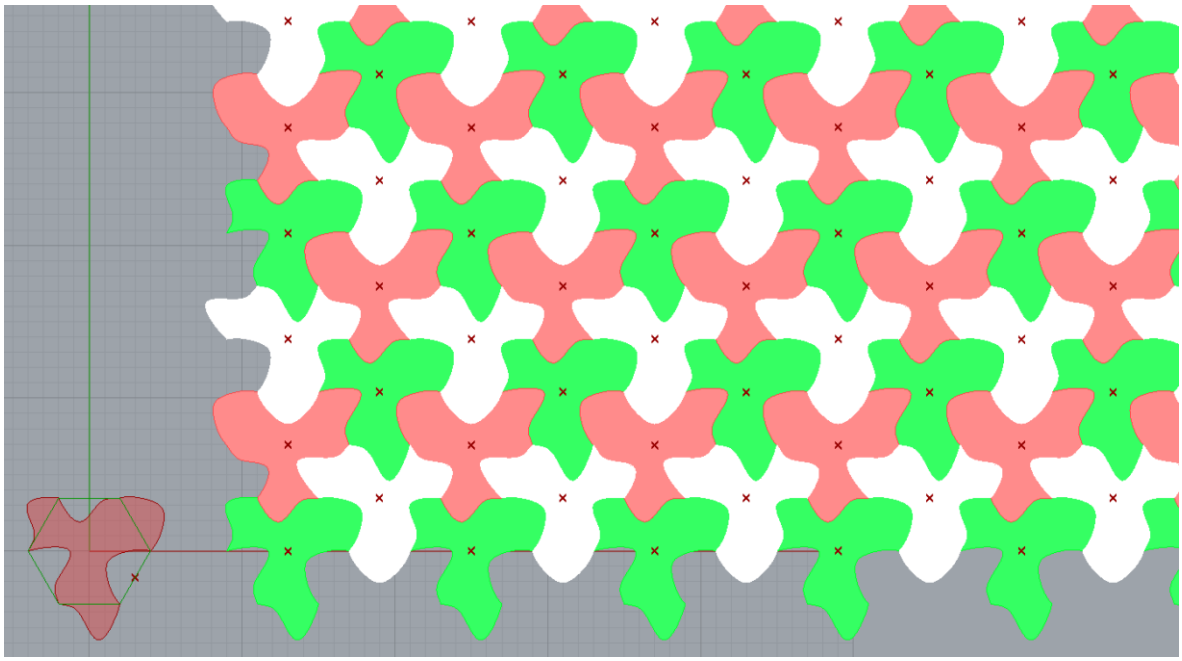
Miniproject 3-TILING IN GRASSHOPPER



What you will learn

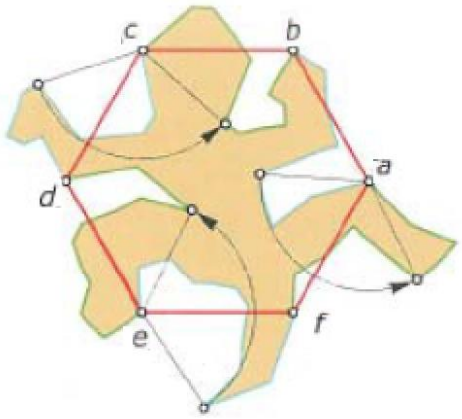
1. combine transformations (rotation and translation)
2. master grasshopper tree data structures management

List of relevant components used

HexGrid
Rotate
Move
Polygon
Eval
Graph Mapper
Cross product

Introduction

The art of designing tiling and patterns has a long history and is therefore well developed.
Planar transformation are an effective tool for positioning objects in the plane. But hey are also very useful in creating regular tessellations and tiling. In this miniproject we learn and apply the basics of designing tiles by implementing the following work by M.C: Escher.



