A Project Report

On

Submitted in requirement for the course

Lab based Project (CSN-300)

Of Bachelor of Technology in Computer Science and Engineering

By

Ankita Saxena	Richa Gautam	Rinkle Jain	Silky Priya	Parul Meena
15114011	15114057	15114058	15114070	14114034

Project taken under

Dr. Ranita Biswas

Assistant Professor

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, ROORKEE

ROORKEE- 247667 (INDIA)

SUMMER, 2018

Introduction

A tessellation of a flat surface is the tiling of a plane using one or more geometric shapes, called tiles, with no overlaps and no gaps.

The original word tessellation comes from its use in art. It comes from ancient Greek word tesseres, which means four, in the reference to the squares used in the first tessellations. Tessellations were discovered and used in art thousands of years ago by many different ancient cultures long before they were studied in mathematics.

Symmetry in tessellating pattern

When drawing tessellating patterns, symmetry is the mathematical idea that can be used. Symmetry is the preservation of form and configuration across a point, a line, or a plane. Symmetry of an object in the plane is a rigid motion of the plane that leaves the object apparently unchanged. Therefore, symmetry is the ability to take a shape and match it exactly to another shape. The techniques that are used are called transformations and include translations, reflections, rotations, and glide reflections. There are several different types of symmetry, but in each type of symmetry, characteristics such as angles, side lengths, distances, shapes, and sizes are maintained. Each of the transformations mentioned above produces a different type of symmetry.

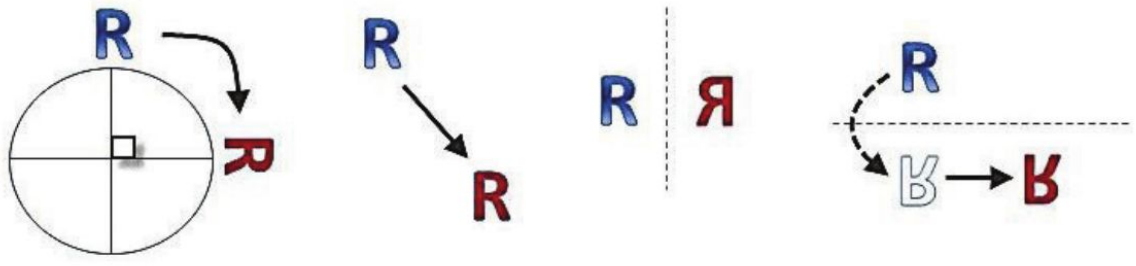


FIgure 1. Rotation Translation Reflection Glide reflection

To rotate an object means to turn it around. Every rotation has a centre and an angle. To translate an object means to move it without rotating or reflecting it.Every translation has a direction and a distance. To reflect an object means to produce its mirror image. Every reflection has a mirror line. A glide reflection combines reflection with a translation along the direction of the mirror line. Glide reflections are the only type of symmetry that involves more than one step.

Regular Tessellation

A tessellation can be created with an equilateral triangle, a square, or a regular hexagon. These are regular and congruent polygons i.e. all of sides of each polygon are the same in length and all polygons within the plane are the same in shape and size. Any given pattern in a tessellation can be continued infinitely in every direction. A regular tessellation is a highly symmetric tessellation made up of congruent regular polygons. Only a few shapes can tessellate correctly. These are equilateral triangles, squares, and regular hexagons.

So, there can only be three regular tessellations on the Euclidean plane which are made from copies of a single regular polygon meeting at each vertex. There are only three, because the inside angles of the polygon must be a factor of 360 degrees so that the polygons can line up at the points leaving no gaps.

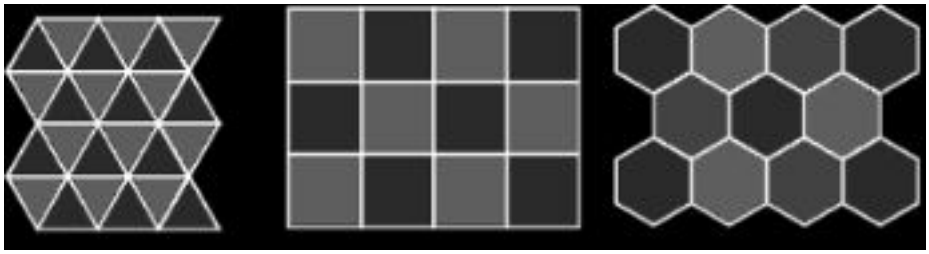


Figure 2. Regular tessellation for triangle, square and hexagon

Uniform Tiling

In geometry, a uniform tiling is a tessellation of the plane by regular polygon faces with the restriction of being vertex-transitive. Uniform tilings are listed by their vertex configuration, the sequence of faces that exist on each vertex. For example 4.8.8 means one square and two octagons on a vertex. These 11 uniform tilings have 32 different uniform colorings. A uniform coloring allows identical sided polygons at a vertex to be colored differently, while still maintaining vertex-uniformity and transformational congruence between vertices.

Figure 3. Set of eleven convex uniform tilings

Tessellations for irregular shapes

At present, there exists no technique to generate tessellations for any given figure. The goal of our project is to use the techniques for regular tessellations and uniform tilings and devise a method for generating tessellations for irregular shapes.

Figure 4. Tessellations generated from image of a penguin

Motivation for the project

Tessellations can be found in many areas of life. Art, architecture, hobbies, and many other areas hold examples of tessellations found in our everyday surroundings. Specific examples include oriental carpets, quilts, origami, Islamic architecture, etc. Algorithms exist for generating tilings of regular polygons but there is no specific methodology to generate tilings for any given figure. The motivation of the project is to devise such method for easy generation of tilings from any given figure which would find applications not only in the digital world but also in the fields of art and architecture.

Papers Referred

Escherization

This paper introduces and presents a solution to the “Escherization” problem:
Given a closed figure in the plane, find a new closed figure that is similar to the original and tiles the plane.

We begin with background on the mathematical theory of tilings, leading into a description of the “isohedral” tilings, on which the rest of this work is based. We then address each of the remaining subproblems in turn:

- parameterizing the isohedral tilings
- developing a measure of “closeness” between two tiles
- designing an optimizer for finding the best tiles
- representing the resulting tilings for efficient editing and viewing
- decorating and rendering the tiles

We end with a discussion of our results and ideas for future work.

Isohedral tilings

For two congruent tiles A and B in a tiling, there will be some rigid motion of the plane that carries one onto the other (there may in fact be several). A somewhat special case occurs when the rigid motion is also a symmetry of the tiling. In this case, when A and B are brought into correspondence, the rest of the tiling will map onto itself as well. We then say

that A and B are transitively equivalent. Transitive equivalence is an equivalence relation that partitions the tiles into transitivity classes. When a tiling has only one transitivity class, we call the tiling isohedral. More generally, a k-isohedral tiling has k transitivity classes. An isohedral tiling is one in which a single prototile can cover the entire plane through repeated application of rigid motions from the tiling's symmetry group.

Parameterizing the isohedral tilings

Besides building a generative model of isohedral tilings, we need a way to explicitly navigate the space of legal tiling polygons. For each isohedral type we need a complete and consistent parameterization of the tiling vertices for tilings of that type.

Such a set of parameterization were derived by determining angle and length constraints from the incidence symbols and parameterizing the unconstrained degrees of freedom. These easy parameterizations are balanced by tiling types with one-of-a-kind structure that can take some thought to derive.

The shape metric

We can compare two generated shapes in the form of a metric that would take two outlines and return a nonnegative real number; zero would mean that the outlines are identical, and higher positive values would denote shapes that are increasingly dissimilar.

We use the metric created by Arkin et al. for comparing polygons. Their metric represents the input polygons as turning functions, functions that map fraction of arc length in a polygon to the angle of the polygon at that point. We use the polygon comparison metric for both polygons and subdivision curves.

