FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Ein Framework für die GPU-gestützte Erzeugung und Gestaltung induktiv rotierter Muster

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Visual Computing

eingereicht von

Sebastian Sippl Bakk techn.
Matrikelnummer 0421271

an der Fakultät für Informatik
der Technischen Universität Wien

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

Wien, 27. August 2015

Sebastian Sippl	M. Eduard Gröller

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

A Framework for GPU-Assisted Generation and Composition of Inductive Rotation Patterns

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Visual Computing

by

Sebastian Sippl Bakk techn.
Registration Number 0421271

to the Faculty of Informatics
at the Vienna University of Technology

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

Vienna, 27th August, 2015

Sebastian Sippl | M. Eduard Gröller

Erklärung zur Verfassung der Arbeit

Sebastian Sippl Bakk techn.
Löhrgasse 22/13 1150 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.

Wien, 27. August 2015

Sebastian Sippl

Kurzfassung

Die Methode der Induktiven Rotation, die vom Künstler Hofstetter Kurt entwickelt wurde, ist eine Strategie, um aufwändige künstlerische Muster zu erzeugen. Die Muster werden generiert, indem wiederholt Rotationen und Translationen auf die Kopien eines sogenannten Prototiles angewendet werden. Inspiriert wurde diese Methode durch aperiodische Muster wie zum Beispiel die berühmten Penrose Parkettierungen. Die Muster und deren nichtperiodische Struktur, die durch die Anwendung der Induktiven Rotation entstehen, haben, sowohl aus künstlerischer als auch aus wissenschaftlicher Sicht, interessante Eigenschaften. Im Rahmen einer vorherherigen Masterarbeit, aus der das Programm "The Irrational Image Generator" hervorging, wurden bereits verschiedene Algorithmen für die Erzeugung solcher Muster implementiert und erforscht. Dieser Software-Prototyp bietet jedoch nur wenige gestalterische Möglichkeiten für den Künstler und kann nur Muster von begrenzter Größe erzeugen. Die limitierte Größe resultiert aus einer Eigenschaft der Muster: Die Anzahl der Elemente in einem Muster wächst exponentiell mit jeder Iterationsstufe.
Das Inductive Rotation Framework, ein Software Framework für die Generierung von Mustern mittels Induktiver Rotation, welches im Rahmen dieser Arbeit entwickelt wurde, vereinigt neue Erzeugungsalgorithmen für Muster mit verbesserten Werkzeugen, wie etwa einem Textur-Editor, die Hofstetter Unterstützung während des Muster-Designprozesses bieten. Einer der bestehenden Algorithmen konnte erfolgreich parallelisiert werden. Damit ist es nun möglich, die Mustererzeugung mittels GPGPU Methoden durchzuführen. Abhängig von der Implementierung des Algorithmus ermöglicht dies entweder Muster viel schneller zu erzeugen oder die Maximalgröße der Muster zu erhöhen. Um ausserdem die Vor- und Nachteile einer kürzlich entwickelten Methode zu erforschen, die es ermöglichen soll, Muster im Sinne der Induktiven Rotation durch Ersetzungsstrategien zu erzeugen, wurde ein weiterer, auf dieser Methode basierender Algorithmus entwickelt und implementiert. Diese neue Ersetzungsmethode kann jedoch nur eine Teilmenge von Mustern erzeugen, die der Defnition der Induktiven Rotation von Hofstetter genügen.
Wenn man die Definition der Induktiven Rotation leicht variiert bilden sich Sierpinski Dreiecke, d.h. fraktale Muster. Die Ähnlichkeit der Induktiven Rotation zu Fraktalen kann auch beobachtet werden, wenn man das Matrixschema des parallelen Erzeugungsalgorithmuses mit Iterierten Funktionssystemen vergleicht, die auch zur Erzeugung von Fraktalen verwendet werden.

Abstract

The Inductive Rotation Method, developed by the artist Hofstetter Kurt, is a strategy for generating elaborate artistic patterns by applying translations and rotations repeatedly to a copy of a so called prototile. The method has been inspired by aperiodic tilings such as the popular Penrose tilings. The Inductive Rotation Patterns and their nonperiodic structure is interesting from both a mathematical and from an artistic point of view. In the scope of a previous thesis different algorithms for the generation of such patterns were already implemented and researched which resulted in a program called the "Irrational Image Generator". However, this software prototype provides only few features which support Hofstetter in designing patterns, and can only produce patterns with limited size. The limited size results from a property of the patterns: The number of tiles grows exponentially with each iteration.

The Inductive Rotation Framework, a software framework for the generation of Inductive Rotation Patterns, was developed in the course of this thesis and unites new generation algorithms with an extended tool-set, like a graphical prototile editor which supports Hofstetter in his pattern design process. One of the existing algorithms was successfully parallelized and now allows the artist pattern generation via GPGPU methods. Depending on the implementation this can increase either pattern generation speed or the maximum pattern-size. In order to research the advantages and disadvantages of a recently developed tile substitution method for the creation of Inductive Rotation Patterns, the framework was extended by an algorithm which is based on this new discovery. Following the definition of the Inductive Rotation Method from Hofstetter, this tile-substitution method produces only a subset of Inductive Rotation Patterns.

By varying the definition of Hofstetter's Inductive Rotation Method only slightly, the Sierpinski gasket, a fractal pattern, emerges. The similarity between the Inductive Rotation Method and fractals can be observed further by comparing the parallel generation algorithm's matrix scheme to Iterated Function Systems (IFSs), which are used to generate fractals.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

To create two-dimensional, aperiodic patterns the artist Hofstetter Kurt developed a method called the Inductive Rotation (IR). A first scientific investigation of IR-Patterns was conducted by Parzer [Par13], who developed different algorithms for the creation of IR-Patterns and provided "The Irrational Image Generator", a program which generates IR-Patterns. However, this tool provides only pattern generation methods but offers no features for a convenient artistic design process. Besides, there are other scientific questions regarding the Inductive Rotation Method: One question is how to efficiently parallelize pattern generation and split the generation process of large patterns into multiple threads. The other question is if it is possible to generate Inductive Rotation Patterns by applying tile-substitution methods. Recent work [FH15] indicates that this is possible for a subset of these patterns.
The aim of this thesis is the implementation of a new framework for the creation of patterns with the Inductive Rotation Method. This new Inductive Rotation Framework should include the features of the precursor software from Parzer [Par13], provide new features which help to improve Hofstetter's design process (e.g., a live prototile editor) and contribute a new method for parallelizing pattern generation. Furthermore, the thesis investigates the advantages and disadvantages of the substitution method from Frettlöh and Hofstetter [FH15], provides an overview of currently existing algorithms for the generation of Inductive Rotation Patterns and gives insights on their structure and relationship to fractals.
The thesis is structured as follows:
Chapter 2 contains an introduction to related topics such as tilings, fractals and Voronoi diagrams. Chapter 3 gives an overview of the Inductive Rotation Method, describes generation algorithms for Inductive Rotation Patterns and provides ideas for possible applications. Chapter 4 describes the implementation process and provides details of the software implementation. Chapter 5 contains algorithm benchmarks, user feedback, images of new patterns and a conclusion. Chapter 6 summarizes the contents of the thesis.

CHAPTER 2

State of the Art

This chapter offers an overview of the state of the art regarding patterns, tilings, fractals and other topics related to the Inductive Rotation Method. The first section reviews the definition of mathematical plane tilings and further describes the difference between the terms periodic and aperiodic in this context. Wang tilings, i.e., aperiodic tilings with interesting applications for progressive texture generation, are presented in the second section and Voronoi diagrams, which are reviewed in the third section, are a widely used and well known partitioning of the plane. Voronoi diagrams create tilings based on distances from a point set. Hyperbolic geometry, presented in Section 2.4, lives in a saddle shaped environment and fractals, self similar sets, are discussed in Section 2.5.

2.1 Tilings of the Plane

This section gives a general overview on plane-tilings. Grünbaum and Shephard [GS86], the two authors of a standard reference book on mathematical tilings, define tilings as "a countable family of closed sets, which cover the plane without gaps or overlaps". Mathematical tilings cover the infinite plane with polygons originating from a set of so called prototiles. To create a valid tiling, the pattern must not contain any overlaps or gaps. The following definitions provide the reader with a more comprehensive categorization of tilings:

- A **tiling** is called **periodic**, if it is possible to map each part of the tiling to itself by applying translations only. This means a translation vector (x, y) with $x \neq 0$ and/or $y \neq 0$ exists that when applied yields the same tiling.
- A **tiling** is called **nonperiodic**, if it is not periodic.
- A **set of prototiles** is called an **aperiodic** tile set, if it is not possible to construct a periodic tiling from the set.

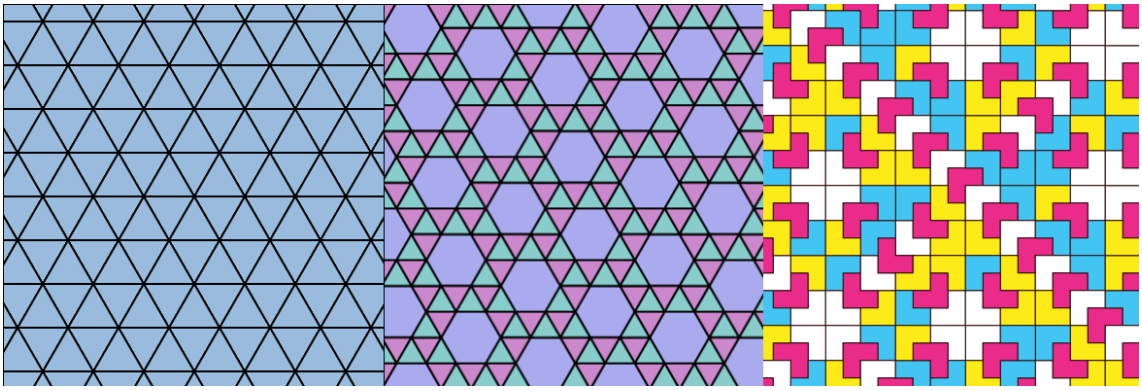

Figure 2.1: Three different possibilities to create plane tilings. The images depict, from left to right, a periodic tiling with a single prototile, a periodic tiling using three different prototiles and a nonperiodic tiling created with a set of four prototiles. Please note that the set of prototiles used to construct the nonperiodic tiling is not aperiodic. [Coma,Comb,Pal]

The Schläfli Symbol is a notation which describes tessellations or regular polytopes in different coordinate spaces and geometries. For tilings the notation is of the form $\{p, q\}$, where p is an integer defining the number of corner vertices of a regular polygon and the integer q defines the number of polygons touching at each vertex.
In the Euclidean plane there exist exactly three different tessellations by regular polygons which use only a single prototile. The tessellation by equilateral triangles $\{3, 6\}$, squares $\{4, 4\}$ and hexagons $\{6, 3\}$. Figure 2.1 depicts a periodic tiling with a single regular polygon, a periodic tiling using a set of regular polygons and a nonperiodic tiling constructed with a set of prototiles.

2.2 Wang Tiles

Wang tiles, developed by Hao Wang, a mathematician and philosopher from the 20^{th} century, are quadratic tiles with color-coded edges. To create a valid Wang tiling, copies of Wang tiles belonging to the set of prototiles are arranged in a way such that only tiles with matching edge colors touch. This approach produces periodic or aperiodic tilings of the plane, depending on the chosen set of prototiles. It has been proven mathematically that it is undecidable if an arbitrary given set of Wang tiles can tile the plane periodically. This fact was discovered by a student of Wang in the course of researching the so called domino problem.

2.2.1 Construction of Wang Tilings

There are several algorithms to efficiently construct Wang tilings. A popular method presented by Cohen et al. [CSHD03] is based on a stochastic scan line algorithm.

Figure 2.2: An example for a set of Wang tiles. The edges of Wang tiles are color coded. In a Wang tiling only tiles with identically colored edges may touch. Multiple subsets of this specific set of tiles may also produce valid Wang tilings. Cohen et al. exploit this property to create Wang tilings with their stochastic tiling algorithm. [CSHD03]

First a non aperiodic set of tiles, which can trivially tile the plane, is selected. Figure 2.2 shows such a set of tiles. In this set several subsets of tiles, e.g. tile b or f, can tile the plane on their own. Cohen et al. then apply the following procedure to generate Wang tilings.

1. Select any tile from the set and place it in the northwest corner
2. Tile the topmost row from west to east by selecting tiles for which the west edge of the new tile matches the east edge of the previous tile
3. Select a tile for the next row whose north edge matches the south edge from the tile in the row above
4. Continue by randomly selecting tiles for this row whose north edges match the south edges of the tiles from above and whose west edges match the east edge of the tile to the west. This is always possible because the set described by Cohen et al. contains tiles with all necessary NW edge-combinations to this end.
5. Repeat step 3 until the desired amount of rows has been generated

Lagae published a summary of algorithms [Lag09] which generate different forms of Wang tilings. One example are Wang tilings which use color coded corners instead of color coded edges. To create a valid tiling with these corner coded tiles, the corner-color of adjacent tiles has to match.

2.2.2 Application of Wang Tilings

Wang tiles have several interesting applications in computer graphics. This kind of tilings can be used for procedural synthesis of two-dimensional data, like textures or height-fields. One advantage of this method is that it avoids apparent repetitions and artifacts occurring through the periodicity of the texture. This leads to more aesthetic results. Figure 2.3 illustrates this fact by comparing a texture generated with a Wang tiling to periodic texture mapping. The idea to use Wang tilings for procedural texture generation was first presented by Stam [Sta97], who applies this method to create realistic ocean scenes with sea waves and caustics. Stam uses a set of 16 different Wang tiles to this end. Compared to periodic texture mapping, this method produces less visible

artifacts and also conserves GPU memory.
Cohen et al. [CSHD03] devised the stochastic algorithm presented in Section 2.2.1. They show, how their approach, which is based on Poisson distributions, is used to generate large sunflower textures and height-fields. These textures are then combined to render realistic outdoor scenes. Figure 2.4 shows an image generated with this method.
Wei [Wei04] extends the work of Cohen et al. by implementing their algorithm in a GPU friendly approach. Wei's scheme generates textures directly, via fragment- or pixel shader programs.
Kopf et al. [KCODL06] present a technique for rapidly generating large point sets with a blue noise Fourier spectrum and high visual quality. They use a deterministic, Wang Tiling based, algorithm with a constant memory footprint. Their algorithm has applications in global illumination, half-toning, non photo-realistic rendering and other fields.

2.3 Voronoi Diagrams

While Wang tiles and other tessellations use regular polygons to partition the plane, the approach of Voronoi diagrams is different. Voronoi diagrams subdivide the plane into regions based on point distances. While Descartes already had a similar idea in 1644 [Kle04], the mathematical theory behind Voronoi diagrams was first developed and published by the Russian and Ukrainian mathematician Georgy Feodosecvich Voronoy in the early 20^{th} century [Vor08]. This section provides an overview of Voronoi diagrams.

2.3.1 A Partitioning of the Plane Based on Point Distances

The construction of a two-dimensional Voronoi diagram requires a set of points which is located on a single, two-dimensional plane (The extension of Voronoi diagrams to arbitrary dimensions is left out here for the sake of simplicity). Given a set of n (generator) points $S = \{p_1, p_2, p_3, ...\}$ in $\mathbb{R}^2$, Voronoi regions are constructed such that each region is comprised of all points in $\mathbb{R}^2$ which are closer to a point p_i in S than any other point in S [Kle04]. To further define the term "close" a distance measure has to be used. To this end the Euclidean distance, the Manhattan distance or any other distance measure can be applied. The points in $\mathbb{R}^2$ where the distance to both generator points p and q is equal form a border of this Voronoi region. Such a set of points is also called a bisector [Kle04] of p and q and is defined as $B(p,q) = \{x \in \mathbb{R}^2; |px| = |xq|\}$. Each bisector forms two half-spaces and all intersections of the half spaces created by the bisectors of p's surrounding generator points form the Voronoi region $VR(p,S)$ of p. As shown in Figure 2.5, Voronoi regions are either formed by closed polylines or by simple polylines (points that lie on the border of the generating cloud which are not surrounded by other points). The set of all Voronoi regions of the generator points in S forms the Voronoi diagram $V(S)$.