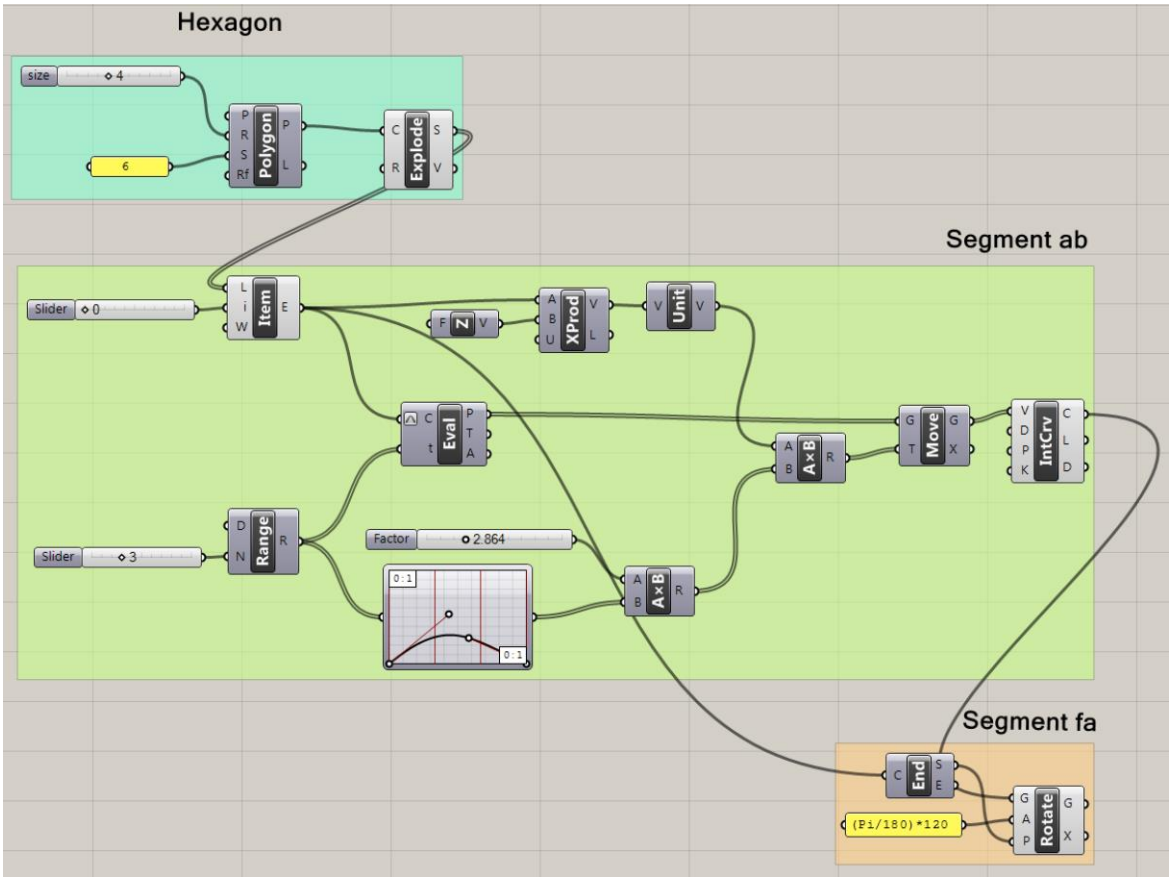
Procedure to create the tile

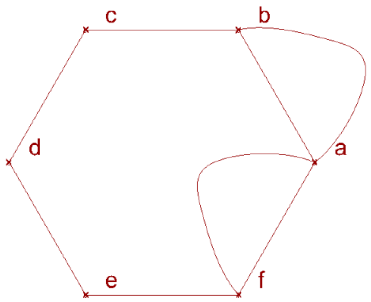
The tile is based on the geometry of three different starting curves, generated along hexagon sides *ab*, *cd*, *ef*. To create the curves for each of the three sides we generate intermediate points, move the points along the direction orthogonal to the side itself (and belonging to the plane where the hexagon lies), and generate a new curve across the new points position. We start from generating the curve along hexagon side *ab*.

