Penrose Pattern Generator Documentation

Many pattern generators use the deflation method to generate the pattern. I wanted one, where the tiles are placed one after another like you would do at home…

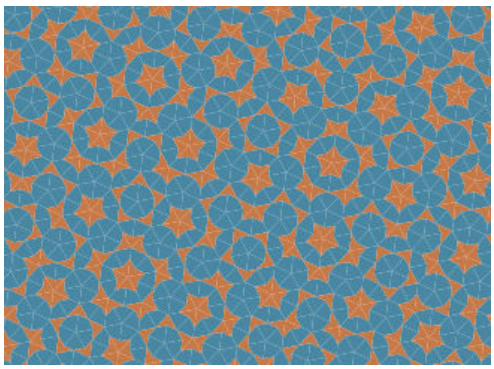
Please note, this documentation is not standalone, but it's for the better and faster understanding for the projects source code.

The Foundations

This Pattern Generator uses the P2 tileing, the **Darts** and **Kites**.

Kite  Dart 

These tiles have some placement rules which must be followed for a continuous pattern to achieve.

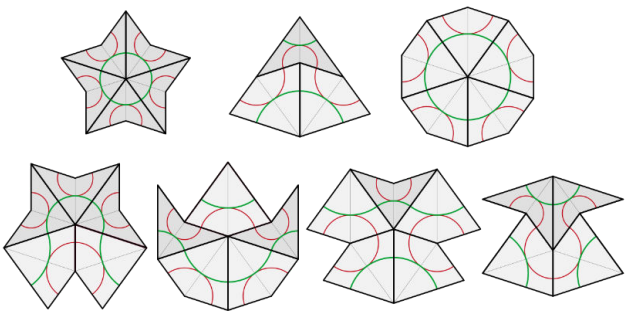


Please watch first the next video: ***https://www.youtube.com/watch?v=48sCx-wBs34***

and read as next the documentation on the following link (I's a little bit deeper and scientific but I think it's necessary): ***https://physics.princeton.edu//~steinh/growthQC.pdf***

or the same document in my GitHub Repo: ***https://github.com/btcreator/PenroseTiles...growthQC.pdf***

Before I start, please consider the next picture. These are the seven possible figures how the tiles can be attached to each other. I call it **vertexRule**.



Source: Wikipedia/Penrose tiling

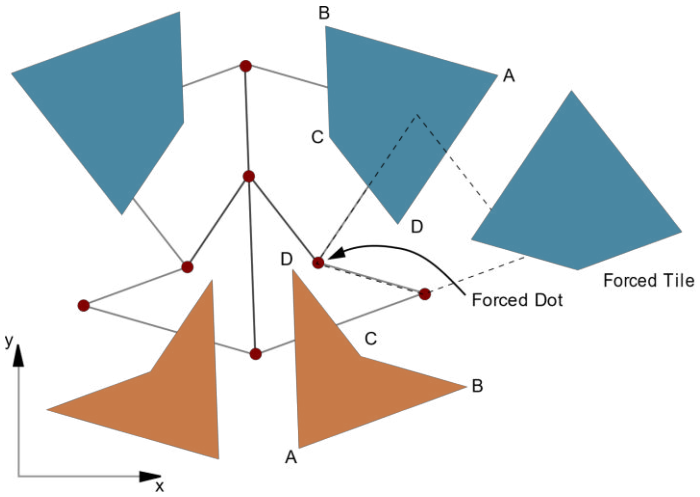


Figure 1

Ok, now. How does the logic work? Each Tile has four **points** A,B,C,D and each point is attached to a **Dot** (red dots on the picture *Figure 1*).

There are just the seven possibilities how to attach tiles to each other and that's why it comes most of the time to a constellation where tiles are forced, which and how one should be attached to a Dot.

Because of this, there are dots, which are **restricted** / forced and **open**. Open dots are all dots, where a tile can be attached (i.e. occupation around a dot is lower than 360 degree). With other words, open dots are all the dots on the edge of the pattern and yes, restricted dots are open dots too.
Now we must always attach tiles to restricted dots first, till there are no more left. Until that point, we have no other choice how to place the tiles. We are forced to place it after rules. When there are no more restricted dots, just open ones, we can choose **randomly** which tile and where to place…

