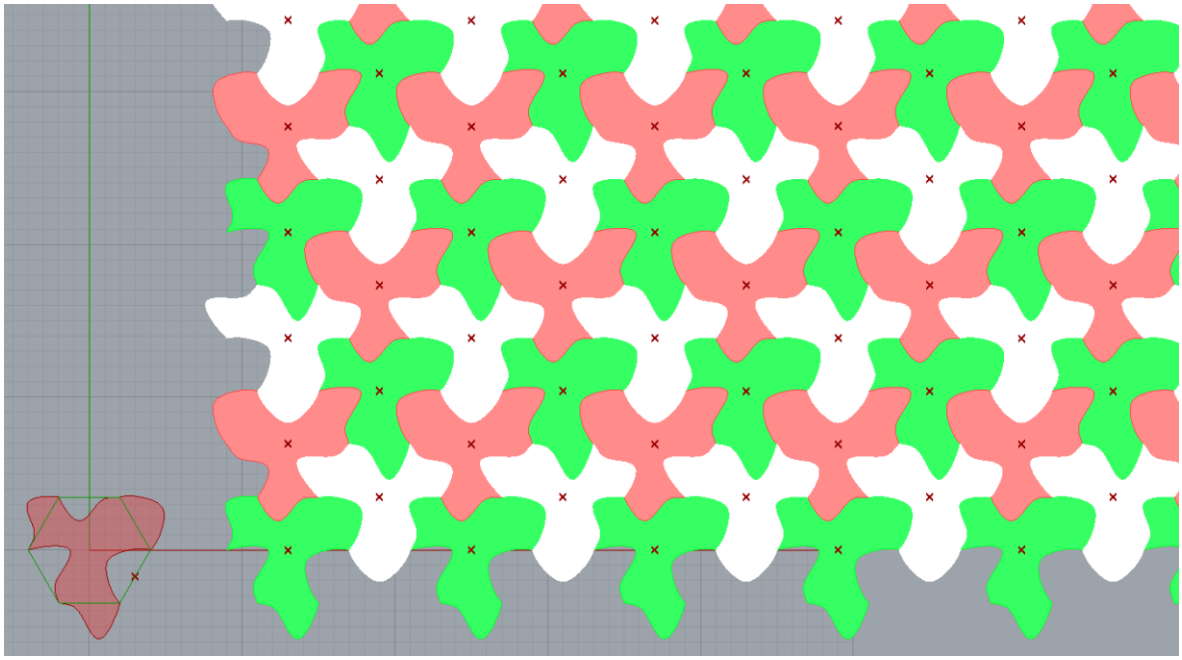
MP4 / TILING IN GRASSHOPPER



What you will learn

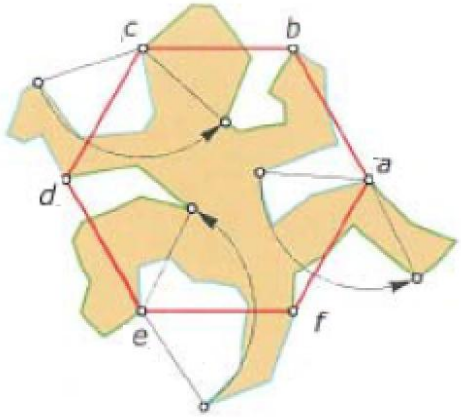
1. combine transformations (rotation and translation)
2. master grasshopper tree data structures management

List of relevant components used

HexGrid
Rotate
Move
Polygon
Eval
Graph Mapper
Cross product

Introduction

The art of designing tilings 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

The tile is based on the geometry of three different starting curves, generated along hexagon sides *ab*, *cd*, *ef*. For creating the curves we use a method similar to the one adopted in MP3: 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*.

-Use a Polygon (**Curve>primitive**) to create a hexagon and set the hexagon size to a desired value

- Select side *ab*, by using **explode** (curve > util) and one **list item** component with the appropriate index.

