GENERATING HYPERBOLIC PATTERNS FOR REGULAR AND NON-REGULAR P-GONS

A Thesis

Submitted to the Faculty

of

University of Minnesota, Duluth

by

Ajit V. Datar

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Computer Science

Jul 2005

UNIVERSITY OF MINNESOTA

This is to certify that I have examined this copy of master's thesis by

Ajit V. Datar

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

_____

Name of Faculty Adviser

_____

Signature of Faculty Adviser

_____

Date

GRADUATE SCHOOL

ACKNOWLEDGMENTS

I thank Dr. Douglas Dunham for his guidance throughout my thesis work. He patiently answered many of my queries which helped me understand the theory. I also appreciate the freedom he gave me to try out different ideas.

I am grateful to Dr. Rocio Alba Flores and Dr. Gary Shute for agreeing to be on the thesis committee. I would also like to thank the faculty of the Computer Science Department. Directly or indirectly I have learned a lot from them. I thank Lori Lucia and Linda Meek from the CS office for their timely help regarding the office formalities.

Finally I thank Joshua Jacobs for many discussion regarding the algorithms. I thank my friends who have helped me test the program and suggested different ideas. All my friends here in Duluth have made my stay an enjoyable experience.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

List of Algorithms

ABSTRACT

Datar, Ajit V. M.S, University of Minnesota, Duluth, Jul, 2005. Generating Hyperbolic patterns for Regular and Non-Regular p-gons. Major Professor: Dr. Douglas Dunham.

The mathematics and art of repeating tessellations is fascinating and intriguing. Repeating hyperbolic patterns is an interesting area of mathematics, dealing with hyperbolic geometry. M. C. Escher, the Dutch artist, created four hyperbolic "Circle Limit" patterns, which are some of his most beautiful patterns.

We generalize and extend the idea behind Escher's "Circle Limit" patterns to regular and non-regular tessellations in the hyperbolic plane. Such tessellations when projected onto the Poincaré disk give us patterns similar to those created by Escher.

The purpose of this work is to provide algorithms to generate such hyperbolic designs. We have written a program Hyper*Art* in C++ to implement such algorithms. By default, Hyper*Art* provides a Poincaré view of the designs, but it also provides a framework to implement additional types of views. Through Hyper*Art* we hope to provide a cross-platform framework for designing, viewing and experimenting with various types of hyperbolic tessellation algorithms.

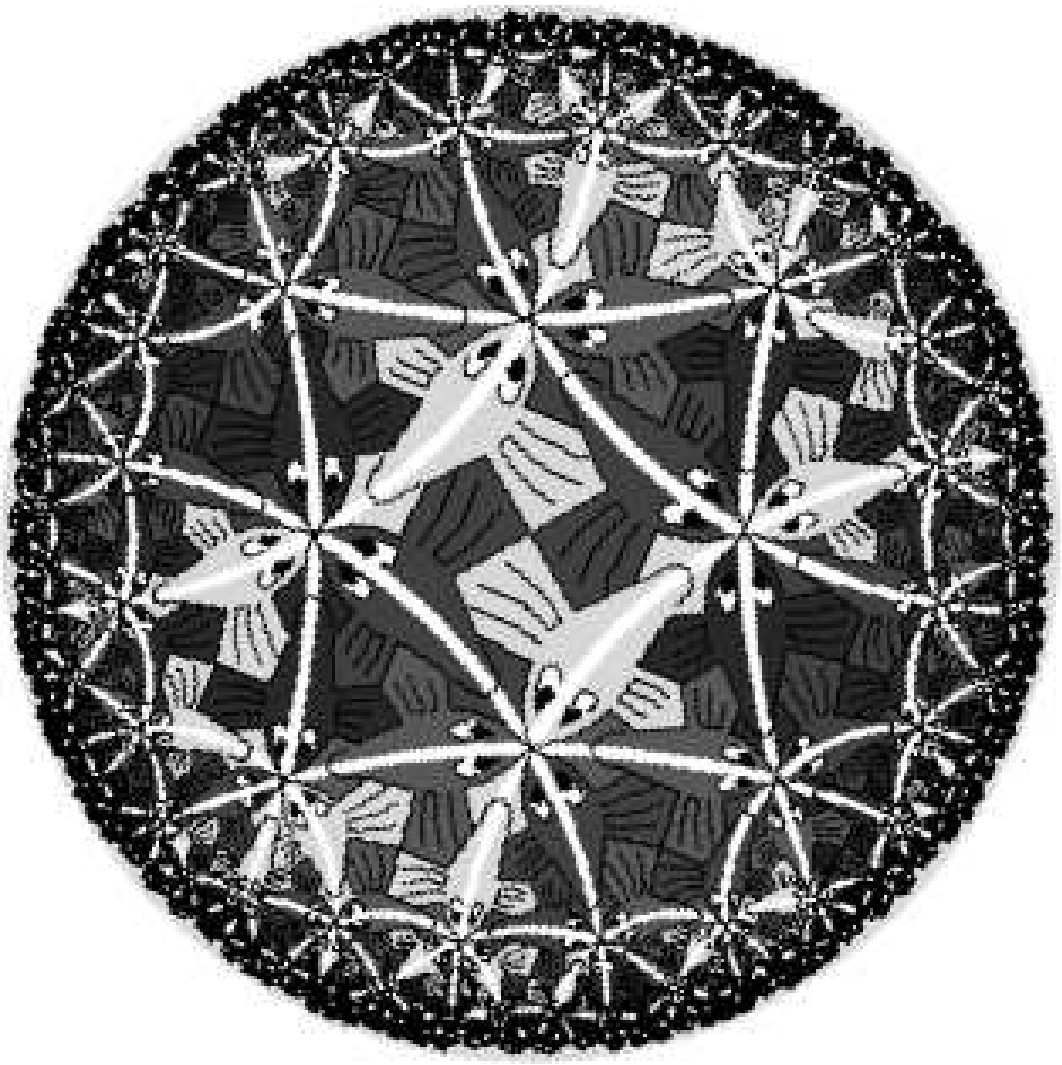


Fig. 1. Circle Limit III

1. INTRODUCTION

A *Tessellation* is a tiling by polygons which may or may not be regular [1]. The topic of repeating tessellations is a fascinating area of mathematics. Depending on the plane which the tilings cover, a tessellation can be classified as Euclidean, hyperbolic or spherical. This thesis describes algorithms for generating repeating hyperbolic tessellations. The software Hyper*Art* , provided with this thesis, helps generate patterns similar to those created by Escher.

M. C. Escher, the Dutch artist, created many such tessellations. He created four patterns called "Circle Limit" patterns, which filled the Poincaré circle with transformed copies of a basic pattern or motif. One such pattern, a rendition of the Circle Limit III, is shown in Figure 1.

Dr. Dougas Dunham, inspired by Escher's work, developed computer algorithms to generate hyperbolic designs. Through all these years he has developed algorithms and programs to generate and visualize regular and non-regular tilings. He has also developed various designs in the family of Circle Limit III or the fish pattern.

In the following chapters, we will describe the algorithms by Dr. Dunham and their implementation in Hyper*Art* . We will first start by describing the mathematical background related to hyperbolic tessellations. Then we will establish the terminology used throughout the thesis. Next, we will delve into different ways in which the algorithms can be classified, discussing specifically the recursive algorithms to generate regular and non-regular hyperbolic tessellations. Finally we will describe the structure of Hyper*Art* , a software implementation of these algorithms. We will then present some designs generated by using these algorithms. We will conclude by listing possible features and add-ons that could be implemented in Hyper*Art* .

2. BASIC THEORY

The history of geometries starts with Euclidean geometry which is still of course an important and fundamental field of mathematics. There have been many great mathematicians in different ancient civilizations who have invented many principles and theorems of geometry (the term Euclidean geometry being redundant because it was the only type of geometry known). The contribution of Euclid (circa 300 B.C.) through his influential textbook *Elements* is seminal of course. *Elements*, divided into thirteen volumes, is a chain of propositions covering most of the geometry known at that time. We will not go into much details on the history of geometry, and interested readers are encouraged to look at [2].

What Euclid and his work *Elements* gave the world was a set of 5 axioms or postulates now known as *Euclid's Postulates*. These five postulates form a foundation for different types of geometries. Euclid's postulates are as follows [3]:

1. A straight line segment can be drawn joining any two points.
2. Any straight line segment can be extended indefinitely in a straight line.
3. Given any straight line segment, a circle can be drawn having the segment as radius and one endpoint as center.
4. All right angle are congruent.
5. If two lines are drawn which intersect a third in such a way that the sum of the inner angles on one side is less than two right angles, then the two lines inevitably must intersect each other on that side if extended far enough.

The first four postulates make up what is called *absolute geometry*, but the fifth postulate is especially interesting and is equivalent to the *Parallel Postulate* [4]:

> Given any straight line and a point not on it, there *exists one and only one straight line which passes* through that point and never intersects the first line, no matter how far they are extended.

The parallel postulate has an interesting history of "non-proofs" from Euclid's other four postulates, which is described nicely in [2]. Many mathematicians such as Janos Bolyai and Nicolai Lobachevsky proposed alternate but self–consistent non–Euclidean geometries which reinterpret the parallel postulate.

2.1 Types of Geometries

Depending on how the parallel postulate is defined, there are three classes of constant curvature geometries. All of these use the first four of the Euclid's postulate as it. What we learn normally is *Euclidean or Parabolic Geometry*. The other two non–Euclidean geometries are hyperbolic geometry and elliptical geometry. We will describe them briefly in the next sections. For a better understanding please see [2].

2.1.1 Euclidean Geometry

Euclidean Geometry is defined by the four Euclidean postulates and the original parallel postulate. This geometry is the most basic one and hence acts as reference for comparing properties of other geometries. As we know, the sum of the angles of a triangle is always 180° in this geometry.

2.1.2 Elliptical Geometry

Elliptical geometry redefines the parallel postulate as "through any point in the plane, there exists no lines parallel to the given line." In this non Euclidean geometry the sum of the angles of a triangle is greater than 180°. Also called Riemannian geometry, it can be seen as the surface of a sphere on which great circles represent

the lines. Spherical geometry, a two-dimensional non–Euclidean geometry, is the simplest model of elliptical geometry.

2.1.3 Hyperbolic Geometry

Hyperbolic geometry, the geometry which is used in this thesis, redfines the parallel postulate as "for any infinite straight line L and a point P not on it, there are many infinitely extending straight lines that pass through P and do not intersect L."
In this non–Euclidean geometry the sum of the angles of a triangle is less than 180° It is also called as Lobachevskian, Bolyai or Gauss geometry. Please refer to [5] for a short description and [2] for an in-depth coverage.

Hyperbolic geometry can be represented by many different models but all of them have 2 types of parallel lines,

1. Asymptotically parallel lines are those lines in the boundary–less models, that have no common point and intersect on the "boundary"[1]

2. Divergently/ultra parallel lines are those lines that share no point within or on the boundary of the model.

Models of Hyperbolic Geometry

There are many different models that are used to represent hyperbolic geometry — *Beltrami-Klein or Klein*, *Poincaré* , *Poincaré half–plane* and *Weierstrass* . A model is called *conformal* if it preserves the hyperbolic angles and *non–conformal* if it distorts them. The *Poincaré* and *Poincaré half–plane* models are conformal while *Beltrami-Klein or Klein* and *Weierstrass* models are not. We are primarily interested in the Poincaré and the Weierstrass models, since those are the models

[1]Boundary in a boundary–less model (such as Klein model) means the actual boundary of the shape (a disk in Klein model) that represents the model.

which are used in our replication algorithms. But we will briefly describe the other models for completeness.