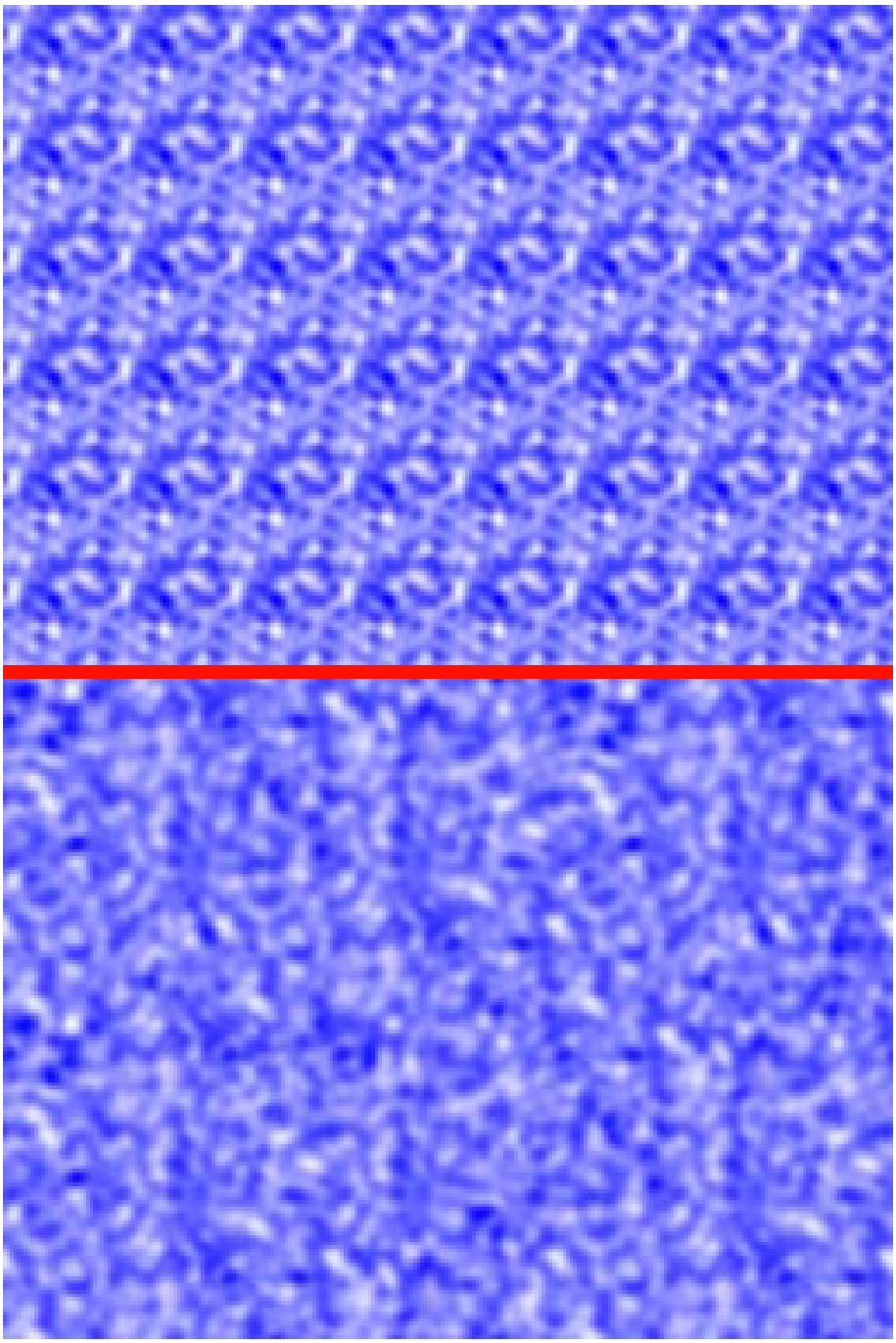

Figure 2.3: Comparison of a texture generated by periodic texture mapping (top) to a texture generated with a Wang tiling algorithm. The texture at the bottom does not suffer from artifacts originating from the periodicity of the texture. [Sta97]

Figure 2.4: An image showing a sunflower field that was created with the Wang tiling algorithm from Cohen et al. [CSHD03]

2.3.2 Delaunay Triangulation: The Dual graph of Voronoi Diagrams

An interesting fact about Voronoi diagrams is that their dual graph is formed by the Delaunay triangulation of the same set of generator points. A Delaunay triangulation for a set of S points in $\mathbb{R}^2$ is defined as a triangulation $DT(S)$, with the constraint that no point of S is located in the circumcircle of any triangle contained in the triangulation $DT(S)$ [Kle04].
A valid Delaunay triangulation $DT(S)$ for the set of generator points in S can be directly obtained from the Voronoi diagram $V(S)$ by connecting all points p and q whose Voronoi regions $VR(p, S)$ and $VR(q, S)$ exhibit a common Voronoi edge in $V(S)$. The inverse of this process, generating a Voronoi Diagram from the Delaunay triangulation $DT(S)$, is also possible. Figure 2.6 shows the Delaunay triangulation of the Voronoi diagram depicted in Figure 2.5.

2.3.3 Construction of Voronoi Diagrams

Besides of computing the Delaunay triangulation for a set of points and then obtaining the corresponding Voronoi diagram as its dual graph, different algorithms which compute

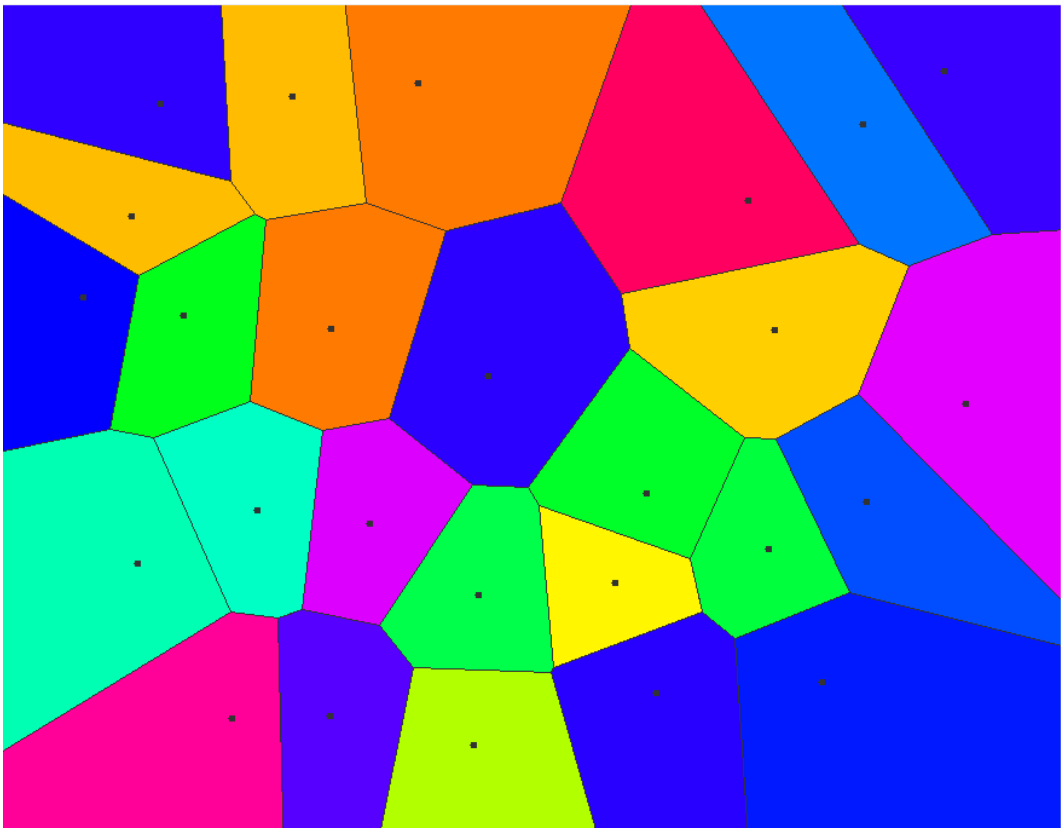

Figure 2.5: A Voronoi diagram. The generator points of the diagram are shown as black dots. Each differently colored cell represents a Voronoi region. Generator points in the outer regions form simple polylines with infinite areas, while points surrounded by a sufficient number of generator points form closed polylines with a finite area. The diagram was generated with a Java applet [Che10].

Voronoi diagrams directly, exist. One popular example is Fortune's Algorithm [For86] which uses a line sweep approach and has a time complexity of $\mathcal{O}(n \log n)$, where n is the number of input points. The algorithm is outlined briefly as follows:

Fortune's algorithm applies a sweep line, which is a line segment, and a beach line, which is a piecewise curve constructed from intersecting parabolas, to construct Voronoi diagrams. Each parabola of the beach line belongs to a specific generator point. The sweep line starts at the top and moves through the point set as depicted in Figure 2.7. While the sweep line progresses through the set of points, the beach line is updated constantly by keeping the base of each parabola halfway between the generator point belonging to the parabola and the position of the sweep line. The points where the parabolas in the beach line intersect construct the Voronoi diagram's edges as the algorithm proceeds. Parabolas are only constructed from points that were already encountered by the sweep

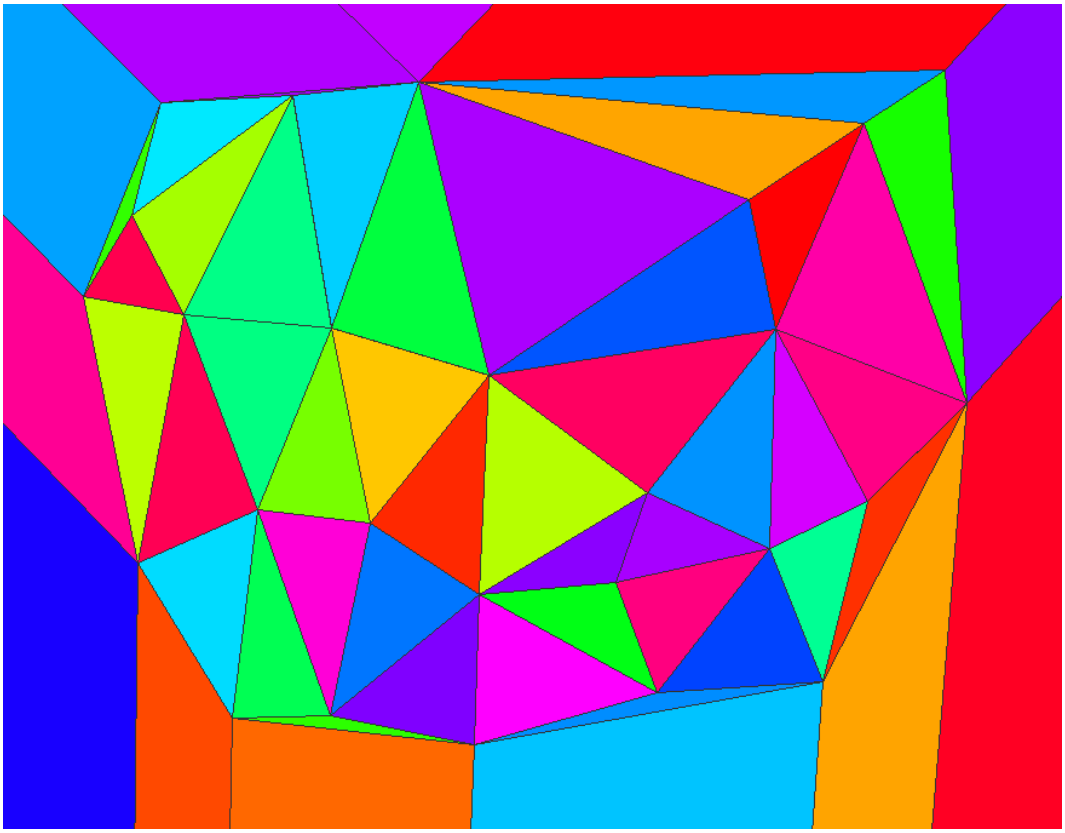

Figure 2.6: An image depicting a Delaunay triangulation. This specific Delaunay triangulation is the dual graph of the Voronoi diagram shown in Figure 2.5. The diagram was generated with a Java applet [Che10].

line. While progressing the algorithm triggers one of two events if the corresponding condition is met:

- Point Event: This event is triggered when the sweep line encounters a new point. This causes a new parabolic arc to appear in the beach line.
- Circle Event: This event is triggered when a parabola shrinks to a point. The according parabolic arc is removed from the beach line and a Voronoi vertex has been found.

The algorithm ends when the sweep line encounters the end of the plane and the whole diagram has been constructed.

Other examples for algorithms that construct Voronoi diagrams are the Divide and Conquer algorithm by Shamos et al. [SH75] and the Randomized incremental construction algorithm by Guibas et al. [GKS90]. Hoff III et al. [HKL$^+$99] present an algorithm

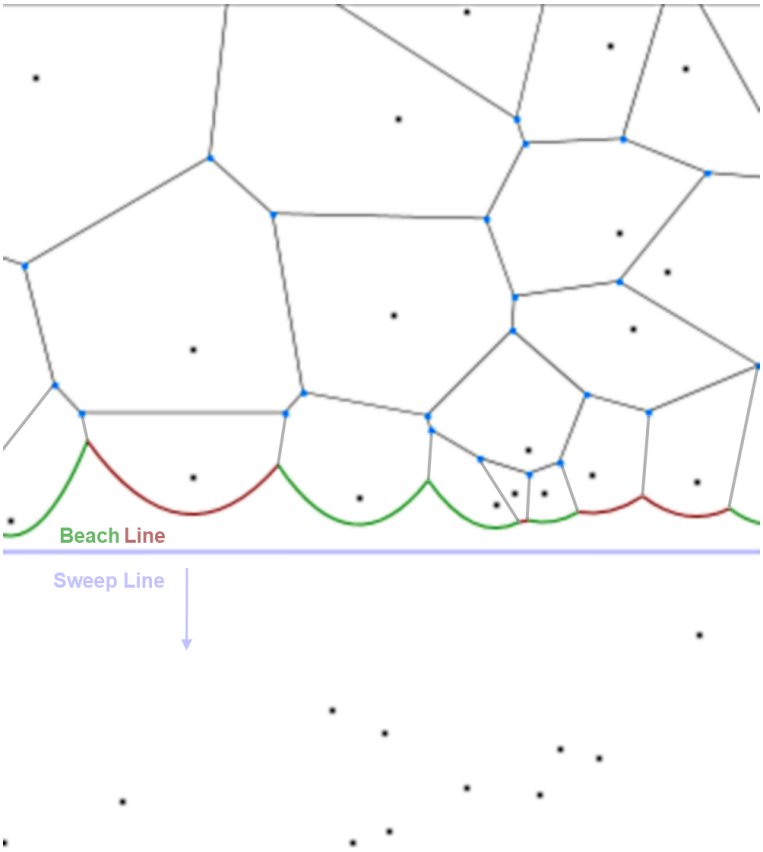

Figure 2.7: A visualization of Fortune's algorithm. The Voronoi diagram is constructed as the sweep line moves through the set of generator points. The beach line is constructed from arcs with bases halfway between their corresponding generator points and the sweep line. Arcs marked in red will collapse in the next iterations and be converted to edges of a Voronoi cell when they disappear. The image was generated with a Java-script applet [Hil].

for the computation of Voronoi diagrams that is based on a Z-Buffer technique and can be implemented directly on graphics hardware. Glanville [Gla04] generate "Voronoi-like" regions (which are not guaranteed to be exact Voronoi regions) with their "Texture Bombing" algorithm. The algorithm is used for procedural texture generation and avoids visual artifacts, similar to Wang Tilings.

2.3.4 Applications of Voronoi Diagrams

Voronoi diagrams are a well researched topic with applications in many fields. A classical problem which is solvable by Voronoi diagrams is the nearest neighbor search: For a set S of points in an n-dimensional space L and a given point l in this space, the closest point to l in S is sought. This is also sometimes called the "post office problem" in two dimensions.
Other applications for Voronoi diagrams are found in biology where they are used to model and analyze the competition of plants, in metallurgy, marketing, robotics and many other fields [Dry15].
In geometric modeling Delaunay triangulations are used for generating surface meshes, since they provide an excellent way for constructing triangulations which avoid narrow triangles. Müller et al. [MCK13] developed an algorithm which is based on Voronoi diagrams for the creation of dynamic fractures. Their algorithm is used to simulate realistically looking, real-time destructions of three dimensional objects in simulations or computer games. Destroyable objects are represented as volumetric approximate convex decompositions, which are Voronoi decompositions of space.

2.4 Hyperbolic Geometry

Hyperbolic space is a non-Euclidean geometric space which is created by replacing the definition of the term "parallel". Two-dimensional, hyperbolic space can be imagined as a saddle-shaped infinite surface in three dimensional Euclidean space (See Figure 2.9). It is not possible to visualize hyperbolic space directly by creating a hyperbolic plane in Euclidean space. Therefore, to visualize geometry located in hyperbolic space, it is necessary to employ a model which maps hyperbolic geometry to Euclidean space. This section provides an overview of two-dimensional hyperbolic geometry, reviews its applications and examines hyperbolic plane tilings.