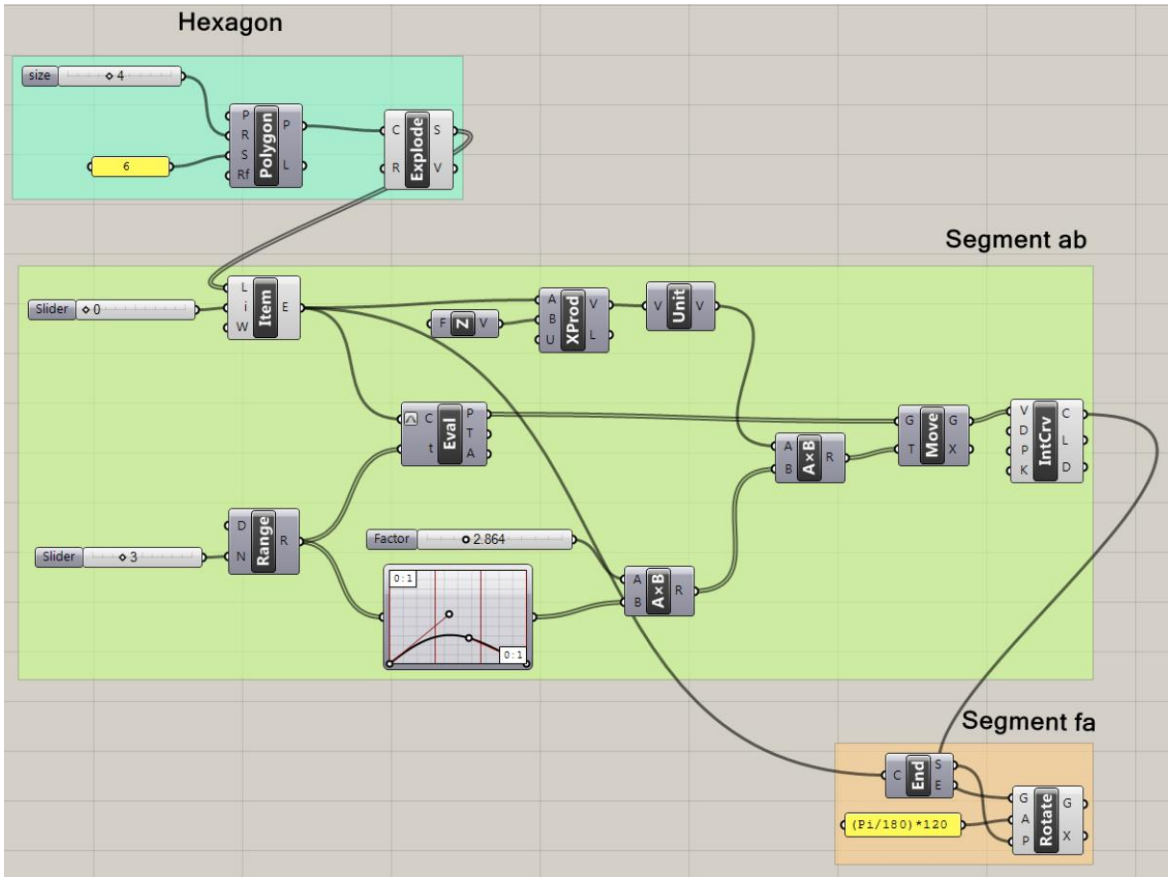
-Generate equally spaced points on the side *ab* with the Eval component. Select Reparametrize (right click C input of Eval) to set the domain of the curve from 0 to 1 (you can look back at detailed Eval explanation in Miniproject 3 documentation), and use the Range component to generate equally spaced numbers between this domain (and consequently equally spaced points on the side *ab*).

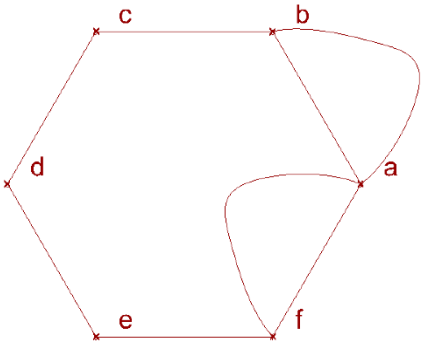
- Determine a *unit direction vector* to be used for points displacement. The direction vector should be orthogonal to the side *ab* and comprised in the xy plane where the hexagon lies. You can determine the direction vector with the use of cross product between the side itself (when connected to a cross product, grasshopper automatically convert a segment to a unit vector that has the segment direction) and another vector: which one? 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*.

