Turtles for Tessellations

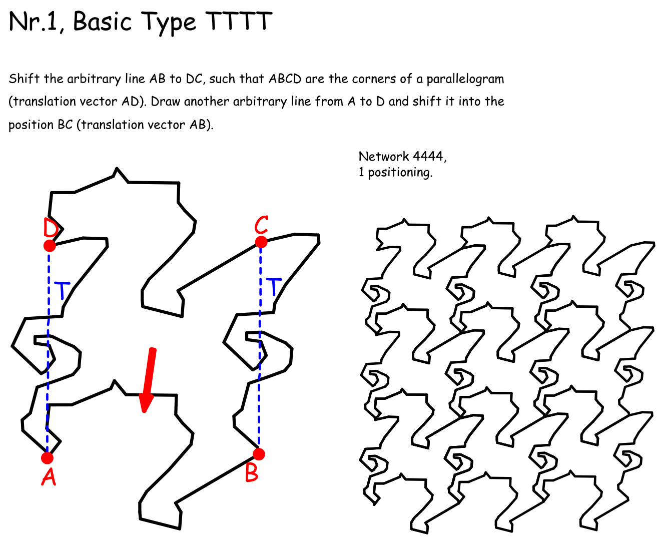
Loe M.G. Feijs, Jun HU
Department of Industrial Design
Eindhoven University of Technology
{l.m.g.feijs,j.hu}@tue.nl

Abstract

We developed an approach to creating vector graphics representations of tessellations for purposes of teaching creative programming and laser cutting. The approach is based on turtle graphics. The lines of the turtle's trail define the tiles of the tessellation. The turtle is defined in an object-oriented style and embedded in the Processing environment as a library. The library is called Oogway. It also facilitates embedding line segments made with different tools such as Illustrator. We present the basic idea, the library, several example and our experiences.

1 Introduction

A tessellation (or tiling) is a collection of figures that fill the 2D plane with no overlaps They are used in textile design [1], interior design [2] and industrial design [3, 4]. Escher [5, 6] added meaning to the tiles, typically fantasy animals. The mathematical theory of tessellation rests on two cornerstones: geometric transformations (and hence group theory), and topology. Figure 1 shows an example of a tessellation (which is a remake of the contours of Escher's Pegasus). The contours of the basic figure are created using turtle commands. Feeding them into another modern tool, a laser cutter, was a source of fresh ideas, inspiration and motivation for our students.


Nr.1, Basic Type TTTT

Shift the arbitrary line AB to DC, such that ABCD are the corners of a parallelogram (translation vector AD). Draw another arbitrary line from A to D and shift it into the position BC (translation vector AB).

Figure 1 : *Pegasus-like figure described by Heesch-Kienzle style recipee.*

We ought to say a few words about the Heesch-Kienzle style recipe mentioned in the caption of Figure 1. The reader is probably familiar with the idea that interesting patterns can be made by repeated application of operations such as translation, rotation, and reflection (mirroring). If there are translations in two or more directions, the patterns are called wallpaper patterns. There is a famous mathematical result which says that in essence there exist precisely seventeen distinct regular wallpaper patterns [7]. Wallpaper patterns appear on wallpaper indeed, in brickwork, in floor tessellations, and so on. It is tempting to assume that each of the 17 regular wallpaper patterns gives rise to precisely one schema of regular tiling. However, the relation is not one-to-one. What is still missing from the wallpaper patterns is: where are the cutting lines between the tiles? This question has been explored by Escher, aiming at his beautiful art. It also has been explored very thoroughly by Heesch and Kienzle (1963) aiming at industrial application and standardization. Heesch and Kienzle were well aware of the 17 essential patterns, but they added an analysis of the networks to be formed by the cutting lines between the tiles. In this way they integrated the network theory with wallpaper theory and found 28 tiling types. They described them in a prescriptive style (in German), which is also the style we used for the (English) description of Figure 1.

Their recipes are very helpful for one who wants to design a new tessellation. This is all well-known and we refer to [3] for the original theory or to [1] and [4] for our earlier applications. Heesch and Kienzle also devised a coding scheme to denote the 28 types. For example TTTT is the type where each tile has four lines and where these lines are related by translation (T) only. Besides translation (T) there is also a code for rotation around a center (C) and for glide-mirorring (G). The coding scheme also includes subscripts, as in the rather complicated type $CG_1CG_2G_1G_2$ but it is outside the scope of this paper to explain all the details.