Optimizing over the space of tilings

Now having the set of tilings, parameterizations over those tilings and a good shape matrix, we can now address the problem of building an optimizer that can search over the space of those tilings to find an instance whose tiles are close to the goal shape.

The optimizer takes as input a goal shape and a set of isohedral families in which to search for an optimal tiling. The optimizer begins by creating a set of multiple instances of tilings from each isohedral family. It then calls a re-entrant simulated annealing procedure to improve each one of these instances. After each of the instances has been optimized to some degree, the instances are evaluated according to the shape metric, and the worst

ones are removed. The annealing is continued on the remaining instances. This iterative process of alternately pruning the search space and then improving the remaining instances is repeated until just a single tiling instance is left. This tiling is returned as the optimal tiling.

Representation of isohedral tilings

We have developed a computer representation of isohedral tilings that allows us to express our Escherization algorithm efficiently and naturally. The key is to factor out the constraints on the tile imposed by adjacencies and internal symmetries, and to store only the minimal set of free parameters that encode the tile shape. We break down the information associated with a tile into two components: the tiling template and the tile instance. The tiling template contains information about a tiling type in general. The tile instance refers to a template and contains a set of parameters for the tiling vertex parameterization, along with the minimal set of information required to reproduce the edge shapes.

Decorations and rendering

The output of the core Escherization algorithm is a geometric description of a tile, not a finished ornamental design. To complete the Escherization process, we need to surround the core algorithm with tools to add decorations to tiles and create high-quality renderings of the results. The use of various available libraries and tools gives the artist creative control over the appearance of the final tiling, and can bring the result closer to the informal hand-drawn style of Escher's notebook drawings.

Results

We have used our Escherization implementation and decoration tools to produce a number of ornamental tilings from various sources of imagery. In all cases, the optimizer generated a tile shape that was then modified slightly in the editor. The source image was warped into the tile shape, and copies of the warped image were recoloured and edited to make the final rendering.

Discussions

Most outlines are not tiles. For just about any goal shape, an Escherizer will have to produce an approximation, and a better Escherizer will produce a closer approximation. A

perfect Escherizer would determine the smallest distance over all possible tile shapes, and return the tiling that achieves that bound. Our imperfect optimizer, by contrast, coarsely samples the space of isohedral tilings in a directed fashion and returns the best sample it finds.

Conclusion

This research suggests many future directions, including generalizing our algorithms to handle multihedral and aperiodic tilings, parquet deformations, or tilings over non-Euclidean domains. Another intriguing idea is to allow some flexibility in the goal shape as well. For instance, instead of a 2D shape, we might use a 3D model and attempt to automatically discover a camera position from which the view of the model is most easily Escherized. Finally, along the lines of creating Escher tilings automatically is the problem of "automatic conventionalization": somehow creating not just the tile boundaries, but the line-art graphical decorations that go inside the tilings, more or less automatically from a reference image.

Duality

The essence of duality is two forms mutually defining each other—knowing one is enough to know the other. Two units [tiles] bordering on each other cannot simultaneously function as 'figure' in our mind; nevertheless, a single dividing line determines the shape and character of both units, serving a double function.

Figure 5. The classic figure/ground dichotomy investigated by psychologist Edgar Rubin in 1915.

Tiling: covering of the plane by tiles, without gaps, and no overlaps of their interiors.

Vertes: a point at which at least three tiles meet

Edge: a boundary curve of a tile that connects two consecutive vertices.

Isohedral tiling: For any chosen pair of tiles in the tiling, there is an isometry (a translation, rotation, reflection, or glide-reflection) that transforms the first tile in the pair into the second tile, and at the same time, superimposes every tile in the tiling onto another tile.

More than two-thirds of Escher's tilings use two colors: each tile is colored in one of the two colors and tiles that share an edge have contrasting colors.
Escher produced three distinct types of 2-color tilings:
(1) Two colors, one tile
In these tilings, the pattern formed by all tiles of one color (say, white) was exactly the same as that formed by all tiles of the other color (say, black). Tiles are said to have counterchange symmetry, i.e there is at least one symmetry that interchanges colors and each symmetry of the tiling either preserves all colors or interchanges all colors.

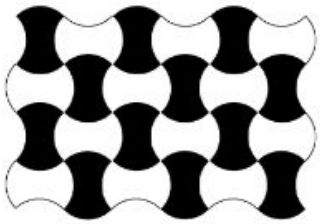


Figure 6. A sample two colour one tile tiling pattern

(2) Two colors, two different tiles, each occurring in a single color
In this tiling pattern, two distinct tiles with contrasting colors interlock, with all repetitions of a single tile the same color. These tilings are 2-isohedral, that is, given any pair of congruent tiles, there is a symmetry of the whole tiling that transforms the first tile in the pair to the second.Since each shape occurs in only one color, all symmetries preserve colors.
This pattern was achieved by Escher by beginning with an isohedral tiling, splitting one tile to produce two separate shapes, then repeating that same split on every tile in the tiling. In performing the splitting, he had to ensure that the resulting tiling could be map-colored with two colors in such a way that all tiles of a single shape were the same color. This process is called transition.

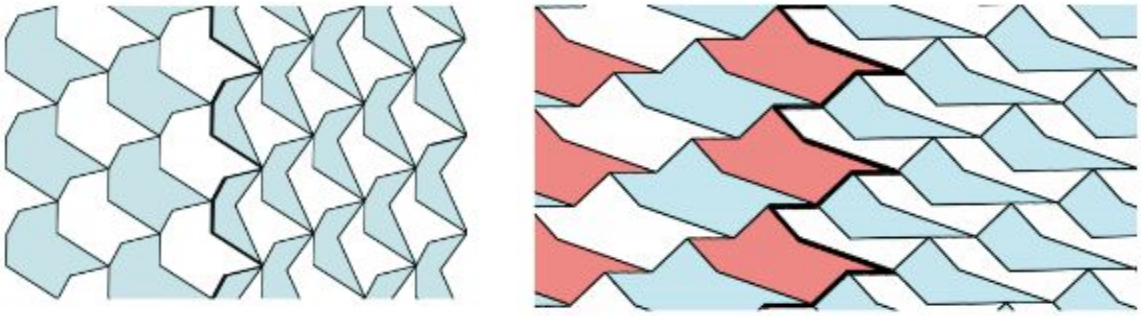


Figure 7. Splitting each tile in an isohedral tiling can produce the above category of tiling pattern.

(3) Two colors, two different tiles, each occurring in both colors

This tiling pattern had two distinct shapes of tiles, but each shape repeated in both the dark and the light color. These tilings, like those discussed in (2), were 2-isohedral, but also had the property of counterchange symmetry discussed in (1), and were perfectly colored.

Figure 8. Escher's drawing no. 34B. 1941.

An Algorithm to Generate Repeating Hyperbolic Patterns

The purpose is to describe an algorithm that can draw patterns based on tilings by a polygon that is not necessarily regular. We can assume that such a polygon is convex, and further attention is restricted to finite polygons (without points at infinity).This paper review hyperbolic geometry and regular tessellations. Then it describe the new algorithm and show some sample patterns. Finally, it conclude, indicating directions of future research.

Poincare´ Circle Model of Hyperbolic Geometry

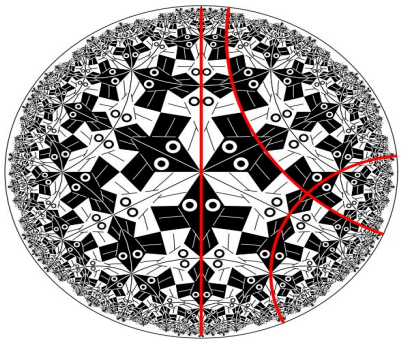


Figure 9. Poincare´ Circle Model of Hyperbolic Geometry

In this figure :

• Points: points within the bounding circle

• Lines: circular arcs perpendicular to the bounding circle (including diameters as a special case)

The Regular Tessellations {p,q}

There is a regular tessellation, {p,q} of the hyperbolic plane by regular p-sided polygons meeting q at a vertex provided $(p-2)(q-2) > 4$

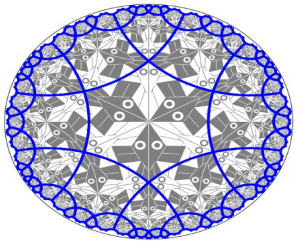


Figure 10. The tessellation {6, 4} underlying the Circle Limit I pattern and superimposed on it.