- 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. Hints: 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 previously for generating the points through the Eval component, you will generate a list of numbers that matches the list of points. You can increase or decrease the numbers values by multiplying them with a factor using a multiplication component.

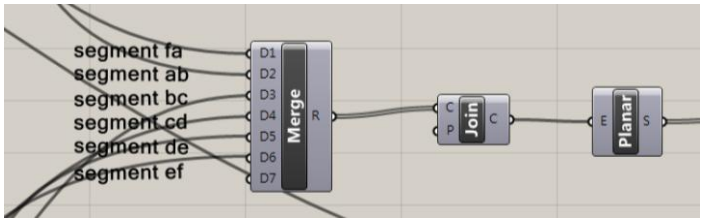
- Displace the points with the displacement vectors just created.
- Use Interpolate curve to generate the new curve geometry comprised by the side end points.
- To create the segment along the side *fa*, you should rotate this curve by 120 degrees CCW (counterclockwise) around vertex *a*





- 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.

-Use a Merge (sets>tree and join curve (Curve>Util) to 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

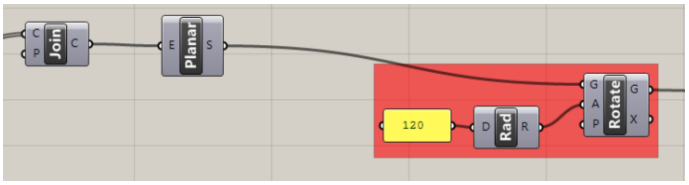


-Create now a hexagonal pattern (vector>Grid), with the hexagons size matching the hexagon used for creating the tile geometry. Use a starting point in the P input so that the grid does not interfere in the Rhino canvas with the drawing of the original tile.

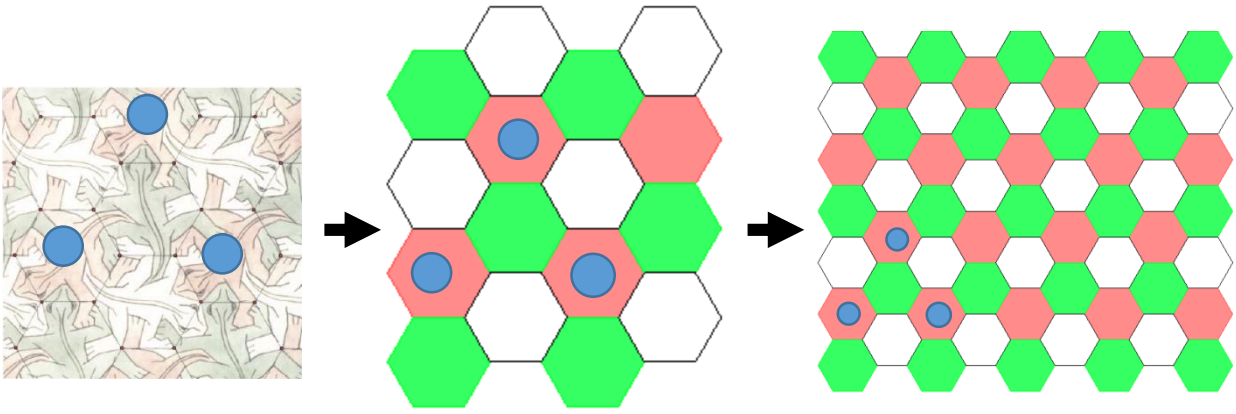
-Now you should "fill" the hexagonal pattern with the tiles according to the pattern scheme. You may notice that the tiles placement follows a scheme where each tile colour represent a different tile rotation. In this case, the green tile is not rotated, the red tile is rotated 120 degrees CCW, and the white tile is rotated 240 degrees CCW. Therefore before placing each tile in the pattern with a translation (move) we have to perform a rotation (except in the case of the green tile).