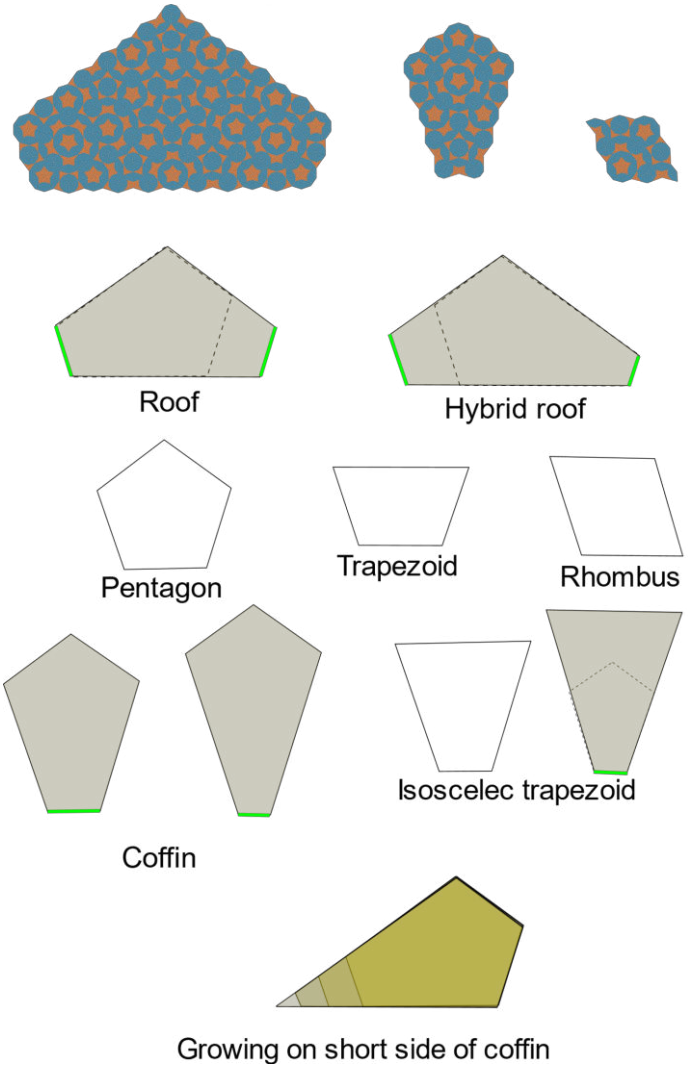


Figure 2

Actually, that's just halfway true, because there is another rule.
I call it **verticesRule.**
When we generate a pattern till no more restricted dots are left, we get some geometrical shapes (dead surfaces *Figure 2*).

To these shapes it's allowed to place a tile randomly, except on the shortest side of the coffin shapes and all other shapes, where the shortest side (of the coffin) is on the edge of the shape (yellow shapes). Aua, that sentence hurts.
Shortly, no random placement on the green lines.
On these sides we are forced to place a tile so, that the next shape must be a coffin too (see on the bottom of the picture). On other sides we can randomly place a tile.

There is one more thing left to know about. The worm (*Figure 3*)

The worm is a "line" of tiles which connects the shapes together. Why it's important? Because of the **verticesRule** from above. When we want place a tile on the "green line" of the coffin shapes, this worm would decide which tile on which place must be placed.
Like I wrote before, the growing on the short side of coffin must be made so, that the next shape is a coffin shape. Better said, for the growth it is not allowed growing over the long sides (*Figure 4*). When the worm is flipped / wrong placed, the Dot on the „**corner**" would be restricted (forced) dot and the growing would be able to go over the long side.

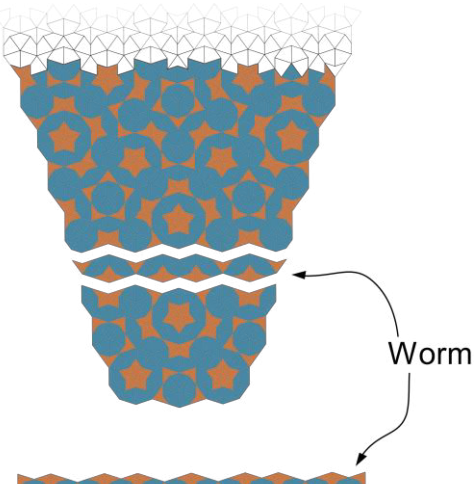


Figure 3

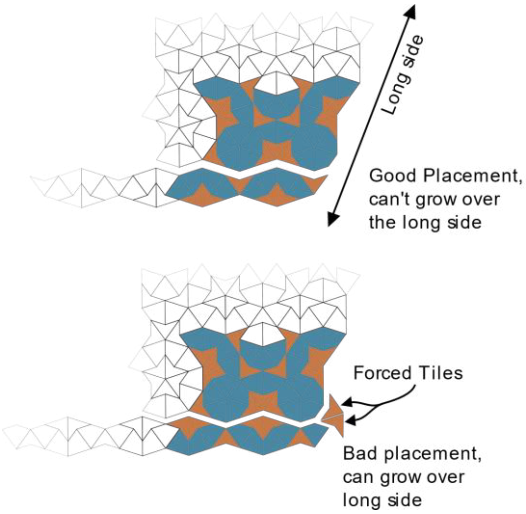


Figure 4

Anywhere later in the pattern, placement error happens, which doesn't follow the rules. (*Figure 5*)

