedots.coe.psu.ac.th](mailto:ad@fivedots.coe.psu.ac.th)

June 2022

The jFAT Java library makes it simple to create tiles (2D shapes) and write code to have them cover the screen. A few examples are shown in Figure 1, with larger versions in the jFAT download (<http://fivedots.coe.psu.ac.th/~ad/jfat/>).



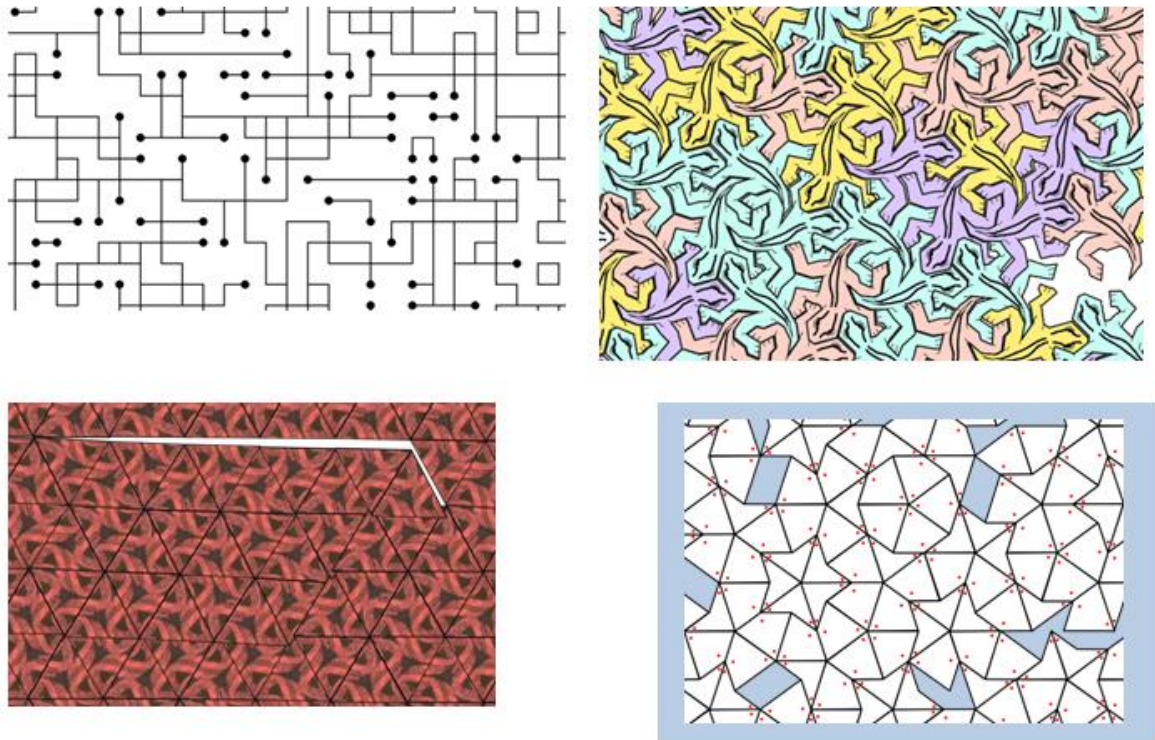


Figure 1. Various Tilings possible with jFAT.

The last four examples in the figure use jFAT's built-in tiling functions, which account for the "fairly accurate" part of the library's name – the tessellations involving the Escher-style lizards [8], the carpeted equilateral triangles, and the Penrose P2 tiles (the dart and the kite) [5] include gaps which should not be present. There are various reasons for these spaces: the first two images suffer from inaccuracies in the definition of their tiles, but the openings in the Penrose tessellation point to a more complex problem related to backtracking, which I'll discuss in section 9.

jFAT can define tiles in four ways – using Java shapes, with textual descriptions, and as PNG or SVG images. During tessellation, a tile can be translated, rotated, flipped, resized, and recolored.

The tiling process can be coded in several ways, the most common is based on nested loops that fill the space row-by-row. It's also possible to use recursion, tile composition, a test-then-draw coding style, or simply rely on jFAT's built-in tiling functions. This document explains these various techniques through examples, all of which are included in the jFAT download.

jFAT can easily handle tiles made from irregular shapes (such as Escher-style fish and lizards) since the library supports a mix of geometric and pixel-based operations. For example, shapes include "inner" and "outer" points placed inside and outside the tile which can be used to test if an area is occupied, or if a tile is adjacent to another one. However, this comes at the cost of accuracy since there may not be enough points to detect every overlap or

adjacent tile. I'll return to these issues several times as I describe examples using more complex shapes.

jFAT offers two built-in functions for automatically tiling a space: `tileLocs()` and `tileSpacey()`, which rely on the presence of inner and outer points, and also point pairs which restrict how tiles can be matched.

## 1. Tile Formats

This section presents examples of the four ways of specifying a tile shape: using Java shapes, with a textual description, and as PNG or SVG images.

### 1.1. Shapes

jFAT's `ShapeOps` class contains shape-related functions, of which the most useful for defining a tile are `nPolygon()`, `radPolygon()`, and `points()`. `nPolygon()` generates a regular polygon of the specified number of sides, each with the given length, as in the following code snippet:

```
// from Pentagons.java
Shape s = ShapeOps.nPolygon(Tile.WIDTH/2, Tile.HEIGHT/2, 30, 5);
                                     // center pt., length, no. sides
s = ShapeOps.rotate(s, 90); // rotate first point to top
Tile t = new Tile(s, "pentagon");
t.view();
t.reportPoints();
```

The five coordinates are placed around the given center, with the first on the x-axis, and the others running counterclockwise around the center. In this example, the shape is also rotated 90 degrees counterclockwise so that the first point is moved to the top. The default size for the new tile is 100 x 100 pixels.

`Tile.view()` displays the tile in a window, and `Tile.reportPoints()` prints information about its coordinates, as shown in Figure 2.

```
> compile Pentagons.java
Compiling Pentagons.java ...

> run Pentagons
Executing Pentagons with jFAT.jar ...
Saved image to TileViewer.png
Points for tile pentagon:
P0 (52.44, 26.92); 108.00 degs; 30.00 pixels
P1 (28.17, 44.55); 108.00 degs; 30.00 pixels
P2 (37.44, 73.08); 108.00 degs; 30.00 pixels
P3 (67.44, 73.08); 108.00 degs; 30.00 pixels
P4 (76.71, 44.55); 108.00 degs; 30.00 pixels
Image size: (97, 94)
No. Inners: 10; No. Outers: 5
```

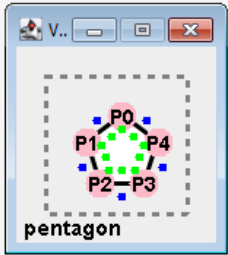


Figure 2. The `Pentagons.java` Tile.

Tile.view() displays labeled coordinates for the shape (P0 to P4; the "P" comes from the tile's "pentagon" ID), and automatically generated inner and outer points (the green and blue dots). These points only become useful when utilizing more advanced drawing operations or the built-in tiling functions, tileLocs() and tileSpacey(), so will be ignored for now.

Tile.reportPoints() prints the image coordinates for the points (i.e. (0,0) is the top-left corner). It also lists the interior angle between a point and its successor in counterclockwise order, and the distance between them. The values reported in Figure 2 confirm that the shape is a regular pentagon.

The dashed line around the pentagon marks out the dimensions of the tile. The shape's internals are colored white and its background is transparent (shown as a light gray in the figure).

Tile.view() automatically saves a copy of the displayed image to TileViewer.png, without the labeled coordinates or inner and outer points. This file can be useful for double-checking the shape, and estimating coordinate positions. The Tile.view() window can also be clicked upon with the mouse to display the current cursor coordinate in a popup window.

ShapeOps.radPolygon() is a close relative of nPolygon(), but specifies its size in terms of a radius from the center:

```
// from DrawATriangle.java
Shape s = ShapeOps.radPolygon(Tile.WIDTH/2, Tile.HEIGHT/2, 30, 3);
                                     // center, radius, no. sides
s = ShapeOps.rotate(s, 90); // rotate first point to top
Tile t = new Tile(s, "triangle");
t.view();
t.reportPoints();
```

Figure 3 shows the output from Tile.view() and Tile.reportPoints() in this case.

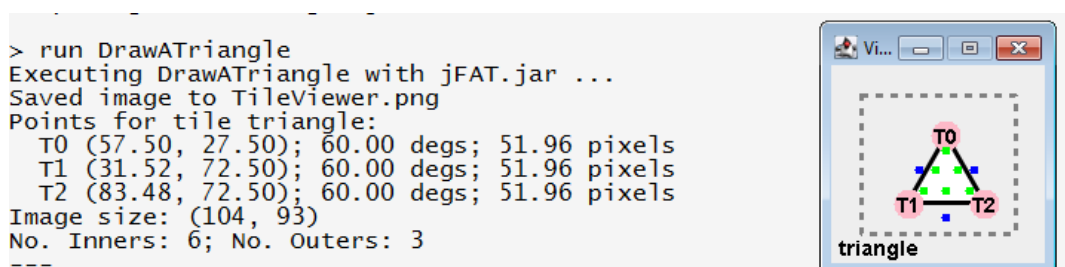


Figure 3. The DrawATriangle.java Tile.

The coordinates are labeled with "T" since the tile's ID is "triangle".

ShapeOps.points() creates a shape from a series of points supplied as two arrays for their x- and y- axis values.

```
// in Carpet.java
double[] xs = {42, 105, 169};
double[] ys = {38, 149, 38};
Shape s = ShapeOps.points(xs, ys);
Tile carpetTile = new Tile(s, "carpet");
    // the tile points will be labeled C0, C1, ...
    : // more code, not relevant here
carpetTile.view();
carpetTile.reportPoints();
```

The coordinate were obtained by examining "carpet.png" (Figure 4) in a graphic applications.



Figure 4. carpet.png

Sadly these points don't quite form an equilateral triangle, and the error gradually becomes evident as its tiles fill the screen (see Figure 1).

The results of calling Tile.view() and Tile.reportPoints() are shown in Figure 5.

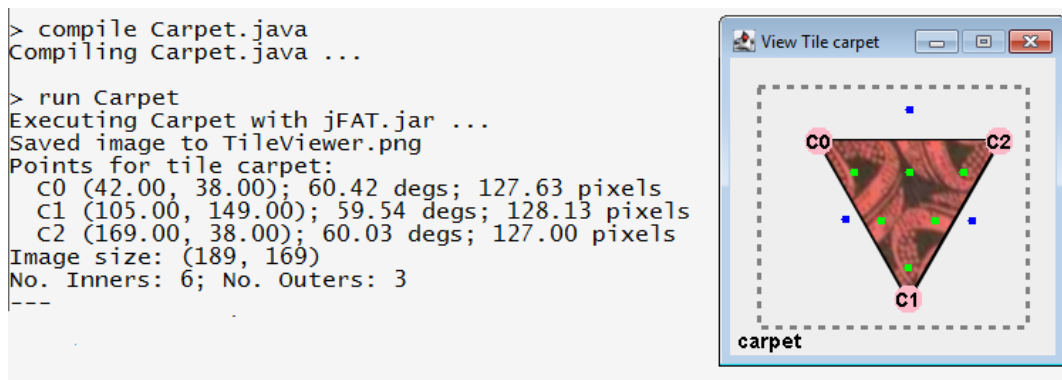


Figure 5. The Carpet.java Tile.

The data produced by Tile.reportPoints() highlight the inaccuracies in the shape's coordinates. The interior angles should all be 60 degrees, and the sides should be the same length.

## 1.2. Tile Description Text

The easiest way to define slightly more complicated geometric shapes, such as an arrow, dart, or kite, is to use jFAT's tile description text, essentially just a series of degree angles and side lengths. As an example, Figure 6 shows the typical angles and dimensions for the Penrose dart and kite [10].

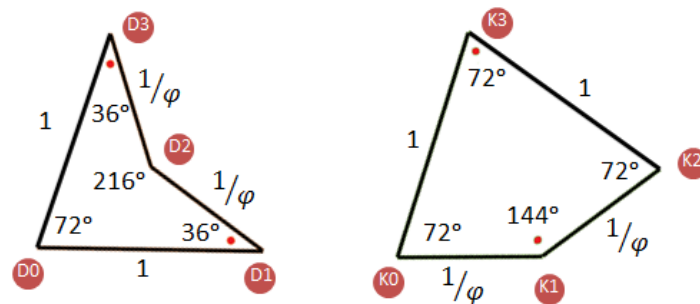


Figure 6. The Penrose Dart and Kite.

The tile description data for these shapes is calculated by choosing a starting coordinate for a point (e.g. "D0" at (20,100), "K0" at (20,120)), and multiplying the dimensions by a suitable value (e.g. 100) to make the shape's sides a reasonable pixel length (see Figure 7). One requirement is that the starting position must be such that all the other points have positive coordinates. The points are now considered in counterclockwise order.

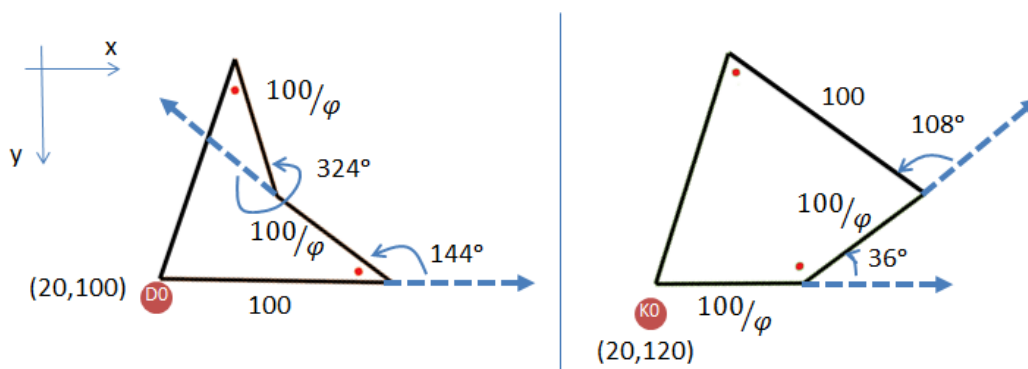


Figure 7. Tile Data for the Penrose Dart and Kite.

Determining the angles between the coordinates is the trickiest step since they have to be specified relative to the current side direction. One way of thinking about this is with turtle geometry – an imaginary turtle moves along the shape's edges, and an angle is measured by turning the turtle counterclockwise from its current direction to face along the next edge.

Fortunately, since the shape is closed, there's no need to calculate a turning angle for the last point.

The resulting data for the dart and kite are:

```
Tile 20 100
  0 100
 144 61.8
 324 61.8  $
```

and

```
Tile 20 120
  0 61.8
  36 61.8
 108 100  $
```

The starting coordinate appears on the first line, followed by a turning angle and distance on each subsequent line. Since both the dart and kite have four sides, only three lines of data need to be calculated. 61.8 is an approximation to  $100/\varphi$ , and "\$" denotes the end of the data.

jFAT includes a small standalone program, ViewTile, which can load a tile data file and display the corresponding tile and coordinate information. Figure 8 shows how the tile data from above (stored in dartTile.txt and kiteTile.txt respectively) are displayed.

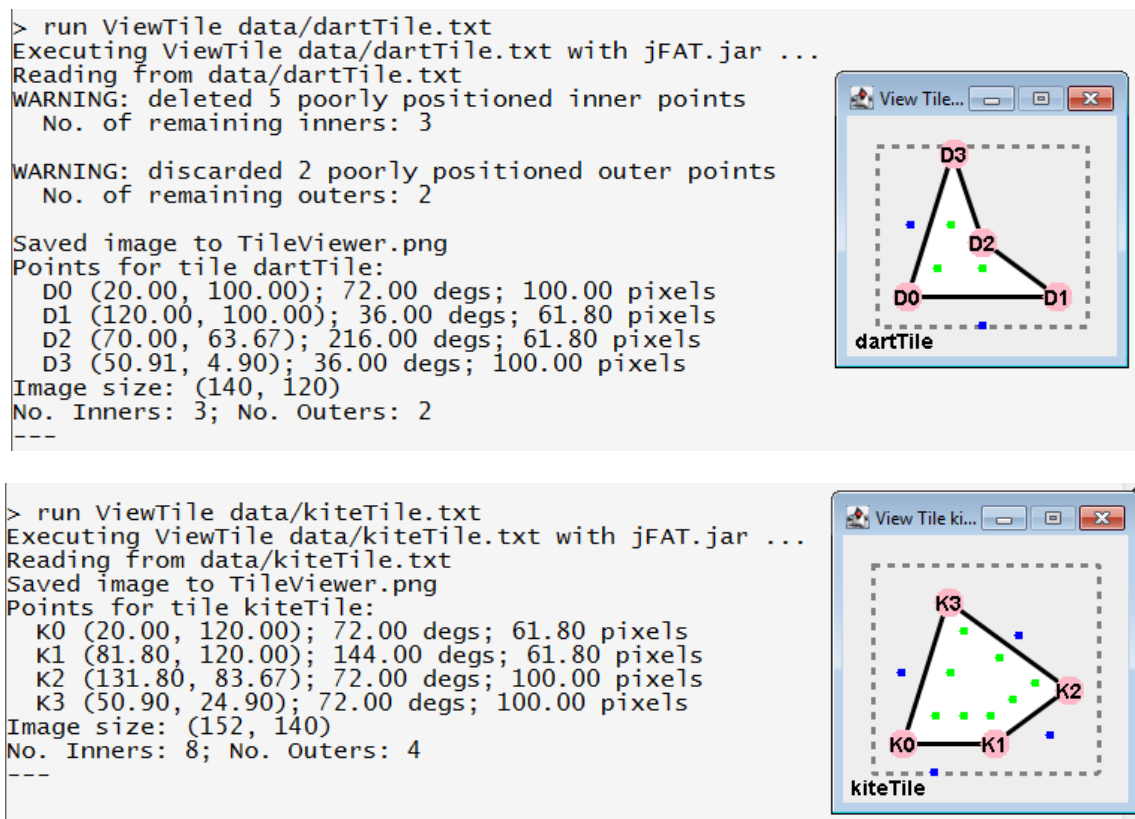


Figure 8. Using ViewTile to Check the Tile Data.

Inner and outer points are generated for the tiles, as are point labels (which use the first letter of the filename). Note that the dart's concave sides has caused the Tile constructor to issue some warnings. I'll return to that issue when I discuss inner and outer points in section 6.

Once the tile data is satisfactory, it can be used to initialize a tile in a program:

```
// in DrawPenrose.java
Tile dart = new Tile("data/dartTile.txt");
Tile kite = new Tile("data/kiteTile.txt");
```

### 1.3. Tiles based on PNG Images

A tile may have such a complex shape that it's not feasible to define it using a shape or tile data. In that case, it can be created from a PNG image, but the programmer must ensure that the image background is transparent, and will also have to supply coordinates for the tile's points. For example, an Escher-style lizard image stored in liz.png can be examined with jFAT's ViewTile as in Figure 9.

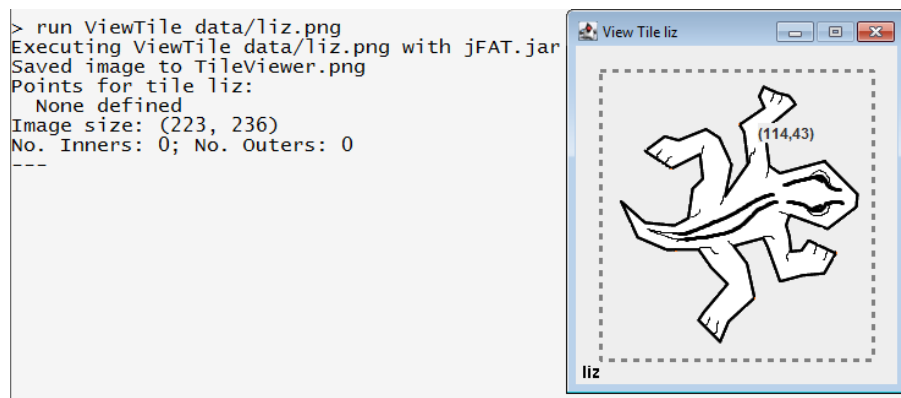


Figure 9. liz.png in ViewTile.

The lack of coordinates and inner and outer points in ViewTile's window indicates that the user still has to supply coordinate information in their code.

One way of getting that data is to click the mouse over the ViewTile image, as shown on the lizard's left arm in Figure 9. However, it's often easier to examine the image with graphics software where the picture can be enlarged so that the pixel selection can be more precise.

The coordinates that define the 'corners' of a tile can be placed anywhere inside the image rectangle marked by the dashed line in ViewTile. In particular, this allows the complicated lizard tile to utilize just six points, relating to the underlying hexagon used by Escher when designing the lizard [1], as illustrated by Figure 10.



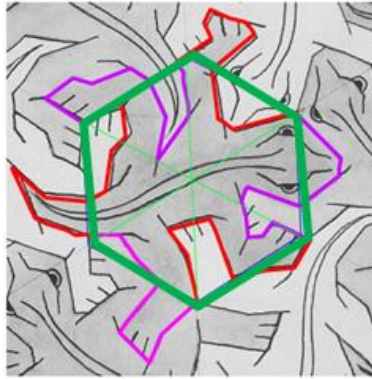


Figure 10. Lizard in a Hexagon.

The six coordinates are applied to the tile by calls to `Tile.addPt()`:

```
// in DrawLizards.java
Tile lizard = new Tile("data/liz.png");
lizard.addPt("C0", 113, 43); // points in ccw order
lizard.addPt("C1", 57, 80);
lizard.addPt("C2", 61, 147);
lizard.addPt("C3", 126, 186);
lizard.addPt("C4", 193, 148);
lizard.addPt("C5", 184, 73);
lizard.view();
lizard.reportPoints();
```

It's good practice to specify the points in counterclockwise order. This is only a requirement when using the built-in tiling functions, but is a good habit to acquire.

Once a tile has coordinates, the `Tile` constructor can automatically generate inner and outer points, as in Figure 11.

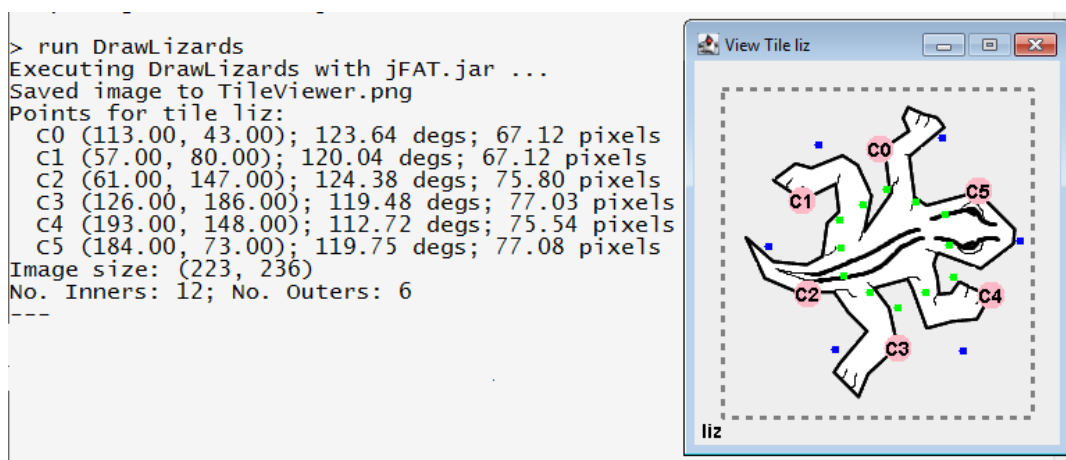


Figure 11. The `DrawLizards.java` Tile.

The coordinate data printed by `Tile.reportPoints()` indicate a problem with the lizard tile, which only becomes apparent when it's used for tiling (as shown in Figure 1). The points do not precisely define a regular hexagon (i.e. all the sides should be the same length, with interior angles of 120 degrees). This inaccuracy gradually builds as the screen is tiled until collision detection will start to leave spaces between the lizards.

The underlying problem is that the image doesn't quite fit into a regular hexagon. This often occurs when using pixel-based drawing tools which lack the precision of vector-based tools (such as Inkscape (<https://inkscape.org/>)).

#### 1.4. Tiles based on SVG Images

The inaccuracies of pixel-based images is the main reason for jFAT also being able to load SVG files as created by Inkscape (and other tools).