Beltrami-Klein model The Beltrami-Klein model consists of an open Euclidean unit disk[2] in which open chords represent hyperbolic lines. It is known more commonly as the Klein model and is a non-conformal model of hyperbolic geometry. There is an isomorphism between Klein and the Poincaré models. Please see [6] and [7] for detailed descriptions.

From the software implementation point of view, the fact that chords represent hyperbolic lines can be used to efficiently approximate hyperbolic lines in the Poincaré model[3].

Weierstrass model The Weierstrass model consists of the upper sheet of the hyperboloid of revolution $z = \sqrt{1 + x^2 + y^2}$ in Euclidean 3-space. Hyperbolic lines are the intersections of the hyperboloid with planes through the origin. The Poincaré model is a projection of the Weierstrass model.

Poincaré upper–half plane model The Poincaré upper–half plane model is usually represented by the upper half of the complex plane. The base B of the upper half plane is the x-axis, and is not included in this conformal model. Hyperbolic lines are either upper half circles orthogonal to B or vertical rays perpendicular to B. Please see [8] for a reference.

Poincaré model The Poincaré model again consists of an open Euclidean unit disk but hyperbolic lines are represented by arcs of circles that are orthogonal to the boundary circle, plus the diameters of the boundary circle. The Poincaré model is conformal, hence all the angle calculations are simplified. There are isomorphisms

[2]only the interior of the disk is used not the boundary

[3]This was used in earlier implementations. Hyper*Art* does not use the Klein model.

between the Poincaré model, Weierstrass model and the Klein model. There is a good description of this model in [7].

Hyper*Art* uses the Poincaré model for creating designs and the Weierstrass model for symmetry transformations used to generate the design. Two different models are used because the Weierstrass model makes it easier to do all the transformations, and interconversion between two models is very straightforward. Points in each of these models can be projected to the other using the following projections,

$$\text{Weierstrass to Poincaré : } \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \frac{1}{1+z} \begin{pmatrix} x \\ y \end{pmatrix}$$

$$\text{Poincaré to Weierstrass : } \begin{pmatrix} x \\ y \end{pmatrix} = \frac{1}{1-x^2-y^2} \begin{pmatrix} 2x \\ 2y \\ 1+x^2+y^2 \end{pmatrix}$$

2.2 Terminology

Now we will look at some terminology used in this thesis. Most of these terms are used commonly in the mathematics of tessellations. Please see [9] for reference. Hyper*Art* arranges the design into hierarchies. This simplifies both logically and implementation-wise the generation and visualization of the design. An *element* is at the lowest level of the hierarchy; the element concept will be implemented by the *Element* class. An element is some basic geometrical figure. Currently Hyper*Art* supports Euclidean and hyperbolic polylines and polygons and Euclidean circles as elements. It is possible to provide other types of geometric figures in later versions of the program. A *pattern* is a collection of elements. A *layer*[4] is a collection of patterns. Finally a *design* or a *diagram* is a collection of layers. The pattern, layer and the diagram concepts will be implemented by the *Pattern*, *Layer* and the *Diagram* classes respectively.

[4]We will describe layers in more detail when we discuss the algorithms.

We define a *p-gon* as a p–sided hyperbolic polygon. We also define a *central p-gon pattern* of a *repeating pattern* as a pattern which remains invariant under certain transformations of the hyperbolic plane.

A Regular Tessellation $\{p, q\}$ is a covering of the hyperbolic plane by regular p-gons meeting only edge to edge and vertex to vertex, with q meeting at a vertex (see [9] and [10]). Whether the tessellation is hyperbolic, Euclidean or spherical is determined by the values of p and q as follows.

$$(p-2)(q-2) \begin{cases} > 4 & \text{Hyperbolic} \\ = 4 & \text{Euclidean} \\ < 4 & \text{Spherical} \end{cases} \tag{2.1}$$

A non–regular tessellation is specified by $p, q_0 \cdots q_{p-1}$ is a covering of the hyperbolic plane by non–regular p-gons metting only edge to edge, q_i at the i^{th} vertex.

In general from [11], we can use the following relation to determine if the tiling is in the hyperbolic, Euclidean or spherical plane.

$$\sum_{i=0}^{p-1} \frac{1}{q_i} \begin{cases} < \frac{p}{2} - 1 & \text{Hyperbolic} \\ = \frac{p}{2} - 1 & \text{Euclidean} \\ > \frac{p}{2} - 1 & \text{Spherical} \end{cases} \tag{2.2}$$

Note that 2.1 is a special case of 2.2 when the tiling is regular.

A color symmetry is defined as a permutation of n colors in the copies of the fundamental region according to the transformations in the symmetry group.

2.2.1 Symmetry groups

A symmetry of a pattern can be defined as a congruence or an isometry of the hyperbolic plane which transforms the pattern onto itself. The set of all symmetries or all transformations of the hyperbolic plane which preserve a pattern is called the *symmetry group* of that pattern. A *fundamental region* for a symmetry group is a region in the hyperbolic plane which when transformed by all the transformations in the symmetry group, will cover the hyperbolic plane without gaps or overlaps (except

along the edge). A *motif* or *fundamental pattern* is a pattern within the fundamental region. If this pattern is interlocking and covers the whole of the fundamental region then the hyperbolic plane is said to be filled with interlocking copies of the motif, since the motif and the fundamental region can be considered to be the same.

There are infinitely many symmetry groups for a particular class of tessellations. For a regular tessellation $\{p, q\}$, the following are some significant symmetry groups. Many of Escher's hyperbolic designs belong to one of these symmetry groups. Please see [10] for details.

Symmetry group $[p, q]$ This is a symmetry group which is generated by the following reflection transformations,

- Reflection across a p-gon edge.
- Reflection across the perpendicular bisector of a p-gon edge.
- Reflection across a radius[5] of the p-gon.

For this group the fundamental region is a hyperbolic right triangle with acute angles $\frac{\Pi}{p}$ and $\frac{\Pi}{q}$, formed by the 3 axes of reflection symmetry mentioned above. These axes form a natural boundary[6] for the fundamental pattern.

Symmetry group $[p, q]+$ This is a symmetry group which is generated by the following orientation preserving transformations,

- Rotation of order p about the p-gon center.
- Rotation of order q about the p-gon vertexes.
- Rotation of order 2 about the center of the p-gon edges.

A fundamental region is an isosceles triangle with angles $2\frac{\Pi}{p}$, $\frac{\Pi}{q}$ and $\frac{\Pi}{q}$, formed by joining 2 fundamental regions of $[p, q]$. However there is no natural boundary for this fundamental region, since we only have axes of rotation.

[5] line joining p-gon center to a p-gon vertex

[6] A reflection axis forms a natural boundary, beyond which the fundamental pattern cannot extend

Symmetry group $[p+, q]$ This is a symmetry group which is generated by the following transformations,

- Rotation of order p about the p-gon center.
- Reflection across a p-gon edge.

Its fundamental region is same as the fundamental region for $[p, q]+$. Since there is only one axis of reflection (the base of the isosceles triangle or the p-gon edge), it forms the natural boundary for the motif. The value q has to be even in order to constrain the reflective symmetry to p-gon edges.

Symmetry group $[p, q+]$ This is a variation of the group $[p+, q]$ and is generated by the following transformations,

- Rotation of order q about the p-gon vertexes.
- Reflection across a p-gon edge-bisector.

The fundamental region for this group is a kite shaped area formed by joining two adjacent fundamental regions of $[p, q]$ along their common p-gon radius. Only the two p-gon edges in the kite form a natural boundary. Here, p must be even in order to constrain the reflective symmetry to p-gon vertexes.

2.2.2 Classification of replication algorithms

As discussed above, a design is generated by covering the hyperbolic plane by transformed copies of the central p-gon pattern. This process is called *replication*. Depending on the way the replication is done the algorithms can be classified as follows,

1. According to p-gon type
    - Regular p-gon algorithms are based on regular p-gons meeting q at a vertex.

- Non-regular p-gon algorithms are based on non-regular p-gons meeting q_i at a vertex, where q_i is the number of p-gons meeting at the i^{th} vertex.

2. According to replication order

   - Hamiltonian methods are queue based and generate the design layerwise. They are non-recursive methods and might be affected by round-off errors since the next transformation is built upon the previous one.
   - Spanning-tree methods are recursive stack based methods. They generate the design not layerwise but by following a spanning tree of the current p-gon. Hence a p-gon in layer 2 might be generated before a p-gon in layer 3. These are less susceptible to round-off since not all transformations are built upon the previous one. Hyper*Art* currently uses spanning tree methods.

3. According to *central* p-gon

   - P-gon center at the origin : Hyper*Art* currently uses this type of algorithm for regular tilings.
   - P-gon vertex at the origin : Hyper*Art* currently uses this type of algorithm for non-regular tilings.

Different combinations of above methods are possible and used in generating hyperbolic designs.

3. ALGORITHMS

Now that we know some basic theory about tilings in the hyperbolic plane, we are ready to look at the replication algorithms. Hyper*Art* currently implements a *spanning tree, p-gon center at the origin* algorithm for regular tilings and a *spanning tree, p-gon vertex at the origin* algorithm for non-regular tilings.

Let start by looking at some concepts used in the replication of algorithms. Since there are infinitely many symmetry group, to describe them we use the concept of *edge adjacency* and *orientation*. An edge of a p-gon in the current layer is adjacent to some edge of a p-gon in the next layer. Usually we specify this adjacency for the central p-gon since it is most intuitive but it is true for any two adjacent p-gons in consecutive layers. This adjacency alone will not give us information about the transformations, since we also have to specify the orientation of an edge. The orientation can be either *reflection* or *rotation*. Together, adjacency and orientation specify how a p-gon in the next layer can be generated from the current p-gon.

We have defined a layer informally as a collection of patterns. More precisely, we define layers inductively as follows (see [10]). The 0^{th} layer is the central p-gon. $(k+1)^{st}$ layer is the set of those p-gons not in any previous layer, but which share an edge or a vertex with the p-gons in the k^{th} layer. See Figure 3.1 where first five layers (0 to 4) for the frame of the leaf design are shown.

The exposure of a p-gon vertex in the k^{th} layer is then defined as the number of p-gons in the $(k+1)^{st}$ layer which share that vertex. A p-gon vertex can have either minimum exposure or maximum exposure depending on the layer to which it belongs and the tessellation. Figure 3.1 shows exposures for p-gon vertexes in the

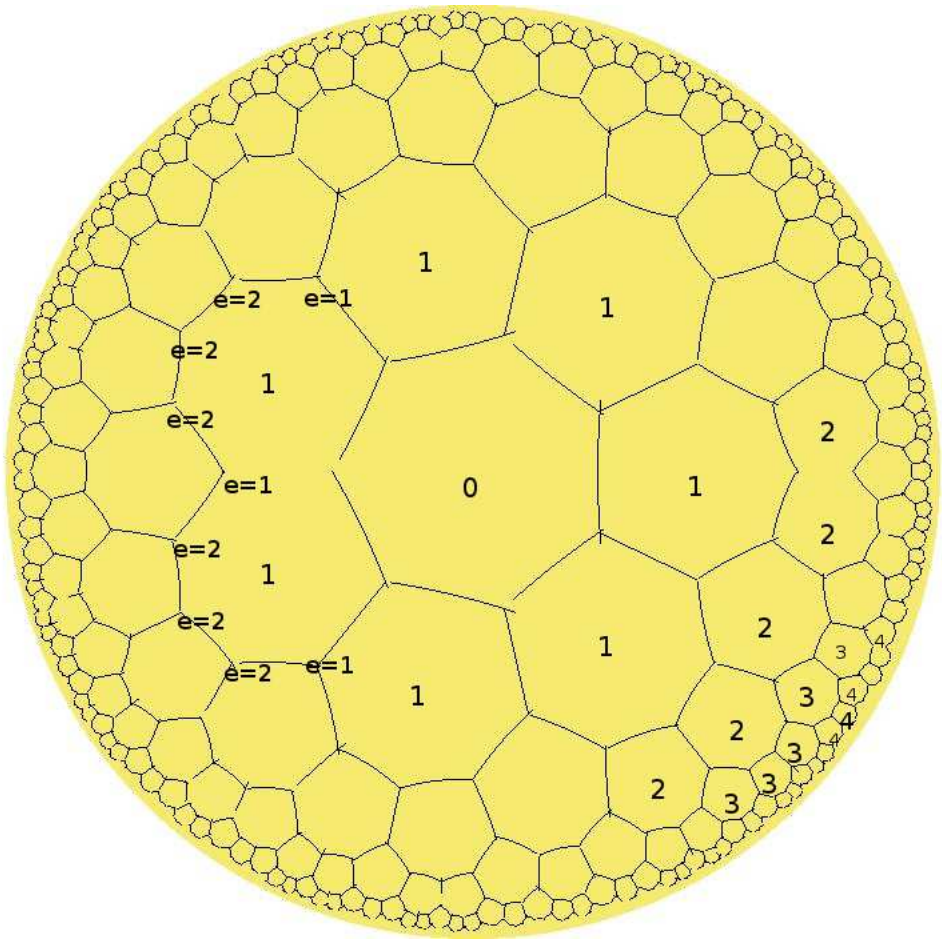


Fig. 3.1. First 5 layers of the regular tessellation $\{7, 3\}$. Also shown is the exposure of some vertexes.