We present a novel way of designing and generating vector graphics representations of new tessellations, based on turtle graphics. The paper is organised as follows: Section 2 presents the precise turtle graphics commands we use and the motivation behind several of the design decisions. In Section 3 we show the coding of one example tessellation in detail. In Section 4 we show several examples of student work.

2 Overview of Oogway

Turtle graphics has been used in children's mathematics education [9] and in 3D art [8]. Feijs (2012) used turtle graphics in understanding geometry and computation of Pied-de-poule (better known as "houndstooth" in English) in textile design [1], with a variation called Compass Logo that extends the turtle graphics with absolute directions of `NORTH, SOUTH, EAST` and `WEST`, which shows the necessity of extending the turtle graphics for a specific purpose.

In our earlier practice in designing and generating creative graphical tessellations[10], we used the Mathematica programming environment, but then we moved to a graphics language called Processing and a library called Terrapin[1]. Mathematica supports concepts such as rotation matrices quite well, but we consider it worthwhile to explore and extend the possibilities of the first-person perspective of turtle graphics, which seems very natural. We found that most of our industrial design students do not like working with raw (x,y) coordinates: they prefer what-you-see-is-what-you-get tools. The advantage of moving from Mathematica to Processing is that it fits more to the industrial design community (where not many Mathematica users are found). Many of our students love Processing (a low-threshold Java environment) and Arduino (an attractive embedded-systems platform). Terrapin provides a drawing object that implements the standard turtle graphics commands for going forward, backward and making turns, as well as Processing-like `moveToward` command that takes absolute `x` and `y` coordinates. The advantage of using turtles is the constructive approach, which integrates well with transformational geometry (as we shall demonstrate).

It is usually a cumbersome process to describe a smooth curve with turtle graphics. One may first design the curve on a grid paper using a pencil, find out the turning points and calculate the orientation for every step of the turtle, then issue the commands step by step. To make the trace smooth, it is necessary to have a sufficient number of these turning points, yet it is often not smooth enough when the shape is enlarged in vector graphics. To support this process we provided students with a Processing program that imports a

[1]http://terrapin.sourceforge.net/

scanned image, uses mouse clicks to help to determine these turning points, then generates proper Processing code for Terrapin (more about this in Section 3). In the classroom [4] we also observed students combining freehand drawing on paper with graphical drawing tools such as Illustrator or CorelDraw to reduce the load of manual work. We observed a need to combine freehand drawing tools with turtle graphics for the best of both creativity in freehand drawing and automation with turtle graphics. The Oogway library[2] for Processing is created to accommodate this need. The theory of smooth parametric curves such as Bézier curves and Catmull-Rom Splines is so well embedded into tools that they are usually not combined with programming at all, only WYSIWYG editing. But for tessellations and creativity one needs the power of programming, for repetitions of course, but most of all for implementing non-standard special ideas.

The Oogway library is based on the original Logo programming language with extensions for the smooth curves and externally defined segments. Unlike Logo, we need not design generic programming devices such as recursion, assignments and lists: we use these things from Processing (essentially Java). One nice advantage of turtle graphics combined with recursion is that it is easy to make fractal structures (which we did indeed, and which can now be combined with tessellations). The library includes commands for moving and drawing (`forward`, `backward`, `right`, `left`, `setHeading`, `setPosition`, `home`), querying the turtle's state (`xcor`, `ycor`, `heading`, `towards` and `distance`) and controlling the pen (`penDown`, `penUp`, `penSize` and `penColor`). In particular, to introduce smooth curves into turtle graphics, it also includes pairs of commands to draw curves. These commands are: `beginSpline`, `endSpline`, and a pair `beginSpline(x,y)`, `endSpline(x,y)` where the floats x and y are the coordinates of extra control points.

```
beginSpline();
//forward and backward commands here
endSpline();
```

The current position of the turtle, and all the new positions reached by `forward()` and `backward()` (not including `setPosition()`) between `beginSpline()` and `endSpline()`, will be used to create a spline curve as the trace on the canvas. The starting position and the ending position are repeated as control points (which is a common practice), unless x and y parameters are used.