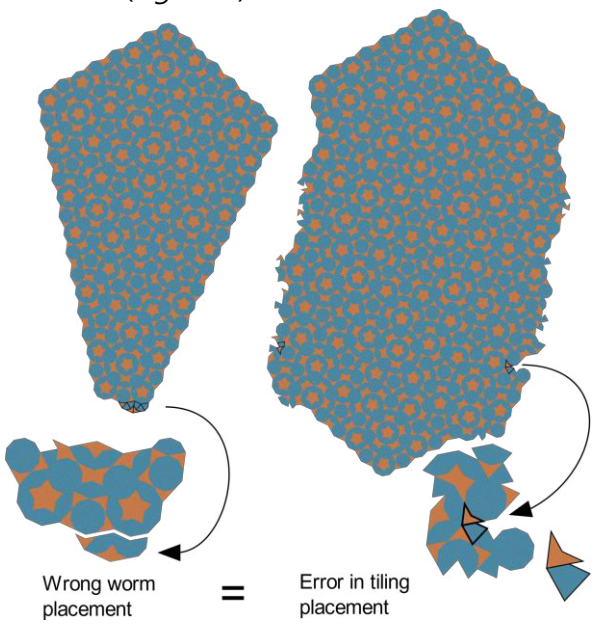


Figure 5

That's it. Now, nothing's left, just implement it to a programming language…for example in javascript.

Program Implementation

class Dart class Kite	class Dot
name rotation coord decor dots	occupy borderPermission #id #coord #totalDegree #nextPossTiles

Kites, Darts and **Dots** are classes to generate the objects and track the properties of them. Dots are collected in four Objects / Arrays by the **dotManager** module (dot holders):

- **allDots** – collection of all Dots
- **restrictedDots** – collection of restricted/forced dots
- **openDots** – collection of all Dots where a tile can be attached
- **inviewOpenDots** – collection of open dots, where tiles attached to it can somehow reach inside of the view area. (dots in the viewport and radius – see bellow the Issues section)

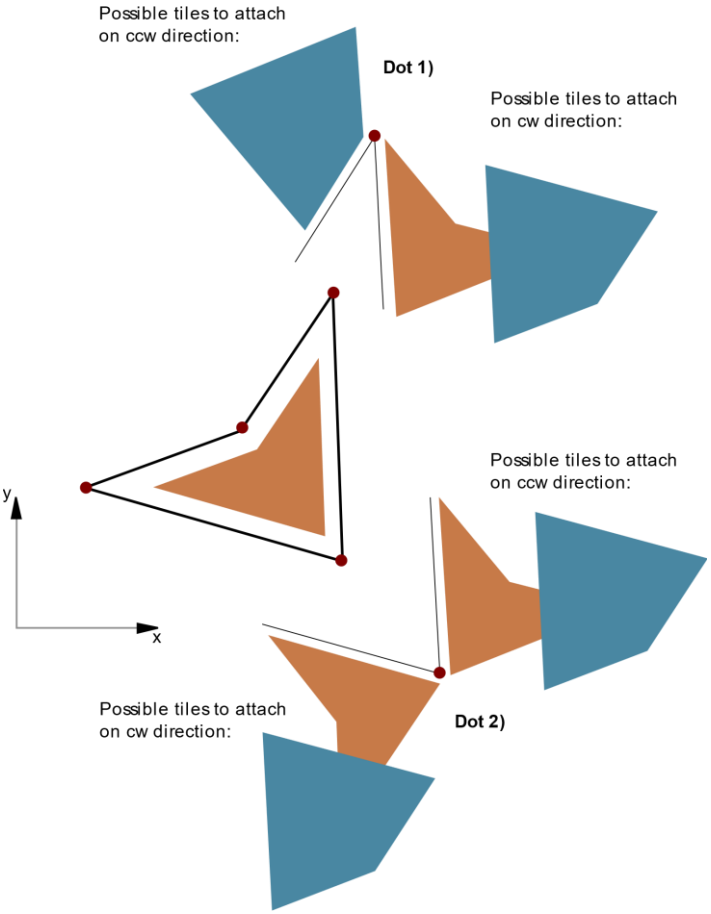


Figure 6

So on the first run, we generate a random tile with a random placement in the view area. For now we assume that is a Dart (*Figure 6*). We generate for this darts each point a new **Dot**. Each Dot object tracks his occupation of tiles (**occupy**), and Tiles tracks his point occupation of **dots**. After that, we look for possibilities for each Dot of the Dart, what can be attached on the left and right sides, or better said, on the clockwise (**cw**) and counterclockwise (**ccw**) direction. This decides that the dot is a **restricted** or just **open** one.

On the picture, the Dot 1) would be a restricted dot, because of the one possibility on the ccw direction. On the other hand, the Dot 2) is just an open dot, which means, we could randomly choose which tile we want to attach to it. But stop, the first rule says, that the restricted dots must be used up first!

After we placed the first tile, we can begin with the **mainLoop** till the Array of **restrictedDots** would be empty.
But not so fast. Back to our first tile. Now we take the next restricted dot, Dot 1) (*Figure 6*) and look on which side (cw / ccw) comes the forced tile, what tile (kite /dart) and with which point connects (A,B,C,D). Then we have a **blueprint** of the next tile (Kite, point C, ccw). Then we let the **tileManager** module create the new Kite. The tileManager does not do too much, it just creates a new tile, rotates it and moves it to the right position (gives just the right coordinates for the tile **coord** property).
Now we pass the new Kite to the **dotManager**, which like before, generates a new Dot for each point, but first of all look for presents in the **allDots.** When no dot exists on the same coordinates, a new one would be created.
Then save the dot to the tile **dots** property and add the tile to the dot **occupy** property (*Figure 7*).