2nd layer. Here, $e = 1$ is minimum exposure whereas $e = 2$ is maximum exposure for this tessellation.

In order to replicate the central p-gon pattern over the hyperbolic plane according to the symmetry group, we associate transformations with the edges of the p-gon. Edge transformations depend on the orientation of the edge. Edge transformations vary slightly according to the class of replication algorithm used. For example, it is different for "p-gon vertex at the origin" compared to "p-gon center at the origin", but the basic idea remains the same. We define all other transformations with respect to the 0^{th} edge of the central p-gon. The central p-gon is aligned such that the origin and the center of the 0^{th} edge lie on the x axis. So in order for these transformations to work for other edges we use edge transformations. For example, symmetry group $[p+, q]$ has two transformations: rotation around the center of a p-gon edge and reflection across a p-gon edge, both of which are defined with respect to the 0^{th} edge of the central p-gon. We also have adjacency and orientation information about each of the p edges. Let the current p-gon edge be i and the edge of the p-gon in the next layer adjacent to it be e. We then build the edge transformation across edge i as follows:

1. First rotate the p-gon pattern clockwise about the origin so that the e^{th} edge coincides with the 0^{th} edge.

2. Then apply the transformation depending on edge orientation. In our example it is *Reflection across p-gon edge* since edge-orientation is reflection.

3. Now rotate the p-gon pattern anti-clockwise about the origin so that we have this transformed copy of the p-gon pattern across i^{th} edge.

We repeat this process for each of the p edges.

For non-regular and other algorithms we might add more transformations to edge transformation, but the crux is as mentioned above.

Shown in Figure 3.2 is a rendition of the Circle Limit IV pattern. Since the symmetry group is from the class $[p, q+]$, it has two transformations which are defined

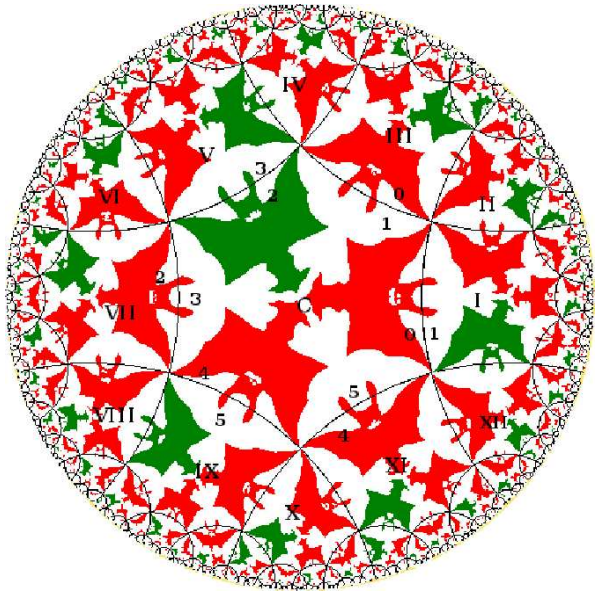


Fig. 3.2. Circle Limit IV – with symmetry group $[6, 4+]$ and adjacency of the edges labeled.

with respect to the 0^{th}*edge – Rotation of order q around p-gon vertex* and *Reflection across a p-gon edge-bisector*. The adjacency is as shown (decimal numbers). The orientation of each edge is Rotation. Edge 1 in a p-gon labeled C is adjacent to edge 0 p-gon labeled III. So p-gon C is first rotated clockwise so that edge 1 coincides with edge 0. Then it is rotated around edge 0, since the orientation is rotation. This gives us the transformed copy which is to be put as p-gon III. But p-gon III is across the 1^{st} edge of p-gon C, so it is again rotated anti-clockwise to coincide with that edge.

Next we will look at specific algorithms to generate regular and non-regular tilings. We will first look at the regular tiling algorithm, since the non-regular algorithm is somewhat related to it.

3.1 Regular tiling algorithm

Some regular tiling algorithms have two steps – (1) generation of the central p-gon pattern, and (2), replication of central p-gon pattern over the hyperbolic plane. All these transformations are done in the Weierstrass model and afterwards projected onto the Poincaré disk. We will briefly describe the algorithms here and focus more on the Hyper*Art* framework in the next chapter. For the original algorithms readers are encouraged to look at [9] and [10]

3.1.1 Generation of central p-gon pattern

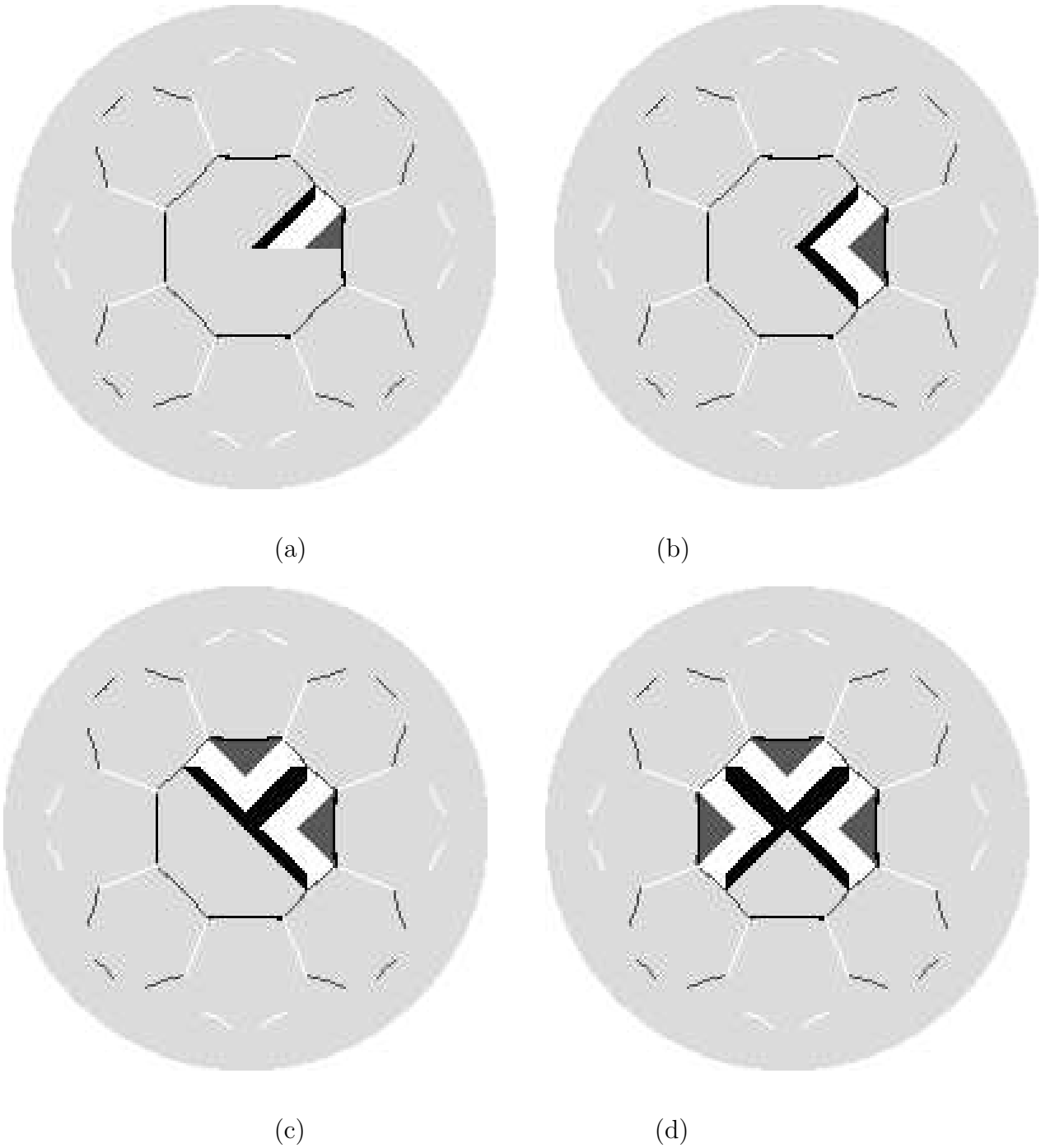
Transformed copies of fundamental pattern make up the central p-gon pattern. The transformations used are reflections and rotation about p-gon center (origin). There are two types of internal reflections: *Reflection across a p-gon radius* and *reflection across a p-gon edge bisector*. Depending on the reflection symmetry (if it is present at all) the fundamental pattern is first reflected, and then copies of this new pattern are rotated around the origin to fill up the central p-gon. If there is no reflection symmetry then the fundamental pattern is simply rotated around the origin

to obtain the central p-gon pattern. Figures 3.3 and 3.4 are examples of reflection across an edge bisector and reflection across a p-gon radius respectively. The figures also show how reflection and rotation transformations generate the central p-gon pattern from the fundamental pattern.

3.1.2 Replication of central p-gon pattern

The central p-gon pattern is then replicated to obtained subsequent layers, the effect of which is to cover the hyperbolic plane. Mathematically, an infinite number of layers are required to completely cover the hyperbolic plane. Practically, however, depending on the pattern, 4 to 5 layers are sufficient. The replication process can be broken down into two parts. If generation of just the 0^{th} layer is requested then we just need to generate the central p-gon pattern. However, if more than 1 layer is requested, then a recursive procedure is invoked which walks a spanning tree of each unvisited polygon, until the requested number of layers are completed. This recursive spanning tree algorithm generates each polygon only once.

