Miniproject 1 - PARAMETRIC WALL

What you will learn

1. How to handle grasshopper "trees data structure"
2. To parametrically define relations between large set of objects

List of relevant components used

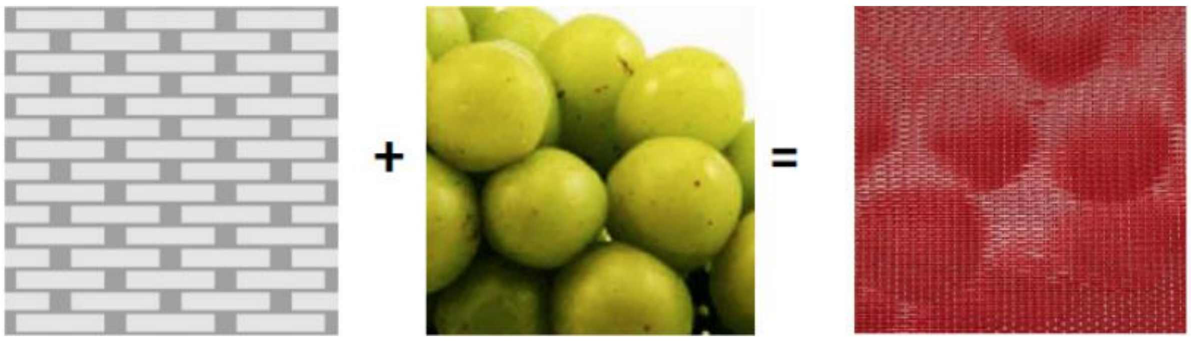
Recgrid
Tree branch
Flip matrix
Cull
Rotate3d
Image Sampler

Introduction

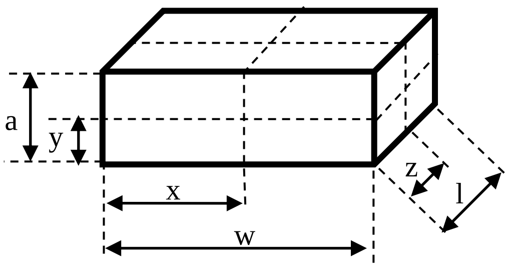
This exercise is inspired by the project for a winery in Gautenbein, Switzerland by Gramazio & Kohler, architects and researchers at ETH Zurich.
http://www.dfab.arch.ethz.ch/web/d/forschung/52.html
The wall is characterized by the “parametric wall”, where bricks position is defined by the geometric relations between bricks dimensions, and their rotation around the vertical axis is parametrically defined to recreate the image of grapes on the winery´s wall.

Goal

In this exercise we are going to replicate the architects’ idea by using a picture of grapes to control the rotation of the bricks. Please note that, because we are working in a parametric environment, the bricks should be set with different size and spacing, therefore the wall can be interpreted and applied at different scales, at both architectural and industrial design level.

