FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

Irrational Image Generator

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Medieninformatik

eingereicht von

Simon Parzer
Matrikelnummer 0726680

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Eduard Gröller
Mitwirkung: Dipl.-Ing. Dr. Christoph Traxler
Kurt Hofstetter

Wien, June 19, 2013

(Unterschrift Verfasser)

(Unterschrift Betreuung)

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Irrational Image Generator

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Media Informatics

by

Simon Parzer
Registration Number 0726680

to the Faculty of Informatics
at the Vienna University of Technology

Advisor:	Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Eduard Gröller
Assistance:	Dipl.-Ing. Dr. Christoph Traxler Kurt Hofstetter

Vienna, June 19, 2013

(Signature of Author)

(Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Simon Parzer
Canalettogasse 12/3/8, 1120 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)

(Unterschrift Verfasser)

Abstract

An approach called Inductive Rotation (IR), developed by artist Hofstetter Kurt, can be used to create intricate patterns that fill the 2D plane from a single prototile by repeated translation and rotation. These patterns are seemingly nonperiodic and have interesting features, both from a mathematical and artistic viewpoint.

The IR method has not yet been described in scientific literature. It is related to and has been inspired by aperiodic tilings like the well-known Penrose tilings.

During the course of this thesis some research on the patterns generated by Inductive Rotation has been done and algorithms that allow for automatic generation of these patterns have been developed. The implementation is then called the Irrational Image Generator, a tool that on the one hand is a reference implementation of the IR method, and on the other hand can be used by the artist for further experimentation to fully utilize the artistic possibilities of the IR approach.

The Irrational Image Generator is preceded by a series of prototypes, that have been developed to get a better grasp of the expected results and performance of the tool. Each prototype as well as the final implementation were tested by Hofstetter Kurt. This iterative development process has led to two different implementation approaches that both have their advantages and disadvantages. For this reason, both methods have been considered in the final implementation.

Generation algorithms that operate on geometry instead of directly manipulating bitmap data have been developed. The program makes use of the GPU through OpenGL to render the resulting patterns through textured polygons.

It turns out that run-time and memory usage of the IR algorithm grow exponentially with the number of iterations. This means that iteration numbers are limited, although the tool's performance is sufficient for artistic purposes.

Kurzfassung

Mit Hilfe eines von Hofstetter Kurt entwickeltem Verfahren, genannt Inductive Rotation (IR), kann auf einer zweidimensionalen Ebene aus einem einzigen Startelement durch wiederholte Translation und Rotation ein komplexes, lückenloses Kachelmuster erzeugt werden. Diese Muster sind allem Anschein nach nicht periodisch und haben sowohl aus mathematischer als auch künstlerischer Sicht interessante Eigenschaften. Das IR-Verfahren wurde noch nicht in der wissenschaftlichen Literatur behandelt, aber ist am ehesten vergleichbar mit aperiodischen Kachelmustern wie sie z.B. bereits von Penrose beschrieben wurden.

Die Diplomarbeit beschäftigt sich näher mit den durch Inductive Rotation erzeugten Mustern. Es wurden Algorithmen entwickelt, mit denen es möglich ist, die Rotationsschritte in einem Computerprogramm zu automatisieren. Die Implementierung schließlich ist der Irrational Image Generator, ein Programm, das einerseits als Referenzimplementierung für das genannte Verfahren dient, andererseits aber vom Künstler verwendet wird, um durch weiteres Experimentieren das IR-Verfahren künstlerisch auszureizen.

Dem Irrational Image Generator gehen eine Reihe von Prototypen voraus, die dazu entwickelt wurden, um das Laufzeitverhalten und die Ergebnisse abschätzen zu können. Sowohl die Prototypen als auch das Endprodukt wurden von Hofstetter Kurt getestet. Dieser iterative Entwicklungsprozess führte dazu, dass das fertige Programm zwei unterschiedliche Implementierungen beinhaltet, die beide ihre Vor- und Nachteile haben.

Ich habe für die Generierungs-Algorithmen einen geometrischen Ansatz gewählt, anstatt nur mit Bitmaps zu arbeiten. Das Programm verwendet die GPU, um die resultierenden Muster aus texturierten Polygonen darzustellen.

Es hat sich herausgestellt, dass sowohl Laufzeit als auch Speicherbedarf des IR-Algorithmus abhängig von der Anzahl der Iterationen exponentiell ansteigen. Dadurch sind Iterationszahlen nach oben hin begrenzt, reichen aber für künstlerische Zwecke aus.

Contents

CHAPTER 1

Introduction

The research on aperiodic patterns is a relatively young field in the history of science. Robinson [37] and Penrose [32] for example have proofed in the 1970s the existence of aperiodic sets of tiles with as few as two prototiles. This thesis is motivated by Inductive Rotation (IR), a new approach of creating a special case of nonperiodic tessellations of the plane, created by a recursive rotation scheme that explicitly allows overlaps and thus creates a three-dimensional structure.

While this approach generates interesting results not only from an artistic, but also from a mathematical standpoint, it has not yet been described in scientific literature. Furthermore, while the mechanics of IR are relatively easy to understand, the results are hardly predictable, and there is a need for an automated tool that can be used for artistic experimentation.

The aim of this thesis is to give a description of the IR method in a scientific context and to create a tool that serves both as reference implementation, but can also be used for artistic experimentation.

Chapter 2 gives insight into the current state of the art for generating nonperiodic patterns, tilings and images. Hofstetter Kurt's IR method cannot be directly compared to already existing approaches, but has been greatly inspired by the work of Penrose on nonperiodic tilings of the plane, as well as other approaches like substitution tilings and fractals. Chapter 3 then gives a formal description of the IR method, along with approaches on how it can be implemented as a computer algorithm. The chapter also states the requirements for the resulting implementation and gives an overview of the artistic opportunities. Chapter 4 describes the programming environment, development process, and technical details of the resulting implementation, while Chapter 5 serves as a short summary of the entire thesis.

CHAPTER 2

State of the art

The following chapter describes the current state of the art for generating nonperiodic patterns, tilings, and images. Hofstetter Kurt's Inductive Rotation method, which is used for this thesis, is a new approach that has not been described in scientific literature yet. A description of this method will be given in Chapter 3, while this chapter will focus on existing methods to provide an overview of related work in the field of aperiodic tilings and patterns.

Nonperiodic tilings and patterns have a long history. The first documented findings of sophisticated aperiodic tilings made by men reach back thousands of years [23]. However, it was not until the second half of the 20th century that nonperiodic tilings got picked up by the scientific community. One reason for that is that they introduce a series of interesting mathematical problems. Another reason is that they can be used to describe quasicrystalline structures, a phenomenon that was first described in detail by Shechtman et al. [39] in 1984.

At this point, a very interesting PhD thesis by Kaplan [19] should be also mentioned, which treats a very similar subject. Kaplan focuses on existing ornamental design. He has developed the mathematical understanding, algorithms and tools necessary to reproduce Islamic Star Patterns as well as the famous tesselations of M. C. Escher. While the topic might by slightly different, the approach on geometric patterns and the intersection with computer graphics makes this thesis noteworthy in this context.

2.1 Tilings of the Plane

Since many of the following sections will be about different approaches on nonperiodic tilings of the plane, this section gives a quick overview on tilings of the plane in general, and the related technical terms.

According to Grünbaum and Shephard [13], a standard reference on tilings and patterns, a tiling of the plane "is a countable family of closed sets, which cover the plane

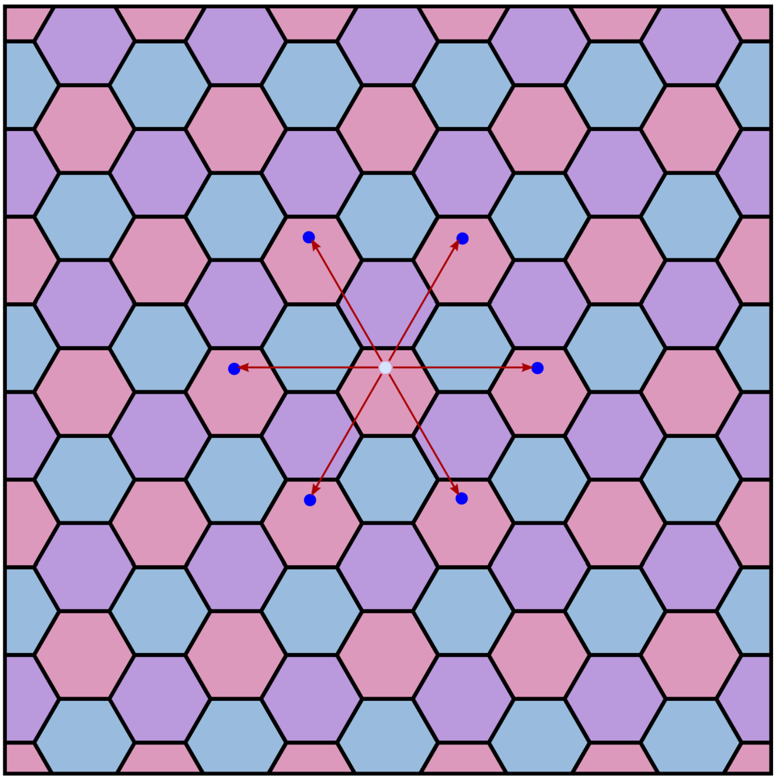


Figure 2.1: This hexagon tiling is an example for a periodic tiling. Every one of the depicted translation vectors maps the tiling onto itself.

without gaps or overlaps". "Plane" means the Euclidean plane known from elementary geometry, which is an infinite two-dimensional space. A tiling tesselates the infinite plane while only using a finite set of closed shapes, called **prototiles**. For a valid tiling, the entire plane must be covered, and all of the tiles must have disjoint interiors, meaning there are no gaps and no overlaps.

This thesis is mostly about nonperiodic tilings of the plane. It is important to differentiate the terms periodic, nonperiodic and aperiodic. A **periodic** tiling can be mapped onto itself by translation, meaning the placement of prototiles is repeating itself infinitely often.

This means, for any infinite periodic tiling, there is at least one translation vector (x, y) with $x \neq 0$ and/or $y \neq 0$ that when applied yields the same tiling again. This is called **translational symmetry**; see Figure 2.1 for an example. Other kinds of symmetry, like **rotational symmetry** or **reflective symmetry** on the other hand give no indication whether a tiling is periodic or not.

A **nonperiodic** tiling, by contrast cannot be mapped onto itself by translation. If there is a set of prototiles that only allows for nonperiodic tilings, i.e., it is impossible

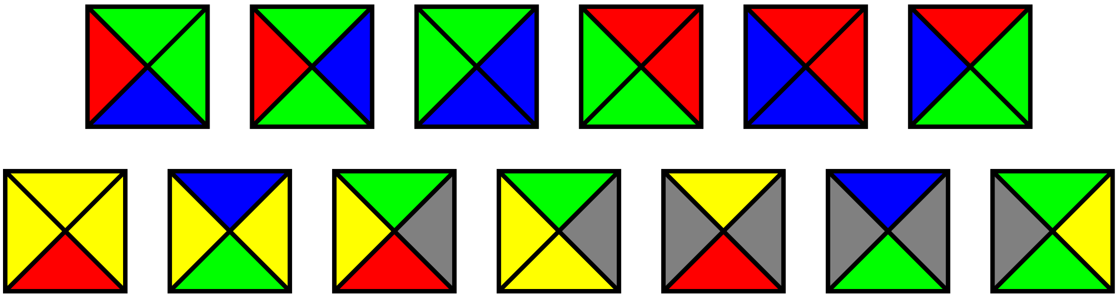


Figure 2.2: A set of 13 color-coded Wang tiles that allows only aperiodic tilings, according to Culik [5].

to construct a periodic tiling from this set of tiles, this is called an **aperiodic** tile set.

Some aperiodic tile sets include matching rules which restrict the placing of tiles, e.g., by coloring the edges and specifying that only edges with matching colors can touch. Thus, it is possible to define aperiodic tile sets where the geometric shape of the tiles alone would allow periodic tilings. Figure 2.2 is an example of such an aperiodic tile set.

2.2 Wang Tiles

Wang tiles, named after mathematician, logician and philosopher Hao Wang, are square tiles with color-coded edges. A Wang tiling allows only a specific set of Wang tiles with exactly the same size. For each tiling, only edges with the same color can touch.

2.2.1 The Domino Problem

In a paper published in 1961, Wang [44] introduces his tiles for the first time for describing the “Domino Problem”, a special case of Hilbert’s decision problem (see Gurevich [15]). A finite set of tiles is called solvable if the infinite plane can be tiled by this set by the rules of only using tiles of the same size and always matching edges of the same color. Transformations other than translation, e.g., rotation or reflection of the tiles, are not allowed.

Wang’s conjecture states that a set of Wang tiles is solvable if and only if it also has a periodic solution. This would also mean that there is no aperiodic set, which by definition allows only nonperiodic solutions. Wang argued that if his conjecture turned out to be true, the Domino Problem would be decidable.

The conjecture, however, was proven wrong in 1966 by Berger [2] who constructed a set of over 20,000 Wang tiles that allows only aperiodic solutions. This number has been decreased significantly in subsequent publications. In a 1996 paper, Culik [5] shows an aperiodic set of only 13 different Wang tiles (see Figures 2.2 and 2.3).

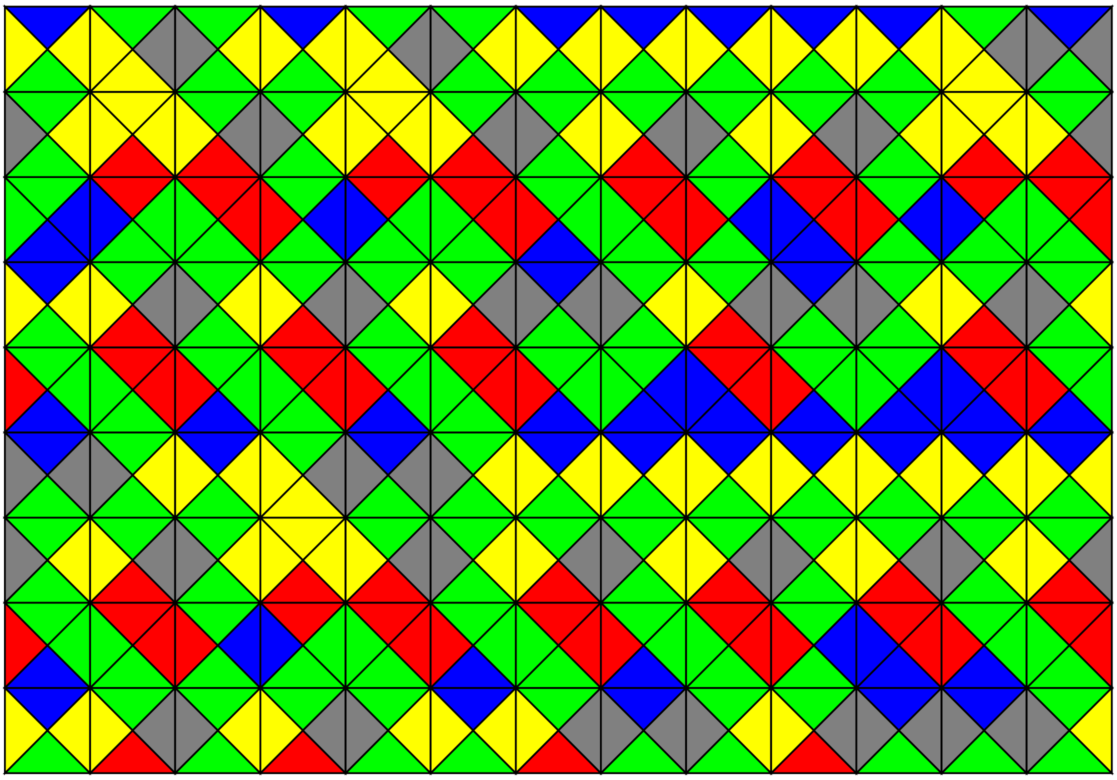


Figure 2.3: Part of a possible tiling using the tiles from Figure 2.2.

2.2.2 Construction of Wang Tilings

Generating a valid tiling from a set of Wang tiles can be done using a recursive backtracking algorithm that iterates through all possible combinations for a fixed-size grid of Wang tiles until the tiling is valid. For practical applications, an approach like this is not very advisable since tilings built from a large number of Wang tiles, like those needed in computer graphics applications for example, would need a very long time to compute.

Grünbaum and Shephard [13] describe a different algorithm. For a set of 16 specific Wang tiles, there exist so-called composition/decomposition rules where each of the 16 tiles can be replaced by a group of either 1, 2 or 4 tiles. This way, one can start with a small tiling, or even just one tile and "inflate" the tiling using those rules, effectively creating a bigger tiling, which is still valid. This method is more efficient, especially for generating large tilings. It has already been used by Stam [40] for generating large texture patterns from Wang tiles (see also Section 2.2.3). A similar approach exists for Penrose tilings, see Section 2.3.2 for more details.

2.2.3 Applications of Wang tiles

While the Domino Problem has been proven undecidable, there are still interesting applications for Wang tiles. They have been used for procedural generation of two-dimensional data, with some interesting applications in computer graphics. Wang tilings have aesthetic qualities in that they do not produce visual artifacts caused by repetition, like it would be the case with periodic tilings.

In a 1997 paper, Stam [40] uses an aperiodic set of 16 textured Wang tiles to generate 2D textures. Making sure that matching edges do not have discontinuities for the individual textures, this produces large homogenous textures with no visible repetitions while in reality only using 16 small images (see Figure 2.4). This has advantages in computer graphics and real-time rendering where storing large textures is expensive compared to a tile-based approach.

An even more sophisticated approach that expands on the idea of using Wang tiles for texture mapping has been described by Cohen et al. [3]. Here, the set of Wang tiles chosen is not aperiodic, i.e., it would be theoretically possible to construct periodic tilings. To ensure non-periodicity, the algorithm for laying out the tiles uses stochastic processes. The paper then goes on into describing methods for designing or automatically generating textures for the tiles. For most examples, a set of 8 Wang tiles with two different colors is used. A different application for these tiles that is shown in the paper is to place 3D models, preventing periodic placements. One of the examples shows a sunflower field with a large number of sunflowers that are placed using this approach, without any visible repeating patterns, see Figure 2.5.

Wei [45] has successfully implemented an algorithm for texture mapping that runs directly on the GPU in a fragment program. It supports very large virtual textures that are composed of a set of 8 Wang tiles, based on the ideas of Cohen et al. [3].

Kopf et al. [22] show a method for generating blue noise point sets with non-uniform density, using matching sets of Wang tiles with varying point densities. Point patterns without visible repetitions are needed in many computer graphics applications, such as anti-aliasing, ray-tracing and non-photorealistic rendering. The paper also shows an example application; a tool for interactive texture painting in real-time.

2.2.4 Robinson Tiles

Further research on Wang tiles and related mathematical problems led to the discovery of other aperiodic tile sets. A set of tiles that is closely related to Wang tiles and shall therefore also be mentioned in this section was introduced by Robinson [37] in 1971.

Instead of using matching rules on plain squares, like color coding or numbering, he modifies the edges of the squares like a jigsaw puzzle. There are two interesting aspects here. First, through modifying the edges, the matching rules are encoded in the geometry of the tiles themselves, avoiding the need for additional rules. Second, it is now possible to further reduce the number of tiles in an aperiodic set by allowing rotation and reflection in addition to translation. Robinson used this to construct an aperiodic set of 6 prototiles (see Figure 2.6).

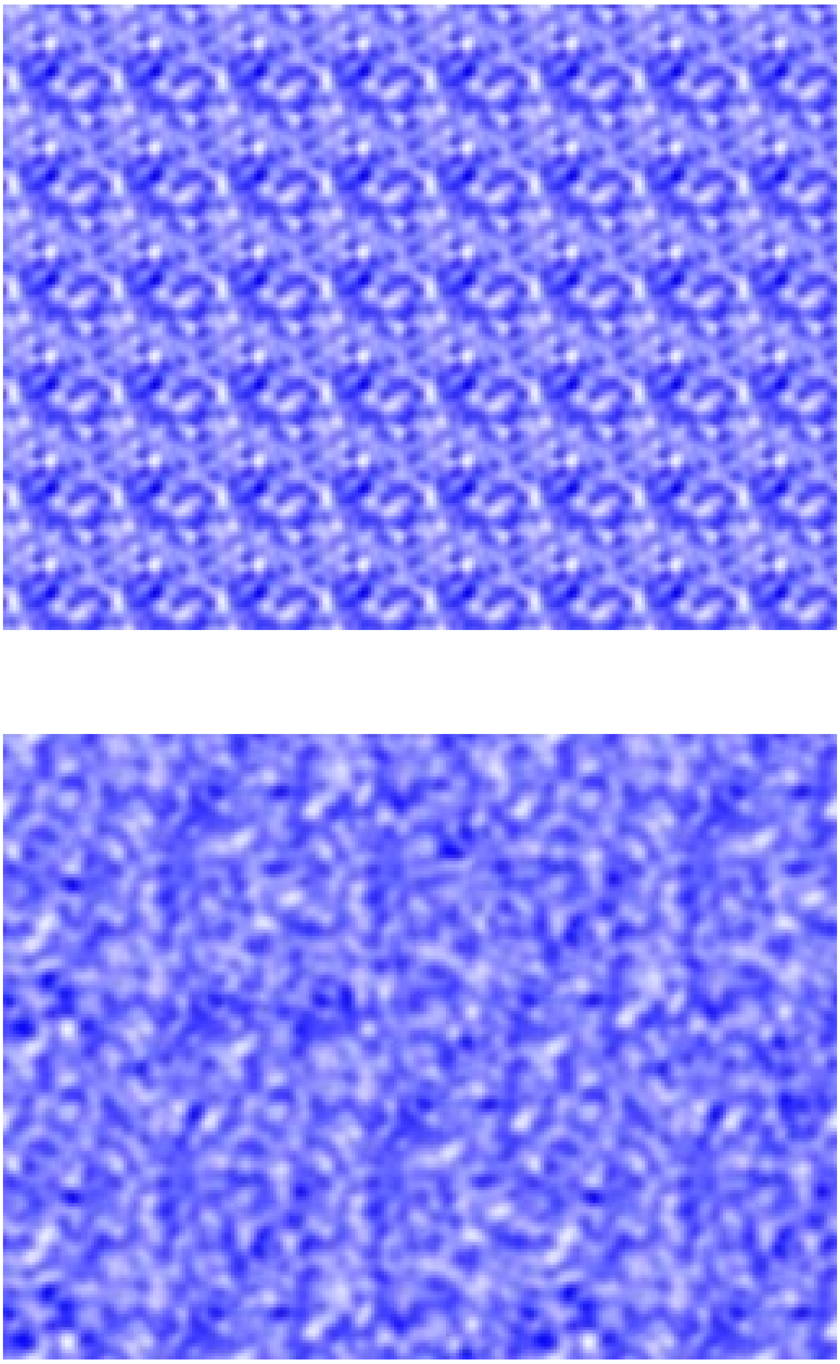


Figure 2.4: A comparison of using just one square tile (top) and using a set of 16 different textured Wang tiles (bottom) [40].

Figure 2.5: Nonperiodic sunflower field generated from Wang tiles [3].

Apart from the bumps, dents and notches used to enforce matching rules, those tiles are still square-shaped and very closely related to Wang tiles. Robinson's work is mainly interesting from a mathematical point of view. It seems like a more beautiful solution than Wang's approach, as only six tiles are needed for an aperiodic set and there is no need for color-coded edges.

2.3 Penrose Tilings

One of the most popular aperiodic tilings of the plane in scientific literature are the *Penrose tilings*. They were discovered by Roger Penrose [32] in the 1970s.

While the Domino Problem and Wang tiles were already known at this point in time, Penrose did not use any of the existing approaches. Instead, according to Grünbaum and Shephard [13], his discovery is inspired from a construction by Kepler [21], where a regular pentagon is subdivided into six smaller ones, leaving only five slim triangular

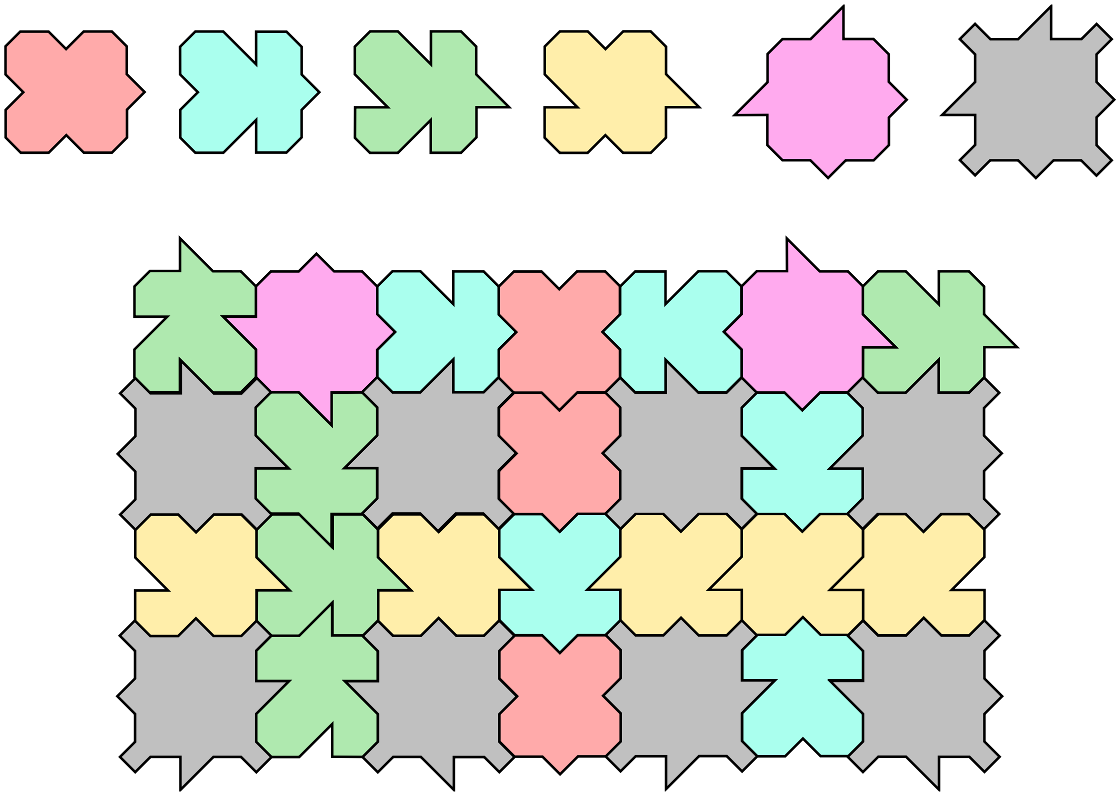


Figure 2.6: Robinson's aperiodic set of 6 tiles (top), and an example tiling (bottom) [37].

gaps. Each of this six pentagons can be subdivided in the same way. Kepler already noted that the gaps in an infinite subdivision can be filled using a finite number of geometric shapes. Penrose expanded on this, defining matching rules for filling the gaps. He ended up with an aperiodic set of six prototiles, having four different shapes. See Figure 2.7 for the prototiles, and Figure 2.8 for an example tiling.

While this result seems similar to Robinson's aperiodic set of six prototiles (see Section 2.2.4), Penrose was able to simplify his tiles further, ending up with two different aperiodic sets with only two prototiles each (see Figure 2.9).

The first of these sets is called **Kite and dart tiling**, using two quadrilaterals called "kite" and "dart", which could be combined to make a rhombus. However, the matching rules do not allow this particular combination (see Figure 2.10).

Another Penrose set with only two prototiles is called **Rhombus tiling** and is probably the most widely known one. It uses a pair of rhombi with equal sides but different angles. Without matching rules these rhombi could easily tile the plane periodically, so matching rules are needed as well (see Figure 2.11).

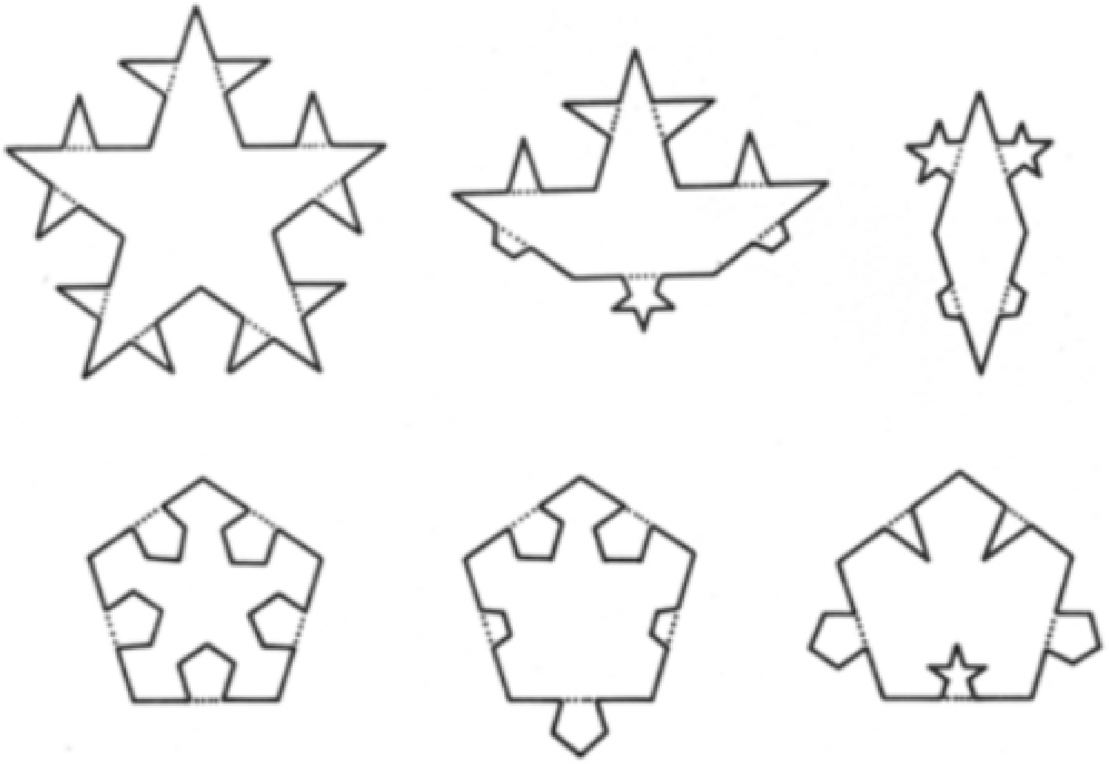


Figure 2.7: Penrose's original pentagonal tiles with edge modifications to enforce matching rules [32].

2.3.1 Properties

Apart from the fact that they are nonperiodic and can be constructed from different aperiodic sets of tiles, Penrose tilings have several remarkable properties [13, 38]. Because the construction of the tiles comes from the tessellation of a regular pentagon, all Penrose tilings have local pentagonal symmetry. This means that in an infinite tiling there are always points that are the center of a local region with five-fold symmetry. There can also be one global center of five-fold symmetry. However, this is not the case with all of the Penrose tilings.