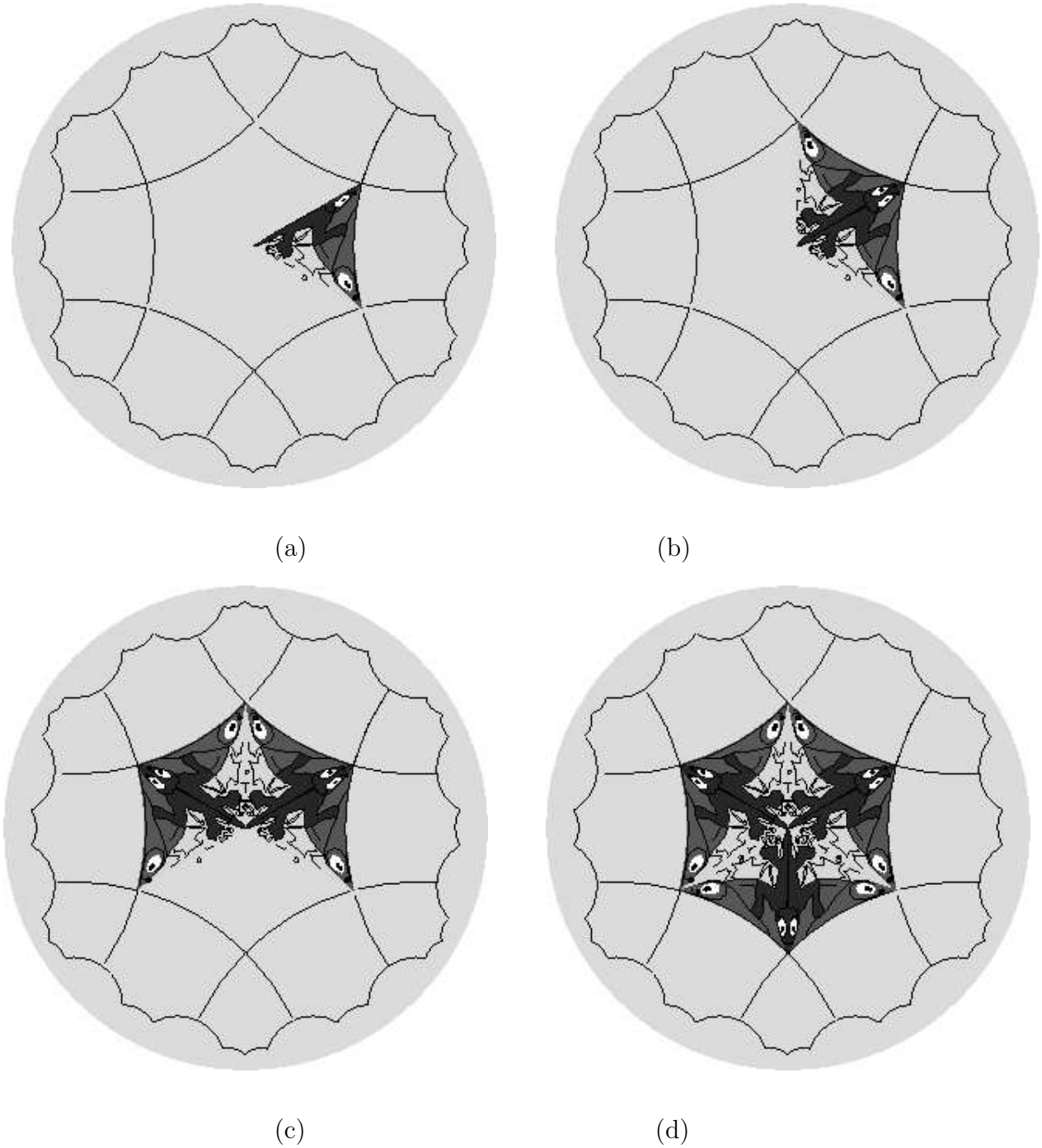
The replication process uses different parameters such as *pskip*, *qskip*, *pgonsTodo*, *verticesTodo* which are dependent on the p and q values and exposure of the vertex being processed. Exposure itself is dependent on the layer, vertex and the p-gon around that vertex that is being processed. The values $p = 3$ and $q = 3$ are special cases so parameters for these cases are different than in the general case. The parameter *pskip* is the number of vertexes to skip when going to a p-gon in the next layer. The parameter *qskip* is the number of p-gons around the current vertex to skip in order to avoid duplication of p-gons. The parameter *verticesTodo* is the number of vertexes of the current p-gon that should be visited. The parameter *pgonsTodo* is the number of p-gons to be generated for the current vertex. The concept of exposure was explained earlier. Table 3.1 shows values of these parameters in the general and the special cases.



(a) (b)

(c) (d)

Fig. 3.3. For the Circle Limit II pattern with symmetry [8.3+], (a) shows kite shaped fundamental region, (b) after one edge bisector reflection, (c)(d) after 1 and 2 rotations respectively of pattern in (b). One more rotation would complete the central p-gon.



(a) (b)

(c) (d)

Fig. 3.4. For Pattern 85 with symmetry [6+, 4], (a) shows isosceles triangle shaped fundamental region, (b) after one p-gon radius reflection, (c)(d) after 1 and 2 rotations respectively of pattern in (b). Here we obtain the central p-gon after 2 rotations.

Parameter	general	$p = 3$	$q = 3$
Exposure for 0^{th} layer [pgon]			
$pgon = 0$	min	min	max
$pgon \neq 0$	max	max	max
Exposure for other layers [vertex,pgon]			
$vertex = 0, pgon = 0$	min	min	min
$vertex = 0, pgon \neq 0$	max	max	min
$vertex \neq 0, pgon = 0$	min	min	max
$vertex \neq 0, pgon \neq 0$	max	max	max
pskip [exposure]			
$exposure = min$	1	1	3
$exposure = max$	0	1	2
qskip [exposure,vertex]			
$exposure = min, vertex = 0$	-1	-1	0
$exposure = min, vertex \neq 0$	0	-1	0
$exposure = max, vertex = 0$	-1	0	0
$exposure = max, vertex \neq 0$	0	0	0
verticesTodo [exposure]			
$exposure = min$	$p - 3$	1	$p - 5$
$exposure = max$	$p - 2$	1	$p - 4$
pgonsTodo [exposure,vertex]			
$exposure = min, vertex = 0$	$q - 3$	$q - 4$	1
$exposure = min, vertex \neq 0$	$q - 2$	$q - 4$	1
$exposure = max, vertex = 0$	$q - 3$	$q - 3$	1
$exposure = max, vertex \neq 0$	$q - 2$	$q - 3$	1

Table 3.1
Parameters for Regular tiling algorithm

Next we show the procedures of the algorithm – make and makeHelper. We omit the special cases $p = 3$ and $q = 3$, readers are encouraged to look at Hyper*Art* [12] code which handles all the cases. The procedures uses following 3 transformations: edge transformations which were explained earlier, ptran for getting to the p-gon to be visited next, and qtran to generate the next p-gon pattern around the current vertex. Figures 3.5 and 3.6 show snapshots of the generation process. The algorithm animation feature of Hyper*Art* is very useful in such visualizations.

Procedure `make(`*numLayers : int*`)`

Generate the central p-gon pattern from the fundamental pattern using internal reflection and rotation, and add it to 0^{th} *layer. This is the only pattern in the* 0^{th} *layer.*

```
makeCentralPgonPat
if numLayers > 1 then
    for i=0 to  p-1 do
        Set initial transformation from the corresponding edge
        qtran ← edgeTran_i
        Visit p-gons around this vertex
        for j=0 to  q-3 do
            Visit the spanning tree for current p-gon
            makeHelper(exposure←exposure(0, i, j), layerId ←1,tran ←qtran)
        Modify qtran so that we can visit the next p-gon around vertex i
        qtran ←shiftTran(qtran,-1)
```

3.2 Non-regular tiling algorithm

Non-regular tiling is in many ways similar to the regular tiling. In fact regular tiling is a special case of non-regular tiling. The most notable difference in non-regular tiling is that now we have p possibly different q values – that is, a q value for each of the p vertexes of the p-gon. Hyper*Art* currently implements the "p-gon

Procedure `makeHelper`(*exposure : Exposure, layerId : int, tran : Transformation*)

```
Add the current copy of the pattern to current layer
addPattern(layer←layerId, pattern←pat)
if layerId < numLayers then
    Get pskip and verticesTodo
    for i=0 to verticesTodo- 1 do
        Get qskip and pgonsTodo
        modify ptran and qtran– special cases not shown
        ptran ←shiftTran(ptran,pskip)
        qtran ←shiftTran(ptran,qskip)
        for j=0 to pgonsTodo- 1 do
            Get exposure for current layerId, i, j
            Visit all the p-gons near this one
            makeHelper(exposure, layerId ←layerId +1,tran ←qtran)
            qtran ←shiftTran(qtran,-1)
        pskip ←(pskip +1) mod p
```

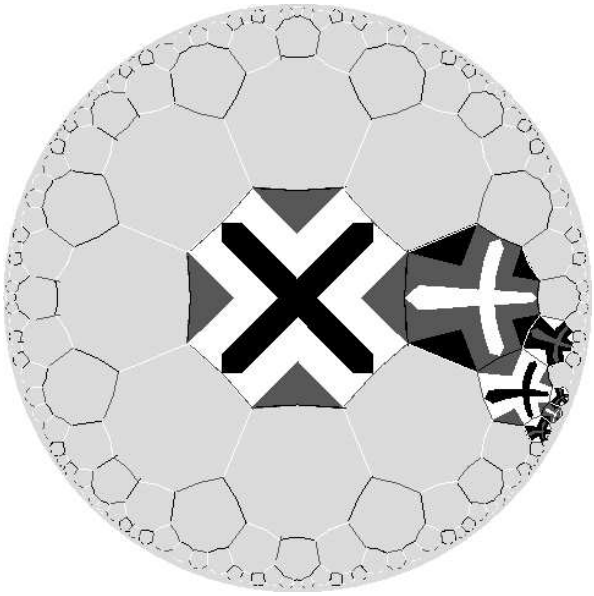


Fig. 3.5. Generation of the Circle Limit II pattern using the spanning tree method.

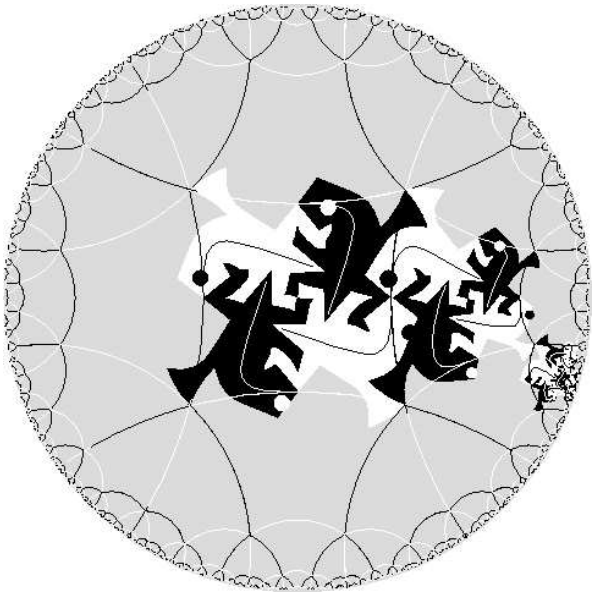


Fig. 3.6. Generation of Pattern 104 using the spanning tree method.

vertex at the origin" algorithm to generate non-regular[1] tilings. A direct consequence of the vertex-at-origin algorithm is that the concept of layers is slightly less intuitive. Now the "central p-gon" is not really a *central* p-gon but the starting p-gon from which rest of the p-gons are generated. For the need of keeping the terminology simple and consistent, we say that this *central* p-gon is still in layer 0. The group of p-gons whose vertexes meet at the origin are said to be in the next layer (layer 1). Another interesting point to note is the fundamental region. For non-regular p-gons the whole p-gon itself is the fundamental region. Because of this there is no need for a *generate central p-gon* procedure.