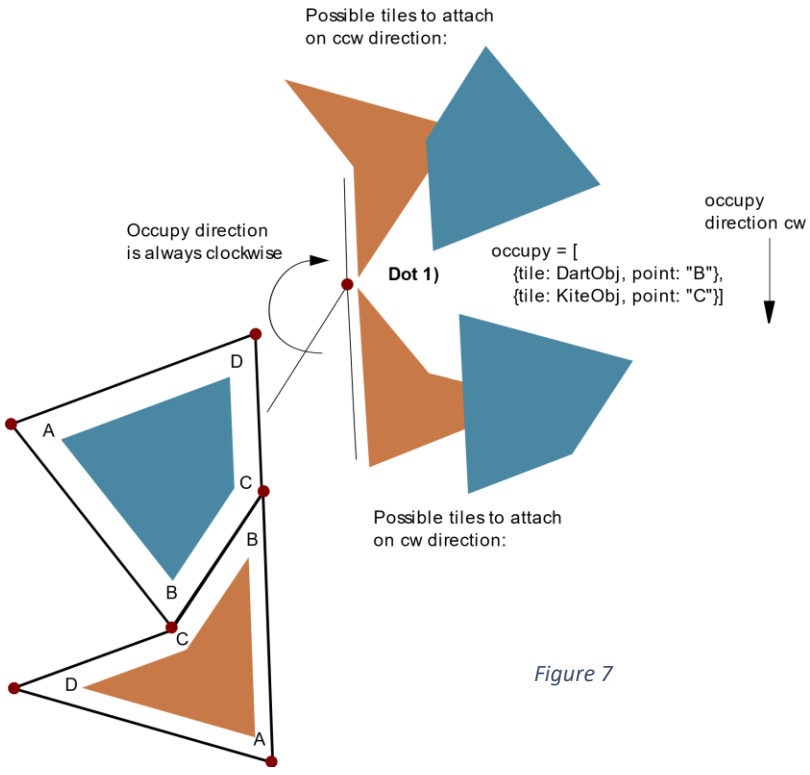


Figure 7

Here something important happens, that we look at the dot for wether it is in the view field or not. If yes, we push it to **inviewOpenDots** too. The same happens with the tile. When its in the view field, we push it to **visibeTiles** (an Array in the **model** module). We take the next restricted dot, and the whole loop goes around…
…till we reach a dead surface (a geomatrical shape) and no restricted dots are left. Now we randomly choose one dot from the **inviewOpenDots** Array and a random tile to place. That placement forces a new restricted dots. With that, the whole process continues to place one tile after another until the next dead surface…
When the inviewOpenDots becomes empty, the whole loop stops and we get a penrose pattern (an SVG) rendered on the screen from the **visibleTiles**.
That's nice and easy. But when you ask what is with the second rule? What if when our randomly chosen dot lies on the short side of a coffin shape (on green line from *Figure 2*)?

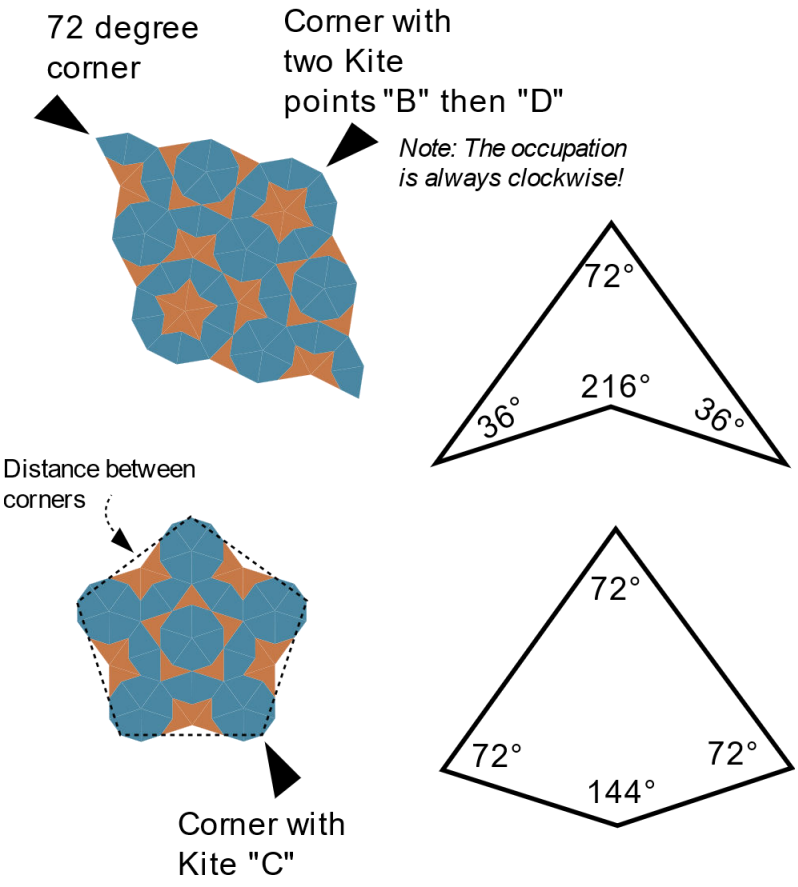


Figure 8

After we randomly choose a dot, we must figure out what shape we have and when it's a coffin(like), then look if the dot lies on a short side. The corners play an essential role to find out which shape we have. The corner can be a dot with a single Kite tile with point "C", with two Kites point "B" and "D" (in this sequence) or dot with **totalDegree** of 72. Just these three variations can be corners. (*Figure 8*)