The golden ratio $\varphi = (1 + \sqrt{5})/2$, which is also the ratio between a side and a chord of the regular pentagon, plays an important role in Penrose tilings. The ratio $\varphi : 1$ is also the ratio of the long to the short side of both kite and dart tiles, as well as the ratio of the short diagonal to the sides in the thin rhombus and the ratio of the long diagonal to the sides in the thick rhombus. The area ratios of kite to dart and thick to thin rhombus have the same property, too.

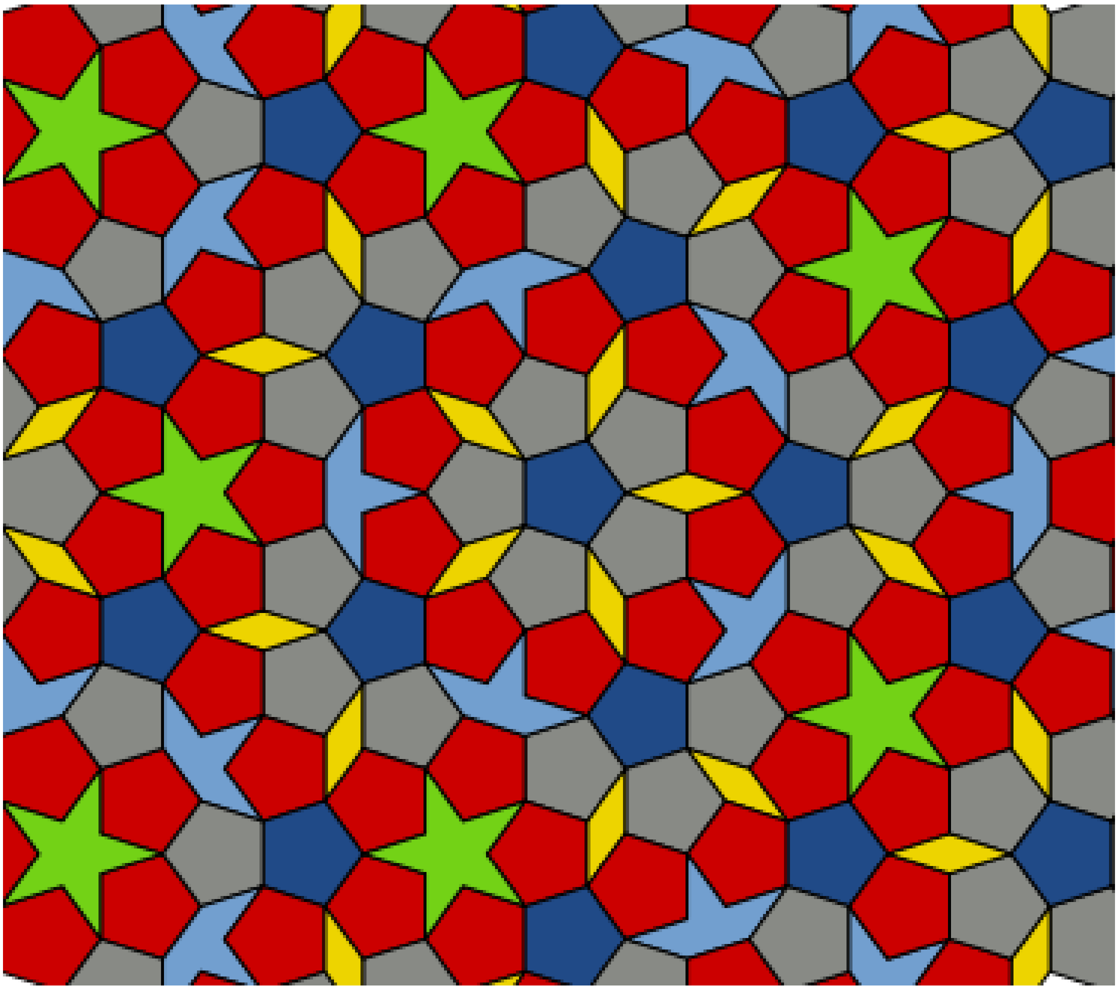


Figure 2.8: Original Penrose tiling with six prototiles.

2.3.2 Construction methods

Looking at the sets with two prototiles, a simple method would be starting with one tile and adding tiles along its edges in a trial-and-error approach. While this may result in an interesting challenge when done by a person, there are better methods that do not require backtracking and allow for the automatic generation of very large patterns.

2.3.2.1 Backtracking

With Penrose tiles it is very easy to end up with a pattern where it is impossible to add new tiles without leaving holes or breaking any of the matching rules. A computer algorithm using only trial-and-error to lay out a pattern would require expensive backtracking, especially as the generated pattern grows. This is not really feasible if

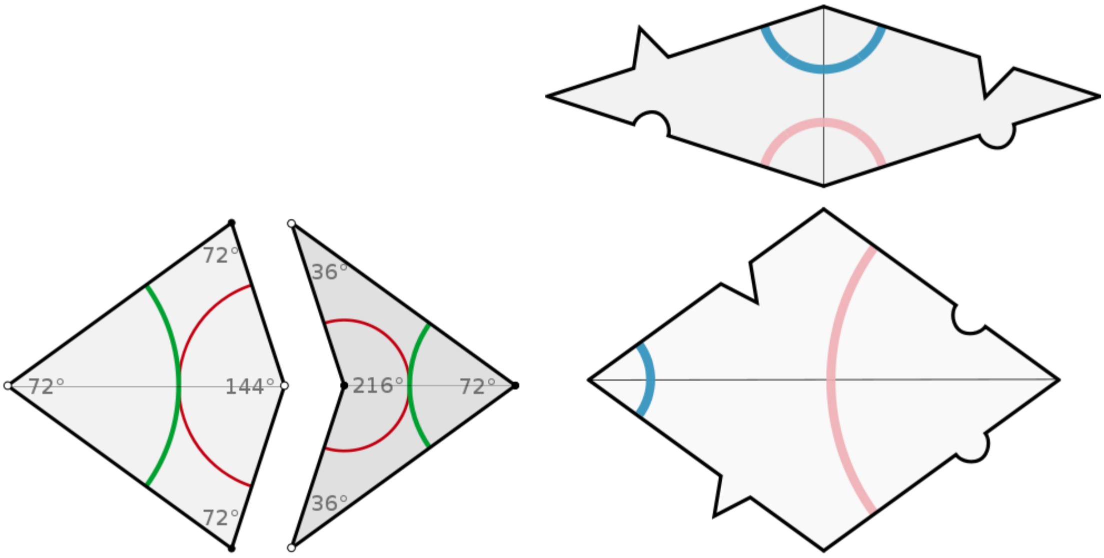


Figure 2.9: The two aperiodic Penrose tile sets. Left: Kite and dart tiling; Right: Rhombus tiling. The colored curves denote the matching rules; they have to be continuous when putting the tiles together.

one wants to generate large Penrose patterns, since recursive backtracking algorithms usually show exponential growth with respect to run-time and memory consumption.

2.3.2.2 Inflation/Deflation

A more efficient way is the introduction of inflation and deflation rules. The inflation rule replaces each tile of an existing pattern by a bigger composition of tiles and/or half-tiles. Applying the inflation rule to an existing, valid Penrose pattern yields another valid pattern that is larger than the first one. The deflation rule is basically the inverse of the inflation rule. This approach is also known as recursive subdivision.

It is easily possible to define such rules for the first construction of the Penrose tiling with six prototiles, which is essentially a hierarchical tesselation of the regular pentagon. For the kite/dart and rhombus tilings, inflation and deflation rules exist as well. Those have been described in detail by Grünbaum and Shephard [13]. David Austin gives a vivid description of this method in his column “Penrose Tiles Talk Across Miles” [31] along with a Java applet where the user can walk through an inflation hierarchy interactively.

Using inflation/deflation rules, a fast algorithm for constructing large Penrose patterns is possible. For example, in Ostromoukhov et al. [30], recursive subdivision according to Grünbaum and Shepherd’s rules (see Figure 2.12) is used for the generation of large Penrose patterns. Since recursive subdivision is possible, the Penrose tiling classifies as a **substitution tiling** (see Section 2.4).

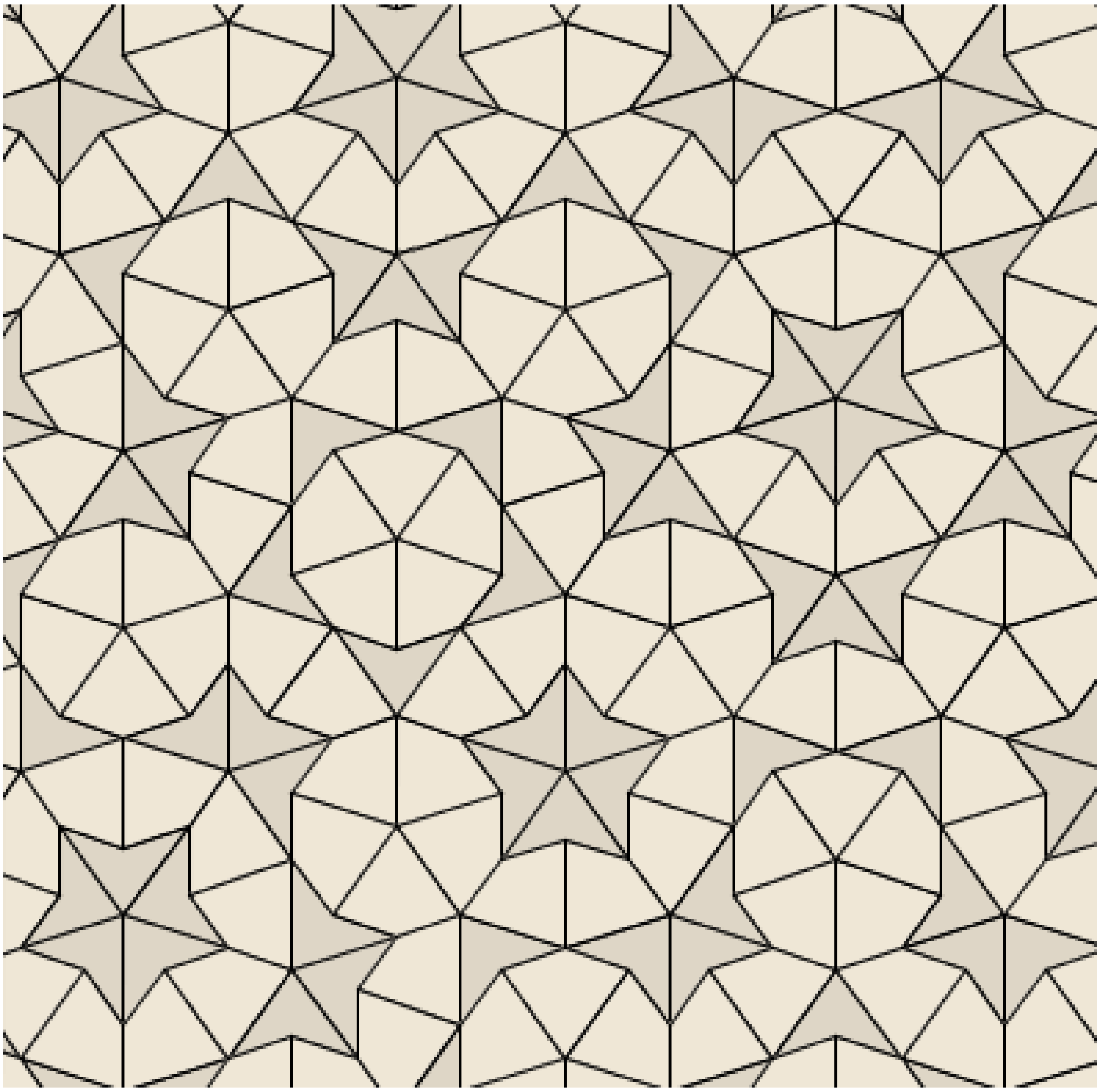


Figure 2.10: Penrose kite and dart tiling. Image generated using the Penrose Tiling Applet [20].

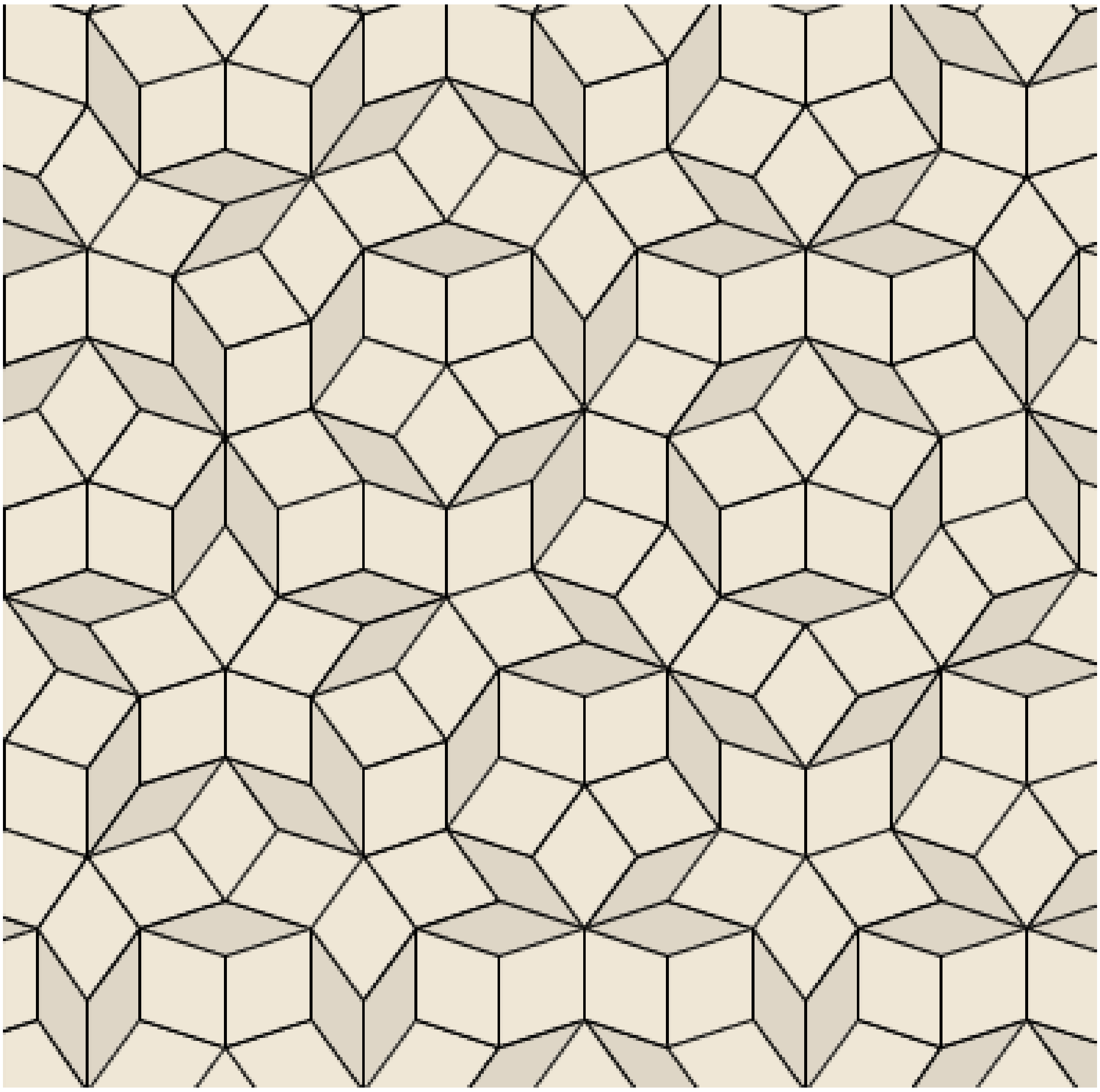


Figure 2.11: Penrose rhombus tiling. Image generated using the Penrose Tiling Applet [20].

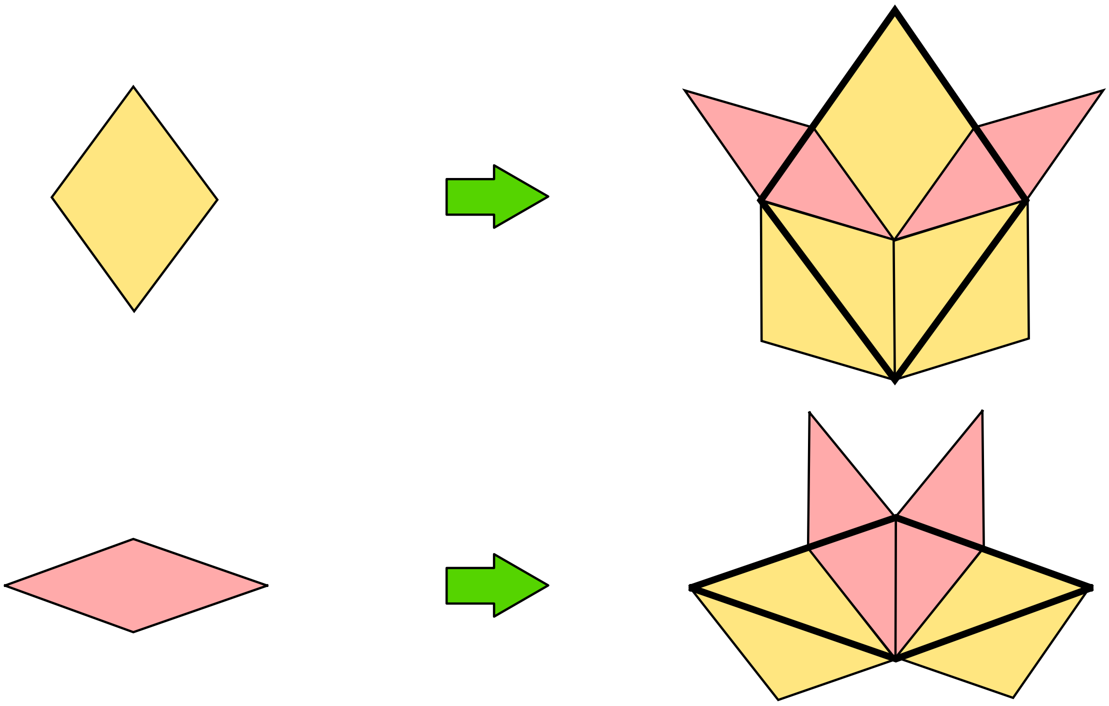


Figure 2.12: Recursive subdivision rules for Penrose thick and thin rhombi. This subdivision results in a valid Penrose tiling [13].

2.3.2.3 Cut and Project

A third method was introduced by de Bruijn [6] in 1981. He shows that it is possible to construct Penrose patterns as a 2D projection from a five-dimensional cubic structure. This has been dubbed the **cut and project method**. For this construction, the tiling is not treated as a collection of shapes or prototiles, but as a collection of connected vertices. A more detailed explanation of this approach would go beyond the scope of this thesis. The method is only mentioned here for the sake of completeness.

2.3.3 Applications

One remarkable property of the Penrose tilings is their aesthetic quality. From floor tilings to ornaments to artworks, the possibilities for using the tilings as a source for nice-looking, non-repeating patterns are endless. Also, the physical process of laying out the tiles proves to be an interesting and challenging game. For this and other reasons, Roger Penrose decided to patent his tiling approach in 1979 [33].

Especially with hierarchical substitution methods, which have been researched deeply in the scientific literature, it is possible to make the generation process really fast. An example is the generation of blue noise sample points for computer graphics from the vertices of a Penrose tiling by Ostromoukhov et al. [30].

2.3.4 Penrose tilings and quasicrystals

In 1984, Shechtman et al. [39] discovered crystalline structures with five-fold symmetry in an aluminium alloy. The basic laws of crystallography disallow the existence of five-fold symmetry in crystals, so this marked the discovery for a new kind of structure, today called quasicrystal. Subsequently, the Nobel Prize in Chemistry 2011 was awarded to Dan Shechtman "for the discovery of quasicrystals".

At some point it was discovered that the structure of this quasicrystals match the structure of the Penrose tiling. This was one of the starting points for extensive mathematical research, trying to explain how quasicrystals can be formed. Still, it was very hard to explain how crystals can form Penrose tilings, which require at least two different building blocks and complex matching rules. A breakthrough was made in 1996 by Gummelt [14], describing a method for constructing Penrose tilings from the overlaps of a single decagonal prototile. This was refined in 1997 by Jeong and Steinhardt [18] and is a possible explanation of how quasicrystals may form physically.

2.4 Substitution Tilings

Tile substitution is the process of repeatedly subdividing shapes according to certain rules. These rules are sometimes also referred to as inflation and deflation rules. With an infinite number of substitution steps, one can create a tiling that covers the entire plane. This is particularly interesting, because tile substitution can also result in nonperiodic tilings. Both Penrose and Wang tilings are examples for this (see Sections 2.2 and 2.3, respectively).

Figure 2.13 depicts a very simple substitution process eventually resulting in an infinite tiling of the Euclidean plane. It should be clear that similar substitutions are also possible in other spaces, including the one-dimensional space and higher-dimensional spaces.

One notable example of a substitution tiling is the so-called **Pinwheel tiling of the plane**. It is based on a construction by John Conway (unpublished) and described in detail in a thesis by Radin [36]. Figure 2.14 shows how a substitution system can produce the Pinwheel tiling. One notable property of this tiling is that it consists of only copies of the same triangle, but in infinitely many rotations. Thus, it is a nonperiodic tiling.

There is also a known substitution system for a certain set of Wang tiles, which has been described in detail along with the Penrose substitution by Grünbaum and Shephard [13]. This is especially convenient if one wants to generate one of those tilings algorithmically, as substitution rules are easy to implement and fast to compute.

Over the time, countless nonperiodic tilings generated by substitution patterns have been found. Dirk Frettlöh, who also wrote a paper on the matter of substitution

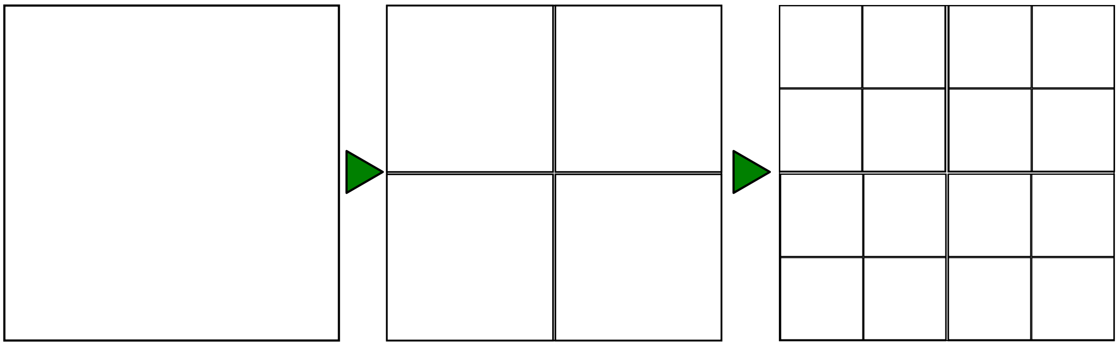


Figure 2.13: A very simple substitution that generates an infinite quad tiling. The second substitution is the first substitution recursively applied to all four sub-quads. The substitution can be repeated ad infinitum to generate the tiling.

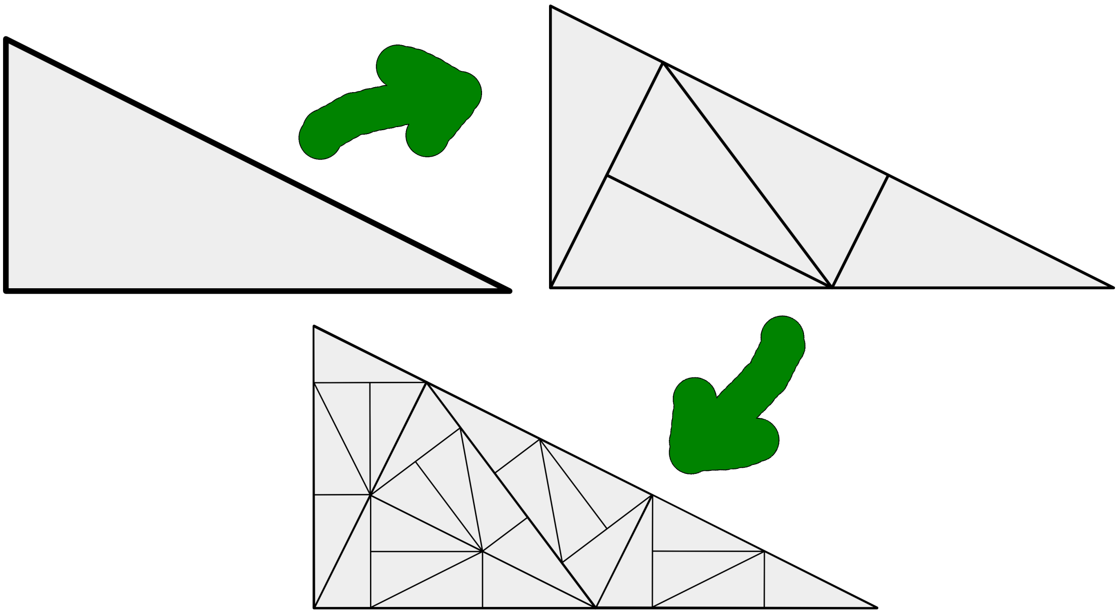


Figure 2.14: The first two substitution steps for generating the Pinwheel tiling, which is nonperiodic [36].

tilings [10], has created an online encyclopedia that lists many of the substitution patterns known from literature and otherwise [11].

2.5 Fractals

Fractals have also been used to a great extent for generating beautiful self-similar patterns. This chapter will give a short overview. A more thorough view into the subject can be acquired from Mandelbrot [25], or Crownover [4].

The term fractal was first used by mathematician Benoît Mandelbrot in 1975 to describe geometric patterns with certain properties. There is no clear formal definition of geometric figures that count as fractals, but they usually have the following properties.

detailed self-similarity Fractals are self-similar, that means either the same or a very similar pattern is repeated in infinitely many scaling levels. The term detailed is used to exclude trivial self-similarity. Fractal patterns show detailed structures at arbitrary small scales. Self-similar parts are scaled down by a characteristic contraction factor. For that reason, fractal curves are not differentiable.

recursive definition All fractals are defined by recursion, i.e., by a dynamic system. Fractal structures can be generated by infinitely many iterations of linear or nonlinear functions or simple algorithms.

Popular examples of fractals include the Koch Snowflake and the Julia and Mandelbrot sets. In the following the concept of fractals will be explained on the basis of the Koch Snowflake.

2.5.1 Koch Snowflake

The Koch Snowflake is one of the earliest described fractals and is based on the Koch curve. The construction of the fractal starts with an equilateral triangle. The following three steps are then executed recursively on each line segment of the shape.

1. Divide the line in three segments of the same length
2. Change the middle segment to an equilateral triangle that points outwards
3. Remove the line segment that was used to construct the new triangle

If this recursion is repeated infinitely many often, the resulting curve consists of an infinite number of line segments. It is continuous everywhere, but nowhere differentiable. The first few steps of the recursion are depicted in Figure 2.15.

The Koch Snowflake, albeit being a very simple construction, already shows the idea behind fractal objects. The construction is based on a simple recursion and the resulting object features strict self-similarity.

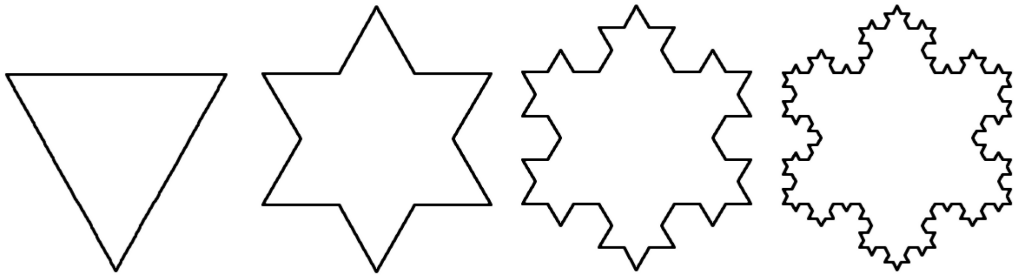


Figure 2.15: From left to right: The equilateral triangle and the first three iterations of the Koch Snowflake fractal.

2.5.2 Fractal Art

With the help of a computer program, fractal objects can be represented as 2D images. By using different fractal objects and adjusting parameters of the 2D representation such as zoom level or color mapping, one can generate an arbitrary number of different images from fractals. An article in the "Scientific American" in 1985 [7] featuring colorful renderings of the Mandelbrot set could be considered as one of the first public displays of fractal art. Mandelbrot gives an introduction to fractal art as well in a 1989 article [24].

Kerry Mitchell tries to give a more detailled definition of fractal art and distinguishes it from other forms of two-dimensional visual art in an online article [29]:

> "Fractal art (FA) is a subclass of two dimensional visual art, and is in many respects similar to photography—another art form which was greeted by skepticism upon its arrival. Fractal images typically are manifested as prints, bringing Fractal Artists into the company of painters, photographers, and printmakers. Fractals exist natively as electronic images. This is a format that traditional visual artists are quickly embracing, bringing them into FA's digital realm."

Considering the mathematical complexity of the calculations, it would be impossible to generate most if not all fractal images without the use of a computer. Therefore, fractal art is a relatively young form of art. Fractal images have their own distinctive aesthetics. Usually the strong self-similarity can be easily recognized. One fascinating feature of fractal images is that, even though they can be generated using a simple formula, they show fine details at arbitrary high zoom levels. Due to the self-similarity one usually cannot determine the zoom level by looking at a fractal image.

Some of the most popular examples of fractal art include images generated from the Mandelbrot and Julia sets. Figure 2.16 shows a graphical interpretation of the Mandelbrot set. Each pixel of the raster image corresponds to a complex number c and is colored according to whether c is part of the Mandelbrot set or not. A more detailed explanation is given in Section 2.5.3.