SVG is an extremely rich format, so jFAT restricts itself to only extracting "path" information from a picture which describes a shape's outline using curves and straight lines. The most common problem with this restriction is that SVG also supports basic shapes, such as rectangles and ellipses, which are not stored as paths by default. For example, the rectangle created by Inkscape in Figure 12 is encoded as:

```
<rect
  style="fill:#0000ff;stroke:#000000;stroke-width:0.264583"
  id="rect10"
  width="83.910713"
  height="55.940475"
  x="27.214285"
  y="38.55357" />
```

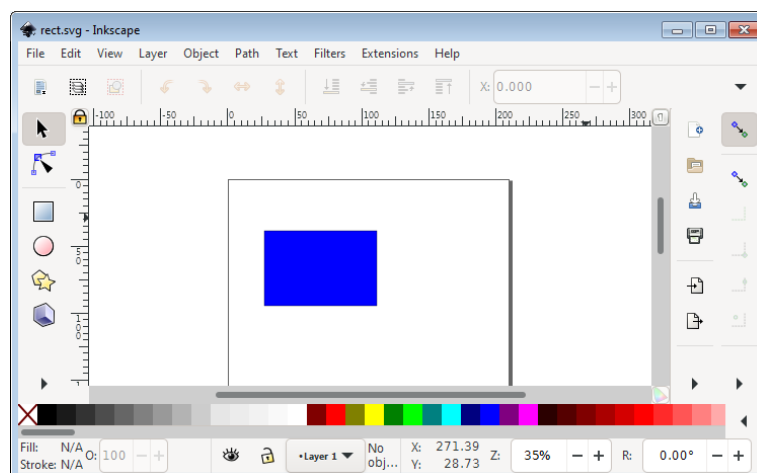


Figure 12. An SVG Rectangle (in rect.svg)

The easiest way to fix this is to select the offending graphic object inside Inkscape and use the "Object to Path" menu item to convert it to a path. The code becomes:

```
<path
  id="rect10"
  style="fill:#0000ff;stroke:#000000;stroke-width:0.264583"
  d="M 27.214285,38.55357 H 111.125 V 94.494045 H 27.214285 Z" />
```

However, it's not apparent by viewing an SVG image in Inkscape whether a shape is encoded as a path. One way of testing for this is to use jFAT's ViewTile tool to load the SVG file. Loading the original rect.svg file triggers an exception:

```
> run ViewTile data/rect.svg
Executing ViewTile data/rect.svg with jFAT.jar ...
Parsing data/rect.svg
Root node is svg
No. of paths: 0
No path shape found
Exception in thread "main" java.lang.IllegalArgumentException: image == null
!
    at java.desktop/javafx.imageio.ImageTypeSpecifier.createFromRendered
Image(ImageTypeSpecifier.java:917)
    at java.desktop/javafx.imageio.ImageIO.getWriter(ImageIO.java:1601)
    at java.desktop/javafx.imageio.ImageIO.write(ImageIO.java:1537)
    at Pics.save(Pics.java:68)
    at TileViewer.<init>(TileViewer.java:68)
    at Tile.view(Tile.java:1483)
    at Tile.view(Tile.java:1465)
    at ViewTile.main(ViewTile.java:65)
Finished.
>
```

After the rectangle has been translated into a path inside Inkscape, a subsequent call to ViewTile will succeed, as shown in Figure 13.

```
> run ViewTile data/rectPath.svg
Executing ViewTile data/rectPath.svg with jFAT.jar ...
Parsing data/rectPath.svg
Root node is svg
No. of paths: 1
Saved image to TileViewer.png
Points for tile rectPath:
R0 (27.21, 38.55); 270.00 degs; 83.91 pixels
R1 (111.13, 38.55); 270.00 degs; 55.94 pixels
R2 (111.12, 94.49); 270.00 degs; 83.91 pixels
R3 (27.21, 94.49); 270.00 degs; 55.94 pixels
Image size: (132, 115)
No. Inners: 8; No. Outers: 4
---
```

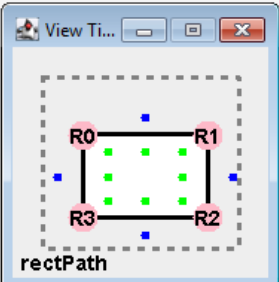


Figure 13. A Tile Using an SVG Path.

Note that any styling information (e.g. the fact that the rectangle is blue) is ignored. Also, no scaling is applied to shape's lengths which are treated as pixel dimensions. Figure 13 also shows that coordinates are generated for the tile.

SVG is probably most useful if a shape involves curves or more complex line combinations. For example, axes2.svg represents an axe head, as shown in Figure 14.

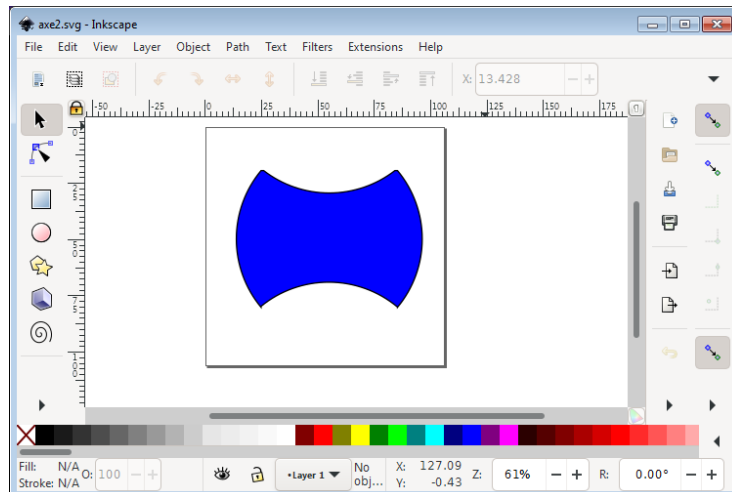


Figure 14. A Blue Axe Head.

Figure 15 shows how ViewTile interprets it as a tile.

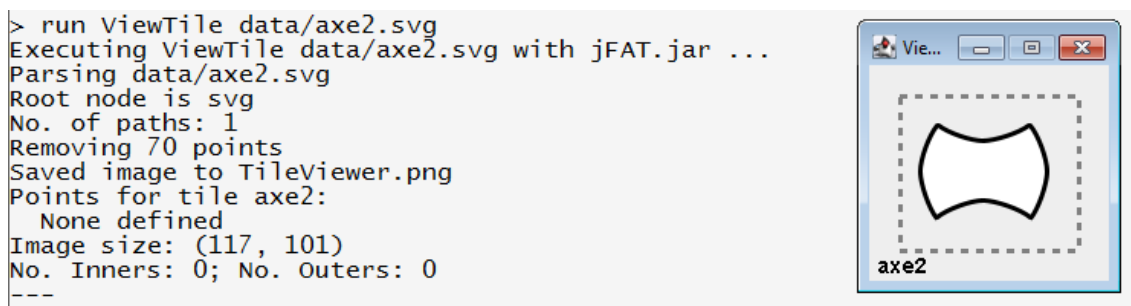


Figure 15. The Axe Tile.

jFAT utilizes the PathParser class written by Nima Taheri (<https://gist.github.com/nimatrueway/cc1668c16096e2f2184275d8f780e3a5>) to convert the path into a Java Path2D object. Except for all but the simplest shapes, the resulting object will contain a very large number of line segments, leading to an excessive number of tile points.

For instance in Figure 15, ViewTile reports that the shape has 70 points, and by default, the Tile constructor automatically deletes all the points if there's more than 30. The programmer must instead define their own points for the axe:

```
// in Axes.java
Tile longAxe = new Tile("data/axe2.svg");
  // removes all 70 points, so the programmer writes...

longAxe.addPt("A0", 24, 18);
```

```

longAxe.addPt("A1", 24, 80);
longAxe.addPt("A2", 84, 80);
longAxe.addPt("A3", 84, 18);

longAxe.view();
longAxe.reportPoints();

```

The calls to `Tile.view()` and `Tile.reportPoints()` are shown in Figure 16.

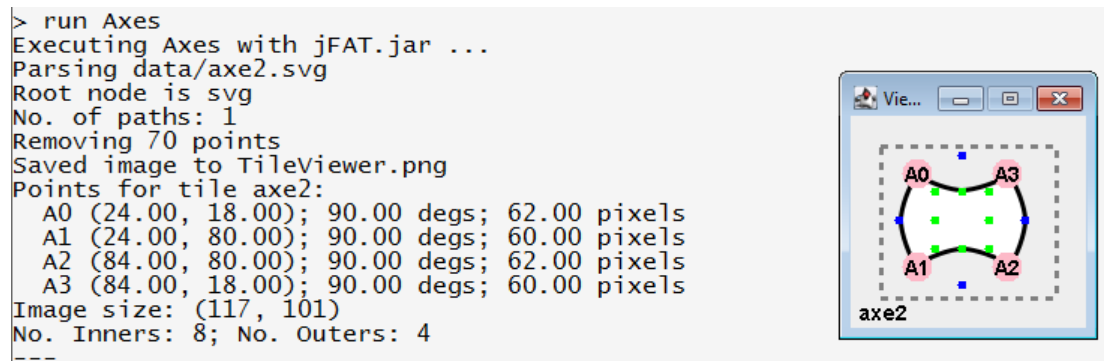


Figure 16. Tile Creation in Axes.java

Four 'corner' points for the axe are sufficient for controlling its tiling. Note that once the tile has coordinates, then inner and outer points are generated automatically.

Another issue is that an SVG file may contain multiple shapes, and therefore multiple paths. The Tile constructor loads all the paths but combines them into a single Path2D object.

### 1.5. Decorating a Tile

The preceding subsections have produced tiles with a black outline and filled with a nondescript white. Fortunately, it's quite easy to enhance a tile's appearance, by accessing its image via a Graphics2D reference. The square and equilateral triangle tiles created by DrawShapes.java use this approach. As Figure 17 illustrates, a red dot is added to the square, and a blue diamond to the triangle, both at the top of the tiles. (Sadly, the green inner points tend to obscure these additions.)

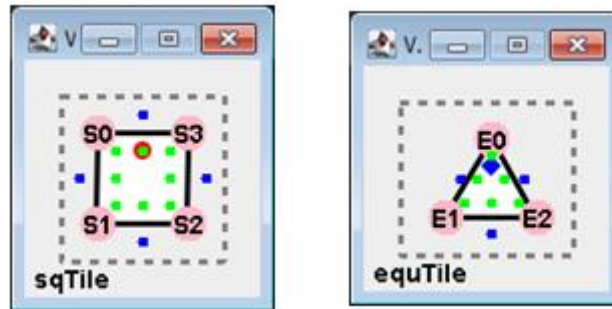


Figure 17. Decorated Tiles.

The code that make these additions:

```
// in DrawShapes.java
Tile sq = new Tile("data/sqTile.txt");

Graphics2D g2d = sq.getGraphics();
// red circle
g2d.setColor(Color.RED);
g2d.fillOval(40, 25, 10, 10); // near top

Tile equ = new Tile("data/equTile.txt");

g2d = equ.getGraphics();
// blue square
g2d.setColor(Color.BLUE);
g2d.fill( ShapeOps.nPolygon(50, 35, 8, 4));
```

The coordinates used in the calls to Graphics2D.fillOval() and fill() were obtained by clicking the mouse over the tile images displayed by Tile.view(). A less 'hacky' approach is to utilize the coordinates for the tile's points, as in Pentagons.java:

```
Shape s = ShapeOps.nPolygon(Tile.WIDTH/2, Tile.HEIGHT/2, 30, 5);
s = ShapeOps.rotate(s, 90); // rotate first point to top
Tile t = new Tile(s, "pentagon");
ArrayList<Point2D.Double> pts = t.getImPts();

Graphics2D g2d = t.getGraphics();
g2d.setColor(Color.RED); // red circle
g2d.fillOval((int)pts.get(0).x-5, (int)pts.get(0).y+10, 10, 10);
// below top point of pentagon

// blue triangle
Shape tri = ShapeOps.radPolygon(50, 64, 7, 3);
// coordinates obtained from looking at Tile.view()
```

```

tri = ShapeOps.rotate(tri, 90); // so tip points upwards
g2d.setColor(Color.BLUE);
g2d.fill(tri);

```

This adds a red circle and a blue triangle to the pentagon tile, as shown in Figure 18.



Figure 18. A Pentagon with a Red Circle and Blue Triangle.

Another approach is possible if the tile is derived from a Java Shape. In that case, the shape can clip the drawing area to constrain any operations to the tile's surface. For example, the "Y" Tile in Stars.java has a red line drawn from the top-left to the bottom right of the image:

```

Tile yTile = new Tile("data/yTile.txt");

g2d = yTile.getGraphics();
g2d.setClip(yTile.getShape());
g2d.setColor(Color.RED);
g2d.setStroke(new BasicStroke(2.0f));
g2d.drawLine(15, 0, yTile.getWidth(), yTile.getHeight()-5);

```

The call to setClip() restricts the drawing operation to the "Y" shape, as shown in Figure 19.

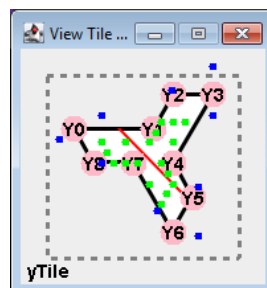


Figure 19. The "Y" Tile in Stars.java

Yet another approach is to use texture painting. The star tile (on the left of Figure 20) created in Stars.java shows that an image (on the right) can be used as a texture.

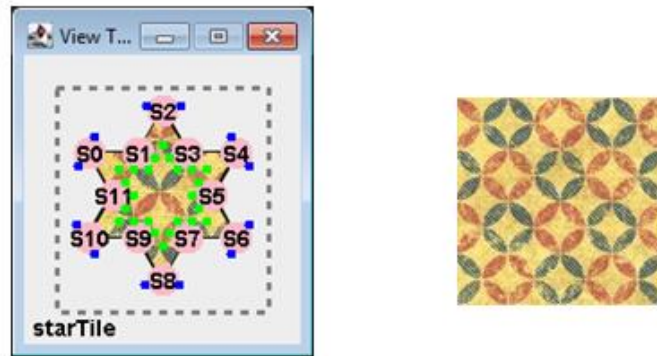


Figure 20. The "Star" Tile in Stars.java, and its Texture Image.

The code in Stars.java:

```
Tile starTile = new Tile("data/starTile.txt");
Graphics2D g2d = starTile.getGraphics();
g2d.setPaint(Pics.loadTex("data/fabric.jpg"));
g2d.fill(starTile.getShape());
// painting is constrained within the shape
```

The final way to change the color of a tile is with `Tile.colorChg()` which, unlike the other approaches, generates a new tile. For instance, it's employed to change the white parts of an hexagon tile to light blue in `DrawShapes.java`:

```
private static final Color LIGHT_BLUE = new Color(173,216,230);
:
Shape s = ShapeOps.nPolygon(60, 60, 50, 6);
Tile hex = new Tile(s, "hexagon").colorChg(Color.WHITE, LIGHT_BLUE);
```

Any non-white parts of the tile, such as its black edges, are unaffected. `colorChg()` uses a color distance metric to find the first color, and any 'similar' colors. This fuzzy matching is useful when modifying a tile where the quality of the image has been affected by operations such as rotation and resizing.

## 2. Basic Tile Drawing

This section describes the `Tile.drawAt()` operation which prints a tile onto an image.



DrawATriangle.java starts by creating an equilateral triangle with a fancy 'f' inside it (see Figure 21):

```
Shape s = ShapeOps.radPolygon(Tile.WIDTH/2, Tile.HEIGHT/2, 30, 3);
s = ShapeOps.rotate(s, 90);

Tile tri = new Tile(s, "triangle");
Graphics2D g2d = tri.getGraphics();
g2d.setFont(new Font("Algerian", Font.PLAIN, 32));
g2d.drawString("f", 48, 67); // centered 'f'
```

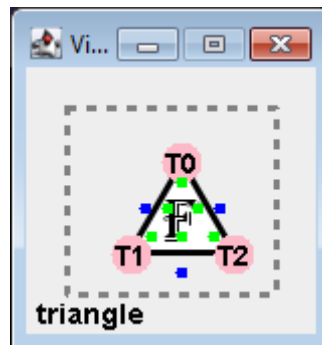


Figure 21. An 'F' Triangle.

The image drawn upon by the tile is managed by an `ImageViewer` object, which is usually created with a title string and window dimensions:

```
ImageViewer iview = Pics.view("Draw a Triangle", 200, 200);
```

This creates an empty window called "Draw A Triangle" of 200 by 200 pixels. A common alternative is to set the window's size with `Pics.altScreen()`:

```
ImageViewer iview = Pics.view("Draw a Triangle", Pics.altScreen());
```

`Pics.altScreen()` tries to find a second monitor, and resizes the window to fill that screen.

`Tile.drawAt()` typically takes four arguments: a reference to the image, a tile point label, and an  $(x, y)$  coordinate on the image where that point should be drawn. For example:

```
tri.drawAt(iview.getImage(), "T0", 100, 50);
```

This draws the tile so that its `T0` point is located at  $(100,50)$  in the image managed by `iview`. The window is updated with a call to `ImageViewer.repaint()`:

```
iview.repaint();
```

The resulting window is shown in Figure 22.

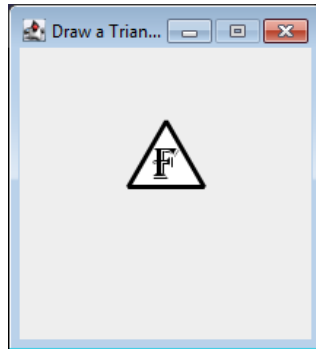


Figure 22. The Tiling Window for DrawATriangle.java

Note that the tiling window does not show the tile's labels or inner and outer points.

At this stage, the tile is now storing **two** sets of coordinates – its *image* coordinates, which are the position of its points relative to the tile image, and its *tiling* coordinates, the position of its points relative to the tiling image. This distinction is illustrated by the code below:

```
System.out.println("Before drawing...");
System.out.println("  Image Center: " + tri.getCenterIm());
System.out.println("  Drawn Center: " + tri.getCenterLoc()); // null
System.out.println("  Image Tip: " + tri.getPtIm("T0"));
System.out.println("  Drawn Tip: " + tri.getPtLoc("T0")); // null

tri.drawAt(iview.getImage(), "T0", 100, 50);
iview.repaint();

System.out.println("\nAfter drawing...");
System.out.println("  Image Center: " + tri.getCenterIm()); //no chg
System.out.println("  Drawn Center: " + tri.getCenterLoc());
System.out.println("  Image Tip: " + tri.getPtIm("T0")); // no chg
System.out.println("  Drawn Tip: " + tri.getPtLoc("T0"));
```

It outputs:

```
Before drawing...
  Image Center: Point2D.Double[57.4999999999999, 57.5000000000001]
getCenterLoc(): Tile triangle has not been drawn
  Drawn Center: null
  Image Tip: Point2D.Double[57.4999999999999, 27.4999999999996]
getPtLoc(): Tile triangle has not been drawn
  Drawn Tip: null

After drawing...
  Image Center: Point2D.Double[57.4999999999999, 57.5000000000001]
  Drawn Center: Point2D.Double[94.5, 68.5]
  Image Tip: Point2D.Double[57.4999999999999, 27.4999999999996]
  Drawn Tip: Point2D.Double[99.9999999999999, 50.0]
```

A tile has image coordinates before it is drawn, but no tiling coordinates. Afterwards, the image coordinate don't change, but its points do now have tiling coordinates.

The most important methods related to these two coordinate systems are `Tile.getPtIm()` (for retrieving an image coordinate) and `Tile.getPtLoc()` (for accessing a tiling coordinate).

A useful way to think about tile drawing is that the tile is a **stencil** printed onto the image. Subsequently, the tile can be moved (or rotated, etc.) and drawn at a new position on the tiling image without affecting the previously drawn copy.

In the example above, the drawing position (100, 50) was hardwired into the code, but after the first drawing it's more usual to use the data returned by `Tile.getPtLoc()` to calculate a new coordinate. For instance, assume that `DrawATriangle` utilizes two tiles – the 'f' tile we've already met, and a rotated version (see Figure 23). Note that they both use T0, T1, and T2 as their point labels, but their tile IDs can be different.

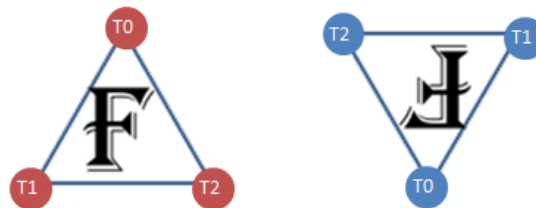


Figure 23. The 'f' and 'rot' Tiles.

The rotated version can be created with one line of code:

```
Tile rot = tri.rotate(180);  
rot.setID("rot"); // this does not change the point names
```

The call to `Tile.setID()` is optional, but it useful for distinguishing between the tiles.

How could the 'rot' tile be positioned next to the 'f' tile? The trick is to think in terms of positioning their points. Once the 'f' tile has been drawn to the screen, its T0 point has a tiling location which can be used to position the 'rot' tile using its T2 point (see Figure 24).

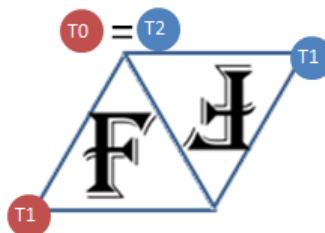


Figure 24. Positioning the 'rot' Tile next to the 'f' Tile.

This can be coded like so:

```
BufferedImage scrIm = iview.getImage();  
  
tri.drawAt(scrIm, "T0", 30, 0);  
Point2D.Double pt = tri.getPtLoc("T0");  
rot.drawAt(scrIm, "T2", pt);  
iview.repaint();
```

Once the 'rot' tile has been drawn, then its T1 tiling location can be used to redraw the 'f' tile (see Figure 25).

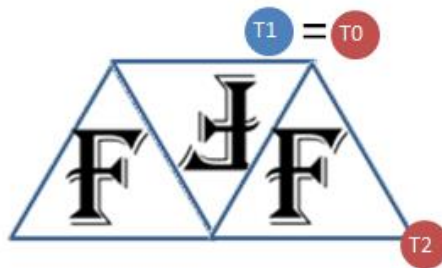


Figure 25. Adding Another 'f' Tile to the Image.

This is coded like so:

```
pt = rot.getPtLoc("T1");  
tri.drawAt(scrIm, "T0", pt);  
iview.repaint();
```

The resulting tiling is shown in Figure 26.

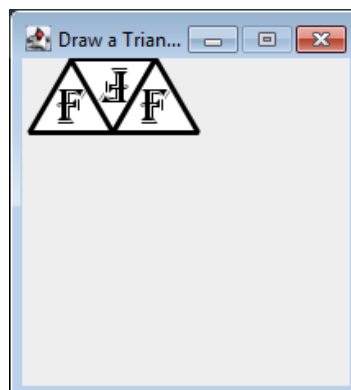


Figure 26. Tiling Carried out in DrawATriangle.java

Note that the calls to `ImageViewer.repaint()` cause the tiling window to be refreshed, which make the changes to the window visible on the screen.

This technique of moving along a row by drawing a new tile based on its predecessor's location is at the heart of the row-by-row tiling approach described in the next section.

The same idea can be employed to draw tiles down the image. The example in Figure 27 draws the T2 point of the 'rot' tile using the 'f' tile's T1 location.

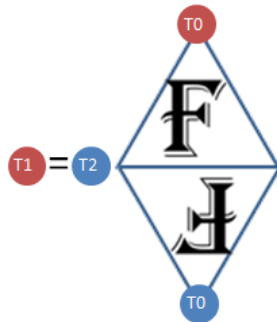


Figure 27. Positioning the 'rot' Tile below the 'f' Tile.

This is coded as:

```
BufferedImage scrIm = iview.getImage();

tri.drawAt(scrIm, "T0", 30, 0);
Point2D.Double pt = tri.getPtLoc("T1");
rot.drawAt(scrIm, "T2", pt);
iview.repaint();
```

which results in Figure 28.

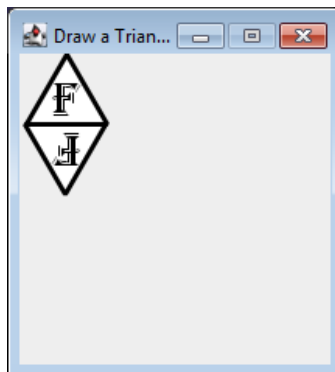


Figure 28. Vertical Tiling in DrawATriangle.java

Incidentally, there are other ways of obtaining these tiling patterns. For instance, the vertical tiling could use the location of 'f's T2 point to position 'rot's T1.

### 3. Row-by-Row Tiling

The jFAT download includes many examples of row-by-row tiling (e.g. see DrawTris.java, DrawArrows.java, DrawBoxes.java, DrawHexs.java, Stars.java, Peanuts.java, and Axes.java). I'll look at DrawTris.java and DrawArrows.java in detail, focusing on how a {draw, down, next} triplet of points allows rows and columns of tiles to be generated using nothing more fancy than nested loops. This is simply a generalization of the approach I used in the last section.

#### 3.1. Drawing Triangles

DrawTris.java uses two equilateral triangles containing the letter 'f', with one a 180 degree rotation of the other. Unlike in DrawATriangle.java, the tiles are created using a trif.png image (see Figure 29).

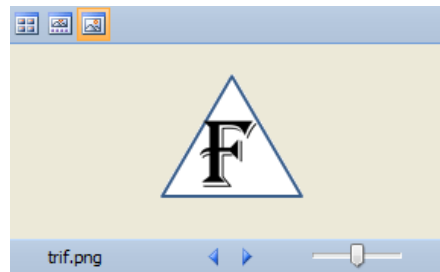


Figure 29. trif.png.

