CS 791 Project Report: Reptile Playground

Patrick Nicholson

May 11, 2010

Abstract

This report describes an interactive software tool, called *Reptile Playground*, which can be used to assist in the specification of substitution tilings. As we will demonstrate, in many cases we can easily draw sets of substitution rules using Reptile Playground that would otherwise be difficult to specify.

1 Introduction and terminology

Reptile Playground is a program for specifying and drawing substitution tilings. Substitution tilings are tilings of the plane which are created by iterating a set of substitution rules. Many intricate designs can be created from a seemingly simple set of rule. Furthermore, these tilings have practical applications in crystallography, as they are used to model quasi-crystals [6]. Substitution tilings are a great example of a mathematical idea which predates an incredibly useful application; in this case by almost about eight years!

Before going any further, we should provide some idea about what substitution tilings are, and how they work. The goal here is to give some intuition without diving into the actual math behind them. Mathematically rigorous definitions for the topics discussed here can be found in [6, 4, 8, 11]. However, to use and enjoy Reptile Playground, only the basic ideas are needed.

I like to think of substitution tilings as a kind of grammar, and substitution rules as production rules for that grammar. Depending on if you have an undergraduate degree in computer science this comparison might be helpful. In a substitution tiling, we begin with a single tile, analogous to a start symbol in a grammar. To proceed, we replace this initial tile with one or more tiles specified by the substitution rules. The orientation and position of these new tiles is governed by the substitution rules. An example of this can be seen in Figure 1. These new tiles are often scaled down, or *deflated*, versions of either the original tile, or possibly other tiles; each having their own substitution rules[1].

This procedure of replacing tiles by smaller tiles can then proceed ad infinitum, allowing us to tile the plane using simple substitution rules. We refer to the process of replacing all of the current tiles by

[1] Alternatively, we may inflate the original tile before applying the substitution rules. There is no difference between these two methods other than that the size of tiles remains fixed if we inflate first. For our purposes we will refer to the deflation method, since Reptile Playground operates in that way.

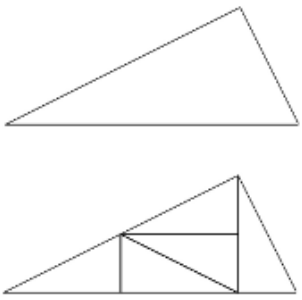

Figure 1: Applying substitution rules: (top) A pinwheel is replaced by (bottom) 5 scaled down pinwheels, oriented to completely cover the original.

Figure 2: Scherer's "B12" irreptile.

new tiles as *iterating* the substitution rules. Since the rules are applied relative to the orientations of the individual tiles, we often abstract the notion of a set of tiles with the same set of substitution rules, referring to this set of tiles as a single *prototile*. Usually a substitution tiling can be represented as a small set of prototiles, and rules which act upon those prototiles.

In general, there is no requirement that the scaled down tiles exactly cover a prototile. In fact, in many cases the scaled down tiles extend beyond the boundary of the original tile, causing *overlapping tiles* as the rules are iterated. In most cases these overlapping tiles are congruent, so there is no difference between iterating the tiling with overlapping tiles and iterating the tiling without overlapping tiles.

We finish our terminology section by explaining the name of the program. A *reptile* is a substitution tiling consisting of only one prototile, in which the rules completely cover the original prototile; see Figure 1. Although it is often the case that the deflation factor is the same for all of the rules within a given tiling, our program supports various deflation factors within a tiling. This allows the user to create so-called *irreptiles* [11], an example of which can be seen in Figure 2.

Reptile Playground is designed so that a user can interactively apply transformations to tiles in order to define substitution rules by what the rules should *do*, rather than specifying transformation matrices manually. Because of this interface, we often abuse proper definitions and refer to the scaled down tiles as rules: for example, we might say Figure 1 illustrates five rules. The reason for this is brevity, as it is much easier to say "five rules" rather than "five tiles specified by five transformation matrices, which when combined comprise the substitution rules for this tiling".