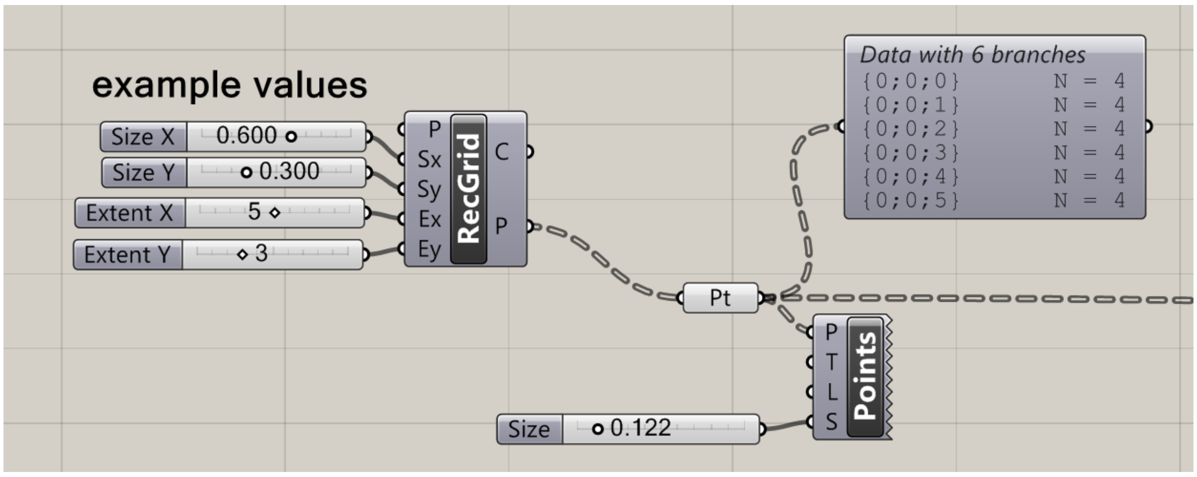


Procedure for creating the parametric wall

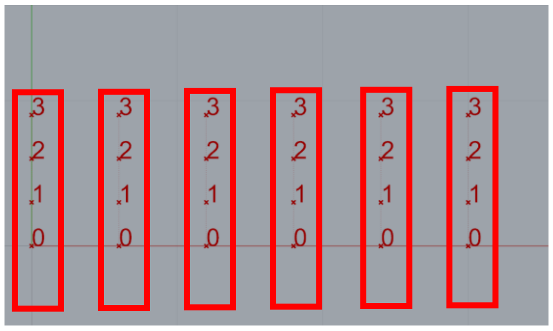


1- The geometry of a brick is created with the component **Center Box (surface> primitive)**. If **w** is the width, **a** is the height and **l** is the length of the brick, **x, y, z** inputs are respectively **w/2**, **a/2** and **l/2.** The B input is the central point of the brick (box)

2- To create a brick wall you should create a grid of points, where **each point is the central point of a brick**. In order to define the wall parametrically, the spacing of the points in the grid in both vertical and horizontal direction should be based on the dimensions of the bricks. This will ensure that when changing the brick dimension, the geometric compatibility between adjacent bricks is maintained. Use a **RecGrid** component to create the starting grid of points. The points in the P output are organized in branches.

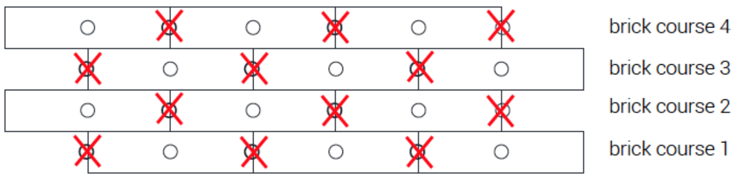


```
Data with 6 branches
{0;0;0}          N = 4
{0;0;1}          N = 4
{0;0;2}          N = 4
{0;0;3}          N = 4
{0;0;4}          N = 4
{0;0;5}          N = 4
```