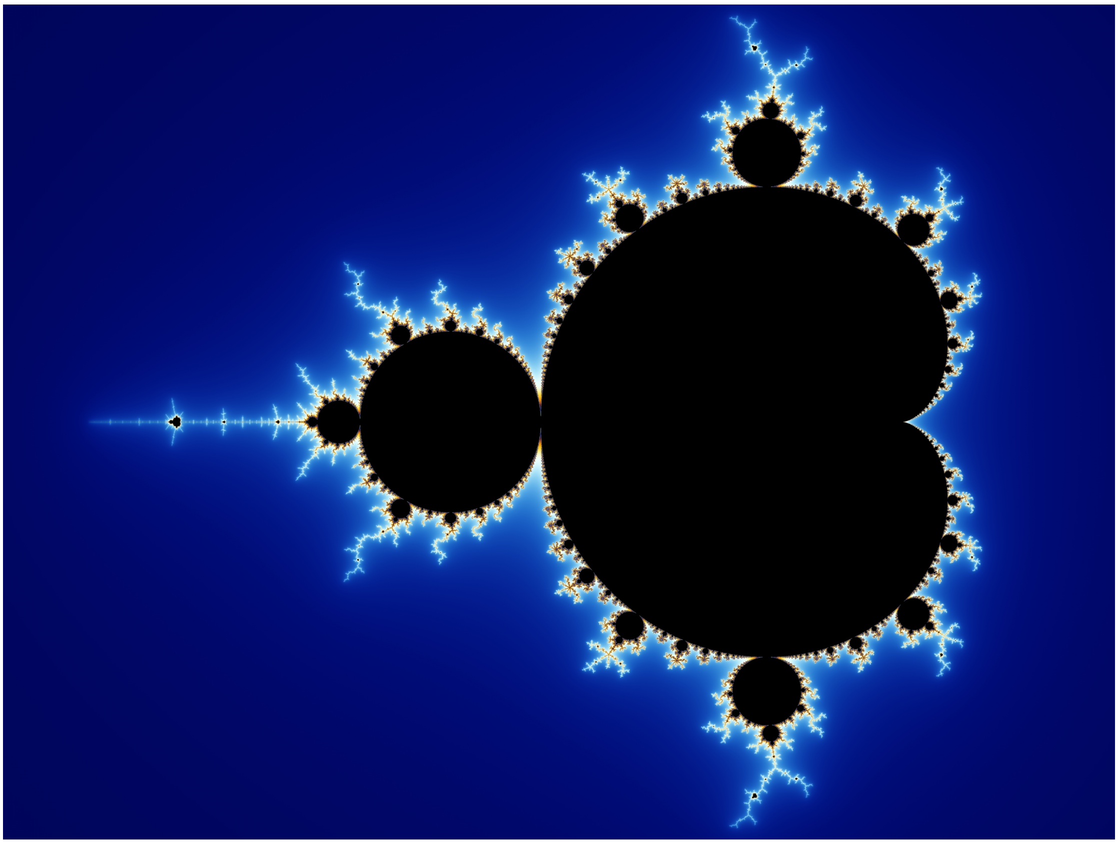


Figure 2.16: Graphical representation of the Mandelbrot set and its environment [27].

2.5.3 Generation Methods for Fractals

Depending on the nature of the fractal object, there are several methods of generating a fractal set, or a graphical representation of it. The most popular examples of fractal art, such as the Mandelbrot and Julia sets, are defined by the **iteration of complex polynomials**. For example, the Mandelbrot set, as shown in Figure 2.16 is defined as the set of complex numbers c where the recursive progression $z_{n+1} = z_n^2 + c$ does not escape to infinity. A computer program could map each pixel of a 2D image to a complex number $c = a + b \cdot i$, mapping the x coordinates to different values for a and the y coordinates to different values for b. For every given number, the program then determines if c is contained in the Mandelbrot set and colors the pixel accordingly.

Another method for creating fractals are **Iterated Function Systems**. Fractals generated by this method are usually called IFS fractals. The method has been described in detail by Barnsley [1]. An iterated function system is defined as a finite set of contraction mappings in a metric space M. For example, one could define an IFS Sierpinski triangle in the 2D Euclidean metric space, with the following contraction mappings: $T_1(x, y) = (\frac{x}{2}, \frac{y}{2})$

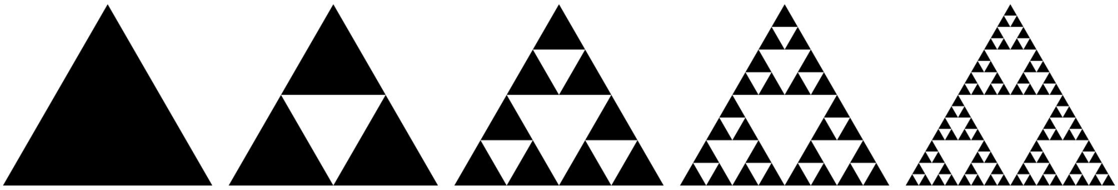


Figure 2.17: Recursive evolution of the Sierpinski triangle.

$T_2(x,y) = (\frac{x}{2} + \frac{1}{4}, \frac{y}{2} + \frac{1}{2})$
$T_3(x,y) = (\frac{x}{2} + \frac{1}{2}, \frac{y}{2})$
A graphical representation of the result is shown in Figure 2.17.

A number of fractals can also be generated using **Lindenmayer-Systems** [34], or short L-Systems, a mathematical formalism suggested by biologist Aristid Lindenmayer to simulate plant growth. An L-System is defined as a triple $G = (V, \omega, P)$, where

- V is a set of symbols,
- ω is the start word,
- and P is a set of replacement rules, each rule given as a pair $(\alpha \rightarrow \beta)$. Both α and β are words consisting of symbols in V. In each step α is replaced by β.

An L-System can produce a fractal if the replacement rules are recursive. The symbols can have semantics assigned, e.g., simple drawing operations for drawing fractal objects. Consider the following L-System that is used to draw the Koch Snowflake: $V = \{F, +, -\}$, $\omega = F - -F - -F$, $P = \{(F \rightarrow F + F - -F + F)\}$. The symbols have the following operations associated to them:

F: Move forward s^n units and draw a line, where s is a constant factor with $s < 1$. n is the number of the current iteration.

$+$: Rotate 60 degrees counter-clockwise.

$-$: Rotate 60 degrees clockwise.

Figure 2.15 shows the first four iterations:

1. $F - -F - -F$
2. $F + F - -F + F - -F + F - -F + F - -F + F - -F + F$
3. $F + F - -F + F + F + F - -F + F - -F + F - -F + F + F + F - -F + F - -F + F - -F + F + F + F - -F + F - -F + F - -F + F + F + F - -F + F - -F + F - -F + F + F + F - -F + F - -F + F - -F + F + F + F - -F + F$

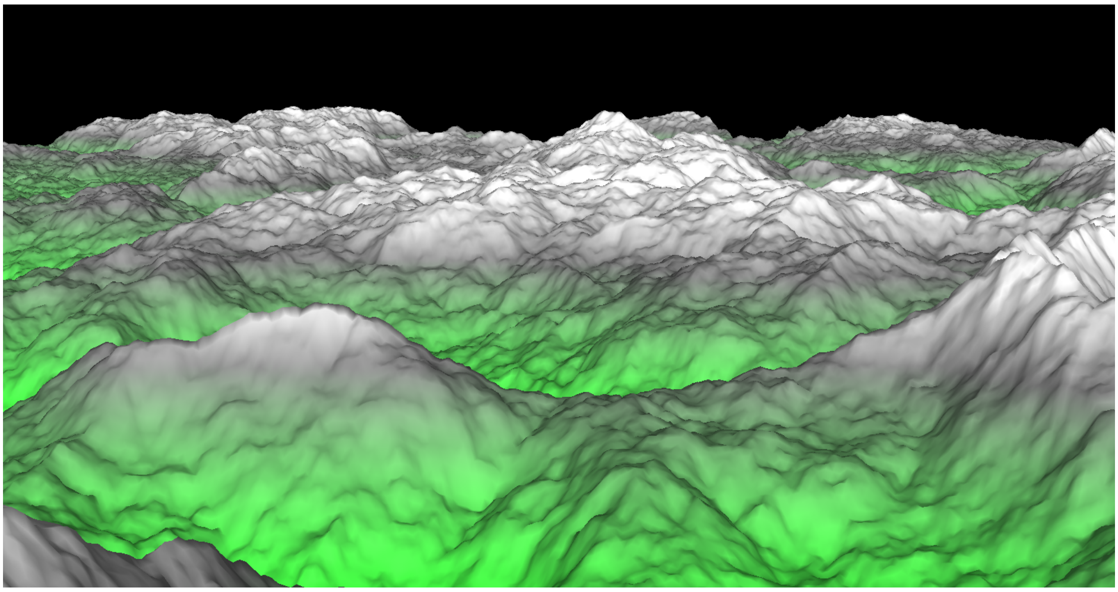


Figure 2.18: Fractal landscape rendered using the random midpoint displacement algorithm [8, 9].

4. $F+F--F+F+F+F--F+F--F+F--F+F+F+F--F+F+F+F-$
   $-F+F+F+F--F+F--F+F--F+F+F+F--F+F--F+F--F+$
   $F+F+F--F+F--F+F--F+F+F+F--F+F+F+F--F+F+F+$
   $F--F+F--F+F--F+F+F+F--F+F--F+F--F+F+F+F-$
   $-F+F--F+F--F+F+F+F--F+F+F+F--F+F+F+F--F+$
   $F--F+F--F+F+F+F--F+F--F+F--F+F+F+F--F+F-$
   $-F+F--F+F+F+F--F+F+F+F--F+F+F+F--F+F--F+F-$
   $-F+F+F+F--F+F--F+F--F+F+F+F--F+F--F+F--F+$
   $F+F+F--F+F+F+F--F+F+F+F--F+F--F+F--F+F+F+$
   $F--F+F--F+F--F+F+F+F--F+F--F+F--F+F+F+F-$
   $-F+F+F+F--F+F+F+F--F+F--F+F--F+F+F+F--F+F$

The Sierpinski triangle, shown in Figure 2.17 can be generated using an L-System as well.

2.5.4 3D Fractals

While most of this section has been dealing with fractals in the 2D space, some of the approaches and use cases for fractal objects in the (topological) third dimension should also be mentioned. One interesting application is the generation of realistic looking landscapes, i.e., terrain data from simple formulas. Here the self-similar structures closely resemble those of natural terrains. Fractal landscapes have been generated on computers as early as 1974, according to Mandelbrot [24]. Figure 2.18 shows an example.

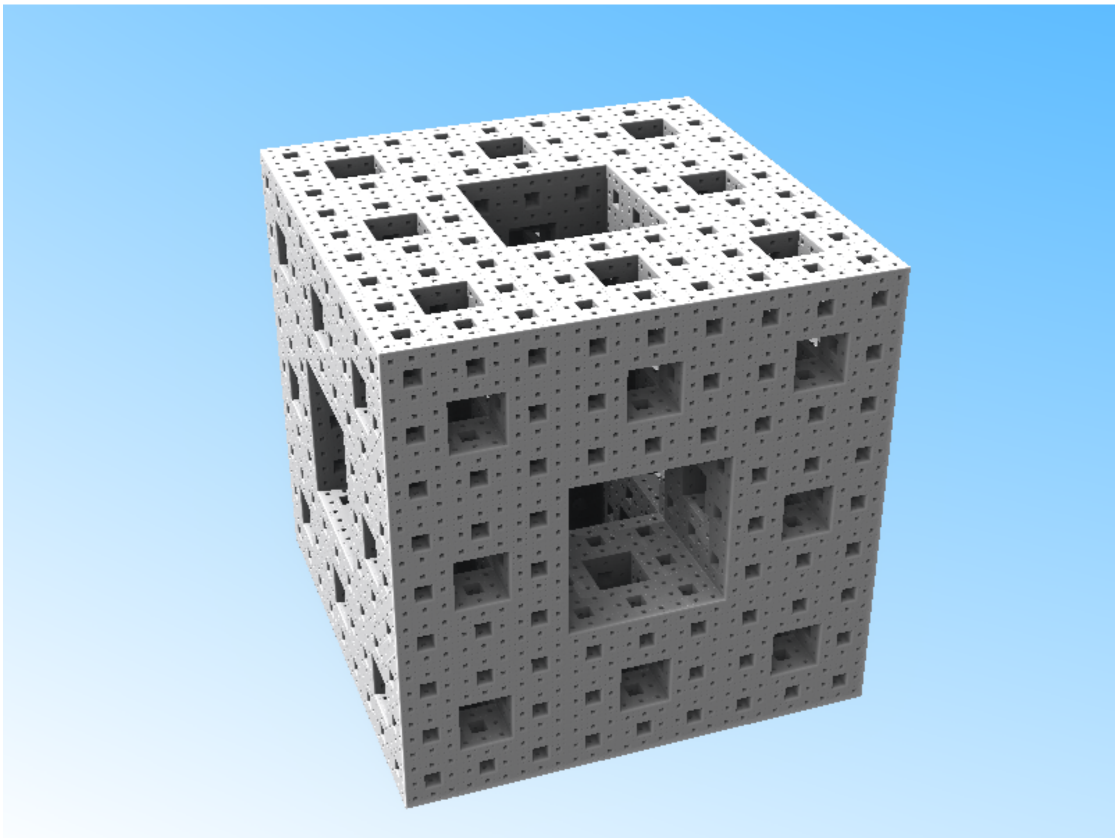


Figure 2.19: Rendering of the Menger Sponge fractal. Image generated using the open-source "Mandelbulber" program [28].

Some two-dimensional fractal objects can be easily extended to the third dimension, including most fractals generated by iterated function systems. One example is the Menger Sponge fractal. It is generated by repeatedly subdividing a cube into a set of 3x3x3, i.e., 27 smaller cubes, removing the middle cube of each face and the cube in the very center of the original cube. This subdivision is then repeated on the 20 remaining smaller cubes (see Figure 2.19).

Transforming fractals based on the iteration of complex polynomials into the third dimension is usually a non-trivial task, as the complex plane is inherently two-dimensional. For that reason, there is no canonical three-dimensional analogue of the Mandelbrot set. Nonetheless, there are numerous approaches for rendering 3D objects based on the Mandelbrot set. From these constructions, White and Nylander's "Mandelbulb"[26] stands out in particular. While it does not exactly resemble the shape of its 2D equivalent, it exhibits fractal detail in all three dimensions. See Figure 2.20 for a rendering of the 3D Mandelbulb set.

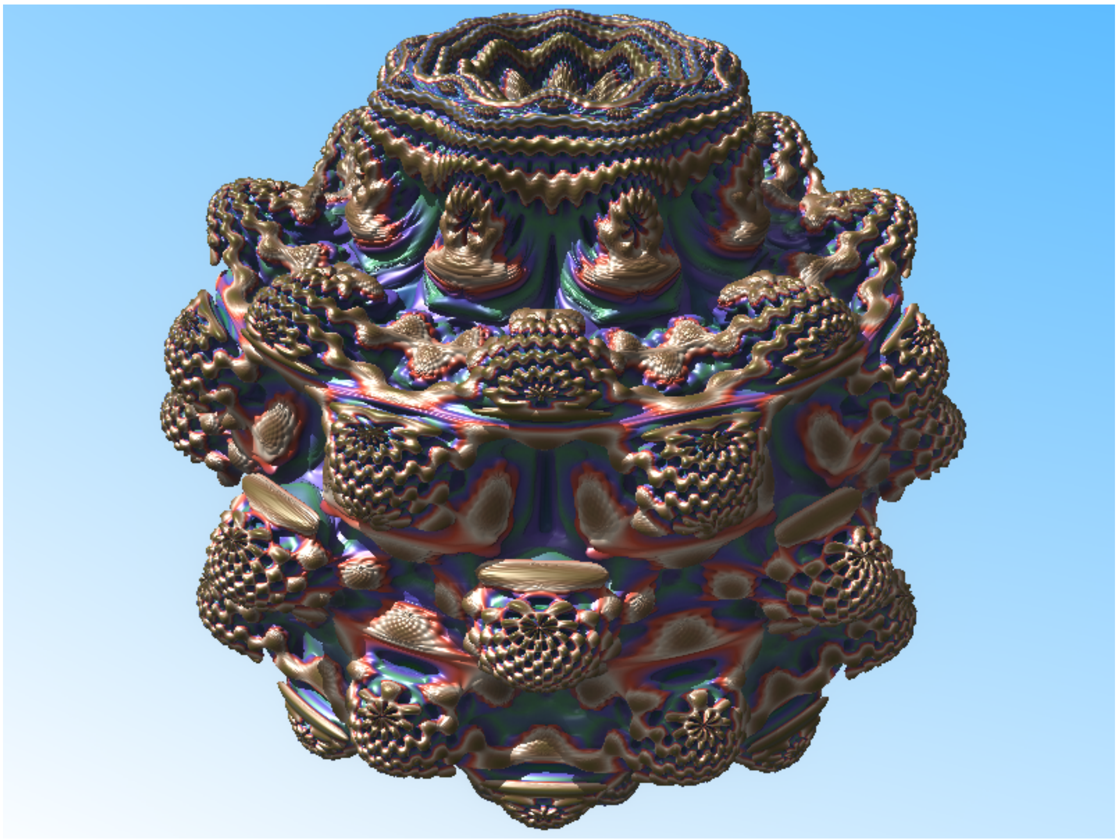


Figure 2.20: Rendering of the Mandelbulb fractal. Image generated using the open-source "Mandelbulber" program [28].

CHAPTER 3

Hofstetter's Inductive Rotation

Inspired by the findings of Penrose (see Section 2.3) and the desire to find a single prototile that would tile the plane aperiodically, artist Hofstetter Kurt experimented with various tilings of the plane. One construction, dubbed Inductive Rotation (IR) is of particular interest. Starting with a single prototile, the plane is filled following a simple recursive rotation scheme. The prototile is copied and rotated multiple times in one step, while the next step works on the result of those rotations, copying and rotating the entire pattern in the same fashion. The main difference to iterated function systems (IFS) is that the pivot point changes with each iteration, based on the resulting pattern.

While Penrose has always worked with tesselations in the mathematical sense, where overlaps are illegal, Hofstetter Kurt's construction explicitly includes overlapping regions, thus "hiding" part of the original prototile in each successive rotation step. This particular feature allows for the generation of seemingly nonperiodic patterns, even though only a single prototile is used.

Since the concept of Inductive Rotation is very young, there is still a lot of work missing from a scientific point of view. The resulting patterns seem to be nonperiodic, but a mathematical proof has yet to be found. While the method is clearly related to, even inspired by, Penrose's tesselations of the plane, in practice the overlapping regions generate a different class of patterns.

The point of my thesis is to implement a software program for generating and rendering Inductive Rotation patterns using different textures. The resulting tool will help the artist in fully realizing the artistic potential of the method and will enable him to explore the resulting patterns in greater detail. For implementation details, see Chapter 4.

This chapter contains a detailed description of the Inductive Rotation method and the properties of the generated patterns. The requirements for an implementation are isolated, and suitable algorithms for efficiently generating IR patterns are described.

3.1 Description of the Inductive Rotation Method

Inductive Rotation is an iterative scheme for generating large nonperiodic patterns from a single prototile. A “prototile” in the sense of Inductive Rotation is always an arbitrary image, confined to an area (subset) of the plane. Each iteration starts with the result of the previous one.

Currently, there are three different IR methods:

- The **3-way Inductive Rotation** consists of 3 90-degree rotations per iteration. The prototile image’s shape is a square.
- The **5-way Inductive Rotation** consists of 5 60-degree rotations per iteration. The prototile image’s shape is a regular hexagon.
- The **2-way Inductive Rotation** consists of 2 120-degree rotations per iteration. The prototile image’s shape is a star figure that can be built from 6 rhombi with angles $\alpha = \frac{\pi}{3}$ and $\beta = \frac{2\pi}{3}$. The prototile can be seen in Figure 3.6.

The pivot point for the rotations changes with each iteration. How exactly the pivot point is determined is dependent on which of the three IR methods is used and will be explained later.

To outline a basic scheme, that is shared between all three IR methods, some variables have to be introduced first.

- Let i be the current iteration number. Before the first iteration, i.e., for starting with a single prototile, let $i = 0$. After the first iteration, let $i = 1$, and so on.
- Let c be the rotation count per iteration. For example, the 3-way IR scheme will use $c = 3$.
- Let α be the rotation angle. For example, the 3-way IR scheme will use $\alpha = 90$. It can be observed that $\alpha = \frac{360}{c+1}$.
- Let $T_{i,0}$ be the pattern resulting from iteration i. $T_{0,0}$ is the prototile image, $T_{1,0}$ the pattern after the first iteration, and so on.
- Let R_i be the axis-aligned bounding rectangle of the pattern $T_{i,0}$. The functions $x(R_i)$, $y(R_i)$, $w(R_i)$, and $h(R_i)$ shall result in the leftmost x-coordinate, topmost y-coordinate, the width and the height of this bounding rectangle, respectively.
- Let $p(i)$ be a function resulting in the correct pivot point for iteration i. This function will be different for the 3-way, 5-way and 2-way IR methods.

Each iteration, regardless of the specific IR method then consists of the following steps.

1. Create c copies of $T_{i,0}$. Refer to them as $T_{i,1}$ through $T_{i,c}$. Handle overlaps so that patterns with a higher index will be placed behind patterns with a lower index. For example, $T_{i,2}$ will be placed behind $T_{i,1}$, which is placed behind $T_{i,0}$.

2. Rotate each copy around the pivot point $p(i)$ by a multiple of α degrees. $T_{i,1}$ will be rotated by α degrees, $T_{i,2}$ will be rotated by 2α degrees, and so on.
3. The resulting pattern is now referred to as $T_{i+1,0}$.

This is obviously a recursive procedure that can be repeated infinitely.

By choosing the right values for α, c, and defining $p(i)$ accordingly, each of the three methods can be executed using this basic scheme that is used as a template for the algorithm described in Section 3.4.2. The definitions for $p(i)$ for the 3-way, 5-way and 2-way methods can be found below, in Sections 3.1.1, 3.1.2, and 3.1.3.

One important aspect of IR is the fact that overlapping regions are not only allowed, but enforced. The pivot point is always chosen so that each newly placed and rotated copy of the current image has a region that overlaps with other images. While the original prototile is a flat 2D image, all subsequent iterations should be seen as a three-dimensional pattern. Patterns generated by IR methods have a property that shall be called Maximum Overlap Depth (MOD). At each point of the resulting pattern, the overlap depth can be determined by counting how many prototile images are hidden at this point through overlaps. The maximum overlap depth of all possible points is the MOD. A prototile image with no overlaps has MOD 0.

The assumed nonperiodicity of the generated patterns stems from the fact that overlaps are ordered in the way described in the rotation steps above. When the overlapped regions are discarded, the result is a seemingly nonperiodic tiling of the plane. In a sense, a single prototile can form an aperiodic set when combined with one of the IR methods.

3.1.1 3-Way Inductive Rotation method

The 3-way Inductive Rotation method is presumably the first conceived IR method. Since only multiples of 90-degree rotations are used, all prototile copies stay axis-aligned. The pivot point $p(i)$ is determined by choosing the rightmost vertical midpoint of the current image $T_{i,0}$. It can be defined as $p(i) = p_x, p_y$ with $p_x = x(R_i) + w(R_i)$ and $p_y = y(r_i) + \frac{h(R_i)}{2}$. Figure 3.1 shows the prototile and the first two iterations. Figure 3.3 shows the overlapping regions of the first three iterations.

Interestingly, while the generated pattern seems nonperiodic as a whole, it can be shown that starting with $i = 2$, a subset of the pattern forms a periodic pattern. The areas that are part of this subset can be referred to as “periodic holes”. Figure 3.2 depicts the concept. As will be seen later, a similar structure can also be found for the 2-way IR method. Observations suggest that the MOD for any iteration of the 3-way IR method with $i > 0$ is 1.

3.1.2 5-Way Inductive Rotation method

For the 5-way variant, 60-degree rotations are used. As with the 3-way Inductive Rotation, the pivot point is the rightmost vertical midpoint and can be defined in the same way. Figure 3.4 shows the prototile as well as the first two iterations.

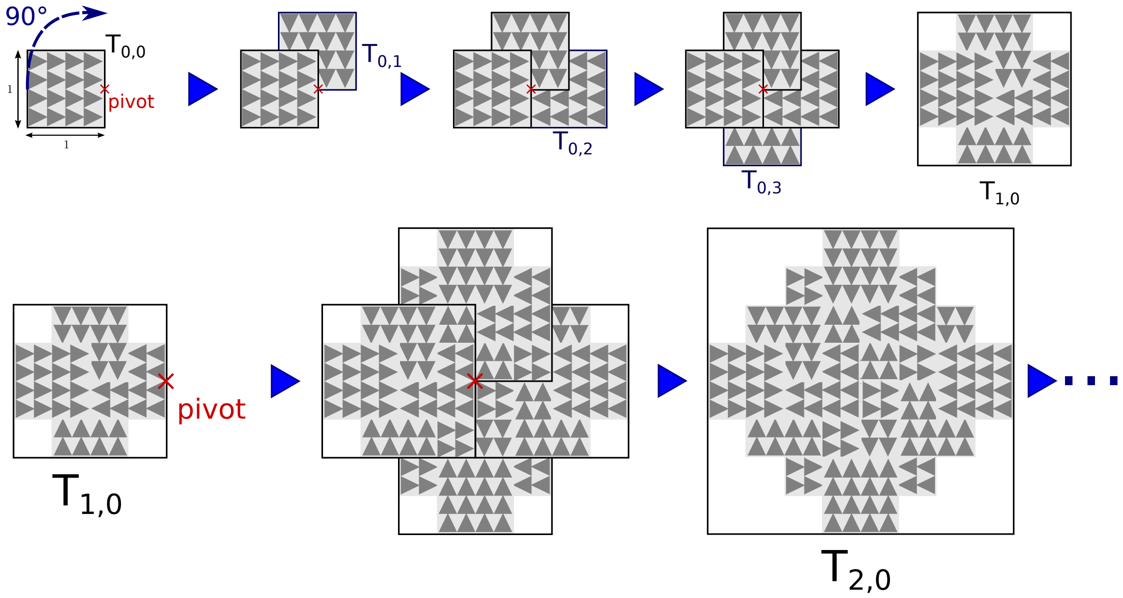


Figure 3.1: The 3-way Inductive Rotation scheme.

An interesting feature of the 5-way Inductive Rotation pattern is that the MOD does not stay constant, but according to results of experimental observation seems to double with every iteration. For an illustration of the first few iterations' overlaps, or hidden regions, see Figure 3.5.

3.1.3 2-Way Inductive Rotation method

For the 2-way variant, 120-degree rotations are used. Figure 3.6 shows the prototile and the first two iterations. Figure 3.8 shows the overlapping regions for the first four iterations.

While the 3-way and 5-way methods have an unambiguous definition of the pivot point, i.e., there are no known alternatives, this is not the case with the 2-way method. The pivot point in the first iteration is the rightmost vertical midpoint of the prototile. After the first iteration, there is an ambiguity, however, as none of the two prototile copies are centered on the same horizontal line as the original prototile and either one could be chosen for determining the pivot point. Hofstetter Kurt thus defined that for his variant of the 2-way method, in case there is more than one possibility, the pivot point shall always be the one with the bottommost y-coordinate. A more formal definition for the pivot point is $p(i) = p_x, p_y$ with
$p_x = x(R_i) + \frac{w(R_i)-w(R_0)}{6}$,
$p_y = y(R_i) + h(R_0)\frac{3^{\lfloor\frac{i+1}{2}\rfloor}-\lfloor\frac{i+1}{2}\rfloor}{2}$.

A notable difference to the other methods is that the outline of the pattern is not regular, but changes with each iteration. This is most likely a direct result from the

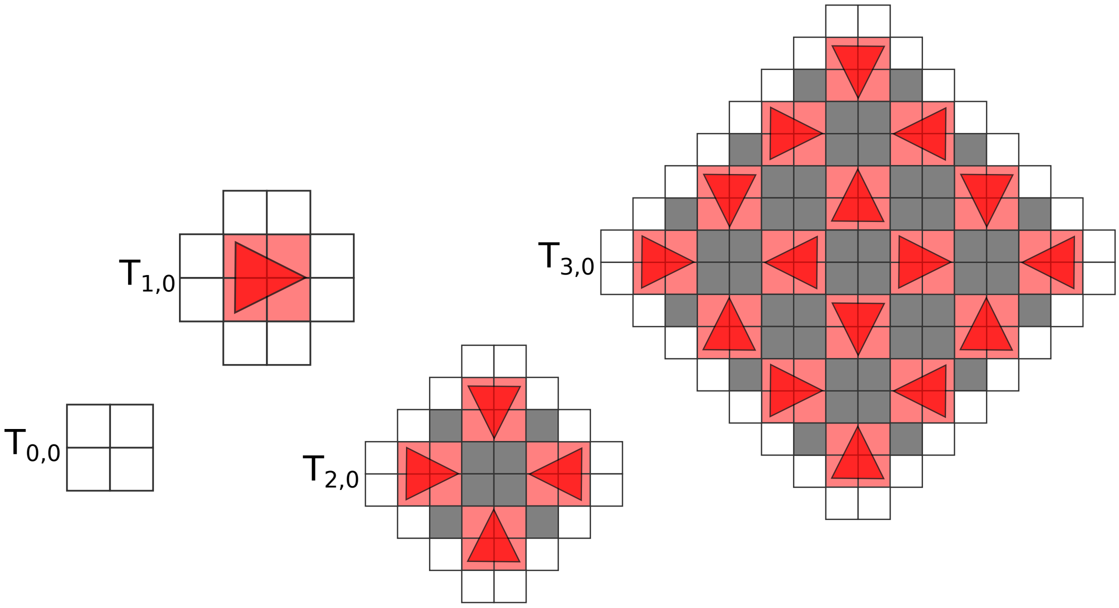


Figure 3.2: Periodic holes in the 3-way Inductive Rotation pattern. Note that the overlaps in $T_{1,0}$, which are illustrated with a red triangle, do not seem to overlap in subsequent iterations. If only the areas marked with red triangles in $T_{2,0}$ are considered, the pattern is invariant to 90-degree rotations, thus forming a regular sub-pattern. It is conjectured that this does not change in later iterations, though a formal proof is still missing. Areas marked white have no overlaps, while grey and red areas have overlap depth 1.

method of choosing the pivot point. On a first glance, the outline is reminiscent of a self-similar fractal curve. Like with the 3-way method, a subset of the pattern is periodic, see Figure 3.7. Observations suggest that the MOD for any iteration of the 2-way IR method with $i > 0$ is 1.

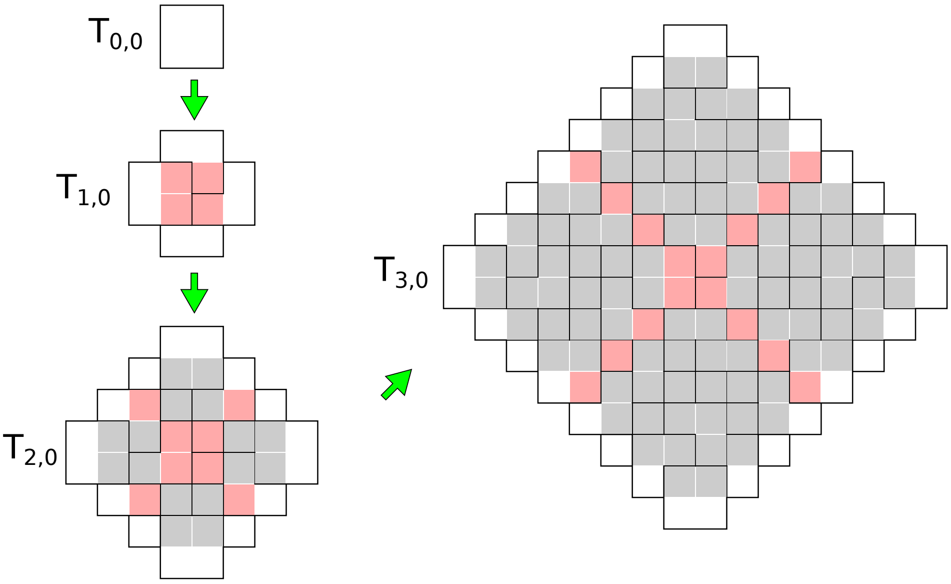


Figure 3.3: Overlapping regions in the 3-way Inductive Rotation pattern. Red areas mark newly overlapping regions, grey areas mark overlapping regions from the previous iterations, and white areas mark regions with no overlaps.