Since the Tile constructor cannot determine points from a plain image, they have to be supplied by the programmer:

```
Tile t = new Tile("data/trif.png");
t.addPt("T0", 51,6); // added in ccw order
t.addPt("T1", 4,86); // coords obtained by viewing the image
t.addPt("T2", 97,86);

Tile rot = t.rotate(180);
rot.setID("rot");
```

Then the tiling image is created:

```
ImageViewer iview = Pics.view("Triangles row-by-row", Pics.altScreen());
BufferedImage scrIm = iview.getImage();
int pWidth = scrIm.getWidth();
int pHeight = scrIm.getHeight();
```

Before I explain the tiling code, it helps to see what pattern I'm trying to achieve. Figure 30 shows a screenshot of the final output.

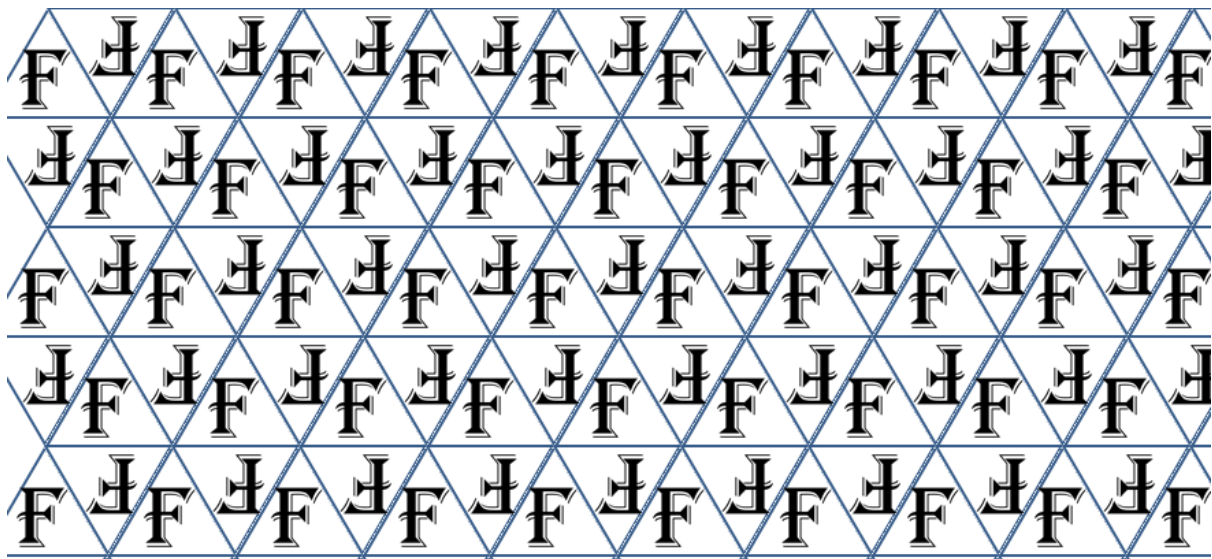


Figure 30. Triangles Tiled by DrawTris.java

A {draw, down, next} triplet states how the current tile in a row ('draw') is related to the tile in the next row down ('down'), and to the next tile in the row ('next').

The regularity of the tiling pattern in Figure 30 suggests a few ways of defining the triplet. One approach would be to create a new tile composed from two 'f' tiles and two 'rot' tiles (see Figure 31), but I'll leave off explaining composition until the next section.



Figure 31. A Tile Composed from two 'f' and two 'rot' Tiles.

Instead, I'll formulate two {draw, down, next} triplets, as shown in Figure 32.

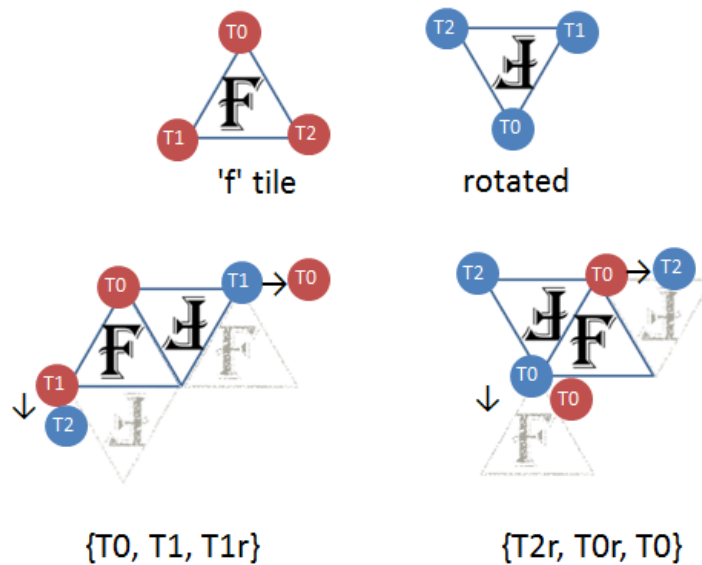


Figure 32. Two {draw, down, next} triplets for Drawing Triangles. The 'r' in the labels stands for "rotated".

The first row of tiles (see Figure 30) is controlled by the left hand triplet which involves printing an 'f' tile using its T0 point followed by a 'rot' tile. The 'next' point is used to repeat this output , and the 'down' point is used to start the next row.

The second row of tiles is controlled by the right hand triplet which involves printing a 'rot' tile using its T2 point and then a 'f' tile. The 'next' point is used to repeat this tiling, and the 'down' point is used to start the next row.

These triplets alternate, the left one controlling the drawing of odd rows and the right triplet managing the even rows. This is translated into the code:

```
// display tiles row-by-row
int rowNum = 1;
Point2D.Double startPt = new Point2D.Double(30,0);
Point2D.Double nextPt;

while (startPt.y < pHeight) {
    // start of a row
    if (rowNum%2 == 1) { // odd row; t then rot
        t.drawAt(scrIm, "T0", startPt); // draw at T0
        rot.drawAt(scrIm, "T2", startPt);
        startPt = t.getPtLoc("T1"); // down is T1
        nextPt = rot.getPtLoc("T1"); // next is T1r
    }
}
```



```

else { // even row; rot then t
    rot.drawAt(scrIm, "T2", startPt); // draw at T2r
    t.drawAt(scrIm, "T0", rot.getPtLoc("T1"));
    startPt = rot.getPtLoc("T0"); // down is T0r
    nextPt = t.getPtLoc("T0"); // next is T0
}

// rest of row
while (nextPt.x < pWidth) {
    if (rowNum%2 == 1) { // odd row; t then rot
        t.drawAt(scrIm, "T0", nextPt); // draw at T0
        rot.drawAt(scrIm, "T2", nextPt);
        nextPt = rot.getPtLoc("T1"); // next is T1r
    }
    else { // even row; rot then t
        rot.drawAt(scrIm, "T2", nextPt); // draw at T2r
        t.drawAt(scrIm, "T0", rot.getPtLoc("T1"));
        nextPt = t.getPtLoc("T0"); // next is T0
    }
    iview.repaint();
}
Pics.pause(100); // delay after each row is drawn
rowNum++;
}

```

At the start of a new row (i.e. at the start of the outer loop) there's an if-test which stores the 'down' information for starting the next row (in the startPt variable), and a second variable for 'next' (nextPt). Note that there are always two calls to Tile.drawAt() since each drawing uses an 'f' and a 'rot' tile.

The inner loop manages the rest of the printing of the row, so only the nextPt variable needs to be updated.

The calls to ImageViewer.repaint() and the use of Pics.pause() mean that the tiling image is refreshed a row at a time, and there's a brief pause after each row is drawn. This delay isn't necessary for the implementation but produces a nice animation effect at run time.

### 3.2. Drawing Arrows

A screenshot of the execution of DrawArrows.java is shown in Figure 33.

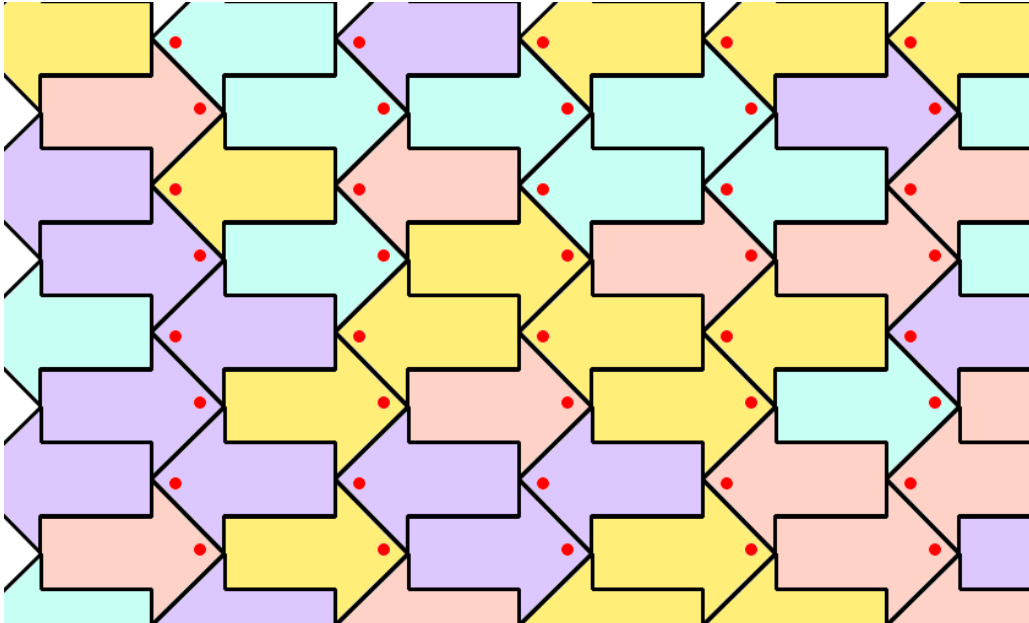


Figure 33. Arrows Tiled by DrawArrows.java

The left-facing arrow is defined in arrowTile.txt:

```
Tile 100 20
225 84.8 90 84.8
135 30 270 90
90 30 0 30
90 90 $
```

Figure 34 show how these lengths and turning angles map to the shape, starting from point A0 at the top and moving counterclockwise.

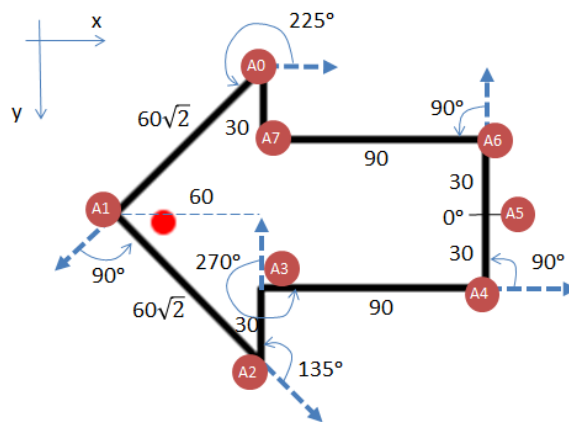


Figure 34. Turning Angles and Lengths for the Arrow Tile.

The angle used for A0 is 225 degrees since the 'turtle' is assumed to be facing along the +x axis before it begins its journey around the shape.

The arrow head lengths are approximated by 84.8, based on their actual lengths of  $60\sqrt{2}$ .

It's far from clear why an A5 point has been included half way along the back edge of the arrow. It will become useful when the {draw, down, next} triplets are formulated.

DrawArrows.java loads the tile description to create a left-facing arrow, and then rotates it 180 degrees to create a right-facing version.

```
// create left- and right- pointing arrow tiles
Tile left = new Tile("data/arrowTile.txt");
left.setID("left");
Graphics2D g2d = left.getGraphics();
// red circle
g2d.setColor(Color.RED);
g2d.fillOval(55, 78, 10, 10); // near to the arrow tip

Tile right = left.rotate(180);
right.setID("right"); // this will not affect the point labels
```

As in DrawTris.java, the tiling will be made up of odd and even rows – the odd rows consist of left-facing arrows, and even rows are a series of right-facing arrows. Each of these require its own {draw, down, next} triplet. Figure 35 shows the two types of row, and their triplets.

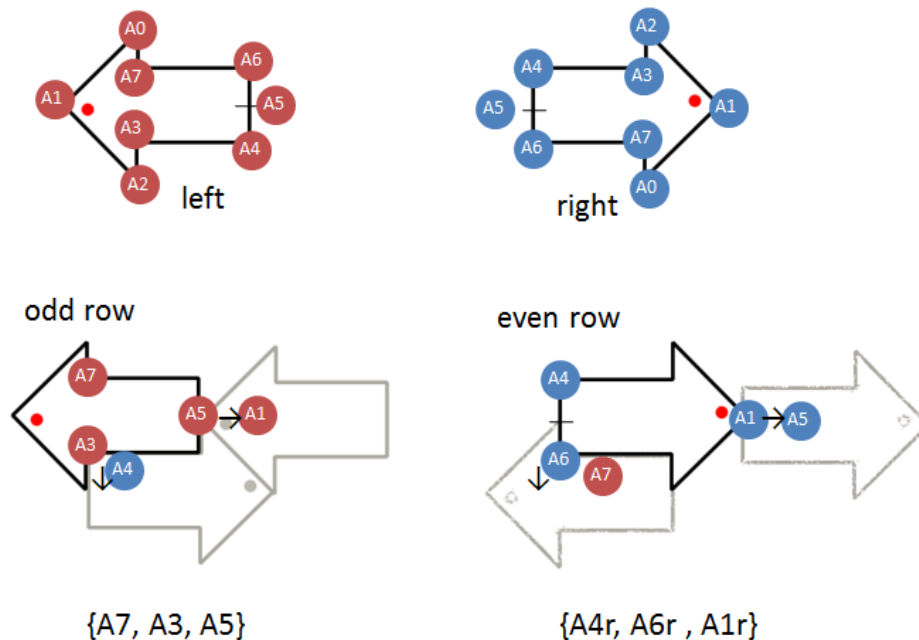


Figure 35. Two {draw, down, next} triplets in DrawArrows.java. The 'r' in the labels stands for "rotated".

The nested loops for generating the tiling row-by-row are similar to the DrawTris.java example – the start of the outer loop decides whether this row is odd or even, and stores the 'down' point for the next row in startPt. The inner loop progresses along the row (be it odd or even) using the 'next' argument.

Note that the 'draw' point changes inside this loop to be the A1 point of the left arrow on even rows, and the A5 point of the right arrow on odd rows. For that reason it may be more accurate to write the triplets as {A7/A1, A3, A5} and {A4r/A5r, A6r, A1r}.

```

ImageViewer iview = Pics.view("Draw Arrows", Pics.altScreen());
BufferedImage scrIm = iview.getImage();
int pWidth = scrIm.getWidth();
int pHeight = scrIm.getHeight();

// display tiles row-by-row
int rowNum = 1;
Point2D.Double startPt = new Point2D.Double(30,0);
Point2D.Double nextPt;
Tile t; // can be either a left or right arrow

while (startPt.y < pHeight) {
    // start of a row
    if (rowNum%2 == 1) { // left arrow
        t = randColor(left);
        t.drawAt(scrIm, "A7", startPt); // draw at A7
        startPt = t.getPtLoc("A3"); // down is A3
        nextPt = t.getPtLoc("A5"); // next is A5
    }
    else { // right arrow
        t = randColor(right);
        t.drawAt(scrIm, "A4", startPt); // draw at A4r
        startPt = t.getPtLoc("A6"); // down is A6r
        nextPt = t.getPtLoc("A1"); // next is A1r
    }

    // rest of row
    while (nextPt.x < pWidth) {
        if (rowNum%2 == 1) { // left arrows
            t = randColor(left);
            t.drawAt(scrIm, "A1", nextPt); // draw at A1
            nextPt = t.getPtLoc("A5"); // next is A5
        }
        else { // right arrows
            t = randColor(right);
            t.drawAt(scrIm, "A5", nextPt); // draw at A5r
            nextPt = t.getPtLoc("A1"); // next is A1r
        }
    }
    iview.repaint();
}

```

```

Pics.pause(100); // delay after each row is drawn
rowNum++;
}

```

Another change is the use of the `randColor()` to select a color at random to modify the tile:

```

private static final Color[] colors = {
    new Color(199,255,244), new Color(255, 210, 199),
    new Color(255, 238, 119), new Color(221, 199, 255)
};

private static Random rand = new Random();

private static Tile randColor(Tile t)
{ return t.colorChg( colors[rand.nextInt(colors.length)]); }

```

## 4. Tile Composition

Tile composition creates a new tile by combining simpler tiles. As an example, I'll explain how to build the composite tile in Figure 31 from four equilateral triangle tiles. The composite will be used to tile the screen, producing the same effect as in Figure 30, but with only one kind of row.

At the heart of composition is the idea that a tile (or tiles) can be drawn to any image, not just one managed by `ImageViewer`. The resulting image can be used to create a new tile in the same way as explained in section 1.3.

### 4.1. Creating a Composite from Triangles

`DrawCTris.java` begins by creating an equilateral triangle, and a rotated version, in the same way as `DrawTris.java` in section 3.1. It uses these triangles to build the composite shown on the right of Figure 36.

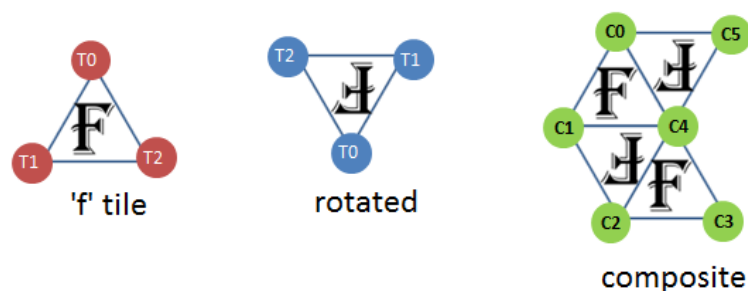


Figure 36. The Tiles Used in `DrawCTris.java`.

The corresponding code:

```
private static Tile makeComposite()
{
    Tile t = new Tile("data/trif.png");
    t.addPt("T0", 51,6);
    t.addPt("T1", 4,86);
    t.addPt("T2", 97,86);
    t.view();
    t.reportPoints();

    Tile rot = t.rotate(180);
    rot.setID("rot");

    BufferedImage im = new BufferedImage(TILE_WIDTH, TILE_HEIGHT,
        BufferedImage.TYPE_INT_ARGB); // alpha channel
    int xc = TILE_WIDTH/2; // center of image
    int yc = TILE_HEIGHT/2;

    Point2D.Double[] corners = new Point2D.Double[6];

    /* The composite is four triangles drawn around a point;
       the corner points are collected */
    t.drawAt(im, "T2", xc, yc);
    corners[0] = t.getPtLoc("T0");
    corners[1] = t.getPtLoc("T1");

    rot.drawAt(im, "T1", xc, yc);
    corners[2] = rot.getPtLoc("T0");

    t.drawAt(im, "T0", xc, yc);
    corners[3] = t.getPtLoc("T2");

    rot.drawAt(im, "T0", xc, yc);
    corners[4] = rot.getPtLoc("T0");
    corners[5] = rot.getPtLoc("T1");

    // create composite tile from the image, and add its corners
    Tile ctTile = new Tile(im, "comp");
    for (int i=0; i < corners.length; i++)
        ctTile.addPt("C"+i, corners[i]);

    ctTile.view();
    ctTile.reportPoints();
    return ctTile;
} // end of makeComposite()
```

makeComposite() creates a BufferedImage object called im with an alpha channel so that its background will be transparent. Then two 'f' tiles and two 'rot' tiles are drawn around its

center, and important coordinates are stored in a corners[] array. These are used at the end to add points to the composite tile. Figure 37 shows the output generated by Tile.view() and Tile.reportPoints() at the end of the function.

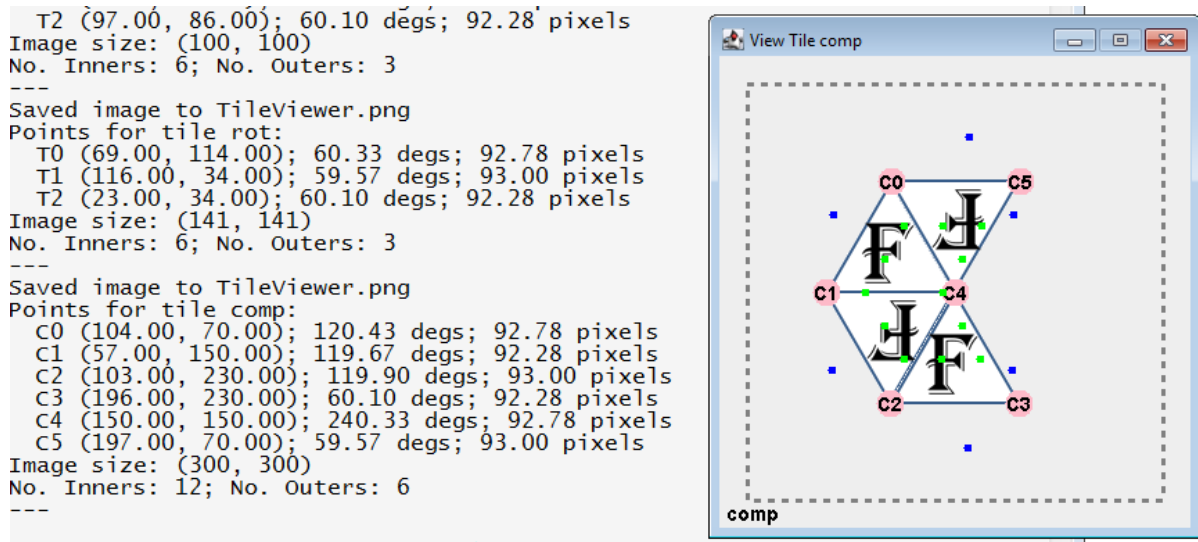


Figure 37. The Composite Tile.

Only a single {draw, down, next} triplet is needed for this composite tile, as shown in Figure 38.

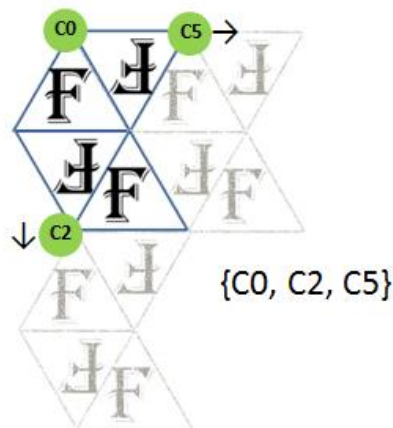


Figure 38. The {draw, down, next} triplet used in DrawCTris.java.

The triplet translates into a nested loop which doesn't require a rowNo variable for extra if-tests:

```

ImageViewer iview = Pics.view("Using a Composite",
                               Pics.altScreen());
BufferedImage scrIm = iview.getImage();
int pWidth = scrIm.getWidth();
int pHeight = scrIm.getHeight();

Point2D.Double startPt = new Point2D.Double(30,0);
Point2D.Double nextPt;

while (startPt.y < pHeight) {
    // start of a row
    ct.drawAt(scrIm, "C0", startPt); // draw at C0
    startPt = ct.getPtLoc("C2"); // down is C2
    nextPt = ct.getPtLoc("C5"); // next is C5

    // rest of row
    while (nextPt.x < pWidth) {
        ct.drawAt(scrIm, "C0", nextPt); // draw at C0
        nextPt = ct.getPtLoc("C5"); // next is C5
        iview.repaint();
    }
    Pics.pause(100); // delay after each row is drawn
}

```

## 4.2. Composites for Penrose Tiles

The Penrose P2 tiles (the dart and the kite; see Figure 7) will be utilized in section 9 by the built-in tiling functions, `tileLocs()` and `tileSpacey()`. The results will be fairly good, although a few unsightly gaps will appear in the tessellation. One way to address this problem is to utilize composite tiles; the seven types [3] are shown in Figure 39.

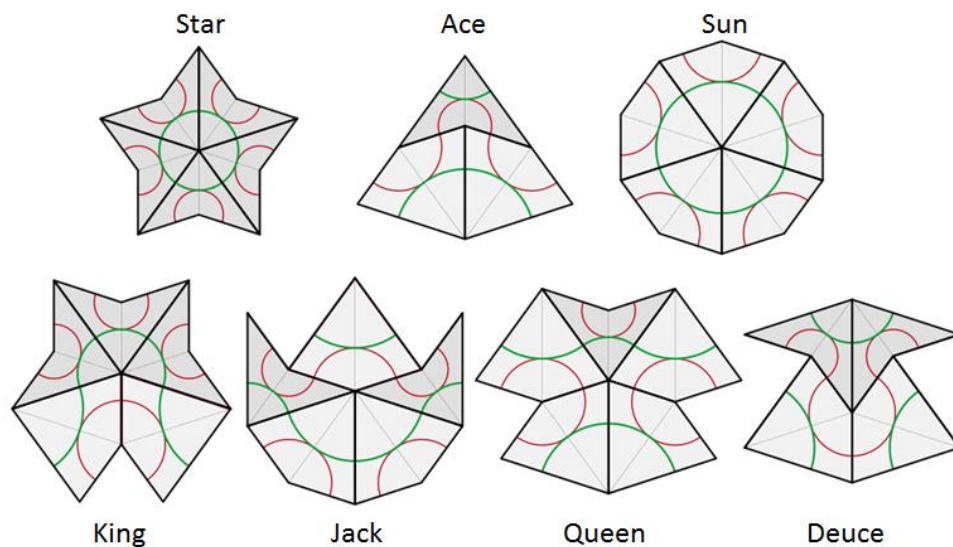


Figure 39. Composite Tiles Using Penrose's Dart and Kite.