There are no drastic changes from the regular p-gon algorithm to non-regular p-gon algorithm. The overall structure of the algorithm remains similar. The algorithm differs in how *edge transformations* for each edge are constructed and the way parameters are calculated. Parameters for non-regular case are similar to the general case of the parameters for regular algorithm (Table 3.1). In fact the parameters for the special cases ($p = 3$, or $q_i = 3$) are same as those of general case. However we calculate pgonsTodo from the parameter pgonsToSkip using the corresponding value of q as follows:

$$pgonsTodo = q_{vert} - pgonsToSkip_i^{exposure}$$

Here i is the current vertex around which p-gons are being visited and *vert* is the adjacent vertex. The values of pgonsToSkip are given in table 3.2.

We will look at how edge transformations are computed before looking at the algorithm and examples. As mentioned earlier, only those non-regular p-gons satisfying the condition in Equation 2.2 can form a non-regular hyperbolic tiling. We use the concept of an in-circle[2] to build transformations. Not all non-regular p-gons have in-circle and only those that have an in-circle are used by our algorithm. Remember that we are dealing with a vertex-at-origin algorithm here. The transformations however are built first with p-gon center (center of the in-circle) at the origin and

[1] called as IrregularPgon in the Hyper*Art* code

[2] A unique circle that is tangent to each of the sides of the non-regular p-gon

Parameter	general
pgonsToSkip [exposure,vertex]	
$exposure = min, vertex = 0$	$q - 3$
$exposure = min, vertex \neq 0$	$q - 2$
$exposure = max, vertex = 0$	$q - 3$
$exposure = max, vertex \neq 0$	$q - 2$

Table 3.2
Parameters for Non-Regular tiling algorithm, only those parameters that are different from table 3.1 are shown. The rest of the parameters are same as the general column of table 3.1.

then the p-gon is translated so that the required vertex of the p-gon lies on the origin.

Figure 3.7 shows the non-regular p-gon and its in-circle. The figure shows the p-gon with the center of its in-circle at the origin. Each edge transformation is calculated with respect to this position. The angles and parameters used in the calculations are shown in the figure. A *move transformation* is then applied to the p-gon and the edge transformations, such that the vertex V lies at the origin. The edge transformations are calculated only once for each edge. The replication algorithm then uses these pre-calculated edge transformations to replicate the *central* non-regular p-gon pattern. For exact transformation matrices please refer to the Hyper*Art* code.

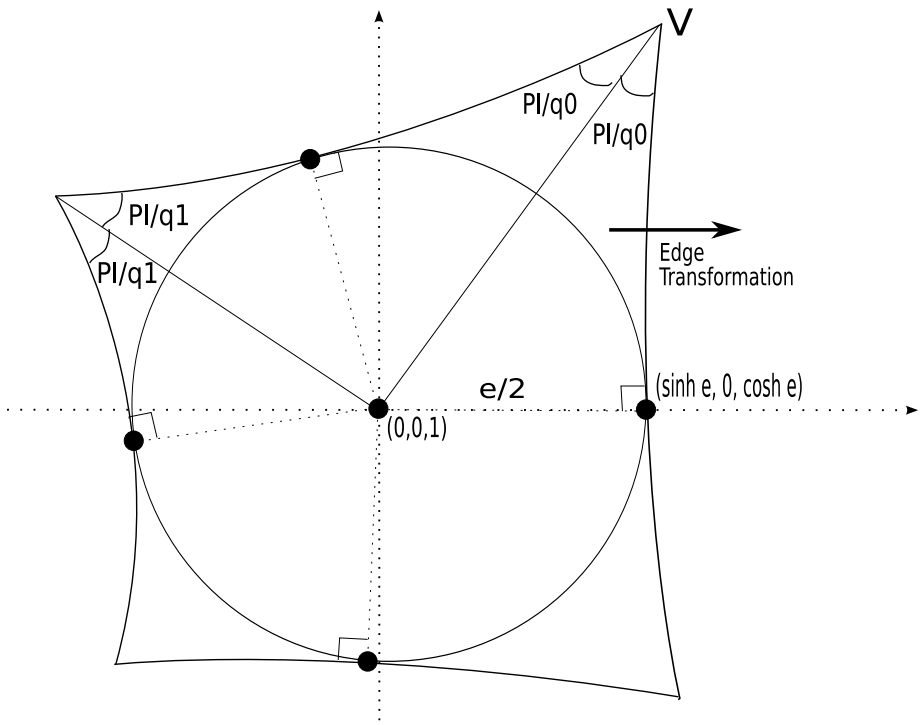


Fig. 3.7. Nonregular p-gon – Use of in-circle and calculation of edge transformations. All co–ordinates are in the Weierstrass space.

Now we will look at the procedures makeNR and makeHelperNR. They are not very different from the regular p-gon procedures. We have already discussed how parameters differ for non-regular algorithm. There is no makeCentralPgonPattern since the non-regular p-gon itself is the motif.

Procedure `makeNR(`*numLayers : int*`)`

```
if numLayers > 1 then
    Add the first p-gon pattern to layer 0
    for i=0 to q_0 − 1 do
        Start visiting p-gons around vertex 0
        qtran ← edgeTran_0
        makeHelperNR(MAXEXPOSURE, layerId ←1,tran ←qtran)
        Modify qtran so that we can visit the next p-gon around vertex 0
        qtran ←shiftTran(qtran,-1)
```

A look at the snapshots of the spanning tree algorithm for a non-regular pattern will make this algorithm easier to visualize. Figures 3.8 and 3.9 below show 8 steps of non-regular vertex-at-origin algorithm. The p-gon has 5 sides and starting from top-right vertex (which is at the origin), there are 4,4,3,4,4 p-gons around each vertex. Layers 0 to 2 inclusive are being generated.

Figure 3.10 shows the same p-gon being replicated over first 4 layers (0 to 3). Note that what we perceive as central-pgon is now different than the regular algorithm.

Procedure `makeHelperNR`(*exposure : Exposure, layerId : int, tran : Transformation*)

```
Add the current copy of the pattern to current layer
addPattern(layer←layerId, pattern←pat)
if layerId < numLayers then
    Get pskip and verticesTodo
    for i=0 to verticesTodo- 1 do
        Get qskip and pgonsToSkip
        modify ptran and qtran– special cases not shown
        ptran ←shiftTran(ptran,pskip)
        qtran ←shiftTran(ptran,qskip)
        Get v the adjacent vertex
        pgonsTodo ←q_v − pgonsToSkip
        for j=0 to pgonsTodo- 1 do
            Get exposure for current layerId, i, j
            Visit all the p-gons near this one
            makeHelperNR(exposure, layerId ←layerId +1,tran ←qtran)
            qtran ←shiftTran(qtran,-1)
        pskip ←(pskip +1) mod p
```

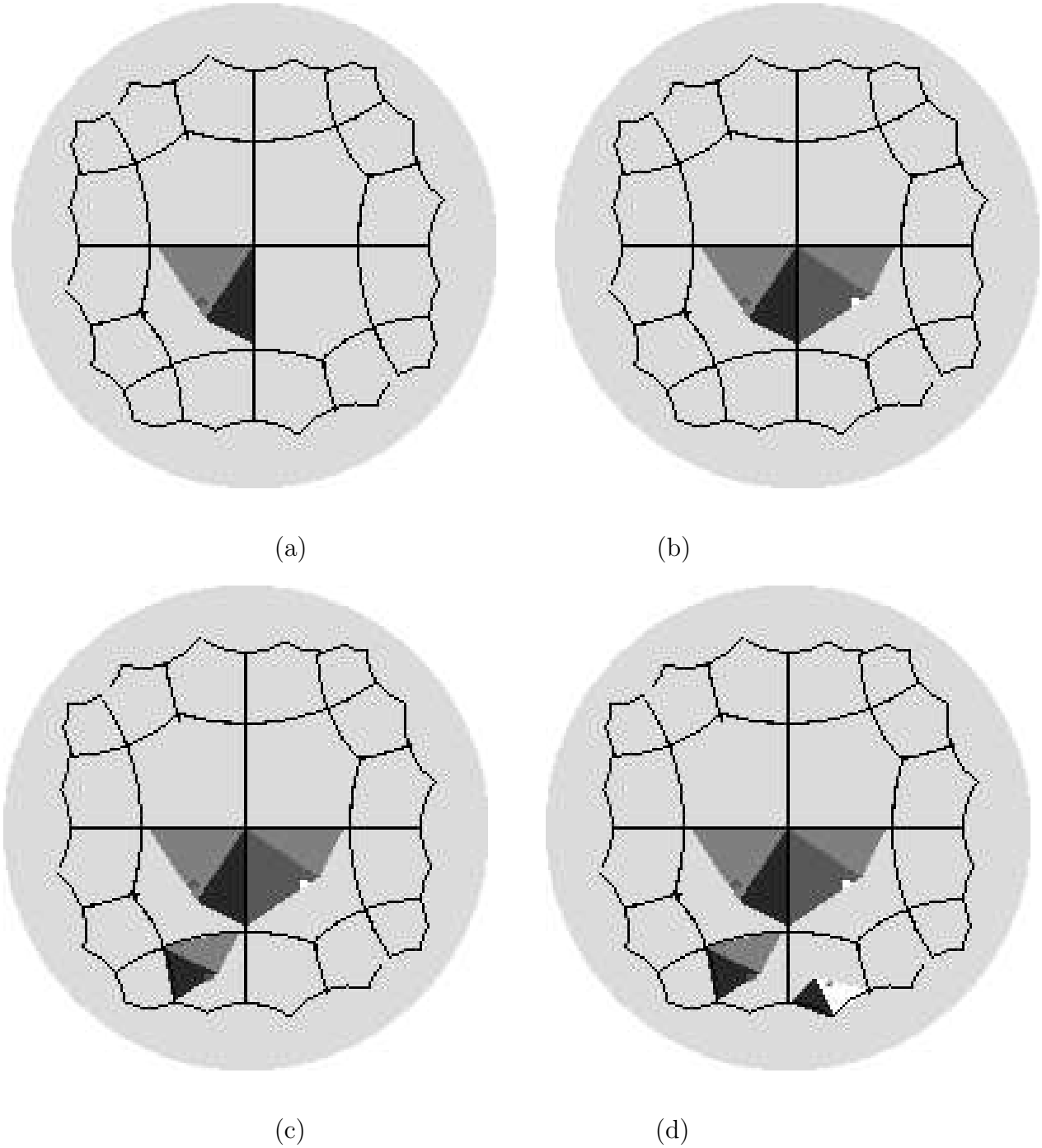
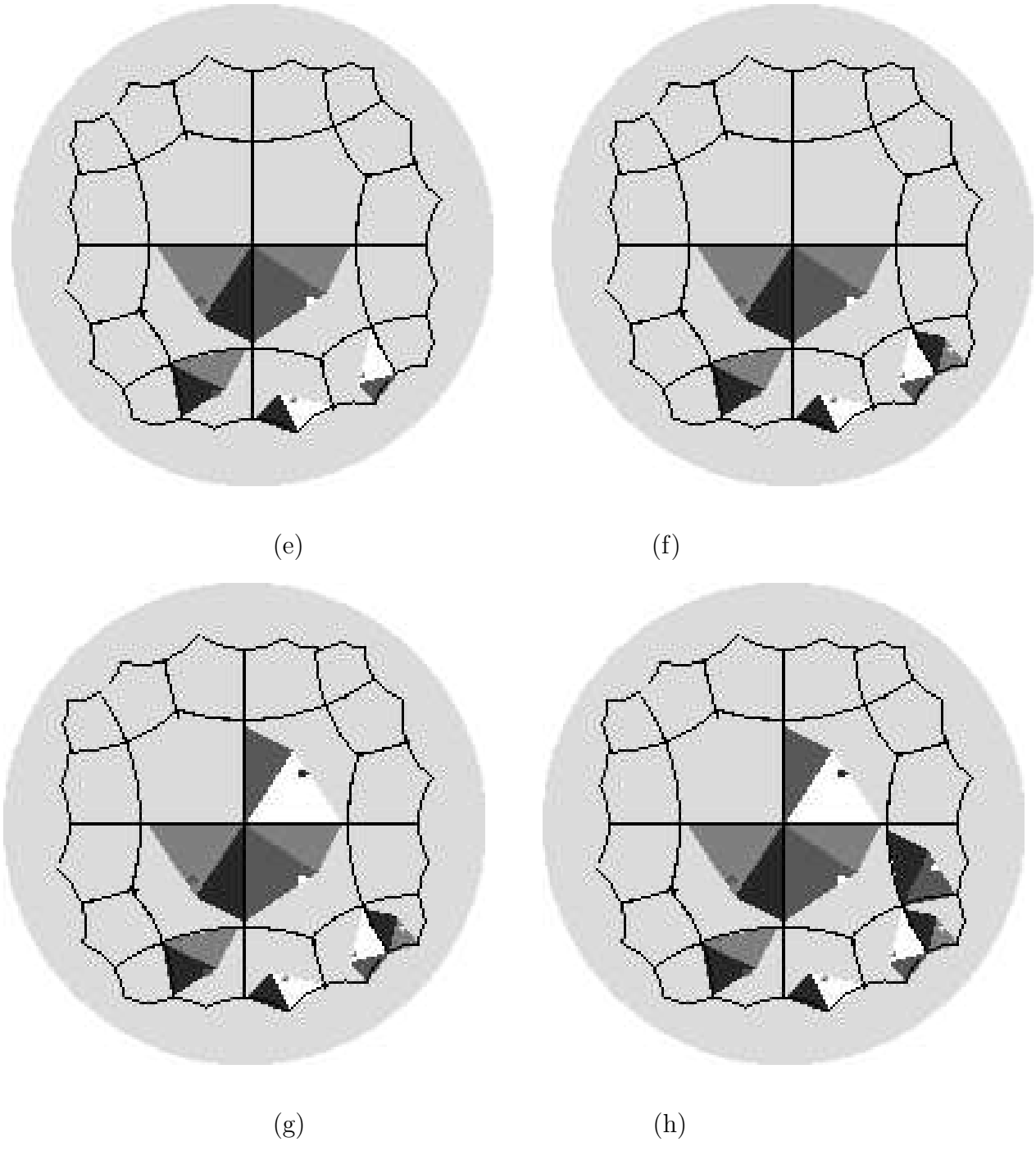


