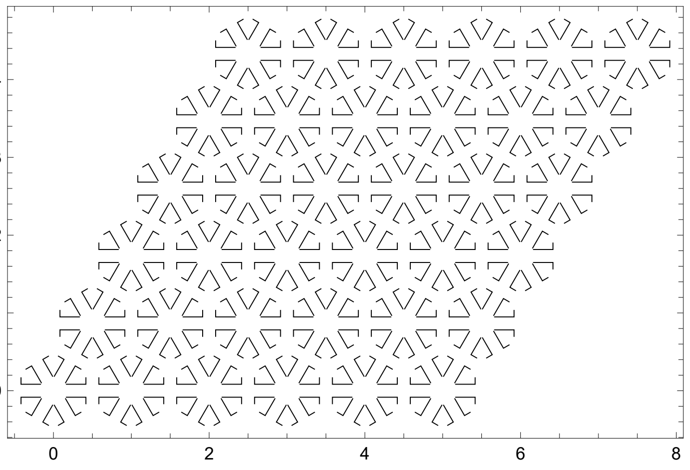
test

```
In[1]:= Get[
         "c:/Users/xah/Desktop/Xah_WolframLang_packages/PlaneTiling/PlaneTiling_dir/PlaneTiling.
           m"]

In[3]:= ? WallpaperPlot
```

```
Symbol

Out[3]=  WallpaperPlot[n] plots a wallpaper having symmetry ID n. 1 ≤ n ≤ 17. WallpaperPlot["Conway
              notation"] and WallpaperPlot["IUC notation"] can also be used. Example: WallpaperPlot[17]
```

```
In[4]:= WallpaperPlot[17, PlotPoints → {6, 6}, Frame → True]
```

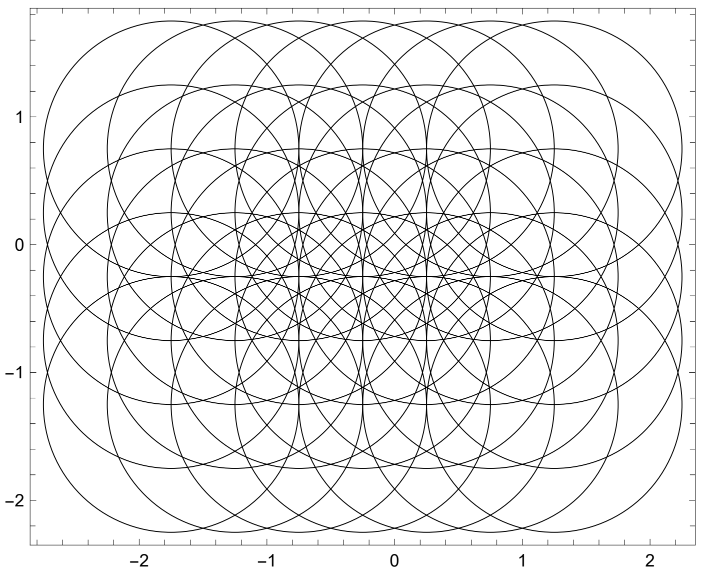
Out[4]=



The default fundamental motif is an L-shaped line segments. You can specify any fundamental motif by the option MotifGraphics. If your fundamental motif is symmetric by itself, then the resulting wallpaper may have more symmetry than specified. In this example, a circle is used as a fundamental motif for the group "o" (p1). The option PlotPoints specifies the number of unit cells to generate.

```
In[5]:= WallpaperPlot["o", MotifGraphics → {Circle[{0, 0}, 1]},
         BasisVectors → {{.5, 0}, {0, .5}}, PlotPoints → {7, 5}, Frame → True, CenterGraphics → True]
```

Out[5]=

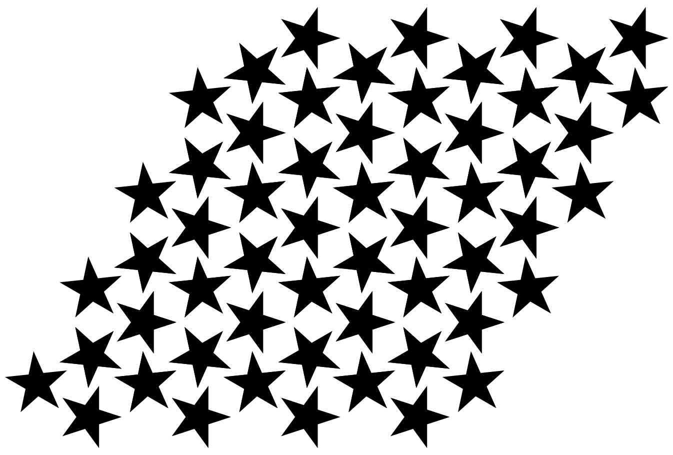


Here's another example, using star as fundamental motif.

In[6]:= WallpaperPlot$\left[15, \text{MotifGraphics} \to \text{StarMotif}\left[5, 3\left\{1, \frac{\text{Cos}[\alpha]}{\text{Cos}\left[\frac{\alpha}{2}\right]} \text{ /. } \alpha \to \frac{2\pi}{5}\right\}, 0.1, 0, \{0.65, 0\}\right],\right.$

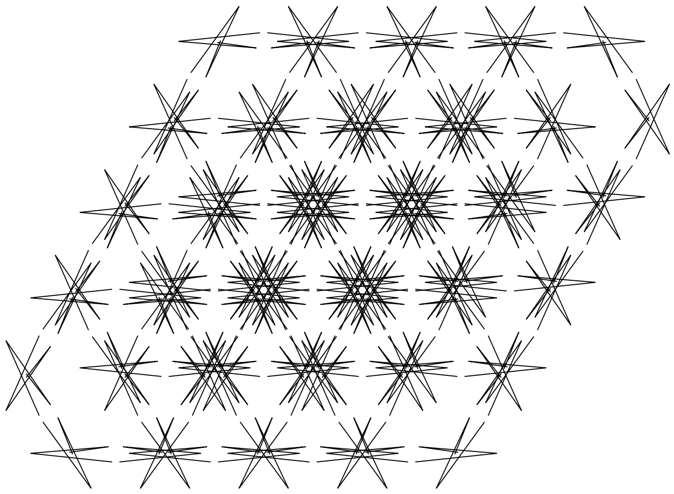
$\left.\text{PlotPoints} \to \{4, 4\}\right]$

Out[6]=



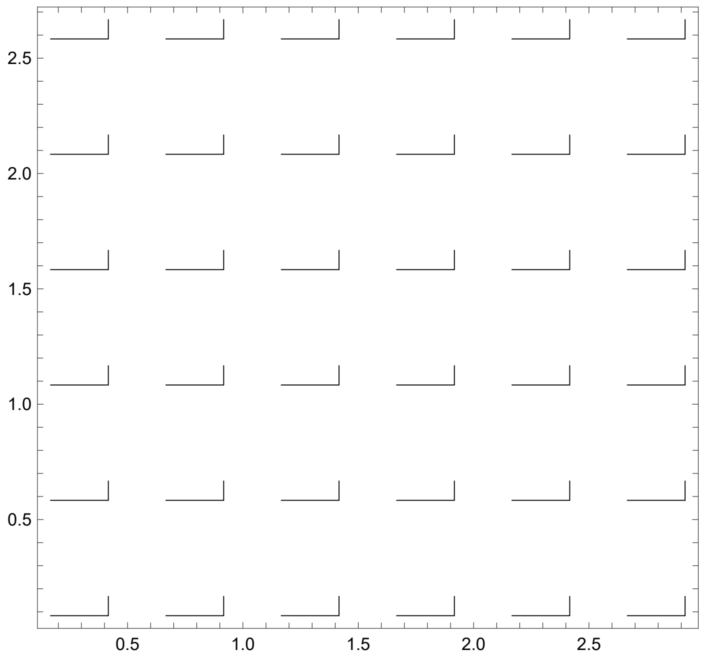
Random lines can be used to generate a wallpaper.

In[10]:=
```
WallpaperPlot[17, MotifGraphics → Line[Table[RandomReal[{0, 1}, 2], {4}]]]
```
Out[10]=



The metrics for the lattice can be specified by the option BasisVectors. In the following example, we use an basis vector {.5,0} and {.5,0} for the group p1. Note that by specifying arbitrary BasisVectors, the resulting wallpaper may not have the original symmetry group.

In[11]:=
```
WallpaperPlot["p1", BasisVectors → {{.5, 0}, {0, .5}}, PlotPoints → {6, 6}, Frame → True]
```
Out[11]=

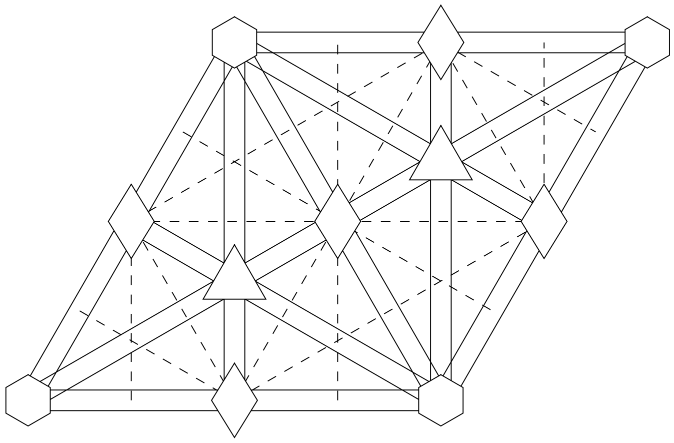


showing the symmetry diagram of a wallpaper

The symmetry elements for the wallpaper can be shown by the option ShowSymmetryAndUnitCell-Graphics->True.

In[12]:=
```
WallpaperPlot[17, MotifGraphics → {}, ShowSymmetryAndUnitCellGraphics → True,
 PlotPoints → {1, 1}, UnitCellGraphicsFunction → ({} &)]
```

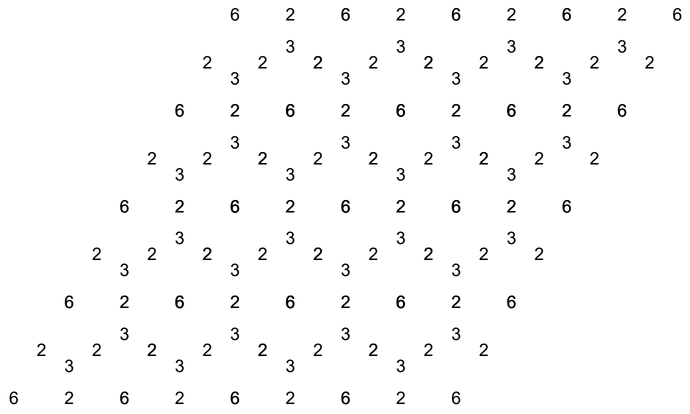
Out[12]=



The default symmetry elements graphics are based on traditional symbols. Rotations are represented by regular polygon. Reflections are double lines. Glide-reflections are dashed lines.
Different graphics for the symmetry elements can be specified with the following options: FundamentalRegionGraphicsFunction, GlideReflectionGraphicsFunction, MirrorLineGraphicsFunction, RotationSymbolGraphicsFunction, UnitCellGraphicsFunction.
In this example, numbers are used to indicate order of rotation.

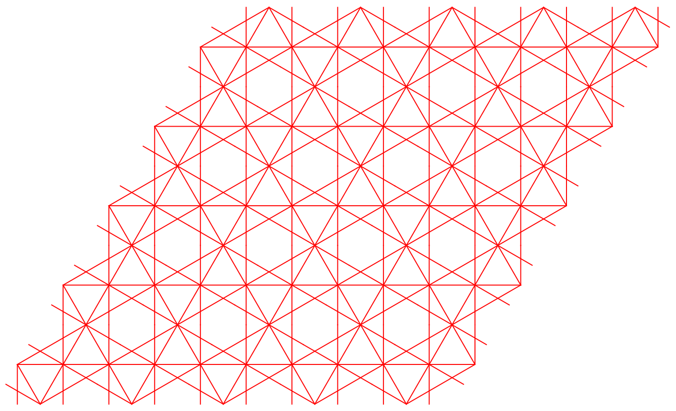
In[13]:=
```
WallpaperPlot[16, MotifGraphics → {}, ShowSymmetryAndUnitCellGraphics → True,
 RotationSymbolGraphicsFunction → ({Text[#1, #2]} &),
 UnitCellGraphicsFunction → ({} &), PlotPoints → {4, 4}]
```

Out[13]=



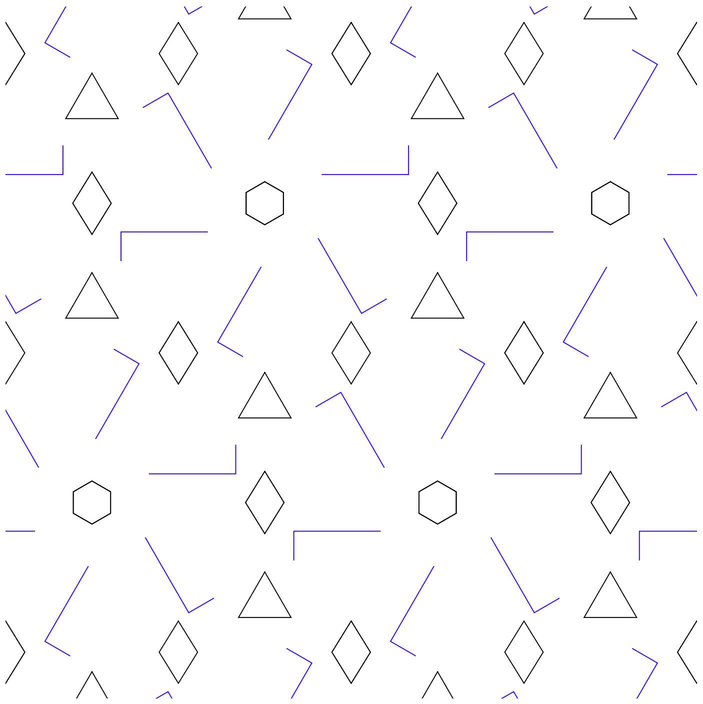
You can also supress plotting of certain symmetry elements, by specifying its graphics function to return an empty list. ({}&). In this example, the non-trivial glide reflection (glide-reflection that is not also a reflection) elements of the group *632 are shown as red lines.

```
In[14]:= WallpaperPlot["*632", MotifGraphics → {},
          ShowSymmetryAndUnitCellGraphics → True, RotationSymbolGraphicsFunction → ({} &),
          UnitCellGraphicsFunction → ({} &), MirrorLineGraphicsFunction → ({} &),
          GlideReflectionGraphicsFunction → ({Hue[0], Line[{#1, #2}]} &), PlotPoints → {5, 5}]
Out[14]=
```



In this example, a wallpaper with symmetry 632 are show together with its symmetry elements superimposed.

```
In[18]:= WallpaperPlot[16,
          MotifGraphics → {Hue[.7], Line[{{0.166, 0.083}, {0.416, 0.083}, {0.416, 0.166}}]},
          ShowSymmetryAndUnitCellGraphics → True, PlotPoints → {5, 5},
          UnitCellGraphicsFunction → ({} &), CenterGraphics → True, PlotRange → {{-1, 1}, {-1, 1}}]
Out[18]=
```



WallpaperPlot can accept all options for Graphics, and the following special ones. All options have on-line documentation.

BasisVectors, CenterGraphics, MotifGraphics, PlotPoints, ShowSymmetryAndUnitCellGraphics.