After that, we have four or five corners. The number depends on the shape. (look back at *Figure 2*.). From the corner coordinates we can now calculate the side lengths of the shape and from that we can figure out which shape we have (*for the calculation see the sourcecode – dotManager –> verticesRuleReferee()*).
When it's one of the coffin(like) shapes, then we look, if our random dot is one from the dots of the short side or not.

This happens as next in the function **checkDotPresenceOnShortSides**. Imagine a square around the short side of the coffin(like) shape. When the dot is inside the square, the dot lies on the short side. The square around the short side is outsized (offset) just so much, that undesirable dots from the neighbour sides can never fall inside of it, just the dots from the short side are inside even when it lies horizontally or vertically (*Figure 9*). When the dot lies on the short side, we just need to decide which tile must be placed for right worm placement.

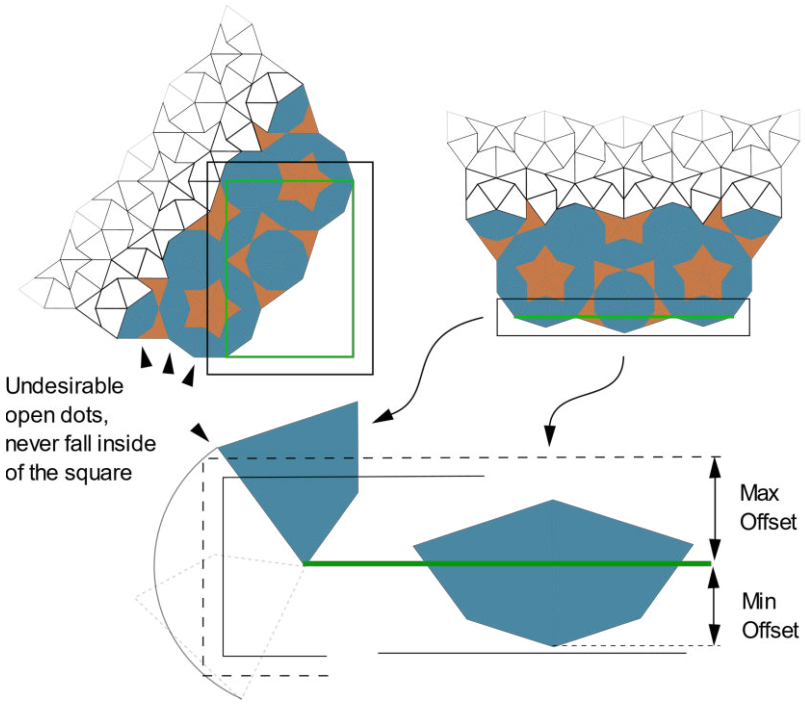


Figure 9

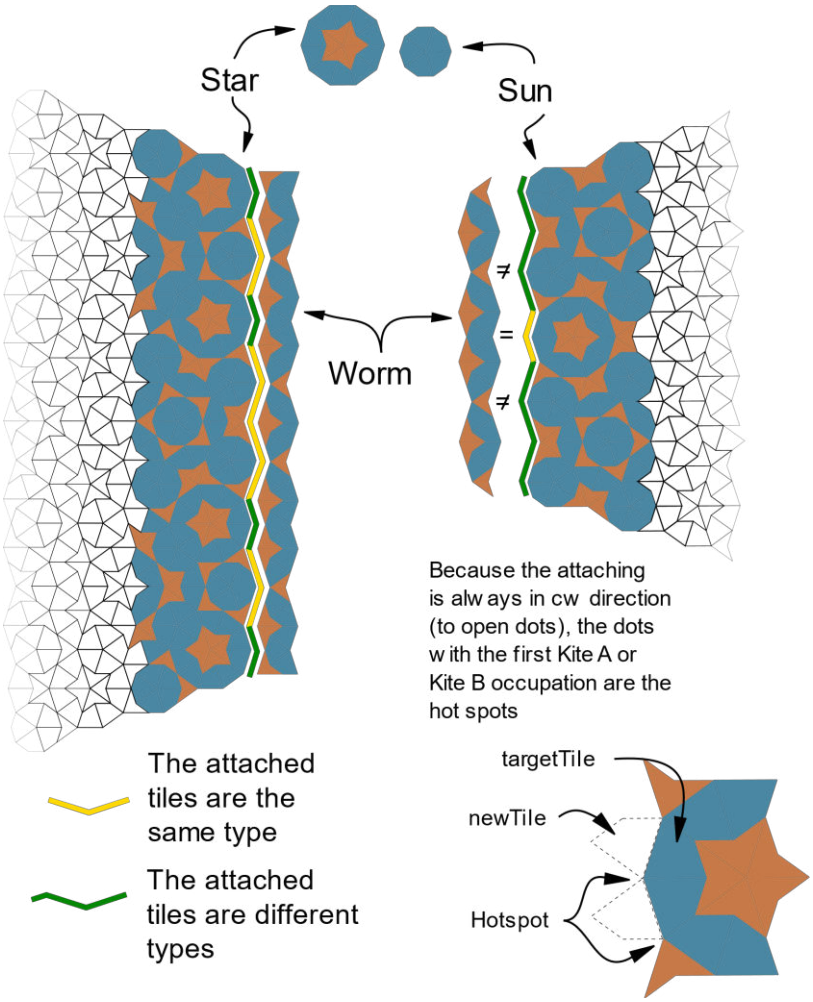


Figure 10

On the first line, the corner formation, the "star" and the "sun" are important as how the worm would be attached. On the second, you can see in the picture (*Figure 10*), there are always just two "hotspots" which we need to aware of, the Kite "A" and "B" points. So, when the corner is sun then the **newTile** on the hotspot dots must be the same as **targetTile**. On other dots must be a different one. With the star on the corner is the opposite true.

This is the core logic. But I will show you some issues and solutions related with the code.

