Tiles of the Plane: 6.891 Final Project

Luis Blackaller Kyle Buza Justin Mazzola Paluska

May 14, 2007

1 Introduction

Our 6.891 project explores ways of generating shapes that tile the plane. A plane tiling is a covering of the plane without holes or overlaps. Figure 1 shows a common tiling from a building in a small town in Mexico.

Traditionally, mathematicians approach the tiling problem by starting with a known group of symmetries that will generate a set of tiles that maintain with them. It turns out that all periodic 2D tilings must have a fundamental region determined by 2 translations as described by one of the 17 Crystallographic Symmetry Groups [1]. As shown in Figure 2, to generate a set of tiles, one merely needs to choose which of the 17 groups is needed and create the desired tiles using the symmetries in that group. On the other hand, there are aperiodic tilings, such as Penrose Tiles [2], that use different sets of generation rules, and a completely different approach that relies on heavy math like optimization theory and probability theory.

We are interested in the inverse problem: given a set of shapes, we would like to see what kinds of tilings can we create. This problem is akin to what a layperson might encounter when tiling his bathroom — the set of tiles is fixed, but the ways in which the tiles can be combined is unconstrained. In general, this is an impossible problem: Robert Berger proved that the problem of deciding whether or not an arbitrary finite set of simply connected tiles admits a tiling of the plane is undecidable [3].

However, a good way of exploring potential tilings is to try to cover any given bounded region of the plane. Indeed, the Extension Theorem [3] states:

> Let S be a finite set of different simply connected tiles. If the tiles of S can tile any arbitrarily large bounded region of the plane, then the tiles of S can tile the whole plane, even if there is no known procedure to do so. [1]

For our purposes, the theorem means that if we can tile arbitrary bounded portions of the plane, then there is hope for a full tiling, and since real-world floors are always bounded (by the bathroom walls, for example), we need only search for the right tiling.

For this project, we limited ourselves to exploring how to make periodic tilings of the plane derived first from a single input shape, usually a triangle, and then from a small collection of different shapes. As a further simplification, we allow our tilings to have holes. We do not view this as a limiting simplification, because it is always possible to fill the holes with combinations of our input shape in a post-processing step.

2 Code Design

We divided our program into two parts, shape generation and tile generation, each of which relies on an additive set of user-defined rules. The user-defined rules embody the approaches a layperson may take when designing tilings. For example, the shape generator might have a rule that combines two triangles to

[1] Theorem text adapted from `http://www.spsu.edu/math/tile/aperiodic/penrose2/extension_theorem.htm`

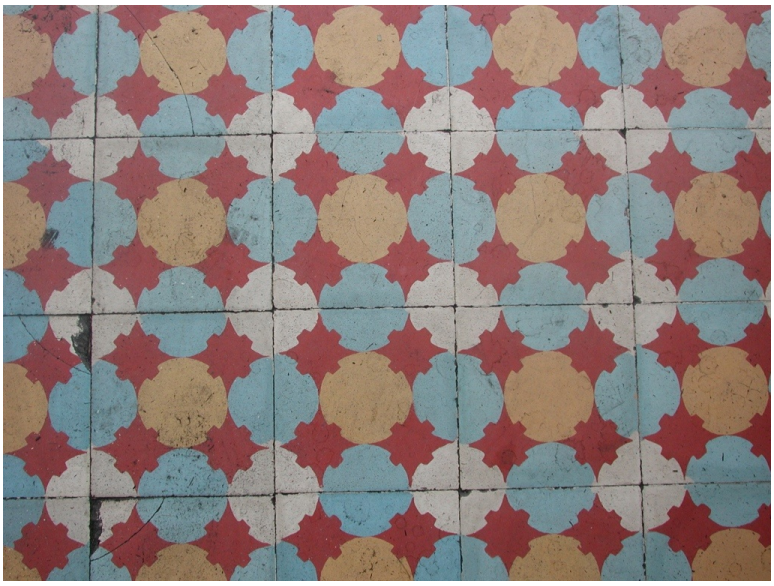


Figure 1: A simple set of tiles with rotational symmetry.

make a couple of parallelograms, while the tile generator may have a rule that tiles the plane by making a fundamental region out of these parallelograms joining them over their common edges.

Our shape generation code is embodied by a shaper module that makes new shapes based on an input shape. The shapes that the shaper generates are then passed to the tiler module, which fills the plane by combining and rotating shapes. Figure 3 illustrates.