The General Replication Algorithm

- A motif is a basic sub-pattern, of which the entire repeating pattern is comprised.
- Replication is the process of transforming copies of the motif about the hyperbolic plane in order to create the whole repeating pattern.
- A fundamental region for the symmetry group of a pattern is a closed topological disk such that copies of it cover the plane without gaps or overlaps.

- In Escher patterns the motif can usually be used as a fundamental region.
- For a pattern with a finite motif, the fundamental region can be taken to be a convex polygon. This polygon will contain exactly the right pieces of the motif to reconstruct it.
- Replication using copies of such a fundamental polygon will also create the entire pattern of motifs.

A Fundamental Polygon Tessellation

A quadrilateral can used as the fundamental region for the Circle Limit III pattern, as shown below:

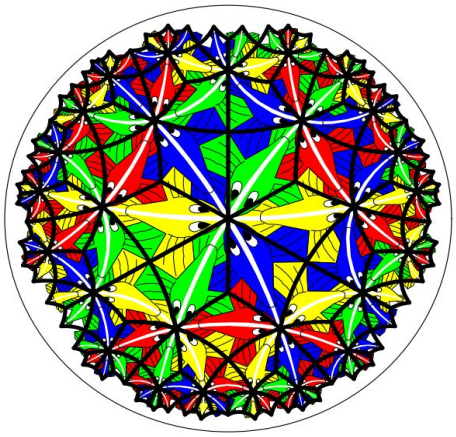


Figure 11. A polygon tessellation superimposed on the transformed Circle Limit III pattern.

Layers of Fundamental Polygons

- The fundamental polygons are arranged in layers (also called coronas in tiling literature), which are defined inductively.
- The first layer consists of all polygons with a vertex at the center of the bounding circle.
- The boldmath k+1 st layer consists of all polygons sharing an edge or vertex with the k th layer (and no previous layers).

A Polygon Tessellation Showing Layers

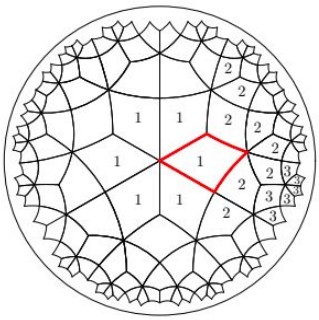


Figure 12. The polygon tessellation, with a fundamental polygon emphasized and parts of layers 1, 2, and 3 labeled.

Specification of the Fundamental Polygon

We use {p; q1, q2, . . . , qp} to denote the fundamental polygon with p sides and qi polygons meeting at vertex i (so the interior angle at the i th vertex is 2π/qi).

The condition that a polygon is a fundamental polygon for a hyperbolic tessellation is that:

$$\sum_{i=1}^{p} 1/q(i) \; < \; p/2 \; - 1$$

(which generalizes the condition (p − 2)(q − 2) > 4 for regular tessellations). If the "<" is replaced with "=" or ">", one obtains a Euclidean or spherical tessellation respectively.

We say a polygon of a tessellation has minimal exposure if it shares an edge with a previous layer; we say it has minimal exposure if it shares a vertex with a previous layer

Minimal and Maximal Exposure

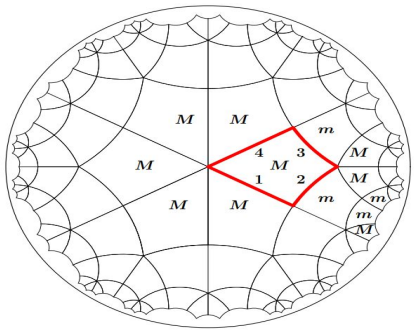


Figure 13. The (6, 4, 5, 4) tessellation showing a fundamental polygon, and with some minimally exposed and maximally exposed polygons marked with m and M respectively.

Some Polygons and Replication

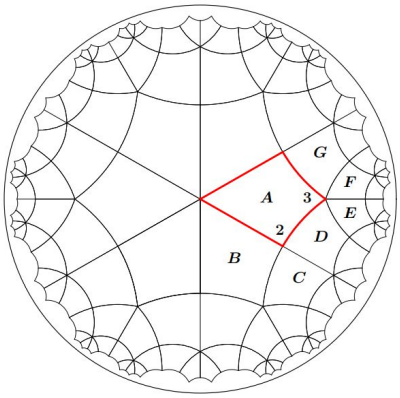


Figure 14. A first-layer polygon A, two of its vertices labeled 2 and 3, and some polygons adjacent to A.

This figure shows how recursive calls in the replication work starting at polygon A. Polygon vertices are numbered in counter-clockwise order with vertex i at the right end of edge i looking outward.

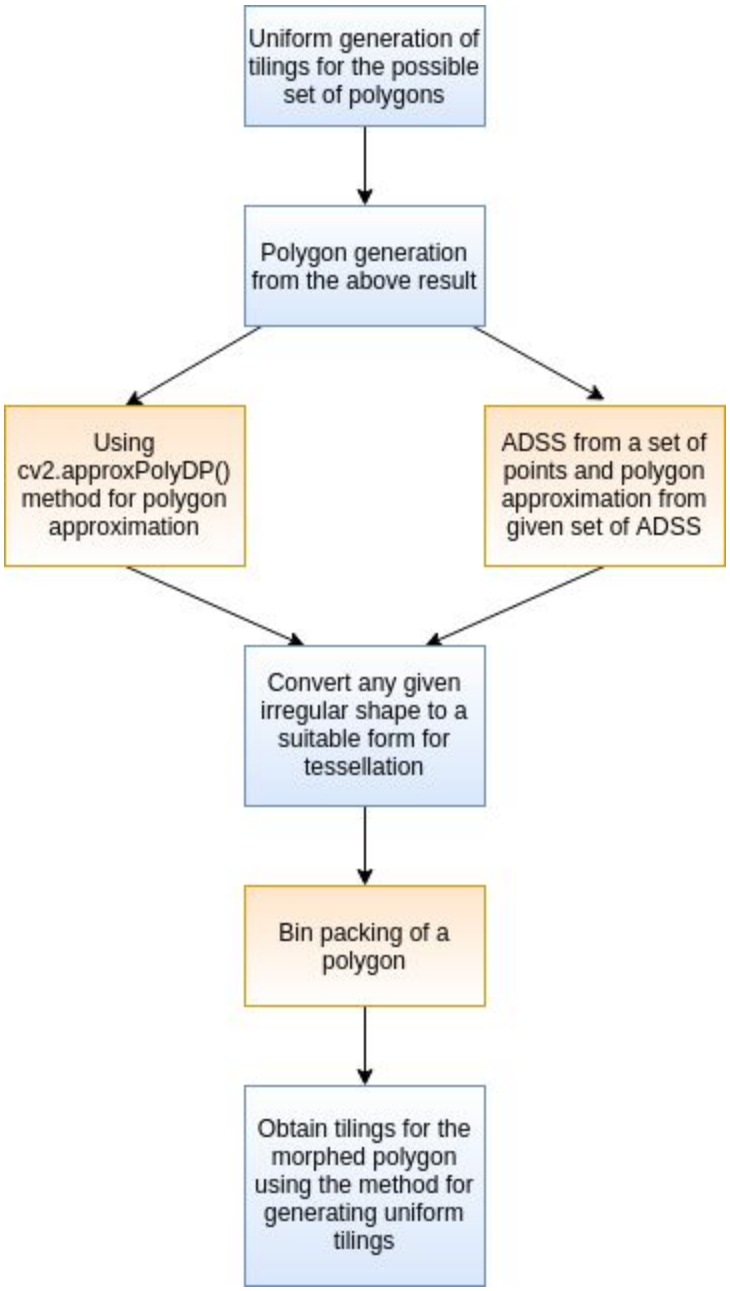
Future Work

- Allow vertices at infinity.
- Create a program to transform between different fundamental polygons.
- Automatically generate patterns with color symmetry.