2.4.1 Overview

Hyperbolic geometry originates from attempts to deduce Euclid's fifth postulate from the remaining four postulates. This was tried by different mathematicians until the end of the nineteenth century. This fifth postulate, which is also called the "parallel postulate" states [SW]:

Given any straight line and a point not on it, there "exists one and only one straight line which passes" through that point and never intersects the first line, no matter how far

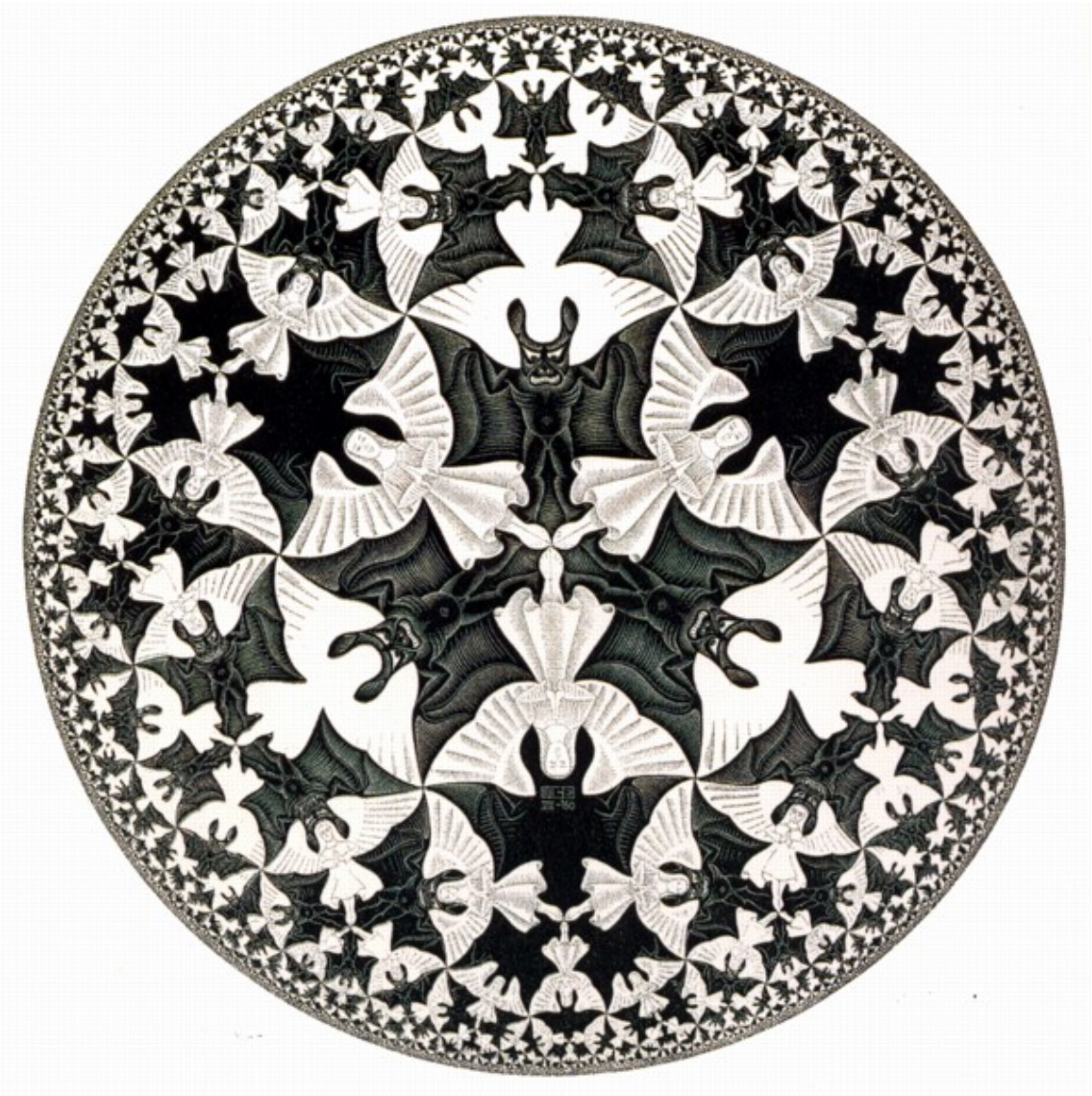

Figure 2.8: M.C. Escher visualized hyperbolic geometry by a woodcarving titled “Circle Limit IV”. The devils and angels have the same size, but since they live in a hyperbolic space, figures located closer to the border appear smaller when they are mapped to Euclidean space. [Esc].

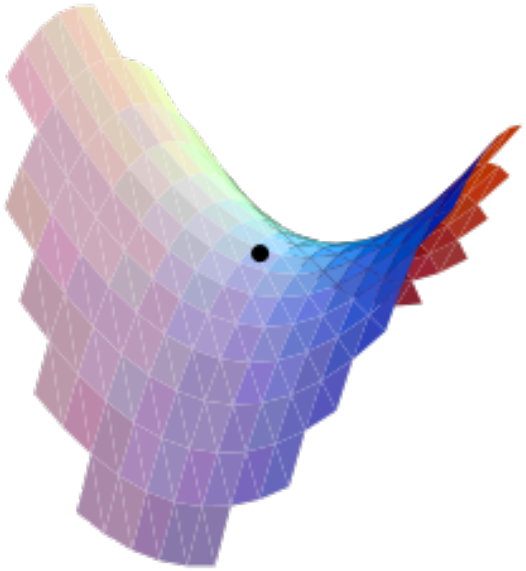

Figure 2.9: In hyperbolic space the environment of each point is saddle-shaped and has a constant negative curvature. [Cla].

they are extended.

None of them ever succeeded. Instead of trying to prove the postulate, mathematicians of the nineteenth century started to examine the consequences of its denial. By a modification of the postulate, Hyperbolic geometry was discovered. The modified fifth postulate states:

Given a line and a point not on it, there is more than one line going through the given point that is parallel to the given line.

This modification has interesting consequences [SW]:

- For any infinite straight line L and any point p not on it, there are at least two infinitely extending straight lines that pass through p which do not intersect L.
- In hyperbolic geometry the sum of all angles in a triangle is always less than 180°.

- Triangles which possess the same angles also possess the same area.
- The sum of angles is not equal in all triangles.
- There are no “similar” triangles as in Euclidean space.

Hyperbolic geometry is characterized by a constant negative curvature and can be imagined as a “saddle” shaped surface, or the infinite surface of a hyperbolic paraboloid. This is depicted in Figure 2.9. To visualize a hyperbolic plane, a model which converts its geometry to Euclidean space has to be applied. Different models have been developed for this purpose: Two examples are the Poincaré Disc Model and the Klein-Beltrami Model. The Poincaré Disc Model represents points in hyperbolic geometry as points in the interior of an n-dimensional unit sphere. Figure 2.10 depicts the projection of a hyperbolic plane to Euclidean space using the Poincaré model. Shapes of the same size appear larger when they are positioned close to the center of the projection, while they appear smaller when they are placed close to the border of the projection.

2.4.2 Uniform Tilings of the Hyperbolic Plane

There exist infinitely many possibilities for tessellations with regular polygons in the hyperbolic plane. This originates from the fact that in hyperbolic geometry the sum of the angles of a triangle is less than 180°. Figure 2.11 shows different hyperbolic tilings.

To visualize uniform hyperbolic plane tilings the General Replication Algorithm from Durnham [Dun07] can be applied. It starts with multiple, textured, convex polygons referred to as the fundamental regions. To construct a tiling it transforms copies of these regions in the hyperbolic plane, a process called ”replication“ by the author of the algorithm. The algorithm was inspired by Escher’s woodcarvings titled “Circle Limit”. Figure 2.8 depicts “Circle Limit IV”.
The steps which lead to the creation of a tiling with this algorithm are outlined briefly as follows.:

First the central layer, the so called p-gon pattern, is created by reflecting and rotating one of the initial fundamental regions according to its reflection symmetries. The algorithm starts with copies of the inner fundamental regions and then continues recursively with the outer layers. The recursion is terminated after a predefined recursion depth has been reached. To fill the complete hyperbolic plane the recursion depth would have to be infinite.

2.4.3 Applications for Hyperbolic Geometry

Hyperbolic geometry and its visualizations were used for artistic purposes like in M.C. Escher’s woodcarving “Circle Limit IV”, which is depicted in Figure 2.8. Lamping et al. [LRP95] were also inspired by this painting and created “a technique for visualizing

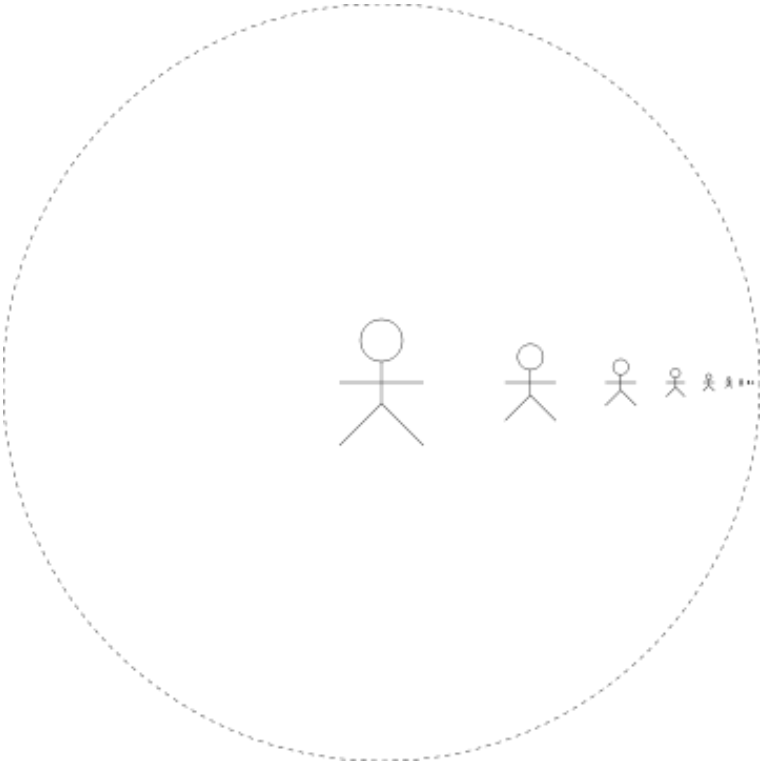

Figure 2.10: A visualization of hyperbolic space with the Poincaré Disk Model. The visualization is distorted. In hyperbolic space all figures possess the same size and the distance between them is equal. The dashed edge of the disk represents infinity. [Cla].

large hierarchies" which exploits the properties of hyperbolic space. To explore hierarchical graphs like directory trees or XML-files they perform the layout of the graphs, which represent the data, in hyperbolic space. The advantage of this concept is an increased perception of the node's environments, which preserves navigational context while still providing an overview of the whole tree. Margenstern [Mar09] presents ideas for applications based on $\{5, 4\}$ pentagonal grid and $\{7, 3\}$ heptagonal grid tilings. Two of Margenstern's ideas are a pentagonal grid for keyboard layouts and a hyperbolic color chooser based on a heptagonal grid.

2.5 Fractals

The term fractal characterizes a family of shapes which can be found in nature or are specified by mathematics and algorithms. A common feature of all fractals is their detailed self similarity and scale invariance. Figure 2.12 depicts an example for a fractal in 3-D. fractals exist in many different shapes and variations. This chapter provides only

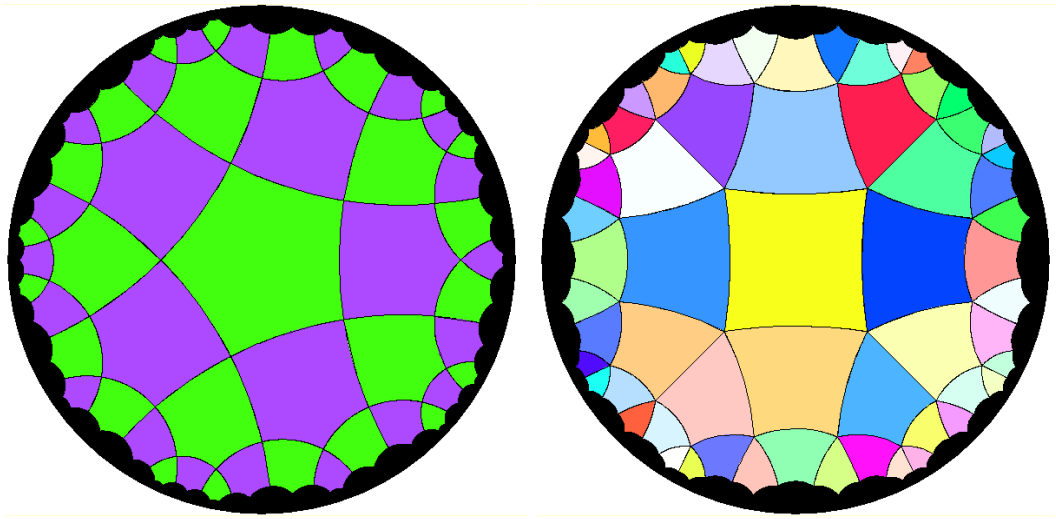

Figure 2.11: Two visualizations of uniform Hyperbolic plane tilings. The tilings are denoted by the Schläfli symbols {5,4} and {4,5} (from left to right). Please note that there are infinitely many possible uniform tilings of the hyperbolic plane. The images were generated with a Java applet from [Joy].

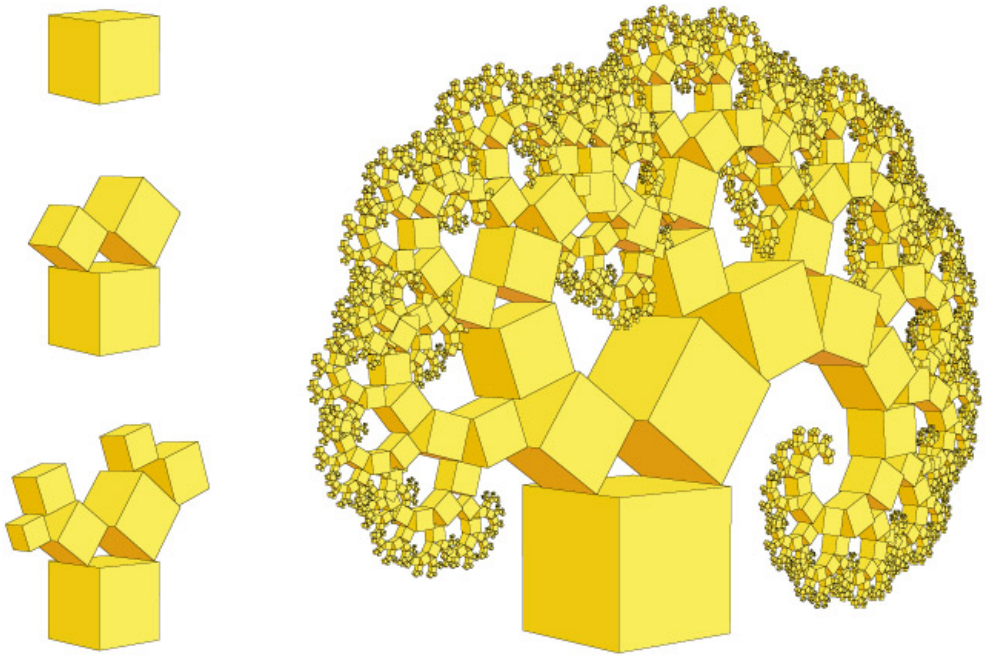

Figure 2.12: The Pythagoras tree fractal. This fractal is composed of self similar shapes, which are tree look-a-likes. [Bro].

a short overview on the topic of fractals. The comprehensive work "The fractal geometry of nature" by Benoit B. Mandelbrot [Man82], who coined the term fractal, offers a much deeper insight into this topic. Mandelbrot was one of the first scientists who employed computer graphics to create beautiful visualizations of fractals.

2.5.1 Overview

The term fractal can be traced back to Bernard Bolzano (1830) or Bernd Riemann (1861) who were already researching fractal curves but did never publish their work [Pic09]. About a century later Karl Weierstrass defined a continuous mathematical function which was nowhere differentiable and would nowadays be considered as a fractal curve. Other noteworthy mathematical contributions to the field of fractals were later given by Helge von Koch (Koch Kurve), Georg Cantor (Cantor Set), Gaston Julia (Julia Set) and several others. However, before Benoit B. Mandelbrot published his famous book, visualizations of fractals were only existent in the form of simple hand drawings. Mandelbrot became popular by applying computer graphics to visualize fractals. He also provided a thorough definition of the term "fractal" and discovered visualizations of the popular Mandelbrot Set. This set, which is a fractal itself, serves as a map for other polynomial fractals (Julia and Fatou Sets).
Keneth Falconer, a mathematician specialized in the research of fractal geometry, identifies fractals as mathematical structures which feature the following attributes [Fal03]:

- Self similarity: Fractals exhibit self similarity on different scales.
  - Exact self-simliarity: The fractal is identical at all scales.
  - Quasi self-similarity: The same pattern is approximated at different scales. The fractal can contain copies of itself at different scales.
  - Statistical self-similarity: A pattern is repeated stochastically, by preserving statistical measures.
- Non differentiable: Fractals are nowhere differentiable.
- Fractal dimension: Fractals do not necessarily need to have a dimension that is an integer value (1-D, 2-D, 3-D,...). The dimensions of fractals are measured differently (e.g., with the Box Counting dimenison or Information Dimension) and also serve as a measure of the fractal's space filling quality.
- Recursive definition: Fractals can be defined by recursive functions.

2.5.2 The Sierpinski Gasket

The Sierpinski Gasket was discovered by Waclaw Sierpinski in 1915 and serves as a simple example for the generation of fractals in this section. The following recursive algorithm produces a Sierpinski Gasket. The number of recursions defines the degree of self similarity (Higher recursion depths produce more self similar details):

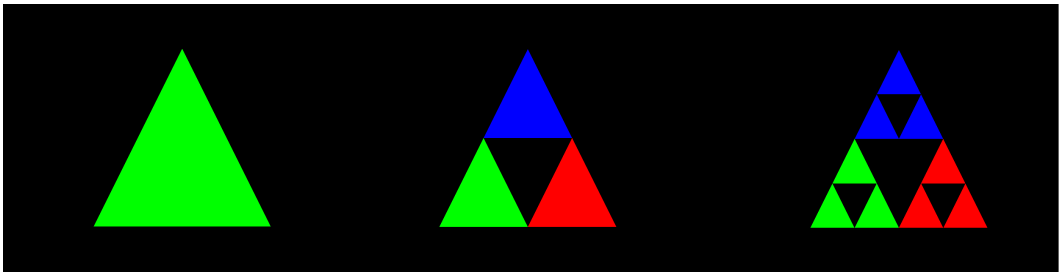

Figure 2.13: The first three steps of a recursive algorithm that generates a Sierpinski Gasket. The green starting triangle is replaced by 4 small equal-sized versions of the original triangle and the center triangle is removed. If this step is repeated recursively with each triangle, the algorithm generates a Sierpinski Gasket.