We kept two design goals in mind for both of the tiler and the shaper:

1. Each module should allow the user easily add new knowledge to the system. For example, if the user knows that certain classes of input shapes can produce a useful set of output shapes, then the user should be able to encode this knowledge in the system without having to modify large amounts of code. This fits in with our layperson's approach to the problem: knowledge may be added to the system as the layperson gains experience with the tiling problem over time.

2. There should be hooks that allow the user to visualize every step in the process. This aids in debugging and makes the system more fun to program.

Our code also includes two visualization programs that we hooked into our search procedures.

2.1 A Language of Shapes

In order to explore shapes, we need a set of abstractions to describe shapes as well as actions that enable us to compose different shapes to make new shapes.

We chose to use a combinatorial abstraction: a shape is a circular, directed list of edges and angles between those edges. Combinatorial representations are useful because they allow us to concentrate on the intrinsic properties of a shape — such as what kinds of angles it has — without having to transform coordinates. For example, we describe a unit square as `((1 1/2) (1 1/2) (1 1/2) (1 1/2))` — four sides of length 1, each with an angle of $\frac{1}{2}$ π-radians.[2]

[2]We chose π-radians as our unit of angle measure so that our system deals primarily with rational angle numbers. In our units, the sum of all angles in a triangle is 1, and in a square, 2.