We start filling the pattern with the red tile.

- The red tile rotation is 120 degrees CCW (use a rotate component)

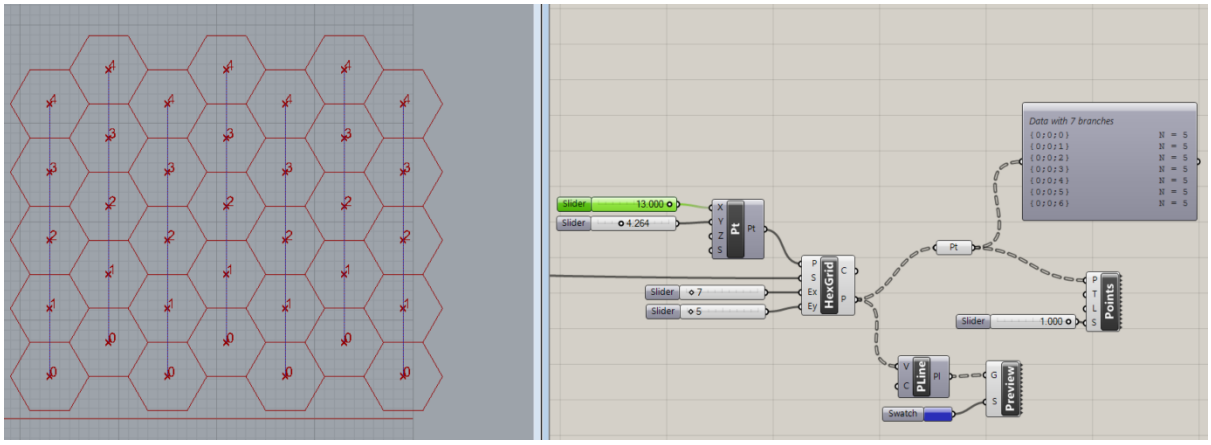


- We have to select, among the centerpoints of the hexagons (P output of HexGrid component) the centerpoints of the hexagons that "host" the red tiles. By looking at the Pattern, we can recognize the patterning structure, highlighted in the scheme below:

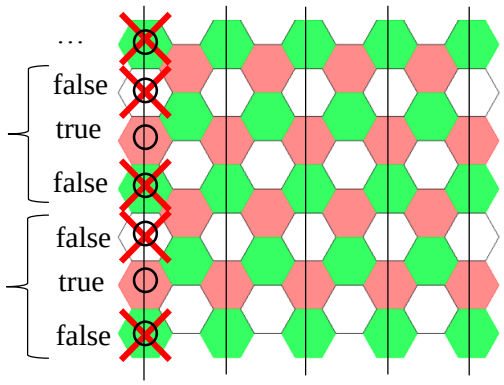


In order to be able to select the correct points we first must know the way points are organized inside the P output of the HexGrid component. With a ParamViewer component and a Point list component we discover that they are organized according to the characteristic "tree data structure" we dealt with also in the exercise GH2 (Parametric wall). **Refer to exercise of lecture GH2 "Parametric wall", introduction to trees data structure paragraph, for refreshing the concepts relatives to "trees data structures".**

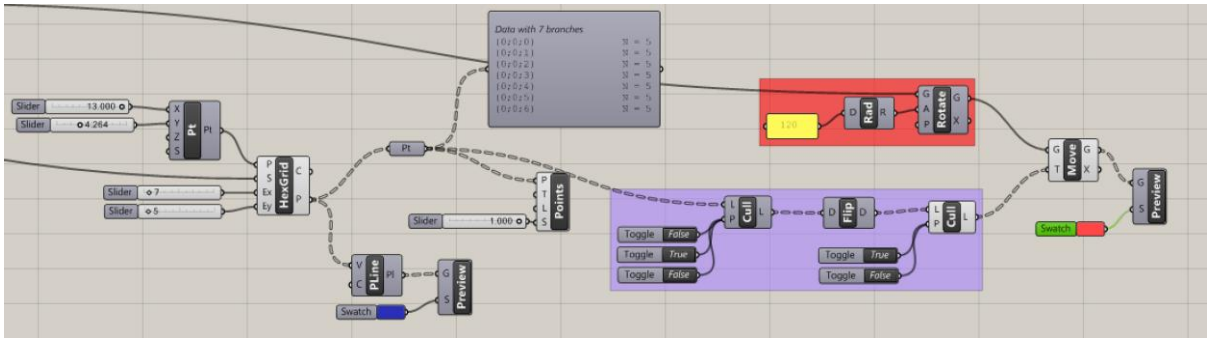
Points are organized into separated "branches", and inside each Branch, in a List. Each list is kept isolated in its branch. For example, if you use a Polyline component with the P centre points as inputs, the points inside each List are connecter together, but not between branches, therefore a polyline is created for each branch, but not "across" branches.