Objectives of the work done

- Generate uniform tilings for the possible set of polygons
- Generate polygon from a given figure
    - Using cv2.approxPolyDP() method for polygon approximation
    - ADSS from a set of points and polygon approximation from given set of ADSS
- Convert any given irregular shape to a form suitable for tessellation
    - Bin packing of a polygon
- Obtain tilings for the morphed polygon using the method for generating uniform tilings

Work plan to meet the objectives



Implementation details

Generating uniform tilings

Tools used

- ***Pycairo***

  pycairo is used for rendering the tiling pattern.

  Cairo is a library for drawing vector graphics. Vector graphics are interesting because they don't lose clarity when resized or transformed. Pycairo is a set of bindings for cairo. It provides the cairo module which can be used to call cairo commands from Python.

  Api reference:

  - ➔ *ImageSurface*

    Image surfaces provide the ability to render to memory buffers either allocated by cairo or by the calling code.

    class cairo.ImageSurface(format, width, height)

    - ◆ Parameters:

      format – FORMAT of pixels in the surface to create

      {*in the above implementation, FORMAT_RGB24 (each pixel is a 32-bit quantity, with the upper 8 bits unused. Red, Green, and Blue are stored in the remaining 24 bits in that order) is used*}

      width – width of the surface, in pixels

      height – height of the surface, in pixel

    - ◆ Returns: a new ImageSurface

  - ➔ Context

    Context is the main object used when drawing with cairo. To draw with cairo, a *Context* object is created, the target surface is set along with the drawing options for the *Context.*

    class cairo.Context(target)

- Parameters: target – target Surface for the context
  {*In the implementation above ImageSurface class object is used to initialise the context}*
- Returns: a newly allocated Context

Models

- Shape: Generates the shape to be rendered in the tiling function.
  - Data members:
    - `sides`: no of sides of the shape
    - `x, y`: determines the center of the polygon
    - `rotation`: angle to rotate the figure by
    - `fill`: color to fill the shape
    - `stroke`: boundary for shape
  - Class methods:
    - `copy(x, y)`: returns copy of shape with center at (x, y)
    - `points(margin)`: returns vertices of the shape
    - `adjacent(sides, edge)`: returns a shape adjacent to the given shape along the edge passed in as parameter along with the number of sides.
    - `render()`: renders the shape

- DualShape: Generates dual of a graph. It inherits model Shape.
  The dual graph of a plane graph G is a graph that has a vertex for each face of G. The dual graph has an edge whenever two faces of G are separated from each other by an edge, and a self-loop when the same face appears on both sides of an edge. Thus, each edge e of G has a corresponding dual edge, whose endpoints are the dual vertices corresponding to the faces on either side of e.
  - Class methods:
    - `Point`: returns vertices for the dual shape

- Model: Generates the required tiling pattern
  - Data members:
    - `height, width`: To set dimensions for the ImageSurface

        - `scale`: scale for cario.Context used for rendering
        - `shapes`: List of base shape objects to be rendered the the model. Repetition of this set of shapes results in the tiling pattern
        - `lookup`: dictionary with maps center of polygon to the shape object denoting the polygon. Contains record of every shape rendered on the plane.
    - Methods:
        - `append(shape)`: appends shape object to the existing model
        - `_add(index, edge, sides)`: appends a shape with number of sides as specified in the third input parameter. The shape is added adjacent to the shape self.shapes[index] of the existing model along the edge passed as parameter
        - `add(indexes, edges, sides)`: for each shape at index in the *indexes* list say s, add shape with sides *sides* for each edge of the shape s in the *edges* list
        - `add_repeats(x, y)`: This method provides translation. For each shape in the existing model translate the shape by (x, y) dimension and add it to the lookup dictionary if it does not exist already.
        - `_repeat(indexes, x, y, depth, memo)`: This method recursively calls itself for each shape s in the *indexes* list depth number of times to translate the shapes by (x + s.x, y + s.y) for generating the repeating pattern of tessellation
        - `repeat(indexes)`: Call _repeat iteratively increasing the depth parameter until the rendered shapes move out of the ImageSurface
        - `dual()`: generate dual for the entire tiling pattern and return a list of DualShape objects
        - `render(dual, background_color, margin, show_labels, line_width)`: set the properties for the cairo.Context and render the model. If dual is true, the dual of the model is rendered. Returns ImageSurface object.

Workflow

The workflow describes creation of one of the 11 patterns of the uniform tiling. A similar process is used for generation of all the patterns.

Step 1: Create a Model that will hold the polygons to be rendered.

```
model = Model()
```

Step 2: Place the first polygon at the origin. We need only specify its number of sides. Let's add a hexagon.

```
model.append(Shape(6))
```

Step 3: Render the model as follows:

```
surface = model.render()
surface.write_to_png('output.png')
```

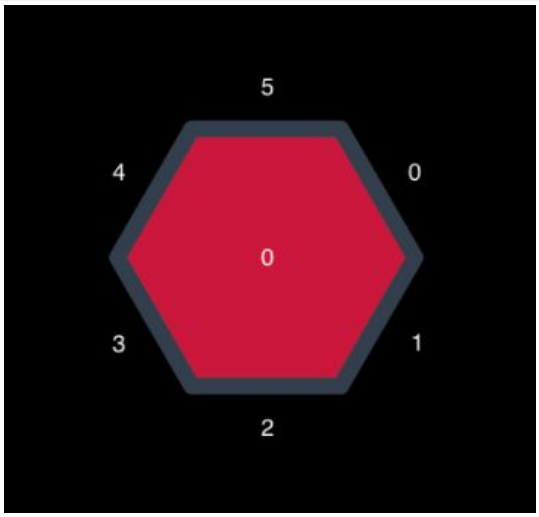


Figure 15. Model rendered with a hexagon

Step 4: Add squares to all sides of the hexagon. This is done as follows:

```
a = model.add(0, range(6), 4)
```

The first parameter, `0`, specifies which shape(s) we're attaching to. Here, we're only attaching to one shape (the hexagon) and it was the first one created, so it's referred to by zero.

The second parameter, `range(6)`, specifies the edges we're attaching to. In this case we want to attach to all six sides of the hexagon. You can see the edges labeled in the output image.

The third parameter, `4`, specifies the number of sides for the new shapes. In this case, squares.

The return value of add tracks the indexes of the newly created squares so we can refer to them later.