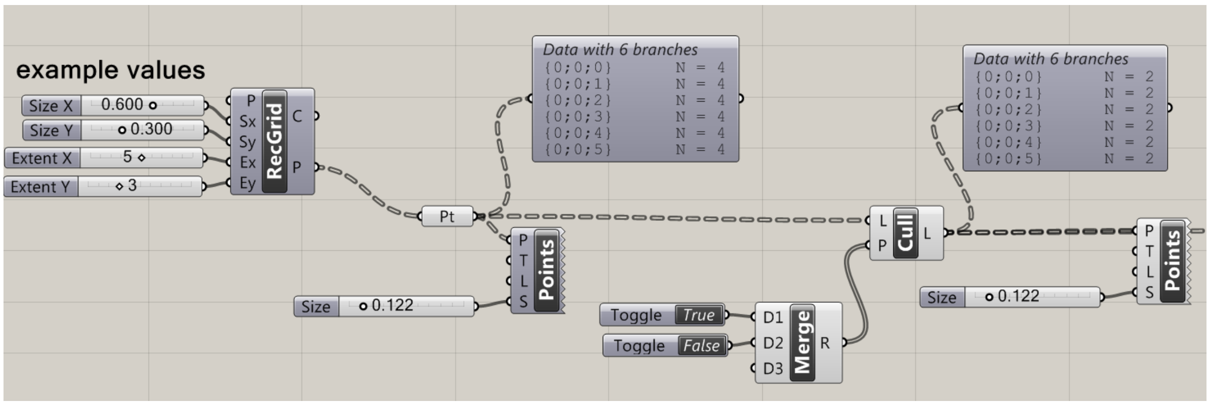


Branch1 Branch2 Branch3 Branch4 Branch5 Branch6

3- Every point in the grid that is not the center point of a brick should be eliminated (red cross in the scheme below). Keep the following scheme as reference for the next step 4

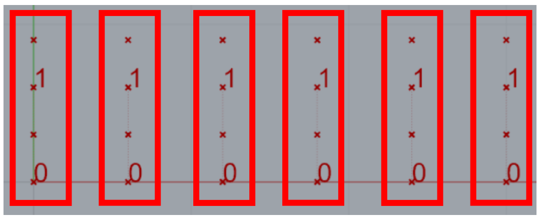


4- Isolate the points of the odd courses with **Cull pattern** component (sets>sequence) and a culling pattern provided with **boolean toggle** components (params>input) collected into a **merge** component (Double clicking on a boolean toggle to change its value from *true* to *false* and vice-versa). **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 use a *true* - *false* pattern to remove all the points sitting on the even courses

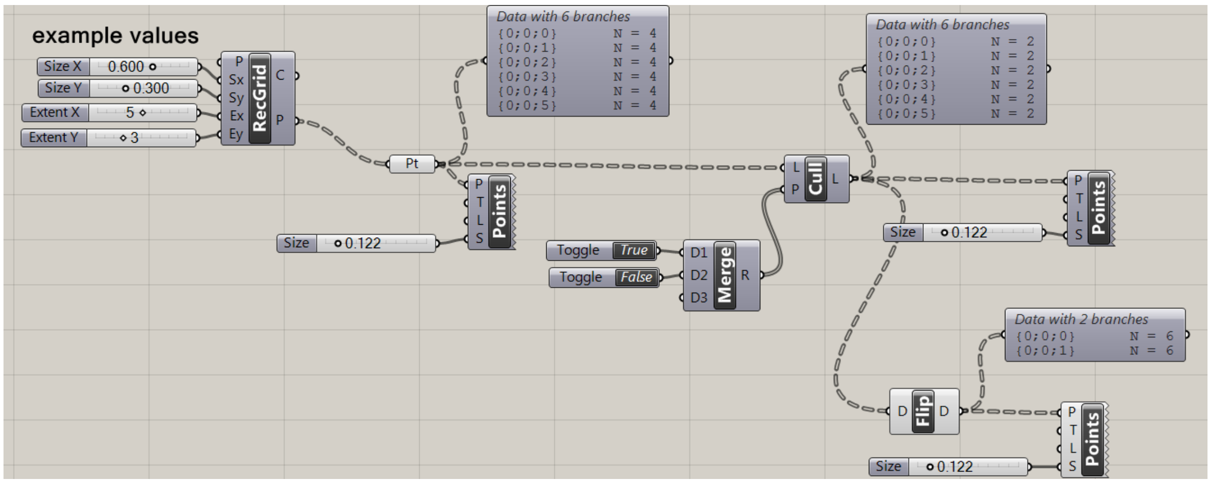


```
Data with 6 branches
{0;0;0}        N = 4
{0;0;1}        N = 4
{0;0;2}        N = 4
{0;0;3}        N = 4
{0;0;4}        N = 4
{0;0;5}        N = 4
```

```
Data with 6 branches
{0;0;0}        N = 2
{0;0;1}        N = 2
{0;0;2}        N = 2
{0;0;3}        N = 2
{0;0;4}        N = 2
{0;0;5}        N = 2
```