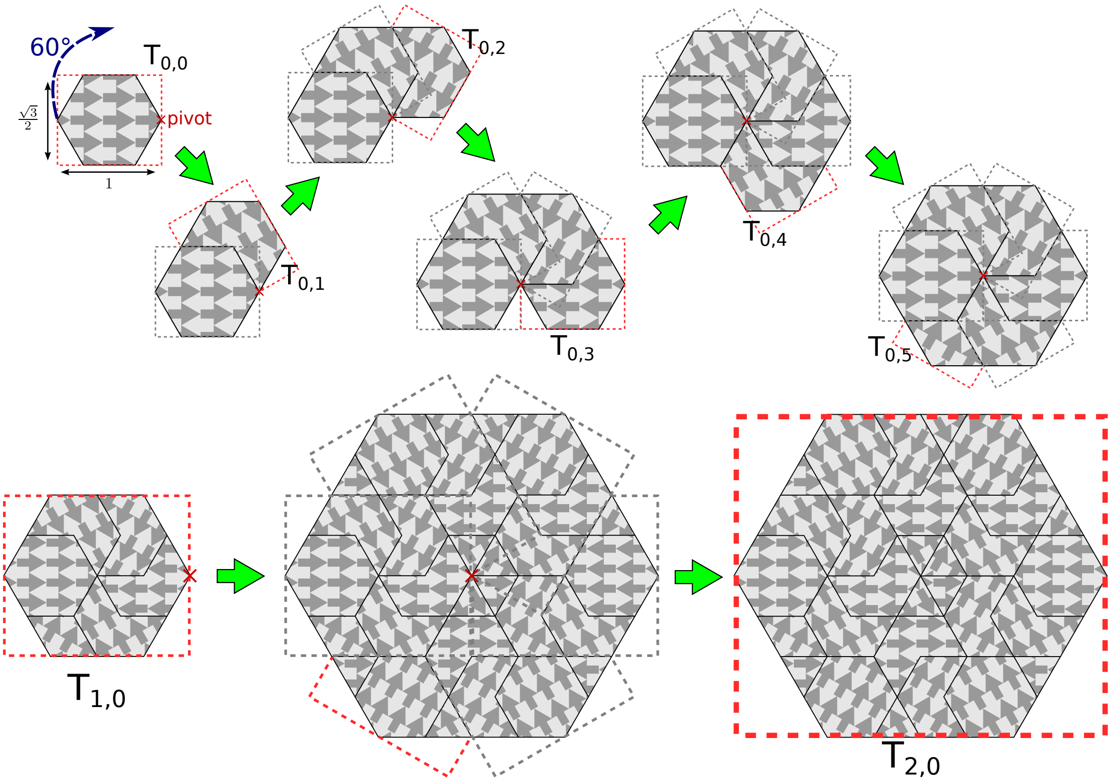


Figure 3.4: The 5-way Inductive Rotation scheme.

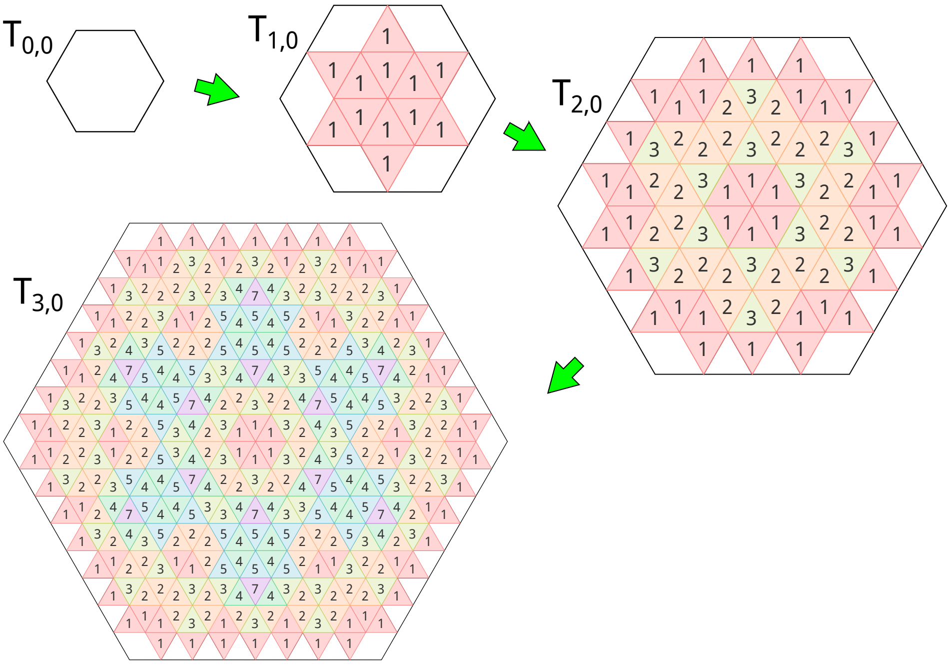


Figure 3.5: Overlapping regions in the 5-way Inductive Rotation pattern. Numbers denote overlap depth. Colors correspond to numbers and are used only for better visualization of the overlap patterns.

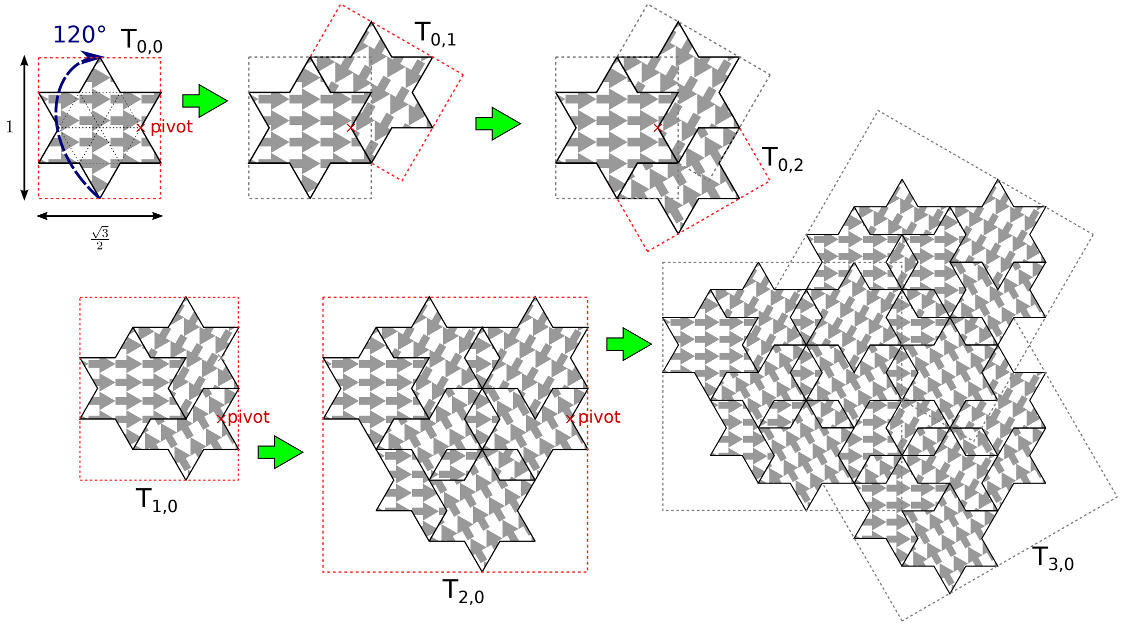


Figure 3.6: The 2-way Inductive Rotation scheme.

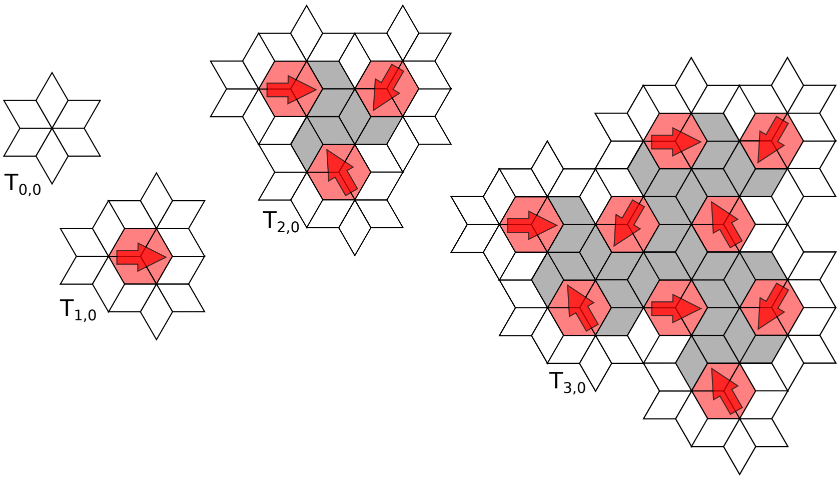


Figure 3.7: Periodic holes in the 2-way Inductive Rotation pattern. Similar to Figure 3.2, the areas forming the periodic sub-pattern are marked with red arrows. Note that the pattern marked in red in $T_{2,0}$ is invariant under 120-degree rotations. These sections also do not seem to overlap in future iterations, thus forming a periodic sub-pattern. It is conjectured that this does not change in later iterations, though a formal proof is still missing. Areas marked white have no overlaps, while grey and red areas have overlap depth 1.

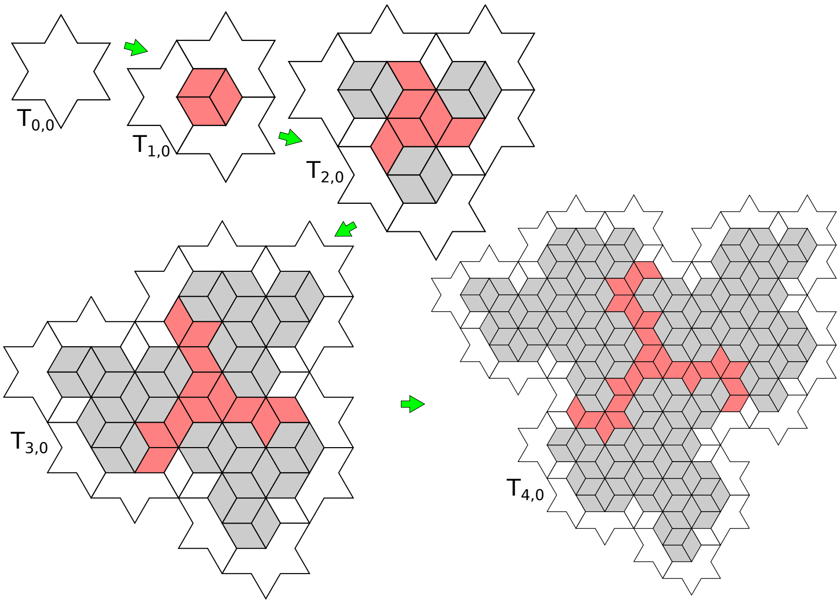


Figure 3.8: Overlapping regions in the 2-way Inductive Rotation pattern. Red areas mark newly overlapping regions, grey areas mark overlapping regions from the previous iterations, and white areas mark regions with no overlaps.

3.2 Artistic Opportunities

As the Inductive Rotation concept has been conceived by artist Hofstetter Kurt, one of the primary concerns from the very beginning was artistic development. It is deemed necessary to explain at least some of the concepts in this section, so the reader can get a glimpse of the fascination that lies behind the artistic experimentation on the thin line between order and chaos, that is inherent to aperiodic patterns such as those generated by Inductive Rotation.

The breaking of symmetry, or breaking of serialism is achieved by a simple recursive rule, yet after only a few iterations the outcome usually exceeds the human imagination, as the patterns expand rapidly. The result is a vivid image. The human brain gets caught endlessly in the pursuit of finding new paths and patterns in the generated images. See Figure 3.9 for an example of such an image.

In an introduction to Hofstetter Kurt's IR patterns, art theoretician Wolf Guenther Thiel states:

> "This interaction between subjective perception, the search for meaning, and objective forms means for the beholder a lasting experience." [17]

The oscillation between order and chaos, symmetry and asymmetry that is inherent to the occupation with such patterns brings new impulses and ideas that can be used for artistic merit. The generation of and experimentation with IR patterns and images is therefore not an end in itself. It can be seen as a creative stimulus for further artistic production. The most essential element thereof is the prototile. Replacing or changing it also changes the essence of the entire generated image.

While the generated patterns can be seen as 2D art, there are always overlaps, parts that are hidden behind, essentially transforming the image into a spatial entity, or a 3D structure. When treated as a spatial entity rather than a flat 2D image, the IR patterns can also serve as artistic input for sculptural works.

Apart from that, there is a multitude of possible uses for these patterns generated through the IR methods, even if some of them are not of particular interest from an artistic point of view. Here is a small selection:

- Generation of textures for architecture, games or simulation
- Moving images (video)
- Musical compositions
- Generation of weaving patterns for the textile industry

3.3 Requirements of the Irrational Image Generator

While the way of generating the resulting patterns, or the outcome on a high level can only change with altering the parameters of the process, i.e., α, c or $p(i)$, as defined

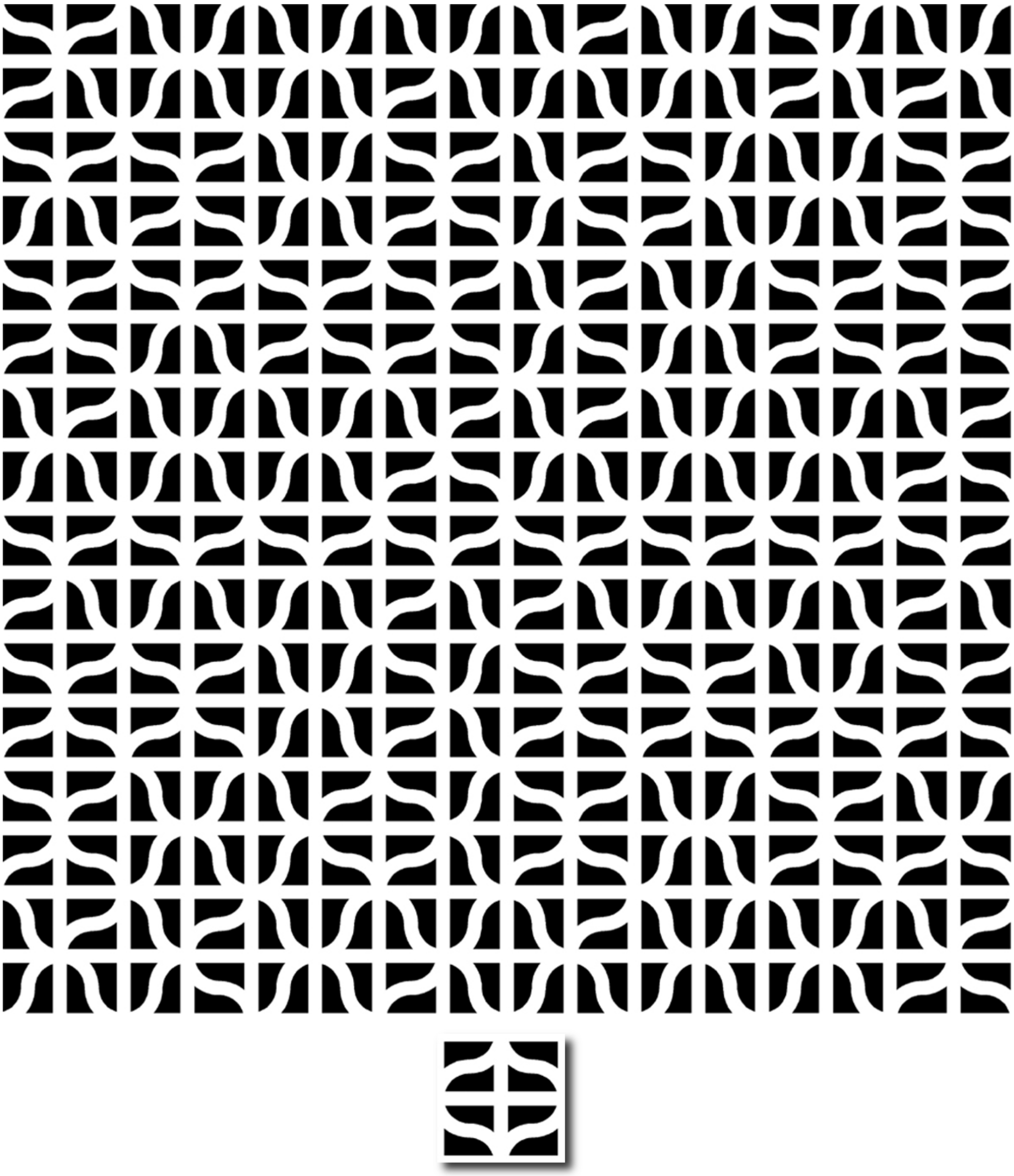


Figure 3.9: “ELEMENTARY WAVE” by Hofstetter Kurt, 2008 ©VBK Wien. This artwork has been created by the 3-way Inductive Rotation method. The single tile at the bottom is the prototile and has been added for better understanding; it is not part of the original artwork.

in Section 3.1, the observed outcome on a lower level, or the general appearance of the generated images, also changes depending on the pattern contained in the prototile $T_{0,0}$.

Since the outcome of Inductive Rotation is hard to predict, the creation of new prototiles often results in an extensive trial-and-error procedure where the artist selects from a multitude of possible images and tries to generate aesthetically pleasing patterns using the Inductive Rotation approach.

The goal of this thesis is to create a new tool for accelerating this process and to give the artist a faster and more intuitive workflow experience for experimenting in the realm of IR patterns. As has been already stated, the basic process, i.e., rotating/duplicating the base pattern $T_{0,0}$, is always the same, and is therefore destined to be performed automatically by a computer program.

The requirements for such a tool have been collected in accordance with the needs of Hofstetter Kurt, who will be the primary user of the tool. The following list states the basic, i.e., mandatory requirements:

- Fast generation and rendering of IR patterns using the 3-way, 5-way and 2-way methods.
- Insertion of arbitrary images as $T_{0,0}$ patterns, also called "prototiles". Images with alpha transparency should also be supported.
- Easy navigation through the generated pattern. It should be possible to view sections of large IR patterns at full resolution by scrolling and zooming in.
- Output of high-resolution IR patterns, or parts of IR patterns as image files.
- Intuitive workflow, e.g., accessing all important operations through keyboard shortcuts, inserting image data from other programs through the clipboard, loading of common image file formats such as PNG.

Apart from these basic features, it would be desirable to provide additional ways to explore the IR patterns. One such feature would be to view the parts of the pattern that are hidden through overlaps. Another advanced feature would be to export the patterns as polygon meshes so that they could be used in 3D modeling software such as Blender or Autodesk Maya.

3.4 Algorithms and Technologies

This section outlines possible algorithms and technologies for the development of an IR pattern generation tool.

3.4.1 Iterated function

Similar to using L-Systems for generating fractals (see Section 2.5), it is possible to build Inductive Rotation iterations recursively from simple drawing operations. It is pointed out that this construction method is not an L-system: Instead of evaluating recursive replacement rules, it uses an iterated function.

Consider the following symbols representing basic drawing operations:

draw (D): Draw a pattern $T_{0,0}$ with its center located at the current position. Omit any overlaps, so the newly drawn pattern does not replace or overlap already existing patterns.

rotate (R): Rotate 90 degrees clockwise.

move up (M): Move up $\frac{w}{2}$ units, where w is the width of $T_{0,0}$.

move right (N): Move right $\frac{w}{2}$ units. This operation is just for convenience and could be replaced by $RMRRR$.

Now consider an iterative function f_i which draws the ith iteration of the 3-way Inductive Rotation method. Defining f_0 is trivial:

$f_0 = D$.

Since f_1 is basically just a composition of 4 translated and rotated instances of f_0, it can be defined as the following:

$f_1 = (f_0 MRM)^4 = DMRMDMRMDMRMDMRM$.

Note that a^x in this context means that the operation a is repeated x times. It can be observed that the resulting pattern $T_{1,0}$ has exactly twice the width and height of $T_{0,0}$. For f_2, the same approach can be used, but the distance has to be doubled for the translation. Additionally, after executing f_1 we do not arrive at the center of the pattern generated by f_1, but in the center of the last f_0 pattern, so an extra translation to the right is needed:

$f_2 = (f_1 N M^2 R M^2)^4 =$
$DMRMDMRMDMRMDMRMNMMRMM$
$DMRMDMRMDMRMDMRMNMMRMM$
$DMRMDMRMDMRMDMRMNMMRMM$
$DMRMDMRMDMRMDMRMNMMRMM$.

It can be observed now that the pattern width and height doubles with each iteration. It is possible to devise an iterative formula for all $n \geq 2$. Each iteration f_i consists of four translated and rotated patterns generated by f_{i-1}. After executing f_{i-1}, a translation to the right of $\frac{w(R_{i-1})}{2} = 2^{i-2} w(R_0)$ is needed to get to the center of the pattern generated by f_{i-1}. This is followed by a translation by $\frac{w(R_i)}{2} = 2^{i-1} w(R_0)$ to the top and to the right, expressed by $M^{2^{i-1}} R M^{2^{i-1}}$, which already includes the rotation by 90 degrees needed before drawing another pattern generated by f_{i-1}. Symbols R_i, and $w(R_i)$ are defined in Section 3.1. The iterative formula for all $i \geq 2$ therefore is:

$f_i = (f_{i-1} N^{2^{i-2}} M^{2^{i-1}} R M^{2^{i-1}})^4$.

While this formula only works for the 3-way method, it should not be too hard to come up with similar recursive formulas for both the 5-way and even the 2-way methods. The advantage of this approach is the concise mathematical description of the underlying concept. It uses relatively few basic operations that should be easy to implement.

This is, however, not the best approach for implementing a tool meeting the requirements stated above. One problem is that the formula has to be adapted for every new Inductive Rotation method. A general approach, where only minor parts or parameters have to be replaced in order to support a new variant of the Inductive Rotation would be more desirable. Another drawback is that this approach has been devised without any specific technologies or data structures in mind. While it is certainly possible to transform this approach into a computer program, there are other approaches resulting in a simpler program and better performance.

3.4.2 Copy and rotate around pivot

Apart from using an iterated function, there is also the possibility of directly implementing the rotation scheme outlined in Section 3.1. This approach was chosen for implementing the Irrational Image Generator, see Chapter 4.

The idea is to define a data structure holding the patterns produced by Inductive Rotation, along with a set of operations that work on the data structure and that can be used to execute the IR scheme. In this section, only the basic algorithm is outlined. The implementation details, such as the concrete data structure and details on how the operations can be implemented are treated in Chapter 4.

For iteration zero, i.e., $T_{0,0}$ with no iterations performed, the data structure holds one instance of the prototile. To iterate on the data structure and generate $T_{1,0}$ — or more generally T_{i+1} — three higher-level operations have to be implemented.

pivot: Determine the rotation pivot point $p(i)$ based on the current iteration number i.

rotate-copy: Create a copy of the entire data structure that is rotated by α degrees around $p(i)$.

merge: Merge one data structure into another one, either omitting overlaps, or, if the data structure supports this, place overlapping data behind existing data.

For better illustration of how these operations relate to the IR scheme described in Section 3.1, a pseudocode-implementation is given in Algorithm 3.1.

```
Data: $S$ is a data structure already containing the prototile, therefore resembling
      pattern $T_{0,0}$.
Input: $n$ is the number of iterations that should be performed.
Input: $p$ holds the current pivot point $p(i)$.
for $i \leftarrow 0$ to $n-1$ do
    $p \leftarrow$ pivot($S$, $i$);
    for $j \leftarrow 0$ to $c-1$ do
        $S_j \leftarrow$ rotate-copy($S$,$p$,$j\alpha$);
    end
    for $j \leftarrow 0$ to $c-1$ do
        merge($D$, $S_j$);
    end
end
```

Algorithm 3.1: Pseudocode for generating the pattern $T_{n,0}$ from a single prototile $T_{0,0}$ using the operations described earlier. For definitions of i, c, α, and $T_{i,j}$, see Section 3.1.

This approach implements the basic concept of the Inductive Rotation that is the same for the 3-way, 5-way, and 2-way methods. In general, it is only necessary to change the **pivot** operation as well as the number of rotations c and the angle α for each method, given that all three methods can use the same data structure. It can be seen in Section 4.4 that this is not always the case.

3.4.3 Grid alignment

A closer examination of the patterns generated by applying the 3-way, 5-way or 2-way Inductive Rotation methods shows that they can be aligned to a regular grid, see Figure 3.10.

For the patterns generated by the 3-way IR method, the grid consists of squares with width and height of half the width and height of the square prototile $T_{0,0}$.

For the patterns generated by the 5-way IR method, the grid is a mesh of equilateral triangles, provided $T_{0,0}$ is limited to the regular hexagon, which can be decomposed to equilateral triangles.

For the patterns generated by the 2-way IR method, the grid is a mesh of rhombi with angles $\alpha = \frac{\pi}{3}$ and $\beta = \frac{2\pi}{3}$, provided $T_{0,0}$ is limited to the star figure described in Section 3.1, which can be decomposed into such rhombi. Note that this rhombi mesh can be transformed into the triangle mesh of the 5-way IR method by splitting each rhombus into two equilateral triangles.

Since all of the prototile edges coincide with grid edges, consequently all of the overlapping regions of the generated patterns can also be matched with the grid. When looking at the overlaps in Figures 3.3, 3.5, and 3.8, and comparing them to the respective grid structures in Figure 3.10, it can be seen that overlapping regions never cross grid borders. Therefore, by using a grid-like data structure, the pattern could be easily decomposed into several layers, where each layer corresponds to a level of overlap, i.e.,

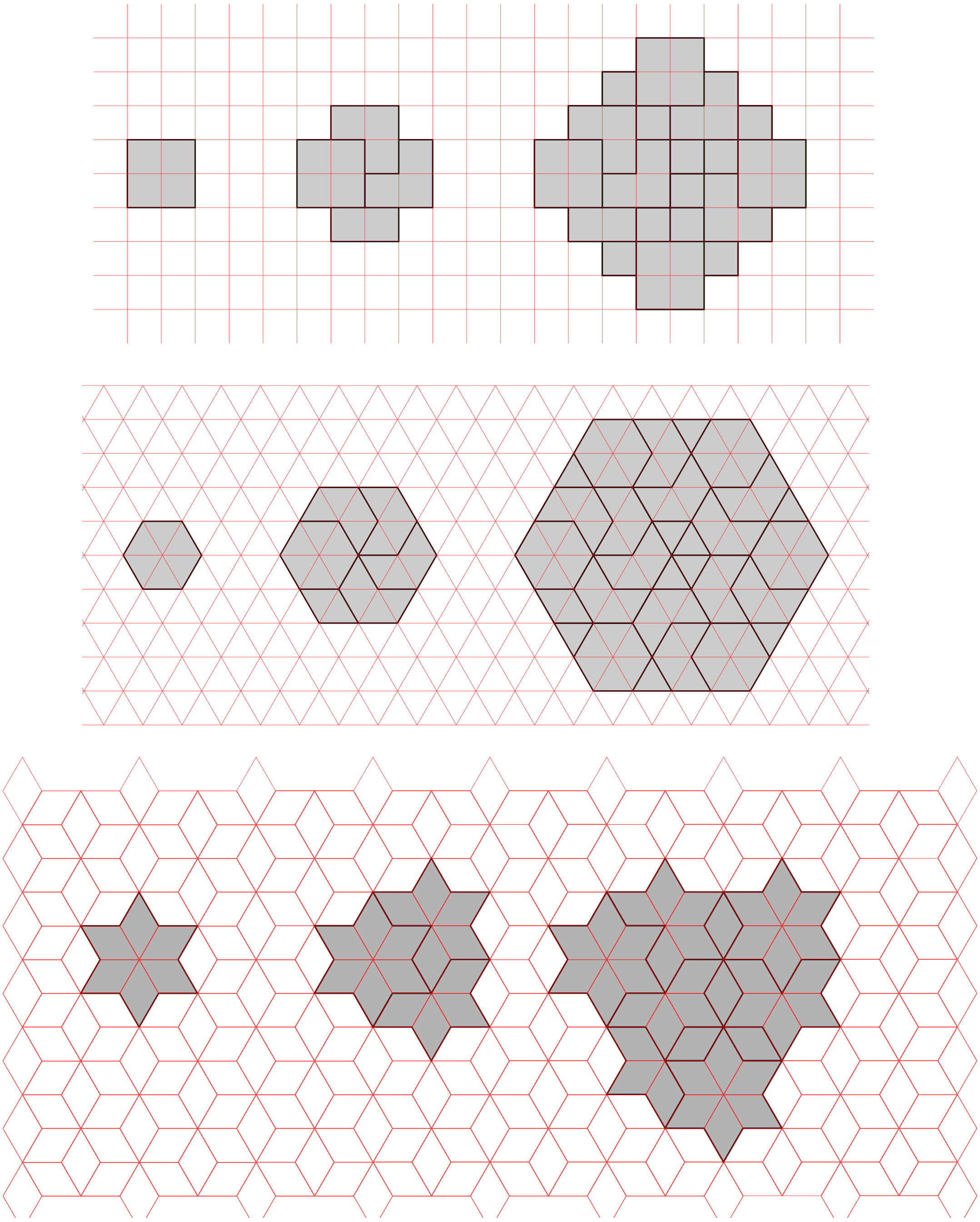


Figure 3.10: From top to bottom: Patterns generated by the 3-way, 5-way and 2-way methods with their corresponding grid alignment.

the first layer would contain the visible portion of the pattern, the second layer would contain all hidden regions that lie directly behind the first layer, and so on. For the 3-way and 2-way methods there would be two such layers. For the 5-way method the number of layers is conjectured to be 2^i for each $T_{i,0}$. Generally, the number of layers is the maximum overlap depth, plus one.

This type of assigning overlapping regions to layers is used in the Irrational Image Generator to allow for better visualization of the overlapping regions. For a more detailed description on how to create such a data structure, see the implementation in Chapter 4.

3.4.4 Rendering methods

For an IR pattern generation tool, there are several possibilities for rendering the patterns as images. The type of renderer also plays an important role in defining the algorithms and data structures. Different rendering methods require different data representations and therefore different algorithms for copying/rotating/merging the contained data.

3.4.4.1 Vector vs Raster graphics

The pattern contained in the prototile $T_{0,0}$ can either be represented by vector graphics or raster graphics. Vector graphics represent image data as a composition of geometric primitives such as lines or polygons with properties such as thickness, width, height, rotation, or color. Raster graphics on the other hand represent images as a regular $m \cdot n$ raster of equally sized dots, called pixels; each pixel has one distinctive color.