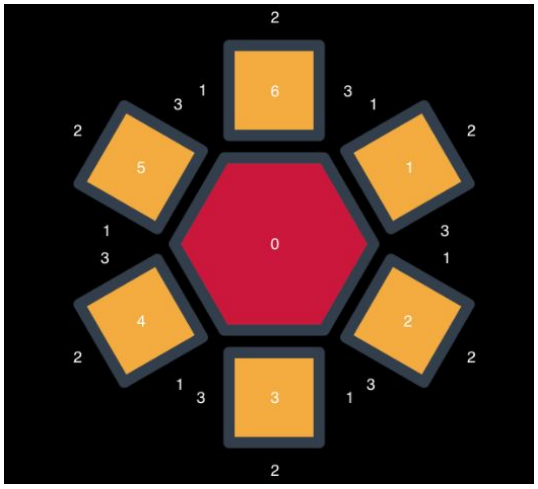


Figure 16. Squares attached to all sides of hexagon

Step 5: Attach triangles to all of the squares, just created in one fell swoop by using the previous return value. Here, we are adding triangles to edge number 1 of each of those squares.

```
b = model.add(a, 1, 3)
```

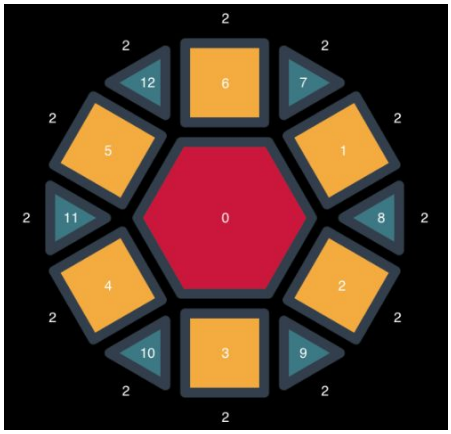


Figure 17. Triangles attached to edge of squares

Step 6: Add more hexagons which will represent the repeating positions of our template.

```
c = model.add(a, 2, 6)
```

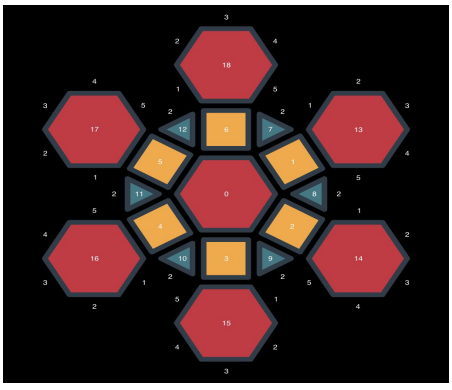


Figure 18. Repeating base produced for tiling

Step 7: Repeat the pattern

Now that we have positions for repeating the pattern, we can use a repeat function to automatically fill in the rest of the surface with our pattern.

```
model.repeat(c)
```

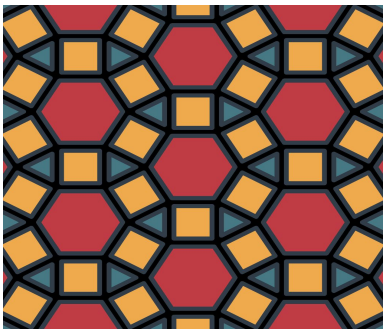
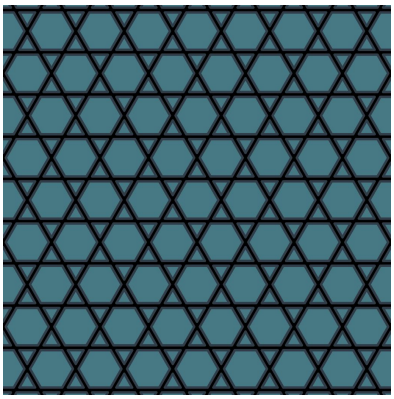
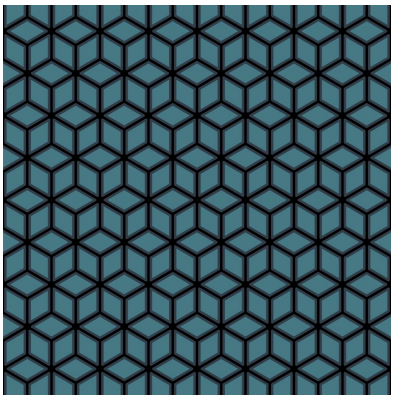
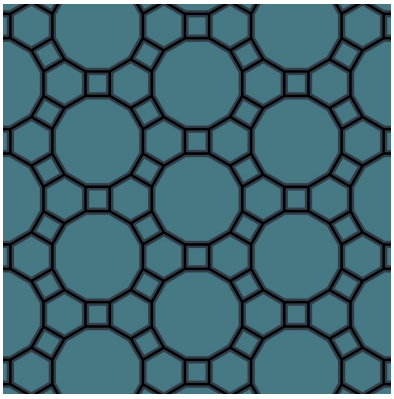
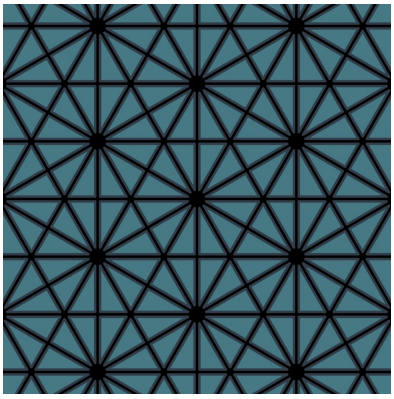
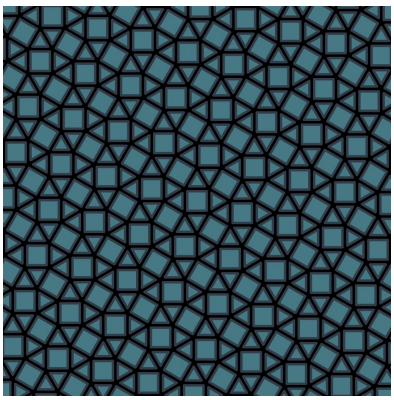
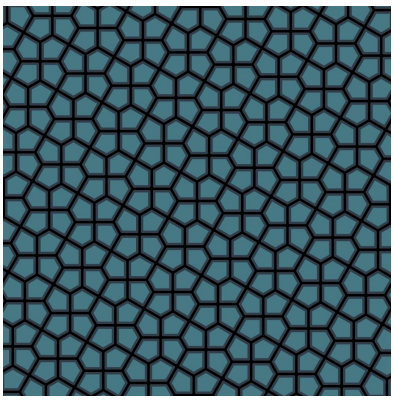


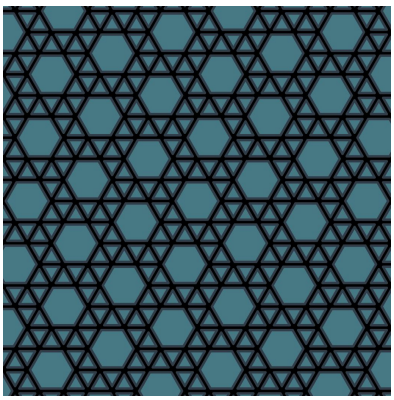
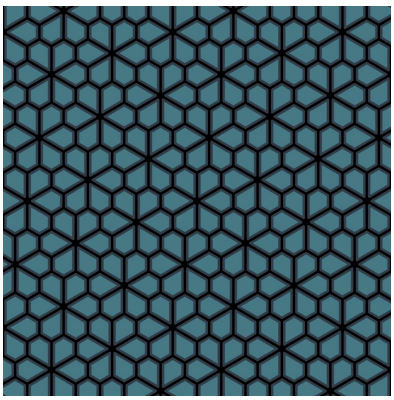
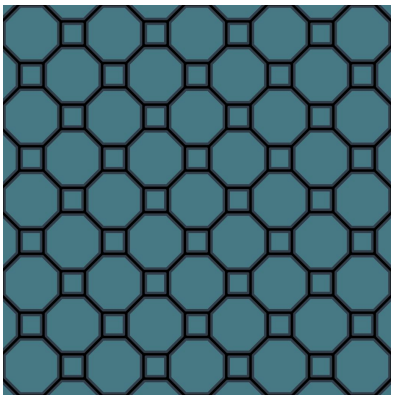
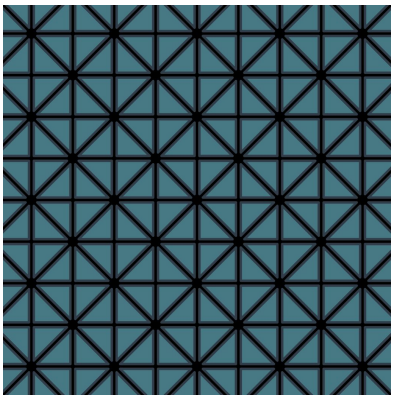
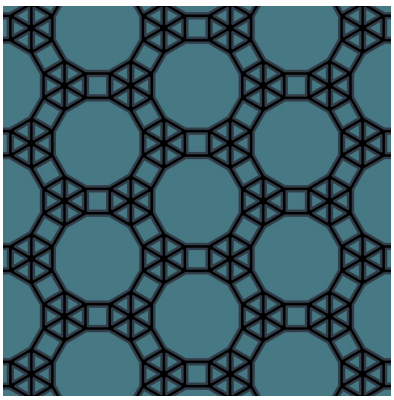
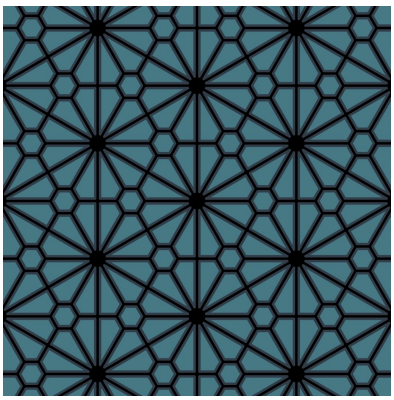
Figure 19. Final tiling of format 3.4.6.4