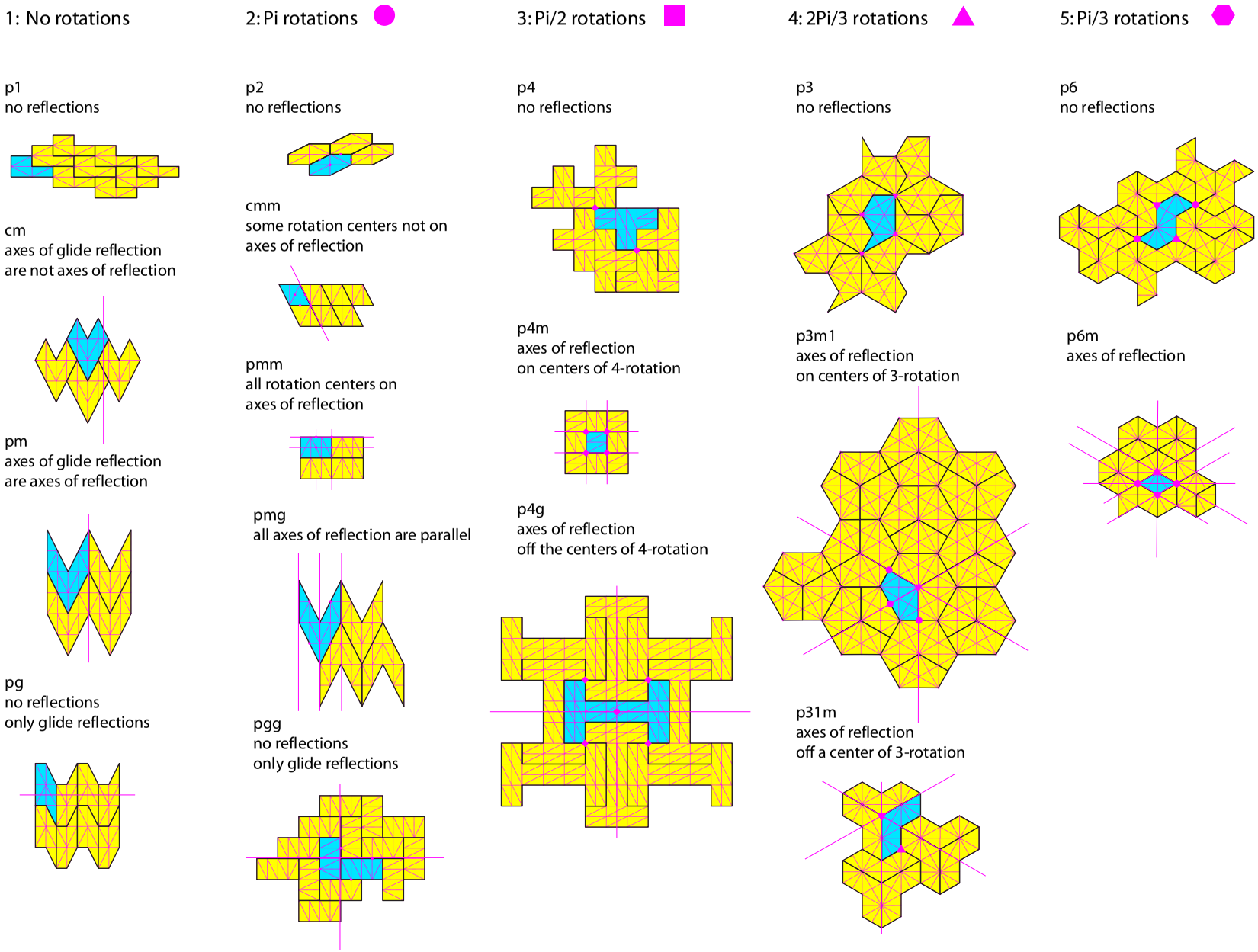


Figure 2: The 17 Crystallographic Groups

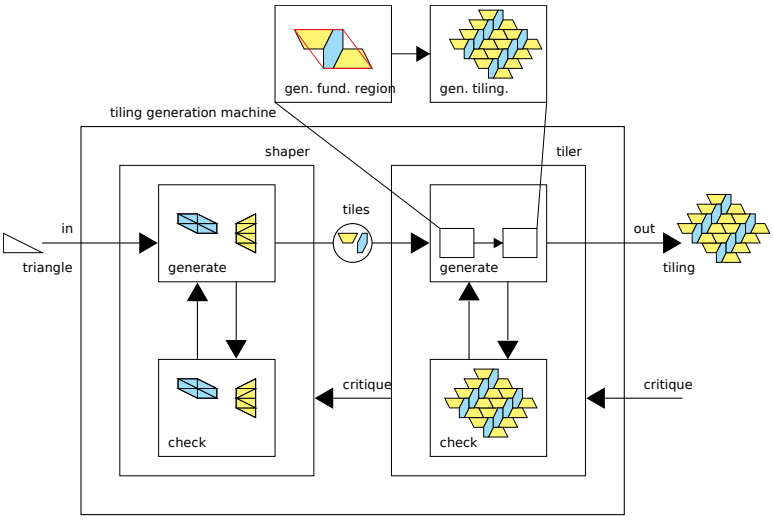


Figure 3: The two-part architecture of our code. Not shown are our visualization procedures.

The two directions in which one traverses a the edge list of a shape correspond to the two orientation vectors a shape may have relative to the plane; traversing the edge lists "backwards" corresponds to flipping a shape over in the plane. We chose to keep orientation in our abstraction because some real-world shapes, such as ceramic bathroom tiles, cannot be flipped over because one side is rougher than the other. Finally, we restrict our abstraction to only simple polygon shapes — a shape cannot have holes or fold onto itself.

On top of our abstraction, we provide two primitive operations that create new shapes. The first operation, `(shape:flip shape)`, creates a new shape that is identical to its argument, but with the opposite orientation. All abstract shapes can be flipped, so this procedure always produces a valid shape.

The second operation, `(shape:fuse-sides shape1 side1 shape2 side2)` creates a new shape by joining `side1` of `shape1` to `side2` of `shape2` and removing any coincident edges. Fusing two sides is equivalent, in the real world, to gluing the edges of two shapes together to produce a new composite shape. Figure 4 illustrates. For simplicity, we require that the lengths of the fused sides be the same. The resulting shape takes the orientation of `shape2`. Unlike `shape:flip`, `shape:fuse-sides` may fail to produce a simple polygon; in such a case, `shape:fuse-sides` returns `#f` to indicate that the fuse has failed.

Our choice of flip and fuse as primitive operations was influenced by the actions a layperson may take with a set of real, physical tiles. A layperson may also cut tiles to make new shapes. We did not implement the cut operation, but it can be implemented within our shape abstraction.

Keeping with our second goal of making it easy to visualize shapes, every shape object keeps a list of "internal edges" that represent the edges that were removed by the fuse operation. Such internal edges allow us to see some of the history of the creation of a shape and help in debugging.

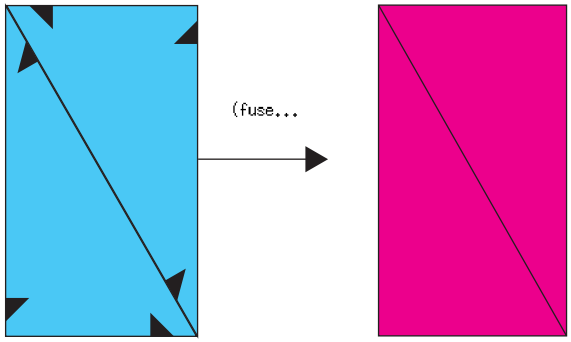
2.2 Shape Generation

Our shaper module exports a simple interface:

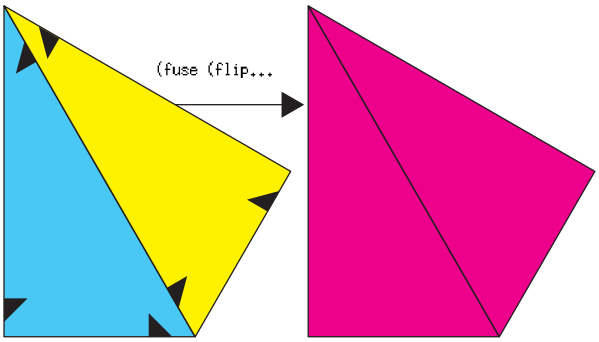
```
(shape-search rules initial-shape)
```

The `shape-search` procedure returns a list of candidate shapes. The search is driven by both the single initial shape and the set of rules that determine how to produce new shapes. We explore shapes by incrementally building a graph[3] until the shaper has produced enough "interesting" shapes. Our search has an option to explore the graph non-deterministically, in case deterministic search gets boring.

[3]We use a version of Problem Set 6's graph code modified to allow dynamically generated graphs.



(a) A shape fused to itself.



(b) A shape fused to its flip.

Figure 4: Two examples of the `shape:fuse-sides` procedure

The set of rules is simply a list of predicates that determine when a rule is applicable and a procedure that takes in a shape and returns a list of new shapes based on its input shape. For example, the code:

```
(shape-search
  (make-regular-triangle 1)
  `((,any? ,fuse-sides)
    (,any? ,flip-a-shape)
    (,triangle:is-regular? ,make-hexagon-parts)))
```

produces the shapes in Figure 5.

In order to make rules easier to write our `shape-search` procedure takes care of coalescing equivalent shapes (to avoid excess work in our search) as well as removing invalid shapes from failed fuses. As a result, rules are fairly simple. For example, the a rule that fuses like sides of the same shape is:

```
(define (fuse-sides shape)
  (map (lambda (idx)
         (shape:fuse-sides shape idx shape idx))
       (list-tabulate (description:length (shape->description shape))
                      (lambda (i) i))))
```

We've coded four example rules:

1. the `fuse-sides` procedure above,
2. a rule that fuses a shape with the flip of the shape,
3. a rule the flips shapes, and
4. a rule that when given an equilateral triangle, produces shapes leading up to, and including, a regular hexagon.

The first three rules are generic and will work on any shape. The last rule is specific to equilateral triangles, but is a simple rule that makes interesting shapes; we added it to our examples because it's one of the first patterns that a layperson might generate with a set of equilateral triangles. Figure 6 shows the shapes produced by the last rule.

2.3 Tiler

As the end result of this project is to generate plane tilings, we have created a mechanism similar to the shape generation procedure to construct fundamental regions from individual tiles. Within the context of tilings, a fundamental region is the smallest shape-containing region that generates the tiling when all of the symmetries are applied. Figure 7 illustrates. Intuitively, we know that the symmetry properties of the individual shapes that compose the fundamental region are relevant to the determination of which symmetry group the region lies within. As a result, the tiler contains procedures that can determine the sets of translations and rotations of individual shapes.

We have constructed our tiler as a graph whose nodes are pairs of shapes to be fused together on same-length edges, according to user-specified procedures. Modifying this mechanism to support input shape lists of greater than two elements is trivial, but substantially increases the complexity of the rules to be implemented by the user.

Similar to the shape-generator, an example tiling technique is as follows:

```
(define tile-test
  (tiler
    (list (make-hexagon 1) (make-right-triangle 1 1))
    `((,any? ,filter-fuse-all))))
```