FundamentalRegionGraphicsFunction, GlideReflectionGraphicsFunction, MirrorLineGraphicsFunction, RotationSymbolGraphicsFunction, UnitCellGraphicsFunction

WallpaperGroupData

WallpaperGroupData gives a table that correlates the Conway notation (orbifold notation) and IUC (International Union of Crystallography) notation. It also gives the basis vectors, fundamental region, and group generators. These information are used by WallpaperPlot.

```
In[21]:= ? WallpaperGroupData
Out[21]=
```

Symbol

WallpaperGroupData is a list of data about wallpaper groups, some of which are default values for WallpaperPlot. Each row contains: {ID Number,Conway notation,IUC notation,basis vectors,fundamental region,generators}. Example: TraditionalForm@TableForm@WallpaperGroupData

Here is the information for the group *632. The elements are: ID number, Conway notation, IUC notation, basis vectors, fundamental region, and generators. The generators include two translations with basis vectors, not shown.

```
In[19]:= WallpaperGroupData[[17]]
Out[19]=
```

$$\left\{17,\ \ast 632,\ \text{p6m} \mid \text{p6mm},\ \left\{\{1,0\},\left\{\frac{1}{2},\frac{\sqrt{3}}{2}\right\}\right\},\ \text{Polygon}\left[\left\{\{0,0\},\frac{\#1}{2},\frac{\#1+\#2}{3}\right\}\right]\&,\ \left\{\text{Reflection}[\#1+\#2]\&,\ \text{Rotation}\left[\{0,0\},\frac{\pi}{3}\right]\&\right\}\right\}$$

The fundamental region is given as a pure function. The arguments #1 and #2 are to be replaced by basis vectors, and the return value is a Polygon object. In this example, we apply the function to basis vectors to get the actual coordinates of a fundamental region.

```
In[23]:= Apply[WallpaperGroupData[[17, 5]], WallpaperGroupData[[17, 4]]]
Out[23]=
Polygon[ Number of points: 3
         Embedding dimension: 2 ]
```

Similarly, the generators are given as a list of pure functions, with basis vectors for their arguments. We can apply the list of generators to the default basis vectors to get the generators in terms of actual coordinates.

```
In[24]:= Through[WallpaperGroupData⟦17, 6⟧[Sequence @@ WallpaperGroupData⟦17, 4⟧]]
Out[24]=
```

$\{\text{Reflection}[\{\frac{3}{2}, \frac{\sqrt{3}}{2}\}], \text{Rotation}[\{0, 0\}, \frac{\pi}{3}]\}$

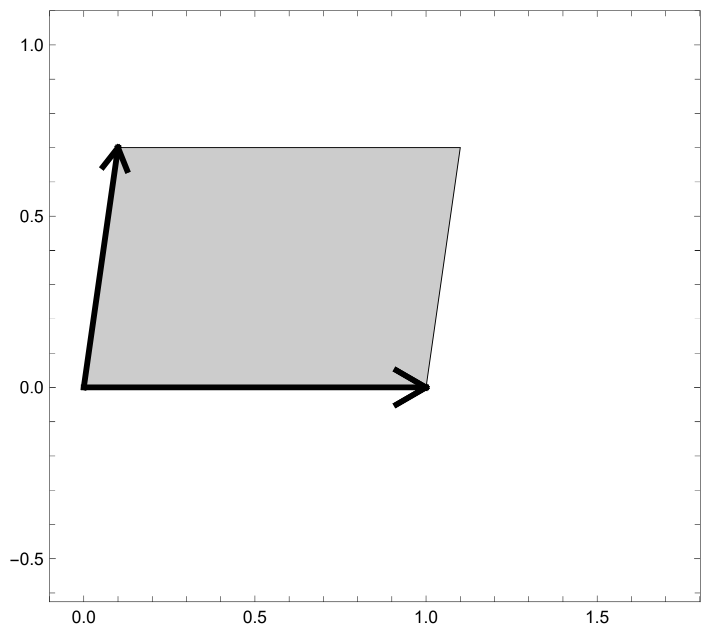
The advantage of using pure functions for the fundamental region and generators is the flexibility of specifying different basis vectors. Suppose you want to know the fundamental region and the generators for the group 2*22 (cmm) (ID: 9) based on the basis vectors {1,0} and {0,1}. You can get the fundamental region by applying the pure function to the basis vectors as in: WallpaperGroupData[[9,5]][{1,0},{0,1}]. For the generators, pass the basis vectors to the pure functions inside the generator pure functions list. This is done by Through[WallpaperGroupData[[9,6]][{1,0},{0,1}]].

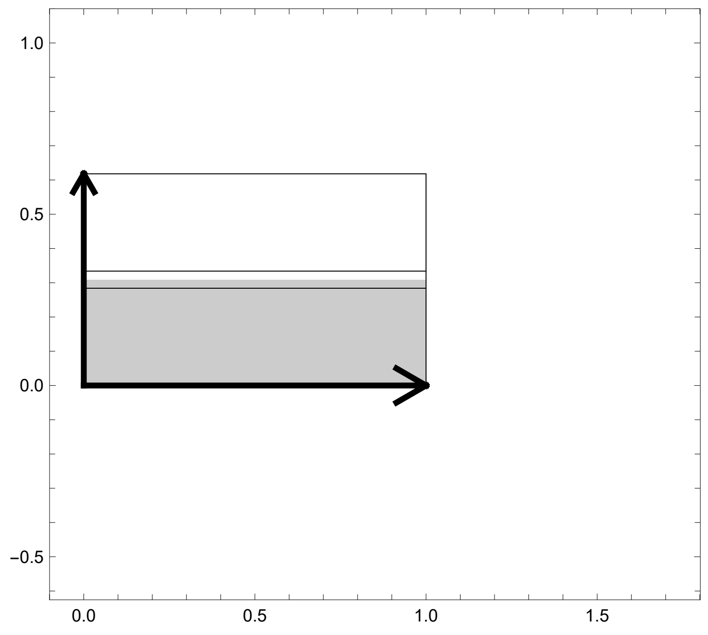
```
In[25]:= WallpaperGroupData⟦9, 5⟧[{1, 0}, {0, 1}]
         Through[WallpaperGroupData⟦9, 6⟧[{1, 0}, {0, 1}]]
Out[25]=
         Polygon[ Number of points: 3
                  Embedding dimension: 2 ]
Out[26]=
         {Reflection[{-1, 1}, {1, 0}], Reflection[{1, 1}]}
```

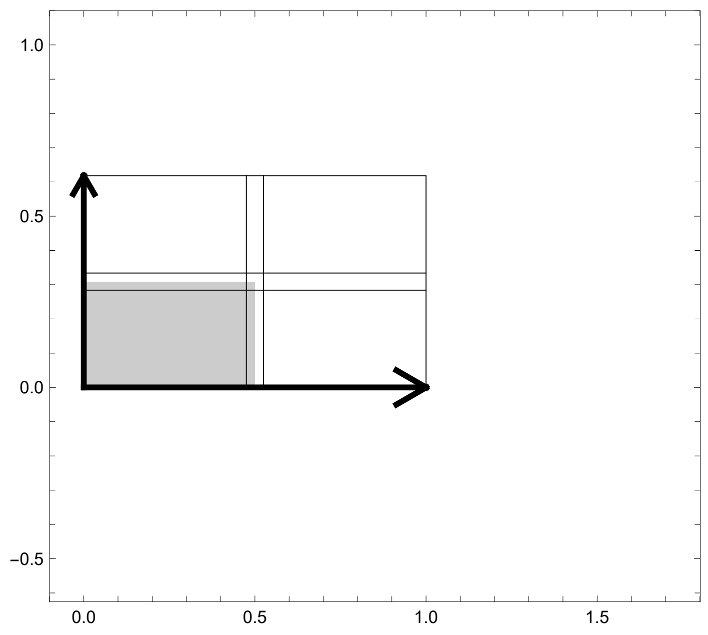
Plotting unit cell, fundamental region, and generators.

We can plot the unit cell, the fundamental region, and the generators of the group. All these informations are in WallpaperGroupData. We just need to apply Graphics primitives to the data. In the following, grayed area is the fundamental region. Dashed lines are glide relection. Double lines are reflection.

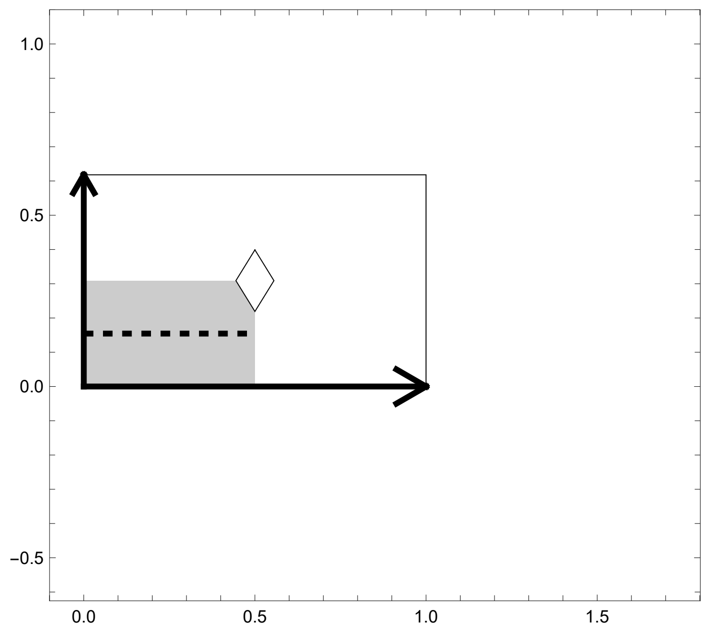
```
In[27]:= Do[gp = {{GrayLevel[0.8`], WallpaperGroupData⟦i, 5⟧[Sequence @@ WallpaperGroupData⟦i, 4⟧]},
            {(Line[{{0, 0}, First[#1], Plus @@ #1, Last[#1], {0, 0}}] &)[WallpaperGroupData⟦i, 4⟧]},
            {({Thickness[0.009`], VectorMotif[#1]} &) /@ WallpaperGroupData⟦i, 4⟧},
            {Through[WallpaperGroupData⟦i, 6⟧[Sequence @@ WallpaperGroupData⟦i, 4⟧]] /.
               {GlideReflection[a_] → GlideReflection[a, {0, 0}],
                Reflection[a_] → Reflection[a, {0, 0}]}} /.{GlideReflection[a_, b_] :→
                {Thickness[0.009`], AbsoluteDashing[{5, 5}], Thickness[0.009`], Line[{b, b + a}]},
               Reflection[a_, b_] :→ DoubleLineMotif[b, b + a, 0.05`], Rotation[a_, α_] :→
                {{GrayLevel[1], RotationSymbolMotif[2π/α, 0.1`, 0, a] /.Line → Polygon},
                 RotationSymbolMotif[2π/α, 0.1`, 0, a]}}}};
  Print[Show[Graphics[gp], AspectRatio → Automatic, Frame → True,
     PlotRange → {{-1, 1}, {-1, 1}} 0.1` + ({Min[#1], Max[#1]} &) /@ Transpose[
         (Flatten[#1, 1] &)[({{0, 0}, First[#1], First[#1] + Last[#1], Last[#1]} &) /@
           Transpose[WallpaperGroupData]⟦4⟧]]]], {i, 1, 17, 2}]
```


1.0
0.5
0.0
−0.5
0.0
0.5
1.0
1.5

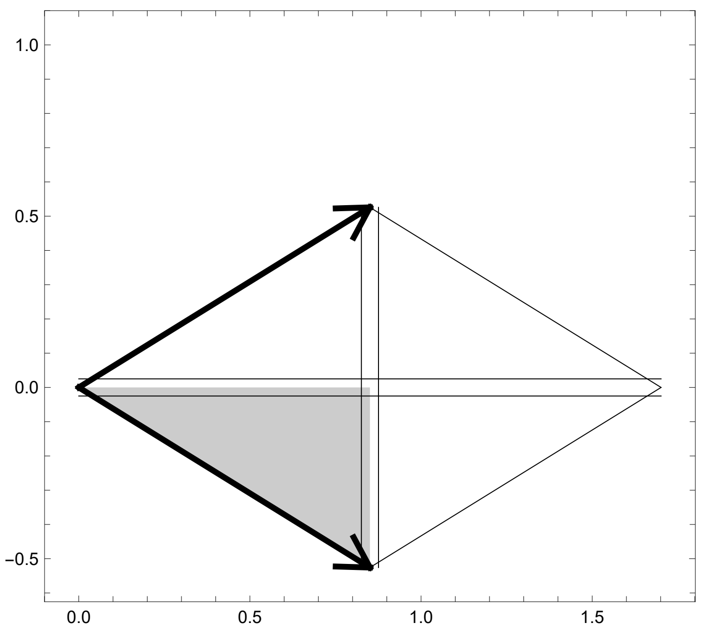

1.0
0.5
0.0
−0.5
0.0
0.5
1.0
1.5



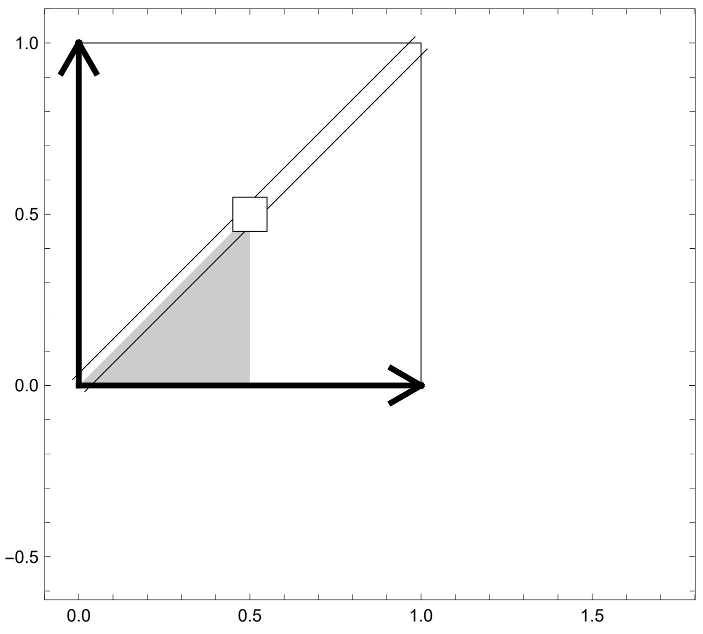
1.0
0.5
0.0
−0.5
0.0
0.5
1.0
1.5



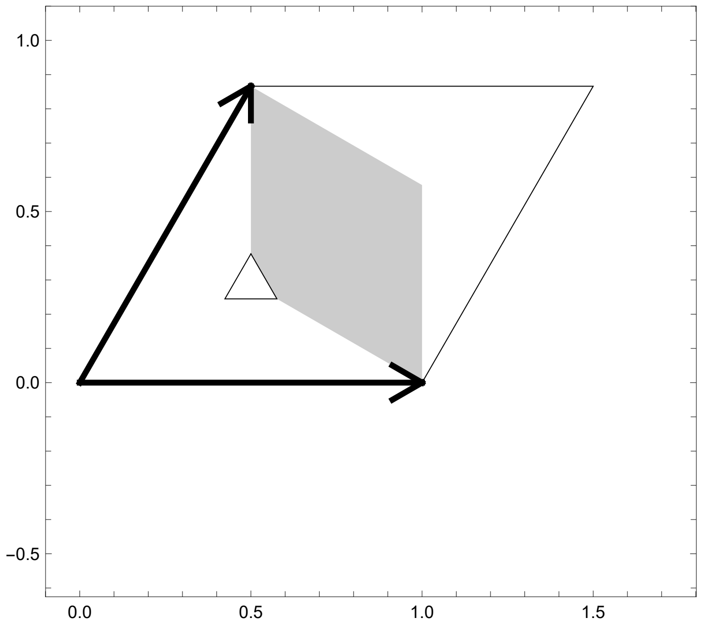
1.0
0.5
0.0
−0.5
0.0
0.5
1.0
1.5

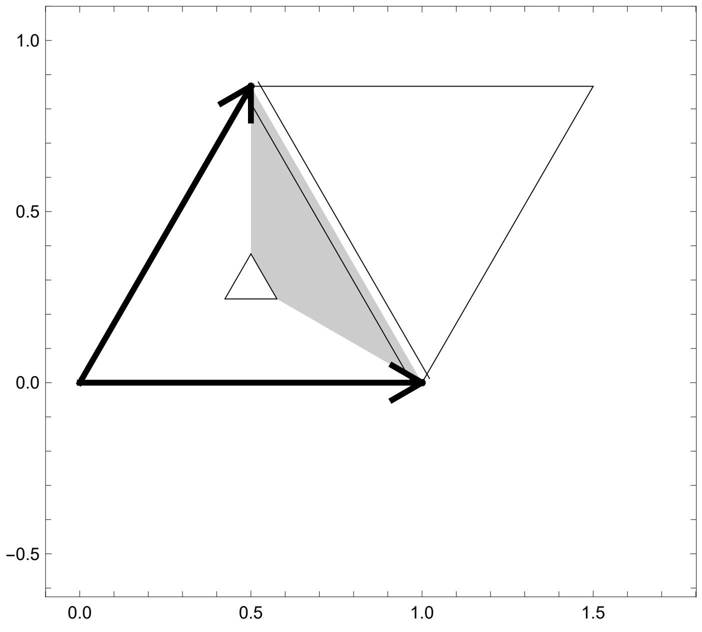


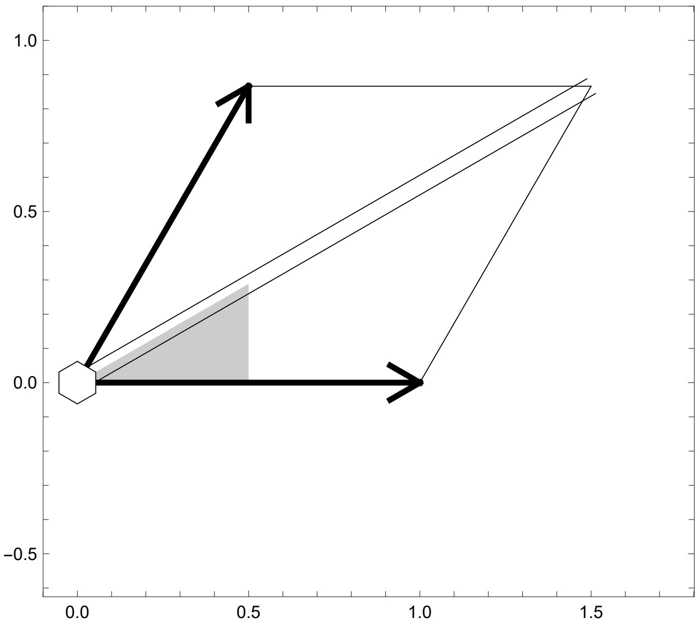
1.0
0.5
0.0
−0.5
0.0
0.5
1.0
1.5



1.0
0.5
0.0
−0.5
0.0
0.5
1.0
1.5


1.0
0.5
0.0
−0.5
0.0
0.5
1.0
1.5


1.0
0.5
0.0
−0.5
0.0
0.5
1.0
1.5



For those technically inclined, here is an explaination on how the above code works.