Fig. 3.8. Steps in generation of non-regular p-gon with $p = 5$ and q values of 4,4,3,4,4 around each of the p vertexes. (a) shows the fundamental region which is same as the first p-gon – this is also the 0^{th} layer (b)(c)(d) show subsequent steps.



(e) (f)

(g) (h)

Fig. 3.9. Steps in generation of non-regular p-gon with $p = 5$ and q values of 4,4,3,4,4 around each of the p vertexes – continued. (e)(f)(g)(h) show further steps.

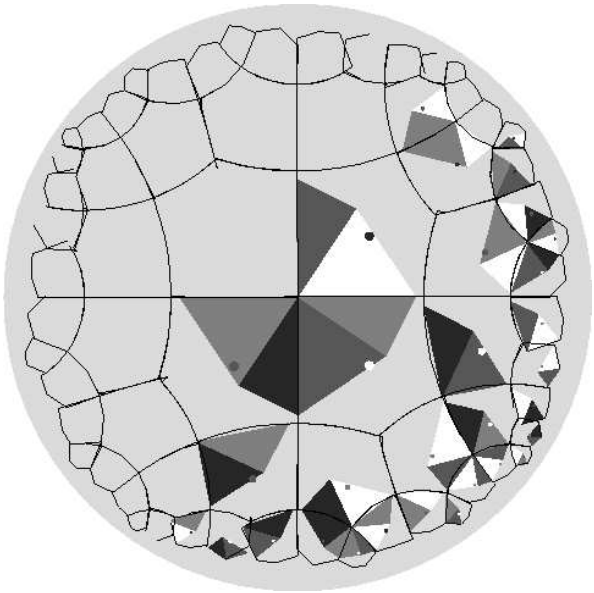


Fig. 3.10. A snapshot while generating 4 layers of non-regular p-gon algorithm with p-gon $p = 5$, $q_i = 4, 4, 3, 4, 4$

4. Hyper*Art* FRAMEWORK

We will now present our cross platform $C++$ class framework Hyper*Art* which helps us experiment with the algorithms presented earlier. Hyper*Art* is a implementation of the algorithms discussed earlier. Although it is based on earlier programs written by Dr. Douglas Dunham and other graduate students, Hyper*Art* introduces many new features not found in earlier software. We feel that through Hyper*Art* it should be straightforward to unify all the different types of existing and new algorithms and present them under one uniform interface. Some of the features of Hyper*Art* are as follows:

- From the user perspective
    1. XML data file format which is easy to understand and extend
    2. Crossplatform – Written in Qt from trolltech [13], it can run on many different platforms
    3. Single unified interface for different types of algorithms
    4. Layer toggling
    5. Frame toggling
    6. Algorithm animation and stepping
    7. A modern UI with zooming, panning and printing support
    8. Diagram export to popular image formats and printing to postscript file.
- From the developer/researcher perspective
    1. Document view architecture achieves good separation of algorithms and design presentation

2. Abstraction of Diagram and View classes makes it easy to extend and add algorithms[1], and present the diagram in different views[2]
3. Importing old dat files to the new XML format.
4. Instead of pen based drawing used earlier, Hyper*Art* introduces *elements*. It also works with a hierarchy of diagram parts such as patterns and layers.
5. Source-code available under GPL license on sourceforge.net.

Hyper*Art* currently lacks a diagram designer – this should be the immediate development goal. It provides a script to import old *dat* files to the native XML format, and you can also create XML design files by hand. However for creating new designs, an interactive designer mode for Hyper*Art* is essential.

4.1 Important Hyper*Art* classes

We will now present some important classes[3] in Hyper*Art* . Understanding these classes is important for extending the program and for adding new algorithms. Hyper*Art* uses the Document view architecture. and is written in Qt, the cross-platform C++ toolkit from trolltech (see [13]). We will not be discussing implementation specifics of Hyper*Art* with respect to Qt but general class design. If you are interested in developing Hyper*Art* please see [12]. Documentation for Hyper*Art* can be generated from source code using the Doxygen documentation generator. Figure 4.1 shows how different Hyper*Art* classes are related to each other.

[1]Hyper*Art* is already been used for animating Hamiltonian path algorithms.

[2]Currently the Poincaré view is implemented.

[3]Hyper*Art* is continuously evolving, though most of the framework is in place. For the latest class design please see the hyperart website [12].

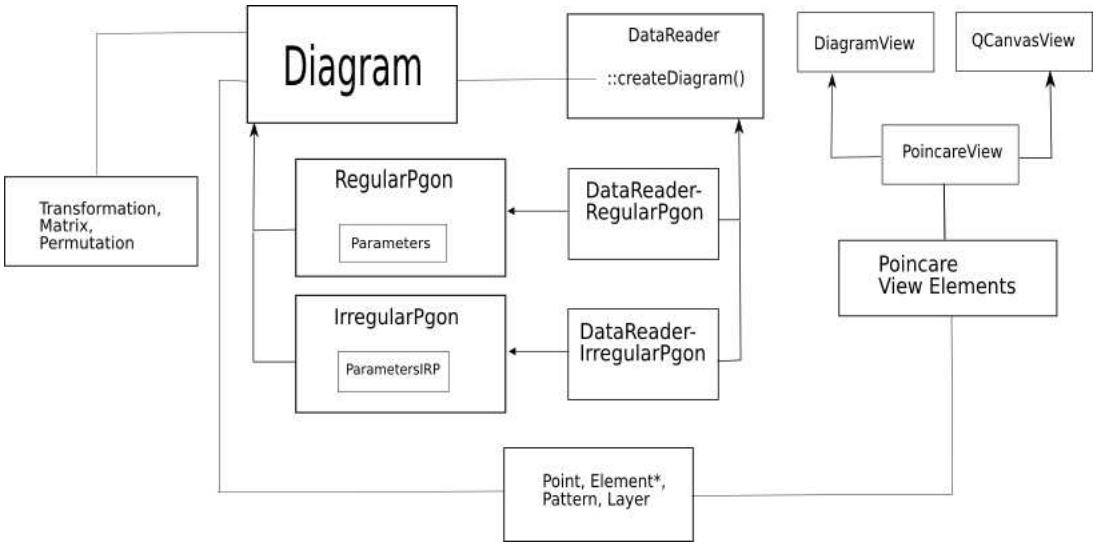


Fig. 4.1. An overview of the Hyper*Art* classes.

4.1.1 Diagram classes

Diagram classes are the document classes. They handle generation of diagrams and storing diagram data. They do not deal with presentation of diagrams. All diagram parts (described below) are independent of the presentation classes.

Diagram is the abstract base class for all types of diagrams. It provides an interface which all types of diagrams should implement. It also implements the most common functionality of the classes. Diagram should not be directly instantiated but one of it's derived classes should be used. This is the Document class in the Document-View architecture. It provides abstract interface methods such as init(), clear(), type() and make(). All derived classes should implement these methods. The numLayers property of the Diagram class should be used by make() to generate the requested number of layers. Depending on the algorithm, make() could be a recursive or non-recursive procedure and the decision is the implementor's choice. The addPattern() method should be used by make() to add the pattern to the layer to which it belongs and to set up the animation queue properly.

RegularPgon is derived from Diagram and as the name suggests implements the Regular p-gon tiling. This class implements a spanning tree (i.e. recursive) p-gon center-at-origin algorithm. All the transformations are initialized in initTrans(). The *Parameters* class is used to provide the parameters required by this algorithm.

IrregularPgon is derived from Diagram and implements the recursive non-regular p-gon tiling with p-gon vertex-at-origin algorithm. It is similar in sementics to the RegularPgon class. The *ParametersIRP* class is used to provide the parameters required by this algorithm.

Diagram parts

Element is the abstract base class for all basic Diagram elements. An elements is a collection of *Point*s which represents some geometrical construct. Element provides a common interface for all types of elements to add Points and set Element attributes such as color, zorder[4] etc. The transform() method applies the given transform to the element. Each element can clone() itself and has a unique Id. As of now Hyper*Art* supports the following types of element classes – *EuclidPoly*, *EuclidPolyLine*, *HyperPoly*, *HyperPolyLine* and *Circle*.

Pattern is a collection of *Element*s. The Pattern class provides methods to transform and clone itself. It also provides methods to get it's collection of elements. Each pattern has 2 sets of Elements - one representing the Pattern itself and the second representing the bounding frame of the Pattern. Each pattern has a unique Id.

Layer is a collection of *Pattern*s. Each layer has a unique Id and provides methods to manipulate the Patterns in it.

Utility classes

Hyper*Art* provides utility classes such as *Transformation*, *Matrix*, *Permutation* and *IdFactory* which provides the basic functionality used by Diagram classes. These classes provide convenient operations for manipulating these primitives.

4.1.2 Reader classes

Reader classes populate and create the appropriate Diagram class. There should be a reader class corresponding to each Diagram type.

[4]the order in which overlapping elements take precedence

DataReader is the abstract base class for all XML file readers. Along with providing support to read the XML-based file format, it also provides a class method to create an appropriate Diagram class. So in that aspect, it acts as a factory class. *DataReaderRegularPgon* and *DataReaderIrregularPgon* are the derived classes for corresponding Diagram classes.

4.1.3 Presentation classes

The presentation classes present the data in the Diagram classes on the output device. This separation between the document and the view allows presenting the same Diagram in multiple (and maybe different) views.