Vector graphics have the great advantage that they can be rendered at virtually any resolution. The pattern could be exported as vector data, e.g., PDF, for high-quality printing. In almost any case, if a vector graphics representation is compared with its high-resolution raster graphics representation, the data needed to construct the vector graphics takes up much less space than the equivalent pixel data from the raster representation. The disadvantage is that vector graphics are usually more difficult to render, both in terms of complexity and CPU usage. Both printers and screens need a raster representation anyway, so the conversion from vector to raster representation is an additional step in the rendering process.

In the case of creating the Irrational Image Generator tool, Hofstetter Kurt specifically asked for the ability to load PNG and JPG files created in Adobe Photoshop, both of which are raster graphics formats. Since there is currently no need for vector-based prototiles, the prototiles in the Irrational Image Generator are treated as raster graphics only. For vector-based prototiles, e.g., in SVG or EPS format, it is always possible to convert them to high-resolution raster graphics if needed.

3.4.4.2 Image representation

Even as raster images are used for prototiles, the next question concerns the representation of the 2D space wherein the pattern, i.e., the rotated and translated copies of the prototile $T_{0,0}$, is placed. Under the previous assumption that the prototiles are raster graphics, there is the possibility of using just one large expandable bitmap buffer and simply copying/rotating the individual pixels for each iteration. Starting from the approach described in Algorithm 3.1, the **rotate-copy** operation would create a copy of the bitmap buffer where all the pixels are translated according to the rotation around the pivot point. The **merge** operation in turn would compare two bitmap buffers and only copy pixels from one buffer onto the other one if they would not replace existing pixels. In practice, the bitmap buffer takes up too much space to be usable for larger patterns. As can be seen in Section 3.4.5, IR patterns expand rapidly after just a few iterations. For any purely bitmap-based approach, the generation cost gets multiplied with the width or height of the prototile image itself. The maximum iteration number — i.e., the number of iterations that a standard PC is able to generate without running out of memory — is lower than with approaches taking into consideration that there exists only a limited amount of raster data, that is duplicated over and over again. Another great disadvantage is that every time the prototile changes, the pattern has to be generated from scratch again. This can be prevented by choosing a data structure that only references the prototile instead of replicating it. This limitations have already been observed in a previous attempt at creating a tool for IR image generation, developed during the course of a practical work at the Institute of Computer Graphics and Algorithms, Vienna University of Technology. The limitations of this tool were one of the reasons for creating a new tool in the course of this thesis.

Since any IR pattern only consists of translated and rotated copies of the base pattern, or prototile, which is limited in size, there is a lot of redundancy in the raster representation of the image. Therefore it makes more sense for the representation to only store references to the prototile instead of copies of the individual pixels. With a grid as described in Section 3.4.3, a raster of grid cells referencing rotated parts of the prototile could be used instead of a pixel raster. But even in the case where the pattern is not aligned to a grid, it is still possible to only store references to the prototile, e.g., in the form of a reference point (x, y) where the prototile is placed in 2D space, and the rotation in degrees. In the latter case, where no grid alignment is used, overlaps are a bit more difficult to process during the **merge** operation. In that case it is practical to store an additional z coordinate for each prototile reference, so the renderer knows in which order to draw them.

With an efficient implementation, the advantage of this approach are reduced memory consumption for storing the generated patterns and faster iteration since less data needs to be copied and merged. Of course, this improvement still does not change the algorithmic complexity of the approach (see Section 3.4.5). Nonetheless, in practice the improvement is quite noticeable, since higher iteration counts can be reached on the same hardware.

3.4.4.3 Image rendering

The requirements outlined in Section 3.3 state that the tool should be able to display arbitrary portions of the generated image at different zoom levels. If a large pattern, that does not fit on a single screen without any scaling, is considered, it should on the one hand be possible to get a downscaled view of the entire pattern, on the other hand it should also be possible to “zoom in” on arbitrary portions of the pattern.

Also, to allow for a fluent interaction with the tool, the renderer should be fast enough to allow for real-time interaction. To achieve this, the renderer could make use of techniques such as texturing and mipmapping. Texturing allows for an efficient display of rotated raster images by a simple lookup instead of rotating the pixel data itself in each iteration. Mipmapping allows for efficient high-quality up- and downscaling of raster graphics. As can be seen in Chapter 4, these features come practically for free, as they are already implemented in modern graphics hardware.

3.4.5 Complexity

It is interesting to note that Inductive Rotation patterns grow exponentially from iteration to iteration. Implementations that use the approach of recursive duplication that has been outlined in Section 3.1 therefore always have exponential run-time complexity for generating the patterns, and possibly also exponential memory consumption, if the generated data is being stored in memory. The implementation created during the course of this thesis is no exception.

For example, the approach of storing and processing prototile references instead of directly manipulating pixel data will reduce the run-time or memory requirements by a constant factor only, leaving algorithmic complexity unchanged. One possibility of improving on the complexity would be to render the result of one of the later iterations, e.g., $T_{5,0}$ as an image and using this image as prototile again, starting at iteration 0 and effectively eliminating the exponential growth of the pattern. This, however, would result in a lower image resolution for the original prototile under the assumption of a fixed resolution for the prototile image. Considering the requirement of interactively zooming into arbitrary portions of the generated pattern (see Section 3.3), the implementation does not include such optimizations.

Currently there are no known approaches that achieve the same results without exponential growth of pattern data stored in memory. Mathematical analysis could reveal algorithms that do not rely on previous iterations to calculate parts of the pattern, but that is only a theoretical possibility as of now.

For the 3-way method, the number of prototiles quadruples with each iteration. After the first iteration, the number of prototiles changes from 1 to 4, in the second iteration, the resulting 4 prototiles get copied and rotated 3 times, making a total of 16 prototiles, and so on. The complexity is therefore $\mathcal{O}(4^n)$. Likewise, the complexity of the 5-way method is $\mathcal{O}(6^n)$, and $\mathcal{O}(3^n)$ for the 2-way method, with n being the number of iterations.

This is unfortunate, as it would be interesting to analyze the results of even higher iteration numbers. As of now it seems unlikely being able to manage iteration counts as low as $n = 20$ (see also benchmark tables in Section 4.10). This poses a limit on the analysis of the patterns generated by Inductive Rotation. The tool developed in the course of this thesis has primarily artistic purposes and hence this is hardly a problem.

CHAPTER 4

Implementation

The implementation of the Irrational Image Generator serves multiple purposes. One important part is the fulfillment of the requirements discussed earlier in Section 3.3. The resulting tool is not only for research purposes, but will also be used extensively by artist Hofstetter Kurt to experiment with Inductive Rotation patterns.

This solution can also be seen as a reference implementation for generating IR patterns. The approach is relatively new, and apart from one exception that is very basic in functionality (see Section 4.1), there are no other other implementations. This implementation could then be used in a scientific context, too. Should there be more research work on Inductive Rotation, it could be possible to build on the results presented in this thesis.

This chapter will describe in detail how the requirements presented in Section 3.3 were met and how the implementation works. The first section is a short overview of an already existing implementation that has several shortcomings and will be superseded by the Irrational Image Generator.

The programming environment and framework chosen for the implementation will be described in Section 4.2. In Section 4.3, the iterative development process that led to the final implementation will be outlined. The rest of this chapter will go into detail describing the implementation itself. Two different Inductive Rotation pattern generators were implemented: The grid-based generator aligns the generated patterns to a regular grid, while the sprite-based generator does not do this, but has a simpler implementation. “Sprite” in this context refers to a single two-dimensional image that is part of a bigger scene.

4.1 Prior Approach

A program similar to the Irrational Image Generator has already been developed during a practical course at the Institute of Computer Graphics and Algorithms, Vienna University of Technology. It is based on C# and uses the WPF framework [16]. However, this tool uses a very simple implementation that is solely based on bitmap operations. Apart from the disadvantages in respect to memory consumption and performance — as mentioned in Section 3.4, the program exhausts memory limits after only a few iterations — the resulting images of the 5-way and 2-way methods are lacking in quality. For any bitmap-based rotation of an angle that is not a multiple of 90 degrees there is a rasterization error, as rotated pixels cannot be matched 1:1 on the bitmap raster. If only bitmap operations are used, the result of subsequent iterations depends on the previous iteration's pixel data, which is already prone to rasterization errors. Consequently, the errors will accumulate for subsequent iterations, even when anti-aliasing is used.

Since there is much room for improvement, and the implementation of the Irrational Image Generator differs considerably from a simple bitmap-based approach, it was decided to start from scratch, using different frameworks and technologies.

However, since the same goals with respect to usability apply, and since the artist already has been using the program, it makes sense to look at the user interface and try to make the new tool's control elements to closely resemble those of the precursor. Figure 4.1 gives an overview of the user interface.

4.2 Technology

Even if the generated images are two-dimensional in nature, an implementation that is not entirely bitmap-based can highly benefit from fast polygon rasterization. It is also desirable to have alpha-transparency, i.e., for prototiles that are not fully opaque, as well as scaling operations for zooming in and out of the rendered image.

For this reasons, 3D hardware acceleration is used. The image is constructed out of texture-filled polygons where the source for the texture data is the prototile in bitmap format. This approach has several advantages:

- It is easy to scale, translate and rotate the generated image.
- Alpha blending is already implemented on the graphics hardware.
- It is possible to keep the polygon data when changing the prototile texture.
- Performance is better than with software rendering. Higher frame rates are possible.

For the API, OpenGL was chosen over DirectX. Subjectively, the API is simpler with OpenGL, and it is available on more platforms.

While the precursor was written in C#, C++ was used for implementing the Irrational Image Generator. C++ is available on many platforms. In contrast to higher-level languages like C# or Java, it does not come with a default user interface toolkit. While the earlier prototypes used the GLFW library [12] for creating the OpenGL context and handling key presses, the final implementation of the Irrational Image Generator uses QT 4.8 [35] to handle the user interface. QT has the added benefit of providing file

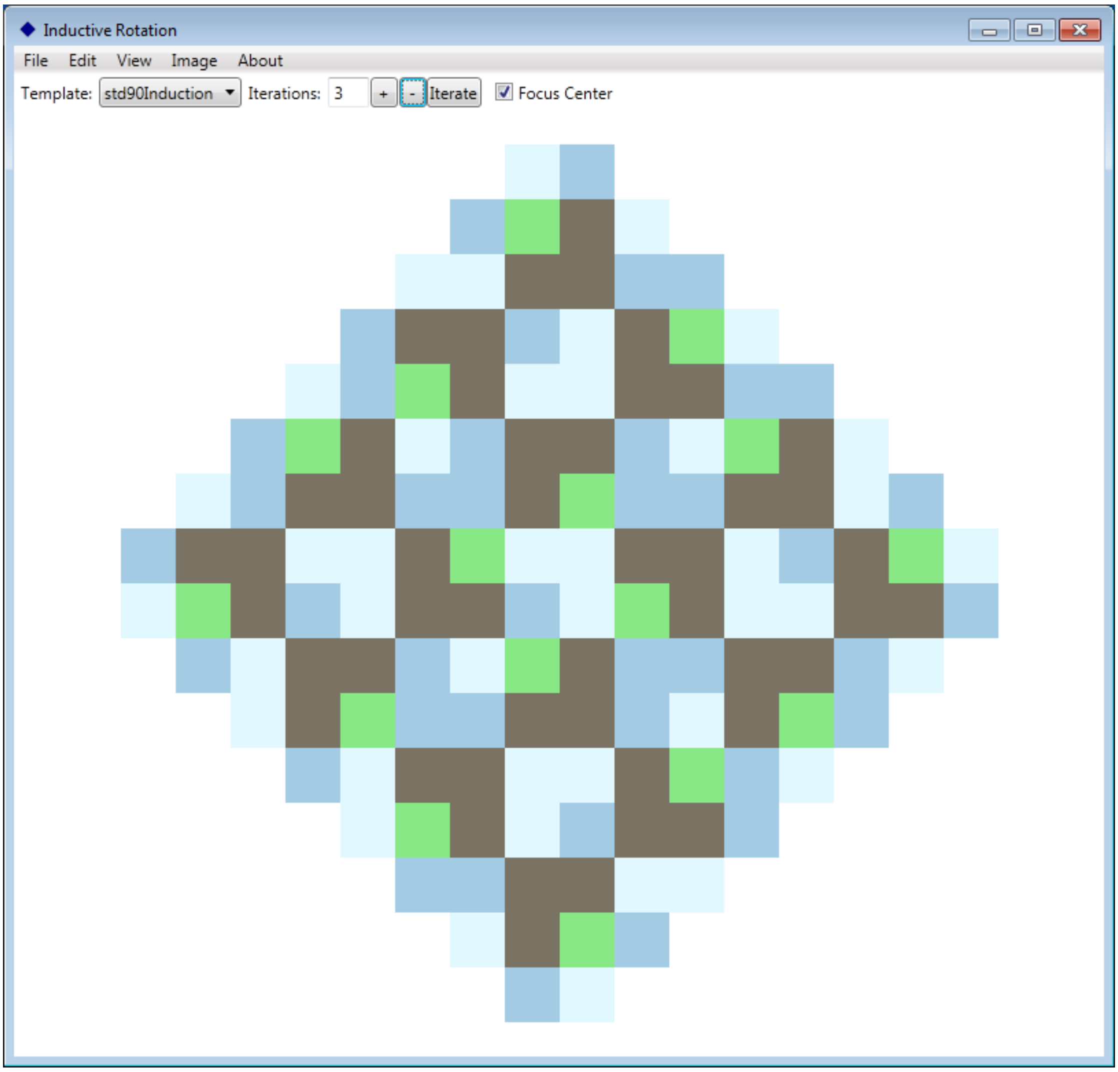


Figure 4.1: Inductive Rotation tool developed by Onur Dogangönül in 2010.

dialogs, clipboard handling and creation of an OpenGL context on multiple platforms, including Linux, Mac OS and Windows.

To avoid confusion, it should be noted at this point that the implementation uses a coordinate system where the y axis goes from top to bottom.

4.3 Development Process

To get an overview of the results as early as possible, an iterative development process was chosen. For the first few iterations, only prototypes with a minimal user interface were implemented, showing key functionalities of the Irrational Image Generator. The classes and algorithms could then be either re-used, adapted or discarded for the final implementation.

After each iteration, the resulting implementation was evaluated in a human-centered approach. Hofstetter Kurt has tested each prototype as well as the final implementation and given his opinion on how the results could be improved from his perspective.

The first prototype **irproto1** only implements the 3-way Inductive Rotation, using a grid of half-squares (see Section 4.4 for details) and displays the resulting image. The maximum size of the grid is limited by a constant value in the code and not by memory constraints, to make the implementation simpler. This way, it has been possible to evaluate the generation algorithms without having to create dynamic data structures. It is already possible to scale and translate the pattern using keyboard shortcuts.

The second prototype **irproto2** builds on the results of the first prototype and adds dynamic data structures. With that in mind, the size of the generated image is only limited by memory and run-time constraints. The generation algorithm is still grid-based, but there is an additional triangle grid to enable support for the 2-way and 5-way Inductive Rotation. As with the first prototype, scaling and translating is possible using keyboard shortcuts. During evaluation of the second prototype it became apparent that the grid-based approach has some drawbacks when using certain types of images as prototiles (see Section 4.4.7 for details). This and the fact that the keyboard-based interaction was not optimal either, led to the creation of a third prototype.

The third prototype **irproto3** uses a different rendering method. Instead of splitting the prototile into either squares or regular triangles depending on the type of Inductive Rotation and handling overlaps based on grid cells, this prototype renders the image as a composition of z-ordered rectangles, each rectangle being a copy of the prototile (see Section 4.5 for details). This approach works for all three Inductive Rotation methods. Additionally, it is possible to translate and zoom the image using the mouse.

While **irproto3** has some advantages over its predecessors with respect to rendering quality, handling of transparency and memory consumption, the generated pattern is no longer assigned to a grid. This makes it harder to group the pattern into logical layers, for example to arrange overlapping regions by their overlap depth as was mentioned in Section 3.4.3. In conclusion, both rendering methods have their advantages and disadvantages. In the final implementation, the behavior of **irproto3** was made the default, while the grid-based rendering method of **irproto2** is still included as an option.

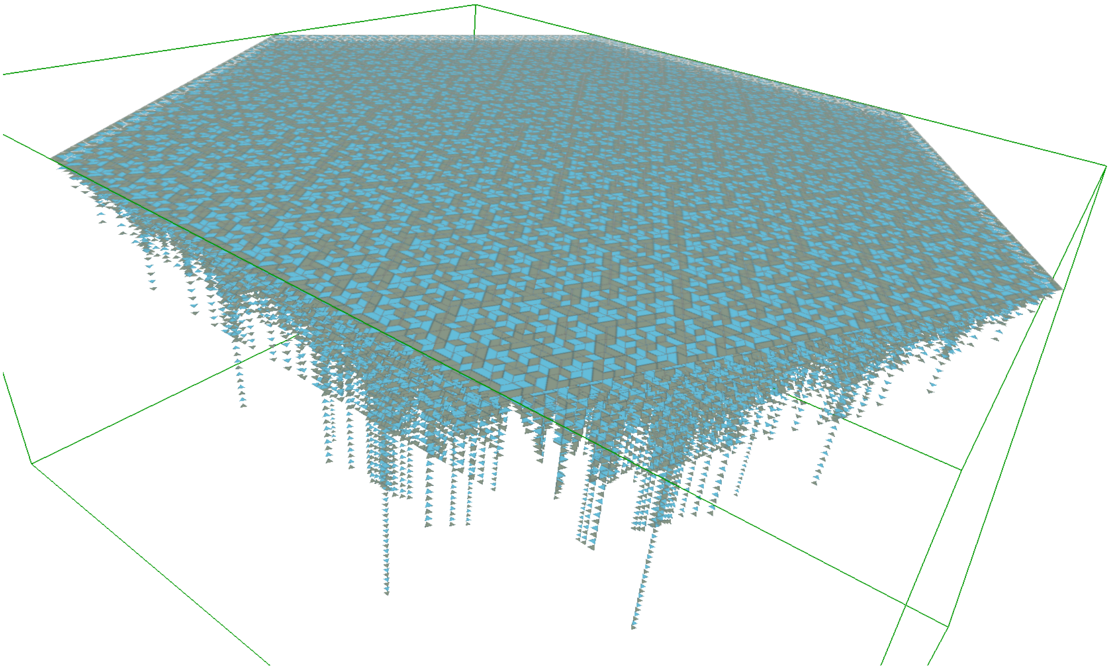


Figure 4.2: Rendering of an X3D file produced by the **irgen** tool. The image was rendered using view3dscene [43] and shows a 3D representation of a 5-way IR pattern where the z-coordinate of overlapping regions corresponds to their overlap depth.

In an attempt to analyze the generated patterns and create more interesting visualizations, a fourth prototype, **irgen** was developed. This prototype consists of the pattern generators of **irproto2** and **irproto3**, and can output the resulting patterns either in a custom binary data format or as polygon meshes in X3D XML format [46]. This tool enables the user to view the generated patterns using different programs, since the X3D XML format is a widely accepted standard and can be opened by 3D modeling tools such as Blender or Autodesk Maya. Figure 4.2 shows a rendering of a pattern generated by **irgen** and exported as X3D.

The final program, dubbed “Irrational Image Generator”, then builds on the implementations of **irproto2**, **irproto3** and **irgen** to generate the image. It has an intuitive graphical user interface and additional features such as the ability to switch to full-screen mode and saving a high-resolution image of the current viewport. This program will be used by the artist in his endeavors to realize the full artistic potential of the Inductive Rotation.

4.4 Grid-Based Implementation

The grid-based generator splits the original prototile into several sub-tiles depending on the method used and then generates the image using a regular grid of sub-tiles. As has already been noted in Section 3.4.3, this regular grid is different for each method. The 3-way Inductive Rotation images can be aligned on a regular square grid, while both the 2-way and 5-way Inductive Rotation images can be aligned on a regular triangle grid.

For the implementation, data representations for all three methods are defined, based on a dynamically expanding two-dimensional array of integers. While the regular square grid for the 3-way Inductive Rotation can be mapped to a two-dimensional array in a straightforward manner, some conventions have to be defined for the regular triangle grid used in the 2-way and 5-way Inductive Rotations so that it can be represented by a 2D array.

The generation algorithm implemented follows the "Copy and Rotate Around Pivot" concept described in Section 3.4.2. This algorithm is adapted to the grid by implementing the **pivot**, **rotate-copy** and **merge** operations for each method, using the 2D array as a data structure.

4.4.1 Dynamically Expanding Data Structure

In order to make the grid-based generation work without introducing artificial constraints on the size of the generated patterns, a dynamically expanding data structure for the regular grid is needed.

All of the grid-based generation algorithms work on a two-dimensional array. Individual integer values can be accessed by a given x- and y-offset. Ideally, this two-dimensional array would be infinitely large, allowing both positive and negative offsets. This would allow to implement the generation algorithms without having to keep memory constraints and allocation/deallocation in mind.

For this, a separate class with the name *Dynamic2DField* has been created. It provides two operations:

get(x,y) returns the integer stored at the given **x** and **y** offset. Both **x** and **y** are integer values that can also be negative. In case the integer at the given position is not set, a default value is returned.

set(x,y,value) sets the integer at the given **x** and **y** offset to **value**.

A default value is needed for all instances where **get** is called before a value has been stored at the given offset through the **set** operation.

Since both **x** and **y** can be very large values, it is critical to only allocate memory that is needed. This can be achieved by using a sparse array. A simple implementation of a two-dimensional sparse array would store values as a linked list of triples of the form (x, y, v) and return a null value in case there is no triple with the requested x, y. For the purpose of Inductive Rotation, however, this simple implementation would not be a good idea regarding performance. Looking at the nature of the generated images, it can be assumed that the data is structured in the form of a coherent, expanding block. Furthermore, there is a very large number of triples and a large quantity of data transfers for each iteration, so linear access time is not good enough.

The data structure chosen for the *Dynamic2DField* consists of a sparse two-dimensional array of fixed-size data blocks. Each of these data blocks represents a contiguous two-dimensional array of size $n \times n$. The value of n is constant for each instance of *Dynamic2DField* and is usually a power of two. In order to achieve better access time than with a linked list, the sparse array containing the data blocks was implemented as a dictionary using the *map* container from the C++ Standard Template Library (STL). This allows for near constant-time dictionary lookups, with an algorithmic complexity of $\mathcal{O}(\log a)$ where a is the number of data blocks stored in the dictionary. For more details on the STL's *map* container, see Stroustrup [41].

To avoid using nested *map* containers and make lookups more efficient, the dictionary's key is a 32-bit number in the form of $2^{16}x_b + y_b$ where x_b and y_b are 16-bit signed block indices. This restricts the block indices to a range of $(-32768, 32767)$, allowing for a maximum number of 2^{32} blocks. Considering that depending on the block size a single block can take up several megabytes of space, this is not a serious limitation. As an additional optimization, the *Dynamic2DField* class can cache the last dictionary lookup, so for two or more subsequent **get/set** operations that access the same block, there is only one dictionary lookup.

For *Dynamic2DField* to access a single value at the offset $(\mathbf{x}, \mathbf{y})$ for reading or writing, the following steps have to be executed:

1. Determine the dictionary key to the data block containing the value. This is achieved by the aforementioned formula $k = 2^{16}x_b + y_b$, where $x_b = \frac{x}{n}$, and $y_b = \frac{y}{n}$.

2. Try to look up the data block in the dictionary using k. If the block does not exist in case of a **get** operation, return the default value and terminate. In case of a **set** operation, create a new data block of size $n \times n$, initialize it using the default value and store it in the dictionary with key k.

3. Determine the offset to the value inside the data block by using the following formula: $o = mod(y, n)n + mod(x, n)$, where n is the block size and $mod(a, b)$ is the "modulo" operation, i.e., the remainder of the integer division of a by b.

4. Access the value at $b + ov$ where b is the starting address of the data block and v is the size of one value.

See Figure 4.3 for an example of the lookup procedure.

```
retrieve value at (522,-213), block size n=256:

block id:  522/256 = 2 (0x0002), -213/256 = -1 (0xffff)
                                                              k = 0x0002ffff
block offset: 522 mod 256 = 10, -213 mod 256 = 43
                                                            o = 10 + 43*256 = 11018

                                                         return D[0x0002ffff][11018]
```

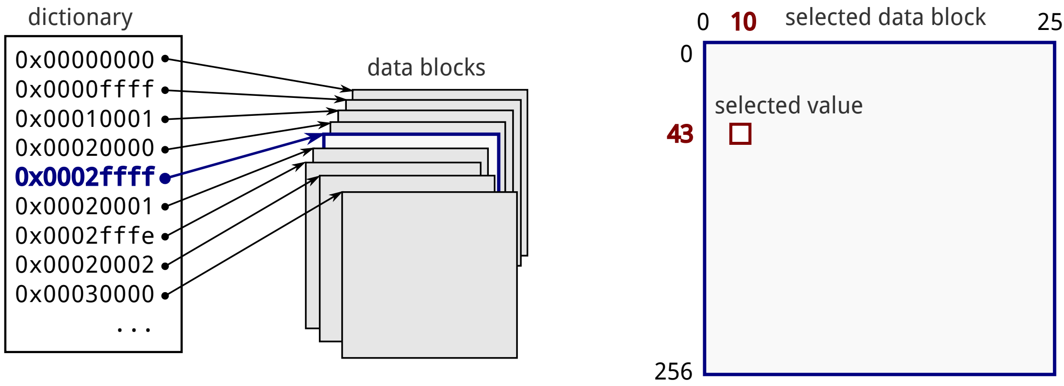


Figure 4.3: Example for the lookup of a single value using *Dynamic2DField*.

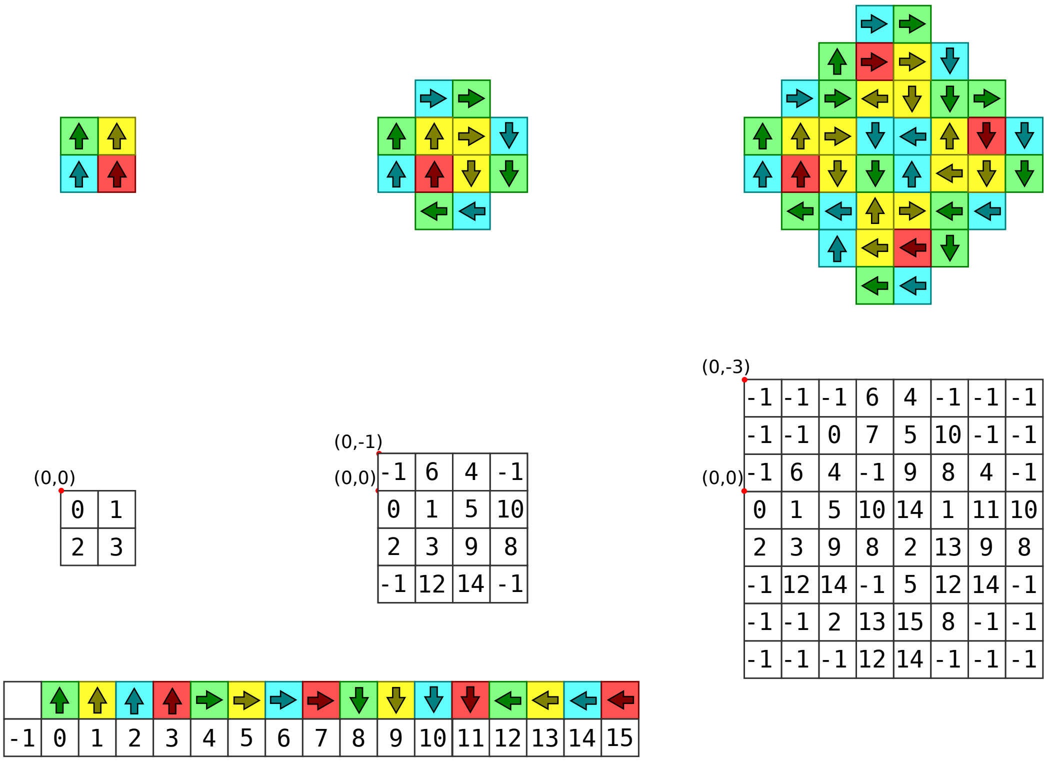
4.4.2 3-way Grid-Based Inductive Rotation

For the 3-way Inductive Rotation, which is based on 90-degree rotations, it suffices to split the prototile, which is a square, into four sub-squares and define a two-dimensional square grid where each cell is exactly the size of one sub-square. See also Figure 3.10 in Section 3.4.3.

The mapping of the square grid to an array is straightforward. The only challenge is defining a data representation for the generated image.

A data representation that translates each sub-square to a number was chosen. This makes it possible to store the generated image as a two-dimensional array of integers. Each of the four sub-squares of the original prototile can be rotated by 90 degrees repeatedly, resulting in additional three rotated states per sub-square: 90, 180, and 270 degrees. This makes a total of $4+3\cdot4 = 16$ different grid cells. Which number represents which grid cell is purely defined by convention. The sub-squares of the original, i.e., unrotated prototile, are numbered from 0 to 3, left to right and top to bottom. For each 90-degree rotation a value of 4 is added. The default value for an empty sub-square is -1 (see Figure 4.4).

As can also be seen in Figure 4.4, all distances in the generated image translate



0	1
2	3

-1	6	4	-1
0	1	5	10
2	3	9	8
-1	12	14	-1

-1	-1	-1	6	4	-1	-1	-1
-1	-1	0	7	5	10	-1	-1
-1	6	4	-1	9	8	4	-1
0	1	5	10	14	1	11	10
2	3	9	8	2	13	9	8
-1	12	14	-1	5	12	14	-1
-1	-1	2	13	15	8	-1	-1
-1	-1	-1	12	14	-1	-1	-1

-1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Figure 4.4: Data representation for the 3-way grid-based Inductive Rotation

linearly to array offsets of the data representation. This makes it possible to implement the generation algorithms without having to resort to additional mapping conventions. As will be explained later, this is not the case for the 2-way and 5-way methods.

Before starting the first iteration, the prototile $T_{0,0}$ has to be placed in the grid. There is the convention that the pattern always starts at position $(0, 0)$ in the array. The array is initialized with the sub-squares 0, 1, 2 and 3 at positions $(0, 0)$, $(0, 1)$, $(1, 0)$ and $(1, 1)$, respectively. The fact that the pattern grows in both vertical directions is not a problem here, negative array indices are explicitly allowed in the underlying data structure of the array.

The **pivot** operation is analogous to the description of the Inductive Rotation methods presented in Section 3.1. The pivot point is always at the vertical center of the right edge of the generated pattern in the 3-way case. Since investigation revealed that the pattern doubles in width for each iteration, and the vertical center always stays the same, this can be implemented as $f_{pivot} = (2^n, 1)$. Since the array is a representation of a square grid, any offset does not map to a point, but to a square region. For the rotation around a pivot point, however, the offset is treated as the upper-left point of the square in question.