The outline of the code has the form Do[gp={...};Show[Graphics[{gp}]],{i,1,17}]. gp is a list of graphics primitives. gp has several sublists, representing the fundamental region, unit cell, translations, and the rest of generators. In each list, we wrap a graphic primitive (e.g. Line,Point) to coordinates pulled from WallpaperGroupData.

The first sublist represents the fundamental region. It's code is

{GrayLevel[.8],WallpaperGroupData[[i,5]]@Sequence@@WallpaperGroupData[[i,4]]]}

The second sublist represents the unit cell. If a and b are the basis vectors then the desired result is Line[{{0,0},a,a+b,b,{0,0}}]. Here's the code.

{Line[{{0,0},First@#,Plus@@#,Last@#,{0,0}}]&@WallpaperGroupData[[i,4]]}

Next we have translations. We map VectorMotif to the basis vectors.

{{Thickness[.009],VectorMotif[#]}&/@WallpaperGroupData[[i,4]]}

The last sublist is the generators. Here's the code:

{Through[WallpaperGroupData[[i,6]][Sequence@@WallpaperGroupData[[i,4]]]]/.{GlideReflection[a_]->GlideReflection[a,{0,0}],Reflection[a_]->Reflection[a,{0,0}]}/.{GlideReflection[a_,b_]:>{Thickness[.009],AbsoluteDashing[{5,5}],Thickness[.009],Line[{b,b+a}]},Reflection[a_,b_]:>DoubleLineMotif[b,b+a,.05],Rotation[a_,α_]:>{{GrayLevel[1],RotationSymbolMotif[2*Pi/α,0.1,0,a]/.Line->Polygon},RotationSymbolMotif[2*Pi/α,.1,0,a]}}}

The code looks complex, but the idea is simple. The outline of the code is this:

{generatorsList/.{rules2}/.{rules3}}

An example of generatorsList is {Reflection[vector1,vector2],Rotation[point,α]}. Our goal is to replace the generators by graphics primitives applied to their coordinates. This is done by ReplaceAll and Rule. rule2 is {GlideReflection[a_]->GlideReflection[a,{0,0}],Reflection[a_]->Reflection[a,{0,0}]}, it puts the reflection and glide reflection in a two point form. Remember that Reflection[a] means reflection through the vector a, while Reflection[a,b] means reflection through the line parallel to vector a and passing b. The rules3 is the actual replacement of graphics primitives.

RandomWallpaperPlot

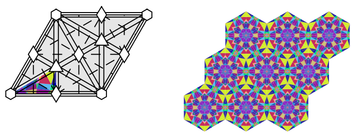
RandomWallpaperPlot plots a random wallpaper design with a specified wallpaper group. The design has the style of classic kaleidoscope. Note: RandomWallpaperPlot may consume as much as 5 megabytes of memory in the kernel. It takes about 20 seconds on a 1996 desktop computer for the function to complete.

```
In[28]:= ? RandomWallpaperPlot
Out[28]=
```

Symbol

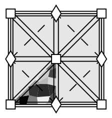
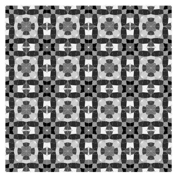
RandomWallpaperPlot[n,(opts)] plots a random wallpaper with nth symmetry group. RandomWallpaperPlot[n,m,s,r,(opts)] base it on m−gons (m = 3, 4, 6) with side length s (0.05 <= s <= .5). r controls the randomness of the wallpaper. (0 < r <=1) Special options are: ColorFunction−>(RGBColor[Random[],Random[],Random[]]&) can be used to specify color. PlotPoints−>{4,4} can be specified to control the number of cells in the plot. Example: RandomWallpaperPlot[17,3,.1,1,PlotPoints−>{3,3}]

```
In[29]:= RandomWallpaperPlot[17, 3, .1, 1, PlotPoints → {3, 3}]
Out[29]=
```



If you want to generate the exact same design, you can do so by setting SeedRandom. The same seed will generate the exact same wallpaper design. You can also control the color using the option ColorFunction.

```
In[30]:= SeedRandom[1]
         RandomWallpaperPlot[11, 4, 0.1`, 1, ColorFunction → (GrayLevel[RandomReal[]] &)]
Out[30]=
         RandomGeneratorState[ Method: ExtendedCA
                               State hash: -3 912 834 453 189 707 681 ]
Out[31]=
```

Lattices and Networks

Lattices and Networks

LatticeCoordinates

The function LatticeCoordinates returns a list of coordinates generated by two vectors. Graphics generating functions can be then mapped to this list to generate a tiling or wallpaper design.

```
In[32]:= ? LatticeCoordinates
Out[32]=
```

Symbol

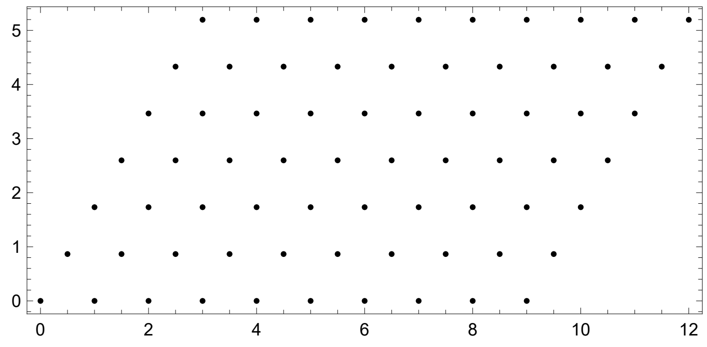
LatticeCoordinates[{a1,a2},{b1,b2},{m,n}] returns a list of coordinates on a grid generated by vectors {a1,a2} and {b1,b2} with m and n points in each direction. LatticeCoordinates[{a1,a2},{b1,b2},{m,n},s,α,{x,y}] scales, rotates, and translates the grid by s, α, and {x,y}. See also: ColoredLatticePoints. Example: Show[Graphics[{Map[Point,LatticeCoordinates[{1,0},{1/2,Sqrt[3]/2},{7,5}],{2}]}],AspectRatio->Automatic]

This gives a square grid generated by vectors {1,0} and {0,1}.

```
In[33]:= LatticeCoordinates[{1, 0}, {0, 1}, {4, 3}]
Out[33]=
         {{{0., 0.}, {1., 0.}, {2., 0.}, {3., 0.}},
          {{0., 1.}, {1., 1.}, {2., 1.}, {3., 1.}}, {{0., 2.}, {1., 2.}, {2., 2.}, {3., 2.}}}
```

This gives a triangular grid generated by vectors {1,0} and {1/2,Sqrt[3]/2}.

```
In[34]:= Show[Graphics[{Map[Point, LatticeCoordinates[{1, 0}, {1/2, √3/2}, {10, 7}], {2}]}],
          AspectRatio → Automatic, Frame → True]
Out[34]=
```



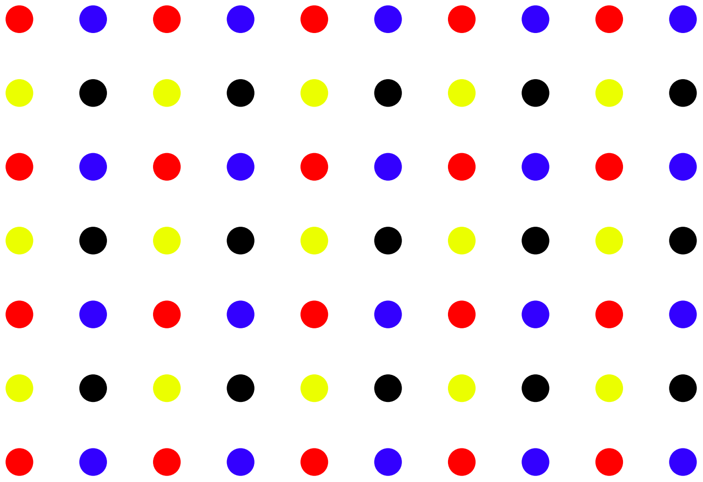
ColoredLatticePoints

ColoredLatticePoints gives you a grid colored in a regular way.

```
? ColoredLatticePoints
```

ColoredLatticePoints[{a1,a2},{b1,b2},{m,n},{colorList1,colorList2,...}] returns a n by m matrix of Points generated by vectors {a1,a2} and {b1,b2}. colorLists are lists of colors (e.g. Hue[0]) or other graphics directives. colorLists are repeated cyclically. ColoredLatticePoints[{a1,a2},{b1,b2},{m,n},colorLists,s,α,{x,y}] scales, rotates, and translates the lattice by s, α, and {x,y}. Example: Show[Graphics[{PointSize[.04],ColoredLatticePoints[{1,0},{0,1},{10,7},{{Hue[0],Hue[.7]},{Hue[.18],GrayLevel[0]}}]}],AspectRatio-> Automatic]

```
In[35]:= Show[Graphics[{PointSize[0.04`], ColoredLatticePoints[{1, 0}, {0, 1}, {10, 7},
             {{Hue[0], Hue[0.7`]}, {Hue[0.18`], GrayLevel[0]}}]}], AspectRatio → Automatic]
Out[35]=
```

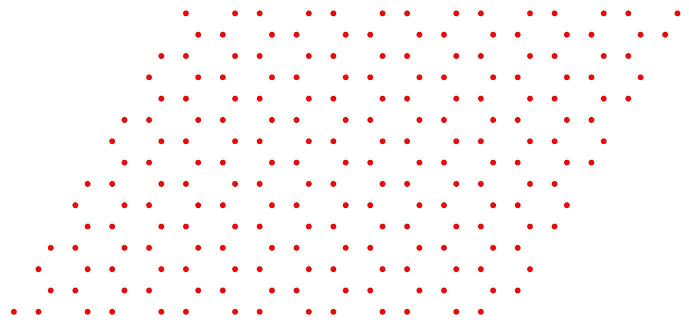


ColoredLatticePoints lets you specify all possible ways to regularly color a square or triangular lattice. ColoredLatticePoints is versatile because it partitions the lattice into subsets by tagging lattice points with a different color. You can target subsets of points for manipulation. For example, you can map different design to lattice points according to their color. Or, you can delete certain color set to create a special lattice.

This example shows a lattice that is the vertex set of a regular hexagon tiling. This is done by creating a triangle lattice with basis vectors {1,0} and {1/2,Sqrt[3]/2}. We give this lattice the color signature {{Hue[0],Hue[0],0},{0,Hue[0],Hue[0]},{Hue[0],0,Hue[0]}}, which is generated by Table[RotateRight[{Hue[0], Hue[0],0},i],{i,0,2}]. Then, we delete points in the result by replacing all occurances of {0,_Point} to {}. The final image is the desired result.

```
In[36]:= Show[Graphics[{ColoredLatticePoints[{1, 0}, 1/2 {1, √3},
             {21, 15}, Table[RotateRight[{Hue[0], Hue[0], 0}, i], {i, 0, 2}]] /.
            {0, _Point} → {}}], AspectRatio → Automatic]
```

Out[36]=

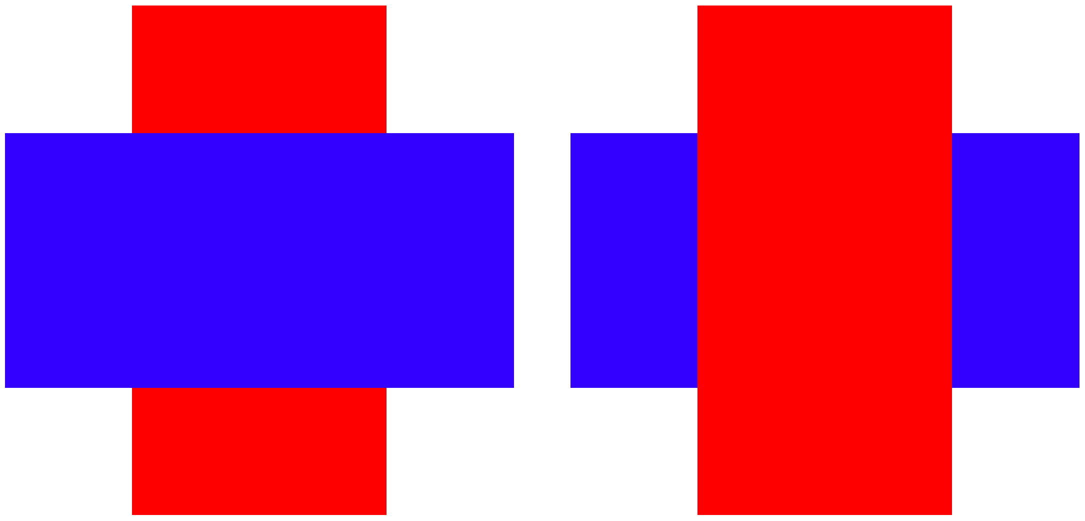


This example shows a simple weaving. tile1 and tile2 are square tiles with different design. They are mapped onto a square lattice. The choice of tile is corresponds to the color of the lattice point, generated regularly by ColoredLatticePoints. Any elaborate weavings or knots can be generated using this technique.