1. Draw the three sides of the initiator triangle.
2. Create four new, equally sized triangles by connecting the centers of the triangle's sides.
3. Remove the center triangle.
4. Apply step 2 on the remaining three triangles.

Figure 2.13 shows the first three steps of this algorithm. However, a variety of algorithms which create Sierpinski Gaskets exist. For example these fractals can also be generated by applying a stochastic algorithm. Figure 2.14 shows several levels of a Sierpinski Gasket generated with a stochastic algorithm.

2.5.3 Construction of Fractals

This section presents different methods for the construction of fractals. Popular methods include iterative/recursive construction, the application of statistical methods and the utilization of formal grammars to generate fractals. Some of these methods are outlined in the following sections. Interestingly, as shown in Section 3.5, slight variations in the definition of the Inductive Rotation Method also result in fractal structures.

Iterated Function Systems

Fractals may be created by recursively applying a set of functions to a point or a geometric shape in a metric space. Such a system, based on a set of functions, is called an Iterated Function System, short IFS. Two algorithms based on IFS are described in this section. A more formal definition of Iterated Function Systems is given by expression 2.1.

$$I = \{\mathbb{X}; f_n, n = 1, 2, ..., N\} \tag{2.1}$$

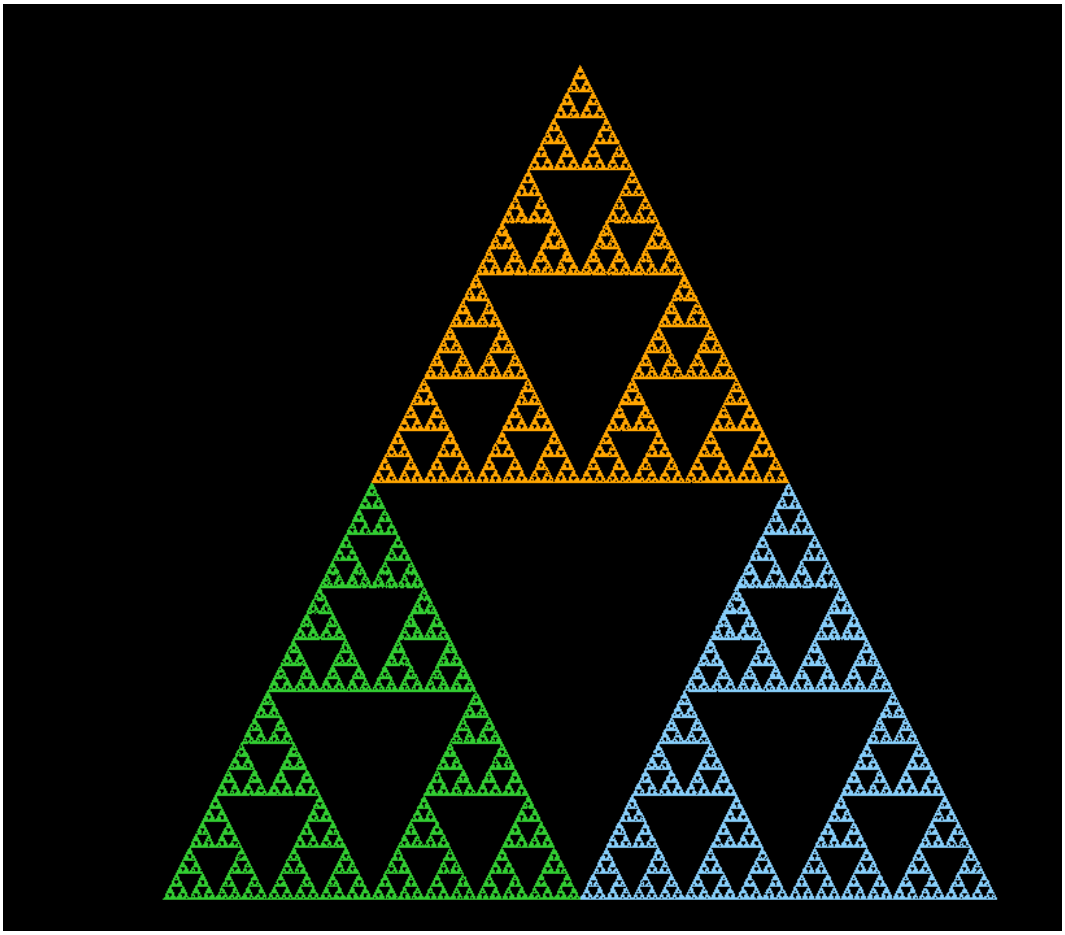

Figure 2.14: A Sierpinski gasket generated by the application of a stochastic algorithm.

In expression 2.1 $\mathbb{X}$ is a complete, metric space and each $f_n : \mathbb{X} \to \mathbb{X}$ defines a contraction mapping. This definition for IFS was proposed by Barnsley [Bar88]. To construct a fractal with an IFS, another operator is necessary [Hut81].:

$$W(A) = \bigcup_{i=1}^{n} f_i(A) \tag{2.2}$$

The operator $W(A)$ defined in expression 2.2 is called the Hutchinson Operator.The Hutchinson Operator decodes fractals when applied recursively. Starting with $A = S_0$ each successive S_i is computed as:

$$S_{n+1} = W(S_n) = \bigcup_{i=1}^{n} f_i(S_n) \tag{2.3}$$

A feedback system like this can be implemented in various ways. A popular algorithm for decoding fractals is called “The chaos game”, introduced by Barnsley [Bar88]. The

algorithm starts by picking a random point p_0 in the plane. Then, a function from the set of functions $f_n \in I$ is selected randomly and applied to the point p_0. This generates the next point p_1. Another function from the set is then randomly chosen to compute p_2. After repeating this process for an arbitrary number of iterations the resulting points approximate the fractal.
An Iterated Function System can also be represented as a graph of affine matrices. Figure 3.5 depicts the graph scheme of an IFS with two different initial matrices.

Iteration of nonlinear Functions

Julia sets are generated by applying rational or polynomial functions iteratively. These sets were first discovered by Gaston Julia and Pierre Fatou while examining complex dynamic systems and their behavior. Such systems possess the form $\{\mathbf{C}, f\}$ where f is a rational or polynomial function. To construct a Julia set the function is applied iteratively as shown in Expression 2.4. Not all dynamic systems of this form generate fractals. For example the system $\{\mathbf{C}, z^2\}$ yields the unit circle. However, many dynamic systems of the form $\{\mathbf{C}, z^2 + c\}, c \in \mathbf{C}$ produce fractal sets.

$$z \longmapsto f(z) \longmapsto f(f(z)) \longmapsto ... \tag{2.4}$$

The Mandelbrot set is constructed by applying Expression 2.5. It is the set of complex numbers that does not approach infinity when applying 2.5. An interesting fact about the Mandelbrot set is that it represents a map of all connected Julia sets. Figure 2.15 depicts the Mandelbrot set.

$$z_{n+1} = z_n^2 + c \tag{2.5}$$

For the algorithmic construction of the Mandelbrot set each pixel of a 2-D image is mapped to a complex number c, by using the x and y screen coordinates of each pixel as a and b in $c = a + bi$. The program then iteratively computes the formula $z = z^2 + c$ and determines if the resulting complex number is part of the set. This is done by checking the condition $|z| > 2$ after several iterations, for each pixel. If the pixel is not part of the set the condition evaluates true. The resulting pixels can also be colored according to the iteration where this condition was fulfilled for the last time to get a more visually appealing representation. The same algorithm may also construct Julia sets by choosing different Expressions to compute z and c.

L-Systems

L-Systems, are named after the Hungarian Biologist Aristid Lindenmayer. Lindenmayer developed these systems for modeling the behavior of plant cells and their growth. Lindenmayer Systems are parallel rewriting systems that operate on a string of symbols. The symbols in an initial string are replaced in several derivation steps. In contrast to Chomsky-Systems, L-Systems replace all symbols simultaneously in each step. An L-System is defined as the triplet $G = (A, \omega, P)$, where:

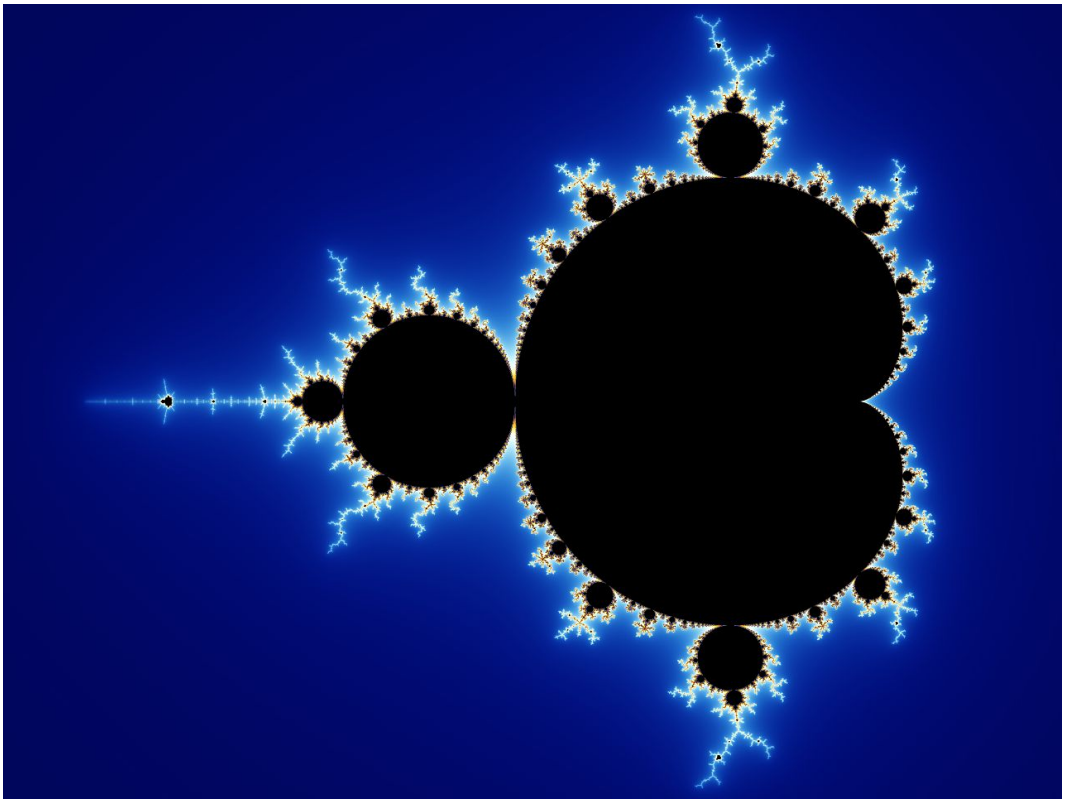

Figure 2.15: A visualization of the famous Mandelbrot set. This fractal was discovered by Benoit B. Mandelbrot and represents a map of all connected Julia sets. [Bey].

- A is a set of symbols, the alphabet.
- ω is a symbol, or string of symbols defining the initial state of the system.
- P is a set of production rules that define how the symbols in the current string will be replaced. Production rules have the form $(\delta \rightarrow \phi)$. With the meaning that the symbol δ is replaced by the set of symbols ϕ in the current replacement step.

L-Systems can also be used to create various fractals. Turtle graphics provides an option to obtain a graphical representation from a string generated by an L-System.
Turtle graphics uses a relative cursor, the turtle, which accepts different commands and has three attributes:

- A location (X,Y) in the plane.
- The orientation of the turtle. Movement is performed relative to the turtles orientation.
- A pen color.

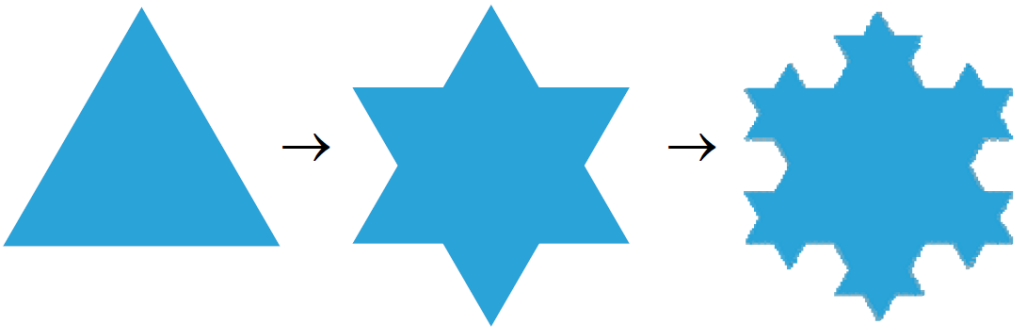

Figure 2.16: Three iterations of an L-System creating a Koch snowflake fractal. The image to the left shows the graphical representation of the initial string while the other two images show the result after one and two derivation steps. [Tra]

These attributes are represented by a triple of the form $(P(x, y), \rho, c)$, where $P(x, y)$ is the position of the turtle on the plane, ρ is the rotation angle and c is the color of the pen, which draws a line as the turtle moves on the plane. To render L-Systems with Turtle graphics, the alphabet of an L-System is simply mapped to turtle commands for movement and rotation. A detailed example, for constructing a Koch curve with Turtle graphics, is given in the following paragraph.
Consider the following L-System described by expression 2.6.

$$G = (\{F, -, +\}, F - -F - -F, \{(F \to F + F - -F + F)\}) \tag{2.6}$$

The first two derivation steps yield the following strings:

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

The symbols of each of these iterations are then mapped to drawing commands for the turtle. F is mapped to the command "draw a line of length d", where the parameter d depends on the current iteration number. The symbols $+$ and $-$ are mapped to the commands "rotate by 60° to the left/right". Figure 2.16 shows the result of mapping the first three strings to turtle commands.

2.5.4 Applications of Fractals

Fractals are popular amongst digital artists who use them for the creation of fascinating art. Besides from their use in arts, there are several other interesting applications of fractals. In construction, engineers use cables with a fractal arrangement of fibers, which

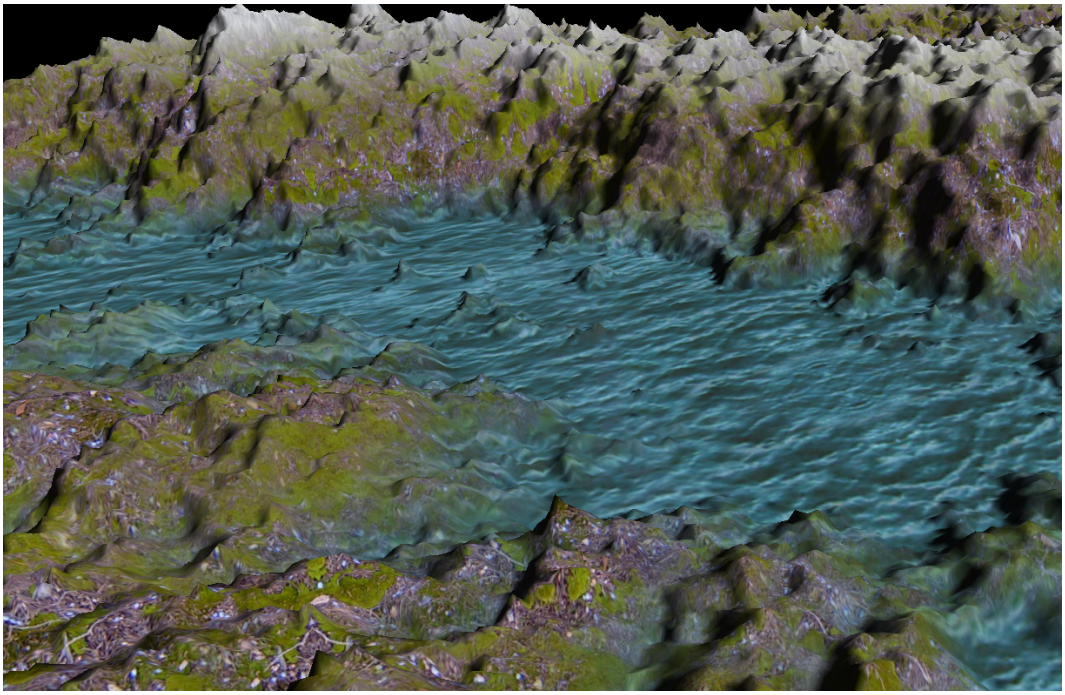

Figure 2.17: A fractal landscape that was generated using the Diamond Square algorithm. This algorithm for the creation of fractal landscapes is used by different terrain generation frameworks.

greatly increases the stability of the cables. This technique is employed for the production of steel cables which are used to construct bridges, such as the golden gate bridge [Fou]. In Electronics, Sierpinski Gasket shaped antennas have proven useful through their ability to receive and transmit over a big band of frequencies. This facilitates the construction of more powerful and compact antennas. In computer science applications for fractals range from compression algorithms to the generation of cities and virtual landscapes. Figure 2.17 shows a fractal terrain generated by the “Diamond Square algorithm”.