Issues

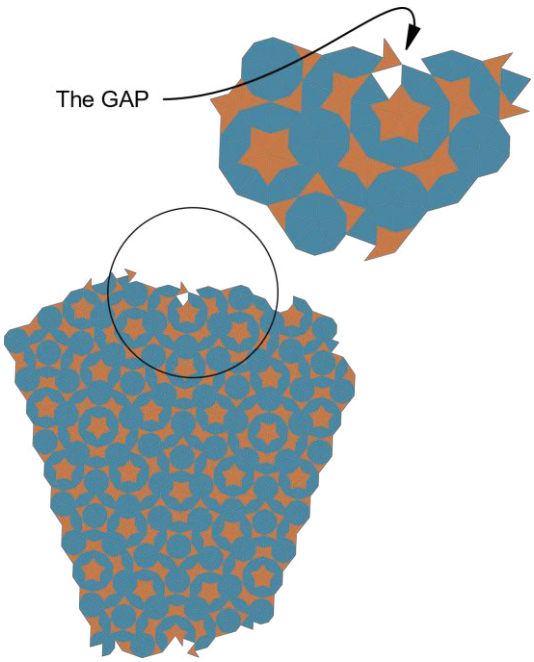


Figure 11

The "GAP"

When you just look at the picture, you know what I mean. It can happen that a tile would be placed so unluckily that no consistent edge would be formed, and one tile bridges over the other so, that an open tile spot arises. In that situation, when the **gapWatchdog** signals**,** we must undo everything related with the newly placed tile and move the **restrictedDot** (to which was the tile attached initially) to the end of the pile. /It happens, ca. 1 time out of 2000 generated tiles/

The "Corner cut"

How do we know, which tile should be rendered and which not? Actulally it's not a big brainer, when at least one dot of the tile is in the **Viewport**, then the whole Tile must be rendered. When no one is, the whole tile is outside and would be not rendered.

But when two dots cut the corner of the viewport, the Tile on the corner is not rendered too, despite it should be. (*Figure 12*) The thing is, there is a lighter way when we just render the tiles on an oversized viewport, but I wanted not pollute the DOM and the SVG (...and then where would be the challenge, by the way ;)).

In the **borderControl** function we look where the dot is. It can be outside of the **renderZone** (then the tile is not rendered), in the **viewport** (the tile must be renderd), or in in the **overlay**. We don't need to render all in the ovelay, just those, which are at the corners, inside of the overlay radius. The distance of overlay is the two most distant points of a tile, which is the BD section (see *Figure 13, I hope it helps more to understand*).