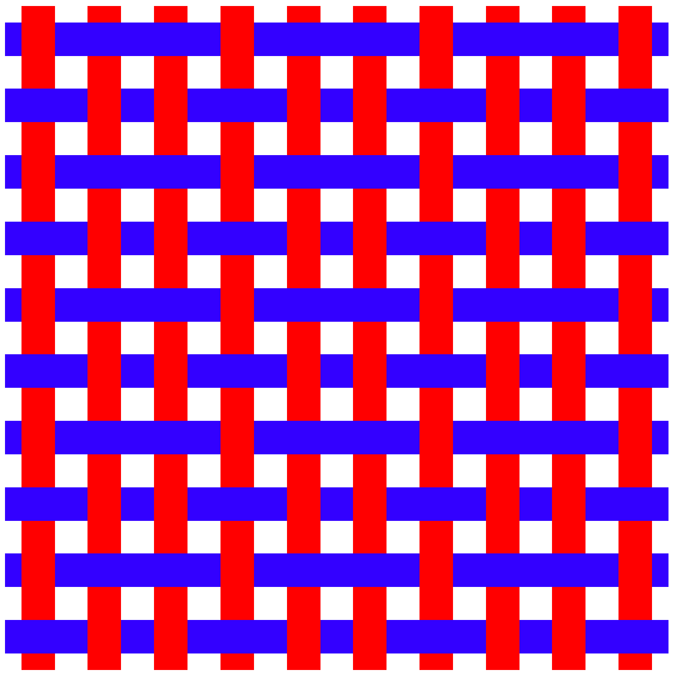
```
In[37]:= Clear[tile1, tile2, gp]
         tile1 =
           {{Hue[0], Polygon[{{0.25`, 0.5`}, {-0.25`, 0.5`}, {-0.25`, -0.5`}, {0.25`, -0.5`}}]},
            {Hue[0.7`],
             Polygon[{{0.5`, 0.25`}, {0.5`, -0.25`}, {-0.5`, -0.25`}, {-0.5`, 0.25`}}]}};
         tile2 = Reverse[tile1]
         Show[GraphicsRow[{Graphics[{tile1}, AspectRatio → Automatic],
             Graphics[{tile2}, AspectRatio → Automatic]}]]
         gp = ColoredLatticePoints[{1, 0}, {0, 1}, {1, 1} 10, {{1, 2, 2}, {2, 1, 1}}] /.
             {{1, Point[pt_]} → Translate2DGraphics[pt][tile1],
              {2, Point[pt_]} → Translate2DGraphics[pt][tile2]};
         Show[Graphics[{gp}], AspectRatio → Automatic]
```

```
Out[39]=
{{🟦, Polygon[ ⊞ Number of points: 4
                  Embedding dimension: 2 ]}, {🟥, Polygon[ ⊞ Number of points: 4
                                                             Embedding dimension: 2 ]}}
```

Out[40]=



Out[42]=



ColoredLatticeNetwork

ColoredLatticeNetwork returns a list of Line graphics primitives that represents a network colored regularly.

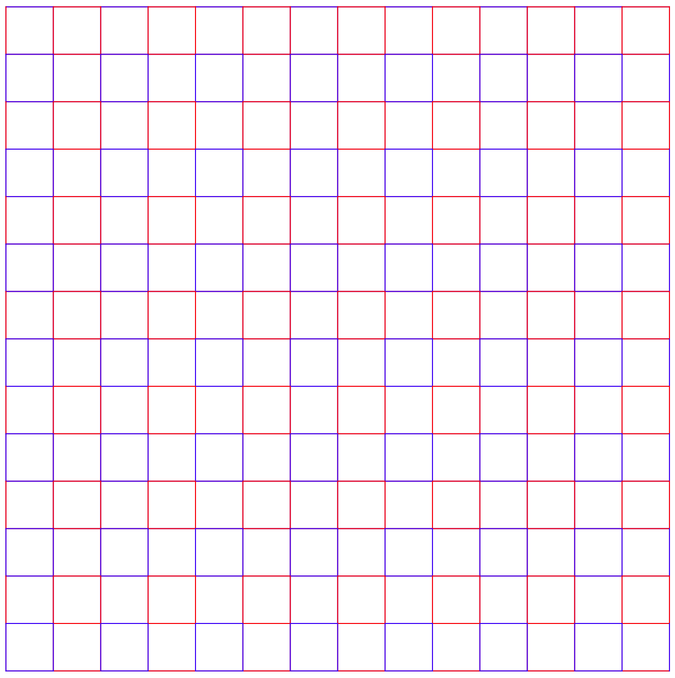
```
In[43]:= ? ColoredLatticeNetwork
Out[43]=
```

Symbol

ColoredLatticeNetwork[n,m,{colorList1,colorList2,...}] returns a lattice of line segments colored regularly. A square grid if n is 4, triangular grid if n is 3. m controls the size of the grid. Returned list has dimensions {n,2*m+1,2*m,2} and n*(2*m+1)*(2*m) Line segments. colorLists are lists of colors (e.g. Hue[0]) or other graphics directives. colorLists are repeated cyclically. ColoredLatticeNetwork[n,m,colorLists,s,α,{x,y}] scales, rotates, and translates the lattice by s, α, and {x,y}. Example: Show[Graphics[{Thickness[.01],ColoredLatticeNetwork[4,3,{{Hue[0],Hue[.7]},{Hue[.35],GrayLevel[0]}}]}],AspectRatio ->Automatic]

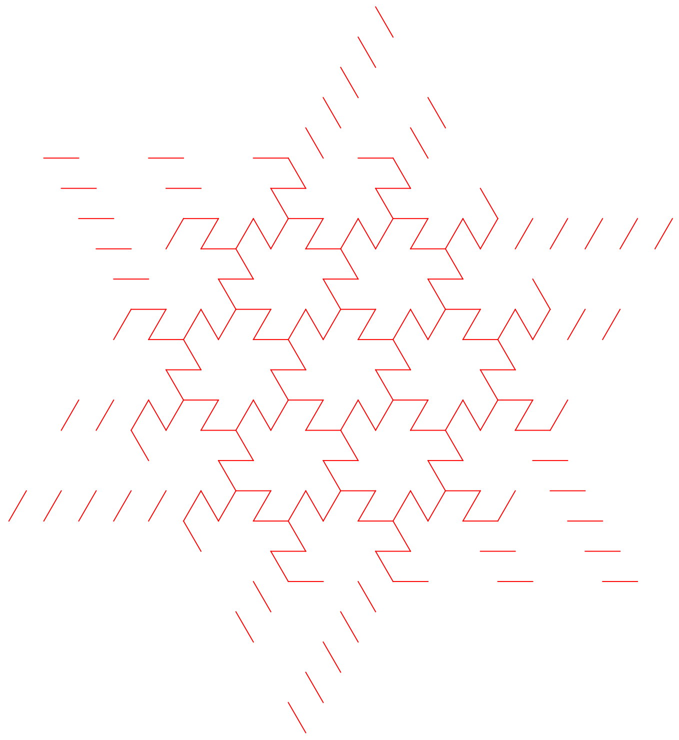
Here's a square network, colored with alternating red and blue segments.

```
In[44]:= Show[Graphics[{ColoredLatticeNetwork[4, 7, {{Hue[0.7`], Hue[0]}}]}],
          AspectRatio → Automatic]
Out[44]=
```



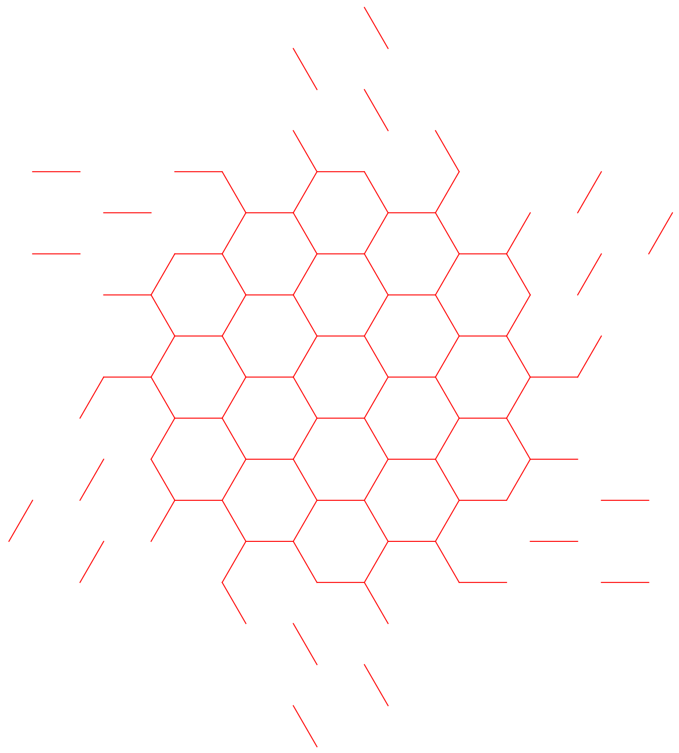
ColoredLatticeNetwork lets you specify all possible ways of coloring edges of a triangular or square tiling. Various color specification can result various designs. Here's a snowflake-like tiling, created by deleting segments with specific color in a triangular lattice colored regularly.

```
In[45]:= Show[Graphics[{ColoredLatticeNetwork[3, 7, {{Hue[0.7`], Hue[0], Hue[0.35`]}}] /.
             {{Hue[0.7` | 0.35`], li_Line} → {}}}], AspectRatio → Automatic]
Out[45]=
```



Starting with a different colored triangular lattice, taking out red and blue segments results a honey-comb design.

```
In[46]:= Show[Graphics[
           {ColoredLatticeNetwork[3, 5, Table[RotateLeft[{Hue[0.7`], Hue[0], Hue[0.35`]}, i],
               {i, 0, 2}]] /.{{Hue[0.7` | 0.35`], li_Line} → {}}}], AspectRatio → Automatic]
Out[46]=
```



DirectedLatticeNetwork

DirectedLatticeNetwork is similar to ColoredLatticeNetwork. It returns a square or triangular grid with regularly directed edges.

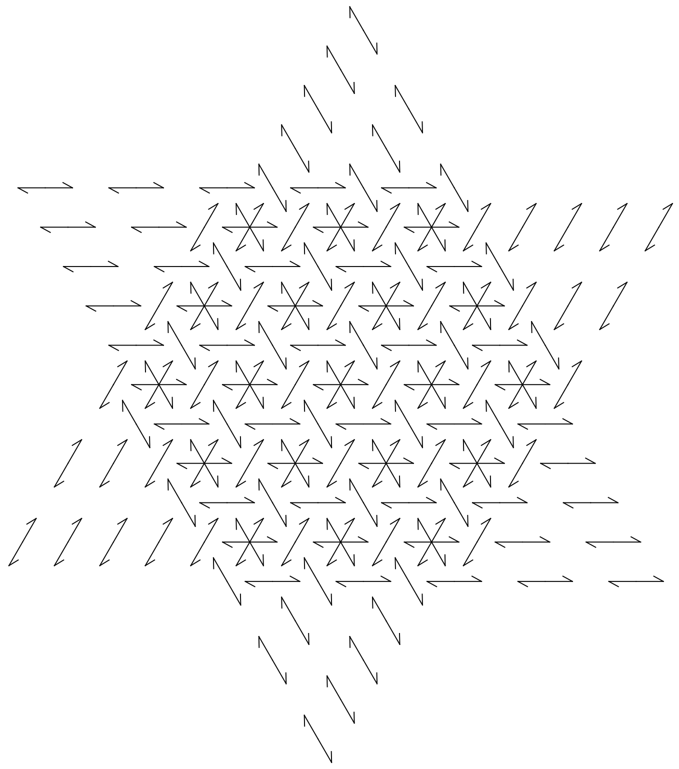
```
? DirectedLatticeNetwork
```

DirectedLatticeNetwork[n,m,{directionList1,directionList2,...}] returns a lattice with regularly directed line segments. A square grid if n is 4, triangular grid if n is 3. m controls the size of the grid. Returned list has dimensions {n,2*m+1,2*m,2} and n*(2*m+1)*(2*m) Line segments. directionLists are lists of True or False values. directionListss are repeated cyclically. DirectedLatticeNetwork[n,m,directionLists,s,α,{x,y}] scales, rotates, and translates it by s, α, and {x,y}. Example: Show[Graphics[{LineTransform[(DirectedLatticeNetwork[3,3,{{True}}]),ArrowMotif[{{.5,.4}},.6,0]]}],AspectRatio->Automatic]

Here the result of DirectedLatticeNetwork is passed to LineTransform, where each line segment is changed to a segment with arrow. The effect is easily seen.

In[47]:=
```
Show[Graphics[{LineTransform[DirectedLatticeNetwork[3, 5, {{True, False}}],
    ArrowMotif[{{0.5`, 0.4`}}, 0.6`, 0]]}], AspectRatio → Automatic]
```

Out[47]=



The power of DirectedLatticeNetwork comes when combined with LineTransform. The combination lets you perform any topological transformation on a regular grid. See the LineTransform section later.

Square, triangle, star, and hexagon lattices

Here we define several more lattices. They are not in the package because in future versions of the package, we may have one single function GeneralLattice[...] that generate them all in a consistent way.

SquareLatticeCoordinates

In[83]:=
```
Clear[SquareLatticeCoordinates]
SquareLatticeCoordinates::"usage" =
  "SquareLatticeCoordinates[(n:5)] returns a list of points
    that represents a square lattice. n controls the number of
    points returned. SquareLatticeCoordinates[n,s,α,{x,y}] scales,
    rotates, and translates the lattice by s, α, and {x,y}. Example:
    Show[Graphics[{Map[Point,SquareLatticeCoordinates[5],{2}]}],AspectRatio->
    Automatic]";
```

```
In[85]:= SquareLatticeCoordinates[n_Integer : 5] := LatticeCoordinates[{1, 0}, {0, 1}, {n, n}]

         SquareLatticeCoordinates[n_Integer, s_] := LatticeCoordinates[{1, 0}, {0, 1}, {n, n}, s]

         SquareLatticeCoordinates[n_Integer, s_, α_] :=
          LatticeCoordinates[{1, 0}, {0, 1}, {n, n}, s, α]

         SquareLatticeCoordinates[n_Integer, s_, α_, {x_, y_}] :=
          LatticeCoordinates[{1, 0}, {0, 1}, {n, n}, s, α, {x, y}]
```

TriangleLatticeCoordinates

```
In[89]:= Clear[TriangleLatticeCoordinates]
         TriangleLatticeCoordinates::"usage" =
           "TriangleLatticeCoordinates[(n:5)] returns a list of points that represents
             a triangular lattice. n controls the number of points on the side.
             TriangleLatticeCoordinates[n,s,α,{x,y}] scales, rotates, and translates
             the lattice by s, α, and {x,y}. (Triangular lattice is defined as
             the vertex set of regular tiling with equilateral triangles) Example:
             Show[Graphics[{Map[Point,TriangleLatticeCoordinates[5],{3}]}],AspectRatio->
             Automatic]";
```

```
In[91]:= TriangleLatticeCoordinates[n_Integer] := TriangleLatticeCoordinates[n, 1, 0]

         TriangleLatticeCoordinates[n_Integer, s_] := TriangleLatticeCoordinates[n, s, 0]

         TriangleLatticeCoordinates[n_Integer, s_, α_] :=
           (Table[Map[{{Cos@i, -Sin@i}, {Sin@i, Cos@i}}.# &, #, {2}],
               {i, α, α + 2 * Pi - 2 * Pi / 3 / 2, 2 * Pi / 3}] &)@(LatticeCoordinates[{1, 0},
              {Cos[2 * Pi / 3], Sin[2 * Pi / 3]}, {n, n}, s, 0, {s / 2, Sqrt[3] / 6 * s}]);

         TriangleLatticeCoordinates[n_Integer, s_, α_, {x_, y_}] :=
          Map[# + {x, y} &, TriangleLatticeCoordinates[n, s, α], {3}]
```

StarLatticeCoordinates

In[95]:=
```
Clear[StarLatticeCoordinates]
StarLatticeCoordinates::"usage" =
  "StarLatticeCoordinates[(n:5)] returns a list of points that
    represents a star lattice. n controls the number of points returned.
    StarLatticeCoordinates[n,s,α,{x,y}] scales, rotates, and translates
    the lattice by s, α, and {x,y}. (star lattice is defined as the
    midpoint of edge set of the regular hexagon tiling) Example:
    Show[Graphics[{Map[Point,StarLatticeCoordinates[5],{4}]}],AspectRatio->
    Automatic]";
```

In[97]:=
```
StarLatticeCoordinates[n_Integer : 5] :=
 N@Map[Table[{Cos@t, Sin@t} + #, {t, 0, Pi - 2 * Pi / 6 / 2, 2 * Pi / 6}] &,
   TriangleLatticeCoordinates[n, 2], {3}]

StarLatticeCoordinates[n_Integer, s_] :=
 N@Map[(# * s) &, StarLatticeCoordinates[n], {4}]

StarLatticeCoordinates[n_Integer, s_, α_] :=
 N@Map[(({{Cos@α, -Sin@α}, {Sin@α, Cos@α}} * s) .#) &, StarLatticeCoordinates[n], {4}]

StarLatticeCoordinates[n_Integer, s_, α_, {x_, y_}] :=
 N@Map[(({{Cos@α, -Sin@α}, {Sin@α, Cos@α}} * s) .# + {x, y}) &,
   StarLatticeCoordinates[n], {4}]
```

HexagonLatticeCoordinates

In[101]:=
```
Clear[HexagonLatticeCoordinates]
HexagonLatticeCoordinates::"usage" =
  "HexagonLatticeCoordinates[(n:5)] returns a list of points that
    represents a hexagon lattice. n controls the number of points
    returned. HexagonLatticeCoordinates[n,s,α,{x,y}] scales, rotates,
    and translates the lattice by s, α, and {x,y}. (Hexagon lattice is
    defined as the vertex set of the regular hexagon tiling) Example:
    Show[Graphics[{Map[Point,HexagonLatticeCoordinates[5],{4}]}],AspectRatio->
    Automatic]";
```

In[103]:=

```
HexagonLatticeCoordinates[n_Integer : 5] :=
 N@Map[Table[{Cos@t, Sin@t} + #, {t, 2 * Pi / 12, 2 * Pi / 12 + Pi - 2 * Pi / 6 / 2, 2 * Pi / 6}] &,
   TriangleLatticeCoordinates[n, Sqrt[3], 0], {3}]

HexagonLatticeCoordinates[n_Integer, s_] :=
 N@Map[(# * s) &, HexagonLatticeCoordinates[n], {4}]

HexagonLatticeCoordinates[n_Integer, s_, α_] := N@Map[
   (({{Cos@α, -Sin@α}, {Sin@α, Cos@α}} * s) .#) &, HexagonLatticeCoordinates[n], {4}]

HexagonLatticeCoordinates[n_Integer, s_, α_, {x_, y_}] :=
 N@Map[(({{Cos@α, -Sin@α}, {Sin@α, Cos@α}} * s) .# + {x, y}) &,
   HexagonLatticeCoordinates[n], {4}]
```

Motif Generating Functions

Motif Generating Functions

Motif generating functions are those with names ending in "Motif" (e.g. PolygonMotif). They return a design. They are to be mapped to lattice points to generate a wallpaper design.