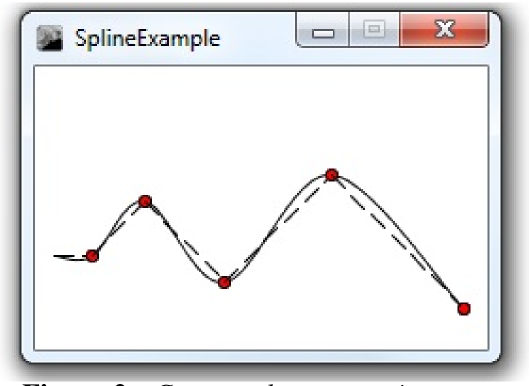


Figure 2: *Command sequence interpretation by classical turtle (dashed) and as a spline.*

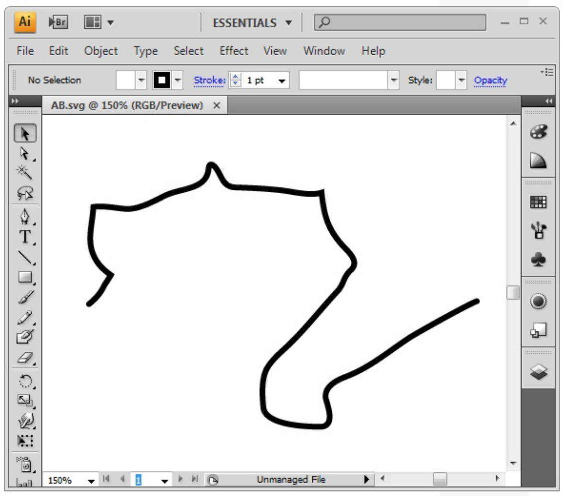

Ai
ESSENTIALS
File Edit Object Type Select Effect View Window Help
No Selection
Stroke: 1 pt
Style:
Opacity
AB.svg @ 150% (RGB/Preview)
150%
Unmanaged File

Figure 3: *Preparing a line segment AB in Adobe Illustrator.*

In Figure 2 we show the result of a turtle performing `o.forward(20); o.right(-45); o.forward(40); o.right(90); o.forward(60); o.right(-90); o.forward(80); o.right(90); o.forward(100);` where

[2]https://github.com/mrhujun/nl.tue.id.oogway/

the dashed line would be a classical turtle and the full line is the trace made by Oogway with the same commands put inside `o.beginSpline()` and `o.endSpline()`. The curved line is a Catmull-Rom spline, which goes *through* the control points (unlike Bézier curves). Formally, it is a parameterized curve: for one segment between P_1 and P_2 the parameter t runs from 0 to 1 and then the point $P(t)$ is a weighted average of four points P_0,P_1, P_2,and P_3. For each segment one needs four control points, but the first two are always taken the same, as usual, and then the last two control points of one segment are made to coincide with the first two of the next segment.

In the same way there is a pair `beginPath` and `endPath`. The first has a string argument, referring to a file in `svg` format (scalable vector graphics). Used as a pair they set a scope for the commands inside.

```
beginPath(svgfile)
//forward and backward commands here
endPath()
```

Additional commands: `beginPath`(svgfile) will load a path/curve from the specified SVG file as the trace for every `forward`() and `backward`() until `endPath`(). Oogway supports SVG files created with Inkscape and Adobe Illustrator that contain simply one path or polylines that are created using multiple vertices. If more than one path is included in the SVG file, only the first one will be loaded.

With these extensions, students are able to first create the curves for constructing the basic shapes with the tools that they are familiar with, then import them into Processing to tessellate these shapes without losing the quality of curves. In the next section we are going to show an example of using the Oogway library in tessellating the well-known Pegasus pattern invented by Escher.

3 Pegasus by Turtle

When working with turtle graphics we like to think in a constructive way. The turtle is our tool to construct things. Not only lines, but also points, distances and angles. It is somewhat like classically constructing geometric objects with only a compass and straightedge. In classical geometric construction one makes circles, lines, but also points (arising as intersections of lines and circles). It is customary to give names to the points thus constructed, writing them onto the paper, like A, B, C, M, etc.