1. Use a **Polygon** (Curve>primitive) to create a hexagon and set the hexagon size to a desired value
2. Select side *ab*, by using **explode** (curve > util) and one **list item** component with the appropriate index.
3. Generate equally spaced points on the side *ab* with the **Eval** component. Select *Reparametrize* (right click C input of Eval) to set the domain of the curve from 0 to 1 and use the **Range** component to generate equally spaced numbers between this domain (and consequently equally spaced points on the side *ab*).
4. Determine a *unit direction vector* to be used for the points displacement. The direction vector should be orthogonal to the side *ab* and comprised in the xy plane where the hexagon lies. You can determine the direction vector with the use of cross product between the side itself (when connected to a cross product, grasshopper automatically convert a segment to a unit vector that has the segment direction) and another vector: **which one and why**? The result of the cross product is the direction vector, that you should reset to length 1, with Unit vector (Vector>Vector) to have the *unit direction vector*.
5. Create the displacement vectors for the point on the side *ab* by multiplying the *unit direction vector* with a list of numbers that contains as many numbers as the points on the segment. The values of the

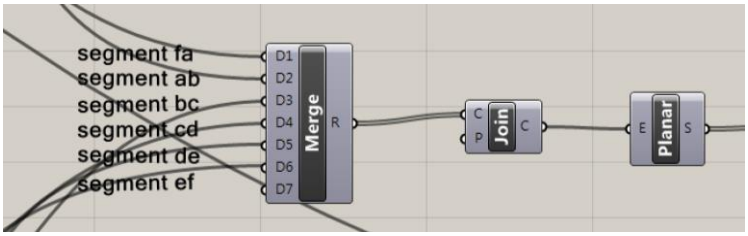
numbers determine the magnitude of displacement vectors, and they can be set by you arbitrarily: a component **Graph mapper** (params>util) can be used to conveniently create and edit a set of numbers with the help of a graphic interface. Hint: use a Bezier graph type (right click>Graph type), and hold the two extremities to a value=0. The component remap a series of numbers according to the graph type, comprised in a domain by default set between 0 and 1. If you reuse the same **Range** component used in step 3 for generating the points through the Eval component, you will generate a list of numbers matching the list of points. You can increase or decrease the numbers values by multiplying them with a factor using a **multiplication** component.

6. Displace the points with the displacement vectors just created in step 5.
7. Use Interpolate curve to generate the new curve geometry comprised by the side end points.
8. To create the segment along the side *fa*, you should rotate this curve by 120 degrees CCW (counterclockwise) around vertex *a*





9. Repeat the procedure with the other two sides *cd* and *ef* . Each one should have a dedicated "graph mapper" component so you can change the geometry of each segment independently from the others.
10. Use a **Merge** (sets>tree) and **join curves** component (Curve>Util) to respectively collect and join in a single component the six curves in a CCW (counter clockwise) or CW (clockwise) order. and a **planar Srf** (surface>freeform) to create a surface for the tile