In[48]:=

```
? *Motif
```

Out[48]=

```
PlaneTiling`
ArrowMotif            PolygonMotif            StainedGlassMotif
DoubleLineMotif       RosetteMotif            StarMotif
NestedPolygonMotif    RotationSymbolMotif     VectorMotif
```

In[49]:=

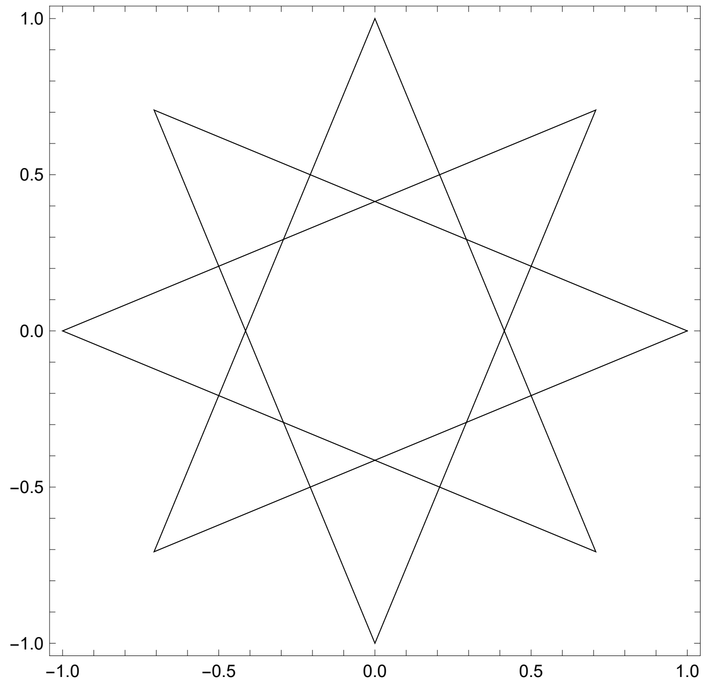
```
? PolygonMotif
```

Out[49]=

Symbol

PolygonMotif[p] returns {Line[...]} that represents a regular p–gon of radius 1. PolygonMotif[{p,q}] gives star polygon with Schlafli notation {p/q}. If p and q are not coprime, then GCD[p,q] number of a reduced p/q polygrams are produced. PolygonMotif[{p,q},s,α,{x,y}] scales, rotates and translates it by s, α, and {x,y}. The return value has Length GCD[p,q]. See also: PolygonMotif2, NestedPolygonMotif, StarMotif. Example: Show[Graphics[{PolygonMotif[{8,3}]}],AspectRatio–>Automatic]

```
In[50]:= Show[Graphics[{N[PolygonMotif[{8, 3}]]}], AspectRatio → Automatic, Frame → True]
Out[50]=
```

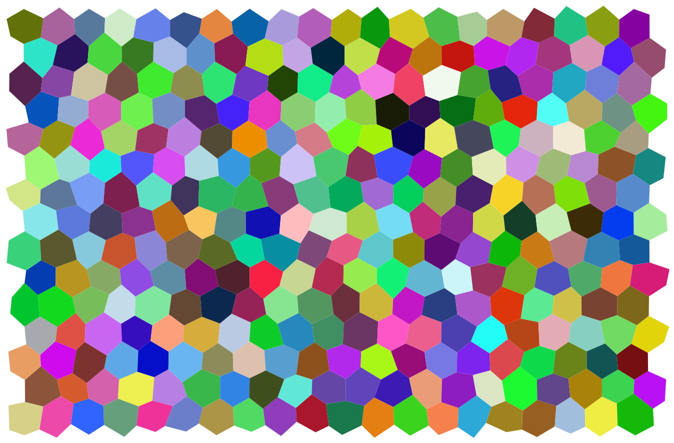


```
In[51]:= ? StainedGlassMotif
Out[51]=
```

Symbol

StainedGlassMotif[n,sideLength,randomParameter,{i,j}] returns a j by i matrix of random Polygon graphics primitives that represents a stained glass. The pattern is based on regular tiling of triangle, square, or hexagon of sideLength, depending on whether n is 3, 4, or 6. randomParameter is a real number between 0 and 1 that controls the randomness of the image. StainedGlassMotif[n,sl,rnd,{{xMin,xMax},{yMin,yMax}}] returns graphics that covers a specified rectangle. Example: Show[Graphics[{StainedGlassMotif[6,1,.5,{5,4}]/.poly_Polygon:>{RGBColor[Random[],Random[],Random[]],poly}}], AspectRatio->Automatic]

```
In[52]:= Show[Graphics[{StainedGlassMotif[6, 1, 0.5`, {20, 15}] /.p_Polygon :→
              {RGBColor[RandomReal[], RandomReal[], RandomReal[]], p}}], AspectRatio → Automatic]
Out[52]=
```



Examples of mapping a motif to a lattice

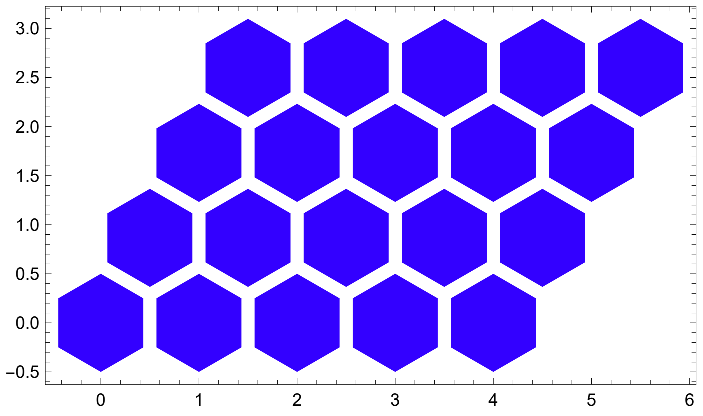
Mapping regular hexagons to a triangular grid, creating a honeycomb wallpaper design.

In[56]:= `Clear[gp]`

$$\texttt{gp = Map}\left[\texttt{PolygonMotif2}\left[6,\ 0.5\text{`}\times 1,\ \frac{2\pi}{12},\ \#1\right]\ \&,\right.$$

$$\left.\texttt{LatticeCoordinates}\left[\{1,0\},\ \frac{1}{2}\left\{1,\sqrt{3}\right\},\ \{5,4\}\right],\ \{2\}\right];$$

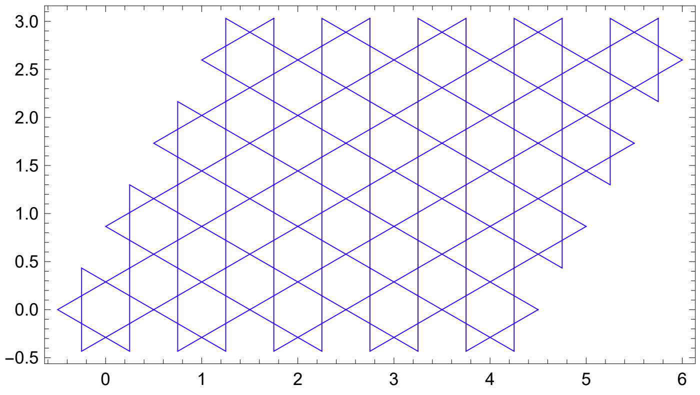
```
Show[Graphics[{Hue[0.7`], gp}], AspectRatio → Automatic, Frame → True]
Out[58]=
```



Mapping a star polygon {6,2} (aka hexagram, Star of David, Magen David, Solomon's seal) onto a triangular lattice creates a {3,6,3,6} tiling.

In[59]:=
```
Clear[gp]
gp = Map[N[PolygonMotif[{6, 2}, 0.5`, 0, #1]] &,
    LatticeCoordinates[{1, 0}, 1/2 {1, √3}, {5, 4}], {2}];
Show[Graphics[{Hue[0.7`], gp}], AspectRatio → Automatic, Frame → True]
```

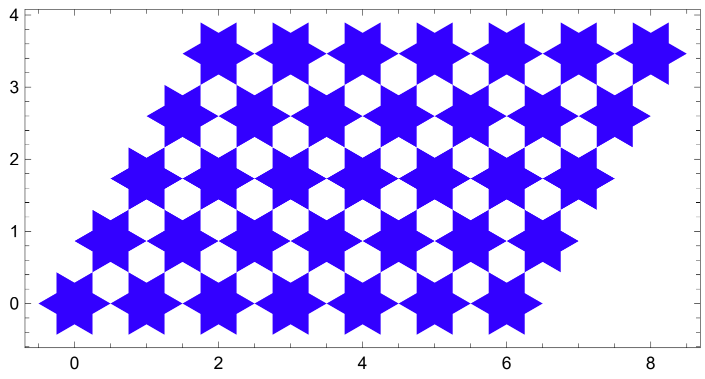
Out[61]=



Mapping a regular hexagram without the inner lines onto a triangular lattice.

In[80]:=
```
Clear[gp]
gp = Map[N[StarMotif[6, {1/2, 1/(2 √3)}, 1, 0, #1]] &,
    LatticeCoordinates[{1, 0}, 1/2 {1, √3}, {7, 5}], {2}];
Show[Graphics[{Hue[0.7`], gp}], AspectRatio → Automatic, Frame → True]
```

Out[82]=



We can also map 3 dimentional tiles to a lattice. The package does not currently contain functions that return 3-dimentional tiles. Here we define a 3 dimentional stile star3DMotif as an example.

In[53]:=
```
Clear[star3DMotif]
star3DMotif::"usage" =
  "star3DMotif[n,h,{r1,r2..}] returns a List of Polygon primitives that
    represents a pyramid of height h with a star shaped base of n-fold
    symmetry and radii r1, r2,...etc. star3DMotif[n,h,{{r1,θ1},{r2,θ2}..}]
    uses polar coordinates as vertexes. star3DMotif[n,{...},s,α,{x,y}]
    scales, rotates, and translates the base by s, α, and {x,y}. Example:
    Show[Graphics3D[{star3DMotif[5,.5,{3,1}]}],AspectRatio->Automatic]";
```

In[55]:=
```
star3DMotif[n_Integer, h_, li_?VectorQ] := star3DMotif[n, h, li, 1, 0, {0, 0}]

star3DMotif[n_Integer, h_, li_?VectorQ, s_] := star3DMotif[n, h, li, s, 0, {0, 0}]

star3DMotif[n_Integer, h_, li_?VectorQ, s_, α_] := star3DMotif[n, h, li, s, α, {0, 0}]

star3DMotif[n_Integer, h_, li_?MatrixQ] := star3DMotif[n, h, li, 1, 0, {0, 0}]

star3DMotif[n_Integer, h_, li_?MatrixQ, s_] := star3DMotif[n, h, li, s, 0, {0, 0}]

star3DMotif[n_Integer, h_, li_?MatrixQ, s_, α_] := star3DMotif[n, h, li, s, α, {0, 0}]

star3DMotif[n_Integer, h_, radii_?VectorQ, s_, α_, {x_, y_}] :=
 star3DMotif[n, h, Transpose[
   {radii, Table[2 * Pi / n / Length@radii * i, {i, 0, Length@radii - 1}]}], s, α, {x, y}]

star3DMotif[n_Integer, h_, coords_?MatrixQ, s_, α_, {x_, y_}] :=
 (Map[Polygon[{{x, y, h}, Sequence @@ #} /. {}] &, #] &) @
  (Partition[#, 2, 1] &) @ ((# /. {frst_, rst___} → {frst, rst, frst}) &) @
    (Flatten[#, 1] &) @Transpose@Map[N@Table[{Cos[t + Last@#], Sin[t + Last@#], 0} *
            First[#] * s + {x, y, 0}, {t, α, 2 * Pi + α - 2 * Pi / n / 2, 2 * Pi / n}] &, N@coords]
```

Now map the 3-dimentional tile to a triangular lattice.

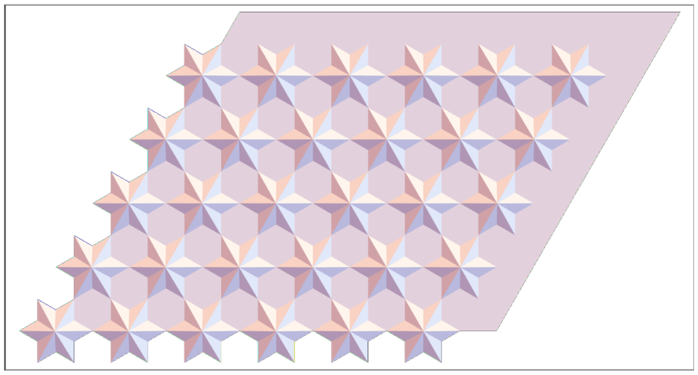
```
m = 6; n = 5;
gp = Map[star3DMotif[6, .1, {.5, Sqrt[3] / 3 / 2}, 1, 0, #] &,
   LatticeCoordinates[{1, 0}, {1 / 2, Sqrt[3] / 2}, {m, n}], {2}];
```

In[107]:=

```
Show[Graphics3D[
  {EdgeForm[], gp, Polygon[{{0, 0, 0}, {m, 0, 0}, {m + n/2, √3 n/2, 0}, {n/2, √3 n/2, 0}}]}],
 ViewPoint → {0, 0, 3}, LightSources → {{{1, 1, 0.7`}, Hue[0]}}]
```

Out[107]=

```
Show[Graphics3D[
  {EdgeForm[], gp, Polygon[{{0, 0, 0}, {m, 0, 0}, {m + n/2, √3 n/2, 0}, {n/2, √3 n/2, 0}}]}],
 ViewPoint → {0, 0, 3}, LightSources → {{{1, 1, 0.7`}, Hue[0]}}]
```



Transformation on Edges of Tilings

LineTransform

Another idea of wallpaper design is to start with a given tiling, then apply a transformation to the edges of the tiling. The function LineTransform transforms a Line primitive by a given rule.

```
? LineTransform
```

LineTransform[graphics1,replacementList] is an L–system like graphical function. graphics1 is a list of Line or an Graphics object. replacementList is a List of Line or graphic directives and primitives. Each Line in graphics1 is replaced by the relation Line[{{0,0},{1,0}}]–>replacementList. LineTransform[graphics1,replacementList,Line[{p1,p2}]] will use the relation Line[{p1,p2}]–>replacementList. Example: LineTransform[Line[{{0,0},{1,0},{1,–1}}],{Line[{{0,0},{1/2,1/2},{1,0}}]}]

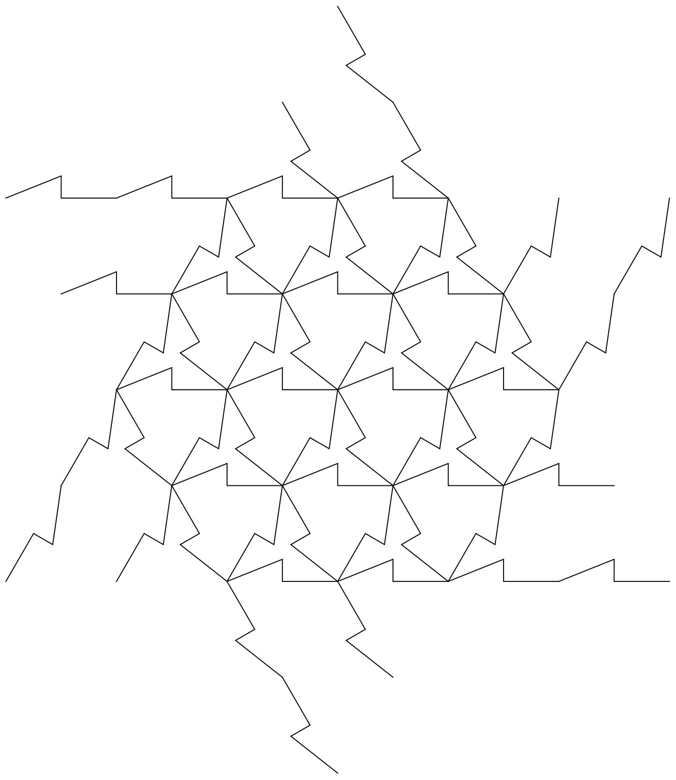
```
In[108]:=
LineTransform[Line[{{0, 0}, {1, 0}, {1, -1}}], {Line[{{0, 0}, {1 / 2, 1 / 2}, {1, 0}}]}]
Out[108]=
{{Line[{{0., 0.}, {0.5, 0.5}, {1., 0.}}]}, {Line[{{1., 0.}, {1.5, -0.5}, {1., -1.}}]}}
```

Here we replace the directed edges of a triangle tiling by a zig-zap pattern.