CHAPTER 3

Hofstetter's Inductive Rotation

The Inductive Rotation Method, which was developed by the artist and mathematician Hofstetter Kurt, defines an algorithm for artistic pattern generation. The method produces aperiodic patterns, which are located in a 2-D plane and which are composed of multiple instances of a prototile. Figure 3.1 depicts an example of an Inductive Rotation Pattern which was created by Hofstetter. The structure of these patterns does not correspond to the definition of mathematical tilings as described in Section 2.1, since there

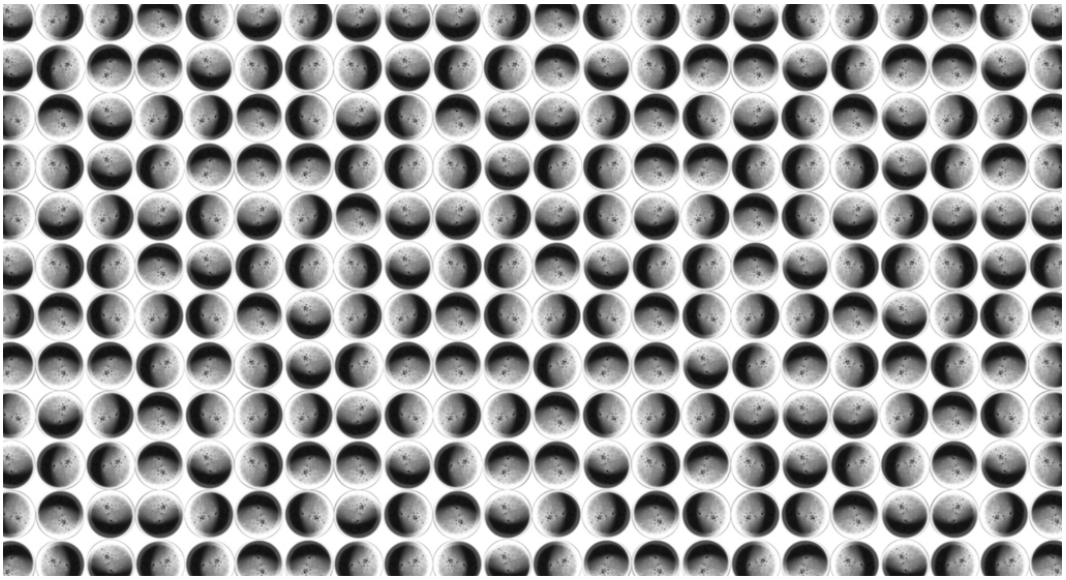

Figure 3.1: An Inductive Rotation Pattern created with the 3-way Method by Hoffstetter Kurt. This specific pattern is titled "CoffeeC" and was generated with a previously developed IR software package. Image courtesy of Hofstetter [Hofb]

are overlapping regions between the tiles composing a pattern. This chapter provides a thorough definition of the Inductive Rotation Method and is structured as follows:

The first section describes the rules for defining Inductive Rotation Patterns. The second section gives an overview of the goals of this thesis, i.e., the required features for the new, so called, Inductive Rotation Framework. The next Section,3.2, summarizes previously developed algorithms for the creation of Inductive Rotation Patterns [Par13]. Two new algorithms for creating Inductive Rotation Patterns, an algorithm that computes Inductive Rotation Patterns in parallel and an algorithm that applies tile substitution to create Inductive Rotation Patterns, are described in Section 3.4. Finally, the last part of the chapter presents new ideas for possible applications of the Inductive Rotation Method.

3.1 Method Description

To create artistic visualizations Hofstetter experimented with different methods to create patterns by superimposing tiles in the plane, according to an iterative scheme. In one of his experiments the Inductive Rotation Method was discovered. The Inductive Rotation Method is a procedure for pattern generation. The procedure recursively copies and rotates a prototile to create a set of partially overlapping tiles, the Inductive Rotation Pattern. The number of rotations is selected according to one of three rules defined by Hofstetter. Independent of the rule, the angle defining the rotations is always equal and their sum is always 360°. Hofstetter defines three different rules for creating Inductive Rotation Patterns:

- The 3-way Inductive Rotation Method performs three rotations with a quad-shaped prototile.
- The 5-way Inductive Rotation Method performs five rotations with a hexagonally-shaped prototile.
- The 2-way Inductive Rotation Method performs two rotations with a star-shaped prototile. (The proportions of the star are depicted in Figure 3.2)

To create a pattern one of these rules is applied iteratively for an arbitrary number of iterations. The Inductive Rotation Method always starts with a single prototile. In each iteration all tiles created in the previous step are copied and rotated around a pivot, according to the number of rotations for the selected rule. The same procedure is performed for an arbitrary number of iterations with different pivot points (in each iteration) to create the final pattern. Figure 3.3 depicts a pattern which was created by applying the 3-way scheme.

Hofstetter defines the pivots for the 5-way and 3-way Methods as the rightmost point on the x-axis which is also contained in the pattern. This definition causes the pivot's x-coordinate to grow exponentially in each iteration. The pivot for the 2-way rotation

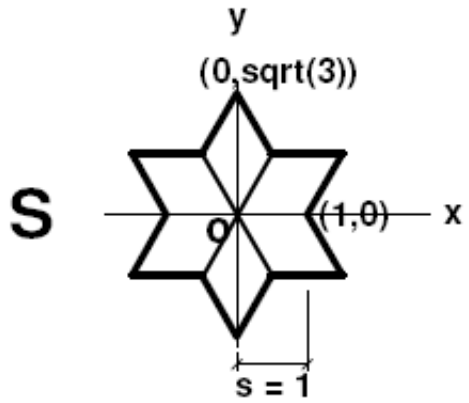

Figure 3.2: A figure showing the position of the first pivot for the creation of 2-way Inductive Rotation Patterns. The image also depicts the proportions of star-shaped prototiles defined by Hofstetter. Image courtesy of Hofstetter [Hofa]

scheme is defined differently: In the first iteration the rightmost vertical midpoint of the prototile is selected as pivot (the corner-vertex of the star between the two apexes pointing to the right. See Figure 3.2). The pivot selection procedure is similar for the following iterations, but instead of the prototile, the rightmost tile in the pattern is used for pivot selection. In situations with an ambiguity, i.e. when there are multiple rightmost tiles, the tile with the lowest y-coordinate is chosen. Hofstetter provides a more detailed definition of the Inductive Rotation Method [Hofb, Kur].

The Inductive Rotation Method was already described in detail by Parzer [Par13]. This thesis uses a slightly different notation which extends the definition of the Inductive Rotation Method from 2-way, 3-way and 5-way patterns to more general n-way patterns that are defined by 3 different parameters. Using this definition patterns like the Sierpinski Gasket can be created. A fact which clearly shows the affinity of the Inductive Rotation Method to fractals.

From a mathematical point of view there are different perspectives from which Inductive Rotation Patterns can be viewed. Parzer stores the center coordinates of each tile in a list and uses a list of integers to represent the rotation angle of each tile:

$$r(k) = k * \frac{2 * \pi}{p_r} \tag{3.1}$$

In Expression 3.1 p_r represents the method-specific rotation number, $r(k)$ evaluates the rotation angle of a tile in degrees and i with $0 \leq i < p_r$ is an integer which represents the rotation of the tile. In Section 3.4.1 a different representation, which defines patterns as chains of matrix transformations, is introduced. Another different mathematical interpretation of Inductive Rotation Patterns was recently developed by Frettlöh

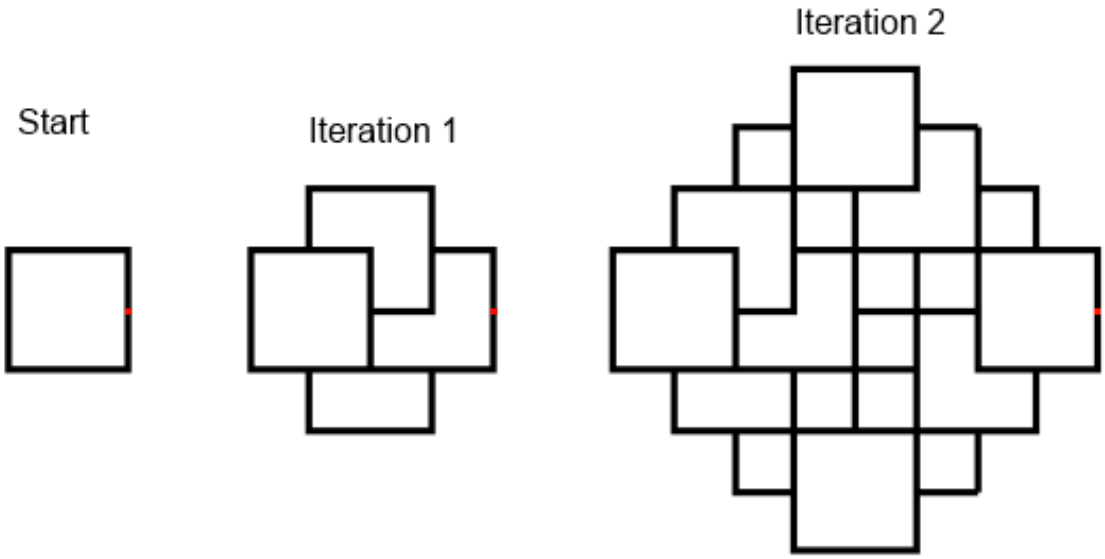

Figure 3.3: Several figures representing the necessary steps to create a 3-way Inductive Rotation Pattern. A square shaped tile is copied three times and rotated around the pivot which is marked as a red dot at the right side of each shape. This produces the first iteration step (Shape 2 from the left). To generate another iteration-step the procedure is repeated with the whole pattern and a new pivot (which is again positioned at the rightmost point in the pattern). In each step of the procedure the new shapes are placed behind the previously created shapes. Original Image courtesy of Hofstetter [Hofa]

and Hofstetter [FH15]. They apply substitution tilings with four different rules to create Inductive Rotation Patterns. The algorithm in Section 3.4.2 is based on these findings.

To establish a consistent notation, the following parameters are used to define Inductive Rotation Patterns throughout this document. The first parameters define the position and orientation of each tile in the pattern:

- The method-specific rotation number p_r. This is the number of rotations performed per iteration plus one for each method. (e.g., for 2-way rotation $p_r = 3$)
- The strategy for choosing the rotation's pivot in each iteration. This strategy is expressed as a vector-valued function with the current iteration number as parameter: $\vec{p}_{pivot}(l) = (f(l), g(l))$, where f and g are functions which evaluate the x and y coordinate of each pivot. The exact definitions for these functions are provided in the next sections.
- The number of iterations: p_i

The other parameter defines the shape of each prototile.:

- A list of points defining the geometric shape of the prototile's polygon p_{shape}.

3.2 Previous Work

This section summarizes algorithms for the creation of Inductive Rotation Patterns which are the results of previous research. Two algorithms for the creation of Inductive Rotation Patterns, a grid based and a sprite based algorithm, were developed by Parzer [Par13].

3.2.1 Grid Based Approach

The grid based approach interprets the Inductive Rotation Method in 3-D. To create an Inductive Rotation Pattern with the grid based approach, Parzer [Par13] represents the pattern as multiple, layered 2-D grids. Each tile is represented as multiple numeral entries in one or more layers of the grid. For example when applying the 3-way Inductive Rotation Method, each tile is represented as four numbers in a grid layer. Hidden parts of the pattern, which result from overlapping tiles, can be stored in a different grid-layer. Parzer's software also allows the artist to visualize each layer independently.
The algorithm creates the pattern iteratively. When an iteration level is requested, a procedure iterates over the numbers that represent the stored tiles in the 2-D grid. For each number in the grid, that represent a tile-part, the procedure determines a target position in the grid and writes the numbers, that define the rotated tile-part, to this position. To find the number and position that define the rotated tile-part Parzer applies multiple expressions [Par13].
The number of possible grid entries depends on the selected method. The 3-way Inductive Rotation Method, for example, requires four different rotations per sub square which

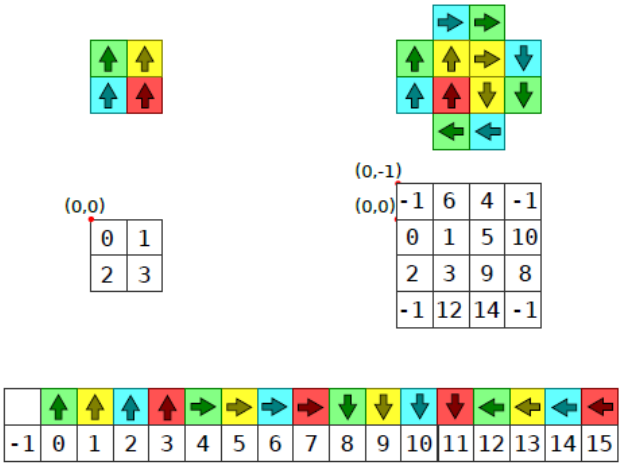

Figure 3.4: Visualization of the indexing scheme for the grid based algorithm. The left side of the figure shows the initial grid with only a single prototile analogous to the first iteration in Figure 3.3. The prototile is represented as four numbers which represent the four portions of the prototile's texture and their rotation. The bottom image shows all possible grid entries and how these grid numbers are mapped to the initial texture. The right side of the figure shows the grid after one iteration. The resulting numbers represent the four prototiles of the first iteration of a 3-way Inductive Rotation Pattern analogous to second iteration in Figure 3.3. [Par13].

leads to a total of 16 possible integer entries (one for each rotation and sub square).
To render the pattern, the grid representation of the tiles is converted to a coordinate based representation. The bottom image in Figure 3.4 depicts all possible grid entries for the 3-way Inductive Rotation Method. The upper images show the 0^{th} and the 1^{st} iteration steps for the creation of a 3-way IR-Pattern with the grid based algorithm. The number -1 represents empty cells. Parzer implemented different versions of this approach to deal with the creation of 2-way and 5-way patterns. While the 3-way pattern uses a rectangular grid, the versions used to create 5-way and 2-way patterns operate on a regular grid of triangles.

3.2.2 Sprite Based Approach

The grid based approach has several drawbacks which result in rendering artifacts for some prototiles and limits the prototile shape to hexagons stars and squares [Par13]. These facts led to the development of the sprite based approach which also excels the grid based approach in speed and memory usage.
To create an Inductive Rotation Pattern with the sprite based approach [Par13] two linked list data structures are defined. The first structure contains the rotation of each tile

of the pattern as an integer, corresponding to Expression 3.1. The second list contains the center coordinates of each tile. When the algorithm starts the structures are initialized with the rotation and position of the prototile. To generate the next iteration level the algorithm copies and rotates all tiles contained in the pattern according to the selected rule and merges the copies into the existing data structures in a back to front order. This scheme works similarly for all Inductive Rotation Methods (2-way, 3-way, 5-way). Parzer's sprite based approach does not directly support rendering hexagons or stars. Hexagon and star shapes can only be achieved through the use of alpha-transparent prototile textures. The idea of the sprite based approach can be extended and parallelized as presented in Section 3.4.1.

Computing Rotation Pivots

To compute the positions of the 3-way and 5-way rotation pivots, Definition 3.2 is used (The original definition by Parzer is slightly different but omitted here for the sake of a constant notation). For the definition it is also assumed that the width and height of each tile is two units and that the prototile's center is located at the coordinate $(1, 1)$.

$$\vec{p}_{pivot}(l) = (2^l, 1) \tag{3.2}$$

The scheme for determining the position of the pivot for the 2-way Method is different. To find the pivot-positions for the 2-way method, Parzer uses a simple search procedure that locates the tile with the maximum x and minimum y coordinate as defined by Hofstetter (Section 3.1).

3.3 Requirements for the New Framework

While the "Irrational Image Generator" already provides sufficient visualization possibilities for IR-Patterns, the software still lacks tools for an efficient and convenient artistic work flow. Firstly, to improve Hofstetter's work flow, the artist needs an integrated prototile editor, which allows Hofstetter to directly manipulate the prototile texture and then updates the resulting pattern instantly. Secondly, the existing algorithms for pattern generation can only produce patterns with a limited size. Thirdly, Hofstetter expressed interest in experimenting with the new tile substitution method from Frettlöh and Hofstetter [FH15]. The following requirements were developed together with Hofstetter in a first meeting where the implementation goals of the Inductive Rotation Framework were coarsely defined:

- The program should be able to produce 2-way, 3-way and 5-way Inductive Rotation Patterns.
- The artist needs an integrated prototile editor which allows him to create prototile sketches and instantly updates the Inductive Rotation Pattern after editing (e.g., after drawing a line).

- The framework should provide a stable algorithm which allows the artist to produce large patterns.
- The framework should contain a tile substitution algorithm, which is based on the research of Frettlöh and Hofstetter [FH15].
- The software should have a user-friendly GUI to control all functions of the improved and extended work flow.

3.4 Algorithms

This section provides an overview of all new algorithms, implemented in the Inductive Rotation Framework. The idea of the first algorithm is to parallelize pattern generation. This can be achieved by defining a scheme that computes all tile coordinates independently. The second algorithm implements the newly developed substitution method by Frettlöh and Hofstetter [FH15].