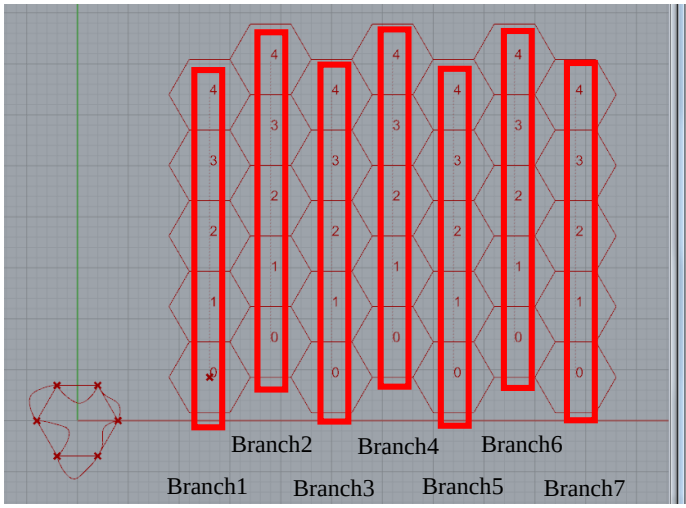


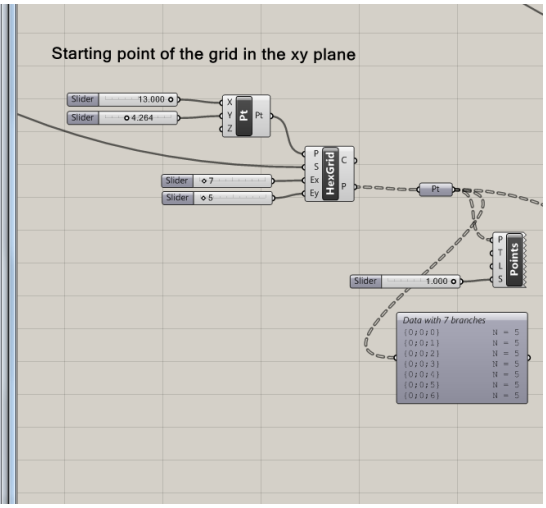
Procedure to create the tiling

11. The tiling is based on a hexagonal pattern. To recreate it, you should create a 2d grid with hexagonal cells, where each point at the hexagons center is the center point of a tile**.** In order to define the grid parametrically, the spacing of the points in the grid should be based on the dimensions of the tile. This will ensure that when changing the tile dimension, the geometric compatibility between adjacent tiles is maintained.

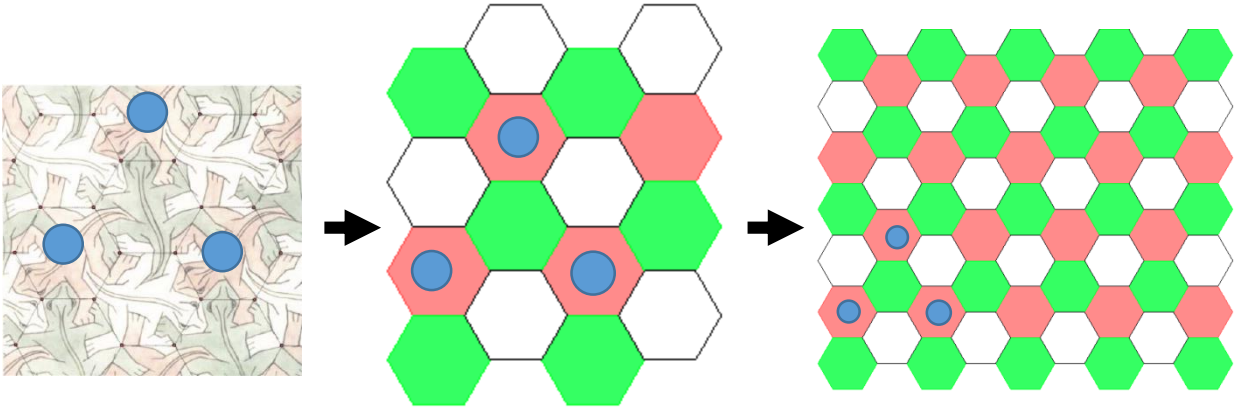
    Use a **Hexgrid** component (vector>Grid>hexagonal) to create the starting grid of points (P output). The points in the P output are organized in branches.

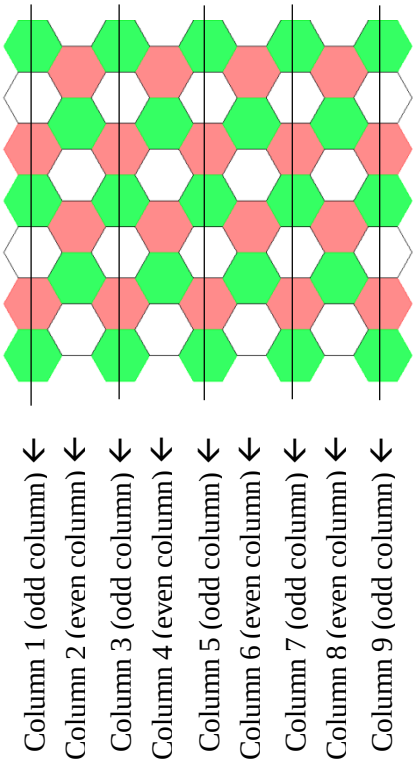
    HINT: Use a starting point in the P input so that the grid does not interfere in the Rhino canvas with the drawing of the tile.