In[109]:=

```
Show[Graphics[{LineTransform[DirectedLatticeNetwork[3, 2, {{True}}],
    {Line[{{0, 0}, {1/2, 1/5}, {1/2, 0}, {1, 0}}]}]}], AspectRatio → Automatic]
```

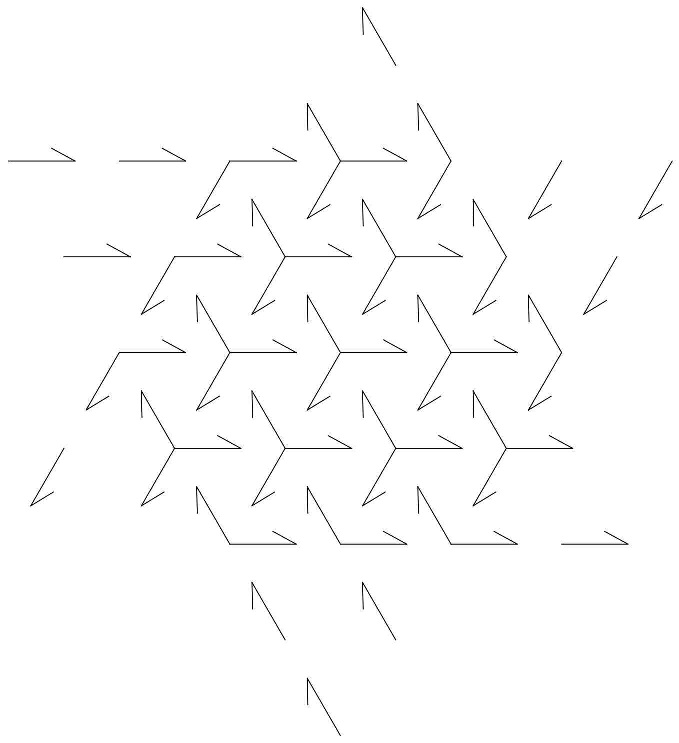
Out[109]=



Directed Network

Here we starts with a triangular lattice with directed line segments, then we replace each segment by an arrowed motif.

```
In[110]:=
        Show[Graphics[{LineTransform[DirectedLatticeNetwork[3, 2, {{True}}],
            ArrowMotif[{{0.5`, 0.4`}}, 0.6`, 0]]}], AspectRatio → Automatic]
Out[110]=
```



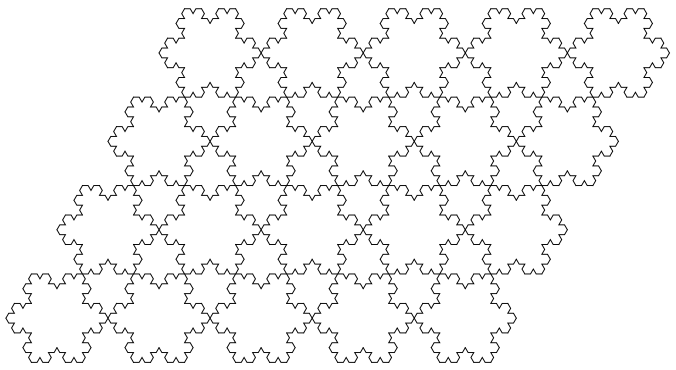
Kosh snowflake tiling

In this example, we map regular hexagons to a triangular lattice, creating a {3,6,3,6} tiling. Then we transform each line segment using a rule same to that of generating the Kosh snowflake. The result is a tiling with Kosh snowflak as prototile.

In[111]:=

```
Clear[triangularGrid, baseTilingGP, replaceRule, gp];
triangularGrid = LatticeCoordinates[{1, 0}, 1/2 {1, √3}, {5, 4}];
baseTilingGP = Map[PolygonMotif[6, 0.5`, 0, #1] &, triangularGrid, {2}];
replaceRule = Line[{{0, 0}, {1/3, 0}, {1/2, √3/6}, {2/3, 0}, {1, 0}}];
gp = Nest[LineTransform[#1, replaceRule] &, baseTilingGP, 2];
Show[Graphics[{gp}], AspectRatio → Automatic]
```

Out[116]=



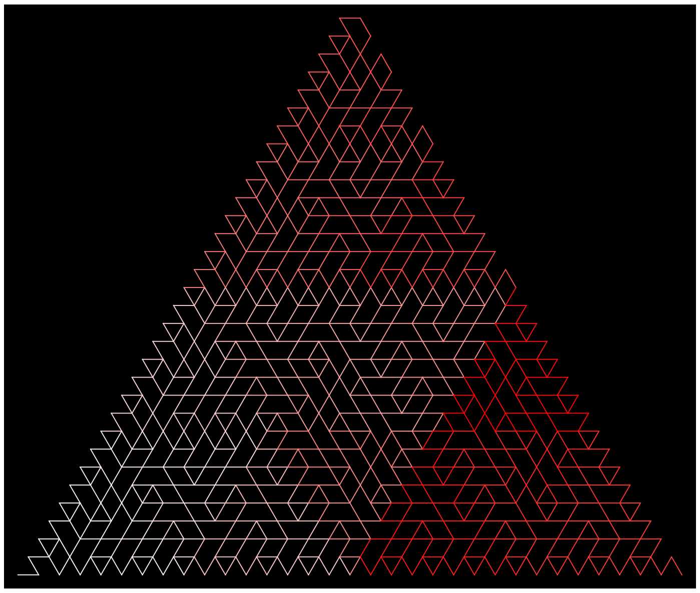
L-systems

Here is an example of repreatedly applying a rule to an image, starting with a single line. LineTransform is essentially a graphics engine for L-systems.

In[117]:=

```
Clear[initialImage, replaceRule, gp, x, y];
x = N[{1, 0}];
y = N[{Cos[2π/6], Sin[2π/6]}];
replaceRule = N[{Line[{0 x, x}], Line[{y, x}], Line[{y, x + y}], Line[{2 x, x + y}]}];
initialImage = Line[{0 x, 2 x}];
gp = Flatten[Nest[LineTransform[#1, replaceRule, Line[{0 x, 2 x}]] &, initialImage, 5]];
gp = Transpose[{Table[Hue[0, i, 1], {i, 0, 1, 1/(Length[gp] - 1)}], gp}];
Show[Graphics[{gp}], AspectRatio → Automatic, Background → GrayLevel[0]]
```

Out[124]=



Graphics Cutting functions

There are a set of functions related to cutting graphics. They are: SlitLine, CutLine, SliceLine. CutPolygon. SlicePolygon. Cut2DGraphics, Slice2DGraphics, CookieStamp2DGraphics. Wireframe2DGraphics. ConvertCircleToLine. Each one is documented on-line.

```
? CutPolygon
```

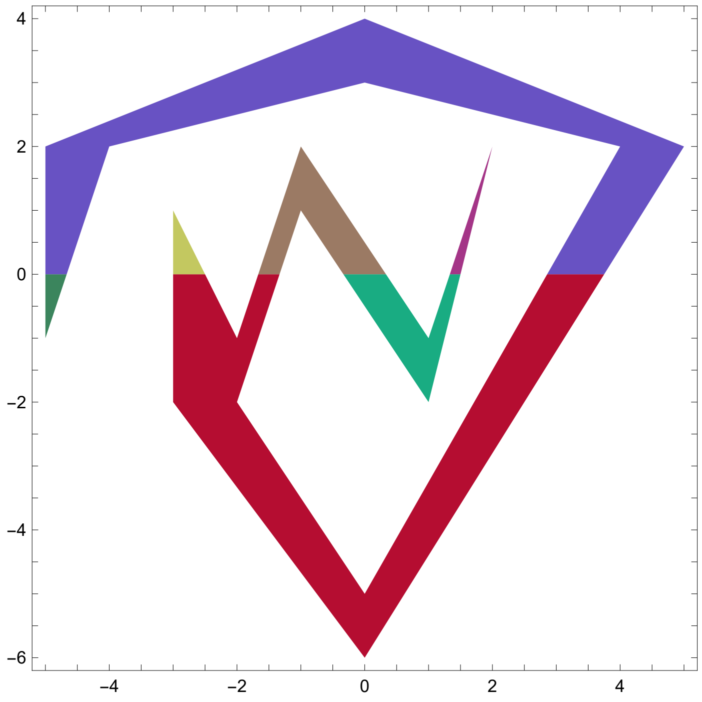
CutPolygon[Polygon[{P1,P2,..}],{A,B}] will cut polygon by the line AB. The polygon
can be nonconvex but must not cross itself. The result polygons have the same orientation as
the original. See also: SlicePolygon. Example: CutPolygon[Polygon[{{0,0},{1,0},{1,1}}],{{1/2,0},{1/2,1}}]

Here's a simple example of using CutPolygon. A non-convex polygon is cut through the line from {0,0} to {1,0}, resulting 7 polygons.

In[125]:=

```
Show[Graphics[
  {CutPolygon[Polygon[{{-2, -2}, {-1, 1}, {1, -2}, {2, 2}, {1, -1}, {-1, 2}, {-2, -1},
        {-3, 1}, {-3, -2}, {0, -6}, {5, 2}, {0, 4}, {-5, 2}, {-5, -1},
        {-4, 2}, {0, 3}, {4, 2}, {0, -5}}], {{0, 0}, {1, 0}}] /.
    poly_Polygon :→ {RGBColor[RandomReal[], RandomReal[], RandomReal[]], poly}}],
 AspectRatio → Automatic, Frame → True]
```

Out[125]=

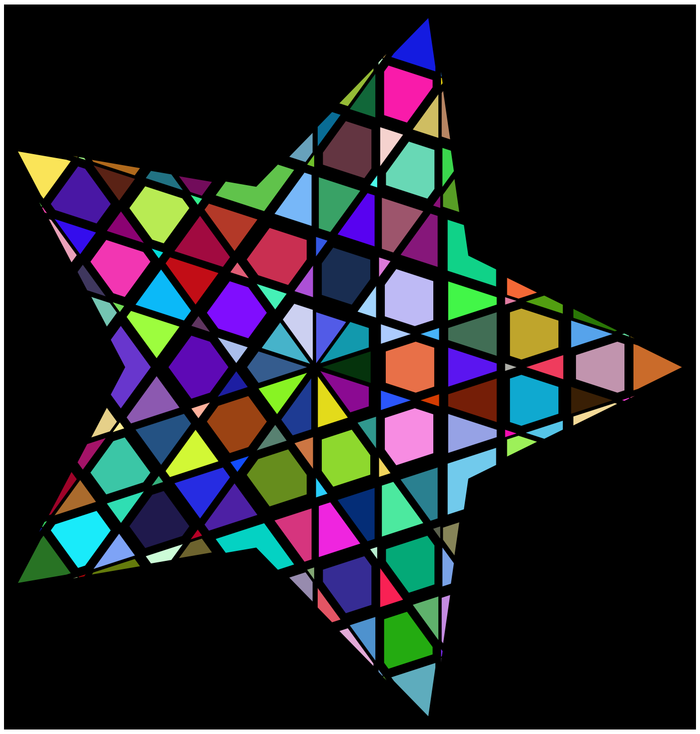


Here's a real example of using CutPolygon. We start with a 5-pointed star polygon, then cut it with five parallel lines repeatedly in 5 directions, then color the result polygons randomly.

In[127]:=

```
Clear[n, m, cuttingLines, gp];
(*n is the symmetry number *)
n = 5;

(*m is the number of parallel lines*)
m = 6;

(*this generates endpoints of m parallel lines*)
cuttingLines = Transpose[
   {Table[{i, 0}, {i, 0, 1 - (1 / m / 2), 1 / m}], Table[{i, 1}, {i, 0, 1 - (1 / m / 2), 1 / m}]}];

(*The parallel lines are rotated n times, resulting m*n number of parallel lines.*)
cuttingLines = (Flatten[#, 1] &)@
   (Table[N@Map[({{Cos@#, -Sin@#}, {Sin@#, Cos@#}} &@i).# &, cuttingLines, {2}],
     {i, 0, 2 * Pi - (2 * Pi / n) / 2, 2 * Pi / n}]);

(*this is our starting polygon*)
gp = (StarMotif[n, {1, .5}]);

(*the polygon is cut repreatedly by the cuttingLines we've setup.*)
gp = Fold[(#1 /. poly_Polygon :→ CutPolygon[poly, #2]) &, gp, cuttingLines];

(*separate the polygons a bit apart, by translate them outwards*)
gp = gp /. Polygon[pts_] :→ Polygon@((((Plus @@ pts) / (Length@pts)) * .3 + #) & /@ pts);
```

In[135]:=
```
Show[Graphics[{Flatten[gp] /.
    poly_Polygon :→ {RGBColor[RandomReal[], RandomReal[], RandomReal[]], poly}}],
 AspectRatio → Automatic, Background → GrayLevel[0]]
```
Out[135]=



Note: *Mathematica* v.2.2 contains a bug on graphics object that contains many nested braces. For example, the following generate errors.
Show[Graphics[{Nest[List,Point@{0,0},50]}]]
The workaround is to use Flatten to get rid of deep nested lists. Recursive use of CutPolygon and other graphics cutting functions will generate expressions with large depth.

Graphics Transformation functions

Several functions are provided that transforms Graphics object. Their names ends with 2DGraphics. Here a complete list.