HyperArt is the MainWindow[5] class which acts as the UI controller. It emits signals (messages in traditional terminology) whenever user requests an action. These signals are handled by the current View. This MainWindow class does not need to be aware of exact view type, since it relies on DiagramView and the signals to carry out actions.

DiagramView is an abstract base class for all Hyper*Art* views. It defines a common interface so that the MainWindow class HyperArt can control the view from its UI. It is intended to be used as the second base class for the view; the first being the class which provides the drawing primitives. It is worth noting now that DiagramView class does not use Qt's object model[6].

PoincareView is derived from *QCanvasView* (the class which provides drawing primitives) and *DiagramView*. As the name suggests it draws a design in the Poincare unit circle. It is connected with the one of the Diagram class objects that it is showing

[5]This class is actually autogenerated from the ui specification file. For changing the UI logic you should modify "hyperart.ui" and "hyperart.ui.h" files from which it is generated.

[6]Multiple inheritance in Qt is tricky, only the first base class should be derived from the QObject class

and synchronizes itself with the changes in the diagram. PoincareView connects itself to the signals sent by the MainWindow class. It implements a hierarchy of drawing methods like *drawDiagram*, *drawLayer*, *drawPattern*, *drawPatternFrame* and *drawElement* along with printing and image export support. Similarly it also handles zooming, panning, animation and other signals from the MainWindow class. It uses a 2D canvas *QCanvas* to draw diagrams.

Canvas Elements are classes which map the *Element* classes to their screen representation in *PoincareView*. These include *CanvasEllipse*, *CanvasPoly*, *CanvasPolyLine*, *CanvasHyperPoly* and *CanvasHyperPolyLine*.

4.2 Hyper*Art* design files

Hyper*Art* design files are written in XML (see [14]). They are based on the earlier plain text files used in program developed by Dr. Douglas Dunham et al and a conversion utility for the same is provided with Hyper*Art* . Being in XML format, these files are human–readable at the cost of slight verbosity. These XML files commonly have an extension *had* which stands for "*h*yper*a*rt *d*esign" and will be called as *had* files here onwards. Presently there is no validation schema or DTD for *had* files. Some validation is done by the Reader classes mentioned before. The *had* files introduce the concept of elements[7] which are groups of points with common attributes, representing a geometrical figure. Appendix A shows a sample *had* file with in-line comments explaining the tags.

[7]As opposed to earlier *dat* files which supported pen based drawing, and hence listed only points.

5. RESULTS

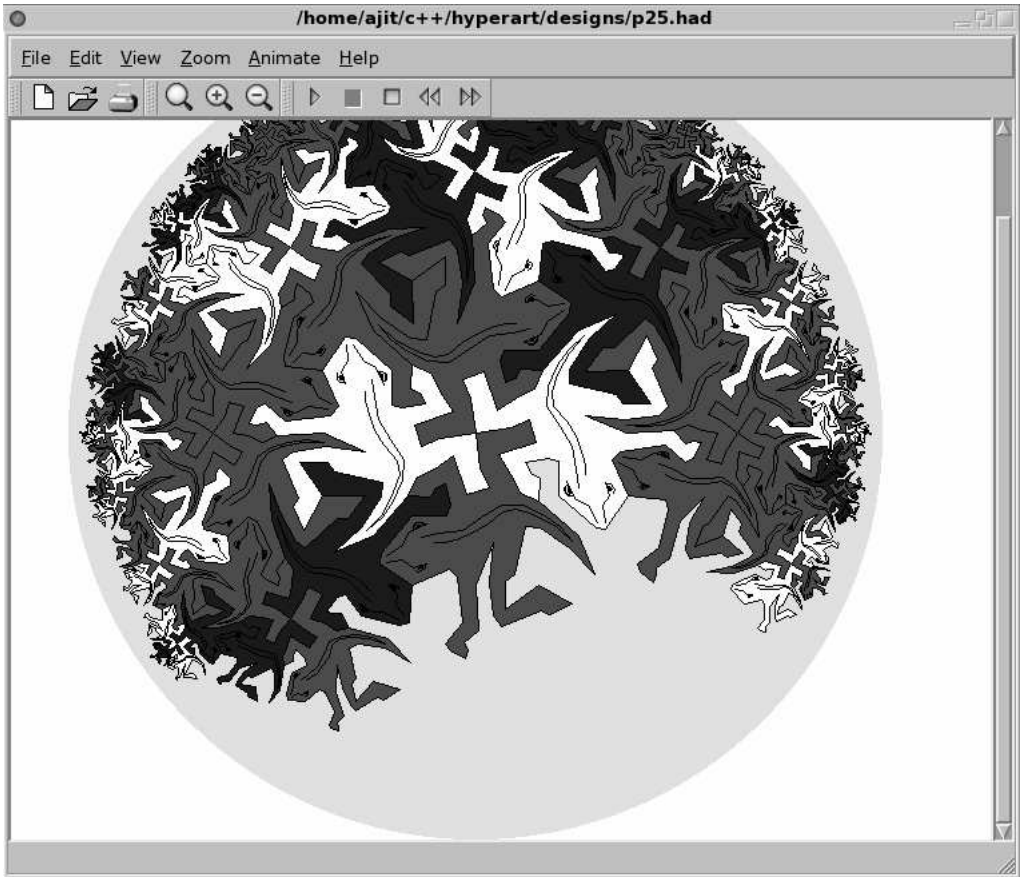


Fig. 5.1. A screenshot of Hyper*Art* in action on a linux machine.

Fig. 5.2. Circle Limit II

Fig. 5.3. Circle Limit III – Bent

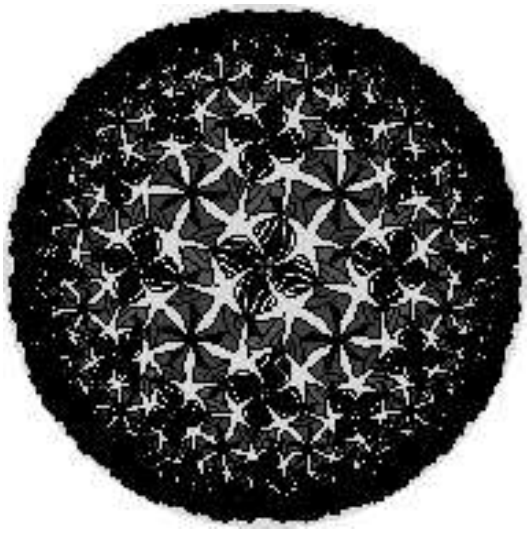


Fig. 5.4. Pattern 42

Fig. 5.5. Pattern 25

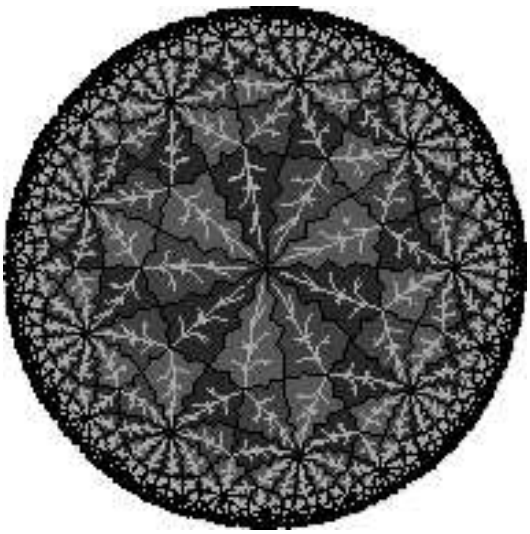


Fig. 5.6. Leaf {9,3}

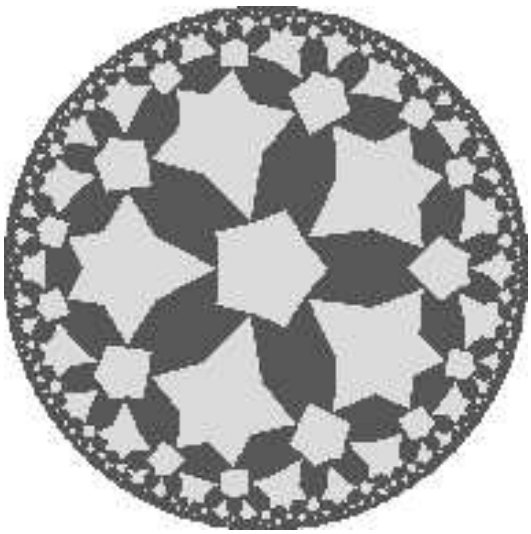


Fig. 5.7. Figure 14

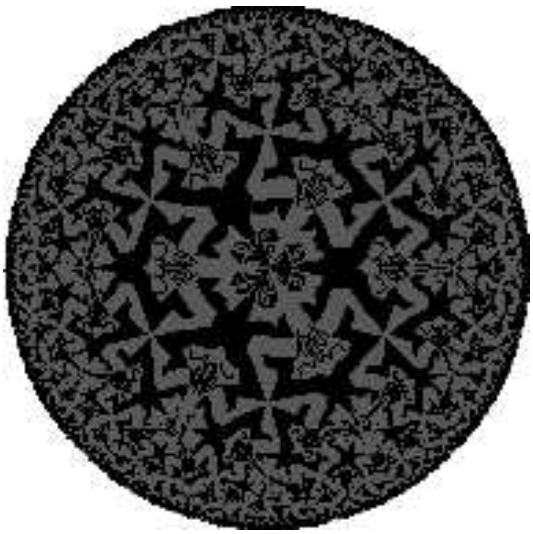


Fig. 5.8. Pattern 53

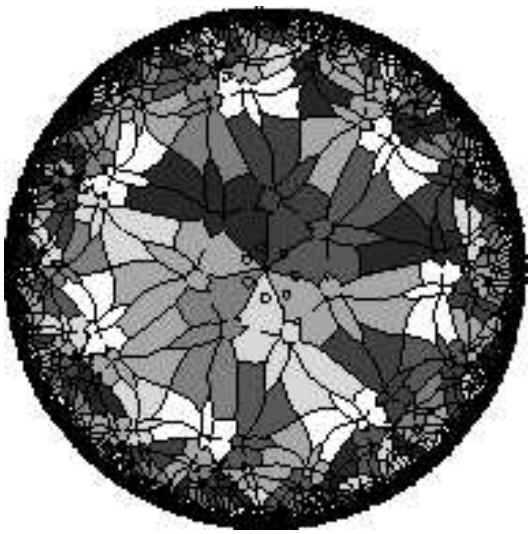


Fig. 5.9. Pattern 70 {7,4}

Fig. 5.10. Pattern 85

Fig. 5.11. Pattern 104 {6,6}

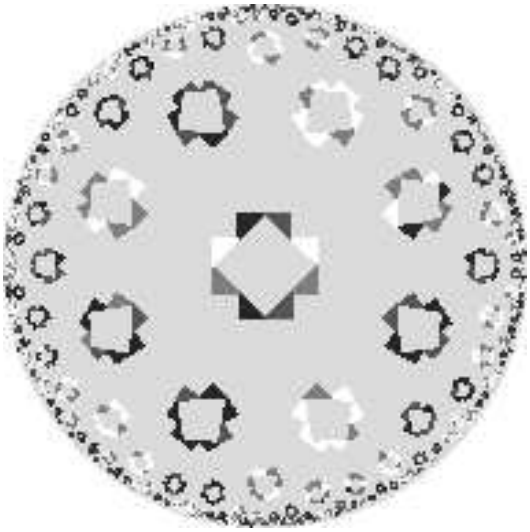


Fig. 5.12. Non-regular design with $p = 4$, $q_i = [4, 6, 3, 6]$

Fig. 5.13. Non-regular design with $p = 5$, $q_i = [4, 4, 3, 4, 4]$

6. SUMMARY AND FUTURE WORK

We have implemented Hyper*Art* – a framework to experiment with hyperbolic tiling algorithms. We have also refined algorithms to generate regular and non-regular tilings. We were able to successfully render many hyperbolic versions of Escher's original designs and their variations. We also tried some new designs.