In the same way, any sequence of turtle command begins from a given point and orientation and then will bring the turtle to a new point (i.e. construct the new point) and a new orientation. Since we do not work on paper but in a programming language we do not write the names down but we use programming variables declared in Processing to store their coordinates. We use the naming convention that the two coordinates of a point A are called `Ax` and `Ay`. So drawing a line AB, from the current position is done in three steps: first mark A to be this current position (two assignments, one to `Ax` and one to `Ay`), then draw the line using a `forward` command (three library calls) and then mark B to be the position thus constructed (two assignments, one to `Bx` and one to `By`). The trick is that later we may want to come back to the points A an B which is now easy - at least until we overwrite the variables. Let us assume that these variables have been declared of type `int` and that `o` is of type `Oogway` (`Oogway` is a class) then these are the seven commands for AB:

```
Ax = o.xcor(); Ay = o.ycor();

o.beginPath("AB.svg");
o.forward(ab*scale);
o.endPath();

Bx = o.xcor(); By = o.ycor();
```

For this to work we need a file `AB.svg`, which could be made, for example using Adobe Illustrator or any other tool. We show it in Figure 3 (inside the Illustrator editor window).

Our happy marriage of turtles and vector-graphics allows us to have smooth curved lines produced by the turtle (unlike the old way of making curved paths with turtles where a circle segment would be approximated by a polygon constructed by a sequence of `forward` and `left` steps). The manager of our laser cutter workshop hates these approximated "smooth" curves because the laser cutter will start and stop at the beginning and end of each approximation segment. Now it is time to give a complete example. We assume the following variables declared globally.

```
float Ax, Ay, Bx, By, Cx, Cy, Dx, Dy;
float degreeDAB = 80, ab = 100, ad = 100;
Oogway o = new Oogway(this);
```

It is also possible to do everything with the turtle, avoiding `.svg` files. This is how Figure 1, inspired by Escher's Pegasus, was made (which is also why the latter figure is more spiky and less smooth than Figure 4 made with real curves). We got the coordinates from an existing drawing, using a simple tool we made ourselves, where one can mouse-click on selected points of a loaded bitmap image and then output the coordinate pairs. The output is already in relative polar coordinates and formatted such that a measured angle, say α becomes a command "`o.left(`α`);`" and a measured distance λ becomes "`o.forward(`λ`);`". In other words, the output can be copied into the program directly. Then later we designed the Oogway library which still could use this "pure" approach, or blend in `.svg`.

We shall construct a Pegasus-like horse figure with four points A, B, C and D according to the TTTT type (Nr.1 in the Heesch Kienzle system). The whole approach is designed so we can closely follow the original prescriptive models of Heesch Kienzle (Figure 1 is an example of that, translated from German to English). We give the name `abcd` to the entire procedure defining it as a function of one parameter in Processing. The parameter allows us to run the procedure at different scales.

```
void abcd(float scale) {
     //arbitrary line AB
     Ax = o.xcor(); Ay = o.ycor();
     o.beginPath("AB.svg");  o.forward(ab*scale);  o.endPath();
     Bx = o.xcor(); By = o.ycor();

     //shift AB to DC
     o.setPosition(Ax, Ay); o.left(degreeDAB);
     o.up(); o.forward(ad*scale); o.down();
     Dx = o.xcor(); Dy = o.ycor(); o.right(degreeDAB);
     o.beginPath("AB.svg");  o.forward(ab*scale);  o.endPath();
     Cx = o.xcor(); Cy = o.ycor();

     //another arbitrary line from A to D
     o.setPosition(Ax, Ay);  o.left(degreeDAB);
     o.beginPath("AD.svg");  o.forward(ad*scale);  o.endPath();

     //shift AD to BC;
     o.setPosition(Bx, By);
     o.beginPath("AD.svg");  o.forward(ad*scale);  o.endPath();
}
```