3.4.1 Parallel Approach

To parallelize the generation of Inductive Rotation Patterns it is necessary to develop a scheme for an independent computation of a pattern's tile coordinates. The parallel approach achieves this by applying a modulo based matrix indexing scheme. The algorithm expresses each tile's coordinates as a function of a positive integer index from $\mathbb{N}^0$ which enables to parallelize the tile generation algorithm. Two different implementations of this algorithm are provided in Section 4.3. The first implementation is a simple, parallel approach which uses a Direct X 11 Direct Compute program to determine each tile's coordinates directly on the GPU. The second implementation splits rendering into multiple passes which reduces the memory consumption of the algorithm and also allows the artist the generation of larger patterns. The algorithm's performance scales with the quality of the graphics hardware and provides many possibilities for extensions and improvements.

Concept

The algorithm's concept is easily explained by comparing it to the procedure of generating fractals with Iterated Function Systems (IFS). To generate the polygons for an iteration of an IFS, multiple series of transformation matrices are applied to copies of an initial polygon. A graph based scheme can be used to generate the transformation matrices. The scheme for an IFS with two matrices is depicted in Figure 3.5. To generate the points of the fractal for the 0^{th} iteration, the starting polygon's points are simply multiplied with the identity matrix. To generate the points for the 1^{st} iteration, all points of the initial polygon are copied and multiplied with the initial matrices of the IFS B_{11} and B_{12}. In each subsequent iteration, in order to find the matrices for level l, all possible matrix combinations b^l of the iteration level $l-1$ with the b initial matrices of the IFS

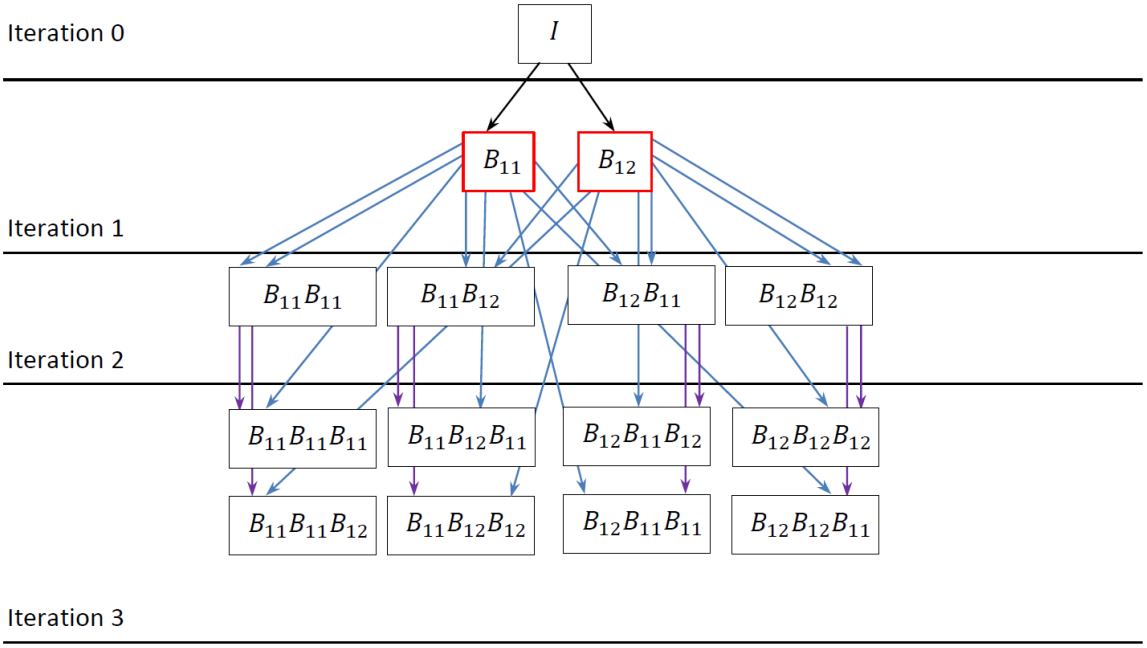

Figure 3.5: Visualization all possible matrix combination for an IFS with two functions. To generate the points for an IFS-iteration step the initial points are copied and transformed by each combination of matrices of a certain iteration step. The Figure shows the initial matrices with a red border. The arrows between the boxes that contain the composite matrices show how they are constructed. Arrows originating from matrix combinations of the same iteration are colored similarly. The resulting structure is a hierarchical graph.

are formed. For the second level of the depicted IFS this would for example yield the matrices: $B_{11}B_{11}, B_{11}B_{12}, B_{12}B_{11}$ and $B_{12}B_{12}$.To create an approximation of a fractal from one or more generator points, all combinations of matrices of a single iteration level are multiplied with the generator points.

The rotations which are needed to place a tile at a certain point in an Inductive Rotation Pattern can also be seen as a series of transformation matrices. Expression 3.3 shows a possible combination of matrices to compute the center point of the tile c_{tile}, by applying a series of matrix multiplications to the generator point, c_{init}. Each transformation matrix is identified by two indices l and k. The index l represents the iteration level and k selects one of a constant number $(p_r - 1)$ of rotation matrices.

$$c_{tile} = c_{init}^{T} B_{31} B_{21} B_{11} ... B_{lk} \tag{3.3}$$

While the basic concept of using a chain of transformations for pattern generation is similar to IFSs, the Inductive Rotation Method uses different graphs to produce patterns which are further referenced to as Inductive Rotation Graphs by this thesis. In contrast to

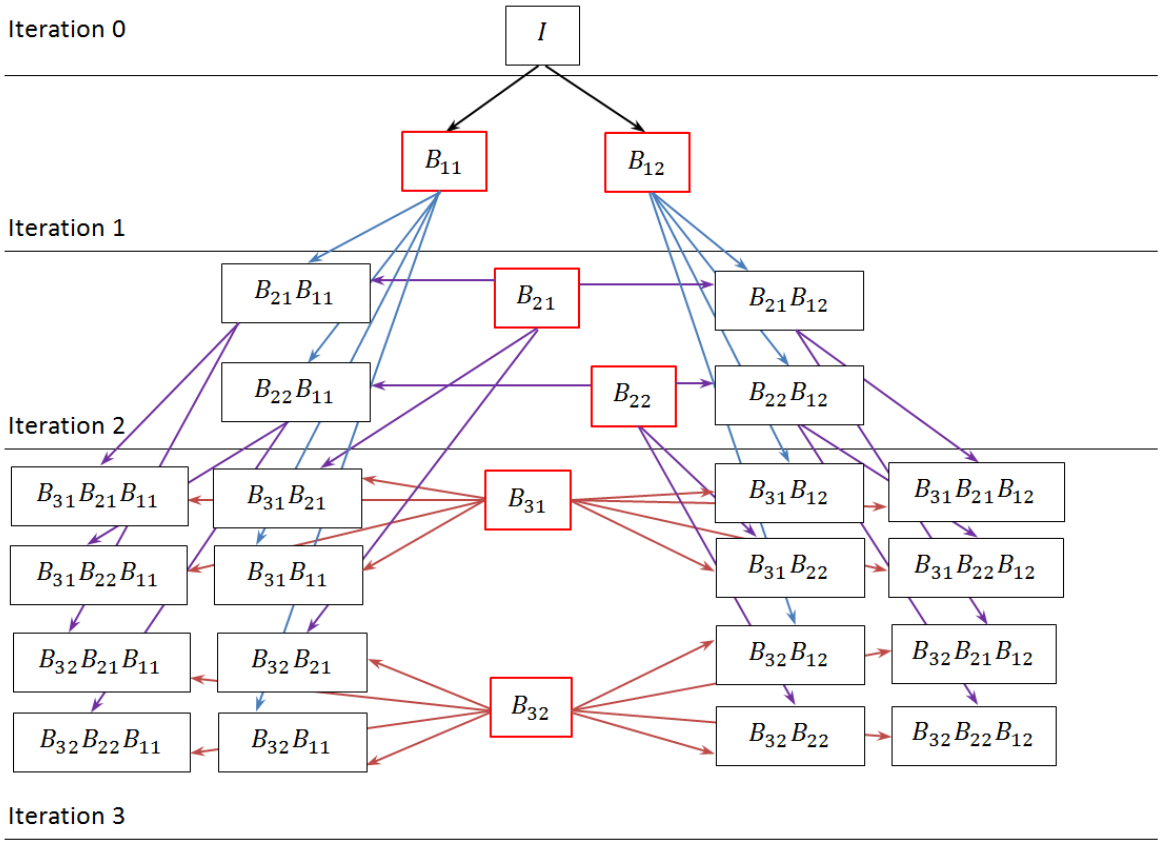

Figure 3.6: Visualization of all possible matrix combinations for a 2-way Inductive Rotation Graph with two initial matrices. In contrast to an IFS the Inductive Rotation Pattern is generated by copying and multiplying all initial points with all matrix combinations up to a certain iteration level. The figure shows the initial matrices and the new matrices of each iteration with a red border. The arrows between the boxes that contain the matrix combinations show how they are constructed. Arrows originating from matrix combinations of the same iteration step are colored similarly. The resulting structure is a hierarchical graph.

IFS the Inductive Rotation Method also inserts new composite transformation matrices into the graph at every new iteration level. Each of these composite transformation matrices consists of a rotation matrix multiplied by two translation matrices. In the first step this works analogous to IFSs. In contrast to IFSs a number of $p_r - 1$ composite matrices are inserted into the graphs in each new iteration level. The composition of these three matrices represents the rotation about each step's pivot. The amount of translation induced by the translation matrices depends on the current iteration step. The graph for a 2-way Inductive Rotation Pattern is depicted in Figure 3.6. In the first iteration the graph consists of the identity matrix only. In each successive iteration a number

of $(p_r - 1)$ new transformation matrices, representing the rotation around a new pivot, are added to the graph. Then the combination of all matrix nodes from the previous iterations with the newly inserted matrices is generated by matrix multiplication. To generate the whole pattern each node of the graph is multiplied with the initial tile (c_{tile}) to generate the complete Inductive Rotation Pattern. The definitions in 3.4 show the sets of rotation and translation matrices for the parallel algorithm.

$$\begin{aligned} R_k &\in \{R_1, R_2, ..., R_{p_r-1}\} \\ T_l &\in \{..., T_{p_i}, T_2, T_1\} \end{aligned} \tag{3.4}$$

Figure 3.6 depicts all possible combinations of initial matrices for the first steps of an Inductive Rotation Pattern with a set of two different rotation matrices: $\{R_1, R_2\}$, $p_r - 1 = 2$ (2-way rotation). Expression 3.5 shows how the matrices for an Inductive Rotation Pattern are generated by using different translation matrices in each iteration step. The index l represents the current iteration and addresses a specific translation matrix. Index k represents a matrix from the set of rotation matrices in 3.4.

$$B_{lk} = T_l^{-1} R_k T_l \tag{3.5}$$

The implementation of the algorithms in Chapter 4 uses the techniques presented in the next sections which are based on the sprite based approach [Par13] and properties of Inductive Rotation Graphs to compute the successive rotations around the pivots.

The Inductive Rotation Indexing Scheme

To create an Inductive Rotation Pattern an indexing scheme is established. This indexing scheme uses the idea of numeral system conversion and was inspired by the algorithm that converts decimal numbers to other numeral systems (binary, octal, hexadecimal) [Tut15]. We successively divide the index of each tile by the number p_r in each step to obtain the matrix indices k and i which are necessary to compute the center and rotation of the tile. The numbers resulting from the divisions and the remainder of the last division represent the indices k for the rotation matrices. The indices l for the translation matrices are found by numbering the calculations top down. The matrices are constructed from left to right by reading the results (k and l) in the table from bottom up. In the case that the remainder of a division step equals zero (i.e., $k = 0$), the matrix B_{lk} of this step is replaced by the Identity-Matrix. Let us assume we are using the 2-way Inductive Rotation Method and we want to compute the matrix indices for the tile with index 22. To find these indices we can use the successive divisions depicted in table 3.1. The calculations in this table yield the index triple (2 1 1) for k which, together with the triple of the order of the computations (3 2 1) for l, defines the matrices $B_{32}B_{21}B_{11}$. When constructing the matrices the table is read from bottom up and the matrix chain is constructed from left to right. By multiplying these concatenated matrices with the generator points, the tile is transformed to the correct position in the pattern. The scheme computes the translation matrices for any arbitrary tile and method-specific rotation number p_r. Table 3.2 shows the mapping of tile indices to matrix indices for the first two iterations of a

Number	Operation	k (Remainder)	l (Order)
22	$22/3 = 7$	1	1
7	$7/3 = 2$	1	2
2	$R : 2$	2	3

Table 3.1: An example on how to compute the indices k and l for the rotation and translation matrices. In this specific example the matrices for the tile with index 22 are sought. To find the indices k the starting number is successively divided by powers of p_r and the remainders of these divisions are kept. The indices l are found by simply numbering the calculations bottom up.

2-way Inductive Rotation Pattern. The total number of tiles in an Inductive Rotation Pattern is $p_r^{p_i}$. Table 3.3 and Table 3.4 show the matrices necessary for computing 2-, 3- and 5-way Inductive Rotation Patterns with the parallel algorithm. Figure 3.7 and 3.8 visualize 2- and 3-way Inductive Rotation Patterns and the respective matrices that generate the positions of the tiles contained in the patterns. Figure 3.9 shows the tiles of a 3-way IR-Pattern and their corresponding indices.

Index	0	1	2	3	4	5	6	7	8
Matrix	I	B_{11}	B_{12}	B_{21}	$B_{21}B_{11}$	$B_{21}B_{12}$	B_{22}	$B_{22}B_{11}$	$B_{22}B_{12}$
k1	0	1	2	0	1	2	0	1	2
k2	0			1			2		
k3	0								
Index	9	10	11	12	13	14	15	16	17
Matrix	B_{31}	$B_{31}B_{11}$	$B_{31}B_{12}$	$B_{31}B_{21}$	$B_{31}B_{21}B_{11}$	$B_{31}B_{21}B_{12}$	$B_{31}B_{22}$	$B_{31}B_{22}B_{11}$	$B_{31}B_{22}B_{12}$
k1	0	1	2	0	1	2	0	1	2
k2	0			1			2		
k3	1								
Index	18	19	20	21	22	23	24	25	26
Matrix	B_{32}	$B_{32}B_{11}$	$B_{32}B_{12}$	$B_{32}B_{21}$	$B_{32}B_{21}B_{11}$	$B_{32}B_{21}B_{12}$	$B_{32}B_{22}$	$B_{32}B_{22}B_{11}$	$B_{32}B_{22}B_{12}$
k1	0	1	2	0	1	2	0	1	2
k2	0			1			2		
k3	2								

Table 3.2: This table depicts how tile indices are mapped to matrices for the 2-way Inductive Rotation Method. The first row shows the matrix chains which correspond to each index of the second row. The numbers in the other rows represent the matrix indices which are computed according to the indexing scheme. To compute the numbers in rows k3 to k1 which represent the indices for the rotation matrices at the different levels, the index is first divided by 9 to get the value for k3, the remainder is then divided by 3 to compute k2's value. The remainder of the last division is the value for k1. For larger indices (>26) the table would be extended by another row labeled k4, k5, ..., to compute the position of the tile.

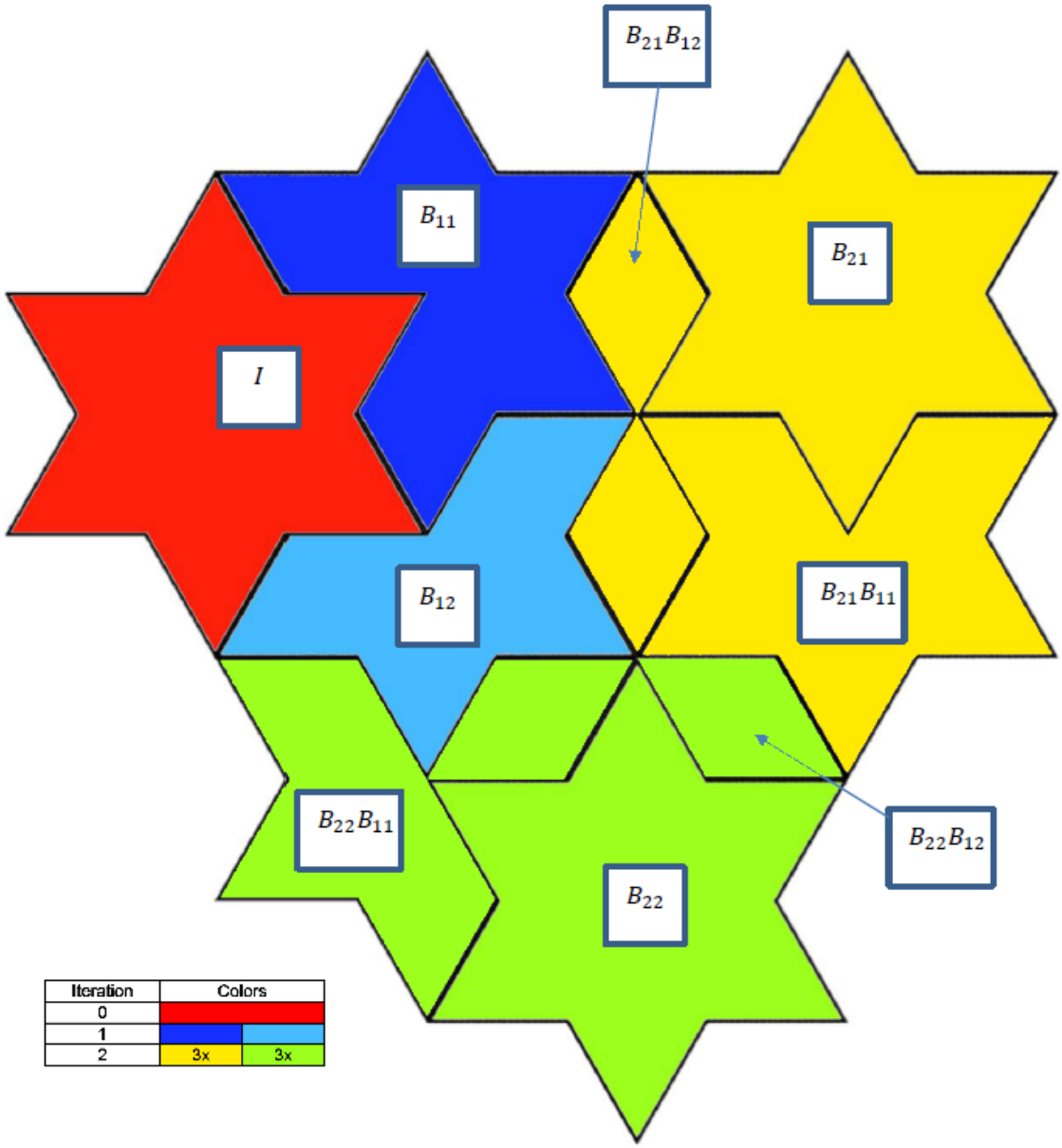

Figure 3.7: This figure shows a 2-way Inductive Rotation Pattern after two iteration steps. The algorithm starts with the red tile. The first iteration introduces the 2 blueish tiles. All other tiles are generated by the second iteration. Similarly colored tiles, that are created in the second iteration share the leftmost matrix in their matrix chain. This leftmost matrix is always responsible for the largest translation of the according tile. The figure also depicts the matrix chain which is responsible for each tile's position in the pattern.