The following are some possible directions for future work.

- Algorithms research
    1. Experiment with infinite p or q algorithms.
    2. Experiment with Hamiltonian path algorithms.
    3. Develop p-gon–vertex–at–origin algorithm for regular p-gons.
    4. Develop p-gon–center–at–origin algorithm for non–regular p-gons.
- Hyper*Art* improvements
    1. Diagram designer[1]
    2. Make the program memory-smart. Generate appropriate layers depending on element density.
    3. Transformation animation.
    4. Weierstrass view.
    5. Scalable Vector Graphics support.

[1]Probably the "most required" feature

LIST OF REFERENCES

LIST OF REFERENCES

[1] Eric W. Weisstein. Tessellation from mathworld–a wolfram web resource. http://mathworld.wolfram.com/Tessellation.html, 1999–2005.

[2] Richard L. Faber. *Foundations of Euclidean and non-euclidean geometry.* marcel dekker,inc., new york.basel, 1983.

[3] Eric W. Weisstein. Euclid's postulates from mathworld – a wolfram web resource. http://mathworld.wolfram.com/ParallelPostulate.html, 1999–2005.

[4] Eric W. Weisstein. Parallel postulate from mathworld – a wolfram web resource. http://mathworld.wolfram.com/EuclidsPostulates.html, 1999–2005.

[5] Eric W. Weisstein. Hyperbolic geometry from mathworld – a wolfram web resource. http://mathworld.wolfram.com/HyperbolicGeometry.html, 1999–2005.

[6] Eric W. Weisstein. Klein-beltrami model from mathworld – a wolfram web resource. http://mathworld.wolfram.com/Klein-BeltramiModel.html, 1999–2005.

[7] David C. Royster. Neutral and non euclidean geometries. http://www.math.uncc.edu/ droyster/math3181/notes/hyprgeom, 1996.

[8] Wikipedia. Hyperbolic geometry from wikipedia – the free encyclopedia. http://en.wikipedia.org/wiki/Hyperbolic_geometry.

[9] Douglas Dunham, John Lindgren, and David Witte. Creating repeating hyperbolic patterns. *ACM Computer Graphics*, 15(3), 1981.

[10] Douglas Dunham. Hyperbolic symmetry. *Computer and Maths with Appls*, 12B(1/2):139–153, 1986.

[11] Douglas Dunham. Hyperbolic archimedian tilings and color symmetry. private notes.

[12] Ajit Datar. Hyperart sourceforge.net project page. http://hyperart.sf.net.

[13] Trolltech. Qt from trolltech. http://www.trolltech.com/products/qt/.

[14] W3C. Extensible Markup Language. http://www.w3.org/XML/.

APPENDIX

APPENDIX A

Hyper*Art* DESIGN FILES

Here is a sample Hyper*Art* design file. Please see the in-line comments for explanation of tags and their attributes.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- A proposed and currently used xml format
     for HyperArt design    data. This file
     represents the circle limit II pattern
     originally found in cl2.dat.
     The comments near the tags explain
     the format. -->

<!-- This is the root node. It must have a
     type attribute which can presently be
     REGULAR_PGON or IRREGULAR_PGON.
     Some of the fields become mandatory
     or optional depending on this attribute.
     As such REGULAR_PGON type has more
     mandatory fields than IRREGULAR_PGON
-->
<design type="REGULAR_PGON">
    <!--
     This is the header for the design.
     Color indexes used in this design are defined here. -->
```

```
<metadata>
    <!-- defines color map, cid : hex
    The 'cid's are then used where ever color info is
    required.
    -->
    <colors count="3">
        <color> <cid>0</cid> <hex>000000</hex> </color>
        <color> <cid>1</cid> <hex>ff0000</hex></color>
        <color> <cid>2</cid> <hex>0000ff</hex> </color>
    </colors>
</metadata>

<p>8</p>

<!-- For IRREGULAR_PGON type there 0 to p-1
different q values. These are inserted between
a <qlist> tag one by one.
eg <qlist>
    <q>3</q>
    ... 7 more <q> tags
    </qlist>
For REGULAR_PGON type there is just one <q> tag
after <p> tag.
-->
<q>3</q>

<!--
This tag is for REGULAR_PGON and derived types only.
IRREGULAR_PGON type design should not have this tag.
```

```
number of "different" sides of the central p-sided
polygon that are used to form the fundamental
region that contains the motif.
( The other sides of the fundamental region are
two radii from the center to two vertices of that
p-sided polygon separated by 2*(2*pi/p) ).
This number must divide p, and p divided by this
number is the number of copies of the motif that
appears in the central p-sided polygon.
-->
<fund_reg_edges>2</fund_reg_edges>

<!--
This tag is for REGULAR_PGON and derived types only.
IRREGULAR_PGON type design should not have this tag.

The kind of reflection symmetry the pattern has within
the central p-sided polygon.
Possible values are,
REFL_NONE
    No reflection symmetry, only rotation symmetry
REFL_EDGE_BISECTOR
    Reflection symmetry across the perpendicular
    bisector of one of the edges of the p-sided polygon
REFL_PGON_RADIUS
    reflection symmetry across a radius (from the
    center to a vertex of the p-sided polygon)
-->
```

```
<refl_sym_type>REFL_EDGE_BISECTOR</refl_sym_type>

<!--
This tag is for REGULAR_PGON and derived types only.
IRREGULAR_PGON type design should not have this tag.

Color permutation induced by rotating by
fund_reg_edges*2*pi/p
Note that this is the "array" representation of
permutations (not the "mathematical" one using cycles).
There should be num_colors number of <perm> tags
If the sequence of color numbers is in order then
it is an identity permutation.
-->
<color_perm_rotn>
    <perm>0</perm>
    <perm>1</perm>
    <perm>2</perm>
</color_perm_rotn>

<!--
This tag is for REGULAR_PGON and derived types only.
IRREGULAR_PGON type design should not have this tag.

Color permutation induced by the reflection
It can be non-identity permutation if refl_sym_type is
REFL_EDGE_BISECTOR or REFL_PGON_EDGE
If refl_sym_type is REFL_NONE then color_perm_refl
has to be an identity permutation
```

```
-->
<color_perm_refl>
    <perm>0</perm>
    <perm>1</perm>
    <perm>2</perm>
</color_perm_refl>

<!--
Edge adjacency between layers
there should be p <entry> tags each having adjacent edge
number and a color permutation.
Each adjacency entry has following format
<entry e="integer"> pgon-edge-index, betn 0 to p-1
    Orientation can be ROTATION or REFLECTION
    <orientation> [ROTATION | REFLECTION] </orientation>
    <edge>
        [ adj layer pgon edge index, betn 0 to p-1]
    </edge>
    Color permutation associated with this adjacency
    <color_perm>
        There should be num_colors <perm> entries here
        <perm>cid</perm>
        <perm>cid</perm>
        <perm>cid</perm>
    </color_perm>
</entry>
-->
<adjacency>
    <entry  e="0">
```

```
        <orientation>ROTATION</orientation>
        <edge>1</edge>
        <color_perm>
            <perm>1</perm>
            <perm>2</perm>
            <perm>0</perm>
        </color_perm>
    </entry>
    <entry e="1">
        <orientation>ROTATION</orientation>
        <edge>0</edge>
        <color_perm>
            <perm>2</perm>
            <perm>0</perm>
            <perm>1</perm>
        </color_perm>
    </entry>
    <entry e="2">
        <orientation>ROTATION</orientation>
        <edge>1</edge>
        <color_perm>
            <perm>1</perm>
            <perm>2</perm>
            <perm>0</perm>
        </color_perm>
    </entry>
    <entry e="3">
        <orientation>ROTATION</orientation>
        <edge>0</edge>
```

```
        <color_perm>
            <perm>2</perm>
            <perm>0</perm>
            <perm>1</perm>
        </color_perm>
    </entry>
    <entry e="4">
        <orientation>ROTATION</orientation>
        <edge>1</edge>
        <color_perm>
            <perm>1</perm>
            <perm>2</perm>
            <perm>0</perm>
        </color_perm>
    </entry>
    <entry e="5">
        <orientation>ROTATION</orientation>
        <edge>0</edge>
        <color_perm>
            <perm>2</perm>
            <perm>0</perm>
            <perm>1</perm>
        </color_perm>
    </entry>
    <entry e="6">
        <orientation>ROTATION</orientation>
        <edge>1</edge>
        <color_perm>
            <perm>1</perm>
```

```
                <perm>2</perm>
                <perm>0</perm>
            </color_perm>
        </entry>
        <entry e="7">
            <orientation>ROTATION</orientation>
            <edge>0</edge>
            <color_perm>
                <perm>2</perm>
                <perm>0</perm>
                <perm>1</perm>
            </color_perm>
        </entry>
    </adjacency>

    <!--
    This describes the elements that make up the fundamental
    pattern. Each element has
        * a group of x, y coordinates which are in poincare
        unit circle system
        * cid
        * fill attribute

    elem types are,
    EUCLID_POLYLINE :
        n points n > 1
        fill = false
    EUCLID_POLY :
        n points, n > 2, last point is joined to first point
```

```
    fill = true/false
CIRCLE :
    2 points, center and any point on the circumference
    fill = true/false
HYPER_POLYLINE
    n points n > 1
    fill = false
HYPER_POLY :
    n points, n > 2, last point is joined to the first
    fill = true/false

Other types of elements might be added later
eg ellipse, spiral, text

All dat file points can be represented by one of these
elements. But only some of these elements can be
represented by dat file that is
dat -> xml is lossless
xml -> dat might not be
-->
<elements>
    <elem type="EUCLID_POLY">
    <fill>true</fill>
    <cid>0</cid>
    <points>
    <pt> <x>0.000000e+00</x> <y>0.000000e+00</y> </pt>
    <pt> <x>1.368406e-01</x> <y>1.379250e-01</y> </pt>
    <pt> <x>2.575625e-01</x> <y>2.575625e-01</y> </pt>
    <pt> <x>2.575625e-01</x> <y>1.893719e-01</y> </pt>
```

```
<pt> <x>1.837032e-01</x> <y>1.160219e-01</y> </pt>
<pt> <x>1.215594e-01</x> <y>5.387817e-02</y> </pt>
<pt> <x>6.756561e-02</x> <y>0.000000e+00</y> </pt>
</points>
</elem>

<elem type="EUCLID_POLY">
<fill>true</fill>
<cid>1</cid>
<points>
<pt> <x>6.756561e-02</x> <y>0.000000e+00</y> </pt>
<pt> <x>1.215594e-01</x> <y>5.387817e-02</y> </pt>
<pt> <x>1.837032e-01</x> <y>1.160219e-01</y> </pt>
<pt> <x>2.575625e-01</x> <y>1.893719e-01</y> </pt>
<pt> <x>2.575625e-01</x> <y>2.575625e-01</y> </pt>
<pt> <x>3.747407e-01</x> <y>1.552227e-01</y> </pt>
<pt> <x>2.188499e-01</x> <y>0.000000e+00</y> </pt>
</points>
</elem>

<elem type="EUCLID_POLY">
<fill>true</fill>
<cid>2</cid>
<points>
<pt> <x>2.188499e-01</x> <y>0.000000e+00</y> </pt>
<pt> <x>3.747407e-01</x> <y>1.552227e-01</y> </pt>
<pt> <x>3.648853e-01</x> <y>0.000000e+00</y> </pt>
</points>
</elem>
```

```
    </elements>
</design>
```