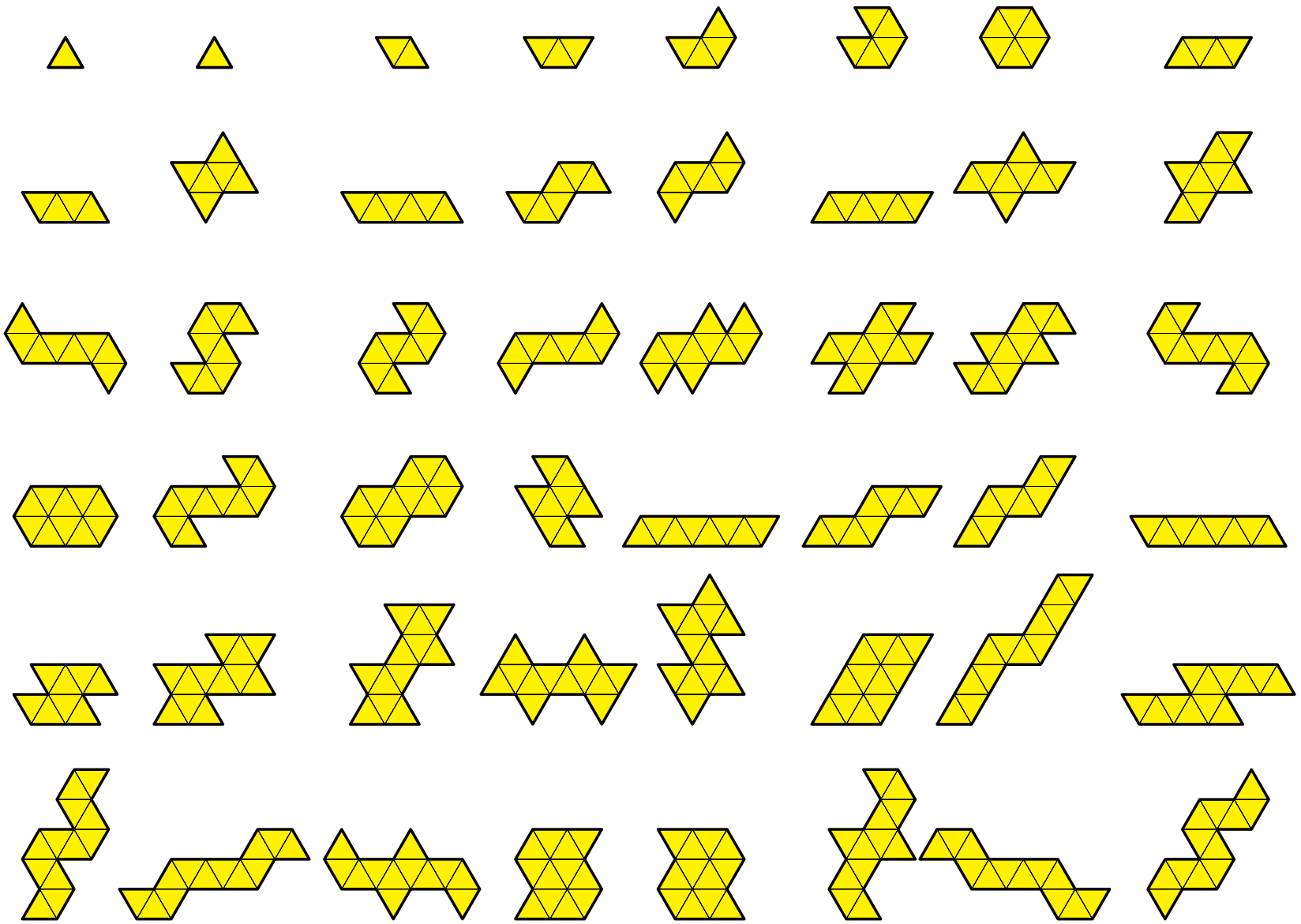


Figure 5: Example shapes produced by the shaper when given an equilateral triangle as input. The shapes are shown with their internal triangles.

Figure 6: The shapes made by our `make-hexagon-parts` rule.