2 Similar software available on the Internet

While developing Reptile Playground, I was able to find a few pieces of tiling software that were related. In no particular order they are:

- Subtile [3]: a Tcl/Tk program designed to display a set of predefined substitution rules to the user. The user can interactively iterate the rules and view the results. However, Subtile does not allow the specification of new rules, other than by hard coding them into the program.
- Quasitiler [1]: a great program for visualizing tilings with quasicrystal structure. The program allows the user to modify several parameters interactively to create colorful tilings. Although Quasitiler uses the "cut and project" method rather than substitution rules, we mention it because the two methods are related.
- Tilewizard [9]: a command line tool for creating substitution tilings. The tilings are specified in a file format, where each rule consists of a transformation matrix and color information. This program was used to create many of the beautiful tilings found on the Tiling Encyclopedia webpage [7].
- We note that there are also several programs available, written in both Matlab or Mathematica, that can be used to generate certain kinds of substitution tilings, for example [5].

Of all of the software that is available, none provide an interactive way of specifying substitution rules. Therefore, interactivity is the main purpose of Reptile Playground.

3 Implementation and Program Description

Reptile Playground was implemented entirely in Java using the Graphics2D library for drawing and Swing for the user interface. For exporting SVG images the Batik SVG Toolkit was used [10]. To give

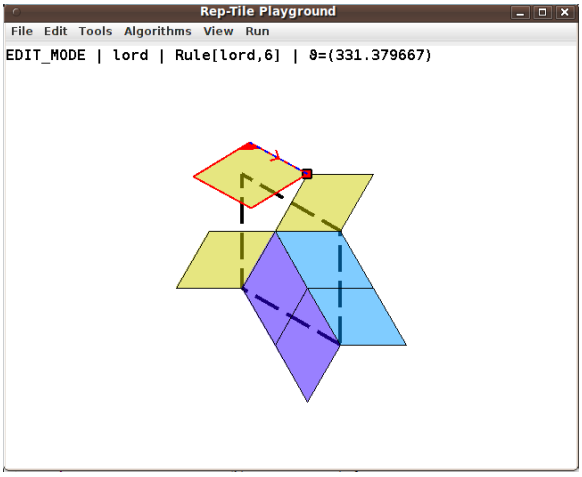

Figure 3: A screenshot of Reptile Playground in edit mode.

an overview of how it works, it is probably most instructive to simply describe the program and how it is used. For this reason, we will jump right into a description of the features by looking at the screenshot in Figure 3.

The first thing to notice is that there is a line of text drawn on the canvas area of the program. This text contains important information about the current state of the program. It can be interpreted as the fields:

`mode` | `prototile name` | `selected rule` | `transformation`

Where each of these fields means the following:

- `mode`: The current mode that the program is in: either `Edit Mode` or `Draw Mode`. `Edit Mode` allows the user to define and manipulate tiles and rules, while `Draw Mode` allows the user to iterate the rules and view the resulting tilings.
- `prototile name`: The name of the prototile that we are editing. In the figure the tile being edited is from the Lord tiling; hence the name "lord". Tile names can be assigned and changed by the user.
- `selected rule`: An identifier for the rule the is currently selected. Since rules map tile A to tile B, this identifier contains the name of tile B as well as a unique number identifier for the rule. In Figure 3, the selected rule maps the lord tile, a rhomb, to another lord tile, and the rule has an id of 6.
- `transformation`: The user can apply certain transformations to rules by dragging them. When the user is dragging the mouse, this field is populated by information about the transformation that is occurring. In Figure 3, a rotation is being applied to a rule, and thus the angle of this rotation is displayed.

The state information string can be toggled via the `View` menu. We will now proceed by describing the two modes of Reptile Playground.

3.1 Edit Mode

`Edit Mode` is the bulk of Reptile Playground, and as such contains most of the features. It is within `Edit Mode` that the user defines tiles and the rules which act upon them. In this section we will describe how this occurs, and the features which make this process easy for the user.