It is nice that the structure of the computer program follows the structure of the Heesch Kienzle prescriptive texts. For example, the original prescriptive text says (in German) "verschiebe die willkürliche Linie *AB* nach *DC*", meaning: shift the arbitrary line *AB* to *DC*. This is done in the first ten lines of `abcd`. The first line, starting with `//` is just a comment. Then the next line memorizes the coordinates of *A* as `Ax` and `Ay`. The third line is about the turtle moving forward over a distance of `ab`, multiplied by a scale factor. Unlike a traditional turtle, which would walk a straight line, the Oogway turtle follows a curved line (e.g. prepared in Illustrator). After that the fourth line memorizes the coordinates of the new point *B* thus constructed as `Bx` and `By`. This concludes the explanation of the first group of four lines: drawing arbitrary AB.

Now we explain the next six program lines, beginning with a comment line again `//shift AB to DC`. The cursor is put back at position *A* and is rotated in order to point towards *D*. Then it travels forward (in a straight line) with the pen up over distance `ad`, multiplied by a scale-factor. That is how *D* is found, after which the rotation toward *D* is un-done (turn right again). Now the same Illustrator-prepared curve is replayed, but from a fresh starting point. And the tenth line, `Cx = o.xcor(); Cy = o.ycor();` records the new position in the coordinates of *C* as `Cx` and `Cy`.

We skip the explanation of the last six lines and mention that the result is shown in Figure 4. Note that the annotations (the letters and the dashed lines) are not yet made by the above program. Although adding such annotations could be done using external tools easily later, here we did it using Processing's built-in commands for texts, ellipses etcetera, using the points constructed by Oogway. Although perhaps "impure", it is an example of Oogway and Processing flexibly blending together.

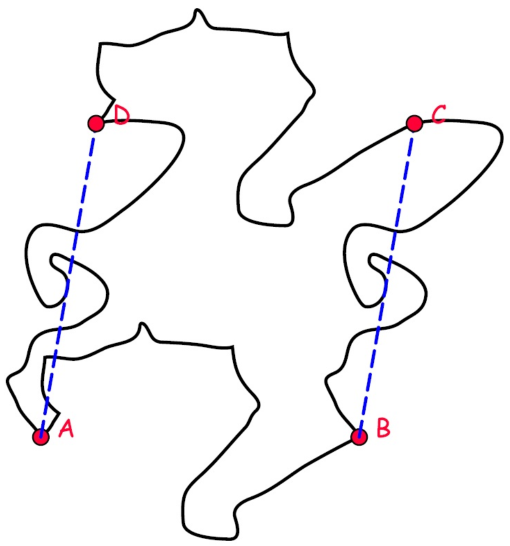


Figure 4 : *Pegasus-like figure and points A-D.*

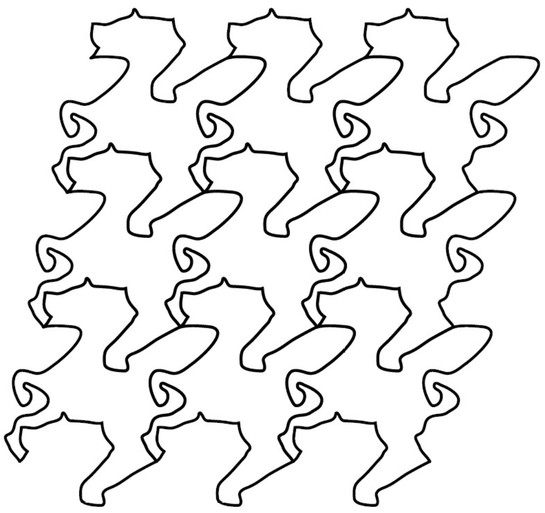


Figure 5 : *Pegasus-like figure tessellated nine times.*

The next thing to be done is to create the tessellation itself. Note how an imperative style of programming is exploited: the points *B* and *D* get updated after each construction by `ABCD` and then the new value of *D* and the new value of *B* are used to play the role of the new starting point (*A*) for the next construction.

```
void tesselate(float scale) {
     for (int i = 0; i<3; i++) {
          ABCD(scale);
          float x = Dx, y = Dy;
          for (int j = 1; j < 3; j++) {
               o.setPosition(Bx, By);
               ABCD(scale);
          }
          o.setPosition(x,y);
     }
}   //the result is shown in Figure 5
```

Besides TTTT, we did CCC (Nr. 3) and CGG (Nr. 21) in the same way, after which we challenged the students of our Golden Ratio module to do their own favourite works and to try other types.