In[136]:=
```
? *2DGraphics
```
Out[136]=

```
PlaneTiling`

CookieStamp2DGraphics    Reflect2DGraphics    Transform2DGraphics
Cut2DGraphics            Rotate2DGraphics     Translate2DGraphics
GlideReflect2DGraphics   Slice2DGraphics      Wireframe2DGraphics
```

In[137]:=

```
? Rotate2DGraphics
```

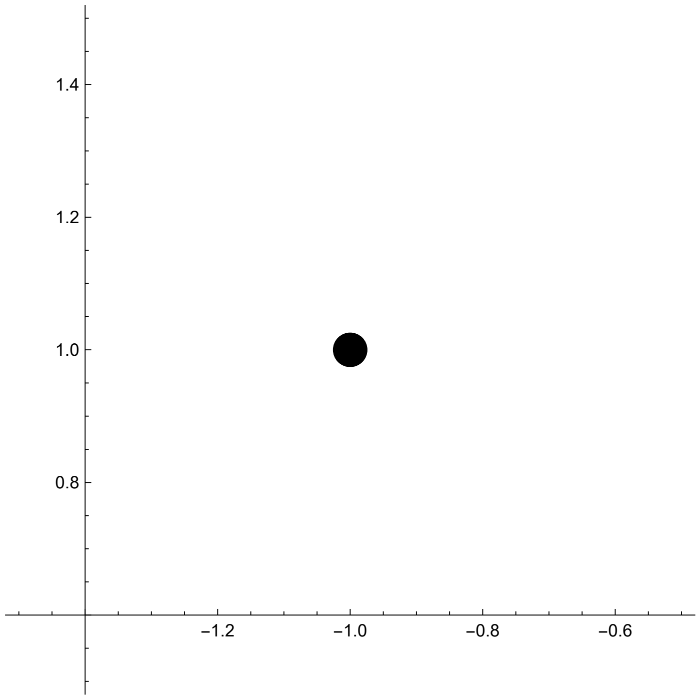
Out[137]=

```
Symbol

Rotate2DGraphics[{c1,c2},α][graphics1] rotates graphics1 by α around {c1,c2}. Example:

    Show[Graphics@Rotate2DGraphics[{0,0},1][Point[{1,0}]]],AspectRatio->Automatic,Axes->True]
```

In[138]:=

```
Show[Graphics[Rotate2DGraphics[{0, 0}, π/2][{PointSize[0.05`], Point[{1, 1}]}]],
 AspectRatio → Automatic, Axes → True]
```

Out[138]=



Triangle related functions

There are several functions that calculate triangle properties.

In[139]:=
```
? *Triangle*
```
Out[139]=
```
System`
AASTriangle          NotRightTriangle          Triangle
ASATriangle          NotRightTriangleBar       TriangleCenter
LeftTriangle         NotRightTriangleEqual     TriangleConstruct
LeftTriangleBar      RightTriangle             TriangleMeasurement
LeftTriangleEqual    RightTriangleBar          TriangleWave
NotLeftTriangle      RightTriangleEqual        UnitTriangle
NotLeftTriangleBar   SASTriangle
NotLeftTriangleEqual SSSTriangle

Global`
TriangleLatticeCoordinates

PlaneTiling`
TriangleCentroid                TriangleInscribedCircle
TriangleCircumcenter            TriangleOrthocenter
TriangleCircumscribedCircle     TriangleSignedArea
TriangleIncenter
```

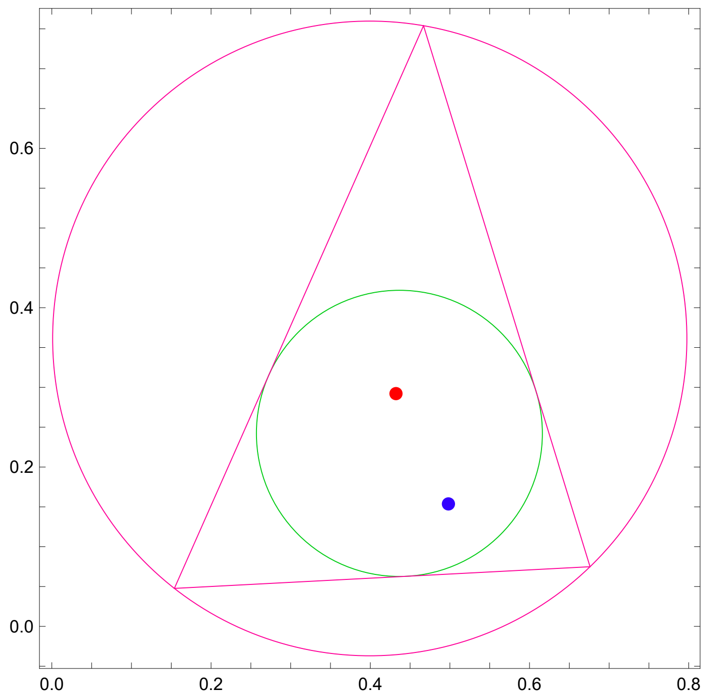
In[140]:=
```
? TriangleInscribedCircle
```
Out[140]=
```
Symbol
TriangleInscribedCircle[{pt1,pt2,pt3}] returns the the inscribed
    circle of triangle {pt1,pt2,pt3}. Example: TriangleInscribedCircle[{{0,0},{1,0},{1,1}}]
```

This example generates a random triangle, and draw its centroid, orthocenter, inscribed circle, and circumscribed circle.

In[149]:=

```
Clear[triangle, triangleGP];
triangle = Table[RandomReal[{0, 1}, 2], {3}];
triangleGP = Line[triangle /.{a_, b_, c_} :→ {a, b, c, a}];
Show[Graphics[{PointSize[0.02`], Hue[0], Point[TriangleCentroid[triangle]],
   Hue[0.7`], Point[TriangleOrthocenter[triangle]], Hue[0.35`, 1, 0.8`],
   TriangleInscribedCircle[triangle], Hue[0.9`], TriangleCircumscribedCircle[triangle],
   triangleGP}], AspectRatio → Automatic, Frame → True]
```

Out[152]=

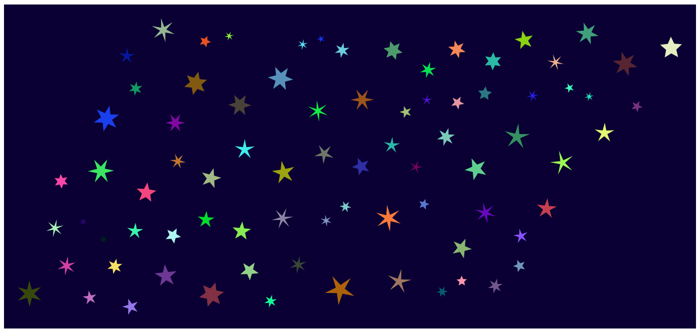


This example, a random triangle network is generated by StainedGlassMotif. Each triangle is replaced by an inscribed circle using TriangleInscribedCircle, and the circle is replaced by a random star using StarMotif.

In[153]:=

```
Clear[gp];
gp = StainedGlassMotif[3, 1, 1, {8, 5}] /.
   Polygon[{a_, rest__}] :→ {TriangleInscribedCircle[{a, rest}] /.
      Circle[c_, r_] :→ StarMotif[RandomInteger[{5, 6}],
        {1, RandomReal[{0.2`, 0.5`}]}, r 0.5`, RandomReal[{0, N[2 π]}], c]};
```

```
In[155]:=
Show[Graphics[
   {gp /.poly_Polygon :→ {RGBColor[RandomReal[], RandomReal[], RandomReal[]], poly}}],
  AspectRatio → Automatic, Background → Hue[0.7`, 1, 0.2`]]
Out[155]=
```



Example of Wallpaper Designs

Here are few examples of using PlaneTiling package. An example notebook that contain high quality wallpaper design is in the works.

Using techniques of mapping a motif to lattice points

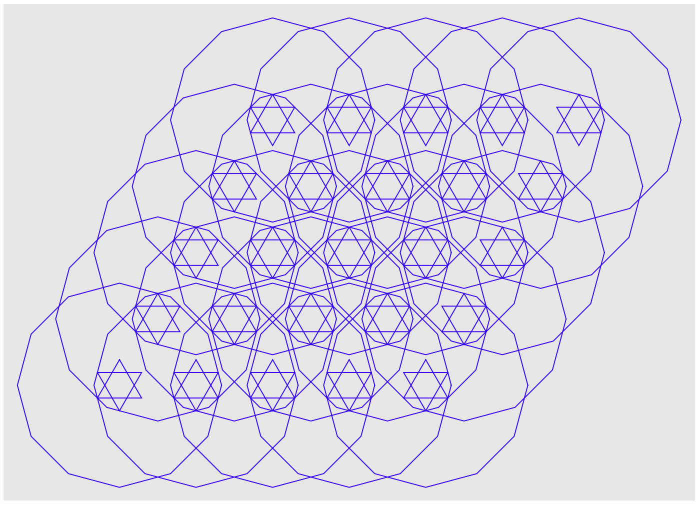
Islamic pattern. Star of David.

The motif is a hexagram and a 12-sided polygon combined. Map the motif to a triangular grid. An effect of weaving stands out.

In[156]:=

```
Clear[motifGP, gp];
motifGP =
  N[{Hue[0.7`], PolygonMotif[{6, 2}, 1/3, 30 °], Hue[0.7`], PolygonMotif[{12, 1}, 4/3]}];
gp = Map[Translate2DGraphics[#1][motifGP] &,
   LatticeCoordinates[{1, 0}, {1/2, √3/2}, {5, 5}], {2}];
Show[Graphics[{gp}], AspectRatio → Automatic, Background → GrayLevel[0.9`]]
```

Out[159]=



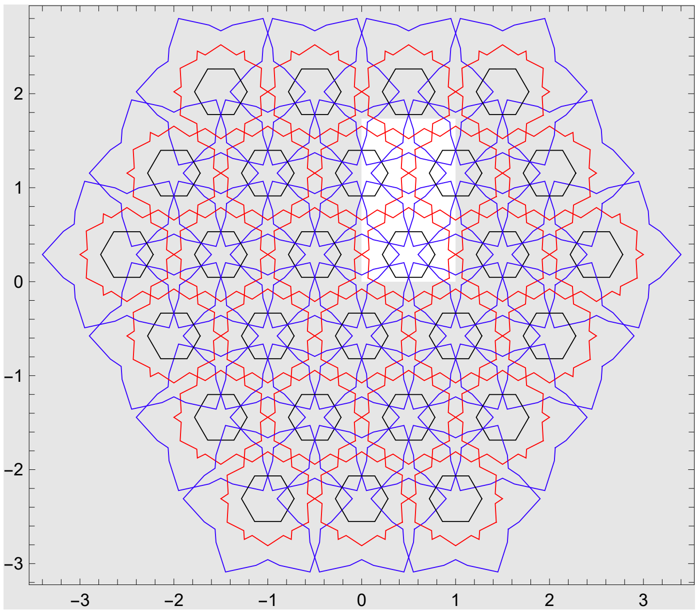
Islamic pattern.

In this example, the motif is made of three polygons with central symmetry. The coordinates of the polygons are carefully picked. Mappnig the resulting motif to a triangular lattice results an Islamic pattern. [Abas and Salman], p.162.

In[160]:=

```
Clear[r, r2, coords, coords2, motif, gp];
r = 0.5`;
coords = {{r, 0}, {0.85` r, 7 °}, {r, 30 °}};
coords = Join[Reverse[({First[#1], -Last[#1]} &) /@ coords], Rest[coords]];
r2 = 0.9`;
coords2 = {{r2, 0}, {0.8` r2, 20 °}, {0.72` r2, 30 °}};
coords2 = Join[Reverse[({First[#1], -Last[#1]} &) /@ coords2], Rest[coords2]];
motif = {N[PolygonMotif[6, 1.12`/4, 0, #1]], Hue[0], Line @@ N[StarMotif[6, coords, 1, 0, #1]],
    Hue[0.7`], Line @@ N[StarMotif[6, coords2, 1, 0, #1]]} &;
gp = Map[motif, TriangleLatticeCoordinates[3], {3}];
Show[Graphics[{GrayLevel[1], Rectangle[{0, 0}, {1, √3}], Hue[0, 0, 0], gp}],
 AspectRatio → Automatic, Frame → True, Background → GrayLevel[0.9`]]
```

Out[169]=



Using techiques of giving Wallpaper plot a motif in fundamental region, and specfying a symmetry.

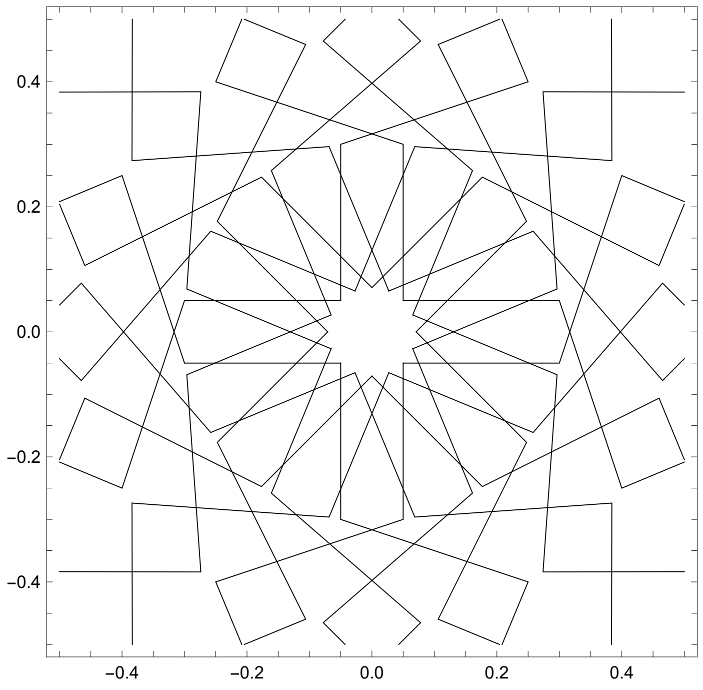
An Islamic pattern. Typical 16-fold star in a square.

Start with a zigzagging line. Use RosetteMotif to trace rotate it to get a n-fold star-like pattern. Then, apply CookieStamp2DGraphics to the graphics so that the boundary is a square. Map the rusult to lattice points to obtain a typical Islamic pattern.

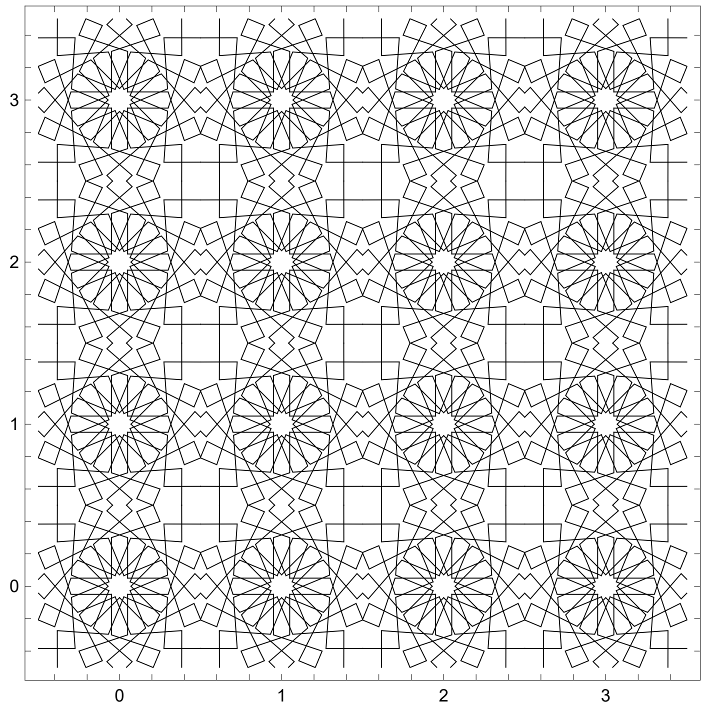
In[170]:=

```
Clear[motifGP, wallpaperGP];
motifGP = CookieStamp2DGraphics[N[
    RosetteMotif[Line[{{0.05`, 0.05`}, {0.3`, 0.05`}, {0.4`, -0.25`}, {1, 0}}], 16, True]],
   Table[1/2 {Cos[t], Sin[t]} √2, {t, 0 + 2π/8, 2π + 2π/8 - 2π/(4×2), 2π/4}]];
wallpaperGP = Map[Translate2DGraphics[#1][motifGP] &,
   LatticeCoordinates[{1, 0}, {0, 1}, {4, 4}], {2}];
Show[Graphics[{motifGP}], AspectRatio → Automatic, Frame → True]
Show[Graphics[{wallpaperGP}], AspectRatio → Automatic, Frame → True]
```

Out[173]=



Out[174]=



Technique of applying a transformation to a tiling recursively.

Given a tiling, we can apply a transformation rule recursively to the edges of teh tile. The result has modern fractal-like pattern.

Modern. Fractal hexagonal net.

Start with a regular tiling by triangles.