The composites are simpler to work with because there's less choice in how they can be placed together as compared to the more versatile dart and kite.

PenroseClusters.java contains seven functions (e.g. makeAceTile()) which build these composites ) The program displays the windows shown in Figure 40.

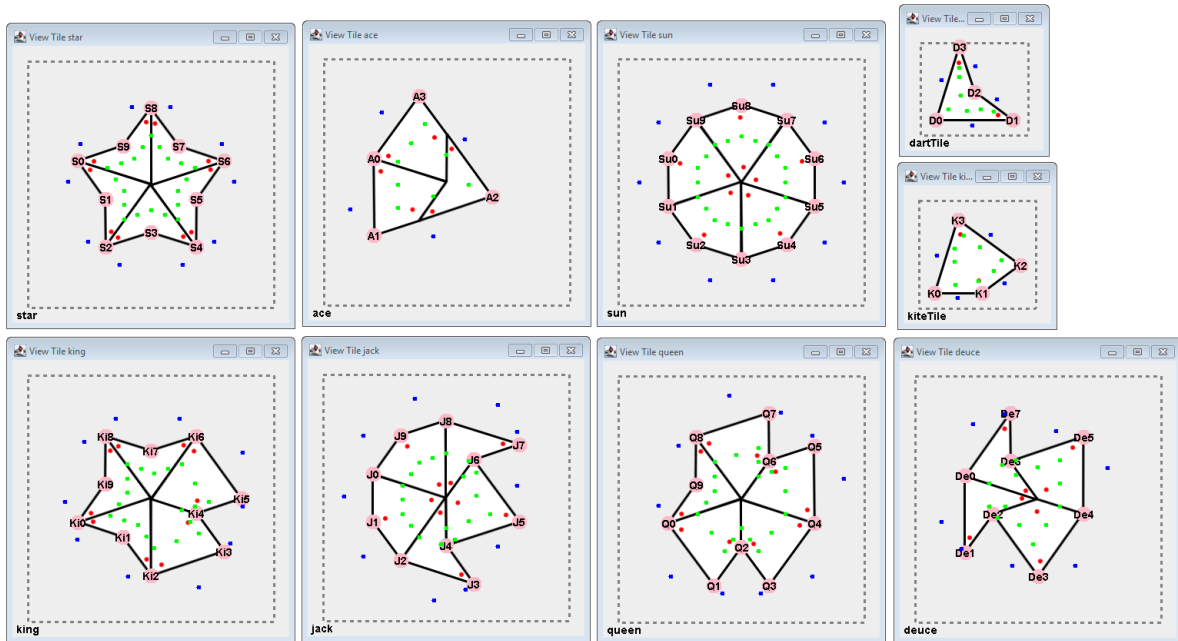


Figure 40. Composites made by PenroseClusters.java.

I'll briefly look at how makeAceTile() produce the Ace tile.

The dart and kite are defined in tile data files, as explained back in section 1.2, and their points are defined as in Figure 41.

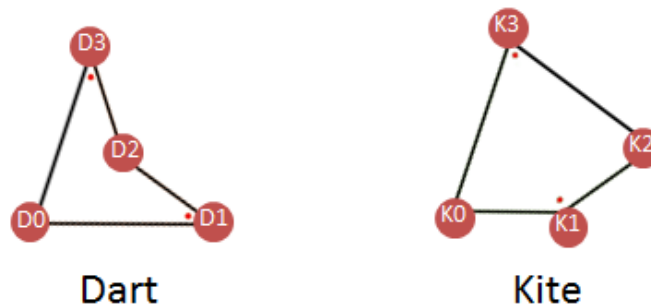


Figure 41. Labeled Dart and Kite Tiles.

An Ace is built from two kites and a dart, which are rotated and drawn onto a blank image. Deciding on the rotation values requires a good knowledge of the angles used in the dart and kite (see Figure 6), but I eventually went for the angles illustrated in Figure 42.

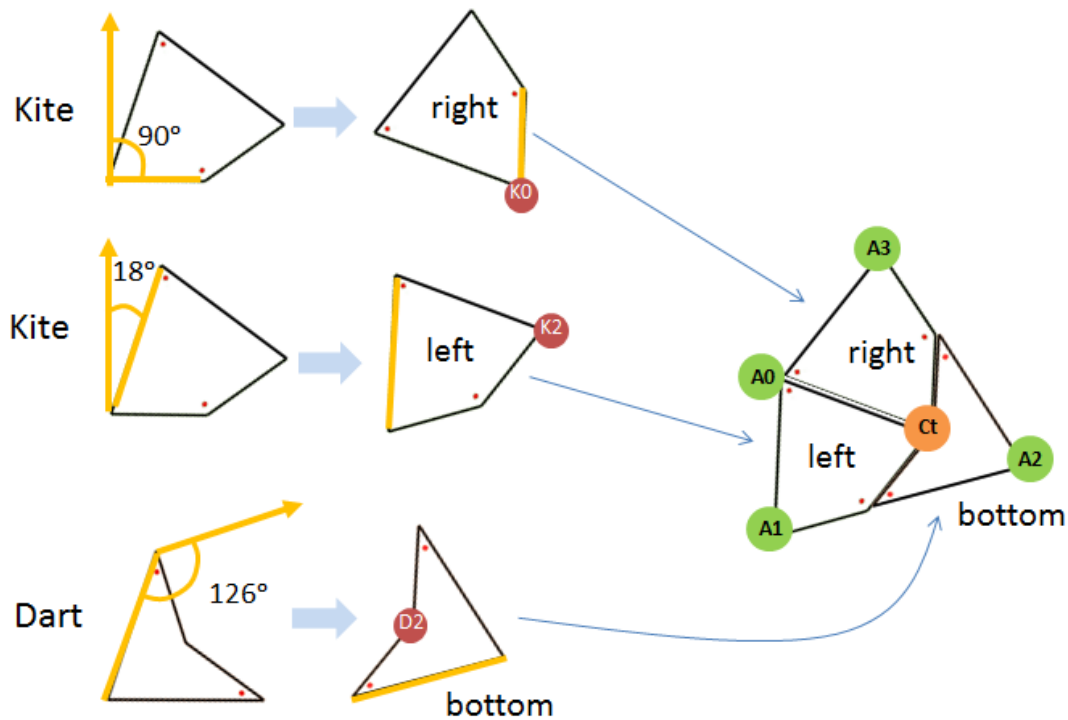


Figure 42. Rotating and Positioning Tiles to make an Ace.

Although Figure 42 shows the angles relative to a corner, `Tile.rotate()` applies a rotation to a tile around its center. Each one is positioned in the final image by drawing the red-colored points (K0, K2 and D2) at the point "ct".

The coordinates for the Ace (labeled as green circles in Figure 42) are collected as each tile is drawn onto the image, and used in calls to `Tile.addPts()` at the end of `makeAceTile()`:

```
private static Tile makeAceTile()
// generate an 'Ace' Penrose cluster
{
    int w = TILE_WIDTH*2;
    int h = TILE_HEIGHT*2;
    BufferedImage im = new BufferedImage(w, h,
        BufferedImage.TYPE_INT_ARGB); // alpha channel

    Point2D.Double[] corners = new Point2D.Double[4];

    // Draw an 'Ace' around center of image, and collect corners
    Tile leftKite = kite.rotate(18);
    leftKite.drawAt(im, "K2", w/2, h/2);
```

```

corners[0] = leftKite.getPtLoc("K3"); // will become A0
corners[1] = leftKite.getPtLoc("K0"); // A1

Tile rightKite = kite.rotate(90);
rightKite.drawAt(im, "K0", w/2, h/2);
corners[3] = rightKite.getPtLoc("K2"); // A3

Tile bottomDart = dart.rotate(36+90);
bottomDart.drawAt(im, "D2", w/2, h/2);
corners[2] = bottomDart.getPtLoc("D0"); // A2

// create a tile using the image and corners
Tile t = new Tile(im, "ace");
for (int i=0; i < 4; i++)
    t.addPt("A"+i, corners[i]);
return t;
} // end of makeAceTile()

```

### 4.3. Reproducing Escher's "Square Limit"

One of my aims for jFAT was to make it powerful enough to generate Escher tessellations such as "Square Limit" [8] shown in Figure 44.

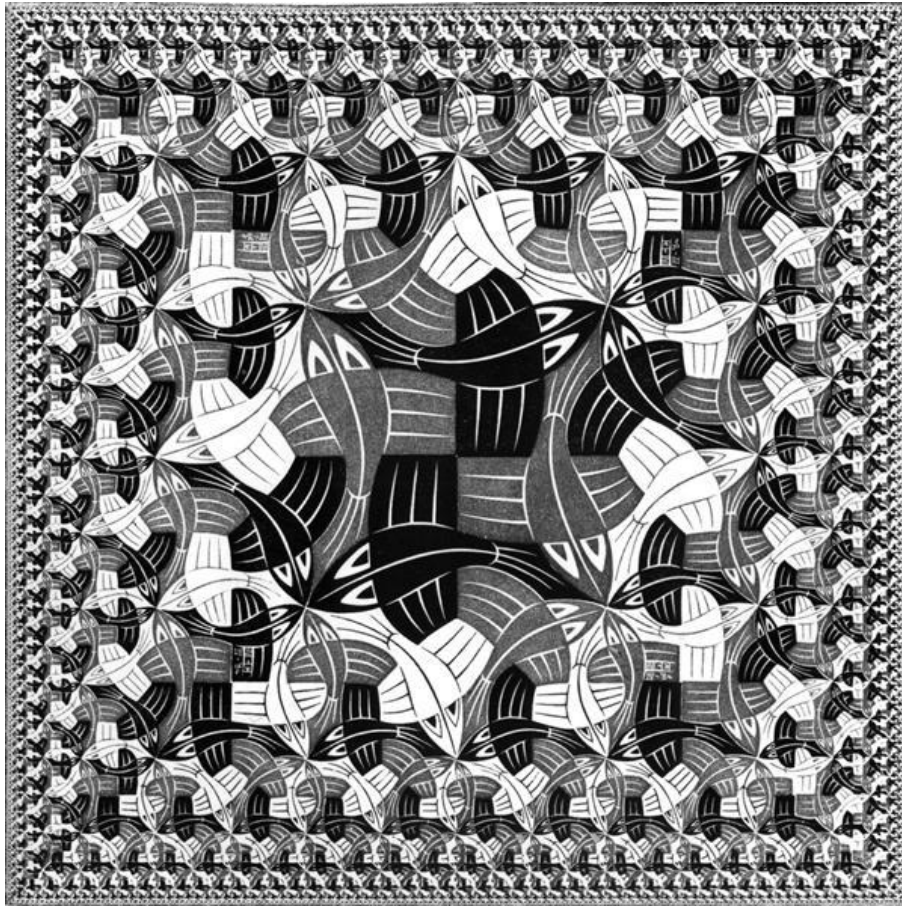


Figure 44. “Square Limit” by M.C. Escher (1964).  
<https://www.wikiart.org/en/m-c-escher/square-limit>

DrawFish.java is my attempt at reproducing the picture, with its output displayed in Figure 45.

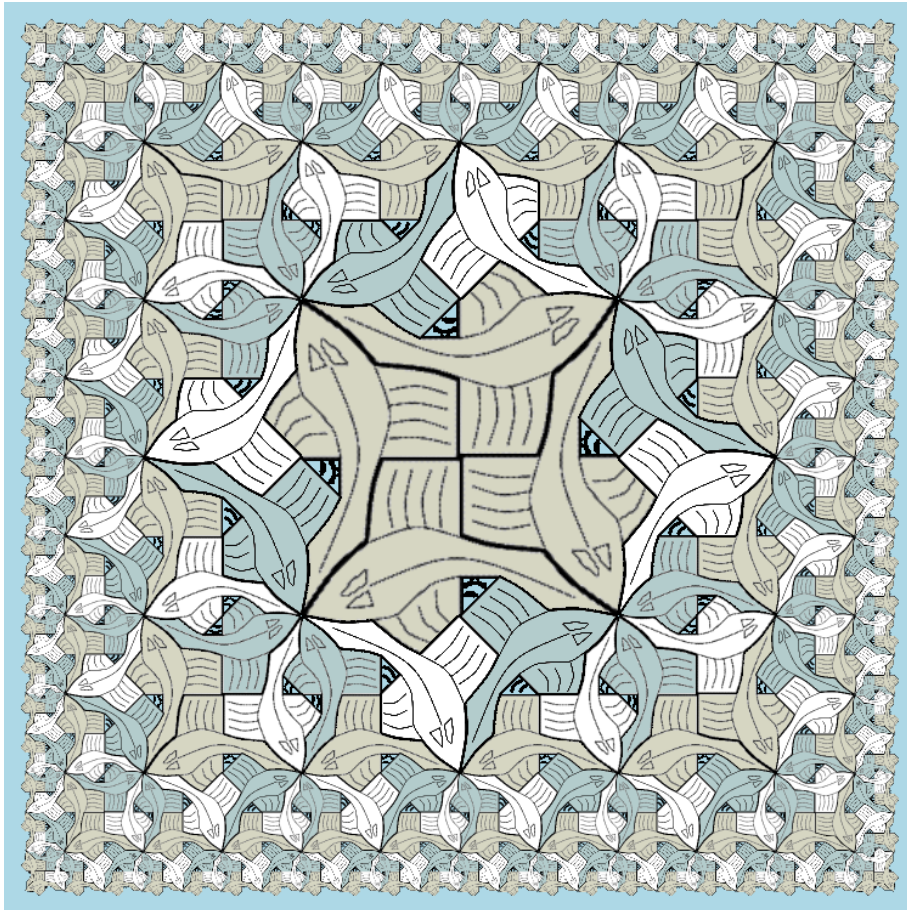


Figure 45. The Output of DrawFish.java.

Before I began coding DrawFish.java I thought I would need to utilize recursive but, to my surprise, it was easier to view the drawing as a series of tile compositions, starting from a single 'fish' tile, building up to a tile representing a quarter of the final image; this 'quarter' tile was rotated three times to produce Figure 45.

The quarter I focused upon is shown in Figure 46, with its component tiles outlined in red.

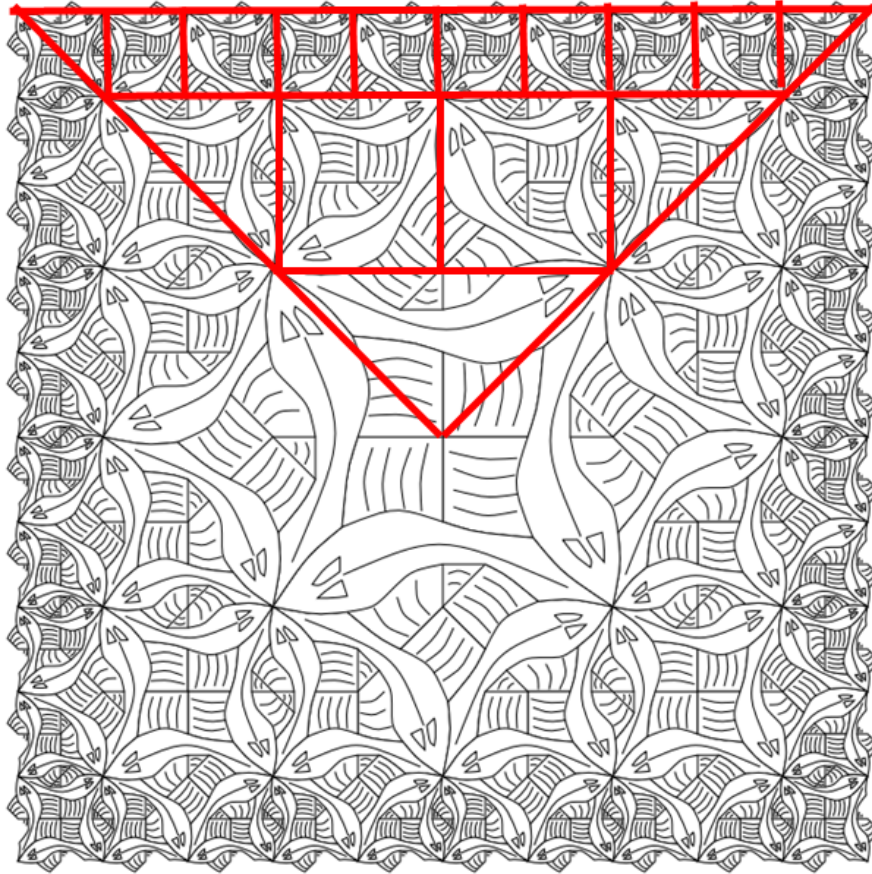


Figure 46. The Quarter Tile and its Components.

The similarities between the component squares and triangles is a little easier to see if they are labeled as in Figure 47.

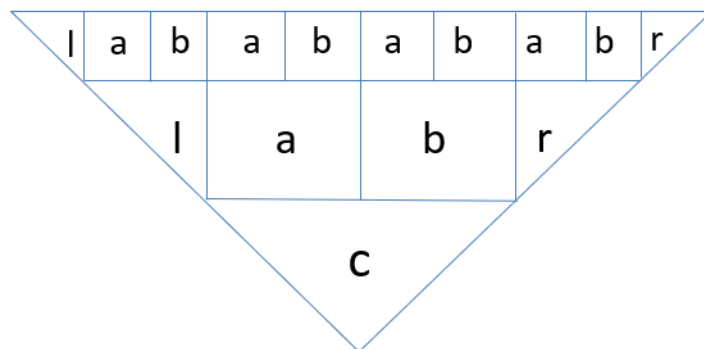


Figure 47. The Labeled Components of the Quarter Tile.

If the issue of size is ignored, the quarter is made from five unique tiles labeled as "c", "l", "a", "b", and "r". Also, an "a" tile is always paired with a "b" tile, and so we could reduce the

five tiles to four. (Incidentally, if we're prepared to ignore rotations as well, then the "b" tile is just the "a" tile rotated clockwise by 90 degrees, but I'm not going to follow that approach.)

Ignoring the "c" tile for a moment, Figure 48 shows enlarged versions of the "l", "a", "b", and "r" tiles.

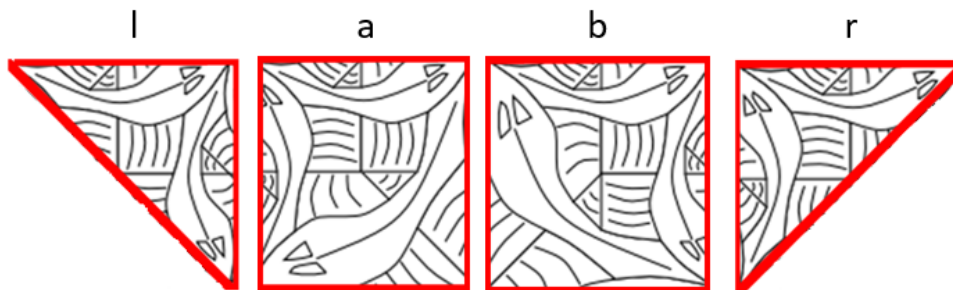


Figure 48. The "l", "a", "b", and "r" Tiles.

These tiles are composed from a single 'fish' tile which has been rotated and sometimes resized. This is made clear by adding dotted lines in Figure 49.

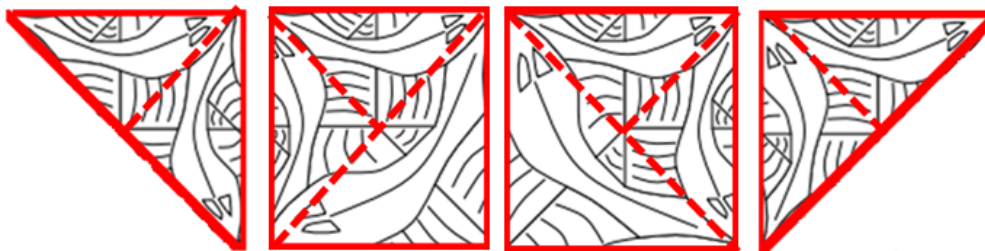


Figure 49. The 'fish' Components of "l", "a", "b", and "r".

It's now necessary to create a tile for every 'fish' orientation employed in Figure 49. Six tiles are needed, as shown in Figure 50.

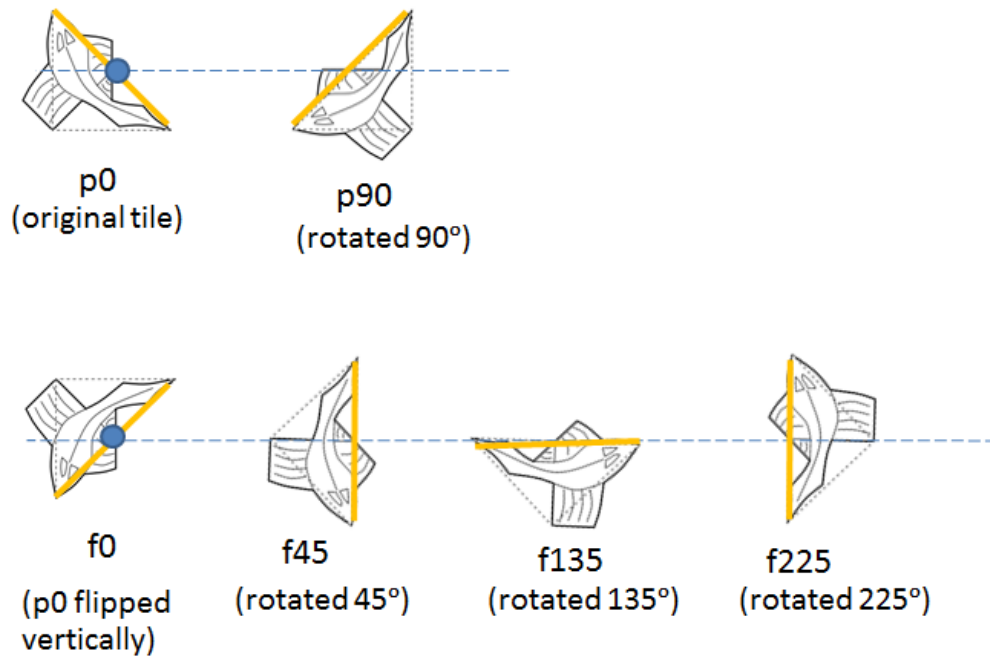


Figure 50. Rotated and Flipped Fish Tiles.

"p0" is the original 'fish', and the others tiles are rotated and flipped versions of that one.

DrawFish.java will start by generating the six fish tiles in Figure 50, and then combine them to create the "l", "a", "b", and "r" tiles of Figure 49. In turn, they will be composed in the style of Figure 47 to make a 'quarter' tile, and that tile will be rotated to generate the final image.

Although I'm going to describe all of DrawFish.java at once, it's useful to note that I actually wrote it in the three steps outlined in the previous paragraph. At each stage, I used `Tile.view()` and `Tile.reportPoints()` to check that the composites were correct before moving onto generating the next larger tile.

### 4.3.1. Building the Basic Fish

The original fish image, and the other five fish tiles are created like so:

```
// in DrawFish.java
tilesMap = new HashMap<String,Tile>();

Tile fish = new Tile("data/fish.png");
fish.addPt("Head", 35,33);
fish.addPt("Left", 35, 163);
fish.addPt("Right", 100, 100); // at the center of this image
fish.addPt("Tail", 165, 163);
```



```

// store the fish for later
tilesMap.put("p0", fish);
tilesMap.put("p90", fish.rotate(90).
                colorChg(Color.WHITE, GRAY_GREEN));

Tile flippedFish = fish.flip(Pics.HORIZ);
tilesMap.put("f45", flippedFish.rotate(45).
                colorChg(Color.WHITE, GRAY_GREEN));
tilesMap.put("f135", flippedFish.rotate(135).
                colorChg(Color.WHITE, LIGHT_BROWN));
tilesMap.put("f225", flippedFish.rotate(225));

```

The "p0" fish is shown in Figure 51, along with its labeled points.

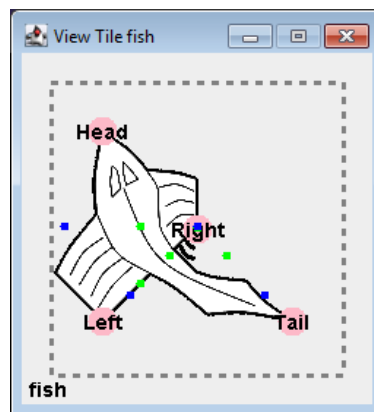


Figure 51. The "p0" Fish.