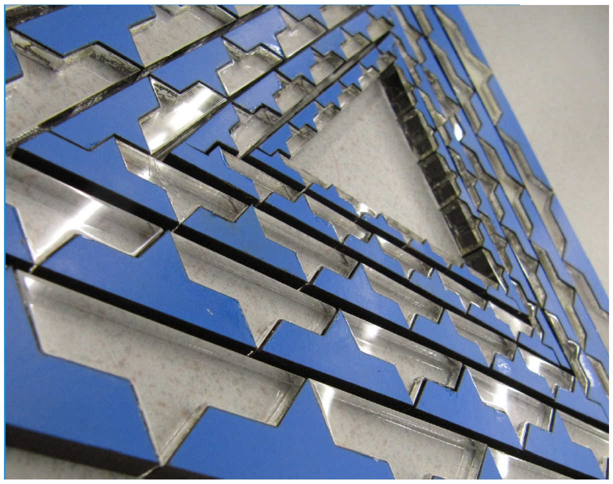


Figure 6 : *Fractal-like artwork by Alberto Gruarin.*

4 Student examples

We show a few of the artworks designed by our students and cut with the laser cutter (a Speedy 300 of TU/e). The first work is by Alberto Gruarin, shown Figure 6. It does not fit in the Heesch typology easily, but it shows something else: that the students picked up the idea of fractals and considered it interesting.

The second work is by Matthijs Willems, shown Figure 7. He deployed the more complicated Heesch-Kienzle type CG1CG2G1G2 (Nr. 28).

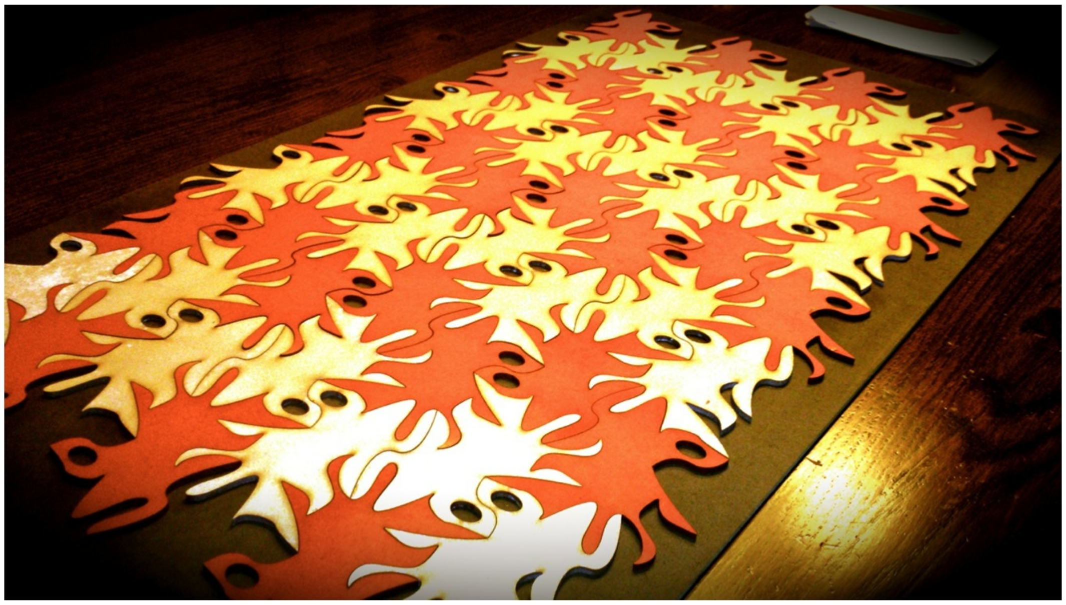


Figure 7 : *Fantasy animals by Matthijs Willems.*

Pepijn Fens had several fantasy figures intertwined (see Figure 8). From a Heesch-Kienzle perspective it is not so complicated, just a TTTT, (Nr1). It is worthwhile mentioning that he took his initial inspiration from company logos. Finally Ardjoen Mangre designed a tessellation of rhinoceroses based on the TGTG type (Figure 9). The animals are cut from wood of different thicknesses, which gives the work an aestehtic quality beyond the pure graphics. (Figure 9)