3.1.1 Defining prototiles by creating polygons

Before the user can do anything, they need to define prototiles. To accomplish this, the user must first define the shape of the prototile, by creating a polygon. There are two methods for creating polygons:

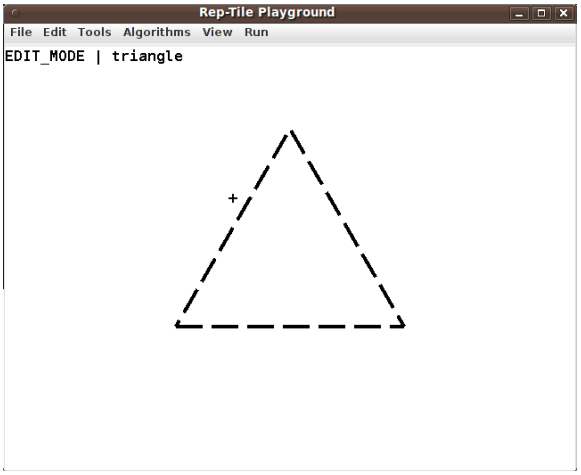

Figure 4: What the user sees when they select the prototile "triangle".

- The first method is to use a predefined macro command. This method can be used if the shape of the tile is somewhat common, such as a regular polygon, rhombus, triangle, etc. The macros usually require the user to enter the necessary information about the polygon into a dialog. For example, the `Triangle (SSS)` macro, located in the `Tools` menu allows the user to draw a triangle by specifying the length of all three of its sides.
- The second method is for the user to draw the polygon using the `Create Polygon` tool. This tool allows the user to draw the polygon they desire with the mouse. A left click adds a point to the polygon, and a right click indicates that the polygon is complete. To allow more exact specification of the polygon, the user can hold `ctrl + shift` when they left click, which will open a dialog allowing them to exactly specify the length and angle of the edge want to add to the polygon.

In both cases, after the user has defined a polygon, they must attach a name to it in order to later identify the polygon. These named polygons are stored internally in the "polygon library". Users can then make a prototile based on any of the shapes in the polygon library. The reason for this level of abstraction is because there may be many prototiles with different substitution rules which all have the same shape. As with the polygons, when a user creates a prototile they are prompted to give it a name.

Defining prototiles is probably the least polished part of Reptile Playground. The problem is that many of the interesting substitution tilings involve very strange polygons, which have weird dimensions. For example, many of the tilings defined by Robert Ammann are rectilinear polygons that have irrational side lengths (see for example the Ammann Chair, A3, A4 [7]). Drawing rectilinear polygons is easy with the `Create Polygon` tool, but there is no good way I can think of for specifying irrational numbers using just the mouse!

3.1.2 Defining Rules

As we discussed earlier, substitution rules are defined by the user in terms of how they should look, rather than the actual transformations that achieve that look. To accomplish this, the user can manipulate the rules via a drag and drop interface.

Before adding a rule, the user must first select a prototile. This can be done using the `Select Prototile` command in the `Edit` menu. Once a prototile is selected, it will appear, centered on the screen, outlined by a thick dashed line; see Figure 4. At this point the user can fix a deflation factor, if they desire, so that all rules added will be automatically scaled to the correct size. This is done by using the `Set Deflation Factor` command in the `Edit` menu.

Now the user can start adding rules. This is done using the `Add Rule` command in the `Edit` menu. A popup will ask the use to select a prototile at this point. This prototile will then appear somewhere on the screen, deflated to be smaller than usual. At this point, we have defined the rule which maps the selected prototile to the smaller prototile. We can define any number of such rules in this way for the selected prototile. However, we need to be able to control the orientation of these smaller prototiles in order to be able to specify nice tilings.