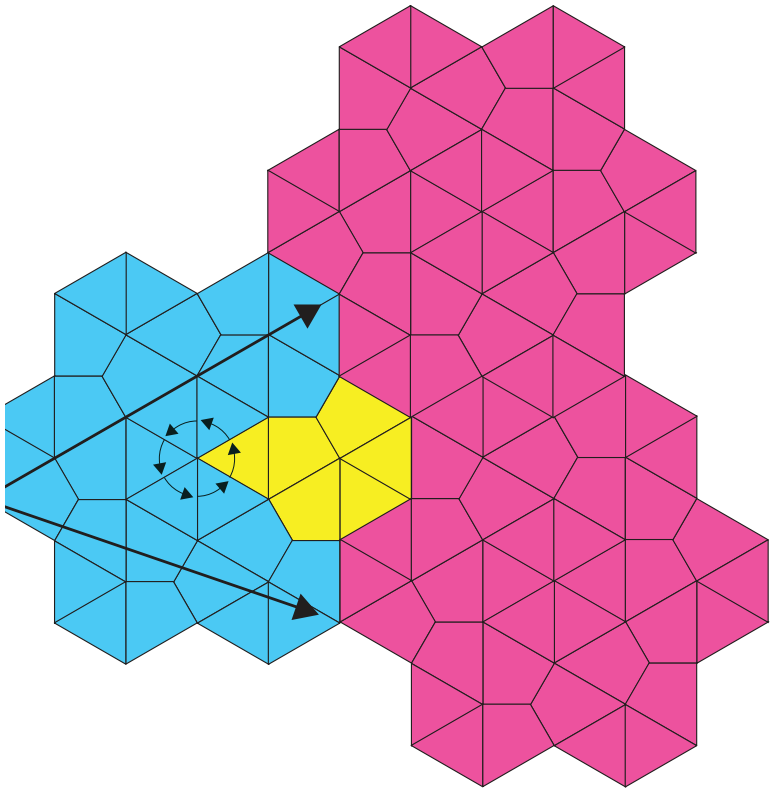


Figure 7: A illustration of a fundamental region with $\frac{\pi}{3}$ rotational symmetry. The yellow shape is the basic tile and the blue shape is the fundamental region.

where the first predicate (`any?`, in this case), is used to select subcomponents of the shapes to which apply the associated rule (`filter-fuse-all`, in this case). In this example, `filter-fuse-all` generates new shapes based on the fusing of sides of equal length from each shape residing within a given node.

The shapes produced by the traversal of this graph may or may not have the symmetry properties necessary to tile the plane: acceptable shapes are those that have a linearly independent set of translations. While this is criterion alone does not guarantee that a given region will tile (our current implementation may generate regions that leave holes, for example), it will produce non-overlapping regions when tiled.

2.4 Visualization and Debugging

Since we were dealing with a visual subject, we needed a natural way to display our data on screen. We wrote a set of procedures to interface between the shapes generated by Scheme and Processing [4], a graphics environment in Java that allows for rapid prototyping of graphics and interactions. We also have a lightweight Python shape visualizer that creates SVG files with the Cairo graphics libraries [5].

It was essential for is to find a tool that could let us look at our processes and data in a variety of ways, because it would have been impossible to debug our program just by looking at the descriptions of shapes and symmetries as nested lists of numbers. It might be easy to recognize a square as the shape with the description `((1 1/2) (1 1/2) (1 1/2) (1 1/2))` after enough experience with the system, but would be nearly impossible to figure out what's going on by looking at a long list of numbers after a few fuse operations.

We hooked our visualization system into the search procedures of both the shaper and the tiler. The shaper sends the visualizer each shape it produces, while the tiler sends both the shapes it uses and the way to translate the shape so it fills the plane. Color support is provided by the visualizer to make it easier to see individual shapes, but there are ways of extending our system to deal with color properties in Scheme, attaching the color information to particular symmetry properties of a tile's role within the tiling, and then using color to describe these properties.

3 Source Code

Our source is available from `http://web.mit.edu/jmp/www/6.891/tiling-src.tar.gz`.

Our system is spread across 12 scheme files. Half of these files are our graph abstraction, and are in the `graph` sub-directory of our source tree. The rest are outlined below.

File	Description
`intersect.scm`	Algorithm to detect intersections of line segments.
`shape.scm`	Our shape abstraction and procedures.
`net-sender.scm`	Integration with our Java-based visualizer.
`triangle-primitives.scm`	Predicates and triangle utility procedures.
`shape-search.scm`	The shaper module.
`tyler.scm`	The tiler module.