All of these fish are stored in a global HashMap to make them easier to access later.

I've also broken with 'tradition' by using descriptive names for the point labels; this makes it considerably easier to visualize the more complex drawings.

### 4.3.2. The Mid-level Fish

The "l", "a", "b", and "r" fish are created in their own functions:

```

// composite tiles
tilesMap.put("mid", makeMid()); // the "a"+"b" tiles
tilesMap.put("rightEnd", makeRightEnd()); // the "l" tile
tilesMap.put("leftEnd", makeLeftEnd()); // the "r" tile

```

I'll explain how the "mid" tile (a combination of "a" and "b") is made by referring to Figure 52 which labels those tiles with their simpler fish elements from Figure 49.

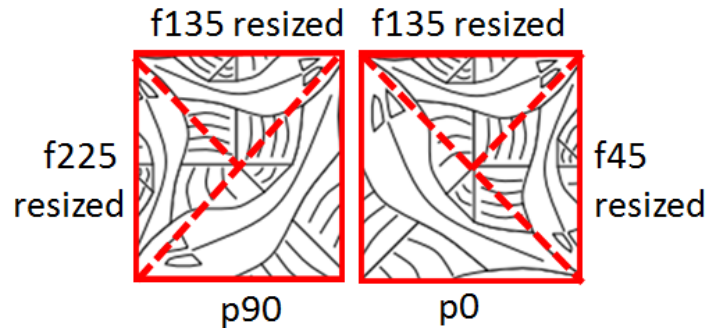


Figure 52. Building the "mid" Tile.

The resizing noted in Figure 52 is by a factor of  $\sqrt{2}/2$ .

makeMid() closely follows Figure 52, with the added requirement to gather the corner points of the new tile, which are passed to Tile.addPt() at the end of the function.

```
private static Tile makeMid()
{
    BufferedImage im = new BufferedImage(460, 260,
        BufferedImage.TYPE_INT_ARGB); // alpha channel

    Point2D.Double startPt = new Point2D.Double(15, 15);
        // near the top left point

    Point2D.Double[] corners = new Point2D.Double[4];

    // left side square (tile 'a') -----

    // tile along the top of the square
    Tile top = tilesMap.get("f135").resize(SCALE);
    top.drawAt(im, "Tail", startPt);
    corners[3] = startPt;

    // tile up the left side of the square
    Tile left = tilesMap.get("f225").resize(SCALE);
    left.drawAt(im, "Head", startPt);
    corners[0] = left.getPtLoc("Tail");

    // tile in the bottom right corner of the square
    Tile right = tilesMap.get("p90");
```

```

right.drawAt(im, "Head", corners[0]);

Point2D.Double rightPt = right.getPtLoc("Tail");

// right side square (tile 'b') -----

// tile along the top of the square
top = tilesMap.get("f135").resize(SCALE);
top.drawAt(im, "Tail", rightPt);
corners[2] = top.getPtLoc("Head");

// tile down the right side of the square
right = tilesMap.get("f45").resize(SCALE);
right.drawAt(im, "Tail", corners[2]);
corners[1] = right.getPtLoc("Head");

// tile in the bottom left corner of the square
left = tilesMap.get("p0");
left.drawAt(im, "Head", rightPt);

Tile midTile = new Tile(im, "mid");
for (int i=0; i < 4; i++)
    midTile.addPt("C"+i, corners[i]);

return midTile;
} // end of makeMid()

```

makeLeftEnd(), makeMid(), and makeRightEnd() create the composite tiles shown in Figure 53. They are stored in the global HashMap with the names "leftEnd", "mid", and "rightEnd".

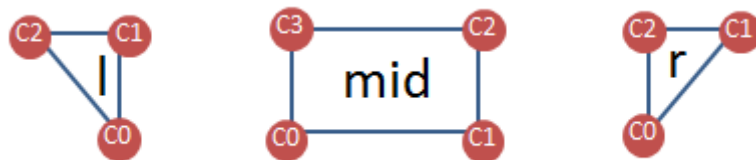


Figure 53. The 'l', 'mid', and 'r' Tiles.

### 4.3.3. Building the 'quarter' Tile

makeQuarter() builds Figure 47, which is redrawn in Figure 54 to use the 'l', 'mid' and 'r' tiles, and to include start points and three levels.

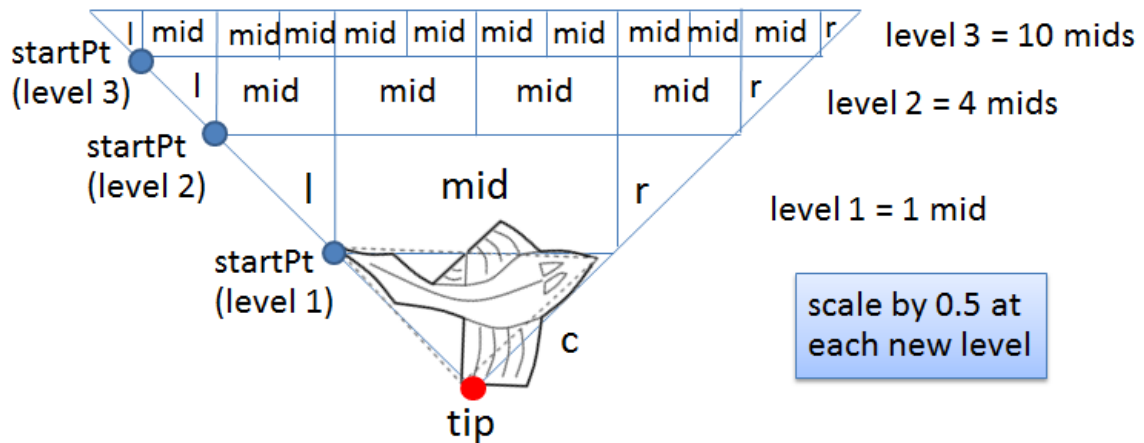


Figure 54. Drawing the 'quarter' Tile.

After the 'c' tile has been drawn, the rest of the quarter can be viewed as three levels (or rows) which start with an 'l' tile, then a certain number of 'mid' tiles, and finishes with a 'r' tile. For each level, it's necessary to calculate the start point, the number of 'mid' tiles, and to resize the tiles appropriately for that level. `makeQuarter()` delegates most of that work to a `drawLevels()` function:

```
private static Tile makeQuarter()
{
    BufferedImage im = new BufferedImage(1000, 500,
        BufferedImage.TYPE_INT_ARGB); // alpha channel

    Point2D.Double tip = new Point2D.Double(500,500);
    // bottom edge, in the middle of the drawing area

    Tile cTile = tilesMap.get("f135").resize(1.0/SCALE); // enlarge
    cTile.drawAt(im, "Left", tip);

    Point2D.Double startPt = cTile.getPtLoc("Tail");

    Pair<Point2D.Double, Point2D.Double> endPts =
        drawLevels(1, 3, 1, im, startPt);
    // returns <left, right> top corners of quarter

    Tile quarTile = new Tile(im, "quarter");
    quarTile.addPt("C0", tip);
    quarTile.addPt("C1", endPts.getY()); // top right corner
    quarTile.addPt("C2", endPts.getX()); // top left corner
    return quarTile;
} // end of makeQuarter()
```

After drawLevels() has finished the three levels, the final image is converted into a tile which has the triangular form shown in Figure 55.

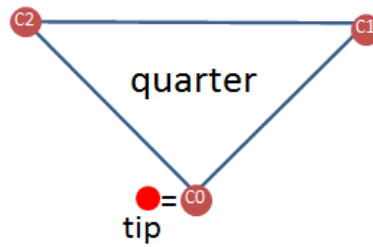


Figure 55. The 'quarter' Tile.

Although I started this subsection by saying that DrawFish.java does not use recursion, that's not quite true since drawLevels() is tail-recursive. However, it could be rewritten to be iterative, so the recursion isn't essential.

Each call to drawLevels() constructs a single level, like the one depicted in Figure 56.

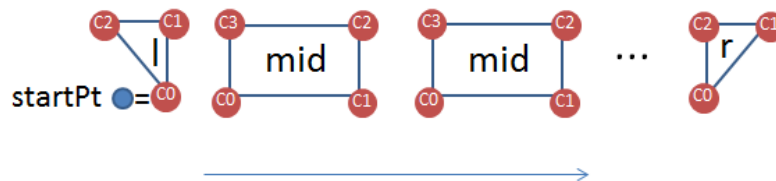


Figure 56. Drawing One Level with drawLevels().

The row begins at startPt, with the 'l' tile drawn using its C0 point. A loop renders the 'mid' tiles by drawing them with their C3 points, and linking the next 'mid' tile to the C2 point of the current tile. Finally the 'r' tile is drawn with its C2 point.

```
private static Pair<Point2D.Double, Point2D.Double>
    drawLevels(int level, int n, int numMids,
              BufferedImage im, Point2D.Double startPt)
{ if (level > n) {
    System.out.println("Level error");
    return null;
  }

  double lvlScale = Math.pow(0.5, level-1);

  // draw one row, left-to-right for level
  Tile t = tilesMap.get("leftEnd").resize(lvlScale);
  t.drawAt(im, "C0", startPt);
  Point2D.Double nextStartPt = t.getPtLoc("C2");
  Point2D.Double topLeft = t.getPtLoc("C1");
```

```

for (int i=0; i < numMids; i++) {
    t = tilesMap.get("mid").resize(lvlScale);
    t.drawAt(im, "C3", topLeft);
    topLeft = t.getPtLoc("C2");
}

t = tilesMap.get("rightEnd").resize(lvlScale);
t.drawAt(im, "C2", topLeft);
Point2D.Double nextEndPt = t.getPtLoc("C1");

if (level == n)    // finished
    return new Pair<Point2D.Double, Point2D.Double>(
        nextStartPt, nextEndPt);
else {
    return drawLevels(level+1, n, 2*numMids+2, im, nextStartPt);
}
} // end of drawLevels()

```

#### 4.3.4. Building the Final Image

`makeQuarter()` generates a triangular tile like the one in Figure 55. The final step is to draw that tile four times around a center point (labeled as “tip” in the figure).

```

Tile quarTile = makeQuarter();

double horizDist = quarTile.length("C1", "C2");
int pWidth = (int)Math.round( horizDist * 1.05);
int pHeight = (int)Math.round( horizDist * 1.05);
    // adds a small border around the rendered image

ImageViewer iview = Pics.view("Square Limit", pWidth, pHeight);
BufferedImage im = iview.getImage();

// color the background
Graphics2D g2d = Pics.createGraphics(im);
g2d.setColor(LIGHT_BLUE);
g2d.fillRect(0, 0, pWidth, pHeight);

// display 4 'quarters' around the center
for (int i = 0; i < 4; i++) {
    Tile t = quarTile.rotate(90*i);
    t.drawAt(im, "C0", pWidth/2, pHeight/2);
    iview.repaint();
    Pics.pause(100);
}

```

#### 6.3.5. Comparison with Henderson's "Functional Geometry" Approach

M.C. Escher's "Square Limit" is the subject of an influential paper by Peter Henderson [7] which utilizes functional programming to present a high-level algebraic description of image composition which hides details such as coordinates and dimensions. The first version of the paper appeared in 1982, but was revised in 2002. The original broke the basic fish into four smaller tiles, as in Figure 57 [9].

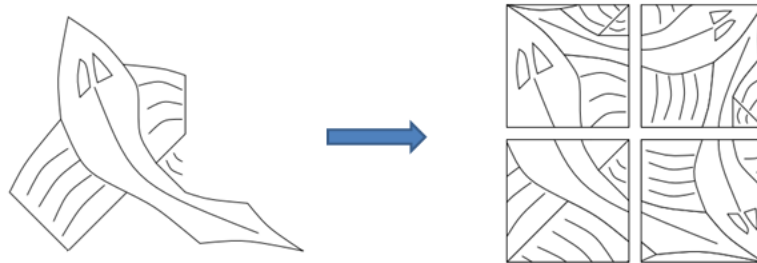


Figure 57. Tiles Used by Henderson in 1982.

Crucially this choice of tiles means that the functional description only uses 90 degree rotations so that the overall image can be viewed as a check board of smaller image frames composed using functional `beside()` and `above()` operations. They build bigger image frames by placing images side-by-side or above each other.

Although very elegant, this approach was not used by Escher himself, and so the 2002 paper took the fish on the left of Figure 57 as the basic tile (which I've also done). This requires the introduction of a `rot45()` function, which is somewhat deceptively named. It does indeed rotate an image counterclockwise by 45 degrees, but also scales the picture by  $\sqrt{2}/2$  and performs the rotation around the tile's top-left corner rather than the center.

The operational semantics is based upon a triplet of vectors that define the position and orientation of an image frame. `rot45()` only works correctly if the frames are square, and Henderson also points out that `rot45()` doesn't follow the same pleasing algebraic rules exhibited by the other operations such as `beside()` and `above()`. A third change is that `above()` and `beside()` must be generalized to allow the specification of subpicture ratios which gives them greater control over the scaling of adjacent frames.

In conclusion, the high-level capabilities deployed in the paper are quite closely tied to the grid-based format of Escher's "Square Limit" picture. Indeed, Henderson argues that a more general algebraic geometry language would need explicit operations for translation, scaling, and rotations, and recent functional graphics libraries have included these features [2, 4].

## 5. Recursion

DrawHorses.java stays with Escher-style tessellation but switches from fish to horses, and employs recursive drawing. This example was suggested by a series of slides written by Piers Chandler, available at <https://slideplayer.com/slide/9037436/>. The output generated by the program is shown in Figure 58.

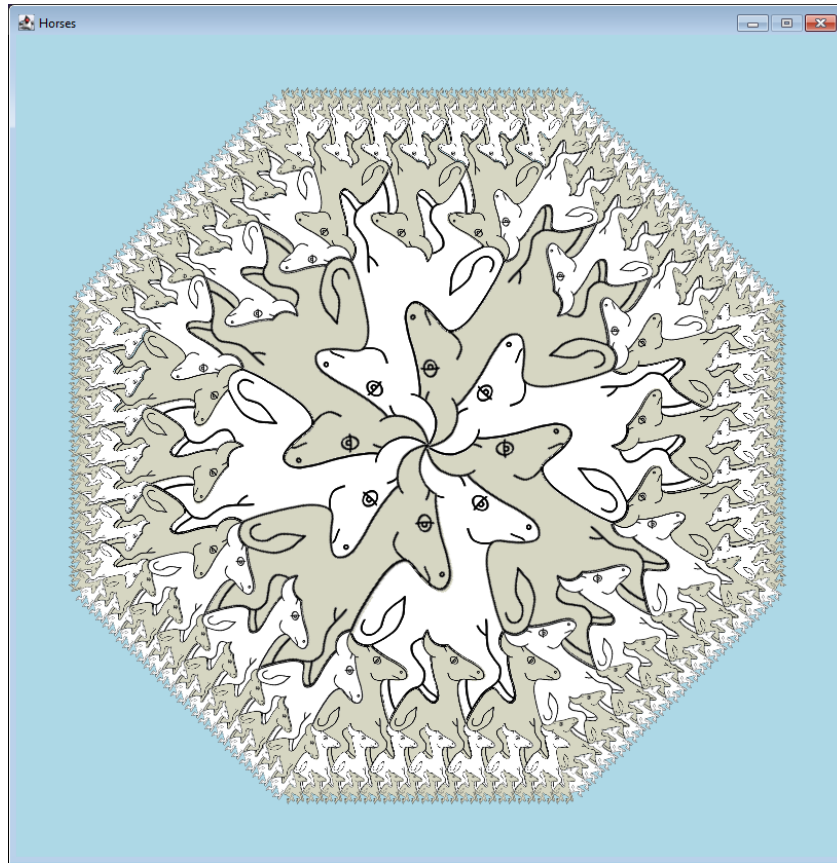


Figure 58. The Output of DrawHorse.java.

The tiling utilizes a single horse image with six points (see Figure 59).

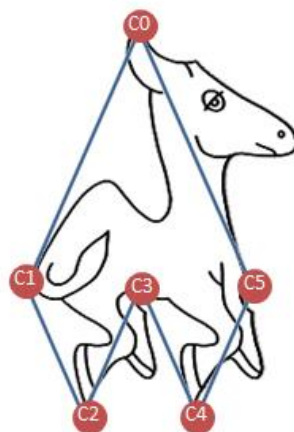


Figure 59. The Horse with Corner Points.  
(The blue lines are not part of the image.)



The tile in Figure 59 is created with the following code, which also generates a light brown horse:

```
Tile horse = new Tile("data/horse.png");
horse.addPt("C0", 165,30);
horse.addPt("C1", 90,210);
horse.addPt("C2", 129,300);
horse.addPt("C3", 165,210);
horse.addPt("C4", 201,300);
horse.addPt("C5", 240,210);

Tile darkHorse = horse.colorChg(Color.WHITE, LIGHT_BROWN);
```

The recursive nature of the tiling is illustrated by Figure 60 – each horse tile is responsible for drawing three smaller horses below it.

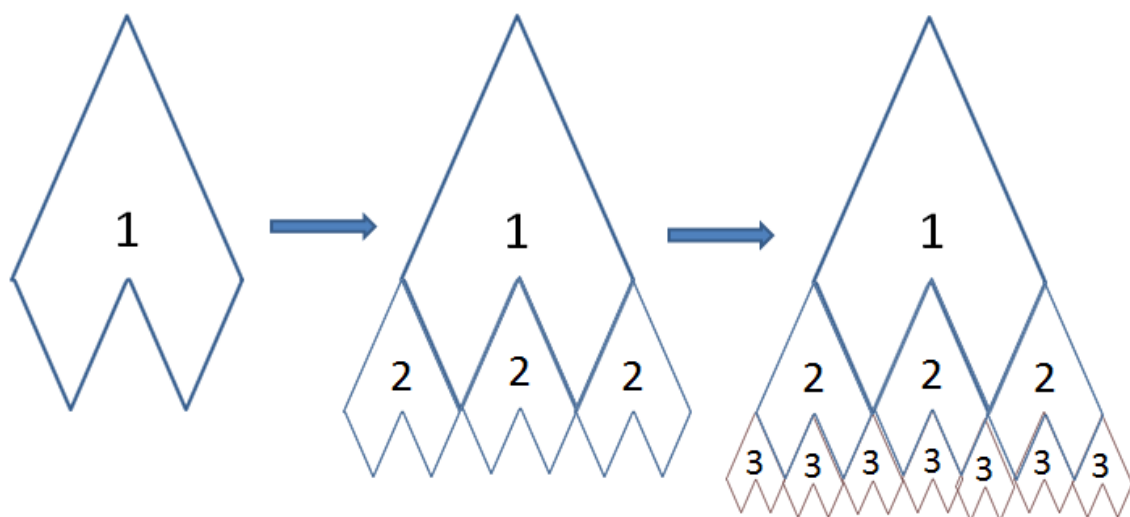


Figure 60. Recursive Horses in Outline (to three levels).

Figure 60 is implemented by drawLevel():

```
private static void drawLevel(int n, int max, Tile h0, Tile h1,
                             Point2D.Double topPt, ImageViewer iview)
{
    if (n > max)
        return;

    Tile t = (n%2 != 0) ? h0 : h1;
    t.drawAt(iview.getImage(), "C0", topPt); // top of horse
```

```

Tile smallH0 = h0.resize(0.5);
Tile smallH1 = h1.resize(0.5);
drawLevel(n+1, max, smallH0, smallH1, t.getPtLoc("C1"), iview);
drawLevel(n+1, max, smallH0, smallH1, t.getPtLoc("C3"), iview);
drawLevel(n+1, max, smallH0, smallH1, t.getPtLoc("C5"), iview);

iview.repaint();
} // end of drawScene()

```

The two tile arguments hold the original horse and the brown version, allowing rendering to swap between them based on the current level.

In main(), a call to drawLevel() creates a single 'tree' of horses, which has to be repeated eight times around the center of the image:

```

ImageViewer iview = Pics.view("Horses", PWIDTH, PHEIGHT);

// color the background
Graphics2D g2d = Pics.createGraphics(iview.getImage());
g2d.setColor(LIGHT_BLUE);
g2d.fillRect(0, 0, PWIDTH, PHEIGHT);

Point2D.Double startPt = new Point2D.Double(PWIDTH/2, PHEIGHT/2);
// center of screen

// draw eight times around the start point
for (int i=0; i < 8; i++) {
    Tile h0 = horse.rotate(45*i);
    Tile h1 = darkHorse.rotate(45*i);
    if (i%2 == 0) // swap horses based on i
        drawLevel(1, 4, h0, h1, startPt, iview); // four levels
    else
        drawLevel(1, 4, h1, h0, startPt, iview);
    Pics.pause(50);
}

```

A close examination of DrawHorses.java reveals that there isn't any need for the points C2 and C4, but I've left them in so that the blue outline used in Figure 59 and 60 is vaguely horse-shaped.

## 6. Extra Data (Part 1): Inner and Outer Points

Although we've seen inner and outer points displayed by Tile.view() and ViewTile.java many times (e.g. see Figures 2 and 5), we haven't used them because all the tilings were created with Tile.drawAt().

```

Tile.drawAt(image, "point name", coordinate)

```

Inner points are employed for collision detection between tiles, which is implemented in terms of whether a tile's inner point has been drawn on the screen at a pixel that was transparent. This relies on the fact that the tiling surface is initially transparent, and also that the non-shape parts of a tile are transparent. We saw these requirements in the composition examples when the image for a new tile was created with a transparent background:

```
BufferedImage im = new BufferedImage(width, height,
    BufferedImage.TYPE_INT_ARGB); // alpha channel
```

Collision detection is utilized in the more complex drawing operations, `Tile.tryDrawAt()` and `Tile.testDrawAt()` which will be explained in section 7, and also by the built-in tiling functions, `tileLocs()` and `tileSpacey()`, in section 9.

Outer points are used to detect if there are other tiles adjacent to a tile. One outer point is usually assigned to each tile side so that the sides occupied by the other tiles can be determined. This functionality is employed in `tileLocs()` and `tileSpacey()` to decide if a tile needs another tile drawn next to it to fill up the screen.

Once a tile has been assigned 'corner' points then jFAT can create inner and outer points automatically, but the results can be rather poor, especially if the shape has concave sides. For that reason, jFAT checks whether any of the generated inner points are outside the shape's borders, and if any of the outer points are inside those borders. It discards those points, and issues warning messages. When such messages appear, it's probably necessary for the programmer to define their own inner and/or outer points.

For a tile based on a shape or tile data, the simplest way to check the inner and outer points is to load it in `ViewTile`, as in Figure 61.

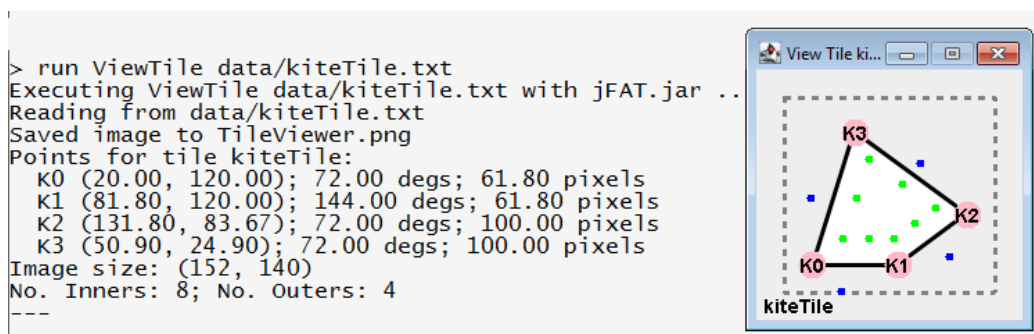


Figure 61. Viewing the Kite Tile.

There are no warnings in Figure 61 since the kite is a convex shape. However, the dart in Figure 62 does produce a few.

```

> run ViewTile data/dartTile.txt
Executing ViewTile data/dartTile.txt with jFAT.jar ...
Reading from data/dartTile.txt
WARNING: deleted 5 poorly positioned inner points
No. of remaining inners: 3

WARNING: discarded 2 poorly positioned outer points
No. of remaining outers: 2

Saved image to TileViewer.png
Points for tile dartTile:
D0 (20.00, 100.00); 72.00 degs; 100.00 pixels
D1 (120.00, 100.00); 36.00 degs; 61.80 pixels
D2 (70.00, 63.67); 216.00 degs; 61.80 pixels
D3 (50.91, 4.90); 36.00 degs; 100.00 pixels
Image size: (140, 120)
No. Inners: 3; No. Outers: 2
---
```

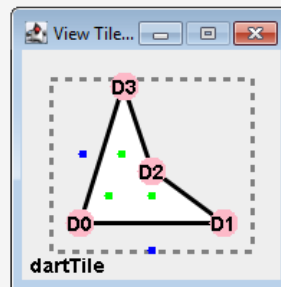


Figure 62. Viewing the Dart Tile.