3.1.3 Manipulating Rules

To manipulate a given rule, it must first be selected by the user. This can be done by right clicking on the rule. To cut down time, the user can select many rules at once by holding down `shift` when they right click. When a rule is selected, a small arrow and marking is drawn on the rule in order to give some indication of its orientation; keep in mind that rules can be reflected. However, when a group of rules is selected only one rule is designated as the *main selected rule*. This rule is shown in a brighter color, and has two other features drawn on it:

- **The selected point**: this is shown in the Figure 3 as a square over one of the vertices of the selected rule. When the user right clicks a rule, the closest vertex of the rule to the clicked point becomes the selected point. The selected point acts as the centre for all transformations that are applied to the selected rules.
- **The vector of rotation**: shown in Figure 3 as the dashed blue line between the selected point and another vertex of the selected rule. Imagine the vector as pointing 'out' from the selected point. This vector can be changed by the user by spinning the mousewheel. The point of the vector is to provide a reference for scaling and rotation transformation applied to the selected rules. For example, if the user drags the mouse to apply a rotation, the vector of rotation will be aimed towards the point at which the user is dragging.

Applying transformations to rules can be done entirely using the mouse. The user can apply the `Translate`, `Rotate`, and `Scale` tools to the selected rules in an intuitive way by dragging the mouse on the canvas. However, this alone is not enough to be allow the user to precisely specify rules. Many rules require very precise transformations that would be impossible to specify by simply dragging the mouse. For this reason, there is also a *snap* feature, which automatically moves the mouse's location towards a set of special anchor points. The following snap options are supported:

- `Snap to vertices and centroids`: The anchor points in this mode are the set of vertices and the centroids of the non-selected rules and the selected prototile.
- `Snap to grid`: The anchor points in this mode are a set of regularly spaced grid points. This mode is useful for drawing polyominos: see Section 3.3.
- `Angular snap`: This is a feature that is enabled by holding the `shift` key while applying a rotation. It rounds the angle of rotation to the nearest degree.

All of the substitution tilings I have ever seen are created such that the rules form what I will refer to as a *closed patch of tiles*: a group of tiles where the edges of the tiles only overlap other edges, each tile shares at least one vertex with some other tile, and each subset of the tiles has these properties. Using the transformations described in this section, along with the snap features, we make the following claim:

Claim 3.1. *Assume that the user is able to specify their prototiles using the `Create Polygon` tool. Then using only the mouse, the user will be able to draw any substitution tiling where the rules form a closed patch of tiles, provided the following conditions are met:*

1. *At least one rule r shares a vertex v_0 with the prototile p.*
2. *Let V_r be the set of vertices in r and V_p be the set of vertices in p. Define C_r be the set of line segments $\{(v_0, v_i) | v_i \in V_r \setminus v_0\}$ and C_p to be the set of line segments $\{(v_0, v_j) | v_j \in V_p \setminus v_0\}$. One (or more) of the line segments from C_r is collinear with a segment from C_p.*

Condition two of the claim is a bit complicated, but basically means that r and p are aligned in some reasonable way. It is easy to see that if the user draws the aligned rule first, then the rest of the rules will be easy to draw. The class of tilings this claim covers are reptiles, irreptiles, and many substitution tilings. Examples of tilings that do not fall into this category seem rare. In fact the only kinds of substitution rules that I am aware of that fall into this category are fractal reptiles. See the Future Work section for more information.

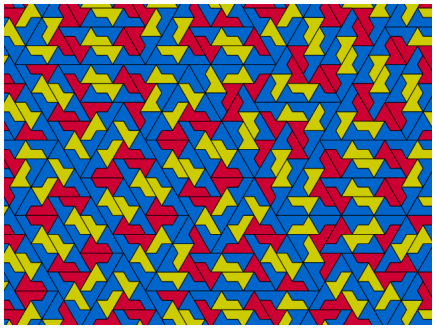

Figure 5: The sphinx-4 tiling drawn using `Rule-based Coloring`. I apologize to all Canadians for the use of "color", Java forced me into it!

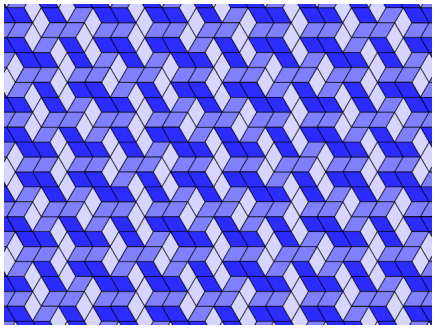

Figure 6: The Lord tiling drawn using `Orientation Coloring`.