12. By looking at the original tiling, we can recognize the following tile disposition scheme:

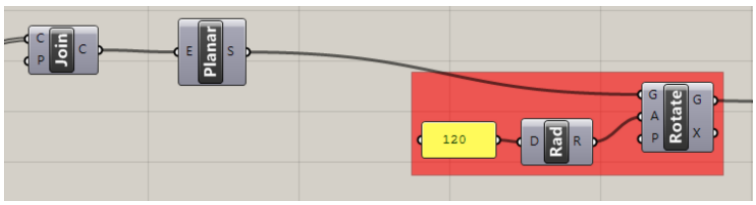




Please note that tiles disposition is regular but varies in odd and even columns, and according to the tile color. Each tile colour represent a different tile rotation: the green tile is not rotated, the red tile is rotated 120 degrees CCW, and the white tile is rotated 240 degrees CCW**.**

Use the scheme as reference for the next steps.

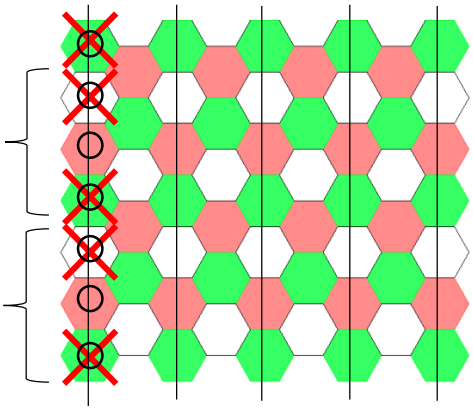
13. We´ll first work on the red tiles in the odd columns. The red tile rotation is 120 degrees CCW (use a **rotate** component)



Isolate, among all the tiles center points (P output of **HexGrid** component) the ones that are the center points of the red tiles in the odd columns with **Cull pattern** component (sets>sequence) and a culling

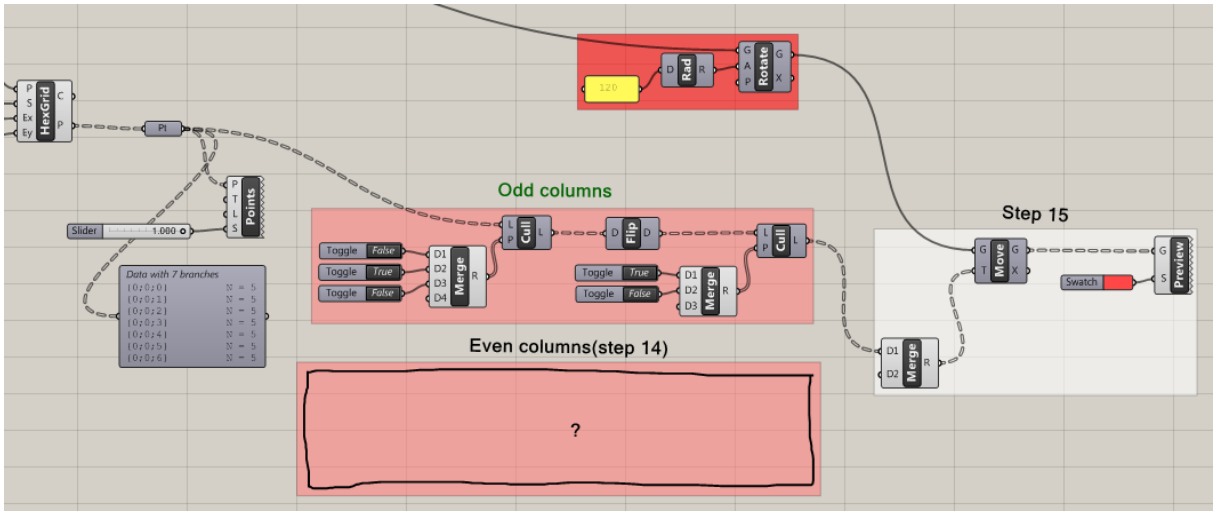
pattern provided with **boolean toggle** components (params>input) collected into a **merge** component (Double clicking on a boolean toggle to change its value from *true* to *false* and vice-versa). **Cull pattern** component removes elements in a list according to the supplied repeating culling pattern, where *false* delete the element, and *true* keeps the element.