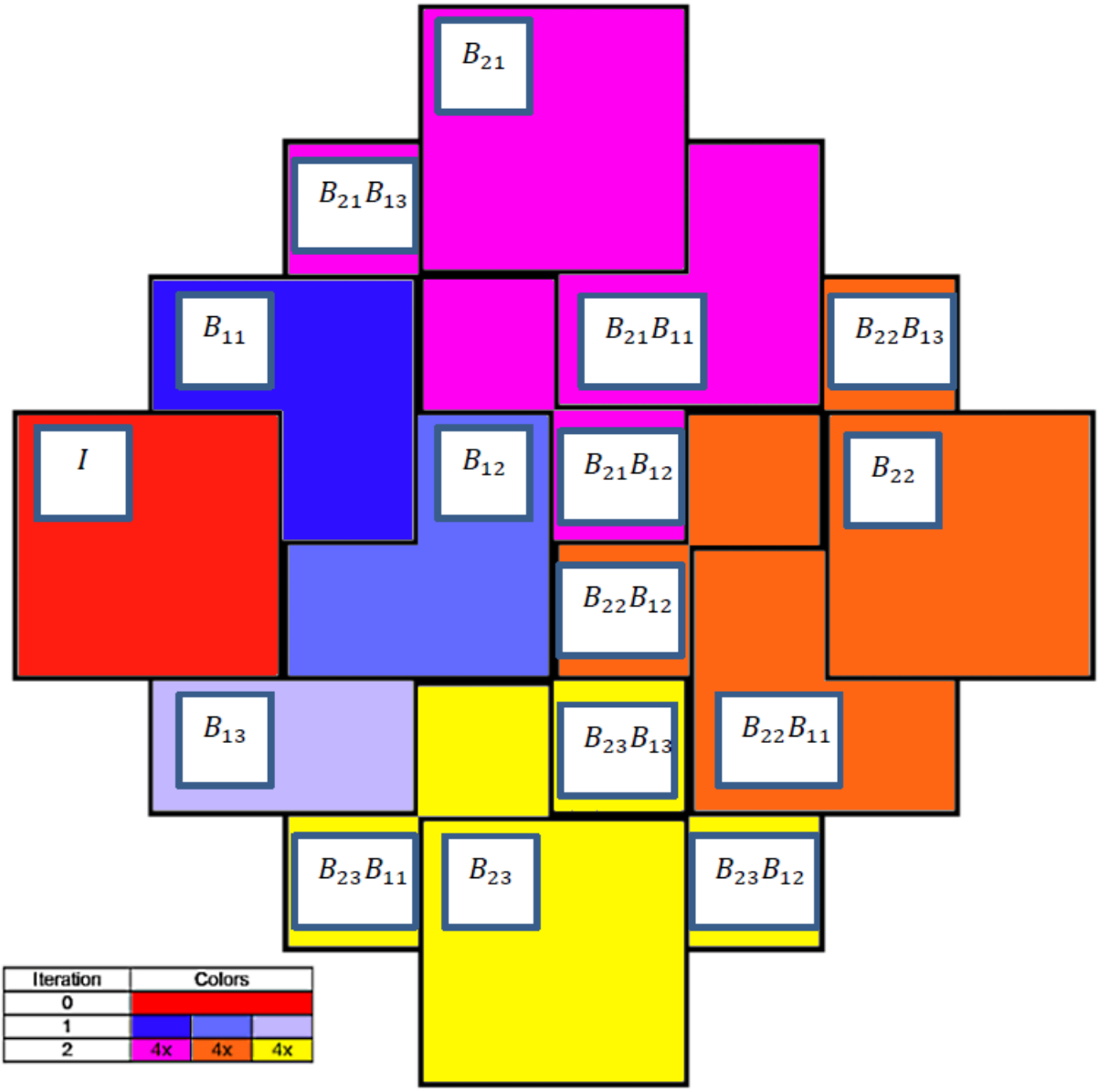

Figure 3.8: This figure shows a 3-way Inductive Rotation Pattern after two iteration steps. The algorithm starts with the red tile. The first iteration introduces the 3 blueish tiles. All other tiles are generated by the second iteration. Similarly colored tiles, that are created in the second iteration share the leftmost matrix in their matrix chain. The leftmost matrix is always responsible for the largest translation of the according tile. The figure also depicts the matrix chain which is responsible for each tile's position in the pattern.

Iteration	2-way	3-way
0	I	I
1	B_{11}, B_{12}	B_{11}, B_{12}, B_{13}
2	$B_{11}, B_{12}, B_{21}, B_{22}$	$B_{11}, B_{12}, B_{13}, B_{21}, B_{22}, B_{23}$
3	$B_{11}, B_{12}, B_{21}, B_{22}, B_{31}, B_{32}$	$B_{11}, B_{12}, B_{13}, B_{21}, B_{22}, B_{23}, B_{31}, B_{32}, B_{33}$

Table 3.3: Matrices which are needed to construct 2-way and 3-way IR-Patterns for different numbers of iterations.

Iteration	5-way
0	I
1	$B_{11}, B_{12}, B_{13}, B_{14}, B_{15}$
2	$B_{11}, B_{12}, B_{13}, B_{14}, B_{15}B_{21}, B_{22}, B_{23}, B_{24}, B_{25}$
3	$B_{11}, B_{12}, B_{13}, B_{14}, B_{15}B_{21}, B_{22}, B_{23}, B_{24}, B_{25}B_{31}, B_{32}, B_{33}, B_{34}, B_{35}$

Table 3.4: Matrices which are needed to construct 5-way IR-Patterns for different numbers of iterations.

Computing the Transformation Matrices

To construct the transformation matrices, the parallel algorithm computes each rotation matrix R_k by the angle determined with Expression 3.1 for each computed index k. To construct the matrix B_{lk} the transformation matrices T_l for the pivot of each step still needs to be found. For 3-way and 5-way Inductive Rotation Patterns the translation matrices T_l for the pivots are simply found by applying Expression 3.2.
An Expression to compute the pivots for 2-way Inductive Rotation Patterns was not given by Parzer who uses a search method to determine the tile containing the pivot.This is done by first determining the tiles which are centered at the largest x-coordinates and then selecting the tile centered at the minimum y-coordinate from this set. This approach, however, cannot be used with the parallel algorithm since the tile coordinates have not yet been found when the pivot's coordinates are computed.
Hence, the Inductive Rotation Framework uses a different method, which computes the pivot positions before creating the pattern by expressing x-and y-coordinates as summed terms.
To devise a scheme for computing the coordinates of 2-way pivots, the differences in the x- and y-coordinates of the successive pivots were analyzed by reverse engineering Parzer's algorithm. The center of the initial tile is assumed at point (1,1) for the definitions in this section.
The pivot's x-coordinates for the first ten iterations and their differences are depicted in Table 3.5 (The maximum number of iterations for the 2-way Inductive Rotation Method that Parzer's software can produce is 14). When taking a closer look at the sequence of numbers it can be observed that, in each iteration, the value increases by 3^i. Starting with $i = 0$ the variable i is incremented by 1 only in every second step. The sequence starts with the number $\frac{5}{3}$, which is the x-value of vector p, depicted in Figure 3.10, plus

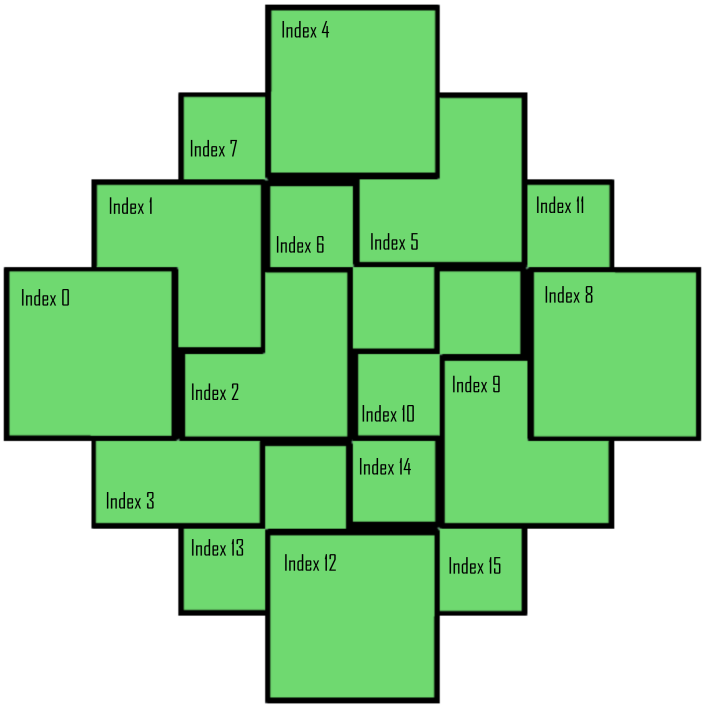

Figure 3.9: This figure depicts which index corresponds to which tile of a 3-way Inductive Rotation Pattern.

the vector pointing to the tile's center. Combining these observations leads to Expression 3.6.

Iteration	1	2	3	4	5	6	7	8	9	10
x	$1.\dot{6}$	$2.\dot{6}$	$4.\dot{6}$	$7.\dot{6}$	$13.\dot{6}$	$22.\dot{6}$	$40.\dot{6}$	$67.\dot{6}$	$121.\dot{6}$	$202.\dot{6}$
x-diff	0	1	2	3	6	9	18	27	54	81

Table 3.5: X-coordinates of 2-way rotation pivots in the first row and the differences between their values in the second row.

$$x(l) = \begin{cases} \frac{5}{3} & \text{if } l = 1 \\ \frac{5}{3} + \left[\sum_{i=0}^{l-2} 3^{\lfloor \frac{i}{2} \rfloor} * ((i \bmod 2) + 1)\right] & \text{if } l > 1 \end{cases} \tag{3.6}$$

The structure of the y-coordinates is different. Firstly, their values change only every second step. When it changes, the difference to the last value is always $m * p_{projy}$. The value of $p_{projy} = 0.58$ is obtained by rotating the vector p by 120 °about the center of the figure and projecting it onto the y-axis as depicted in Figure 3.10. The values of m, which are all multiples of 3, are depicted in Table 3.6. The combination of these facts

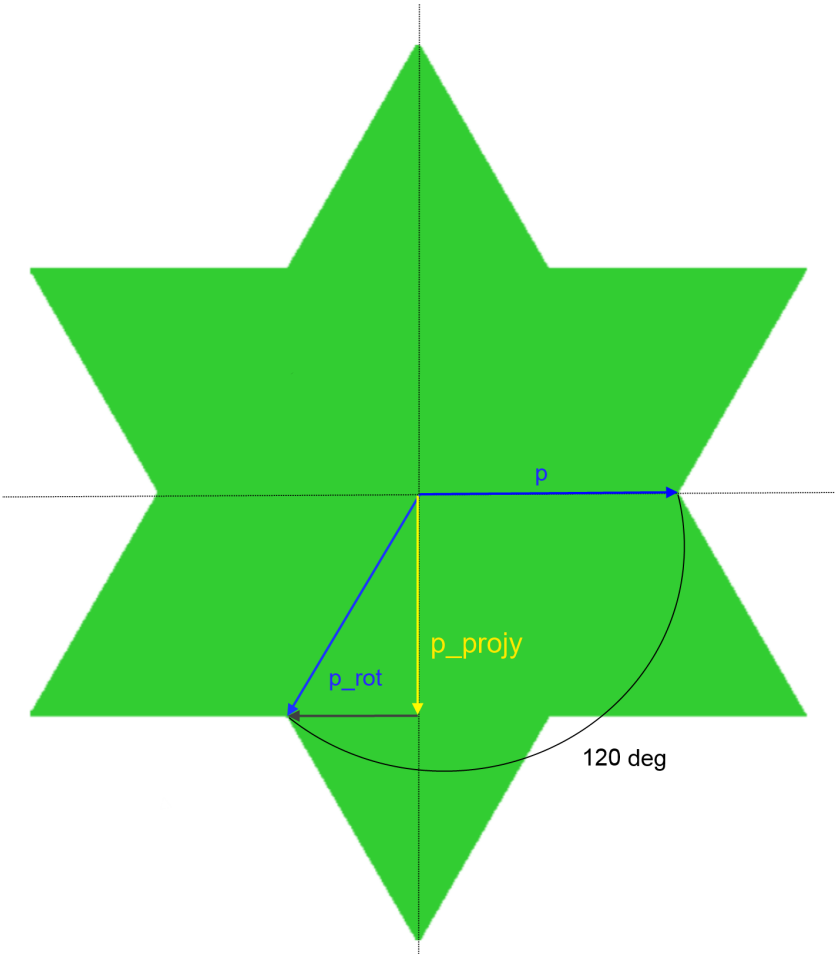

Figure 3.10: The component p_{projy} for Expression 3.7 is computed by first rotating the vector p by 120° and projecting it onto the y-axis.

leads to Expression 3.7.

$$y(l) = \begin{cases} \left(\sum_{i=1}^{l-1} 3^{\lfloor \frac{i-1}{2} \rfloor} * (i \bmod 2)\right) * p_{projy} & \text{if } l \geq 2 \\ 1 & \text{if } l < 2 \end{cases} \tag{3.7}$$

Each pivot's coordinates for the 2-way Inductive Rotation Method are then computed with Expression 3.8. Expression 3.8 provides the same results as the search method from Parzer's software for the iterations 11 to 14, which is the maximum number of iterations that Parzer's software can produce for 2-way Inductive Rotation Patterns. The visual results of the implementation in Chapter 4 provide reason to believe that the Expression is also correct for higher iteration numbers. A mathematical proof or a geometrical meaning for this Expression can, however, not be given at this time.

Iteration	1	2	3	4	5	6	7	8	9	10
y	1	0.42	0.42	-1.3	-1.3	-6.5	-6.5	-22.0	-22.0	-68.8
y-diff	0	-0.58	0	-1.74	0	-5.2	0	-15.5	0	-46.8
m	1	1	0	3	0	9	0	27	0	81

Table 3.6: Approximate y-coordinates of 2-way rotation pivots in the first row and the differences between their values in the second row. Numbers in the third row show how often the approximate factor $p_{projy} = -0.58$ is contained in the y-diff value above each column.

$$\vec{p}_{pivot}(l) = (x(l), y(l)) \tag{3.8}$$

3.4.2 Substitution Tiling Approach

Tile substitution cannot directly generate Inductive Rotation Patterns. The mathematical theory, developed by Freetloeh et al. [FH15], produces only 3-way Inductive Rotation Patterns that are contained in larger tilings. Therefore this method creates, in contrary to the previous methods, tilings of the plane in accordance with the mathematical definition (there are no tile overlaps). The tilings, which are created by this method do not represent Inductive Rotation Patterns themselves, but they contain Inductive Rotation Patterns. The idea of this approach is to apply a rule-based tile-substitution scheme, consisting of four different rules, to create Inductive Rotation Patterns. The substitution scheme is depicted in Figure 3.11. Additionally each of the four different tiles identified by T_1, T_2, T_3 and T_4 needs to be mapped to a sub square of the prototile of an Inductive Rotation Pattern for visualization. A pattern is represented by squares of 1/4 the size of the prototile with this approach. The theory works only with squares. To apply the substitution tiling approach, objects which store the properties of each tile are defined first. Each instance of these tile objects stores a tile's background-texture, its rotation, represented as an integer and its corner vertices.