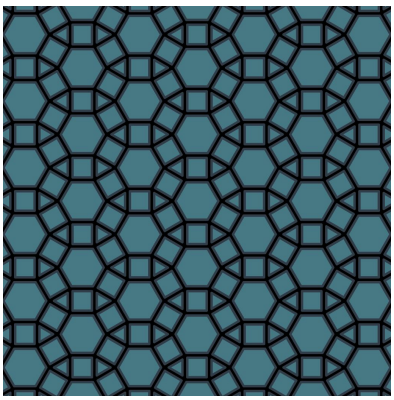
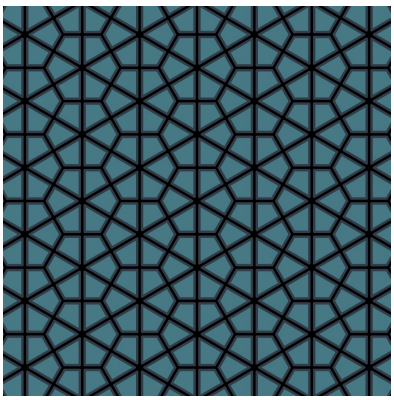
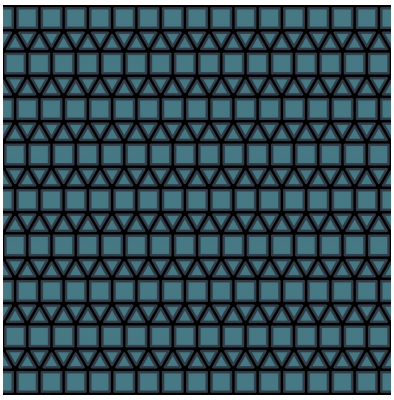
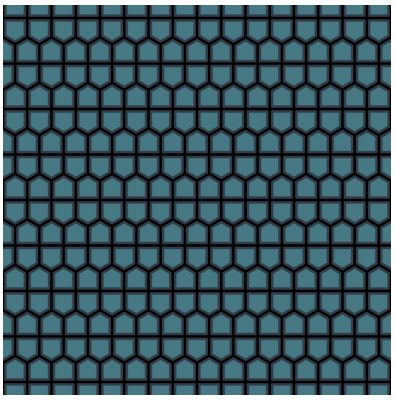
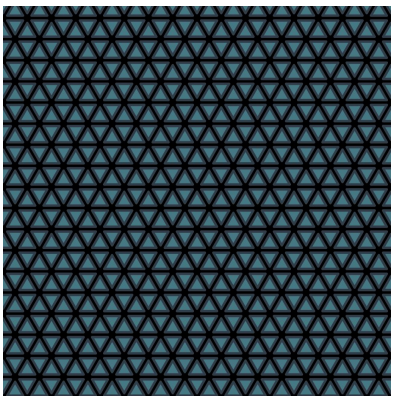
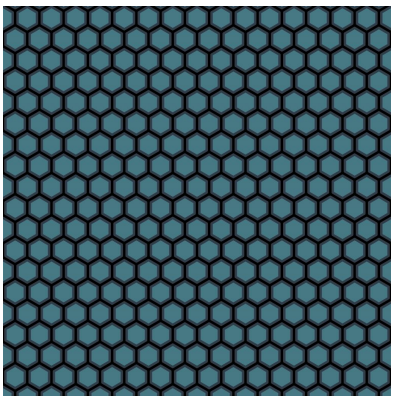
Here's all the code needed for this pattern:

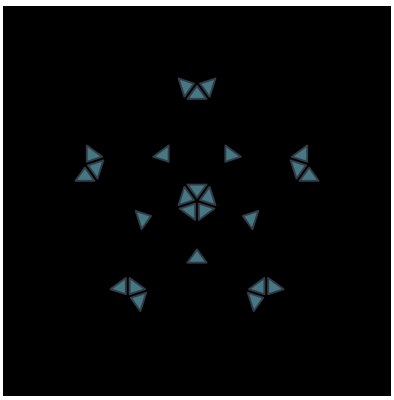
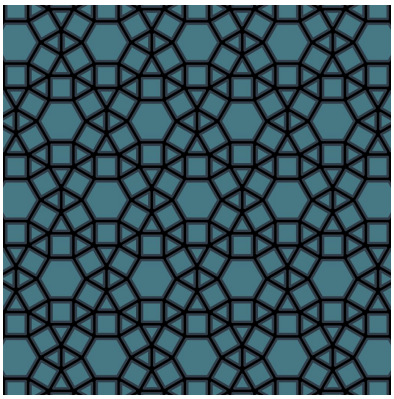
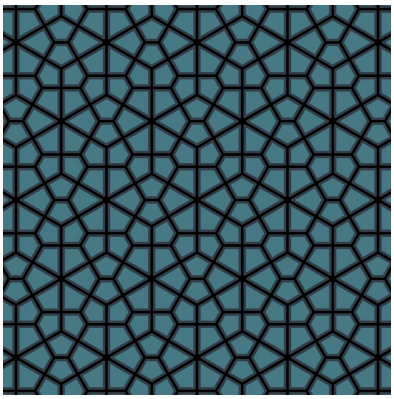
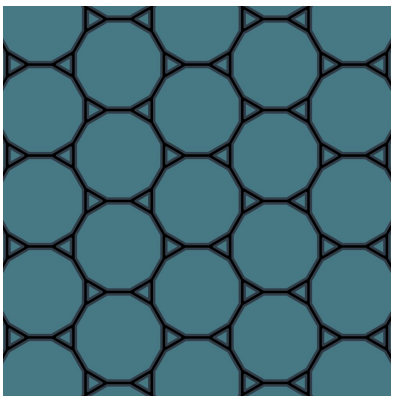
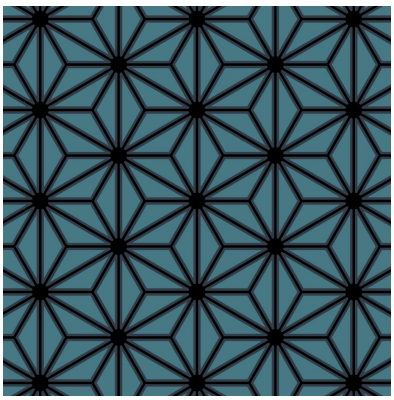
```
from tile import Model, Shape

BLUE = 0x477984
ORANGE = 0xEEAA4D
RED = 0xC03C44

model = Model()
model.append(Shape(6, fill=RED))
a = model.add(0, range(6), 4, fill=ORANGE)
b = model.add(a, 1, 3, fill=BLUE)
c = model.add(a, 2, 6, fill=RED)
model.repeat(c)
surface = model.render()
surface.write_to_png('output.png')
```

The final results obtained for the uniform tilings are listed below:

Vertex configuration	Tiling	Dual of tiling
3.6.3.6		
4.6.12		
3.3.4.3.4		

3.3.3.3.6		
4.8.8		
3.3.4.12 / 3.3.3.3.3.3		

3.4.6.4		
3.3.4.4		
3.3.3.3.3.3		

-		
3.3.4.3.4/3.4.6.4		
3.12.12		

Obtaining a polygon from a given figure

Method 1: Using approxPolyDP()

Tools used

- **NumPy** is the fundamental package for scientific computing with Python. It contains among other things:
  - a powerful N-dimensional array object
  - sophisticated (broadcasting) functions
  - tools for integrating C/C++ and Fortran code
  - useful linear algebra, Fourier transform, and random number capabilities
- **OpenCV-Python** is the Python API of OpenCV. It combines the best qualities of OpenCV C++ API and Python language.