In this case we remove the points that are not the center point of a red tile using the repeating pattern "false-true-false".

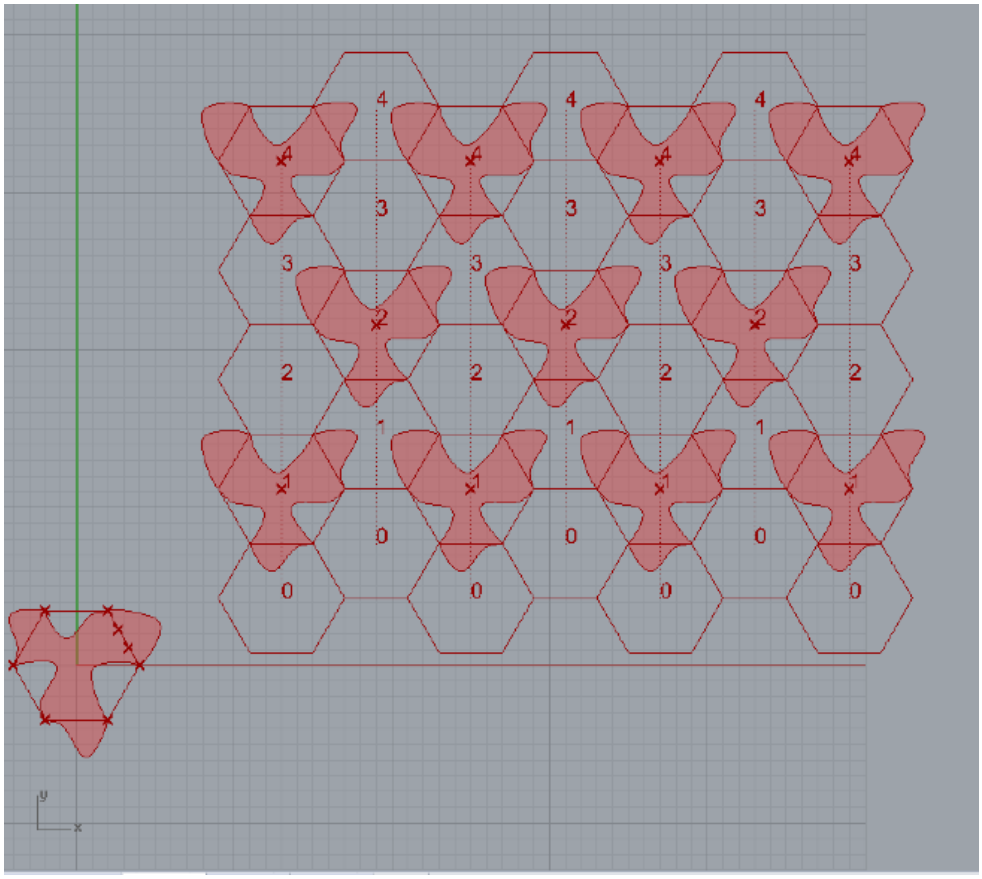


- Because this pattern scheme applies only to odd column, we should eliminate the points sitting in the even columns. To do so flip the tree data structure, and apply a cull pattern "true false".

The points you have left now are the **center points of the red tiles in the odd columns.**



14. **Use a similar procedure but with different cull patterns to isolate the center points of the red tiles in the even columns.**
15. Now you can **merge** the two resulting sets of points (center points of the red tiles in the odd and even columns). In order to place the red tile on these points use a **move** component, with the red tile in the G input and the red tile center point in the T input. Colour the tile with a preview component.



16. **Repeat similar procedures for the green and white tiles in the odd and even columns to obtain the complete tiling.**
17. Can you find alternative methods to create the pattern? In case of affirmative answer pinpoint the advantages and limits of alternative methods. F. ex, could you simplify/reduce the number of components used?
18. **Optional**: can you design/reproduce other tilings and patterns, using a similar grasshopper definition?