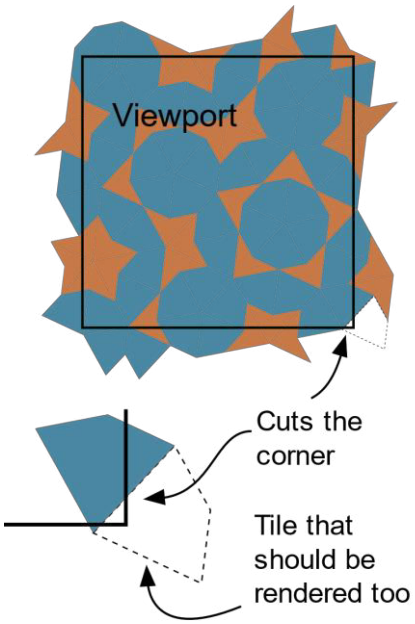


Figure 12

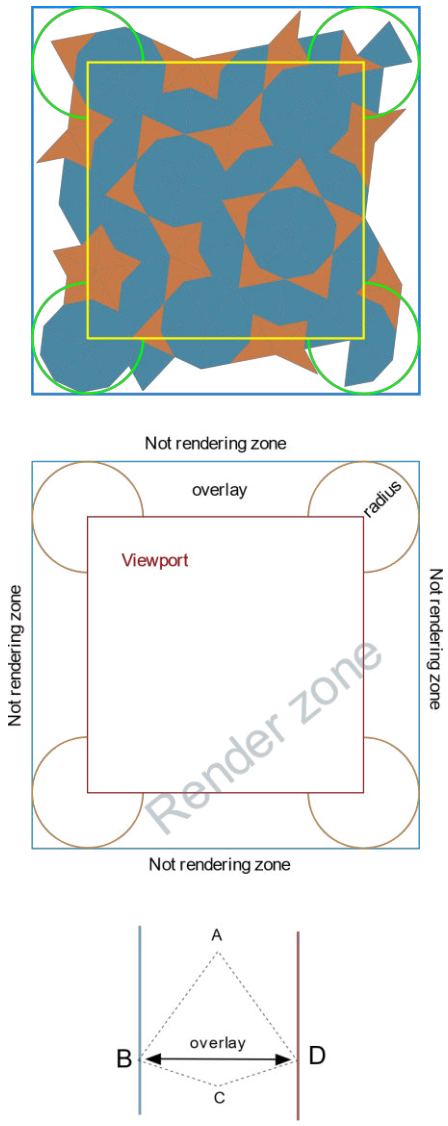


Figure 13

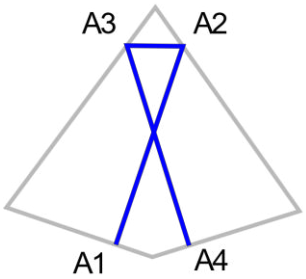
So each dot gets a **borderPermission** with

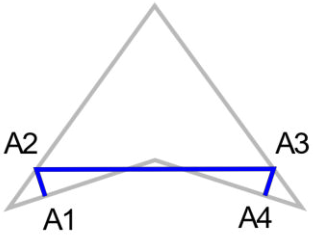
- 1 – true -> the dot is inside of **viewport** (permission for rendering would be granted to all tiles that are attached to it)
- 0 – false -> the dot is outside of the **renderZone** (permisson would be denied for all tiles that are attached to it)
- -1 – weak true -> the dot is inside of the radius (permission for rendering depends on and would be granted when **all** other dots of the tile have permission of -1 /or 1/ - in that case it mean, the whole tile is inside of the radius or reach inside the viewport).
- -2 – weak false -> the dot is in the **overlay** but not in the radius (permission for rendering depends on and would be granted when **minimum one dot** of the tile has permission of 1, in other cases would be denied – when denied, it means, the whole tile lies partially or whole outside of radius, inside of overlay, and not reaching inside the viewport).

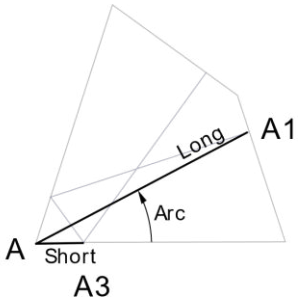
So are all tiles rendered, which somehow reaches inside of the viewport.

Decorations

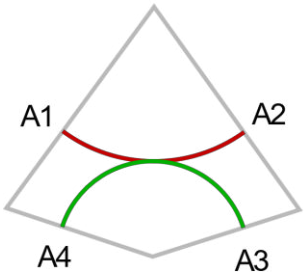
I set 2 decoration styles to add to the pattern, the **Amman** lines/bars or the **Arc**.
The Amman lines looks as next (first two pictures):

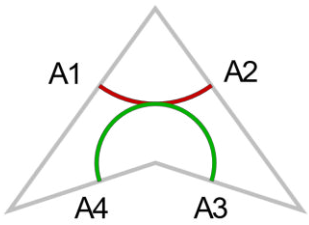


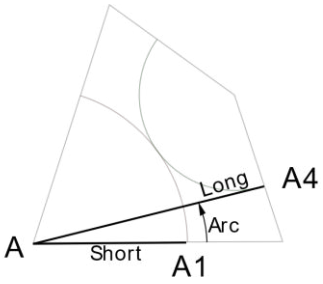




The Arc:



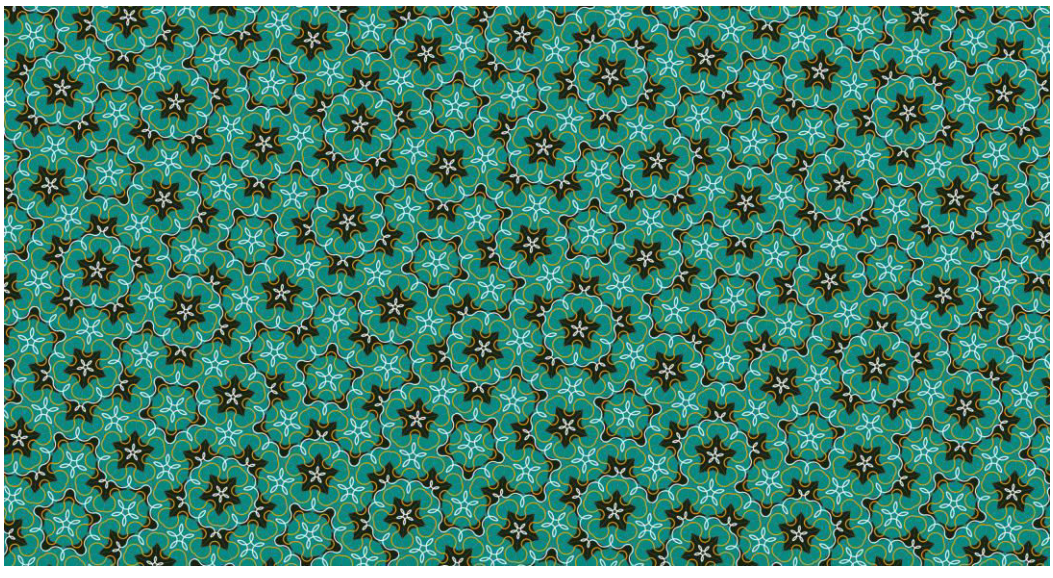




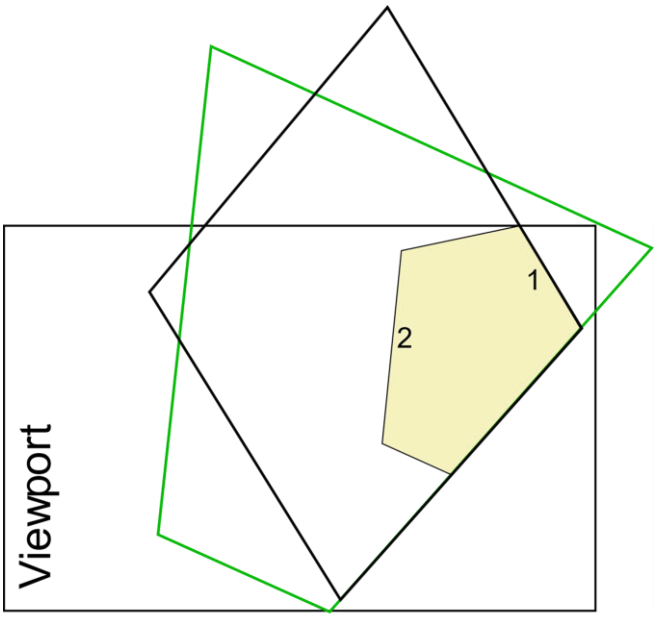
Every decoration point has its own coordinates, which is set in the A1 to A4 (A as Amman or Arc) in **decor** property of the tile. I have hardcoded the **short**, **long** and **arc** values in the tile prototype (for not JS coders, prototype is a JS term), which are needed for calculating the coordinates of the decoration points (see source code). The decor property is then needed next when we generate the polygons for the svg in **renderView** module. For more about decoration please read:

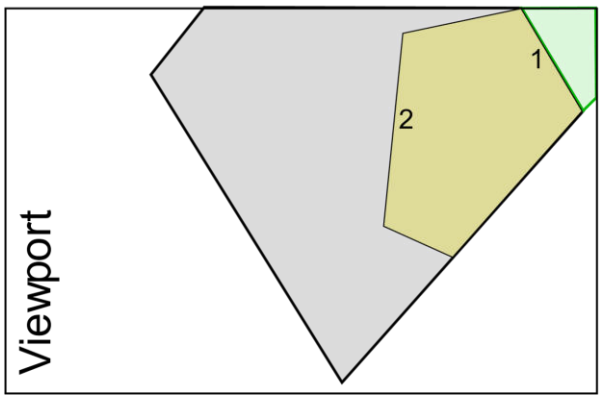
- Amman lines/bars
  https://ksuweb.kennesaw.edu/~sedwar77/tile/aperiodic/empires/ammanbars.htm
- Arc
  https://mathworld.wolfram.com/PenroseTiles.html

Tip: When you know a little in svg, then you can play around in source code with the arc decoration path and you can get some interesting, "nonstandard" pattern decorations like this:



Appendix





Example (demonstration): There is a coffin shape generated on the viewport...

(top) When we continue growing on the side '1' of the coffin, assume we get the green line shape.Then the side '2' would be occupied i.e. in this example the side '2' depends on growth on side '1'. But when we continue the growth on the side '2' and not side '1', assume getting the black shape.

(bottom) When we would stop with the grow because of the border of vewport, and we generate the same tile on side '1' and side '2' like previously, then it happens that, what souldn't be happening... an unaccurate pattern.

And the last thing you can ask is, when we generate tiles till to dead surface, then many of them would lie outside of the viewport and would be not rendered, then why we generate them at all? So, yes many of them or most of them would be not rendered, but when we don't do it, then it can be, that our pattern wouldn't be accurate. The growing shape sometimes overgrows the starting shape from more or each side. This mean, it is common that the tiles on one side depends from the growth of the other side. When we stop the growth because there is the viewport border, and then generate an other constellation to the side where should be tiles already (but because we stopped the growth, there are no tiles), we get an unaccurate pattern, which doesn't exist on the whole infinite penrose world...
(read the picture example)

I hope it helps the logic better to understand and pick-up faster the source code.

Please let me know when you have questions,issues or something related with the project.

Thank you for reading,
Regards
Peter Duricek