4 Gallery

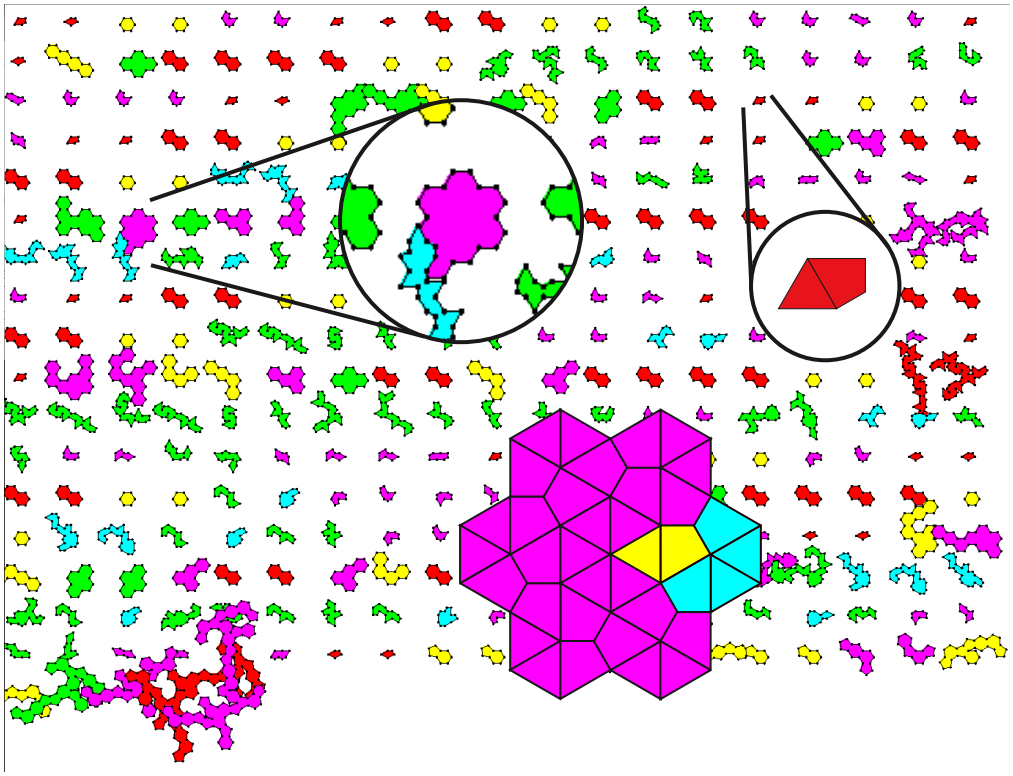


Figure 8: Output of the tiler search showing a fairly complex shape (with $p6$ symmetry) derived from the fuse of an equilateral triangle with a kite.

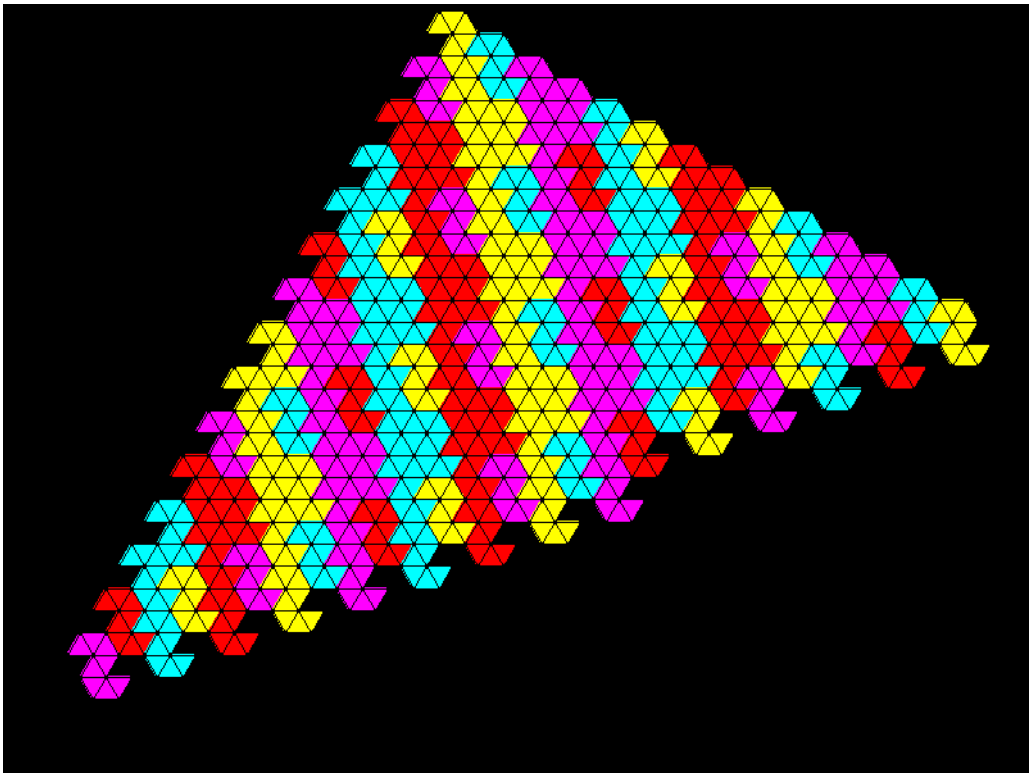


Figure 9: A planar tiling of an s-like shape composed of 8 equilateral triangles.