A tile created from an image (PNG or SVG) will require the user to add their own ‘corner’ points by calling `Tile.addPt()`, as in the case of the horse tile in section 5 (see Figure 59):

```

Tile horse = new Tile("data/horse.png");
horse.addPt("C0", 165,30);
horse.addPt("C1", 90,210);
horse.addPt("C2", 129,300);
horse.addPt("C3", 165,210);
horse.addPt("C4", 201,300);
horse.addPt("C5", 240,210);
horse.view();
horse.reportPoints()

```

The output is shown in Figure 63.

```

> run DrawHorses
Executing DrawHorses with jFAT.jar ...
WARNING: deleted 3 poorly positioned inner points
No. of remaining inners: 9

WARNING: discarded 1 poorly positioned outer points
No. of remaining outers: 5

Saved image to TileViewer.png
Points for tile horse:
C0 (165.00, 30.00); 45.24 degs; 195.00 pixels
C1 (90.00, 210.00); 133.95 degs; 98.09 pixels
C2 (129.00, 300.00); 45.23 degs; 96.93 pixels
C3 (165.00, 210.00); 316.40 degs; 96.93 pixels
C4 (201.00, 300.00); 45.23 degs; 98.09 pixels
C5 (240.00, 210.00); 133.95 degs; 195.00 pixels
Image size: (330, 330)
No. Inners: 9; No. Outers: 5
---
```

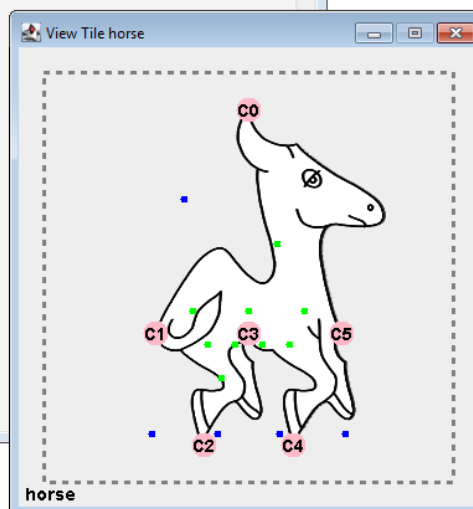


Figure 63. Viewing the Horse Tile.

The irregular shape causes three inner points and an outer point to be dropped. However, since the tiling is only going to utilize `Tile.drawAt()`, it doesn't really matter.

## 6.1. Fixing the Escher Lizard

Section 9.1 will explain how to tile a surface with Escher-style lizards by calling `tileLocs()`. Since `tileLocs()` relies on inner and outer points, its data must be correct. However, the automatically generated points for the lizard image are less than stellar, as shown in Figure 64.

```
// in DrawLizards.java
lizard = new Tile("data/liz.png");
lizard.addPt("C0", 113, 43);    // obtained from MS paint
lizard.addPt("C1", 57, 80);
lizard.addPt("C2", 61, 147);
lizard.addPt("C3", 126, 186);  // defines a hexagon
lizard.addPt("C4", 193, 148);
lizard.addPt("C5", 184, 73);

lizard.view();
lizard.reportPoints();
```

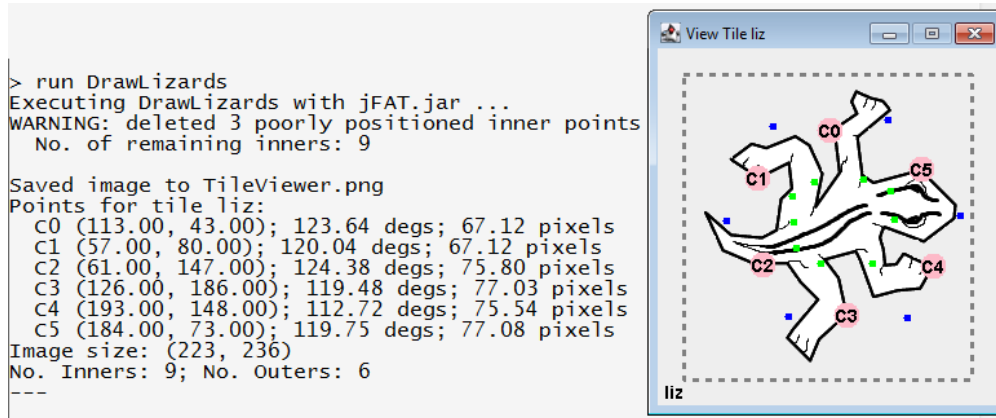


Figure 64. Viewing the Lizard Tile.

Since three inner points are discarded, the coverage is quite poor with no points along the lizard's arms, legs, or tail. It's likely that collision detection will fail to detect when the arm of a lizard crosses the leg of another.

Also, although none of the outer points were rejected, a few more could profitably be added along several of the tile's sides.

## Spotty to the Rescue

User-defined inner and outer points are added by `Tile.addInner()` (or `Tile.addInners()`) and `Tile.addOuter()`. However, how should their coordinates be determined? One (slowish) approach is to click the mouse over a view window, and note down the cursor information that appears in the popup window (e.g. see Figure 9). A faster solution is to call the jFAT Spotty application shown in Figure 65.

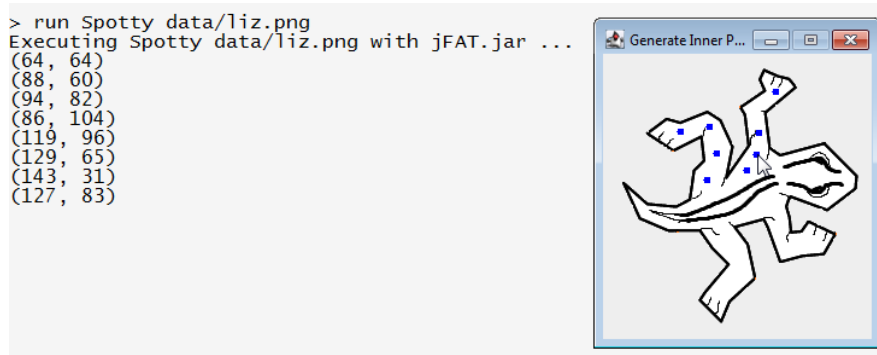


Figure 65. A Spotty Lizard.

When the user clicks the mouse over the image, a blue dot appears and its coordinate is printed. If a spot is poorly positioned, the user can type 'u' to remove it. When Spotty is closed, a useful snippet of code is printed:

```
> run Spotty data/liz.png
Executing Spotty data/liz.png with jFAT.jar ...
(64, 64)
(88, 60)
(94, 82)
(86, 104)
(119, 96)
(129, 65)
(143, 31)
(127, 83)
double[] xs = new double[] {
  64, 88, 94, 86, 119, 129, 143, 127 };
double[] ys = new double[] {
  64, 60, 82, 104, 96, 65, 31, 83 };
Finished.
>
```

These array definitions can be pasted into code as inputs to `Tile.addInners()`, as in `DrawLizards.java`:

```
lizard = new Tile("data/liz.png");
lizard.addPt("C0", 113, 43);
lizard.addPt("C1", 57, 80);
lizard.addPt("C2", 61, 147);
lizard.addPt("C3", 126, 186);
lizard.addPt("C4", 193, 148);
lizard.addPt("C5", 184, 73);

double[] xs = new double[] { // obtained using Spotty.java
  57, 86, 91, 82, 70, 103, 85, 94,
  112, 116, 138, 149, 172, 182, 197, 179,
```

```

    154, 156, 126, 126, 141, 111 };
double[] ys = new double[] {
    66, 58, 88, 112, 132, 155, 137, 197,
    181, 123, 123, 158, 151, 115, 103, 84,
    89, 105, 86, 49, 29, 103 };
lizard.addInners(xs, ys);

// outers are automatically generated

lizard.view();
lizard.reportPoints();

```

Figure 66 shows a view of the revised lizard tile, with the default inner points replaced by the user's data. Note that the default outer points are still being utilized.

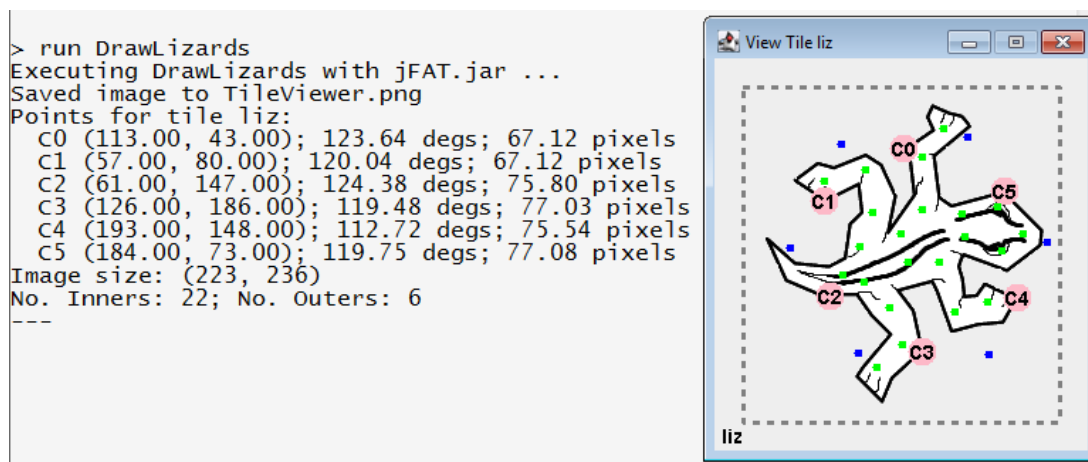


Figure 66. Viewing the Lizard Again.

## 6.2. Supplying Outer Points

Isocles.java employs `tileSpacey()` to tile the screen with triangles, producing something like Figure 67.

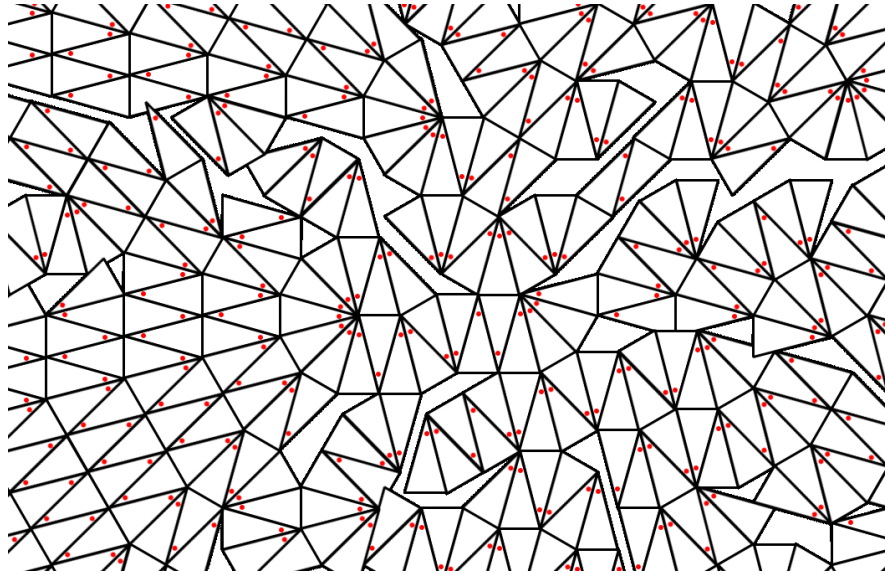


Figure 67. Tiling Using Isoceles.java

The triangle tile utilizes user-defined outer points:

```

tri = new Tile("data/isoTile.txt");
Graphics2D g2d = tri.getGraphics();
g2d.setColor(Color.RED);
g2d.fillOval(48,40, 6, 6); // coord obtained by using ViewTile

double[] xs = new double[] {
    50, 49, 45, 53, 40, 49, 59, 38,
    51, 59, 35, 50, 61 };
double[] ys = new double[] {
    36, 47, 64, 63, 80, 80, 80, 93,
    93, 93, 105, 103, 103 };
tri.addInners(xs, ys);

tri.addOuter("I0", "I1", 29, 63);
tri.addOuter("I1", "I2", 48, 122);
tri.addOuter("I2", "I0", 72, 67);

tri.view();
tri.reportPoints();

```

A call to `Tile.addOuter()` is passed two point labels which define the side associated with the outer point. Usually a single outer point located a short distance from the side's midpoint is sufficient; that kind of positioning can be seen in Figure 68.



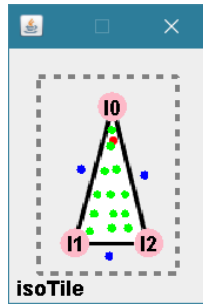


Figure 68. The Isosceles Triangle Displayed in Tile.view().

The Spotty application can also be used to generate outer points, although it doesn't output `Tile.addOuter()` code snippets.

## 7. Testing a Tile for Overlaps

Inner points are used by `Tile.tryDrawAt()` and `Tile.testDrawAt()` to detect collision detection between tiles. Both functions work in a similar manner – a copy is made of the surface image, and `Tile.drawAt()` is used to draw the tile onto that copy. Once printed, the location of the tile's inner points can be retrieved and used for collision detection tests back on the original, unmodified surface image. If those pixel locations are transparent then that space is not being used by another tile.

The drawback is that only the inner point coordinates are checked, not the entire area taken up by the tile, and so there's a chance that an overlap will be missed. For example, if you look closely at Figure 67, you can see that such overlaps do occasionally occur when placing the triangle tiles.

If no overlaps are detected then `tryDrawAt()` returns the modified surface image, and the programmer can use it to update `ImageViewer`. Alternatively, `testDrawAt()` does that update for you, and only returns a boolean to indicate if any collisions were detected.

`tryDrawAt()` and `testDrawAt()` do not use outer points.

In the following example, `Peanuts.java` tiles the screen with a peanut shape (two hexagons sharing a side; see Figure 69). The tile has a blue dot near one corner to help the user discern how a peanut is oriented in the final tiling (see Figure 74).

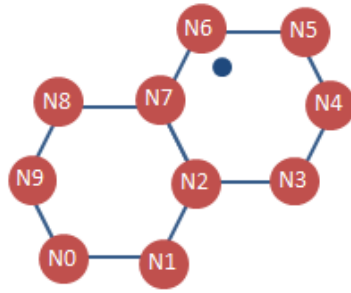


Figure 69. The Peanut.

The tiling utilizes six peanut orientations shown in Figure 70.

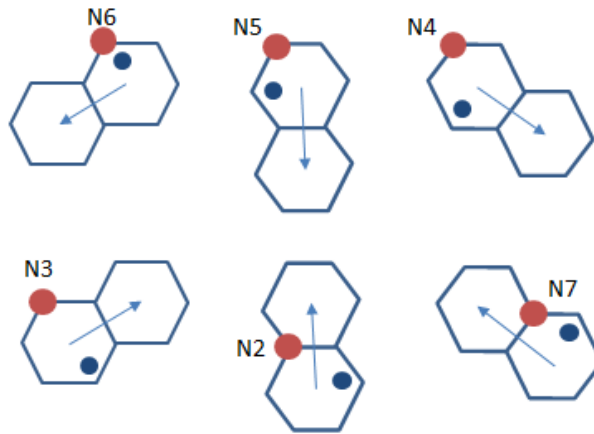


Figure 70. Six Peanuts.

Figure 70 includes an arrow in each peanut to indicate its orientation relative to its blue dot. The arrows and the points marked with red dots are not drawn at tiling time.

The tiling is performed row-by-row, with the surface image treated as a hexagonal grid to be filled with random nuts. Drawing starts on a row at a grid location (marked in gray in Figure 71), and moves to the right in steps of two hexagons. When one of the peanuts in Figure 70 is drawn, the point labeled with a red dot is used to position the peanut at the top-left corner of the hexagon.

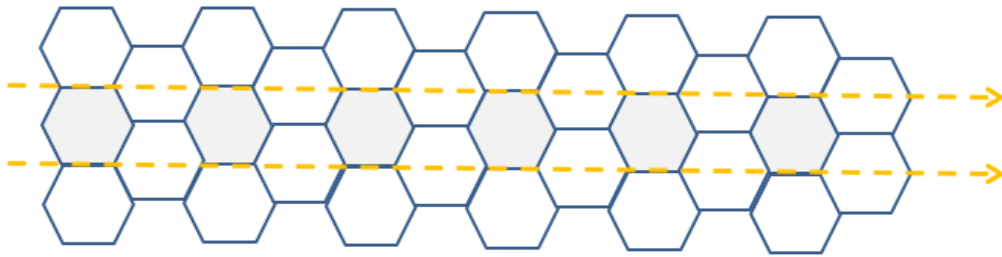


Figure 71. The Stages in Filling a Row.

The difficulty with this approach is that a peanut takes up two hexagons, and the random choice means that a collision may occur when a peanut is drawn. This disaster is depicted in Figure 72 when the intended position of the fourth peanut will overlap with the previous one.

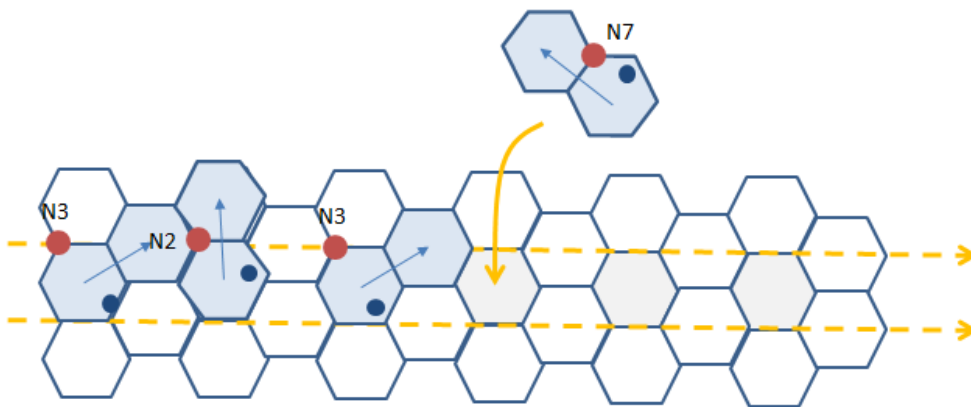


Figure 72. When Peanuts Collide!

Peanuts.java solves this problem by calling `Tile.TestDrawAt()`. If it returns true then there 's no problem. If it returns false then a collision means that the screen image wasn't updated. In that case, a different peanut is selected and the drawing retried.

The six peanuts are derived from a single tile which is rotated in steps of 60 degrees to generate the others. Those tiles, and their labels (the red dots in Figure 70), are stored in arrays:

```
// A peanut with a blue dot at one end
Tile p = new Tile("data/nutTile.txt");
Graphics2D g2d = p.getGraphics();
g2d.setColor(Color.BLUE);
g2d.fillOval(110, 36, 10, 10);

// create tiles for all the possible rotations of the peanut
```

```

Tile[] peanuts = new Tile[NUM_NUTS];
for (int i=0; i < NUM_NUTS; i++) {
    peanuts[i] = p.rotate(60*i);
    peanuts[i].view();
}

String[] drawPts = new String[] { // the labels in Figure 70
    "N6", "N5", "N4", "N3", "N2", "N7" };

```

The row-by-row algorithm utilizes precalculated offsets (see Figure 73) to move over the drawing surface as if it contained hexagon areas. The offsets are based on the horizontal and vertical distances between the corners of a hexagon in a peanut.

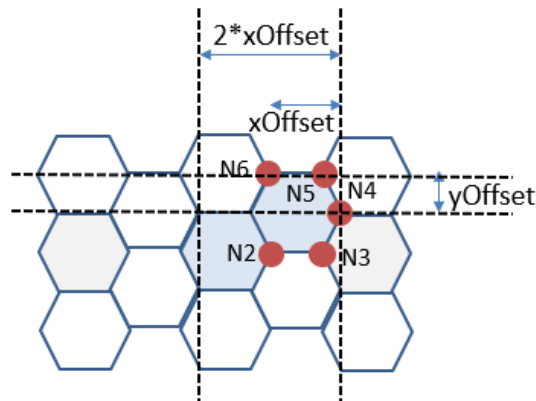


Figure 73. Offsets Between Hexagons.

The code uses nested loops as usual, but the inner loop does something new with `Tile.TestDrawAt()`.

```

// calculate offsets between an hexagon
Point2D.Double pt6 = peanuts[0].getCartesian("N6");
Point2D.Double pt2 = peanuts[0].getCartesian("N2");
Point2D.Double pt4 = peanuts[0].getCartesian("N4");
double xOffset = Math.abs(pt4.x - pt6.x);
double yOffset = Math.abs(pt2.y - pt6.y)/2.0;

ImageViewer iview = Pics.view("Peanuts", Pics.altScreen());
BufferedImage scrImage = iview.getImage();
int pWidth = scrImage.getWidth();
int pHeight = scrImage.getHeight();

int rowNum = 0;
double y = 0;
while (y < pHeight) {

```

```

double x = (rowNum%2 == 0)? 10 : 10+xOffset;
    // calculate starting position for the new row
while (x < pWidth) {
    if (Pics.isTransparent(scrImage, x, y+yOffset)) {
        // check if this drawing spot is empty
        boolean isDrawn =
            randPeanut(peanuts, drawPts, x, y, scrImage);
        if (isDrawn)
            iview.repaint();
    }
    x += 2*xOffset; // jump two hexagons to the right
}
y += yOffset; // move down to next row
rowNum++;
}

```

`Pics.isTransparent()` checks if the next drawing spot along the row is empty. If it is then `randPeanut()` randomly selects a peanut to be drawn. This function may return false if none of the peanuts can be rendered.

`randPeanut()` generates a random sequence of indices made up of the integers from 0 to 5, and tries to draw each corresponding peanut until it succeeds, or the sequence is exhausted

```

private static boolean randPeanut(
    Tile[] peanuts, String[] drawPts,
    double x, double y, BufferedImage scrImage)
{
    int[] randIdxs = shuffleSeq(NUM_NUTS); // e.g. {3,5,1,0,2,4}
    for (int i=0; i < NUM_NUTS; i++) {
        int idx = randIdxs[i];
        if (peanuts[idx].testDrawAt(scrImage, drawPts[idx], x, y))
            return true;
    }
    return false;
}

```

A typical tiling generated by `Peanuts.java` is shown in Figure 74.

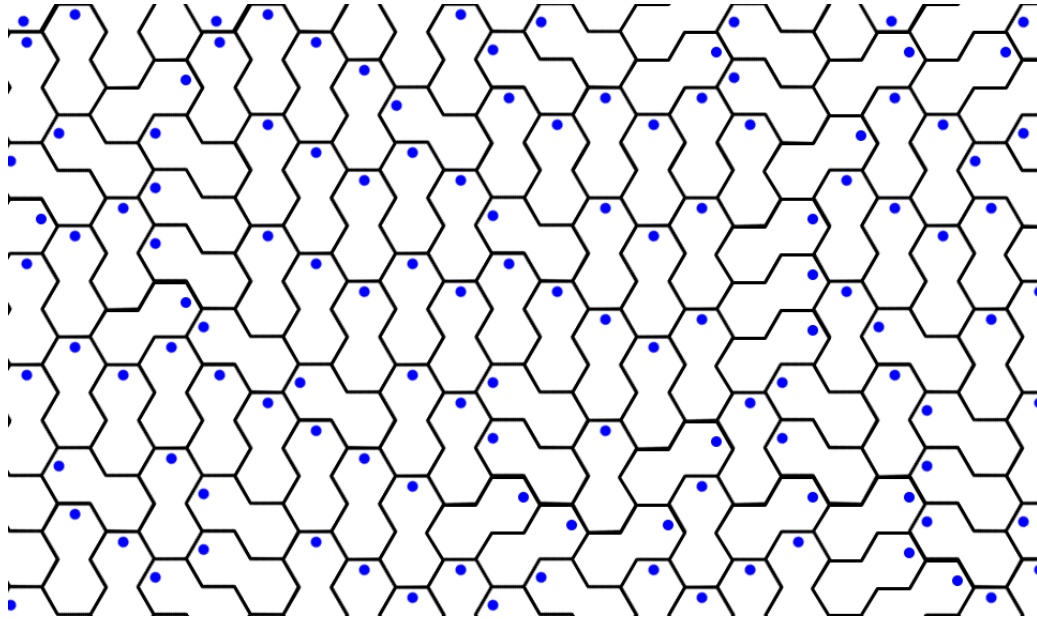


Figure 74. Tiled Peanuts.

## 8. Extra Data (Part 2): Point Pairs and the ‘genTile’ Function.

`tileLocs()` and `tileSpacey()` require two more pieces of information in addition to inner and outer points: point pairs, and a Java functional interface for creating a tile.

A point pair is a shorthand way of specifying what *sides* of different tiles can be paired together. Sides pairing can be simplified to a point pair by assuming that a tile’s coordinates are specified in counterclockwise order.

The functional interface is part of the `java.util.function` package which allows a function to be passed to a method as an argument. This capability is employed in `tileLocs()` and `tileSpacey()` to pass them a `genTile()` function that creates a new tile.