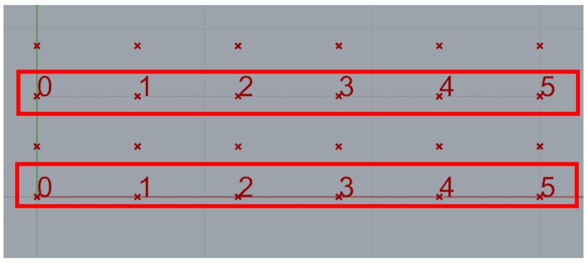
Flip the tree data structure: points will be organized in branches each containing the list of points of an odd course.



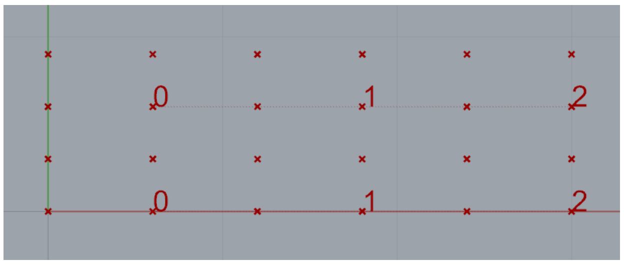
```
Data with 6 branches
{0;0;0}        N = 4
{0;0;1}        N = 4
{0;0;2}        N = 4
{0;0;3}        N = 4
{0;0;4}        N = 4
{0;0;5}        N = 4
```

```
Data with 6 branches
{0;0;0}        N = 2
{0;0;1}        N = 2
{0;0;2}        N = 2
{0;0;3}        N = 2
{0;0;4}        N = 2
{0;0;5}        N = 2
```

```
Data with 2 branches
{0;0;0}        N = 6
{0;0;1}        N = 6
```

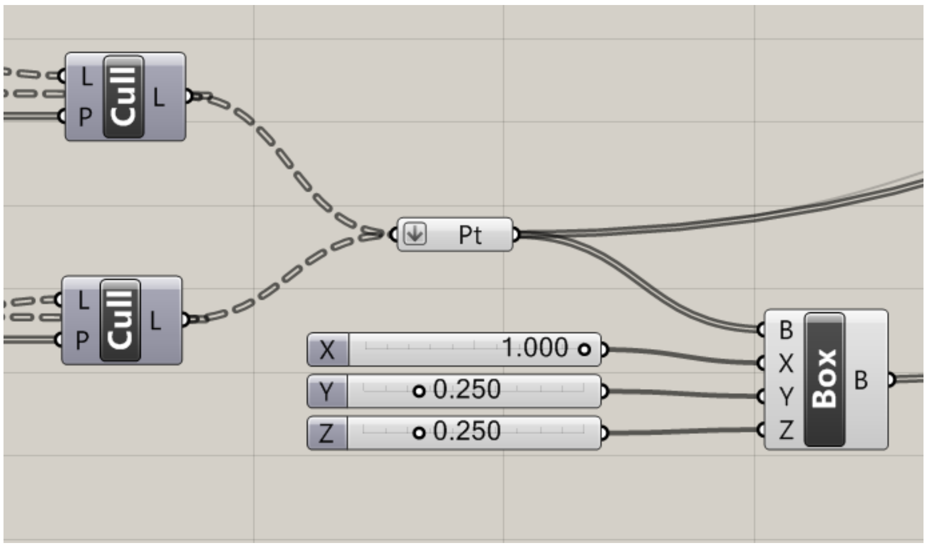


Now apply the appropriate culling pattern using as a reference the image in step 3 to eliminate every second point in the odd courses. The points you should have left now are the **center points of the bricks in the odd courses.**