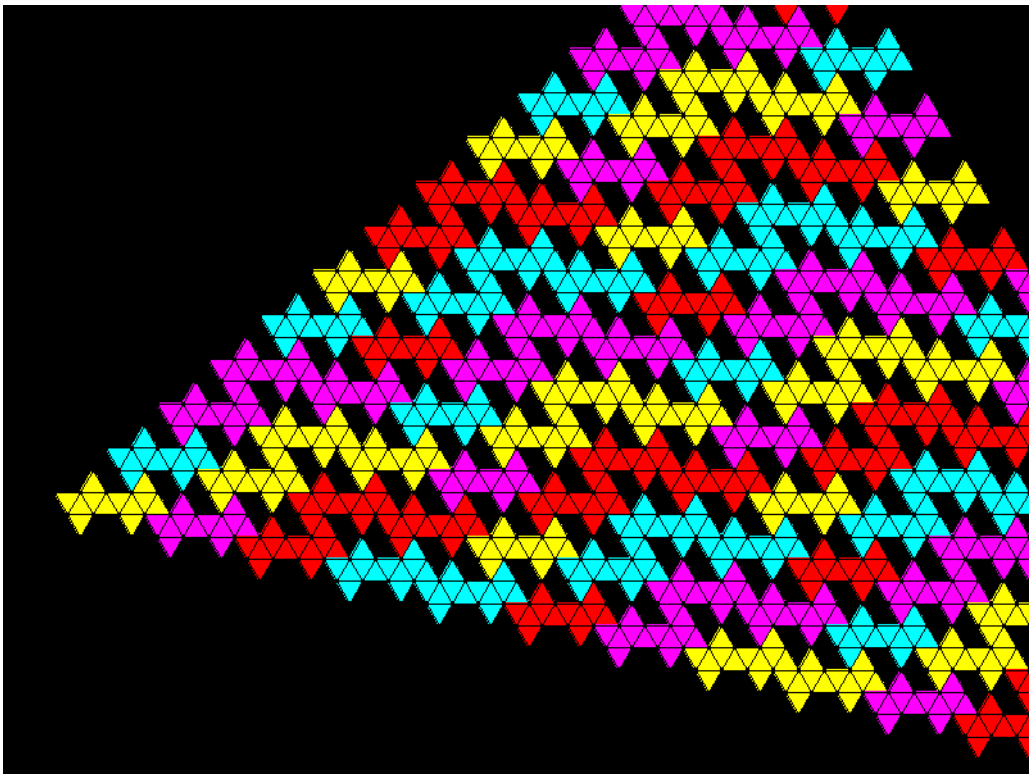


Figure 10: A less-than-perfect planar tiling of shape composed of equilateral triangles. The shapes in the holes are can be generated by the shaper.

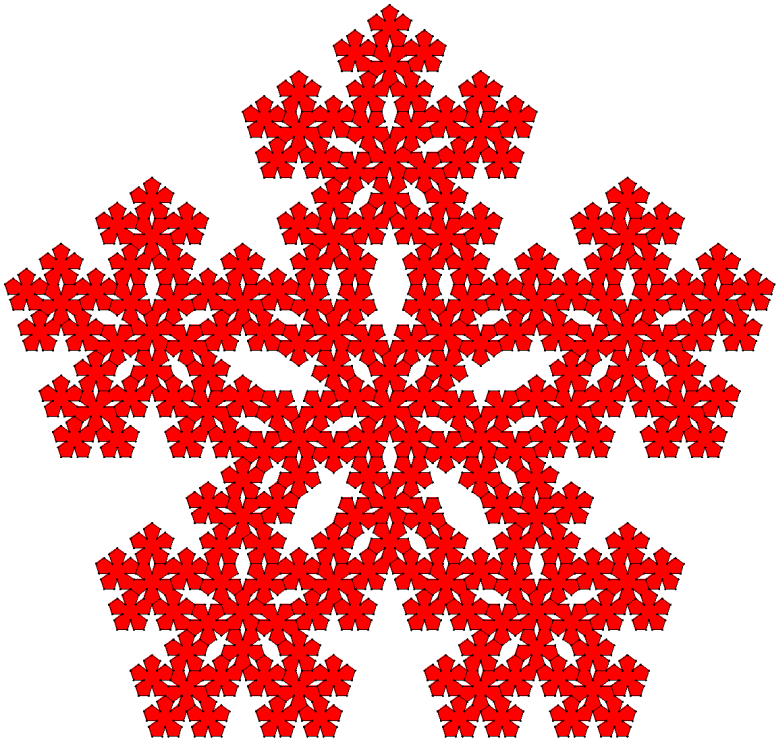


Figure 11: A snowflake tiling manually created by fusing pentagons.

References

[1] Wallpaper groups. http://en.wikipedia.org/wiki/Wallpaper_groups.

[2] Penrose tiling. http://en.wikipedia.org/wiki/Penrose_tiling.

[3] Doris Schattschneider and Marjorie Senechal. Tiling. In *Handbook of Discrete and Computational Geometry*, pages 43–62. CRC Press, 1997.

[4] Processing. http://processing.org/.

[5] Cairo. http://www.cairographics.org/.