Point pairs and `genTile()` will be explained through several examples in the following subsections.

### 8.1 Point Pairs

jFAT supports two common cases for defining point pairs: when every tile side can be paired with every other one (e.g. when the tile is a regular polygon, such as a square), and pairing based on equal side lengths (e.g. when the tile is a rectangle).

#### All Sides can be Paired

Carpet.java illustrates a situation when every tile side can be placed against another one – when the underlying shape is an equilateral triangle (see Figures 4,5, and Figure 75).

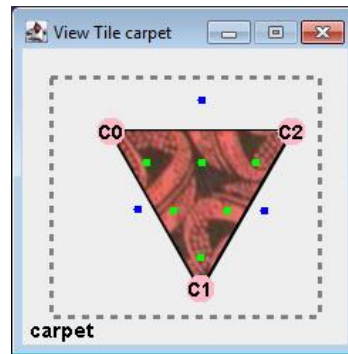


Figure 75. The Carpet Tile.

The relevant code snippet:

```
double[] xs = {42, 105, 169};
double[] ys = {38, 149, 38};
Shape s = ShapeOps.points(xs, ys);
carpetTile = new Tile(s, "carpet");
carpetTile.view();
```

```
PointPairs ptPairs = new PointPairs(carpetTile);
ptPairs.print();
```

A reference to the tile is passed to the PointPairs constructor which returns a PointPairs object that will be later utilized by tileLocs() or tileSpacey() (see the next section). The call to PointPairs.print() produces:

```
Point Pairs:
C0->C0; C0->C1; C0->C2; C1->C0; C1->C1; C1->C2; C2->C0; C2->C1; C2->C2;
```

The nine pairs means that any two sides can be composed.

A variant of this approach appears in DrawShapes.java which utilizes several regular polygons of the same side lengths. The square and equilateral triangle tiles are:

```
Tile sq = new Tile("data/sqTile.txt");
Tile equ = new Tile("data/equTile.txt");
```

Every side of the square *and* the triangle can be paired, which is specified by passing both tile references to the PointPairs constructor:

```
PointPairs ptPairs = new PointPairs( new Tile[] {sq, equ } );
ptPairs.print();
```

This call to `PointPairs.print()` generates a much longer list of pairings:

```
Pair pts: [S0, S1, S2, S3, E0, E1, E2]
Point Pairs:
S0->S0; S0->S1; S0->S2; S0->S3; S0->E0; S0->E1; S0->E2; S1->S0; S1->S1;
S1->S2; S1->S3; S1->E0; S1->E1; S1->E2; S2->S0; S2->S1; S2->S2; S2->S3;
S2->E0; S2->E1; S2->E2; S3->S0; S3->S1; S3->S2; S3->S3; S3->E0; S3->E1;
S3->E2; E0->S0; E0->S1; E0->S2; E0->S3; E0->E0; E0->E1; E0->E2; E1->S0;
E1->S1; E1->S2; E1->S3; E1->E0; E1->E1; E1->E2; E2->S0; E2->S1; E2->S2;
E2->S3; E2->E0; E2->E1; E2->E2;
```

S0 through S3 are the points for the square, and E0 to E2 are for the triangle.

### Same Side Lengths can be Paired

Another common matching is of all sides of the same length. For example, `Traps.java` employs the trapezoid in Figure 76.

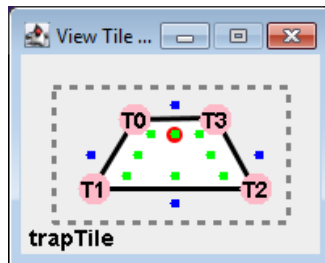


Figure 76. The Trapezoid Tile.

The sides T0-T1, T2-T3, and T3-T0 have the same length, which is handled by calling the `PointPairs.matchLen()` static method:

```
trap = new Tile("data/trapTile.txt");
:
PointPairs ptPairs = PointPairs.matchLens(trap);
ptPairs.print();
```

The output from `PointPairs.print()` is:



Point Pairs based on side lengths:

Added: T0 -> [T0, T1, T3]

Added: T1 -> [T2]

Added: T2 -> [T0, T1, T3]

Added: T3 -> [T0, T1, T3]

Point Pairs:

T0->T0; T0->T1; T0->T3; T1->T2; T2->T0; T2->T1;

T2->T3; T3->T0; T3->T1; T3->T3;

The 'Added' lines list pairings by point, so the first line states that T0 can be paired with T0, T1, and T3. It can be hard to visualize what this means in terms of sides, and it sometimes helps to check the pairings by drawing the shape combinations, as illustrated in Figure 77.

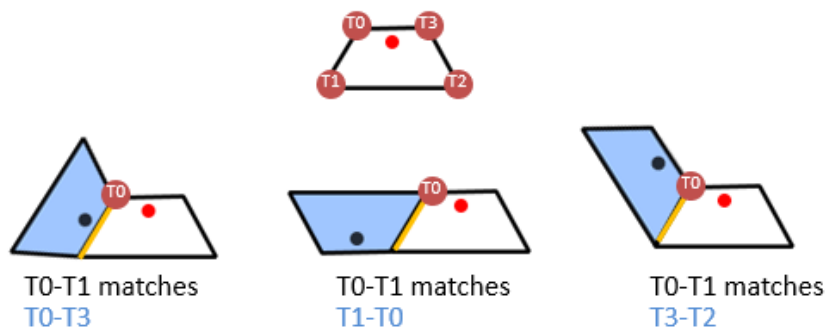


Figure 77. The pairings T0 -> [T0, T1, T3].

T0 is the first point of a side defined in a counterclockwise direction, and so represents T0-T1. A sanity check is that the integer argument of the second point should be one greater (i.e. increasing from 0 to 1), modulo the number of points in the shape.

The matching sides are in clockwise order, and so are T0-T3, T1-T0, and T3-T2 in this case. The sanity check here dictates that the side integer should *decrease* by 1 modulo the number of points.

Isoceles.java is another example of side lengths matching; the tile is shown in Figure 78.

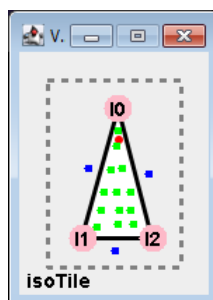


Figure 78. The Isosceles Tile.

The relevant code:

```
tri = new Tile("data/isoTile.txt");
:
PointPairs ptPairs = PointPairs.matchLens(tri);
ptPairs.print();
```

The output from PointPairs.print() is:

```
Point Pairs based on side lengths:
  Added: I0 -> [I0, I1]
  Added: I1 -> [I2]
  Added: I2 -> [I0, I1]
Point Pairs:
  I0->I0; I0->I1; I1->I2; I2->I0; I2->I1;
```

### Defining Point Pairs by Hand

More complex shapes generally require the programmer to define their own point pairs.

The Penrose dart and kite become aperiodic by restricting which of their sides can be brought together. This is usually indicated by adding dots to the compatible points (as in Figure 79) or by drawing differently colored curves through their sides (as in Figure 39).

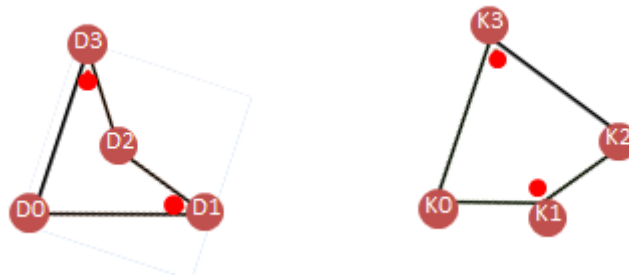


Figure 79. The Penrose Dart and Kite.

A point with a red dot can be paired with other dotted point, and undecorated points can also go together.

The simplest way to set up these pairings is by multiple calls to PointPairs.add(), as in DrawPenrose.java:

```

PointPairs ptPairs = new PointPairs();
ptPairs.add("D0", "D0"); // dart pts
ptPairs.add("D0", "K0");
ptPairs.add("D1", "K1");
ptPairs.add("D2", "K2");
ptPairs.add("D3", "K3");
ptPairs.add("D3", "D1");

ptPairs.add("K0", "D2"); // kite pts
ptPairs.add("K0", "K2");
ptPairs.add("K1", "D3");
ptPairs.add("K1", "K1");
ptPairs.add("K2", "D0");
ptPairs.add("K2", "K0");
ptPairs.add("K3", "D1");
ptPairs.add("K3", "K3");

```

The first two calls to `PointPairs.add()` are shown visually in Figure 80.

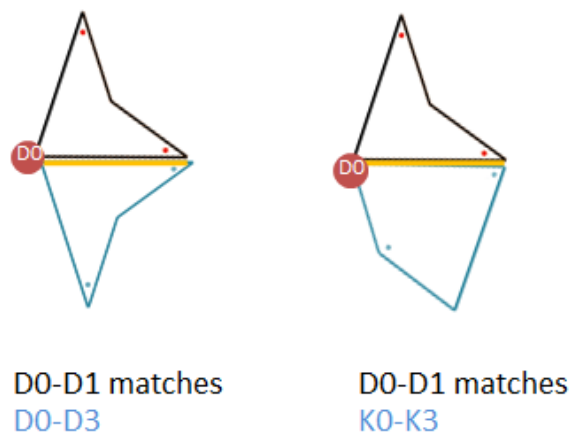


Figure 80. The pairings  $D0 \rightarrow D0$  and  $D0 \rightarrow K0$ .

The same checks can be utilized as in the trapezoid example. The side under consideration ( $D0-D1$ ) is in increasing counterclockwise order, and its matching sides ( $D0-D3$ ,  $K0-K3$ ) are in decreasing clockwise order (modulo 4).

As before, it really helps to check these pairs by drawing the darts and kites.

`DrawLizard.java` utilizes what looks like a very complex shape (see Figures 9 to 11), but the corner points for the lizard mark out a regular hexagon (see Figure 81).

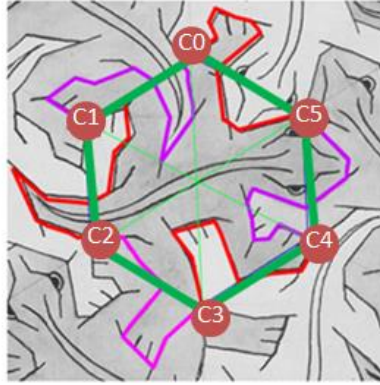


Figure 81. Lizard in a Hexagon.

The lizard's point pairs only need to define how the six sides of the hexagon can be matched with the sides of other tiles. This is fairly straightforward because the complexity of the image means that each side only has a single match. The resulting code:

```
PointPairs ptPairs = new PointPairs();
ptPairs.add("C0", "C2");
ptPairs.add("C1", "C1");
ptPairs.add("C2", "C4");
ptPairs.add("C3", "C3");
ptPairs.add("C4", "C0");
ptPairs.add("C5", "C5");
```

For instance, C0->C2 pairs the side C0-C1 with C2-C1 which corresponds to matching up the tail of the lizard between two tiles.

DrawMaze.java employs six images as tiles (Figure 82) and uses them to build a maze.

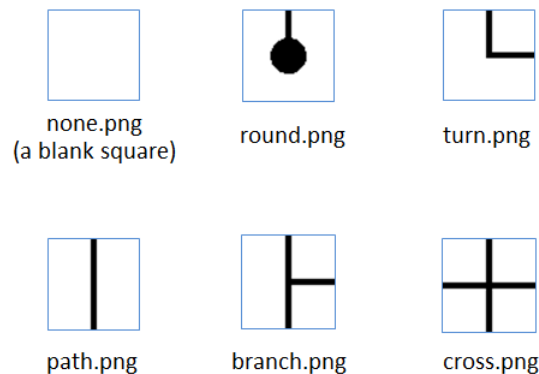


Figure 82. The Maze Tiles.

The following code creates the tiles:

```
Point2D.Double[] pts = { // four points; defined ccw
    new Point2D.Double(0,0), new Point2D.Double(0,60),
    new Point2D.Double(60,60), new Point2D.Double(60,0) };

// note: each tile uses a unique label
noneTile = new Tile("data/none.png");
noneTile.addPts("N", pts);

roundTile = new Tile("data/round.png");
roundTile.addPts("R", pts);

turnTile = new Tile("data/turn.png");
turnTile.addPts("T", pts);

pathTile = new Tile("data/path.png");
pathTile.addPts("P", pts);

branchTile = new Tile("data/branch.png");
branchTile.addPts("B", pts);

crossTile = new Tile("data/cross.png");
crossTile.addPts("C", pts);
```

A typical maze is shown in Figure 83.

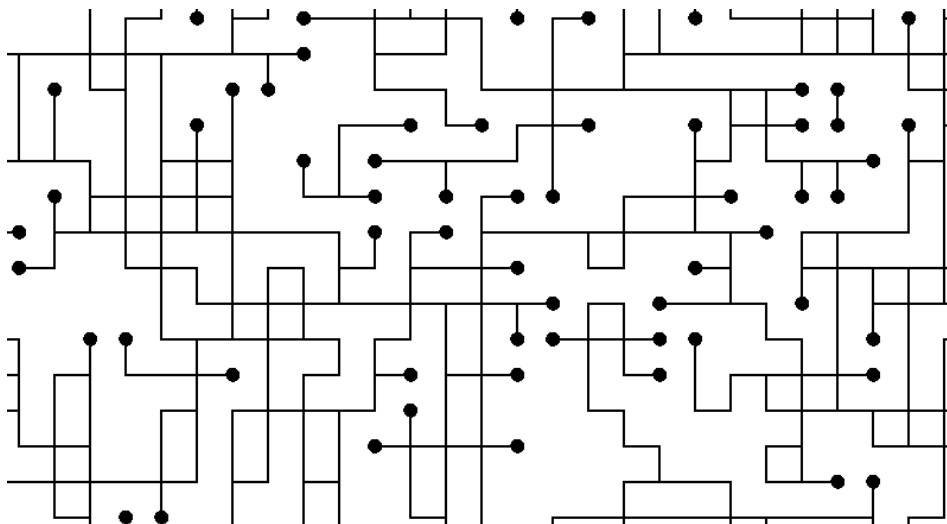


Figure 83. A Mazing Output.

Each tile is a square so the tiling complexity arises from specifying which sides can be paired. There are only two cases: when a side includes a line and when it's blank; any 'line' side can be paired with any other, and any blank side can be paired with any blank. This is implemented by pairing four arrays of point names:

```

/* Each side either has a maze line or is blank.
   "zero" means that no line enters/leaves a side;
   "one" means there is a line entering/leaving a side
*/
// ccw 'from' pts
String[] zeroPts = { // no line leaving the side
    "N0", "N1", "N2", "N3", "R0", "R1", "R2",
    "T0", "T1", "P0", "P2", "B0" };

String[] onePts = { // line leaving the side
    "R3", "T2", "T3", "P1", "P3",
    "B1", "B2", "B3", "C0", "C1", "C2", "C3" };

// clockwise to pts
String[] toZeros = { // no line entering the side
    "N0", "N1", "N2", "N3",
    "R1", "R2", "R3", "T1", "T2",
    "P1", "P3", "B1" };

String[] toOnes = { // line entering the side
    "R0", "T0", "T3", "P0", "P2",
    "B0", "B2", "B3",
    "C0", "C1", "C2", "C3" };

// create "no line" and line pairs
PointPairs ptPairs = new PointPairs();
ptPairs.add(zeroPts, toZeros); // no line
ptPairs.add(onePts, toOnes); // line

```

Note that the zeroPts[] and toZeros[] and the onePts[] and toOnes[] are similar, but **not** the same.

## 8.2. The 'genTile() Function Interface

genTile() defines how tileLocs() and tileSpacey() can create a new tile when they need one. It's coded using java.util.function so that it can be passed as an argument to those tiling functions. The function must have one input argument, which tileLocs() and tileSpacey() assume to be a point label, and returns a Tile result.

In the cases when the tiling only involves one tile type, there's no need to use the label argument, and the code becomes very short. This allows us to define the function as a lambda expression. For example, in Pentagons.java:

```
Function<String, Tile> genTile = (String nm)->{ return penta.clone(); };
```

The `genTile` variable is passed to `tileLocs()` or `tileSpacey()` which uses it to generate a new pentagon tile using `Tile.clone()`, and the label isn't used. One drawback of this approach is that the `penta` tile must be declared globally so that `genTile()` will be able to reference it at runtime. For more examples like this one, see `Carpet.java`, `Isoceles.java`, and `Triangles.java`.

A slightly more complicated `genTile()` usage can be found in `Traps.java`, when the trapezoid tile is cloned and given a random color:

```
Function<String, Tile> genTile =
    (String nm)->{ return trap.colorChg(
        colors[rand.nextInt(colors.length)]); };
```

Since `Traps.java` also only utilizes a single tile type, there's still no need to use the string input. However, the `trap` tile, the `rand` random number object, and the `colors[]` array must all be global so that `genTile()` can reference them during its execution.

`DrawMaze.java` employs six tiles (see Figure 82), so its `genTile()` does use the string argument. `tileLocs()` or `tileSpacey()` will pass it a point name (e.g. "N1", "P2", "C0"), and `genTile()` uses this argument to decide which type of tile to create. Of course, for this approach to work, the programmer should ensure that the different tiles use different point labels. The code extracts the first character of the string and uses it to switch to the correct call to `Tile.clone()`:

```
private static Tile genTile(String ptName)
// point names begin with 'N', 'R', 'T', 'P', 'B', or 'C'
{
    char ch = ptName.charAt(0);
    switch(ch) {
        case 'N': return noneTile.clone();
        case 'R': return roundTile.clone();
        case 'T': return turnTile.clone();
        case 'P': return pathTile.clone();
        case 'B': return branchTile.clone();
        case 'C': return crossTile.clone();

        default: System.out.println("Unknown point name: " + ptName);
    }
    return null;
} // end of genTile()
```

Since this function is a few lines long, I wrote it as a real function rather than as a lambda expression, which means that `tileLocs()` or `tileSpacey()` must refer to it as

DrawMaze::genTile. More examples of this kind can be found in DrawPenrose.java and DrawShapes.java

## 9. Built-in Tiling

After several sections of buildup, we can finally describe tileLocs() and tileSpacey().

Both functions work upon a list of 'placed' tiles (i.e. those tiles already drawn on the image), looping through them either until the list is empty or a maximum number of tiles have been drawn. A tile is removed from the list and each one of its points is examined in order to draw tiles adjacent to its sides.

tileLocs() utilizes a backtracking algorithm to try to find a set of tiles that completely surround the current point, and so ensure there are no spaces left in the tiling. However, if the search fails, tileLocs() does **not** backtrack to earlier points or tiles. This has the advantage of increasing its tiling speed, but at the expense of poor tiling coverage in some cases (as we'll see below).

tileSpacey() also considers every point of a tile but only tries to draw a single new tile adjacent to the point's side. If there's no suitable choice then the side is left unused, and the algorithm moves on to the next point in the tile. This makes tileSpacey() even faster than tileLocs(), but the function's main reason for existing is to handle tilings that deliberately include spaces (we'll see some examples below).

I've decided not to describe tileLocs() and tileSpacey() in any greater detail than this, but both functions can be found in Search.java in the jFAT JAR file, and are generously documented.

### 9.1. Using tileLocs()

The jFAT download contains many examples that use tileLocs(), including: Arrows.java, Carpet.java, DrawArrows.java, DrawCurves.java, DrawLizards.java, DrawMaze.java, DrawPenrose.java, DrawQuarters.java, DrawShapes.java, DrawTris.java, ImPentagons.java, Isoceles.java, Pentagons.java, Traps.java, and Triangles.java.

Traps.java was first described in section 8 (see Figure 76), where the focus was on its point pairs. The code below is its entire main() function, but only the last two lines are new:

```
private static Tile trap; // global so can be used in genTile()

public static void main(String args[])
{
    trap = new Tile("data/trapTile.txt");
    Graphics2D g2d = trap.getGraphics();
    // red circle
```



```

g2d.setColor(Color.RED);
g2d.fillOval(70, 25, 10, 10);
trap.view();
trap.reportPoints();

PointPairs ptPairs = PointPairs.matchLens(trap);

Function<String, Tile> genTile =
    (String nm)->{ return trap.colorChg(
        colors[rand.nextInt(colors.length)]); };

// draw tiles on the screen
ImageViewer iview = Pics.view("Draw Trapezoids", Pics.altScreen());
Search.tileLocs(trap, ptPairs, iview, genTile);
Pics.save(iview.getImage(), "SavedPics/trapezoids.png");
} // end of main()

```

tileLocs() is passed a starting tile (trap), the point pairs, a reference to the ImageViewer, and the genTile() function.

Part of the output is shown in Figure 84.

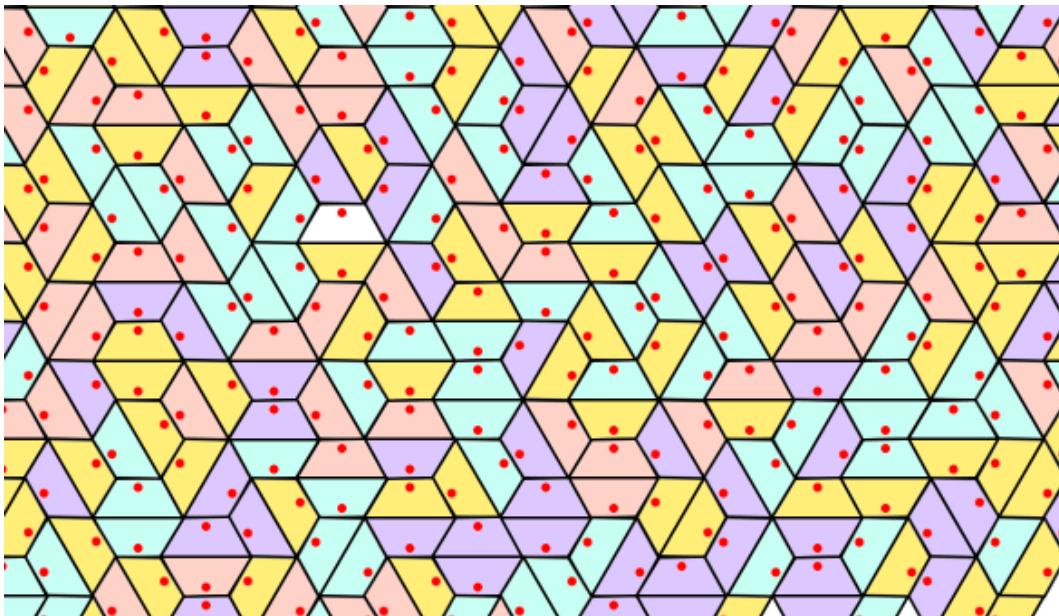


Figure 84. Trapezoids Output.

When tileLocs() (and tileSpacey()) can choose between several point pairs for matching a tile, they make a random selection. In the case of the trapezoid, this means that any of the T0 pairings in Figure 77 might be employed. The random coloring carried out by genTile() also causes additional variations between runs of the program.

When `tileLocs()` finishes, it prints a percentage for the amount of uncovered space in the image, which for this example is just 0.12%.

`DrawLizards.java` was first described back in section 1.3 (see Figures 9 to 11). The code below is its entire `main()`:

```
private static Tile lizard;

public static void main(String args[])
{
    lizard = new Tile("data/liz.png");
    lizard.addPt("C0", 113, 43);
    lizard.addPt("C1", 57, 80);
    lizard.addPt("C2", 61, 147);
    lizard.addPt("C3", 126, 186);
    lizard.addPt("C4", 193, 148);
    lizard.addPt("C5", 184, 73);

    double[] xs = new double[] {
        57, 86, 91, 82, 70, 103, 85, 94,
        112, 116, 138, 149, 172, 182, 197, 179,
        154, 156, 126, 126, 141, 111 };
    double[] ys = new double[] {
        66, 58, 88, 112, 132, 155, 137, 197,
        181, 123, 123, 158, 151, 115, 103, 84,
        89, 105, 86, 49, 29, 103 };
    lizard.addInners(xs, ys);

    // outers are automatically generated

    PointPairs ptPairs = new PointPairs();
    ptPairs.add("C0", "C2");
    ptPairs.add("C1", "C1");
    ptPairs.add("C2", "C4");
    ptPairs.add("C3", "C3");
    ptPairs.add("C4", "C0");
    ptPairs.add("C5", "C5");
    ptPairs.print();

    Function<String, Tile> genTile =
        (String nm)->{return lizard.colorChg(Color.WHITE,
            colors[rand.nextInt(colors.length)]); };

    ImageViewer iview = Pics.view("Lizards", Pics.altScreen());
    Search.tileLocs(lizard, ptPairs, iview, genTile);
    Pics.save(iview.getImage(), "SavedPics/lizards.png");
} // end of main()
```

The call to `tileLocs()` is virtually identical to the one in `Traps.java`, although the starting tile is a lizard of course.

Part of the output is shown in Figure 85, and `tileLocs()` reports an unfilled space percentage of 0.54%, which can be observed by looking closely at the top left of the screenshot.