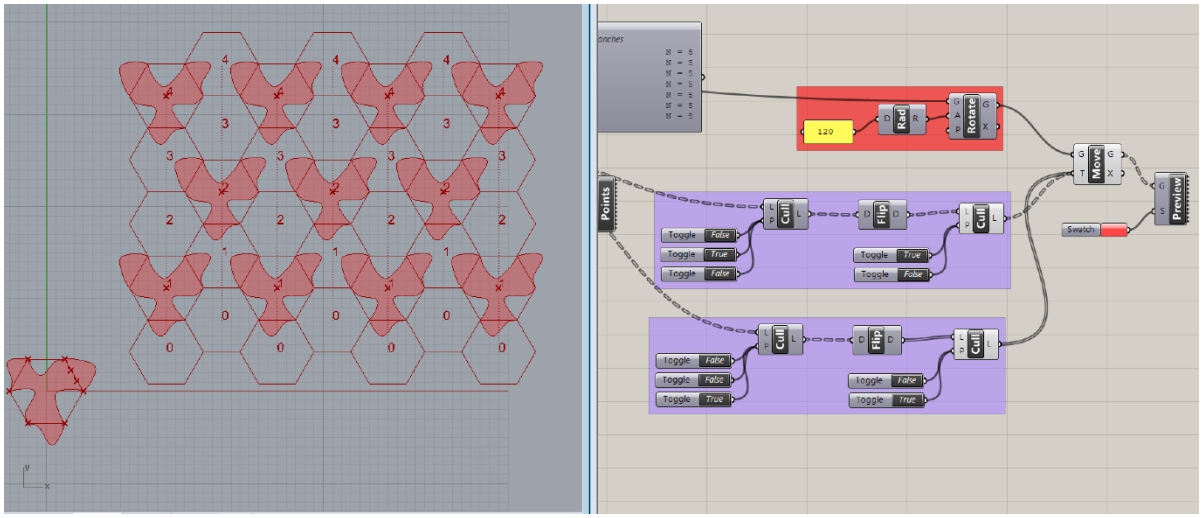
We will take advantage of the tree data structure to conveniently select the points corresponding to the canter point of the red tiles. We use the same method used for the exercise of Lecture 2 (parametric wall), that uses the components "cull pattern", and "flip matrix". "Cull pattern" eliminates points in lists according to a given pattern. Use one or more Boolean toggle (param>input) to conveniently control the cull pattern "*P*" input (rather than right click > set multiple boolean). The Boolean values (true or false) can be changed with a double click, and they are stored inside the Cull pattern component in the order they are connected to the P "culling pattern" input of cull component (hold shift to connect multiple Boolean toggle to P).



-We notice at first that the odd branches have an equal repetition scheme for the three tiles. In particular the centre points that correspond to the red tiles can be selected by using a cull pattern "false, true false". To eliminate the points on the even branches (that follow a different scheme), we "flip" data structure, and then apply a cull pattern "true false".



-Use a similar procedure for placing the red tiles on the even branches by using the appropriate cull patterns.



-Connect the two set of points in the odd and even branches to the same T input of a move component, that moves the red tile, in the input G, to the points. Colour the tile with a preview component.

-Repeat similar procedures for the green and white tiles to obtain the complete tiling.