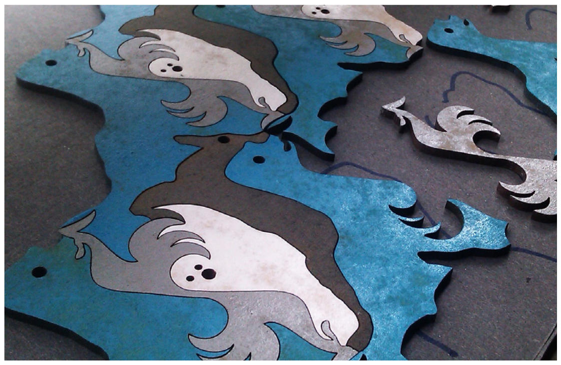


Figure 8 : *Fantasy figures.*

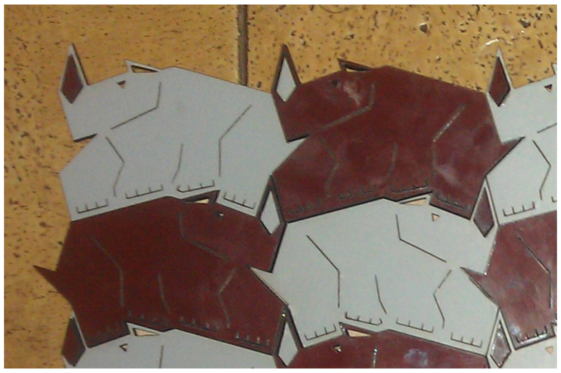


Figure 9 : *Rhinoceros tessellation.*

5 Conclusions

The library was well-received by students and turned out useful, as demonstrated by the student work. We also did a more theoretical study, formally proving that all four classical transformations of figures such as translation, rotation, reflection and glide-reflection can be achieve can be done using turtle commands, but we plan to publish that at a later occasion. The library and its documentation will be made available open-source (see the Bridges CD ROM). Also for readers who prefer their own turtle implementation, the point-sampling tool can be made available.

Acknowledgements: We like to thank Christoph Bartneck, Marina Toeters, Chet Bangaru, Jan Rouvroye and our students for the cooperation in creative programming.

References

[1] Feijs, Loe MG. Geometry and Computation of Houndstooth (Pied-de-poule), In: Robert Bosch, Douglas McKenna, and Reza Sarhangi (Eds.) Proceedings of the 2012 Bridges Conference, Baltimore, Maryland (2012).
[2] Siddiqui, I. Tessellated Floorscape, interior acts of production, siting and participation. IDEA JOURNAL 2010 Interior Ecologies, (www.idea-edu.com) pp.42-53.
[3] Heesch, H. and Kienzle, O. (1963). Flächenschluß; System der Formen lückenlos aneinanderschliessender Flachteile. Berlin,: Springer.
[4] Feijs, L. and Bartneck, C. (2009) Teaching Geometrical Principles to Design Students. Digital Culture & Education, 1(2), 104-115.
[5] M.C. Escher, the graphic work. Taschen 2001.
[6] Doris Schattschneider. M.C. Escher: Visions of Symmetry. W. H. Freeman (1992).
[7] Fejes Tóth, L. (1964). Regular figures. New York,: Macmillan.
[8] Verhoeff, T. 3D Turtle Geometry: Artwork, Theory, Program Equivalence and Symmetry. Int. J. of Arts and Technology, 3(2/3):288319 (2010).
[9] Seymour Papert. Mindstorms: children, computers, and powerful ideas. 2nd edition, 1993, Basic Books.
[10] Christoph Bartneck, Jun Hu, Loe Feijs (teachers), Rick van de Westelaken, Wouter Kersteman, Thomas van Lankveld, Man-shaped figures, Tessellation of Skunks, and Stealth. Fathauer, R. and Selikoff, N. 2012 JMM Art Exhibition Catalog, The Bridges Organization.