The **rotate-copy** operation involves making a copy of the entire array that is rotated

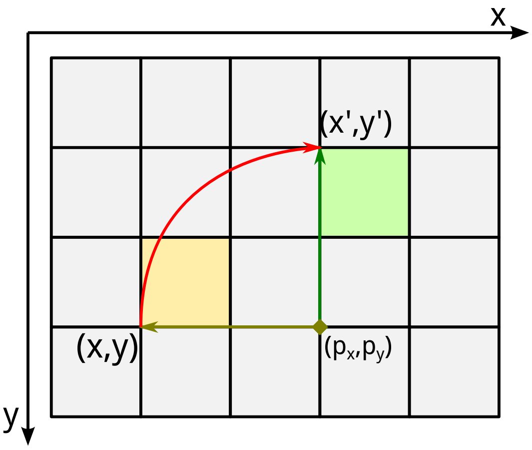


Figure 4.5: Rotating a grid cell by 90 degrees around pivot point (p_x, p_y). Note that while the offset (x, y) points to the lower-left corner of the original grid cell (yellow), the new offset (x', y') points to the upper-left corner of the rotated grid cell (green).

by exactly 90 degrees around the pivot point determined by the **pivot** operation. Making a copy is as easy as copying the contents of the array to a new memory location. Rotating around the pivot point involves some math, however. The rotated point (x', y') resulting from the rotation of 2D point (x, y) around the point of origin $(0, 0)$ by angle α clockwise can be determined by using the following formula:

$x' = x \cdot \cos\alpha - y \cdot \sin\alpha$
$y' = x \cdot \sin\alpha - y \cdot \cos\alpha$

For rotating around the pivot (p_x, p_y) as is needed for Inductive Rotation, the point (x, y) needs to be translated to the origin first, a reverse translation has to be done after rotation. This results in the following:

$x' = (x - p_x) \cdot \cos\alpha - (y - p_y) \cdot \sin\alpha + p_x$
$y' = (x - p_x) \cdot \sin\alpha - (y - p_y) \cdot \cos\alpha + p_y$

In the case of the 3-way Inductive Rotation, α is fixed at 90 degrees, and $\sin\alpha = 1$ and $\cos\alpha = 0$ can be inserted into the equation, yielding the following formula:

$x' = -(y - p_y) + p_x = -y + p_x + p_y$
$y' = (x - p_x) + p_y = x - p_x + p_y$

Since the array offsets for x, y, p_x and p_y by convention refer to the upper left corner of the square represented by the array cell, one additional translation has to be added so that the upper-left coordinate of the target cell is obtained. A clockwise 90-degree rotation changes the upper-left point into the upper-right point. So $(y + 1)$ instead of y can be used for the rotation to refer to the lower-left coordinate instead (see Figure 4.5. After rotation, the upper-left coordinate of the target cell will be obtained:

$x' = -(y + 1) + p_x + p_y$
$y' = x - p_x + p_y$

An alternative would be to refer to the center points of grid cells instead, by adding an offset of $(0.5, 0.5)$. This would solve the problem of the changing reference point since the center point does not change during rotation of the square.

Since the problem of determining the offset for the rotated grid cell is solved, it is now possible to implement the **rotate-copy** operation:

- Create a new, empty array.
- Iterate over every cell with offset (x, y) and value v in the original array.
    - Determine the rotated offset (x', y') as well as the rotated value $v' = mod(v + 4, 16)$. It has been mentioned before that adding 4 to the value represents a rotation by 90 degrees. By using the modulo operation it is ensured that the rotation "wraps around" at 360 degrees.
    - Set the value at offset (x', y') to v' in the new array.

For the **merge** operation, one array has to be merged into the other one, omitting overlaps. By the rules of Inductive Rotation, the newly created, rotated image is always overlapped, or lies behind the existing image. This is as simple as iterating over each sub-square of the newly created array and setting the same value at the same position in the original array if it is empty. In case there is already a value other than the default value of -1 present, nothing is done. It is also possible to create multiple layers to store hidden sub-squares in the data structure; this is covered in Section 4.4.5.

With all three operations implemented, the program then follows the algorithm outlined in Section 3.4.2 to populate the array. For rendering, the data then needs to be converted to a polygon mesh that can be displayed on the screen using the OpenGL API. For more details on the rendering part, see Section 4.4.6.

4.4.3 5-Way Grid-Based Inductive Rotation

The prototile for the 5-way Inductive Rotation is a regular hexagon and can be split into six equilateral triangles. It is possible to align these triangles to a regular triangle grid for all iterations, as can be seen in Figure 3.10.

To achieve a grid-based implementation similar to the one for the 3-way Inductive Rotation, it is necessary to map the triangles of the grid to offsets in a two-dimensional array. This is not as straightforward as with a square grid, as the triangle grid is not axis-aligned and therefore the adjacency is different from that of a two-dimensional array. The definition of left/right or up/down neighbors is ambiguous for edges that are not parallel to either the x- or y-axis.

For the implementation, a simple numbering that goes left-to-right and top-to-bottom was chosen. It has to be ensured that triangles on the same row have the same y-offset in the array (see Figure 4.6). A similar approach for hexagonal meshes has already been shown by Theußl et al. [42]. For mapping array offsets to grid points, the upper-left coordinate of the selected triangle is chosen. This leaves some ambiguity for converting grid points to array offsets, since each grid point is always the upper-left coordinate for two triangles. Another problem is that with regular triangles, the coordinates of grid points usually are no integers since the y-coordinate is increased by the height of a triangle, which is $h = \frac{\sqrt{3}}{2}a$ if a is the length of a triangle edge.

Consequently, there is a need for converting between array offsets and grid coordinates. Rotations use grid coordinates and involve some inaccuracies due to the way floating-point operations work. This problems and ambiguities need to be considered.

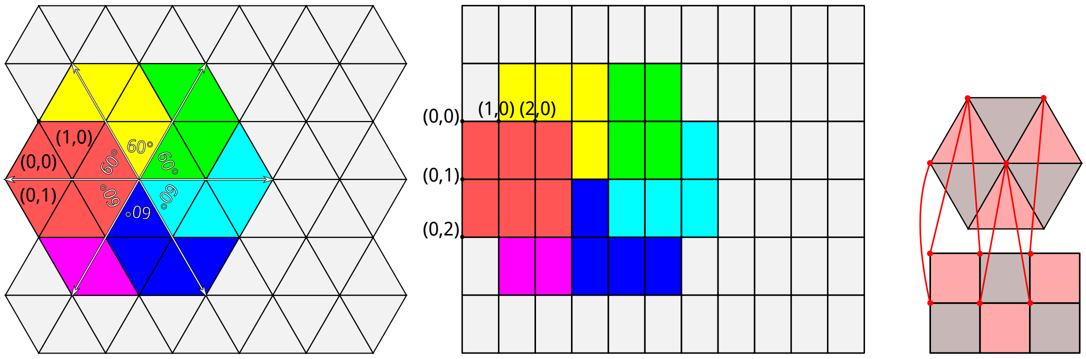


Figure 4.6: Array representation for the regular triangle grid needed by both the 5-way and 2-way grid-based Inductive Rotation implementation.

The data representation works like with the 3-way method, numbering the sub-triangles of the hexagonal prototile left-to-right and top-to-bottom from 0 to 5. For a 60-degree rotation, 6 is added and the value wraps around 36. By convention, the pattern starts at position $(0, 0)$ in the array, so it is initialized with sub-triangles 0, 1, 2, 3, 4, and 5 at positions $(0, 0)$, $(0, 1)$, $(0, 2)$, $(1, 0)$, $(1, 1)$ and $(1, 2)$, respectively.

The **pivot** operation works similar to the one of the 3-way method, since the pattern width also doubles with each iteration in the 5-way method. This can be implemented as $f_{pivot} = (2^n, 1)$. Note that the pivot point is stored as an array offset. The pivot point is ambiguous as there are always two array offsets for each grid point. This is resolved by always using the array offset with the smallest x-coordinate as pivot point.

The **rotate-copy** operation works analogous to the one for the 3-way method, with the exception that a conversion from array offsets to grid coordinates and vice-versa is needed. Like with the 3-way method, grid points, i.e., points where grid lines intersect, are mapped to array offsets. In the case of the regular triangle grid, an ambiguity arises, as each grid point is shared between six triangles.

For resolving the ambiguity resulting from mapping one grid point to two different array offsets it is important to note that there are two types of triangles in the grid with respect to orientation. When looking at their height vectors originating on edges parallel to the x-axis, it is possible to distinguish between up-facing and down-facing triangles. In the infinite 2D array, the cell with offset $(0, 0)$ is mapped to an up-facing triangle, like in Figure 4.6. It can be observed that the triangles are placed in a checkerboard pattern in the array. Increasing either the x- or y-offset alternates the triangle type.

Following this convention, it can be determined which type of triangle is selected for any array offset (x, y):

- If both x and y are even, or both x and y are odd, an up-facing triangle is selected.
- If x is even and y is odd, or x is odd and y is even, a down-facing triangle is selected.

Based on this logic, it is also possible to map from a grid point to an array offset. For that, both the x- and y-coordinate as well as the orientation of the selected triangle have to be known, i.e., either up-facing or down-facing. It has been established before that each triangle can be selected by specifying its top-left coordinate. For simplicity, $a = 1$ is always specified as the length of a triangle edge. It follows that the height of a triangle equals to $h = \frac{\sqrt{3}}{2}$. The up-facing triangle at array offset $(0, 0)$ has its upper point located at $(0.5, 0)$, in grid coordinates.

To map a grid coordinate (x_g, y_g) to an array offset (x_a, y_a), the y-offset is calculated as $y_a = y_g \frac{2}{\sqrt{3}}$, rounded to the nearest integer. Depending on the array row with offset y_a being even or odd, the x-offset is determined as follows:

- $x_a = 2x_g - 1$ for selecting a down-facing triangle in an even row, or selecting an up-facing triangle in an odd row.
- $x_a = 2x_g$ for selecting an up-facing triangle in an even row, or selecting a down-facing triangle in an odd row.

The reverse operation, i.e., mapping an array offset (x_a, y_a) to its corresponding grid point (x_g, y_g) — that is the upper-left coordinate of the corresponding triangle — does not need the triangle's orientation as an extra parameter as the mapping is unambiguous in that direction, see also Figure 4.6. Depending on the y-offset y_a being even or odd, the x-coordinate x_g is determined as follows:

- $x_g = 0.5 + \frac{x}{2}$ for y_a being even.
- $x_g = \lfloor \frac{x+1}{2} \rfloor$ for y_a being odd.

The y-offset y_g is then calculated as $y_g = \frac{\sqrt{3}}{2} y_a$.

With these mapping conventions for converting array offsets to triangle-grid intersections and vice-versa established, now the **rotate-copy** operation can be implemented analogous to the one discussed for the 3-way method:

- Create an empty array.
- Convert the current pivot point's array offset to grid coordinates (p_x, p_y).
- For each offset (x_a, y_a) in the existing array:
    - Determine the corresponding grid coordinates (x_g, y_g) as well as the orientation of the corresponding triangle.

- Rotate (x_g, y_g) around (p_x, p_y) using the already established formula with α set to 60 degrees:
  $x'_g = (x_g - p_x) \cdot \cos 60 - (y_g - p_y) \cdot \sin 60 + p_x$
  $y'_g = (x_g - p_x) \cdot \sin 60 - (y_g - p_y) \cdot \cos 60 + p_y$
- In case of rotating an up-facing triangle, add a correction factor of $(-1.0, 0)$ to the rotated point (x'_g, y'_g). This is done because a clockwise rotation of the upper point of an up-facing triangle by 60 degrees results in the upper-right corner of the rotated, now down-facing triangle. This can be seen in Figure 4.7. To comply with the convention of always addressing triangles by their upper-left coordinate, 1 is subtracted from the x-coordinate to correct this.
- Convert the point (x'_g, y'_g) back to an array offset (x'_a, y'_a). For this also the orientation is needed, which is always the opposite of the unrotated triangle's orientation for a 60-degree rotation. If the original triangle was facing up, the rotated triangle is facing down.
- Determine the value in the original array at (x_a, y_a), and add 6 for the rotation, wrapping around 36. Set this value in the newly created array at (x'_a, y'_a).

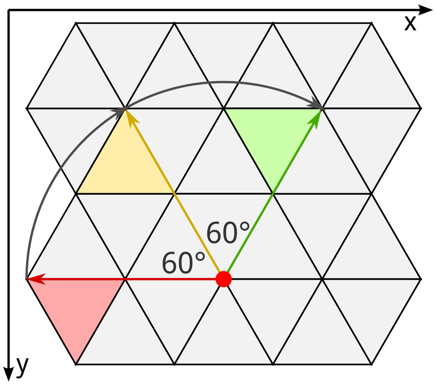


Figure 4.7: Rotating triangles in the grid clockwise by 60 degrees using the upper-left point as a reference. Notice that after rotating an up-facing triangle, the upper-right point is obtained instead. To always get the upper-left point after a rotation, which is needed for selecting the appropriate array offset, a correction is needed.

The rest of the generation algorithm, including the **merge** operation is the same as for the 3-way method. See Section 4.4.2 for details.

4.4.4 2-Way Grid-Based Inductive Rotation

In theory, the 2-way Inductive Rotation is aligned to a rhombus grid (see Figure 3.10). It is however also possible to re-use the triangle grid from the 5-way grid-based Inductive

Rotation as each rhombus can be split into two triangles. This path has been chosen to simplify the implementation, as a rhombus grid would require its own array mapping.

By re-using the triangle grid described in Section 4.4.3, the implementation stays roughly the same, with some minor differences:

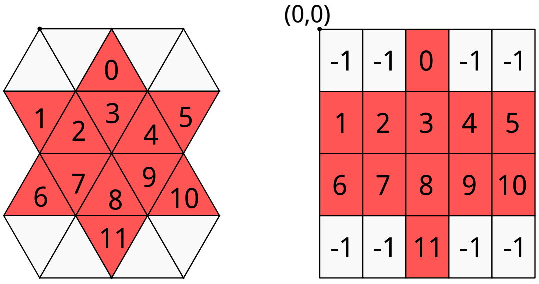
- The star-shaped prototile is split into 12 sub-triangles instead of 6 for the hexagon.
- The rotation angle is 120 degrees instead of 60, resulting in $12 \cdot 3 = 36$ different sub-triangles. For a 120-degree rotation, the orientation of the sub-triangle stays the same, but a correction factor is needed nonetheless. In the data representation, a value of 12 is added for a 120-degree rotation.
- The pivot point is determined in a different way.

Initializing the array, the 12 sub-triangles numbered 0 to 11 of the prototile are placed as shown in Figure 4.8. The pivot point is calculated as an array offset, using the following algorithm:

- Select all cells with maximum x-offset where the value $v \geq 0$.
- From these cells, select the one with the largest y-offset.

The **rotate-copy** operation works analogous to the one described in Section 4.4.4, the only difference being that the orientation of the triangles does not change during rotation. Following the convention of always selecting triangles by their upper-left coordinate, correction factors are needed as well. As can be seen in Figure 4.9, these are $(-0.5, \frac{\sqrt{3}}{2})$ and $(-1.0, 0)$ for up-facing and down-facing triangles, respectively.

The **merge** operation does not change and is in fact the same for all three grid-based Inductive Rotation implementations.



-1	-1	0	-1	-1
1	2	3	4	5
6	7	8	9	10
-1	-1	11	-1	-1

Figure 4.8: Prototile (left) for the 2-way grid-based Inductive Rotation along with its array representation (right).

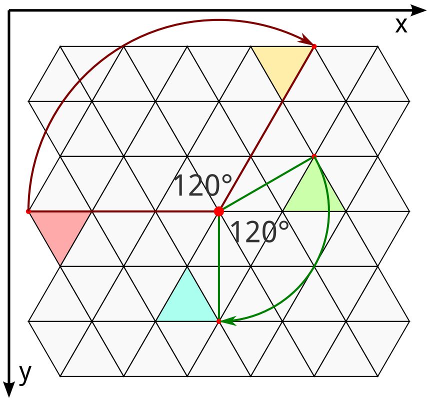


Figure 4.9: Rotating triangles in the grid clockwise by 120 degrees using the upper-left point as a reference. Notice that the orientation of the triangles does not change. When rotating an up-facing triangle, the lower-right point is obtained; when rotating a down-facing triangle, the upper-right point is obtained. Both values need to be corrected before being converted back to array offsets.

4.4.5 Three-Dimensional Data for Grid-Based Generation

For the Irrational Image Generator, it is desired to not only store the visible pattern, but to store the parts hidden by overlaps as well. A feature of the tool is to explicitly show these underlying patterns as well.

This is easily possible with the implementation already covered, but some adjustments have to be made to how the data structure works. Instead of directly using an instance of *Dynamic2DField* for the 2D array as well as the copies produced by the **copy-rotate** operations, a class called *RotationGrid* is now used, which stores a dynamically expanding array of *Dynamic2DField* instances. Each of these instances has an index, with index 0 representing the topmost plane, index 1 representing the layer containing all regions with overlap level 1, and so on. The array index therefore represents the overlap level. For simplicity, the **get** and **set** methods have been reimplemented, adding a third offset called z, which is the index of the *Dynamic2DField* instance where the value should be retrieved/stored. New instances are created on demand.

Now, the **merge** operation can be adjusted to not discard overlapping array cells, but instead store them at the first level where the corresponding array cell is empty. Both the 2-way and 3-way implementation only need two array instances, as the maximum overlap level is 1. The 5-way iteration's overlap level however doubles with each iteration. For the dynamically expanding array, the STL's *vector* class is used.

4.4.6 Grid-Based Rendering

While the generation of the Inductive Rotation pattern has been described at length in previous subsections, there is still the question on how to render the array representation.

Since the implementation uses OpenGL, it makes sense to convert the array to a polygon mesh. Depending on the type of Inductive Rotation, there are two different polygon meshes: A regular square grid for the 3-way method, and a regular triangle grid for the 5-way and 2-way methods.

In the square grid, each array cell produces 4 vertices, forming a square. With OpenGL, polygon vertices are usually specified in counter-clockwise order. Starting with the top-left coordinate, which is just the array offset (x, y), the second coordinate in counter-clockwise order then would be $(x, y + 1)$, the third one $(x + 1, y + 1)$ and the fourth one $(x + 1, y)$.

In the triangle grid, each array cell produces 3 vertices, forming a triangle. Each array offset is converted to the matching top-left point (x, y) using the already known algorithm outlined in Section 4.4.3. This is the first generated vertex. The remaining two vertices are then generated depending on whether the triangle is an up-facing or down-facing one:

- For an up-facing triangle, the remaining two vertices are generated as $(x - 0.5, y + \frac{\sqrt{3}}{2})$ and $(x + 0.5, y + \frac{\sqrt{3}}{2})$.
- For a down-facing triangle, the remaining two vertices are generated as $(x + 0.5, y + \frac{\sqrt{3}}{2})$ and $(x + 1, y + \frac{\sqrt{3}}{2})$.

This already covers the creation of the polygon mesh. The program simply iterates over all non-empty grid cells in the array and depending on the type of grid generates a square or regular triangle. Note that the polygons contained in the generated mesh still do not represent the value stored in the grid cells. Since single cells can be empty, the mesh may also have holes.

What is still missing is the texture mapping needed to display prototile images according to the values stored in the array cells. A single value stores both the offset and rotation of the prototile section that should be used for filling the polygon.

The prototile in this case is a bitmap that has been uploaded to VRAM as a 2D OpenGL texture. QT is used to decode several file formats including JPG and PNG. It is assumed that the texture fully covers the prototile. An optional aspect ratio correction that adds transparent borders to textures not matching the aspect ratio of the prototile can be enabled by the user. Considering that OpenGL texture coordinates always go from 0 to 1, this yields the coordinates specified in Figure 4.10. In this way, a set of texture coordinates can be assigned to each integer value stored in the grid. Rotation of the sub-images is done purely by changing the texture coordinates assigned to the vertices.

For the rendering in OpenGL, one array per iteration is generated, containing both the position (x, y, z) and texture coordinates (s, t) for each vertex. Rendering is only a matter of pointing the OpenGL state to that array and calling *glDrawArrays*. The user can then switch forward and backward between iterations. For example, if iteration 3 is selected by the user, the program will render the first four vertex arrays, iteration 0 being just the prototile with no rotations added. In case there is more than one layer,

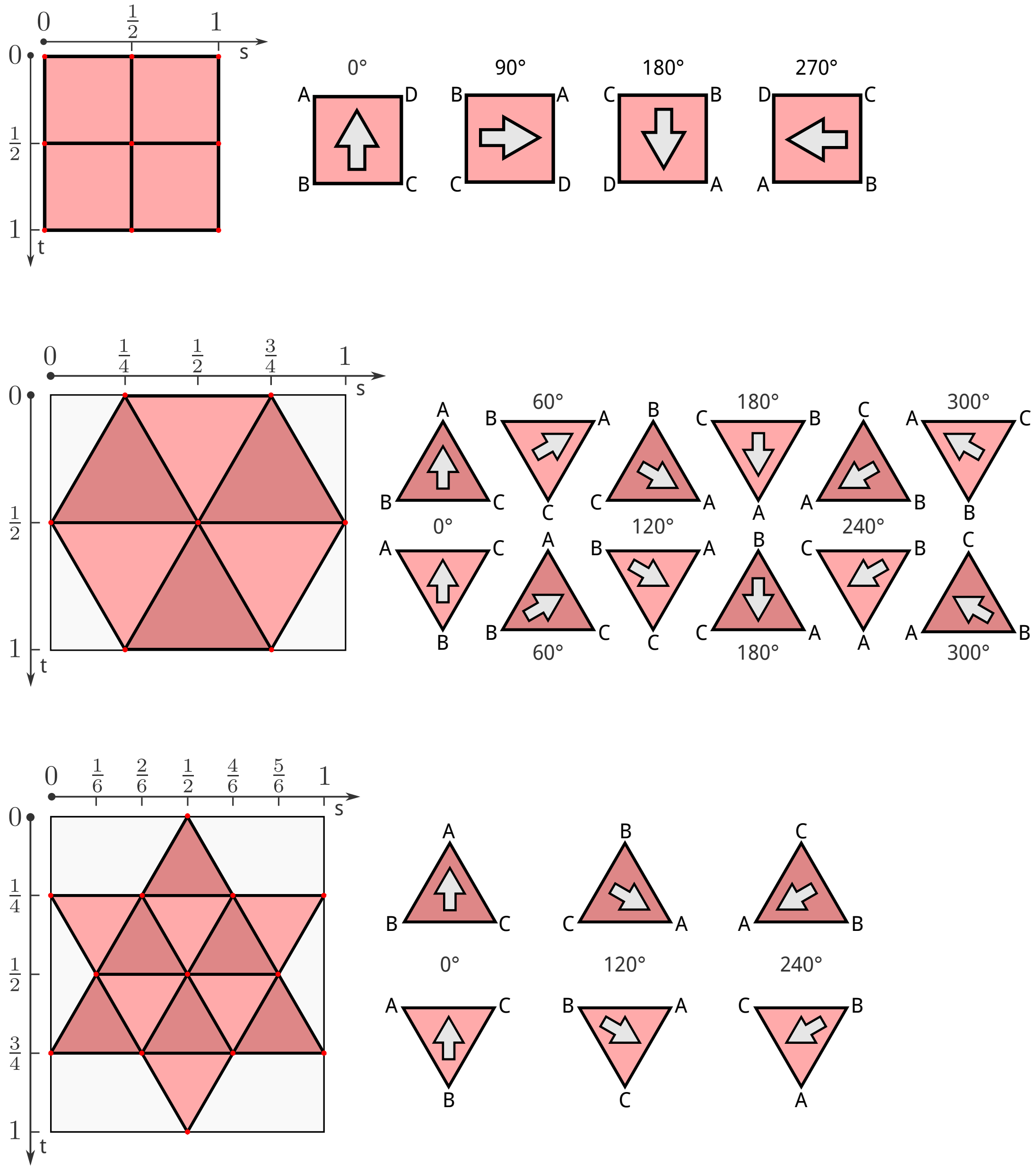


Figure 4.10: Texture coordinates for rendering of grid cells in the grid-based implementation.

the user can switch between layers, too, looking at the hidden parts of the pattern. For this, the iterations have to be split into layers, too. For example, iteration 2 of the 5-way Inductive Rotation consists of 4 layers and therefore also 4 different vertex arrays that can be cycled through.

4.4.7 Results

While the grid-based generation is sufficiently fast and supports large enough pattern sizes for artistic experimentation, the image quality for the 2-way and 5-way methods suffers from the fact that the prototile is split into sub-squares along edges that do not lie at exact pixel boundaries. For some prototiles, this will result in visible artifacts, even though the rendering is implemented correctly.

Careful evaluation has also revealed another drawback: Sometimes it is desired to use prototiles for 2-way and 5-way rotations that are not limited by the corresponding hexagon or star figure. Alignment to a grid enforces this, however. The parts outside of the hexagon or star figure are not used for texture mapping and are therefore cut off.

For these cases it is desirable to have another generation/rendering implementation that does not respect the underlying grid structure. This second method is described in Section 4.5. The tool allows the user to switch between the two methods on the fly.

4.5 Sprite-Based Implementation

The sprite-based generator creates rotated copies of the original prototile and renders the resulting scene using a determined rendering order, or z-order. In contrast to the grid-based implementation, the sprite-based implementation ignores the underlying structure of the pattern. This gives better looking results for some special cases and makes this method slightly faster and more memory-efficient, as there is no need to store a grid of sub-squares/-triangles. The drawback of this approach, however, is that any information about the underlying pattern and grid alignment is lost. If the aim is just creating images, this does not matter so much. Advanced features, like specifically displaying hidden regions, are not possible with this implementation anymore.

Since both approaches have their advantages and disadvantages, both implementations are included in the Irrational Image Generator, with the sprite-based implementation being the default one.

4.5.1 Sprite-Based Inductive Rotation

For sprite-based generation of IR patterns, it is no longer necessary to store the pattern in a two-dimensional data structure, as it is not organized in a grid. Instead, a list of triples (x, y, r) is stored, each triple representing a copy of the prototile. The x and y values give the center position of the prototile on the plane, while r is the rotation in degrees. For the implementation it is sufficient to store r in 30-degree steps, as this is the greatest common denominator of 60, 90 and 120, which are the rotation angles for the 5-way, 3-way and 2-way Inductive Rotation, respectively.

For each iteration, the existing triples are then copied, rotated by α degrees around the pivot point (x_p, y_p), and added to the end of the list. This step is repeated multiple times, the exact number depending on the IR method selected. For example, using the 3-way Inductive Rotation, the list of existing triples is copied and rotated by 90 degrees around the pivot point. This step is executed another two times, with 180 and 270 degrees, to complete the iteration.

This behavior follows the copy-rotate algorithm described in Section 3.4.2. The **copy-rotate** operation iterates through every triple of the previous iteration, rotates the center point x, y by α degrees and increases the rotation angle r by α. The **merge** operation simply adds a list of triples generated by the **copy-rotate** operation to the end of the list of existing triples. This list then already specifies the correct rendering order. If the triples are rendered back-to-front as prototiles, the image is displayed correctly.

By convention, each prototile has width $w = 2$. The height is dependent on the selected method. It is $h = 2$ for the 3-way method, $h = \sqrt{3}$ for the 5-way method and $h = \frac{4}{\sqrt{3}}$ for the 2-way method. For every method, the first prototile's center point is placed at position $(1, 1)$.

The **pivot** operation is responsible for determining the pivot point for iteration i. For the 3-way and 5-way methods this is the vertical center of the right edge of the pattern. This can be determined as $(2^i, 1)$, as each prototile has width 2 and the width of the pattern doubles with each iteration. For the 2-way method, this is more complicated, but can be implemented analogous to the grid-based method. The pivot point is always located inside the bottommost of those prototiles with maximum x coordinate, with an offset of $(\frac{2}{3}, 0)$ to the center point, assuming the prototile has width $w = 2$. To determine the pivot point at the beginning of a new iteration, the bottommost of those triples of the previous iteration with maximum x-coordinate is picked. The pivot point is then $(x + \frac{2}{3}, y)$ if (x, y) is the center position of the selected triple.

4.5.2 Sprite-Based Rendering

For OpenGL based rendering, a set of vertices and texture coordinates is generated for each triple in the sprite list and added to a corresponding vertex array. As with the grid-based rendering described in Section 4.4.6, there is one vertex array for each iteration so the switching between iterations can be handled easily.

With all three IR methods combined, there are only $\frac{360}{30} = 12$ different rotation steps, and each triple is expected to generate a textured rectangle. It makes sense to pre-calculate the vertices relative to the center point for those 12 rotation steps and store them in a lookup table instead of re-calculating them for each list triple. The aspect ratio of the rectangle is different for each of the three methods, so each method needs its own lookup table.

Since the vertices already correspond to the rotation stored in the list triple, and assuming the prototile texture fully covers each rectangle, the texture coordinates are always the same. Starting counter-clockwise from the upper-left coordinate of the unrotated rectangle, they are $(0, 0)$, $(0, 1)$, $(1, 1)$ and $(1, 0)$.

The list of rectangles has to be rendered back-to-front. To achieve this, the corresponding vertex array is generated after the end of the current iteration by iterating through the list triples in reverse order. When rendering a specific iteration > 0, the vertex arrays are rendered in reverse order, too, e.g., for rendering iteration 3, the vertex arrays 3, 2, 1 and 0 are rendered in that order, vertex array 0 being only the prototile.

Another option for ensuring correct overlaps would be generating z-coordinates accordingly and letting the GPU handle the rendering order through depth testing. However, this would only work correctly for fully opaque prototiles.

4.5.3 Results

At least for artistic purposes, the sprite-based rendering method has been proven the most useful one. The performance is slightly better and there are less rendering artifacts since only one polygon is rendered per prototile. This rendering method is the default one for the Irrational Image Generator.

The grid-based implementation can be switched on and off using a check box in the program's GUI. To make the transition between the two approaches seamless, the grid-based implementation has been adapted so that both implementations produce exactly the same image in regard to size and position. In some cases, especially when using the 3-way IR method with fully opaque prototiles, switching from grid-based to sprite-based generation yields no visible change to the rendering output. In other cases, for example when using a prototile with alpha transparency, or using an image that has non-transparent pixels outside of the regular hexagon for the 5-way IR method, there is a visible difference to the rendered output. Results can be easily compared by switching back and forth between the two implementations. Figure 4.11 shows an example where there is a clear difference between the implementations.

In case the prototile includes transparent or translucent pixels, the hidden parts show through in the sprite-based implementation, while the grid-based implementation only displays one layer at once. This is on purpose, as both behaviors could be needed during experimentation.