3.2 Draw Mode

Now that we have shown `Edit Mode` is powerful enough to create the rules for wonderful substitution tilings, we will turn our attention to actually drawing the tilings. This is done by switching into `Draw Mode` located in the `View` menu. Once in draw mode, the only commands which can be used are those in the `View` and `Run` menus.

As we discussed in the introduction, to begin iterating the substitution rules we need to specify a start tile. This can be done using the `Set Start Tile` command in the `Run` menu. Once the start tile is set, it will appear in the center of the screen. The user can then `Iterate` the substitution rules a few times to see how the tiles change. Since just drawing the tiles in black and white is boring, there are several methods to spice up a substitution tiling. These methods can be toggled in the `View` menu.

3.2.1 Rule-based coloring

When a user defines rules, they have to option to assign them a color. This is done using the `Set rule color` command in the `Edit` menu. This is the simplest method of decorating a tiling, allowing the user to select a color from a Swing ColorPicker dialog box. The sphinx-4 tiling, drawn using rule-based coloring, can be seen in Figure 5.

3.2.2 Orientation coloring

Some substitution tilings, like the Lord tiling, look substantially better when colored not by fixed rules, but rather by how the tiles are oriented. The color this method chooses is determined by the angle of the first two points in a tile, relative to a horizontal line (i.e. take the angle modulo 180 degrees). Call this angle θ. We then define a color having RGB values $(\frac{\theta}{\pi}, \frac{\theta}{\pi}, 1)$, where the values are in the range $[0, 1]$. The last value is simply an arbitrary choice to make the colors have a bluish tint, see Figure 6.

3.2.3 Texture tiles

The final, and most complicated option for decorating the tilings is to use textures. Java's Graphics2D library doesn't really support textures, so this feature is a bit of a hack. The idea is to allow the user to create an image which has the same proportions as a prototile, using a 3rd party graphics program. To

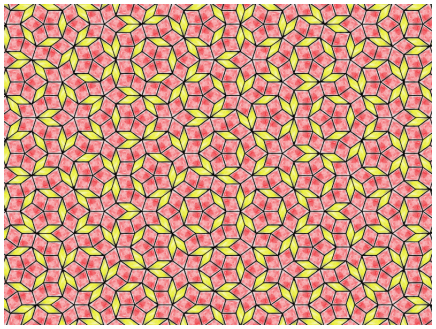

Figure 7: The Penrose rhomb tiling drawn using `Texture Tiling`.

work properly the image needs to have the same width to height ratio as the bounding box of the tile, and transparency set in all pixels outside the boundary of the tile.

After creating a texture, the user can associate it with the selected prototile using the `Set Texture To Prototile` command found in the `Edit` menu. Once the user is back in `Draw Mode`, they can select the `Texture Tiling` option. The images for each textured prototile will be properly transformed, creating a very pleasing tiling, as can be seen in Figure 7.

3.3 Algorithms

The `Algorithms` menu is kind of a mysterious part of the program. Internally, Reptile Playground was developed so that anyone can develop their own algorithms to be run inside `Edit Mode`. The idea was to eventually develop an algorithm that could be used to discover reptiles for arbitrary prototiles. However, designing such an algorithm is quite ambitious, so instead I focus only on polyomino prototiles.

3.3.1 Polyomino Reptile Search

This feature attempts to determine whether or not a given polyomino is a reptile. To use `Polyomino Reptile Search`, the polyomino in question must have been created using `Snap To Grid`. It will not work properly unless the vertices of the polyomino are exactly on grid points. Once a prototile has been created in this manner, the user can select it and execute the search.

As input to the search, the user will be asked to input a scaling factor. The scaling factor indicates how many copies of the polyomino the algorithm needs to pack into the original prototile. For example, a scaling factor of two will get the algorithm to try to pack four copies of the polyomino, each having edge lengths $\frac{1}{2}$ that of the original.