Workflow

Step 1: Load and shrink a large image to limit the number of points obtained in the final polygon.

```
img = cv2.imread('image.png')
img = cv2.resize(img, None,fx=0.25, fy=0.25, interpolation =
cv2.INTER_CUBIC)
```

Step 2: Get a blank canvas for drawing contour on and convert img to grayscale.

```
canvas = np.zeros(img.shape, np.uint8)
img2gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
```

Step 3: Threshold the image and extract contours.

```
ret,thresh = cv2.threshold(img2gray,250,255,cv2.THRESH_BINARY_INV)
im2,contours,hierarchy = cv2.findContours(thresh, cv2.RETR_TREE,
cv2.CHAIN_APPROX_NONE)
```

Step 4: Find the biggest area

```
cnt = contours[0]
max_area = cv2.contourArea(cnt)

for cont in contours:
    if cv2.contourArea(cont) > max_area:
        cnt = cont
        max_area = cv2.contourArea(cont)
```

Step 5: Define main contour approx. and hull

```
perimeter = cv2.arcLength(cnt,True)
epsilon = 0.003*cv2.arcLength(cnt,True)
approx = cv2.approxPolyDP(cnt,epsilon,True)

hull = cv2.convexHull(cnt)
```

Step 6: Draw the contours

```
cv2.drawContours(canvas, cnt, -1, (255, 0, 0), 1)
cv2.drawContours(canvas, approx, -1, (0, 0, 255), 3)

cv2.imshow("Contour", canvas)
```

Step 7: Join the points obtained to get the approximate polygon

```
l = len(approx)

for i in range(l):
      pa = (approx[i].item(0), approx[i].item(1))
      pb = (approx[(i+1)%l].item(0), approx[(i+1)%l].item(1))
      cv2.line(canvas, pa, pb, (255, 0, 0), 2)
```

The result obtained is as shown in the figure below:

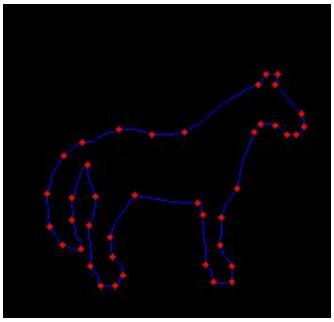
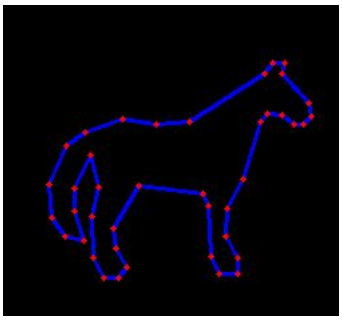
Input Image	Polygon vertices	Generated approx polygon
		

Figure 20. Approx polygon of horse obtained from grayscale image of horse

Method 2: Using ADSS for polygon approximation

ADSS of any digital curve is the approximate line segment which can be obtained from it. Extracting all ADSS from a figure and then merging them provides an approximate polygon of the figure. Since the set of ADSS provides an elegant and compact representation of DC, it is very effective in producing approximate polygons (or polychains) using a single parameter. The whole process consists of two stages—first, extraction of ADSS and, second, polygonal approximation.

Step 1: Find parameters of ADSS n,s and l

The parameters of ADSS are defined as follows:

- Orientations parameters given by n (nonsingular element), s (singular element), l (length of the leftmost run of n), andr (length of the rightmost run of n). For example, the andchain code $0^4 10^5 10^5 10^4 10^4 10^5$ has n=0, s=1, l=4 and r=5.
- Run length interval parameters given by p and q, where [p, q] is the range of possible lengths (excepting l and r) of n in C that determines the level of approximation of the ADSS, subject to the following conditions:

```
                    q - p <= d = floor( (p + 1) / 2)
                (l - p), (r - p) <= e = floor( (p + 1) / 2)
```

```
def find_param(C, u):
    """Find parameters for ADSS: n, s and l."""
    global n,s,l
    i = u
    # Cases where only one element is left in chain code
    if i+1 > len(C):
        s = -1
        n = C[i]
        l = 1
        return

    # Cases of type (00..00)1..
    if C[i] == C[i+1]:
        n = C[i]
        l = i
        i += 1
        while i < len(C) and C[i] == n:
            i += 1
        l = i - l
        if i == len(C):
            s = -1
            return
        s = C[i]
        return

    # Cases of form 01... or 10...
    n = C[i]
    s = C[i+1]
    l = 1
    i += 1
    while i+1 < len(C) and C[i+1] in (n,s):
        if C[i] == C[i+1]:
            if C[i] == s:
                temp = n
                n = s
                s = temp
                l = 0
            return
```

```
        else:
            i += 1
```

Step 2: Find run length of parameters

Run length is the length of consecutive occurrence of the parameter n in the chain code. The function below finds the run length from a specified starting point.

```
def find_run(C, st):
    """Find next run."""
    global n, s, l
    cnt = 0
    i = st
    if i+1 > len(C):
        return -1
    else:
        i += 1
        while i<len(C) and C[i] == n:
            cnt += 1
            i += 1
        return cnt
```

Step 3: Extract ADSS using parameters and run length

For any line segment to be an ADSS of a DC, the following properties must be satisfied:

- The runs have at most two directions, differing by 45 degrees, and, for one of these directions, the run length must be 1.
- The run lengths of n can vary by more than unity, depending on the minimum run length of n as follows:

```
                q - p <= d = floor( (p + 1) / 2)
(l - p), (r - p) <= e = floor( (p + 1) / 2)
```

```
def extract_adss(C):
    """Extract ADSS from chain code of DC."""
    global n, s, l
    A = [0]
    u = 0
    length = len(C)
    while u < length:
        find_param(C, u)
        c = l
        if (s - n) % 8 == 1:
            p = q = find_run(C, u + c)
            if p != -1:
                d = e = (p + 1) / 2
                c = c + 1 + p
                if l - p <= e:
                    while q - p <= d:
                        k = find_run(C, u + c)
                        if k != -1:
                            if l - k > e:
                                c = c + 1 + k
                                break
                            if k - p <= e:
                                c = c + 1 + k
                            else:
                                c = c + 1 + p + e
                            if k < p:
                               p = k
                               d = e = (p + 1) / 2
                            if k > q:
                                q = k
        u = u + c
        A.append(u)
        u = u + 1
```

Step 4: Approximate polygon by merging the obtained sets of ADSS

Extraction of the ADSS for each curve C_k in the given set of DC generates an ordered set of ADSS, namely, $A_k = \{L^{\{k\}}_i\}$ where i = [1, n_k], corresponding to C_k. In each such set A_k, several

consecutive ADSS may occur, which are approximately collinear and, therefore, may be combined together to form a single segment.

Let $\{L^{\{k\}}\}_{j1}^{j2}$ be the maximal (ordered) subset of the ADSS starting from $L_{j1}^{\{k\}}$ that conforms to some approximation criterion. Then, these `j2-j1+1` segments in A_k are combined together to form a single straight line segment starting from the start point of $L_{j1}^{\{k\}}$ and ending at the endpoint of $L_{j2}^{\{k\}}$. This procedure is repeated for all such maximal subsets of A_k in succession to obtain the polygonal approximation.