4.6 Class Structure

An UML class diagram for the Irrational Image Generator can be seen in Figures 4.12, and 4.13. The class structure can be divided into three parts:

- The first part consists of the grid-based implementation, including all needed data structures. While the class *Dynamic2DField* implements the dynamically allocated 2D array, the abstract *RotationGrid* class provides another abstraction where one instance of *Dynamic2DField* is used for each layer and iteration. *RotationGrid* also includes pure virtual methods that are overridden by subclasses to implement the **rotate-copy** and **pivot** operations. The **merge** operation is the same for all subclasses and is already implemented in the *RotationGrid* class. While the *QuadGrid* class inherits directly from *RotationGrid*, the *HexGrid* and *StarGrid*

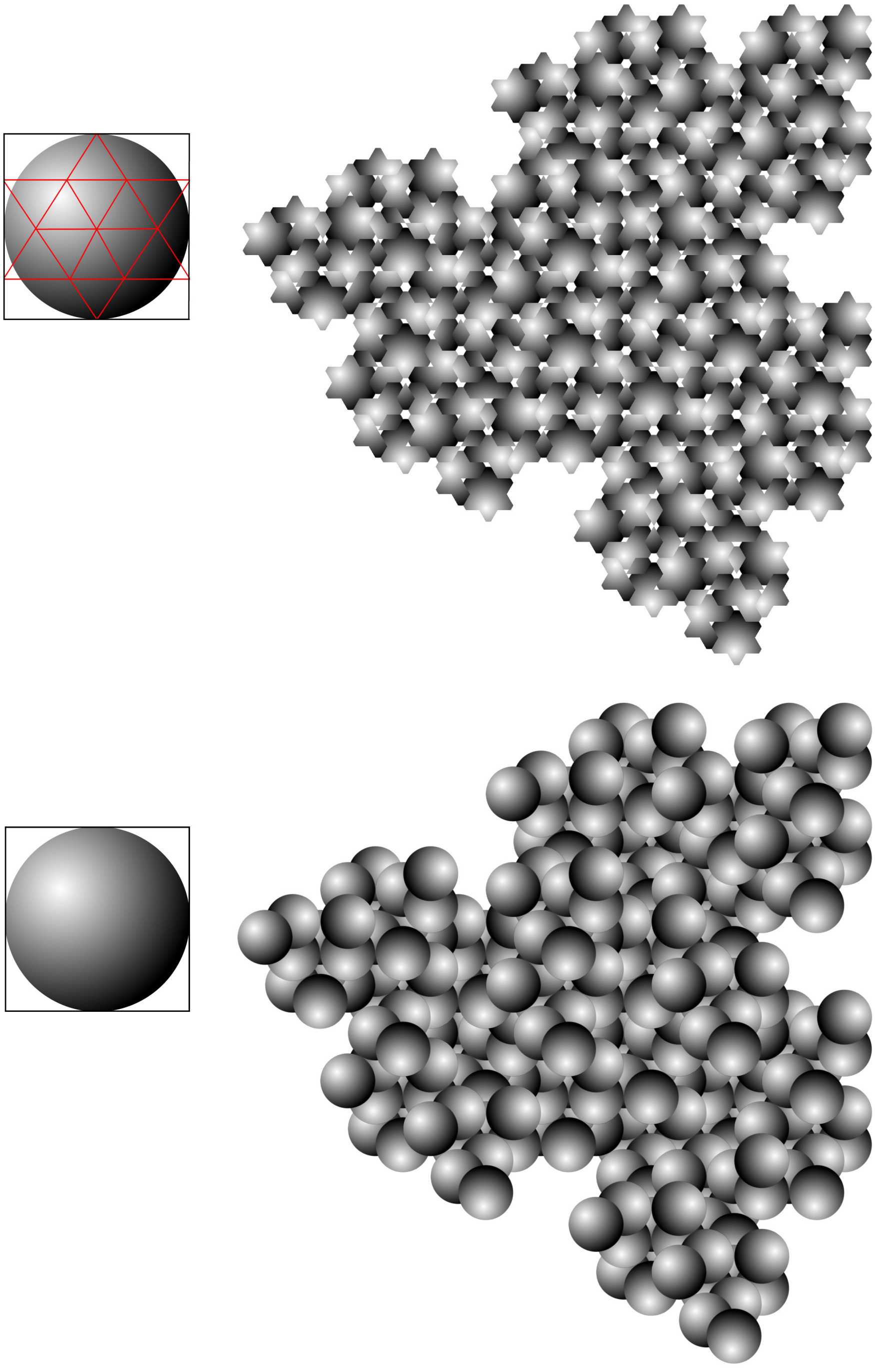


Figure 4.11: Example of how the grid-based (top) and sprite-based (bottom) implementation can produce different results. The example shows the 2-way Inductive Rotation with a prototile with alpha-transparency that is not limited to the star pattern used for the grid-based implementation.

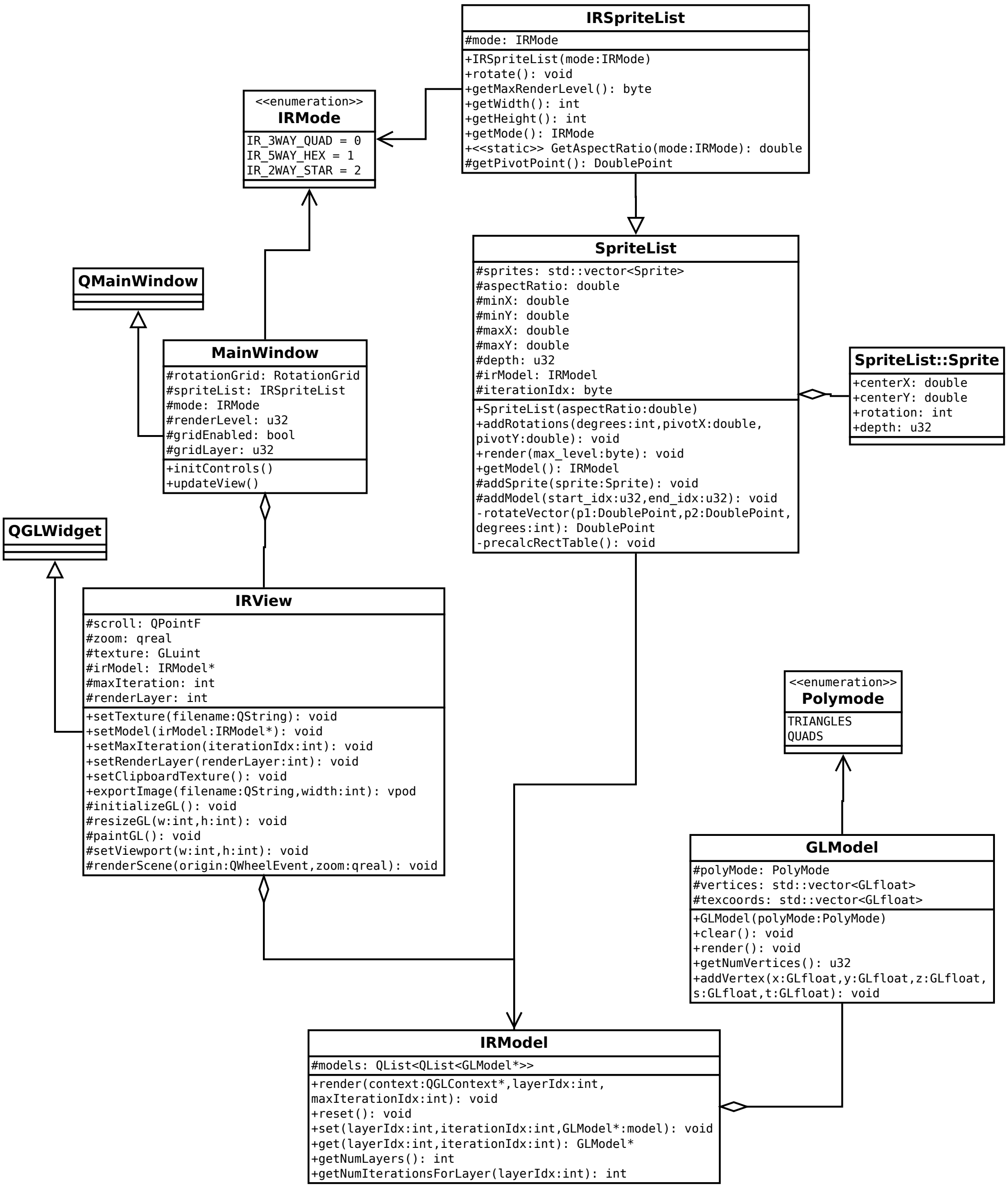


Figure 4.12: UML class diagram for the Irrational Image Generator (continued in Figure 4.13).

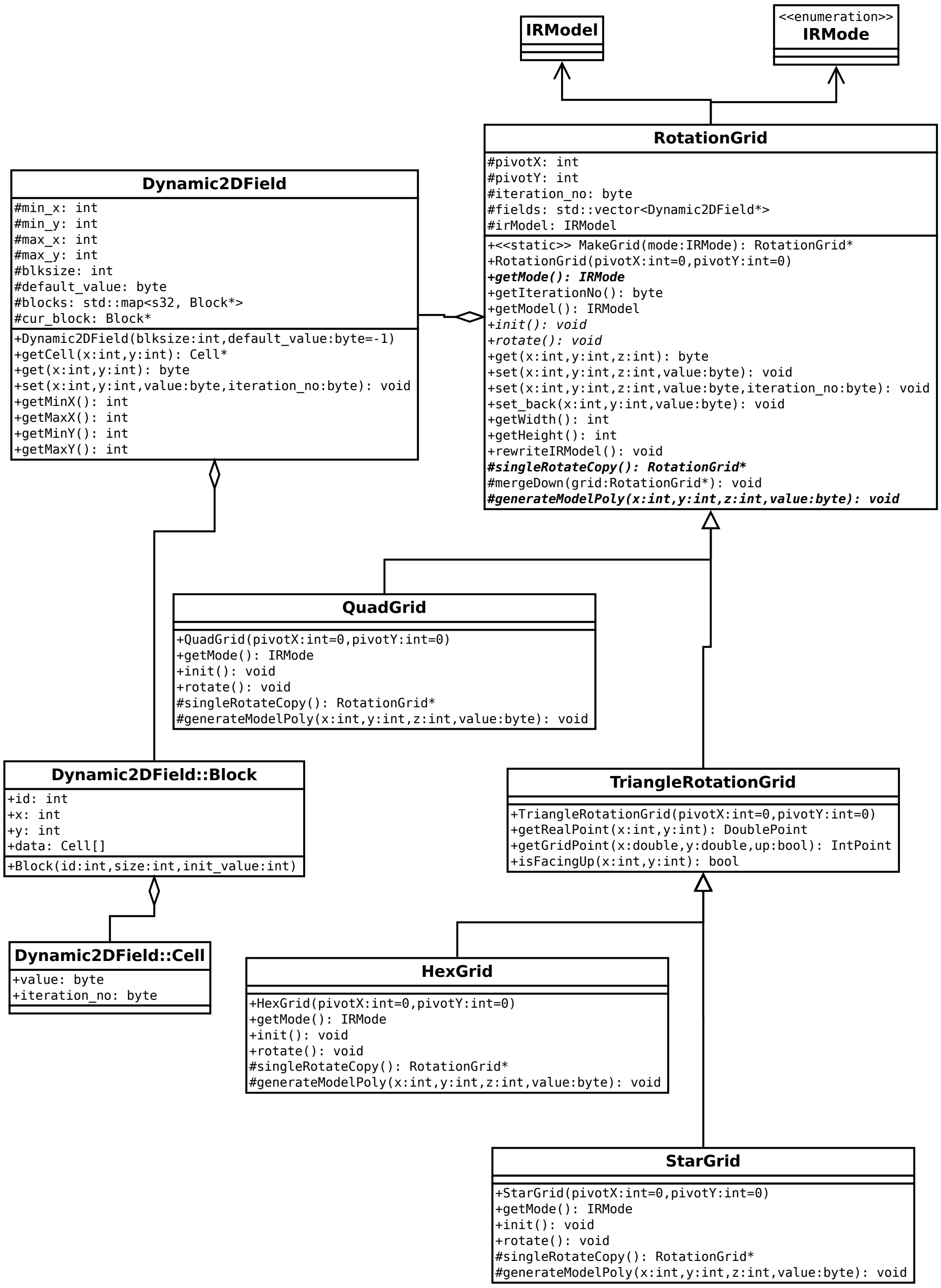


Figure 4.13: UML class diagram for the Irrational Image Generator (continued, see Figure 4.12 for the first part.

classes have some shared code that is located in the common *TriangleRotationGrid* base class which in turn inherits from *RotationGrid*.

- The sprite-based implementation consists of a more general base class *SpriteList* which implements all basic operations, but is not suited for any special IR method, e.g., has no operations to determine aspect ratio or pivot point. These operations are implemented in the *IRSpriteList* subclass for all three IR methods. Depending on a construction parameter, an instance of *IRSpriteList* uses a different lookup table and pivot formula to implement a specific IR method.
- The last part contains the GUI, using the QT framework and consisting of the *MainWindow* and *IRView* classes. *IRView* encapsulates the OpenGL context and provides a widget interface so the rendered output can co-exist with other QT elements like buttons, checkboxes, or menus. The *GLModel* class implements a simple vertex array that can be directly passed to OpenGL. The *IRModel* class, which is used by both implementations, can store and render multiple vertex arrays, grouped by both layer and iteration number. Depending on the rendering method selected by the user, either an instance of *RotationGrid*, or an instance of *IRSpriteList* is used for generating and rendering the image.

4.7 Export of Images

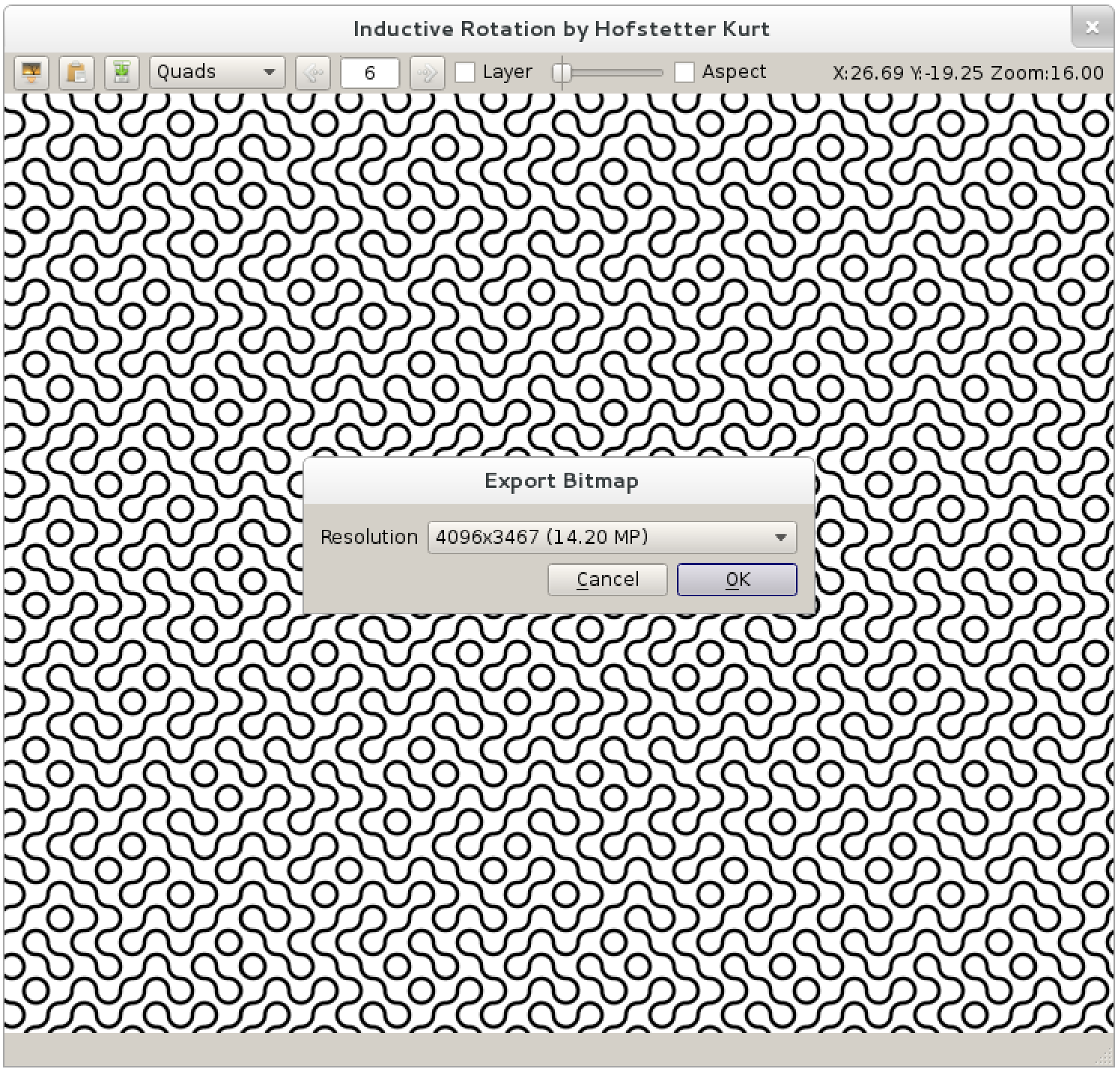
One important feature of the Irrational Image Generator is the export of the current viewport to a high-resolution compressed image file in PNG format. This is done through off-screen rendering to a texture. The texture's RGB data is then transferred to main memory and converted to the target format using the *QImage* class provided by the QT framework.

Both texture size and rendering viewport size are constrained in OpenGL. For high resolutions, the program has to render the image in multiple passes. For this, an image buffer large enough to hold the entire high-resolution image is allocated in main memory. The texture used as a render target then covers only a part of this image, and the image buffer is filled through multiple render passes with different translation offsets.

The zoom level for off-screen rendering is also different from the zoom level chosen by the user and is determined automatically by the program so that the high-resolution image has exactly the same contents as the current viewport visible through the program. This gives the user an instant preview before exporting to the image file.

4.8 User Interface

The program makes use of QT to display a tool bar with buttons and controls for loading new prototiles either from a file or the operating system's clipboard, for switching between methods and rendering modes and for showing different iterations or layers. See Figures 4.14, and 4.15 for some screen shots of the Irrational Image Generator. Example



Quads

6

☐ Layer

☐ Aspect

X:26.69 Y:-19.25 Zoom:16.00

Resolution 4096x3467 (14.20 MP)

Figure 4.14: Screen shot of the Irrational Image Generator, showing a 3-way IR pattern and the dialog to export the current viewport to a high-resolution bitmap file.

images that have been generated and exported using the Irrational Image Generator can be seen in Section 4.9.

Scrolling and zooming is done intuitively using the mouse while the cursor is placed over the OpenGL widget. Moving the scroll wheel changes the zoom level while locking the view on the mouse cursor. Clicking and dragging with pressed mouse button moves the viewpoint relative to the current zoom level and mouse cursor position.

The program can also switch to a full-screen or presentation mode using a keyboard shortcut. In this mode, all controls are hidden, and the OpenGL view takes up the

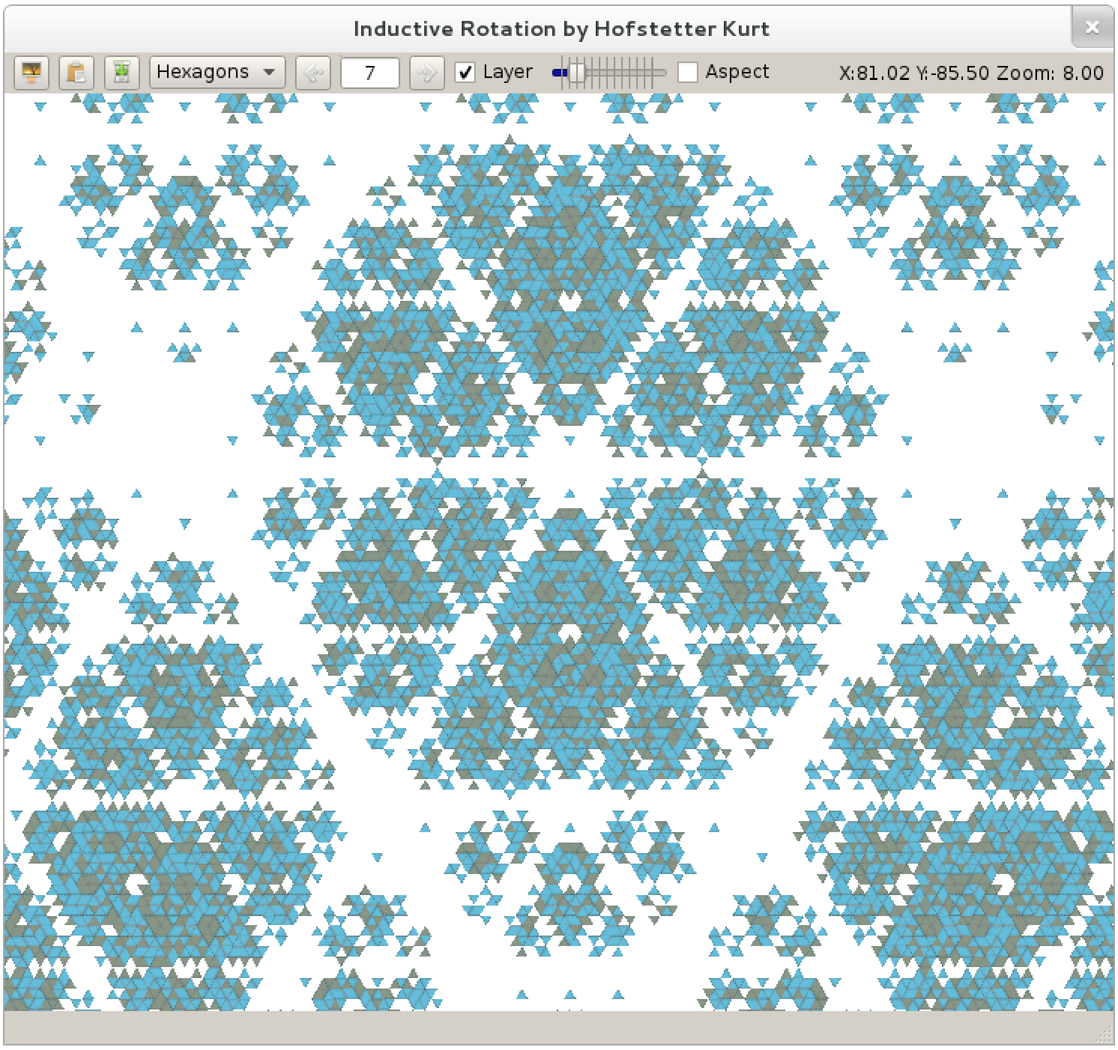


Figure 4.15: Screen shot of the The Irrational Image Generator, showing a hidden layer of a 5-way IR pattern. See also Figure 4.2, which shows a 3D rendering of the same pattern.

entire screen. The user can still interact with the pattern using the mouse and keyboard shortcuts for the most important operations, such as translating, zooming and changing iterations and layers.

Since calculating the pattern can take up some time for a larger number of iterations, the program disables the user interface and shows a loading animation in case the calculation takes longer than a second. When the user clicks on the image export button, a small pop-up menu appears where the user can, depending on the aspect ratio of the current viewport, select between multiple resolutions. While higher resolutions produce better quality, e.g., for print and post-processing, the resulting images can also take up considerably more disk space and need more memory during image compression. A loading animation is displayed, while the program renders, compresses, and writes the exported image to the file system.

4.9 Example Images

The following images have been created using the Irrational Image Generator. The prototiles have been kindly provided by Hofstetter Kurt to show how the program is already being used. Figure 4.16 shows the prototiles. Figures 4.17, 4.18, and 4.19 show images generated using those prototiles.

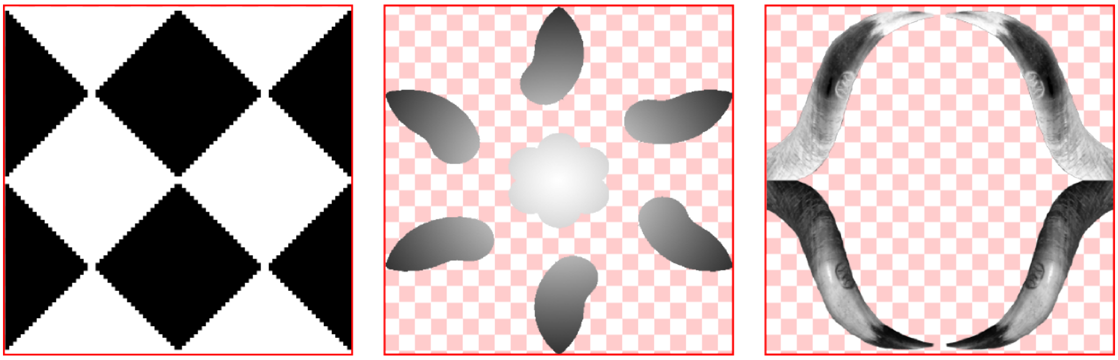


Figure 4.16: Prototiles that were used for generating Figures 4.17, 4.18, and 4.19; in this order. The checkerboard pattern in the background symbolizes transparent areas of the prototile image.

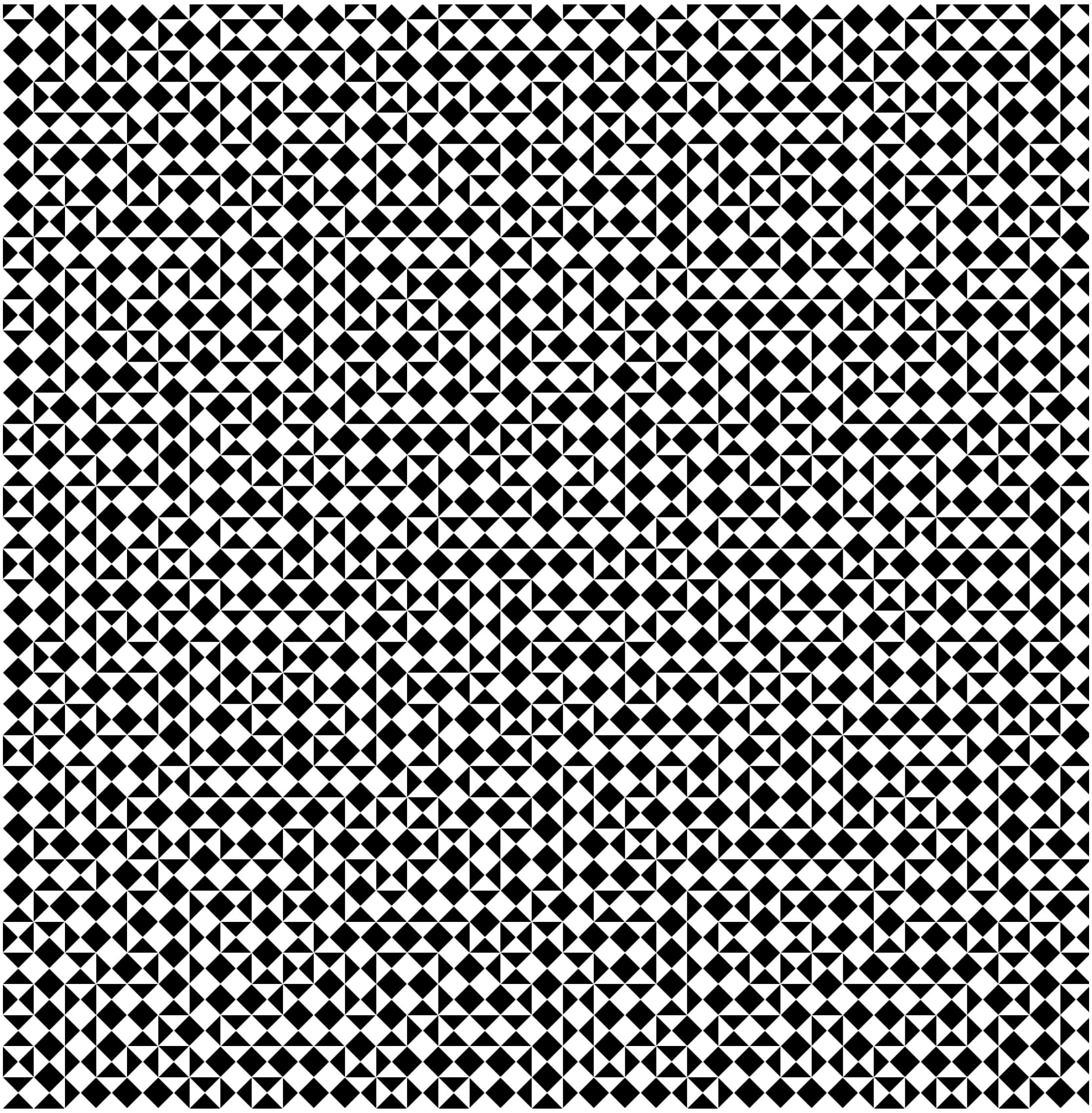


Figure 4.17: Image generated from the first prototile in Figure 4.16, using the 3-way IR method.

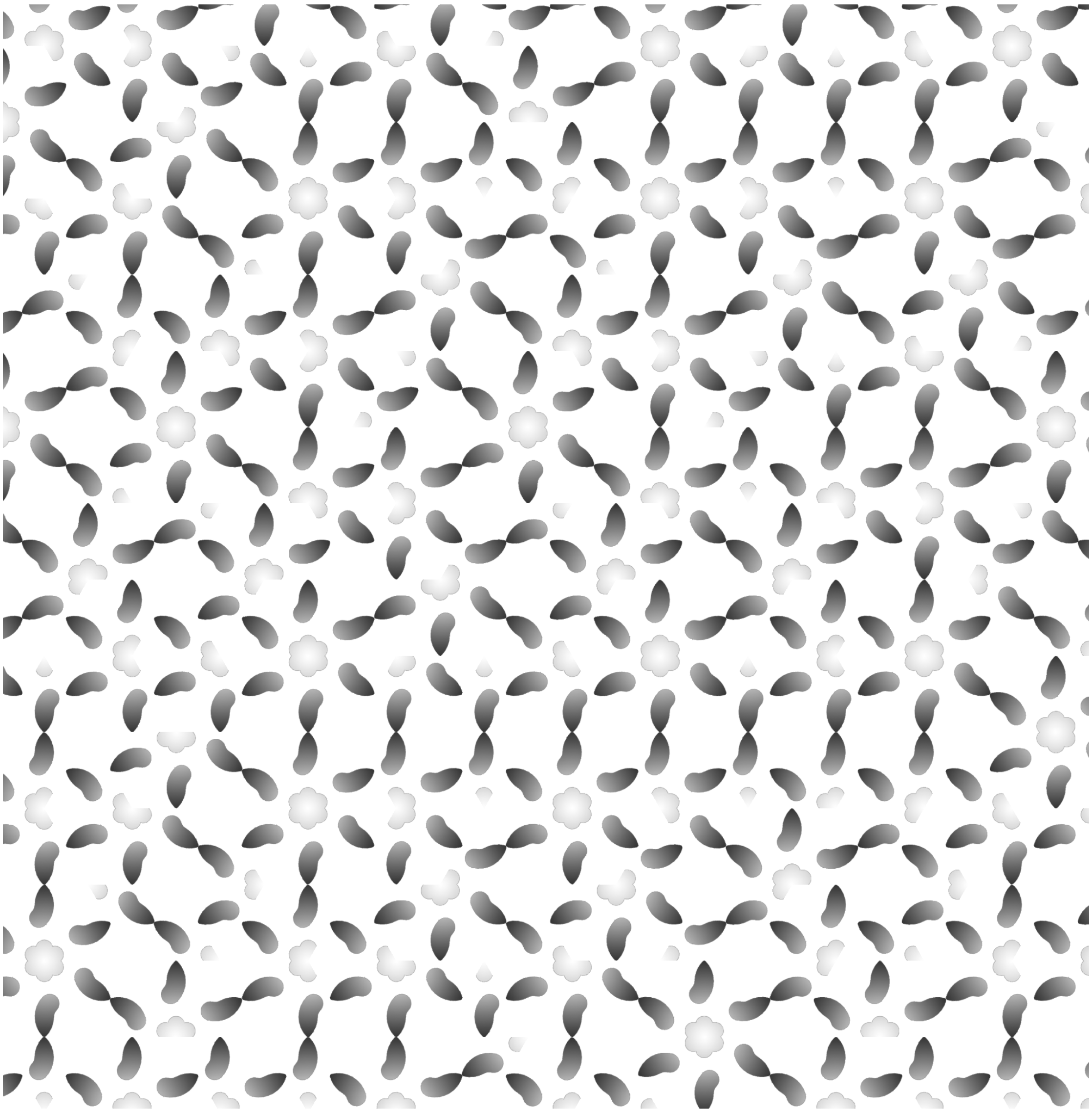


Figure 4.18: Image generated from the second prototile in Figure 4.16, using the 5-way IR method.