The second part of the input is the number of seconds the user is willing to wait for an answer. Determining whether or not a polyomino packs itself for some scaling factor is not an easy problem. The algorithm I use is a very simple backtracking algorithm, which runs into problems once the scaling factor gets to around ten[2]. The program is set up so that the search will terminate and display a message if it fails to find a solution within the allotted time. If however, it does find a solution, the set of rules will be automatically created and displayed to the user.

Unfortunately, going into this, I didn't realize that most polyomino reptiles are pretty boring. With the exception of the P-pentomino, most of the packings that the simple algorithm can discover are trivial, and do not make very interesting tilings. There are several interesting packings that exist at high scaling factors, which can be viewed at [2]. However, without a significant rewrite of the backtracking algorithm, these kinds of packings are out of reach.

4 Examples

With the exception of the drawing features, which are easy to described using still images, most of the features we have discussed are purely interactive. As such, I have created several videos to illustrate the features. The videos can be found at `http://cs.uwaterloo.ca/~p3nichol/rtpg/videos`.

[2]The algorithm does contain a few simple checks to prune away bad search branches, but nothing fancy.

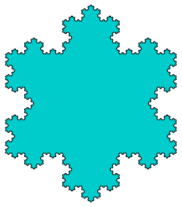

Figure 8: The Koch snowflake

5 Future Work

There are many paths for future work on this program. The main feature that I wanted to implement but did not get around to was the ability to draw fractal reptiles. As it turns out, the rules that govern fractal reptiles are actually quite different than those that govern regular reptiles. Implementing the fractal rules would not be an immense amount of work, but to get them to working alongside one another elegantly seems difficult. It would probably be easier to write an entirely new program just for fractal reptiles.

I did however implement a very basic motif drawing algorithm. A motif is a simple rule that replaces an edge by a more complicated series of edges at each iteration. An example of a fractal created by iterating motifs in can be seen in Figure 8. So, as it stands, Reptile Playground can do basic fractal drawing, but not fractal reptiles. However, there is currently no good way of specifying motifs within the program.

Finally, as I mentioned in Section 3.3, I wanted to implement a general reptile search algorithm. The input to the algorithm would be the similar to the input to `Polyomino Reptile Search` algorithm: a prototile with an arbitrary shape, and a scaling factor. Since any prototile created by the program has a fixed number of points, I believe it would be reasonable to implement a backtracking algorithm for discovering reptiles. The algorithm would certainly be a lot slower than the polyomino backtracking algorithm, as it would have to perform complex transformations at each step in the search. However, I believe it would work for reasonably small scaling factors.

References

[1] Pierre Baillargeon. Quasitiler. Sourceforge project, September 2008. `http://sourceforge.net/projects/quasitiler/`.

[2] Andrew L. Clarke. The poly pages. `http://www.recmath.com/PolyPages/PolyPages/index.htm?Reptiles.htm`.

[3] Roger Evans Critchlow. Subtile. `http://elf.org/pub/subtile-0.3.tcl`.

[4] N.P. Frank. A primer of substitution tilings of the Euclidean plane. *Expositiones Mathematicae*, 26(4):295–326, 2008.

[5] N.P. Frank and B.J. Rinaldi. Tiling generator 3.0. MATLAB freeware available at `http://vassar.edu.faculty/Frank/default.html`.

[6] C. Goodman-Strauss. Matching rules and substitution tilings. *The Annals of Mathematics*, 147(1):181–223, 1998.

[7] E. Harriss and D. Frettlöh. Tilings encyclopedia. A very large collection of tilings with descriptions and references, `http://tilings.math.uni-bielefeld.de/`.

[8] C.S. Kaplan. *Introductory Tiling Theory for Computer Graphics*, volume 4. Morgan & Claypool Publishers, 2009.

[9] Jan Pieniak. Tilewizard. A Perl program for generating substitution tilings, written at Bielefeld University. Personal communication with D. Frettlöh.

[10] Apache XML Graphics Project. Batik svg toolkit. `http://xmlgraphics.apache.org/batik/`.

[11] C. Richter. Families of irreptiles. *Elemente der Mathematik*, 64:109–121, 2009.