The criterion used for approximation are as follows:

Cumulative error criterion	$\sum_{j=j_1}^{j_2-1}\left\|\triangle\left(s\left(\mathbf{L}_{j_1}^{(k)}\right),e\left(\mathbf{L}_{j}^{(k)}\right),e\left(\mathbf{L}_{j_2}^{(k)}\right)\right)\right\| \leq \tau d_{\mathrm{T}}\left(s\left(\mathbf{L}_{j_1}^{(k)}\right),e\left(\mathbf{L}_{j_2}^{(k)}\right)\right),$
Maximum error criterion	$\max_{j_1\leq j\leq j_2-1}\left\|\triangle\left(s\left(\mathbf{L}_{j_1}^{(k)}\right),e\left(\mathbf{L}_{j}^{(k)}\right),e\left(\mathbf{L}_{j_2}^{(k)}\right)\right)\right\| \leq \tau d_{\mathrm{T}}\left(s\left(\mathbf{L}_{j_1}^{(k)}\right),e\left(\mathbf{L}_{j_2}^{(k)}\right)\right).$

```
def merge_adss(A, n, alpha):
    """Merge ADSS to obtain an approximate polygon."""
    for i in range(n):
        S = 0
        for j in range (n-m-1):
            S = S + find_triangle_area(A[m], A[m + i], A[m + i + 1])
            dx = abs(A[m].x - A[m + i + 1].x)
            dy = abs(A[m].y - A[m + i + 1].y)
            d = max(dx, dy)
            if S <= d * alpha:
                delete(A, A[m + i])
            else:
                break
        m = m + i + 1
```

Convert any figure to form suitable for tessellation: Bin packing

Packing problems arise in a wide variety of application areas. The basic problem is that of determining an efficient arrangement of different objects in a region without any overlap. Many research works have been done on two and three dimensional rectangular packing. However there are many situations when either objects or the containing region is irregular in shape. In this report, we concentrate on two-dimensional packing problems involving irregular shaped objects.

Rectilinear Representation and Heuristics

We first adopted a rectilinear representation for objects to be packed. The major reason is due to its simplicity - like integer coordinates and easy checking for overlapping, etc. After the rectilinearization, objects become more regular (though we will lose some information). Hence it becomes easy to perform advanced heuristic methods.

Figure 21. Rectilinear representation and edge counting

There are three advantages for us to use this representation.

- Any arbitrarily shaped object, including convex and concave, etc, can be represented easily.
- Rotation and checks for overlapping are easy.
- Advanced heuristics (GA & TS) can be applied.

On the other hand, there are also two major disadvantages.

- Complexity of representation and computation grows quadratically with the size of each object.
- Only four orientations are possible.

We use the goodness factor for an object (or a set of combined objects) as a measurement for the quality of packing. The goodness factor is dependent on two values.

1. The total number of edges, which is defined to be the total number of turns when going along the contour of an object.The rationale is that the lesser the edge number is, the better the packing is.
2. The measure of wasted area, which can be obtained by counting the empty cells of the minimum bound rectangle of such configuration. Since our objective is to improve the utilization of the container, higher weightage may be set for the number of wasted area.

The combined function will return the best combination for two objects.

Compaction

The packing configuration generated by combining can be further improved via compaction. Our compaction routine tries to move every object left-most and down-most, starting with the right-upper-most object.

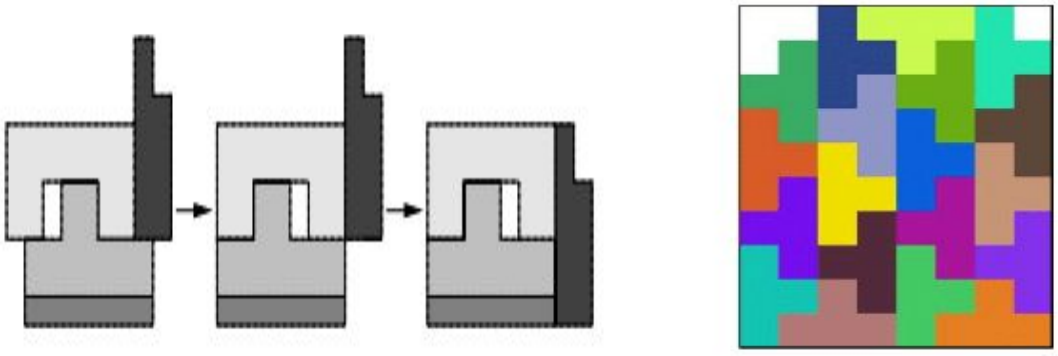


Figure 22. Compacting of objects and packing of T shapes

Boundary Representation and Heuristics

The rectilinear representation introduced in the previously is based on the discretization of irregular objects. Its disadvantages were also discussed.

The use of polygon boundary representation overcomes those mentioned shortcomings. In fact, such an approach is also more accurate due to the use of real number coordinates.

Advanced heuristic methods require an abstract solution representation to perform the search technique. However, it is not easy to find such abstractions under the boundary representation. Hence we use a greedy based algorithm to solve the problem.

We have three ways to move to the local optimum, namely shifting down-most, shifting left-most and rotation. The algorithm continues pushing objects downwards and leftwards until the objects reach stable positions. The use of the rotation operator in this process is to ensure that the highest point of the object being pushed is kept as low as possible.
Since the algorithm is greedy in nature, the order of the objects being packed will essentially affect the packing result. We proposed three packing sequences and did experiments for all of them.

- Pack object with largest area first.
- Pack object with smallest area first.
- Pack objects in random order.

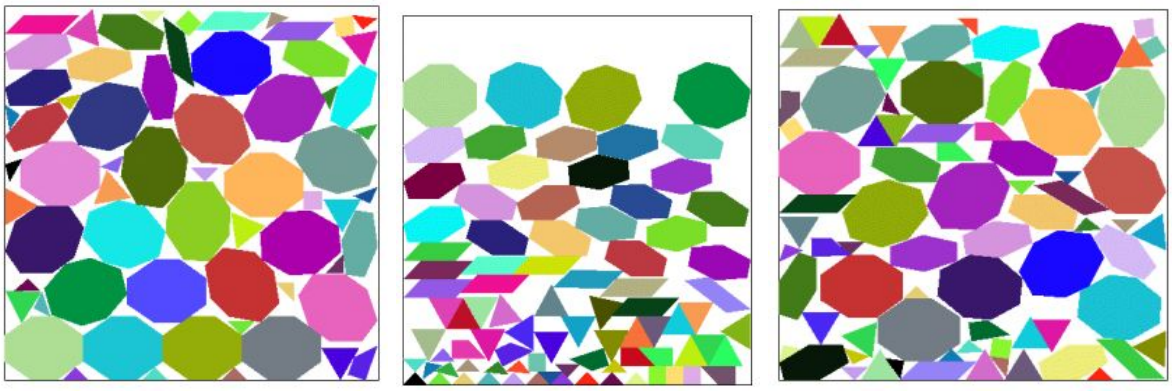


Figure 23. Largest area first. Utilization: 80%, smallest area first. Utilization: 60%, packing in random order. Utilization: 74%

Conclusion

Generating tessellations for any given figure will be a significant contribution not only to the digital world but also to the fields of Mathematics, art and architecture. Generating tessellation for regular polygons has a well defined algorithm and the same can be used for irregular shapes if we are able to obtain an efficient bin packing of the shape. Our future

goal is to implement bin packing for 2D shapes and figures and depending on the utilization efficiency, decide whether the figure can be tessellated or not. Our project aims to provide an efficient method for the tessellation of irregular shapes and objects using the simple techniques of bin packing and regular polygon tessellation.