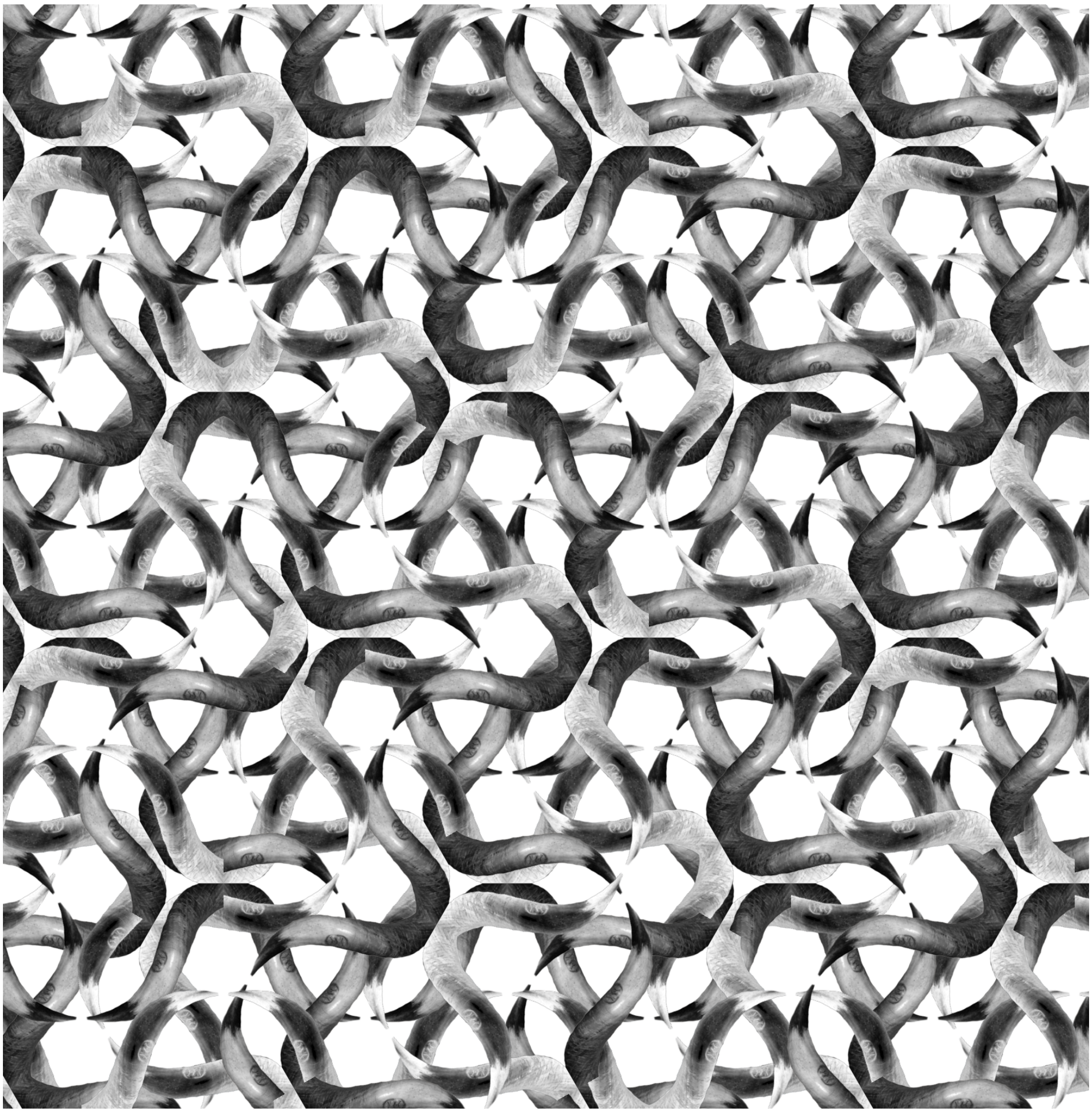


Figure 4.19: Image generated from the third prototile in Figure 4.16, using the 2-way IR method.

4.10 Benchmarks

The benchmark results are from testing the program on the following hardware:

- Intel E6750 dual-core CPU with 2.66 GHz
- 4 GB RAM
- GeForce 8800 GT GPU with 256 MB VRAM

The tests were performed with Windows 8 Professional 64-bit with no additional programs running in the background. The run time has been measured using the *QTime* class provided by QT. The memory usage information results from calling the WinAPI *GetProcessMemoryInfo* function. The following benchmark tables show the `WorkingSetSize` field, which counts both the private and shared memory pages currently in use by the process, in bytes.

The first benchmark shows the run time and memory needed for generating the patterns. All three methods were tested in both the sprite-based and grid-based implementations up to the highest iteration number where the program would not run out of memory (see Tables 4.1 through 4.6). Note that for the 5-way grid-based method (see Table 4.5), both run-time and memory usage are much higher than with the 5-way sprite-based method. This is due to the fact that the grid-based implementation stores overlapping regions in separate layers, ordered by their level of overlap. Since the maximum overlap depth of patterns generated by the 5-way IR method seems to double with each iteration, the amount of run-time and memory dedicated to arranging the overlapping regions into layers also increases. In contrast, the maximum overlap depth of both the 2-way and 3-way IR methods is limited to 2, therefore the difference in performance between grid-based and sprite-based implementations for these methods is not so big.

The second set of benchmarks shows the run time and memory usage for exporting an image in different resolutions. The memory usage in this case is a peak value as the memory needed to hold the exported image is freed after the conversion of the raw image data to PNG and writing the resulting file to disk (see Table 4.7).

Iteration	**Run time**	**Memory (total)**	**Memory (incremental)**
1	3 milliseconds	28,839,936 bytes	0 bytes
2	3 milliseconds	28,848,128 bytes	8,182 bytes
3	2 milliseconds	28,852,224 bytes	4,096 bytes
4	3 milliseconds	28,893,184 bytes	40,960 bytes
5	3 milliseconds	29,057,024 bytes	163,840 bytes
6	3 milliseconds	29,626,368 bytes	569,344 bytes
7	3 milliseconds	30,683,136 bytes	1,056,768 bytes
8	16 milliseconds	35,946,496 bytes	5,263,360 bytes
9	65 milliseconds	59,256,832 bytes	23,310,336 bytes
10	250 milliseconds	141,643,776 bytes	82,386,944 bytes
11	1,062 milliseconds	473,260,032 bytes	331,616,256 bytes

Table 4.1: Benchmark results for the 3-way sprite-based Inductive Rotation.

Iteration	**Run time**	**Memory (total)**	**Memory (incremental)**
1	3 milliseconds	28,307,456 bytes	0 bytes
2	3 milliseconds	28,319,744 bytes	12,288 bytes
3	3 milliseconds	28,336,128 bytes	16,384 bytes
4	2 milliseconds	28,635,136 bytes	299,008 bytes
5	2 milliseconds	29,253,632 bytes	618,496 bytes
6	12 milliseconds	33,591,296 bytes	4,337,664 bytes
7	70 milliseconds	60,698,624 bytes	27,107,328 bytes
8	481 milliseconds	214,183,936 bytes	153,485,312 bytes

Table 4.2: Benchmark results for the 5-way sprite-based Inductive Rotation.

Iteration	Run time	Memory (total)	Memory (incremental)
1	3 milliseconds	28,332,032 bytes	0 bytes
2	2 milliseconds	28,344,320 bytes	12,288 bytes
3	3 milliseconds	28,348,416 bytes	4,096 bytes
4	2 milliseconds	28,348,416 bytes	0 bytes
5	2 milliseconds	28,372,992 bytes	24,576 bytes
6	2 milliseconds	28,430,336 bytes	57,344 bytes
7	2 milliseconds	28,680,192 bytes	249,856 bytes
8	3 milliseconds	29,073,408 bytes	393,216 bytes
9	4 milliseconds	30,494,720 bytes	1,421,312 bytes
10	13 milliseconds	36,990,976 bytes	6,496,256 bytes
11	37 milliseconds	48,939,008 bytes	11,948,032 bytes
12	128 milliseconds	86,061,056 bytes	37,122,048 bytes
13	354 milliseconds	205,197,312 bytes	119,136,256 bytes
14	1,191 milliseconds	541,155,328 bytes	335,958,016 bytes

Table 4.3: Benchmark results for the 2-way sprite-based Inductive Rotation. Note that iteration 4 does not seem to increase memory usage. This is most likely due to the way the operating system reserves memory. The amount of memory reserved for a process does not always reflect the memory actually used.

Iteration	Run time	Memory (total)	Memory (incremental)
1	3 milliseconds	28,549,120 bytes	0 bytes
2	3 milliseconds	28,692,480 bytes	143,360 bytes
3	3 milliseconds	28,721,152 bytes	28,672 bytes
4	2 milliseconds	28,856,320 bytes	135,168 bytes
5	3 milliseconds	29,138,944 bytes	282,624 bytes
6	6 milliseconds	30,138,368 bytes	999,424 bytes
7	25 milliseconds	34,324,480 bytes	4,186,112 bytes
8	96 milliseconds	53,059,584 bytes	18,735,104 bytes
9	386 milliseconds	119,521,280 bytes	66,461,696 bytes
10	1,674 milliseconds	384,229,376 bytes	264,708,096 bytes

Table 4.4: Benchmark results for the 3-way grid-based Inductive Rotation.

Iteration	Run time	Memory (total)	Memory (incremental)
1	3 milliseconds	28,925,952 bytes	0 bytes
2	2 milliseconds	29,515,776 bytes	589,824 bytes
3	6 milliseconds	30,801,920 bytes	1,286,144 bytes
4	16 milliseconds	33,312,768 bytes	2,510,848 bytes
5	53 milliseconds	40,624,128 bytes	7,311,360 bytes
6	265 milliseconds	65,486,848 bytes	24,862,720 bytes
7	1,903 milliseconds	201,539,584 bytes	136,052,736 bytes
8	14,478 milliseconds	891,453,440 bytes	689,913,856 bytes

Table 4.5: Benchmark results for the 5-way grid-based Inductive Rotation.

Iteration	Run time	Memory (total)	Memory (incremental)
1	3 milliseconds	28,684,288 bytes	0 bytes
2	3 milliseconds	28,745,728 bytes	61,440 bytes
3	3 milliseconds	28,917,760 bytes	172,032 bytes
4	3 milliseconds	28,950,528 bytes	32,768 bytes
5	4 milliseconds	29,093,888 bytes	143,360 bytes
6	5 milliseconds	29,491,200 bytes	397,312 bytes
7	12 milliseconds	30,601,216 bytes	1,110,016 bytes
8	39 milliseconds	34,308,096 bytes	3,706,880 bytes
9	108 milliseconds	44,494,848 bytes	10,186,752 bytes
10	323 milliseconds	74,829,824 bytes	30,334,976 bytes
11	1,017 milliseconds	172,720,128 bytes	97,890,304 bytes
12	3,068 milliseconds	440,029,184 bytes	267,309,056 bytes

Table 4.6: Benchmark results for the 2-way grid-based Inductive Rotation.

Resolution	Run time	Memory usage
1024x689	327 milliseconds	48,996,352 bytes
2048x1378	484 milliseconds	61,603,840 bytes
4096x2756	1,625 milliseconds	95,698,944 bytes
8192x5512	5,493 milliseconds	231,432,192 bytes
16384x11025	19,830 milliseconds	774,492,160 bytes

Table 4.7: Benchmark results for exporting an image of iteration 5 of the 3-way Inductive Rotation at various resolutions.

CHAPTER 5

Summary

Inductive Rotation is a relatively new approach for generating a family of patterns that are assumed to tile the plane nonperiodically. As of today, this approach has not been described in the scientific literature.

One aim of this thesis is to give a description of the method, compare it to similar approaches, such as other nonperiodic tilings of the plane or fractals. The main part of this work, however, is an implementation named “Irrational Image Generator” that serves both as a reference implementation of the Inductive Rotation method, as well as a tool for artistic experimentation.

5.1 Nonperiodic Tilings

To properly understand the concept of aperiodic tilings of the plane, some terms have to be defined beforehand. According to Grünbaum and Shephard [13], a tiling of the plane “is a countable family of closed sets, which cover the plane without gaps or overlaps”. A tiling in 2D space tessellates the infinite plane while only using a finite set of closed shapes, called **prototiles**. The entire plane is covered, without any tiles overlapping.

Another important term is the periodicity of a tiling. A **periodic** tiling by definition has **translational symmetry**, meaning that for the infinite tiling, there is a translation vector that maps the tiling onto itself. A tiling that cannot be mapped onto itself using only a translation is called a **nonperiodic** tiling.

A set of prototiles that only allow nonperiodic tilings, i.e., it is not possible to create a periodic tiling using the defined set of prototiles, is called an **aperiodic** set. Some aperiodic tile sets include matching rules to enforce the aperiodicity, even though the geometric shape of the prototiles alone would allow periodic tilings. One example of this are the Wang tiles, consisting of a set of square tiles with color-coded edges. The matching rule specifies that only edges with the same color can touch.

Wang’s reason for introducing this kind of color-coded tiles in a paper published in 1961[44] was to illustrate the “Domino Problem”, a special case of Hilbert’s decision problem [15]. Wang's conjecture states that if any set of tiles is "solvable", i.e., allows an infinite tiling of the plane, it also has a periodic solution.

Berger [2] has proven this conjecture wrong in 1966 by constructing a set of over 20,000 Wang tiles that allow only nonperiodic tilings. As of today, the smallest aperiodic set of Wang tiles consists of 13 different prototiles [5].

Probably the most noteworthy example of aperiodic tilings is an aperiodic set of only 2 prototiles introduced by Penrose [32], subsequently known as "Penrose Tiling". The aperiodicity can either be enforced by matching rules, or by modifying the geometric shape of the tiles so that only matching edges fit together. As of today, the Penrose Tiling is also the smallest known aperiodic set of prototiles.

Apart from creating a set of prototiles, with or without matching rules, there are also other known methods of creating an infinite nonperiodic tiling of the plane. For example, the class of substitution tilings is based on the principle of infinitely subdividing a shape according to pre-defined rules. Both Penrose and Wang tilings can be also generated using this method. One notable substitution tiling is the Pinwheel tiling of the plane[36], originally based on an unpublished construction by John Conway, which only consists of copies of the same triangle, yet is nonperiodic due to the fact that this triangle appears in uncountably many rotations.

The matter of substitution tilings has been extensively covered in a paper by Frettlöh [10], who also has created an online encyclopedia that tries to list all substitution patterns that have been found so far [11].

5.2 Fractals

Fractals are another well-known approach for creating beautiful self-similar patterns. The term fractal was first used by mathematician Benoît Mandelbrot to describe recursively defined patterns that also feature detailed self-similarity. Self-similarity means that a pattern repeats itself over and over again, in infinitely many scaling levels. Fractal patterns show detailed structures at arbitrary small scales while always repeating the same, or at least very similar patterns.

One popular example of fractals is the Koch Snowflake, essentially an equilateral triangle where each edge is a recursively defined curve. At an infinite recursion level, the Koch Snowflake is continuous everywhere, but differentiable nowhere.

The rendering of fractal objects, usually with the aid of a computer program, can be considered an art form. By choosing appropriate parameters for zoom levels and color mappings, beautiful images can be created. Mandelbrot gives an introduction to fractal art in an article published in 1989 [24].

Probably the most well-known example of fractal art are renditions of the Mandelbrot and Julia sets. These fractal objects are based on the iteration of complex polynomials. The rendering usually maps each pixel with offset (x, y) of a raster image to a complex number $c = ai + b$, e.g., $x := a$ and $y := b$. The color of each pixel is determined by properties of the number c, e.g., white for numbers that are contained in the Mandelbrot set and black for numbers that are not.

5.3 Inductive Rotation

Hofstetter Kurt's development of the Inductive Rotation method has been inspired by the ongoing search for an aperiodic tiling consisting of only a single prototile. Inductive Rotation describes an iterative construction that starts with a single prototile. In each iteration, the existing pattern is copied and rotated around a pivot point repeatedly. Hofstetter Kurt's constructions explicitly allow overlapping regions. Overlapping parts are always hidden behind the already existing pattern.

This approach allows for interesting patterns created using only a single prototile as well as creating a three-dimensional structure if the hidden regions are taken into consideration. So far, three different rotation methods that generate interesting, nonperiodic patterns have been found:

- The 3-way Inductive Rotation. Starting with a quad as prototile, the pattern is three times copied and rotated around its rightmost vertical center.
- The 5-way Inductive Rotation. Starting with a regular hexagon as prototile, the pattern is five times copied and rotated around its rightmost vertical center.
- The 2-way Inductive Rotation. Starting with a star figure that is constructed by extending the edges of a regular hexagon by equilateral triangles, the pattern is two times rotated around a pivot point that is always the rightmost vertical center of the rightmost star figure prototile. In case more than one prototile qualifies, the bottommost one is used.

The prototile is not only defined by its geometric shape, but can also contain a pattern, usually defined by a 2D image. The artist experiments with different images. While the rotation methods in itself are simple to explain, the results are hard to predict.

The goal of this thesis was to create a new tool for accelerating this process and giving the artist a faster, more intuitive workflow experience. As the recursive rotation steps are performed automatically by the computer, the artist can fully focus on the creation and selection of different prototiles to create aesthetically pleasing patterns.

For the implementation, a simple high-level algorithm for creating Inductive Rotation patterns has been described. Also, a data structure suitable for storing and manipulating the pattern has been designed. The pattern is initialized with a single prototile. The algorithm uses three operations for generating the Inductive Rotation patterns. The **pivot** operation determines the rotation pivot point for the current iteration number i. The **rotate-copy** creates a copy of the entire data structure that is rotated by α degrees around the pivot point. The **merge** operation merges a rotated copy created by the **rotate-copy** operation, considering the rule that overlaps are either omitted or placed behind existing data.

For one iteration of the algorithm, the **pivot** operation is called first, determining the pivot point used in the entire iteration. This is followed by a number of **rotate-copy** operations and the same number of **merge** operations. This number depends on the method implemented. For example, the 3-way Inductive Rotation needs three

rotate-copy operations with angles 90, 180, and 270 degrees followed by three **merge** operations that merge these rotated copies into the initial data structure.

The complexity of the algorithm depends on the number of rotations per iteration r and is generally $\mathcal{O}((r+1)^n)$ where n is the number of iterations performed.

5.4 Implementation

The implementation is based on QT and OpenGL. The resulting tool is called "Irrational Image Generator". A previous implementation created during the course of a practical work at the Institute of Computer Graphics and Algorithms, Vienna University of Technology works directly on an image buffer. This requires rotation of the individual pixels of the prototile image and has a negative impact on performance, memory requirements as well as the quality of the rendered image.

From the algorithm described earlier, two different implementations that both have their advantages and disadvantages, have been developed. They are called the grid-based and sprite-based implementation. Instead of working directly on image buffers, the data structures are based on the fact that the prototile itself does not change, i.e., is only rotated. With this in mind, only polygons have to be stored. They are then filled with pixel data from the prototile image. This allows for higher iterations and yields better image quality since rasterization artifacts do not accumulate.

The user interface consists of a minimalistic tool bar for actions like switching between iterations/layers, or loading new prototiles either from image files or the operating system's clipboard. Translating and zooming is done with the mouse. The program also features keyboard shortcuts, a full-screen presentation mode and the ability to save the image contained in the current viewport in high resolution to disk.

5.4.1 Grid-Based Implementation

The grid-based implementation is based on the discovery that the generated patterns can be aligned on a regular grid. This is a regular quad grid for the 3-way Inductive Rotation and a regular triangle grid for both the 5-way and 2-way Inductive Rotation methods. The algorithm works on a dynamically allocated two-dimensional array that is only limited by memory constraints.

The prototile is split into grid tiles, and each possible rotation of each prototile part is mapped to a positive integer value in the array. If an array cell contains a negative value, the cell is considered empty. While the mapping from a quad grid to an array is straightforward, the triangle grid needs extra conventions to make the mapping unambiguous.

Since the grids and mapping conventions differ for each of the three Inductive Rotation methods, each needs its own implementation of the **pivot** and **rotate-copy** operations, while the **merge** operation stays the same. All three operations work on the two-dimensional array, filling the array cells with numbers corresponding to prototile parts. Overlaps can either be omitted or handled by creating multiple layers as separate array instances. The resulting pattern is then converted to a polygon mesh that is filled by texture data from the prototile image according to the rotation of the prototile part specified by the mapping conventions.

Evaluation of the grid-based method has revealed two drawbacks. Firstly, prototiles for both the 5-way and 2-way methods are split in parts along edges that are not aligned to pixel borders. This can result in visible artifacts in the rendered image. Secondly, when using images not confined to the limiting hexagon or star figure for the 5-way or 2-way method, image parts outside of the limiting shape are cut off.

5.4.2 Sprite-Based Implementation

While the grid-based implementation performs well enough for artistic experimentation, the drawbacks mentioned earlier led to a second implementation that can be used as an alternative. The tool supports both implementations and allows switching on the fly.

The sprite-based implementation no longer makes use of a grid alignment. Instead, the prototile is treated as a textured rectangle and represented by its center point (x, y) and rotation r in degrees. The underlying data structure is a list of such triples (x, y, r). The **rotate-copy** operation makes a copy of the previous iteration's triple list, rotates each triple's center position around the current iteration's pivot point by α degrees and increases the rotation r by α. The **merge** operation appends such a copied list to another list.

The convention that newly created prototile copies must be rendered behind existing prototile copies is enforced by rendering the polygons created from the triple list in reverse order, i.e., from back to front. Compared to the grid-based implementation, this approach yields less rasterization artifacts, does not cut off image data stored in the prototile texture and has slightly better performance due to a simpler data structure. The grid alignment is lost, however, and so is the underlying two-dimensional structure of the pattern. It is assumed that in the case of further research, the grid-based implementation will be the more useful one. Compared to the sprite-based implementation its data structure offers more possibilities for analysis of the overlapping regions.

Bibliography

[1] M.F. Barnsley. *Fractals Everywhere*. Academic Press, 1988, p. 394. ISBN: 0120790629.

[2] R. Berger. *The Undecidability of the Domino Problem*. Memoirs No 1/66. American Mathematical Society, 1966. ISBN: 9780821812662.

[3] M.F. Cohen, J. Shade, S. Hiller, and O. Deussen. "Wang Tiles for image and texture generation". In: *ACM SIGGRAPH 2003 Papers*. SIGGRAPH '03. San Diego, California: ACM, 2003, pp. 287–294. ISBN: 1-58113-709-5. DOI: `10.1145/1201775.882265`.

[4] R.M. Crownover. *Introduction to fractals and chaos*. Jones and Bartlett books in mathematics. Jones and Bartlett, 1995. ISBN: 9780867204643.

[5] K. Culik II. "An aperiodic set of 13 Wang tiles". In: *Discrete Math.* 160.1-3 (Nov. 1996), pp. 245–251. ISSN: 0012-365X. DOI: `10.1016/S0012-365X(96)00118-5`.

[6] N.G. de Bruijn. "Algebraic theory of Penrose's non-periodic tilings of the plane. I". In: *Indagationes mathematicae Koninklijke Nederlandse Akademie van Wetenschappen, Series A* 43.1 1 (1981), pp. 39–39.

[7] A.K. Dewdney. "Computer Recreations". In: *Scientific American* 253.6 (Dec. 1985), pp. 16–26. ISSN: 0036-8733. DOI: `10.1038/scientificamerican1285-16`.

[8] A. Fournier and D. Fussell. "Computer rendering of stochastic models". In: *Communications of the ACM* 25.6 (June 1982), pp. 371–384. ISSN: 00010782. DOI: `10.1145/358523.358553`.

[9] *Fractal Terrain: Midpoint Displacement Algorithm*. URL: `http://tim.hibal.org/blog/?p=139` (visited on 2013-04-23).

[10] D. Frettlöh. "Duality of model sets generated by substitutions". In: *Arxiv preprint math/0601064* (2006), pp. 1–16.

[11] D. Frettlöh. *Tilings Encyclopedia*. URL: `http://tilings.math.uni-bielefeld.de` (visited on 2013-04-23).

[12] *GLFW - An OpenGL Library*. URL: `http://www.glfw.org` (visited on 2013-04-23).

[13] B. Grünbaum and G.C. Shephard. *Tilings and Patterns*. W. H. Freeman & Co. New York, NY, USA, 1986, p. 700. ISBN: 0-716-71193-1.

[14] P. Gummelt. "Penrose tilings as coverings of congruent decagons". In: *Geometriae Dedicata* 62.1 (1996), pp. 1–17.

[15] Y. Gurevich. "On the classical decision problem". In: *Current Trends in Theoretical Computer Science* October 1990 (1993), pp. 254–265.

[16] *Introduction to WPF.* URL: `http://msdn.microsoft.com/en-us/library/aa970268.aspx` (visited on 2013-04-23).

[17] *Irrational Patterns by Hofstetter Kurt.* URL: `http://hofstetterkurt.net/ip.html` (visited on 2013-04-23).

[18] H.C. Jeong and P.J. Steinhardt. "Constructing Penrose-like tilings from a single prototile and the implications for quasicrystals". In: *Phys. Rev. B* 55 (6 Feb. 1997), pp. 3520–3532. DOI: `10.1103/PhysRevB.55.3520`.

[19] C.S. Kaplan. "Computer graphics and geometric ornamental design". PhD thesis. University of Washington, 2002.

[20] C.S. Kaplan. *Penrose Tiling Applet.* URL: `http://www.cgl.uwaterloo.ca/~csk/software/penrose/` (visited on 2013-06-03).

[21] J. Kepler. *Harmonices mundi. 1619.* Sommer, 2005.

[22] J. Kopf, D. Cohen-Or, O. Deussen, and D. Lischinski. "Recursive Wang tiles for real-time blue noise". In: *ACM SIGGRAPH 2006 Papers.* SIGGRAPH '06. Boston, Massachusetts: ACM, 2006, pp. 509–518. ISBN: 1-59593-364-6. DOI: `10.1145/1179352.1141916`.

[23] P.J. Lu and P.J. Steinhardt. "Decagonal and quasi-crystalline tilings in medieval Islamic architecture." In: *Science (New York, N.Y.)* 315.5815 (Feb. 2007), pp. 1106–10. ISSN: 1095-9203. DOI: `10.1126/science.1135491`.

[24] B.B. Mandelbrot. "Fractals and an art for the sake of science". In: *Leonardo. Supplemental Issue* (1989), pp. 21–24.

[25] B.B. Mandelbrot. *The Fractal Geometry of Nature.* Henry Holt and Company, 1983. ISBN: 9780716711865.

[26] *Mandelbulb: The Unravelling of the Real 3D Mandelbrot Fractal.* URL: `http://www.skytopia.com/project/fractal/mandelbulb.html` (visited on 2013-04-23).

[27] *Mandel_zoom_00_mandelbrot_set.jpg.* URL: `http://de.wikipedia.org/w/index.php?title=Datei:Mandel_zoom_00_mandelbrot_set.jpg&filetimestamp=20061204213914` (visited on 2013-04-23).

[28] K. Marczak. *Mandelbulber: 3D Fractal Explorer.* URL: `http://www.mandelbulber.com` (visited on 2013-04-23).

[29] K. Mitchell. *The Fractal Art Manifesto.* URL: `https://www.fractalus.com/info/manifesto.htm` (visited on 2013-04-23).

[30] V. Ostromoukhov, C. Donohue, and P.M. Jodoin. "Fast hierarchical importance sampling with blue noise properties". In: *ACM Transactions on Graphics (TOG).* Vol. 23. 3. ACM. 2004, pp. 488–495.

[31] *Penrose Tiles Talk Across Miles.* URL: `http://www.ams.org/samplings/feature-column/fcarc-penrose` (visited on 2013-04-23).

[32] R. Penrose. “Pentaplexity a class of non-periodic tilings of the plane”. In: *The mathematical intelligencer* 2.1 (1979), pp. 32–37.

[33] R. Penrose. “Set of tiles for covering a surface”. Patent US 4133152 (US). Jan. 9, 1979.

[34] P. Prusinkiewicz and J. Hanan. *Lindenmayer systems, fractals, and plants.* Vol. 79. Springer-Verlag Berlin, 1989.

[35] *Qt Project.* URL: `http://qt-project.org` (visited on 2013-04-23).

[36] C. Radin. “The pinwheel tilings of the plane”. In: *The Annals of Mathematics* 139.3 (1994), pp. 661–702.

[37] R.M. Robinson. “Undecidability and nonperiodicity for tilings of the plane”. In: *Inventiones mathematicae* 12 (1971), pp. 177–209.

[38] M. Senechal. *Quasicrystals and geometry.* Vol. 9. 12. Cambridge Univ Pr, 1996, pp. 994–996. DOI: `10.1002/adma.19970091217`.

[39] D. Shechtman, I. Blech, D. Gratias, and J.W. Cahn. “Metallic phase with long-range orientational order and no translational symmetry”. In: *Physical Review Letters* 53.20 (1984), pp. 1951–1953.

[40] J. Stam. *Aperiodic texture mapping.* ERCIM research reports. European Research Consortium for Informatics and Mathematics, 1997.

[41] B. Stroustrup. *The C++ Programming Language.* Vol. 3rd Editio. Addison-Wesley, 1997, p. 1022. ISBN: 0201889544.

[42] T. Theußl, T. Möller, and M.E. Gröller. “Optimal regular volume sampling”. In: *Visualization, 2001. VIS ’01. Proceedings.* 2001, pp. 91–546. DOI: `10.1109/VISUAL.2001.964498`.

[43] *view3dscene.* URL: `http://castle-engine.sourceforge.net/view3dscene.php` (visited on 2013-05-22).

[44] H. Wang. “Proving theorems by pattern recognition II”. In: *Bell Systems Technical Journal* 40.1 (1961), pp. 1–42.

[45] L.Y. Wei. “Tile-based texture mapping on graphics hardware”. In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware.* ACM, 2004, pp. 55–63.

[46] *X3D Specification Schema.* URL: `http://www.web3d.org/specifications/x3d-3.0.xsd` (visited on 2013-04-23).