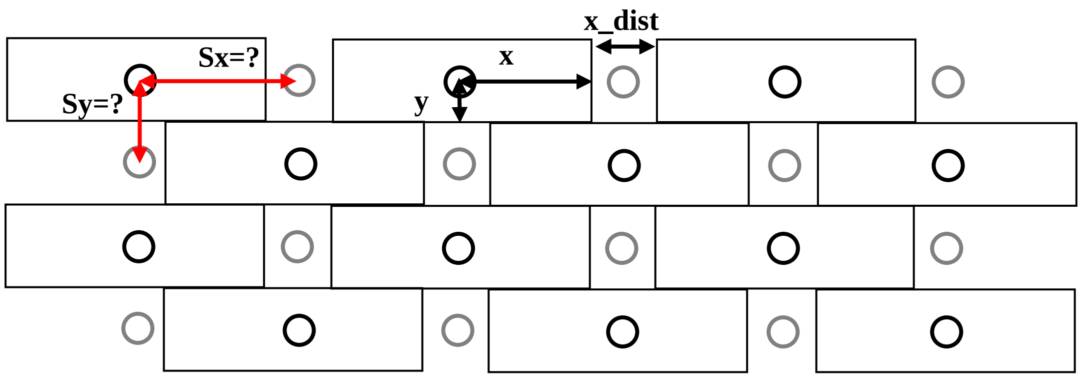


5- **Repeat steps similar to steps 4, but with different culling patterns, to create the center points of the bricks on the even courses.**

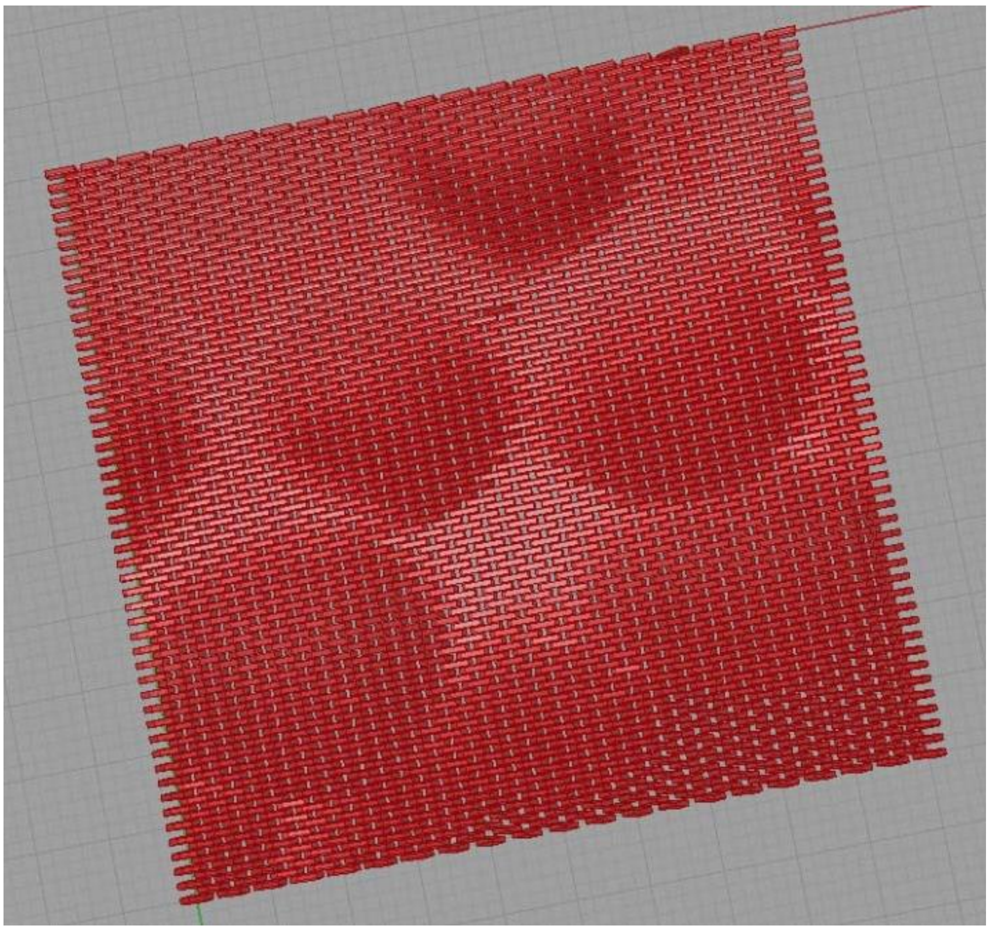
6- Now you can merge the two resulting sets of points (odd and even courses center points) in a **point** component (params>geometry) and flatten the list (right click on the point component input and select flatten) to eliminate the branches and have the points collected in a single list. Connect the points to the box component created in step 1 to create a brick in each point