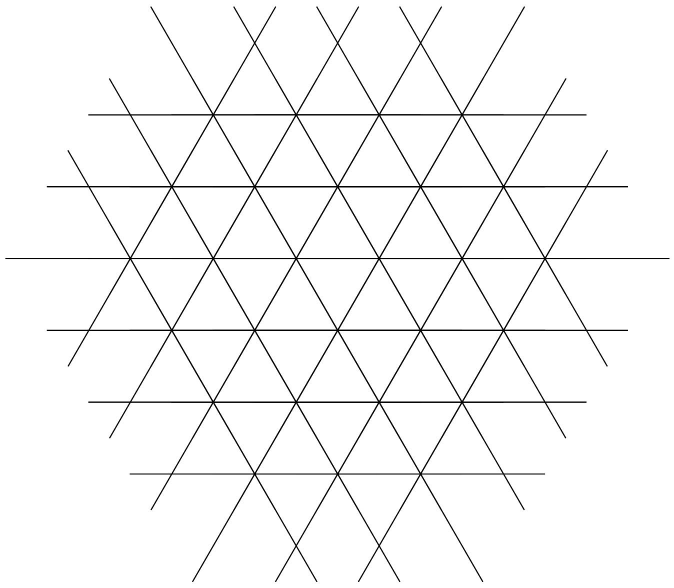
In[175]:=

```
Clear[replaceRule, gp, gp2, gp3];
gp = Map[N@PolygonMotif[{6, 3}, 1.5, 2 * Pi / 6, #] &, TriangleLatticeCoordinates[3], {3}];
```

In[177]:=

```
Show[Graphics[gp], AspectRatio → Automatic]
```

Out[177]=



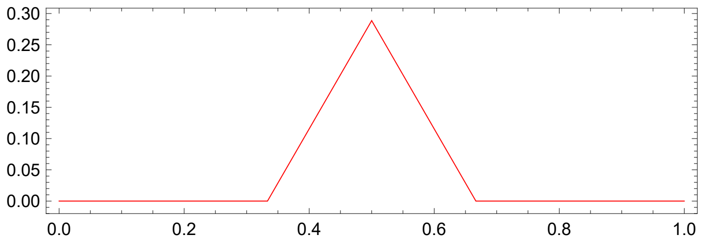
Define a line replacement rule similar to that generating the Kosh-snowflake.

In[178]:=

$$\text{replaceRule = Line}\left[\left\{\{0, 0\}, \left\{\frac{1}{3}, 0\right\}, \left\{\frac{1}{2}, \frac{\sqrt{3}}{6}\right\}, \left\{\frac{2}{3}, 0\right\}, \{1, 0\}\right\}\right];$$

```
Show[Graphics[{Hue[0], replaceRule}, AspectRatio → Automatic, Frame → True]]
```

Out[179]=



Apply the rule to the tiling recursively 3 times results a fractal style tiling.

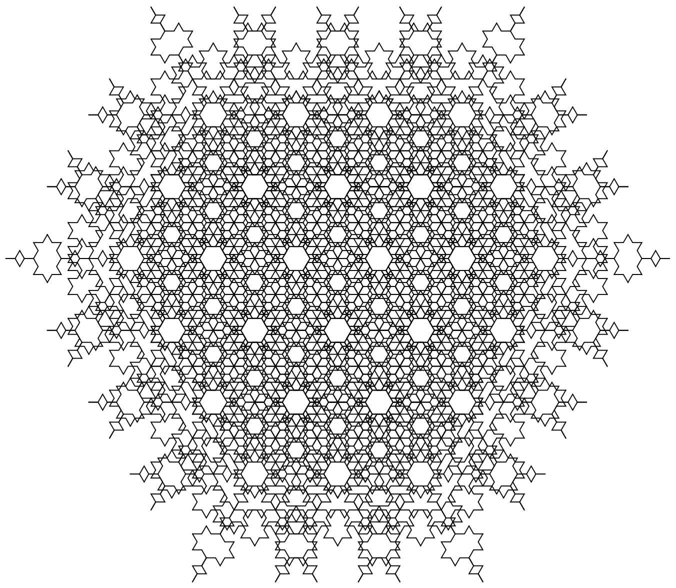
In[180]:=

```
gp2 = Nest[LineTransform[#, replaceRule] &, gp, 3];
```

In[181]:=

```
Show[Graphics[gp2], AspectRatio → Automatic]
```

Out[181]=



The boundary of the tiling is ugly. We can trim it using CookieStamp2DGraphics. Using Table to generate a hexagon outline as our cookie cutter.

In[182]:=

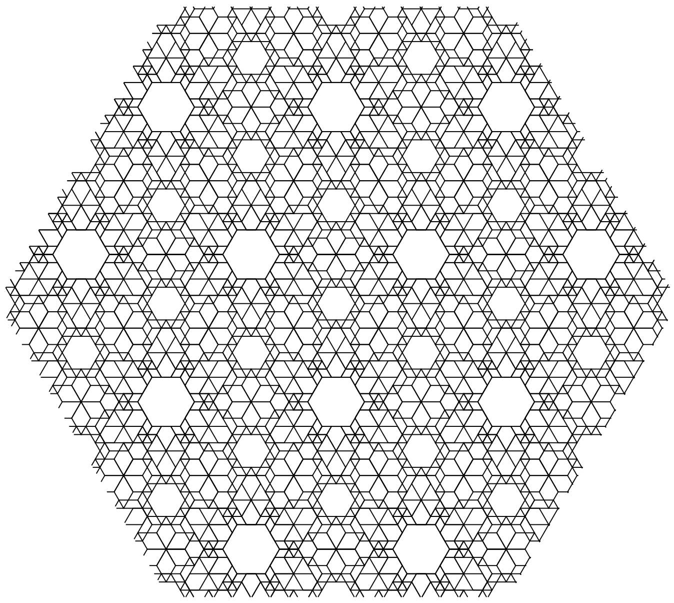
```
gp2 = CookieStamp2DGraphics[gp2,
   Table[{Cos@t, Sin@t} * 2 + 0.01, {t, 0, 2 * Pi - 2 * Pi / 6 / 2, 2 * Pi / 6}]];
```

```
CompiledFunction: Numerical error encountered; proceeding with uncompiled evaluation.

CompiledFunction: Numerical error encountered; proceeding with uncompiled evaluation.
```

In[185]:=

```
Show[Graphics[gp2], AspectRatio → Automatic]
```

Out[185]=



A wallpaper design palette.

Here we define a function wallpaperDesignPlot[symmetryID,motifGP,basisVectors,plotPoints] that will plot in one shot the wallpaper's symmetry diagram, fundamental region, unit cell, lattice, rectangular unit cell, and the wallpaper design itself. This would make it convenient when designing wallpapers.

Definitions

In[ ]:=
```
Clear[rectangularUnitCellRange];
rectangularUnitCellRange::usage =
  "rectangularUnitCellRange[{a1,a2},{b1,b2}] returns a range in the form
    {{0,xMax},{0,yMax}}, where xMax and yMax is the minimum lengths of the
    horizontal and vertical periodicy of the parallelogram tiling generated
    by the vectors {a1,a2} and {b1,b2}. That is, it is a rectangular unit
    cell. This rectangle does not always exists for all basis vectors. The
    essense of a proof is that irrational slope exists. However, for practical
    purposes, an approximate rectangle works in all cases. If no basis vectors
    lie on the axis, and they are not symmetric to an axis, then {{0,0},{0,0}}
    is returned. Example: rectangularUnitCellRange[{1,0},{1/2,Sqrt[3]/2}]";

rectangularUnitCellRange[a : {_, _}, b : {_, _}] := Module[{c, aLength, bLength,
    diagonalLength1, diagonalLength2, θ, oneBasisVectorParallelToAxisQ,
    basisVectorsAreSymmetricToAxisQ, rectangularQ, triangularQ},
       aLength = Sqrt[Plus @@ (a^2)];
       bLength = Sqrt[Plus @@ (b^2)];
       c = a + b;
       diagonalLength1 = Sqrt[Plus @@ (c^2)];
       diagonalLength2 = Sqrt[Plus @@ ((b - a)^2)];
       θ = ArcCos[(a.b) / (aLength * bLength)];
       oneBasisVectorParallelToAxisQ = MemberQ[{a, b}, 0 | 0., {2}];
       basisVectorsAreSymmetricToAxisQ =
    TrueQ@Or[N@First@a === N@First@b, N@Last@a == N@Last@b];
       rectangularQ = TrueQ[N@θ == N@(2 * Pi / 4)];
       triangularQ = TrueQ[(N@θ == N@(2 * Pi / 6)) || (N@θ == N@(2 * Pi / 3))];

    If[basisVectorsAreSymmetricToAxisQ, {{0, diagonalLength1}, {0, diagonalLength2}},
    If[oneBasisVectorParallelToAxisQ, Which[rectangularQ,
      {{0, aLength}, {0, bLength}}, triangularQ, If[MemberQ[{a, b}, {_, 0 | 0.}, {1}],
       {{0, aLength}, {0, diagonalLength1}}, {{0, diagonalLength1}, {0, aLength}}],
      True(*scaline*), {{0, 0}, {0, 0}}], {{0, 0}, {0, 0}}]]
];
```

In[ ]:=
```
Clear[wallpaperDesignPlot];

wallpaperDesignPlot::usage =
  "wallpaperDesignPlot[symmetryID,motifGP,basisVectors,plotPoints] makes various
    plots of a wallpaper design, including its symmetry diagram, fundamental
    region, unit cell, lattice, rectangular unit cell, and the wallpaper design
    itself. This function is used when designing wallpapers. You can change
    its definition to suit your needs. This function returns Null. Example: ";
```

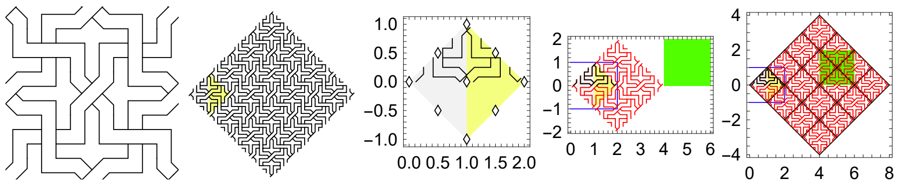
```
In[ ]:= wallpaperDesignPlot[symmetryID_Integer,
          motifGP_, basisVectors : {{_, _}, {_, _}}, plotPoints : {_, _}] :=
         Module[{symmetriesGP, latticeGP, wallpaperGP, fundamentalRegion, fundamentalRegionRange,
            fundamentalRegionArea, unitCell, unitCellRange, unitCellArea, centroid,
            rectangularUnitCellRectangle, allGroupDefaultUnitCellsMaxRange, unitCellColor,
            fundamentalRegionColor, rectangularUnitCellRectangleColor}, symmetriesGP =
            First[WallpaperPlot[symmetryID, MotifGraphics → {}, BasisVectors → basisVectors,
              PlotPoints → {1, 1}, ShowSymmetryAndUnitCellGraphics → True,
              UnitCellGraphicsFunction → ({} &), DisplayFunction → Identity]];
          latticeGP = First[WallpaperPlot[symmetryID, MotifGraphics → {}, BasisVectors →
                basisVectors, PlotPoints → plotPoints, ShowSymmetryAndUnitCellGraphics → True,
              UnitCellGraphicsFunction → (Line[Append[#1, First[#1]]] &),
              MirrorLineGraphicsFunction → ({} &), GlideReflectionGraphicsFunction → ({} &),
              RotationSymbolGraphicsFunction → ({} &), DisplayFunction → Identity]];
          wallpaperGP = First[WallpaperPlot[symmetryID, MotifGraphics → motifGP,
              BasisVectors → basisVectors, PlotPoints → plotPoints,
              ShowSymmetryAndUnitCellGraphics → False, DisplayFunction → Identity]];
          fundamentalRegion = WallpaperGroupData⟦symmetryID, 5⟧ @@ basisVectors;
          unitCell =
            Polygon[({{0, 0}, First[#1], First[#1] + Last[#1], Last[#1]} &)[basisVectors]];
          unitCellRange = ({Min[#1], Max[#1]} &) /@ Transpose @@ unitCell;
                       1
          centroid =  — Plus @@ (basisVectors plotPoints);
                       2
          rectangularUnitCellRectangle =
            Rectangle @@ Transpose[rectangularUnitCellRange @@ basisVectors + centroid];
          unitCellColor = Hue[0.18`, 0.5`, 1];
          fundamentalRegionColor = GrayLevel[0.95`];
          rectangularUnitCellRectangleColor = Hue[0.28`];
          Show[Graphics[{{unitCellColor, unitCell}, {fundamentalRegionColor, fundamentalRegion},
               {symmetriesGP}, motifGP}], AspectRatio → Automatic, Frame → True] ×
            Show[Graphics[{{unitCellColor, unitCell}, {rectangularUnitCellRectangleColor,
                 rectangularUnitCellRectangle}, {Hue[0.7`], Line[({{#1⟦1, 1⟧, #1⟦2, 1⟧},
                      {#1⟦1, 2⟧, #1⟦2, 1⟧}, {#1⟦1, 2⟧, #1⟦2, 2⟧}, {#1⟦1, 1⟧, #1⟦2, 2⟧}} &)[
                   unitCellRange]]}, {fundamentalRegionColor, fundamentalRegion}, {Hue[0],
                 wallpaperGP⟦{1, 2}, {1, 2}⟧}, motifGP}], AspectRatio → Automatic, Frame → True] ×
            Show[Graphics[{{unitCellColor, unitCell}, {rectangularUnitCellRectangleColor,
                 rectangularUnitCellRectangle}, {Hue[0.7`], Line[({{#1⟦1, 1⟧, #1⟦2, 1⟧},
                      {#1⟦1, 2⟧, #1⟦2, 1⟧}, {#1⟦1, 2⟧, #1⟦2, 2⟧}, {#1⟦1, 1⟧, #1⟦2, 2⟧}} &)[
                   unitCellRange]]}, {fundamentalRegionColor, fundamentalRegion}, {latticeGP},
               {Hue[0], wallpaperGP}, motifGP}], AspectRatio → Automatic, Frame → True] ×
            Show[Graphics[{wallpaperGP, motifGP}], AspectRatio → Automatic,
             PlotRange → rectangularUnitCellRange @@ basisVectors + centroid] ×
            Show[Graphics[{{unitCellColor, unitCell}, {fundamentalRegionColor,
                 fundamentalRegion}, wallpaperGP}], AspectRatio → Automatic]]
```

testing

```
Clear[symmetryID, motifGP, basisVectors, plotPoints]
```

```
symmetryID = 2; motifGP =
   {Line[{{3/4, 3/4}, {3/4, 7/12}, {7/12, 7/12}, {7/12, 1/4},
      {1/12*(9 + Sqrt[2]), 1/4}, {1, Sqrt[2]/12}, {1/12*(15 - Sqrt[2]), 1/4},
      {17/12, 1/4}, {17/12, 5/12}, {13/12, 5/12}, {13/12, 1/12*(11 - Sqrt[2])},
      {1, 1/12*(12 - Sqrt[2])}, {11/12, 1/12*(11 - Sqrt[2])}, {11/12, 5/12},
      {7/12, 5/12}}], Line[{{1, 1/12*(12 - Sqrt[2])},
      {1/24*(24 + Sqrt[2]), 1/24*(24 - Sqrt[2])}}],
    Line[{{5/4, 3/4}, {5/4, 7/12}, {17/12, 7/12}}],
    Line[{{5/12, 5/12}, {5/12, 1/12}, {1/12*(11 - Sqrt[2]), 1/12},
      {1/12*(12 - Sqrt[2]), 0}, {1, Sqrt[2]/12}}],
    Line[{{1/12*(12 + Sqrt[2]), 0}, {1/12*(13 + Sqrt[2]), 1/12}, {19/12, 1/12},
      {19/12, 5/12}, {17/12, 5/12}}],
    Line[{{1/4, 1/4}, {1/4, 1/12*(3 - Sqrt[2])}, {Sqrt[2]/12, 0}}],
    Line[{{7/4, 1/4}, {7/4, 1/12*(3 - Sqrt[2])}, {1/12*(24 - Sqrt[2]), 0},
      {1/24*(48 - Sqrt[2]), Sqrt[2]/24}}]}; basisVectors = {{1, 1}, {1, -1}};
  plotPoints = {4, 4};
```

```
wallpaperDesignPlot[symmetryID, motifGP, basisVectors, plotPoints]
```


1.0
0.5
0.0
−0.5
−1.0
0.0 0.5 1.0 1.5 2.0
2
1
0
−1
−2
0 1 2 3 4 5 6
4
2
0
−2
−4
0 2 4 6 8