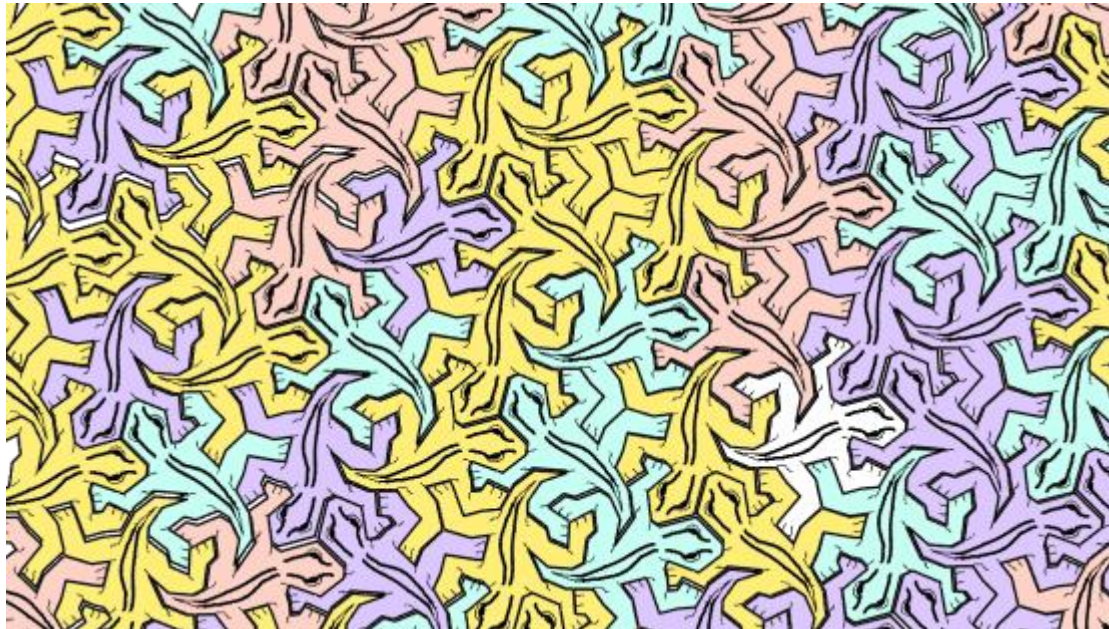


Figure 85. Lizards Output.

One common way of improving tile coverage is to replace the automatically generated inner and outer points for a tile. This process can be seen in action in `DrawLizards.java` if the call to `Tile.addInners()` is commented out, switching the program back to using the default inner points. Figure 86 shows the worsening in the tile coverage.

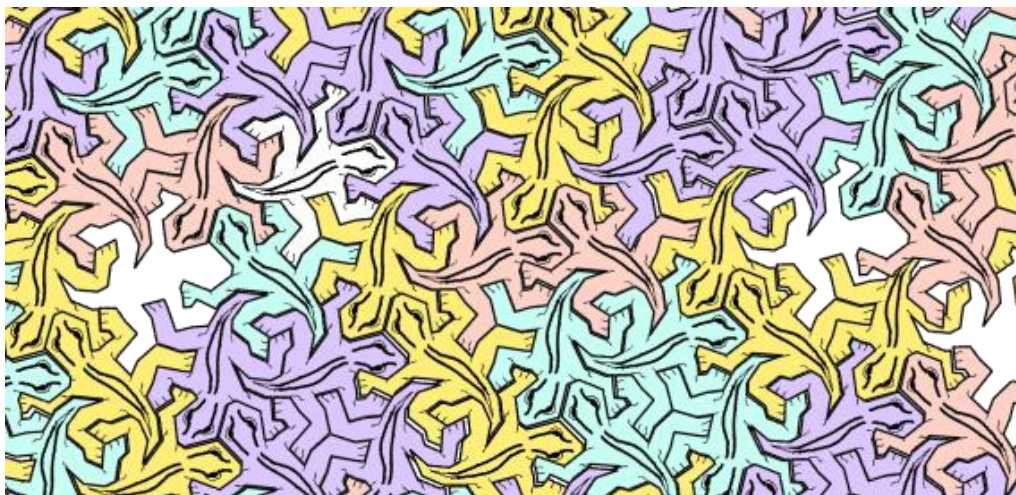


Figure 86. Lizards Output with Default Inner Points.

The reason for this drop off in coverage can be seen by looking at the `Tile.view()` images for the lizard in both cases (e.g. in Figures 66 and 64). In the latter (worse) case, the inner points (drawn as green dots) are very close to the edge of the lizard and don't include its arms, legs, and tail.

Another problem with the lizard image is that it doesn't quite match the hexagon defined by its points (see Figure 81). This is easiest to see by looking at the output from `Tile.reportPoints()`:

```
Points for tile liz:
  C0 (113.00, 43.00); 123.64 degs; 67.12 pixels
  C1 (57.00, 80.00); 120.04 degs; 67.12 pixels
  C2 (61.00, 147.00); 124.38 degs; 75.80 pixels
  C3 (126.00, 186.00); 119.48 degs; 77.03 pixels
  C4 (193.00, 148.00); 112.72 degs; 75.54 pixels
  C5 (184.00, 73.00); 119.75 degs; 77.08 pixels
Image size: (223, 236)
No. Inners: 9; No. Outers: 6
```

Each interior angle should be 120 degrees and the side lengths should be the same. This indicates that the image should be redrawn, and the points more accurately positioned.

A similar problem is behind the poor tiling generated by `Triangles.java`. It uses an image of an equilateral triangle containing an 'f' (`trif.png`), and its `Tile.reportPoints()` output is:

```
Points for tile trif:
  A (51.00, 6.00); 60.33 degs; 92.78 pixels
  B (4.00, 86.00); 59.57 degs; 93.00 pixels
  C (97.00, 86.00); 60.10 degs; 92.28 pixels
Image size: (100, 100)
No. Inners: 9; No. Outers: 3
```

The interior angles should all be 60 degrees, and the side lengths should be the same.

As a third disappointing example, consider the output from `Isoceles.java` in Figure 87.

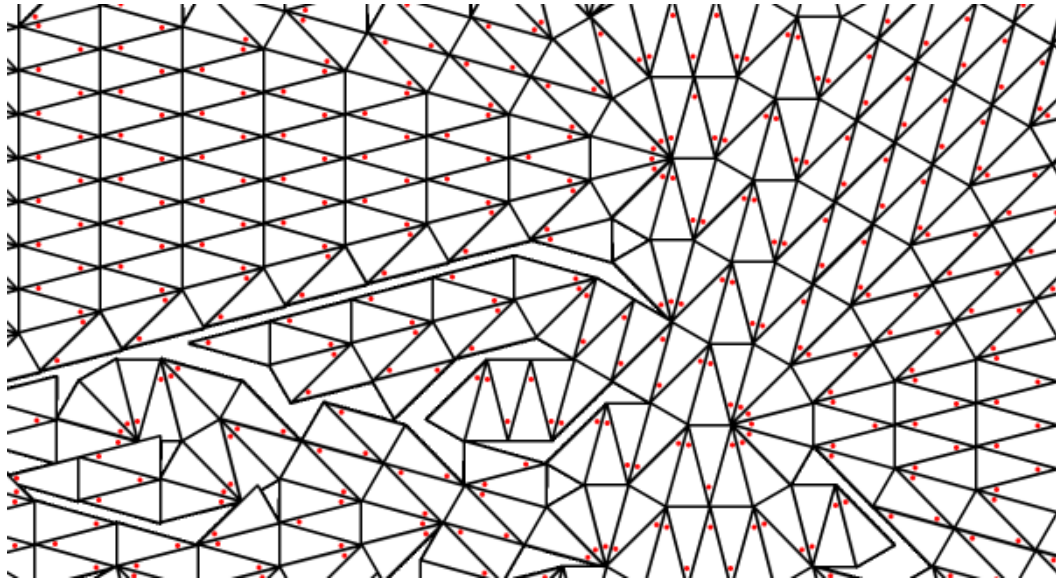


Figure 87. Isosceles Output.

The problem here is a more fundamental one, caused by the `tileLocs()` algorithm. The point pairs definition for the tile encourages the development of two different large-scale patterns – layers of parallelograms, and wheels of triangles radiating from a central point. When these patterns meet each other, one will block the other, and `tileLocs()` will be unable to completely surround a point with tiles. When a point cannot be completely covered with tiles, `tileLocs()` gives up and moves onto the next point in the tile. A better solution would be to have the tiling function backtrack over its earlier drawing choices and try different tile pairings (see section 9.3. for more discussion of this).

## 9.2. Using `tileSpacey()`

The jFAT download contains many examples using `tileSpacey()`, including: `DrawCurves.java`, `DrawPenrose.java`, `DrawQuarters.java`, `DrawShapes.java`, `ImPentagons.java`, `Isoceles.java`, and `Pentagons.java`.

When a programmer is trying to decide whether to use `tileLocs()` or `tileSpacey()`, `tileSpacey()` is a better choice if the tiling is meant to include gaps. For example, `DrawQuarters.java` uses an irregular shape [6] shown in Figure 88.

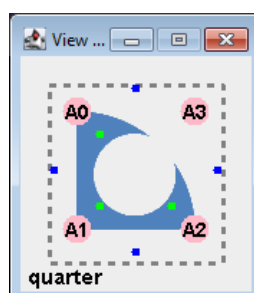


Figure 88. The Quarter Tile.

When this is tiled over the screen, roughly one quarter of each tile is transparent, leaving a lot of intended space (see Figure 89).

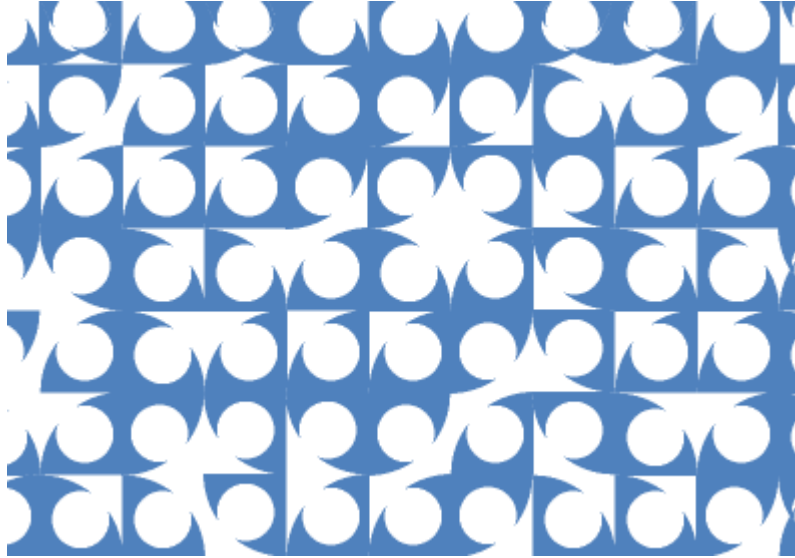


Figure 89. Quarters Output using `tileSpacey()`.

Incidentally, there are some incorrectly placed tiles in Figure 89, along the top row on the right hand side. This indicates that the placement of the inner points could be improved.

The `main()` function for `DrawQuarters.java`:

```
private static Tile quart;

public static void main(String args[])
{
    quart = new Tile("data/quarter.png");
    quart.addPt("A0", 19, 18);
    quart.addPt("A1", 19, 100);
    quart.addPt("A2", 101, 100);
    quart.addPt("A3", 101, 18);

    PointPairs ptPairs = new PointPairs(quart);

    Function<String, Tile> genTile =
        (String nm)->{return quart.clone();};

    // draw tiles on the screen
    ImageViewer iview = Pics.view("Draw Quarters", Pics.altScreen());
}
```

```

Search.tileSpacey(quarter, ptPairs, iview, genTile);
Pics.save(iview.getImage(), "SavedPics/quarters.png");
} // end of main()

```

The arguments passed to `tileSpacey()` are exactly the same as for `tileLocs()`.

If the call to `tileSpacey()` is replaced by `tileLocs()`, then the result is worse (see Figure 90), and takes considerably longer to be generated due to excessive backtracking between tile choices. The main reason for that is because the tile's A3 point is not over an image pixel.

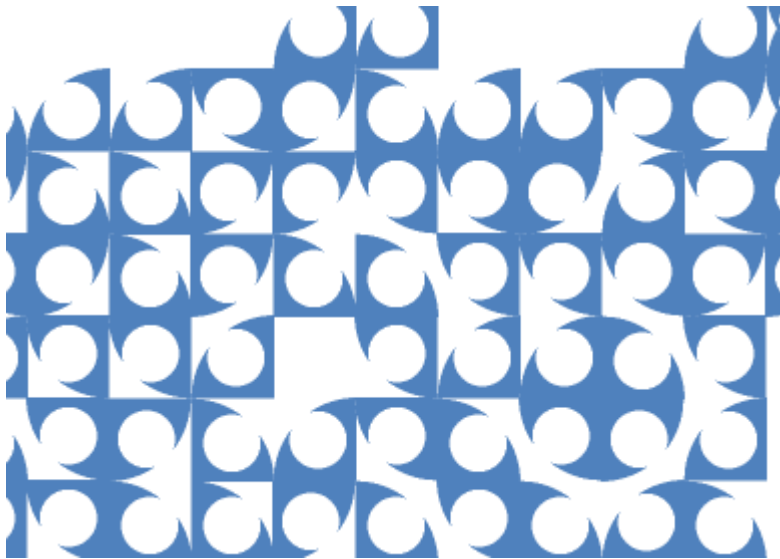


Figure 90. Quarters Output using `tileLocs()`.

Another example of the superiority of `tileSpacey()` over `tileLocs()` can be seen in `ImPentagons.java` which tiles the screen with regular pentagons like the one in Figure 91.

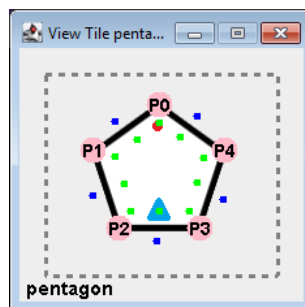


Figure 91. The Pentagon Tile.

Tiling with regular pentagon will leave gaps, but `tileSpacey()` doesn't mind, as shown in Figure 92.

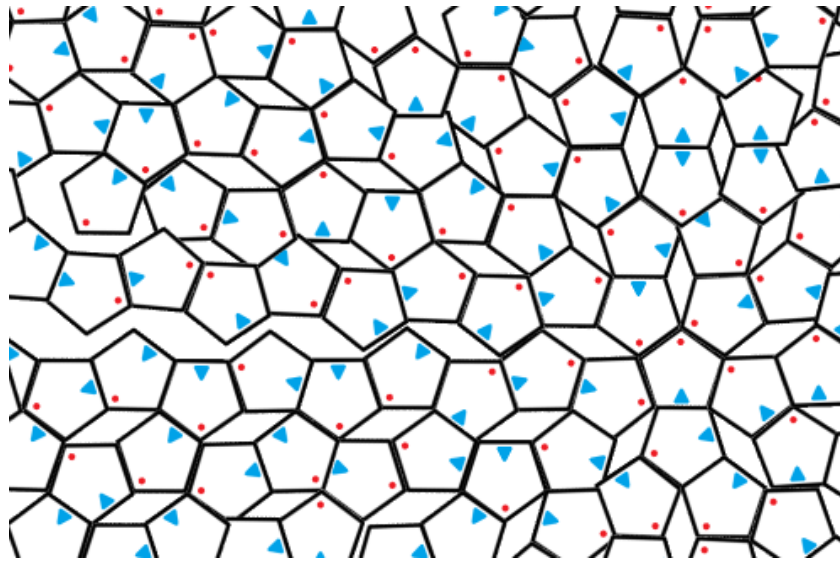


Figure 92. Pentagon Output using `tileSpacey()`.

Figure 92 also highlights inaccuracies in the placement of the P0 to P4 points on the image which introduces some unintentional gaps between the tiles.

If the call to `tileSpacey()` is replaced with `tileLocs()`, all of these spaces defeat the algorithm and only the first tile is drawn (see Figure 93).

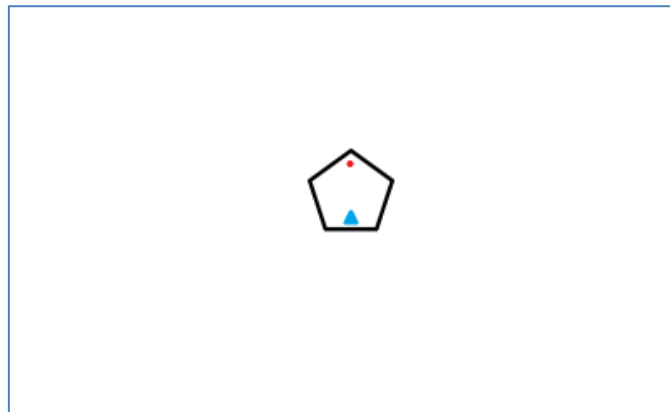


Figure 93. Lonely Pentagon Output using `tileLocs()`.

### 9.3. Improving the Tiling Algorithm

Before I talk a little about how to make jFAT's tiling algorithms better, Figure 94 shows the tiling generated by `DrawPenrose.java` when it utilizes `tileLocs()`.



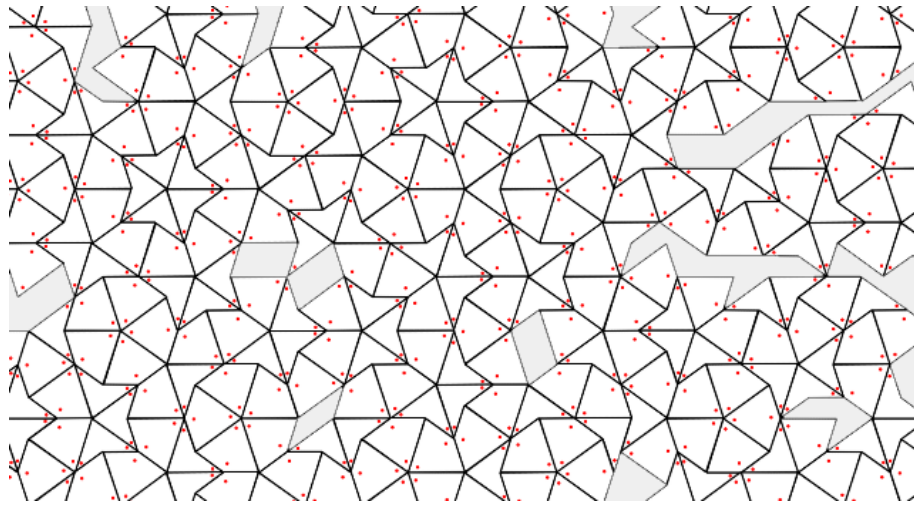


Figure 94. Penrose Tiling Output.

Figure 94 is a good example of why jFAT stands for "Fairly Accurate Tiling". The large-scale patterns formed by the kite and the dart, such as the star and the sun (see Figures 39 and 40), are present, but so are quite a lot of spaces.

Although `tileLocs()` utilizes a limited form of backtracking when trying to surround a point with tiles, backtracking is not employed at a higher level. If it's not possible for a point to be surrounded by tiles then `tileLocs()` moves onto the next available point or tile. What it should really do is backtrack through its list of already drawn tiles, undoing those drawings in order to try alternative tile matches.

The drawback with "full" backtracking is the computational expense. There may be many tiles between the one that reached a dead end (i.e. a point that could not be fully tiled) and the earlier tile that caused this problem. For example, that tile could stick out too far or be positioned incorrectly. Backtracking to that badly behaved tile will most likely have to undo (and redo) the work of many perfectly fine intermediate tiles.

A glimpse of what I'm talking about is possible by uncommenting a debugging line of code in `tileLocs()` which prints a drawing number in the center of each tile. When `DrawPenrose.java` is executed with this version of `tileLocs()`, it produces Figure 95.

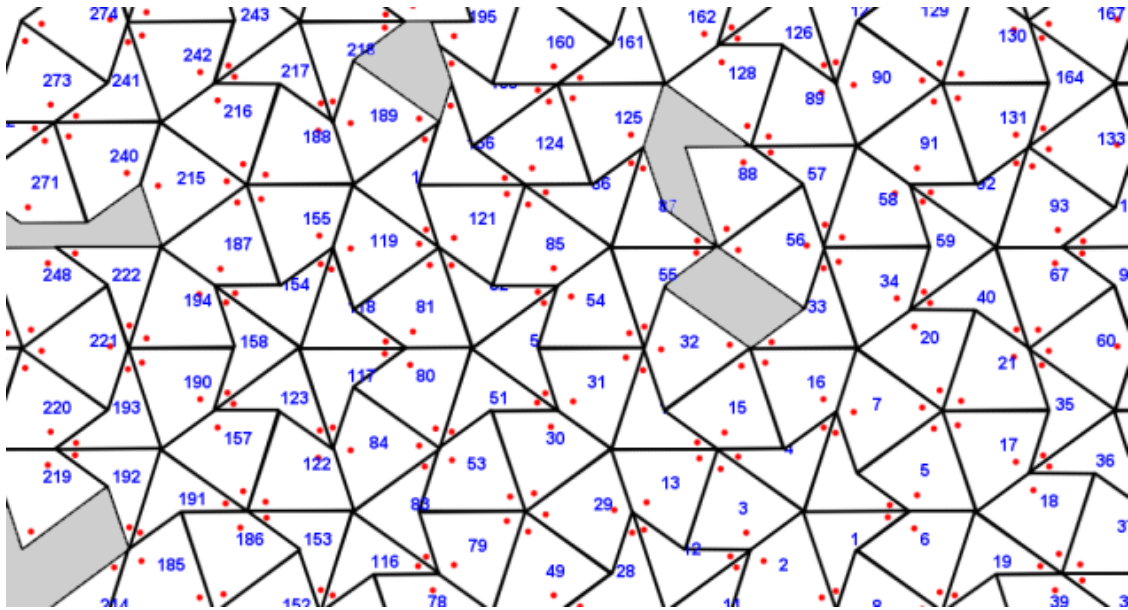


Figure 95. Penrose Tiling Output with Debugging Turned On.

The numbers on the tiles give a rough indication of the order in which they were drawn. The parallelogram-sized gap just to the right of the center of the image is surrounded by tiles numbered 15, 16, 32, 33, 55, and 56. In order to 'fix' that gap, the algorithm must backtrack from tile 56 to tile 33, or perhaps even to tile 16, an interval of at least twenty 'correct' tiles.

"Full" backtracking may theoretically solve this problem, but is too complex and slow to be a feasible solution. Instead, some form of "intelligent" backtracking is needed which could allow intermediate tile drawings to be unwound as a single operation (e.g. undo the work of tile 56 back to tile 33 in one step). Alternatively, tile 33 could be moved up the ordering of drawn tiles so it precedes tile 56.

Unfortunately, these approaches require spatial knowledge of the tiling (i.e. that tile 56 is 'next to' tile 33, or 'near to' tile 32), and some way to identify an 'incorrectly positioned' tile.

## References

- [1] Jinny Beyer, 1999, *Designing Tessellations: The Secrets of Interlocking Patterns*, McGraw-Hill.
- [2] Emmanuel Chailloux and Guy Cousineau, 1992, "Programming Images in ML", Proc. of the ACM SIGPLAN Workshop on ML and its Applications.
- [3] Robert Fathauer, 2020, *Tessellations: Mathematics, Art, and Recreation*, AK Peters/CRC Press.
- [4] Sigbjorn Finne, and Simon Peyton-Jones, 1995, "Pictures: A Simple Structured Graphics Model", In the Glasgow Functional Programming Workshop, Ullapool, July, <https://www.microsoft.com/en-us/research/wp-content/uploads/2016/01/picture.pdf>, last accessed: June 2, 2022.
- [5] Martin Gardner, 1997, *Penrose Tiles to Trapdoor Ciphers... and the Return of Dr. Matrix*, Chapters 1 and 2, Cambridge University Press, Revised, Originally published by W. H. Freeman, 1988.
- [6] Branko Grünbaum and G.C. Shephard, 2016, *Tilings and Patterns*, Dover Pub., 2nd Ed., Originally published by W H Freeman & Co., 1987.
- [7] Peter Henderson, 2002, "Functional Geometry", Dept. of Elec. and Comp. Sci., Univ. of Southampton, <https://eprints.soton.ac.uk/257577/1/funcgeo2.pdf>, last accessed: June 2, 2022. See also <https://mapio.github.io/programming-with-escher/>
- [8] J L. Locher, 2013, *The Magic of M.C. Escher*, Thames & Hudson, Reprint.
- [9] J.M. Spivey, 2005-2012, "GeomLab Workbook", Dept. of Comp. Sci., Univ. of Oxford, <https://www.cs.ox.ac.uk/geomlab/>, last accessed: June 2, 2022.
- [10] Eric W.Weisstein, 2022, "Penrose Tiles." From MathWorld – A Wolfram Web Resource. <https://mathworld.wolfram.com/PenroseTiles.html>, last accessed: June 2, 2022.