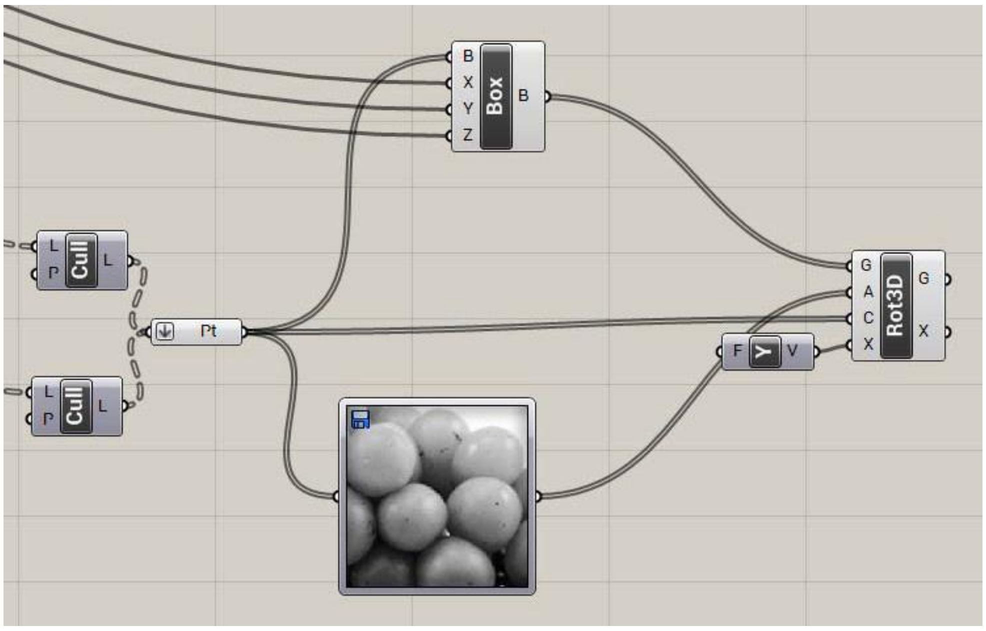


7- Define the correct spacing of the grid in the **RecGrid** component defined in step 2 in the *Sx* and *Sy* input (replace the example values) to ensure that it is based on the brick dimensions *x* and *y* and on the desired spacing *x_dist* (an additional slider)



8- Apply a rotation around the vertical axis of each brick, using a **rotate 3d** component (transform>euclidean), setting the boxes as geometry, a **unit y** vector as rotation axis, and the brick center points as center of rotation. Each brick rotation angle value will be inherited from an **image sampler** (params>special) component, which evaluates various characteristics of an image (saturation, brightness, etc..) at the input position and outputs the correspondent value (double click on it for options). If we want the image to fill the entire wall, the dimension of the wall should match the domain of the image (double click and enter options to change its value): increase the number of bricks and/or adjust the domain range to have the images filling the entire wall.





9- Can you find alternative methods to create the brick wall? In acse of affermative answer pinpoint the advantages and limits of alternative methods.