To create a pattern by tile substitution the following simple steps are executed:

1. Create a set of four tiles, also called seed tiles. Create a linked list data structure and append the seed tiles. Each of the seed tiles has 1/4 of the prototile's size.

2. In an iteration step: Replace all tiles according to the given substitution rules (T_1 to T_4). The rotation of each replaced tile is kept.

3. Perform another iteration if the desired pattern size has not yet been reached.

To execute an iteration step, each element of the linked list data structure is visited and the following operations are performed:

1. Inspect the tile type (T_1, T_2, T_3 or T_4) and select a rule from the rule set.

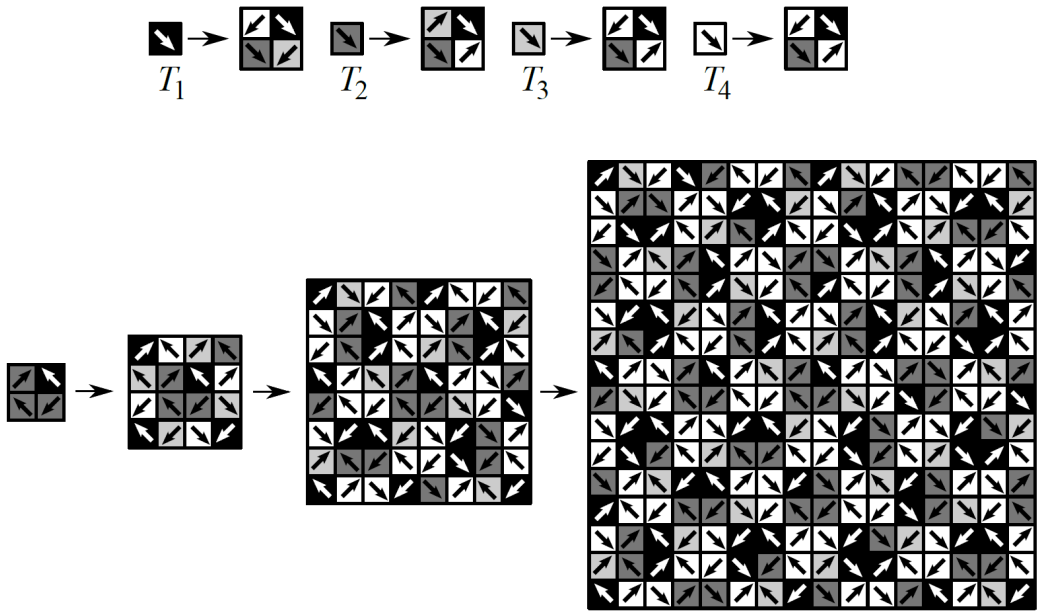

Figure 3.11: 3-way IR-Patterns can be created by applying substitution rules. Starting with four initial seed tiles, the rules T_1, T_2, T_3 and T_4 are applied to replace each tile in every step while keeping the rotation of the original tile, by rotating the tile given by the rule accordingly. After three steps this yields the tiling to the right. [FH15].

2. Create four new tiles, according to the selected rule.

3. Inspect the rotation of the current tile that is being replaced.

4. If the rotation of the tile in the rule does not match the rotation of the tile to be replaced but it has the same color it has to be aligned first. To align it, the left and right part of the specific rule are rotated until the left side of the rule matches the rotation of the tile to be replaced. This is done by adding the integer that represents the tile's rotation to each of the four sub squares rotations and then performing a modulo four operation .

5. Compute the corner and texture coordinates of the sub-tiles.

6. Replace the current tile in the linked list data structure with the sub-tiles.

3.5 Possible Applications for the Inductive Rotation Method

This section summarizes different experiments with the Inductive Rotation Method and provides ideas for future work. The first section shows how the Inductive Rotation Method can be used to generate fractals, by varying its generation parameters. The second section was inspired by Wang tilings and uses the Inductive Rotation Method for procedural texture generation. The third section provides ideas on how to implement animation features in the Inductive Rotation Framework.

3.5.1 Generation of Fractals

An interesting fact about the Inductive Rotation Method is that it can be used to generate fractals by variations of its generation parameters. For example the Sierpinski Gasket is generated by setting p_r to three and using $\vec{p}_{pivot}(l) = (2^l, 1)$ to determine the pivots for each iteration. Figure 3.12 shows the result of these settings. Other versions of the Sierpinski Gasket have also been generated in the experiments. Besides the Sierpinski Gasket several other structures with fractal properties emerge when the p_r parameter and the $\vec{p}_{pivot}(l)$ functions are varied. Results of experimenting with these parameters are shown in Figure 3.13.

3.5.2 Texture Rendering

Inspired by Wang Tilings, we conducted several experiments with the Inductive Rotation Method regarding procedural texture generation. Instead of simply stitching the textures together in x- and y-direction we used the Inductive Rotation Method to create procedural textures.
The best results were achieved with textures containing only few, small distinctive features and a non-periodic, chaotic structure. The water texture depicted in Figure 3.14 which was used to create Figure 3.15, with 5-way Inductive Rotation, provides an example for such a texture. In our experiments the 5-way Inductive Rotation Method provided the best results in all cases. Other experiments are depicted in Figure 3.16. The left side of the figure shows a texture that was created by simply stitching textures together, while the images to the right were created with the 5-way rotation method. The zoom level of both images in the figure is nearly the same.

While the images on the left side suffer from the repeating texture patterns and contain different visual artifacts, the images on the right side benefit from the asymmetrical distribution and do not exhibit these artifacts. The image still contains artifacts, which can be recognized at a closer zoom level, but these artifacts appear at less periodic intervals. Figure 3.17 shows the grass pattern from Figure 3.16 at a closer zoom level. The most noticeable artifacts were marked by red boxes in the image. The visual quality of the generated pattern also depends on the rotation of the prototile texture. To find the optimal rotation and position for the prototile's texture, the prototile editor which is included in the Inductive Rotation Framework proofed to be very useful. This tool,

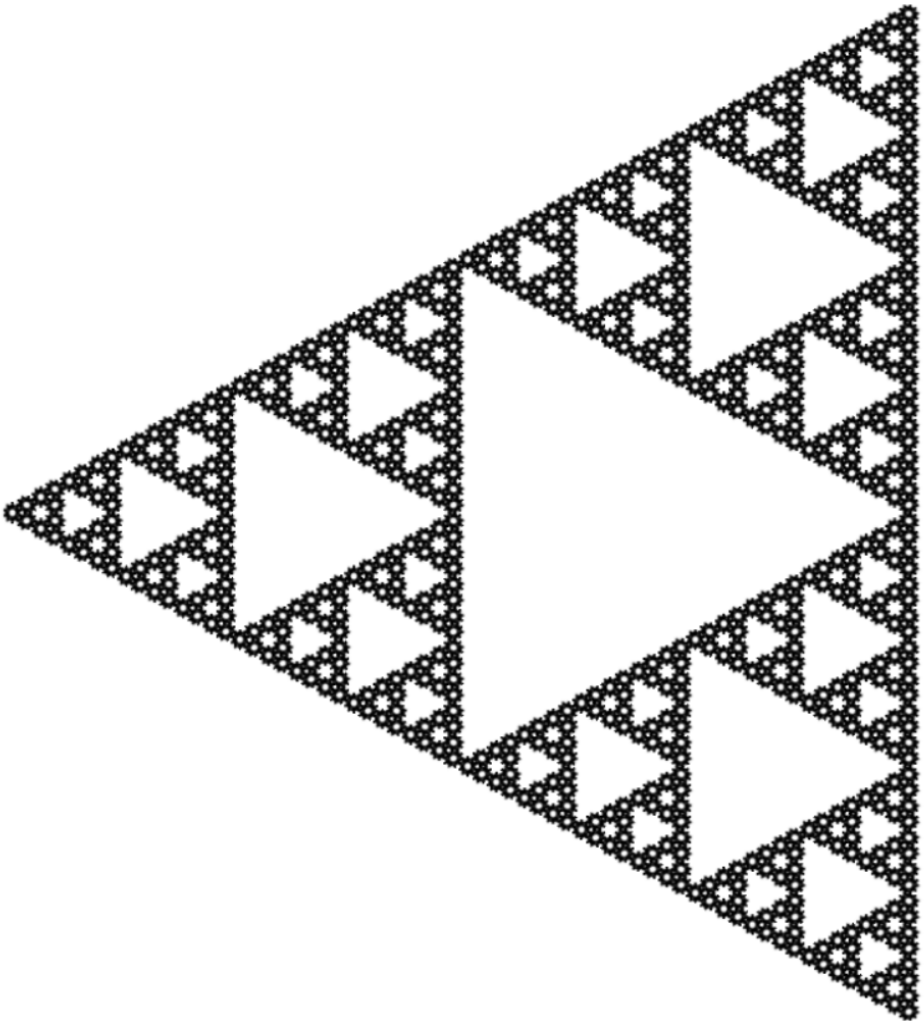

Figure 3.12: A Sierpinski Gasket generated by a variation of the parameters of the Inductive Rotation Method. This image was generated by setting $p_r = 3$ and using $\vec{p}_{pivot}(l) = (2^l, 1)$.

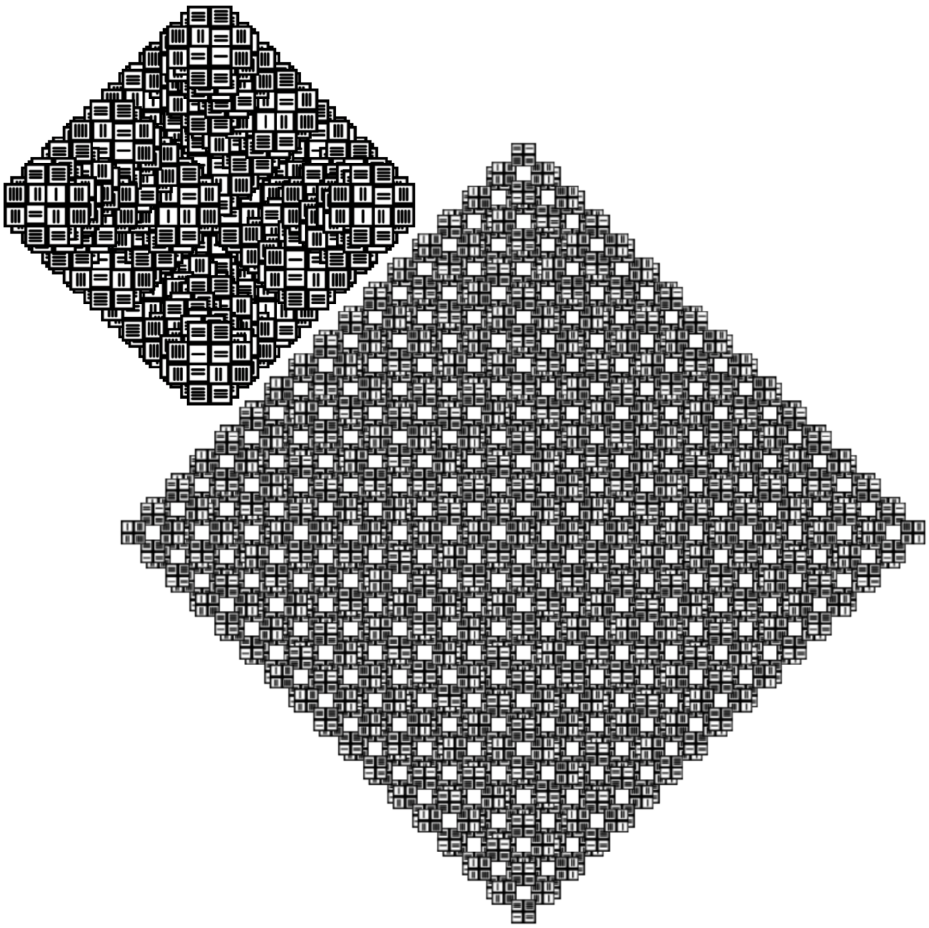

Figure 3.13: Different fractals that were produced by experimenting with the parameters of the Inductive Rotation Method. The top left shape was created by setting $\vec{p}_{pivot}(l) = (2^l - 1.9^l, 1)$ and $p_r = 5$ the other shape was created by setting $\vec{p}_{pivot}(l) = (2^l + 0.55, 1)$ and $p_r = 5$

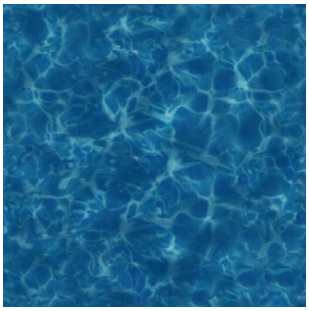

Figure 3.14: The water tile used to produce the Pattern in Figure 3.15

which is described in Chapter 4.6, permits the user to change the alignment and rotation of the initial texture.
To evaluate the performance in terms of quality a through comparison not only to simple stitching but to more advanced texture generation methods would be necessary. However, since this technique generally produces discontinuities at tile border it is doubtful that this method can compete with the state of the art methods.

3.5.3 Artistic Animation

The Inductive Rotation Framework also includes an experimental texture animation feature. To create animated Inductive Rotation Patterns the position of the image in the prototile editor is animated by using the WPF (Windows Presentation Foundation) rendering system. This generates fascinating pattern animations for performing arts. While the current version contains only an experimental feature that animates the texture by moving it up and down [Sip], a broader range of transformations could be supported by a future version of the Inductive Rotation Framework.

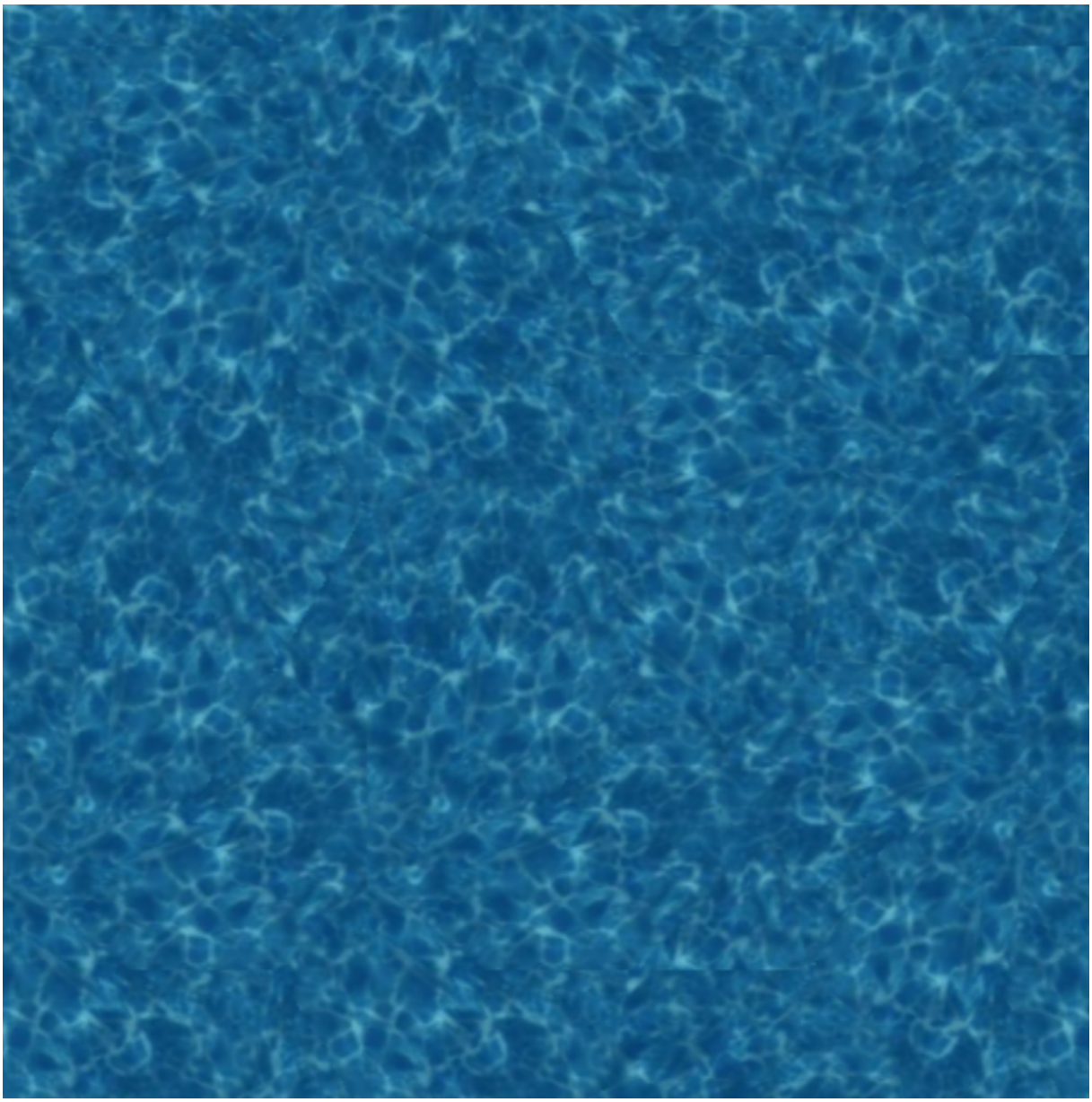

Figure 3.15: A water texture, created by applying the 5-way Inductive Rotation Method and the water tile depicted in Figure 3.14 for five iterations. The resulting pattern contains fewer artifacts than periodic tilings created with the same prototile.

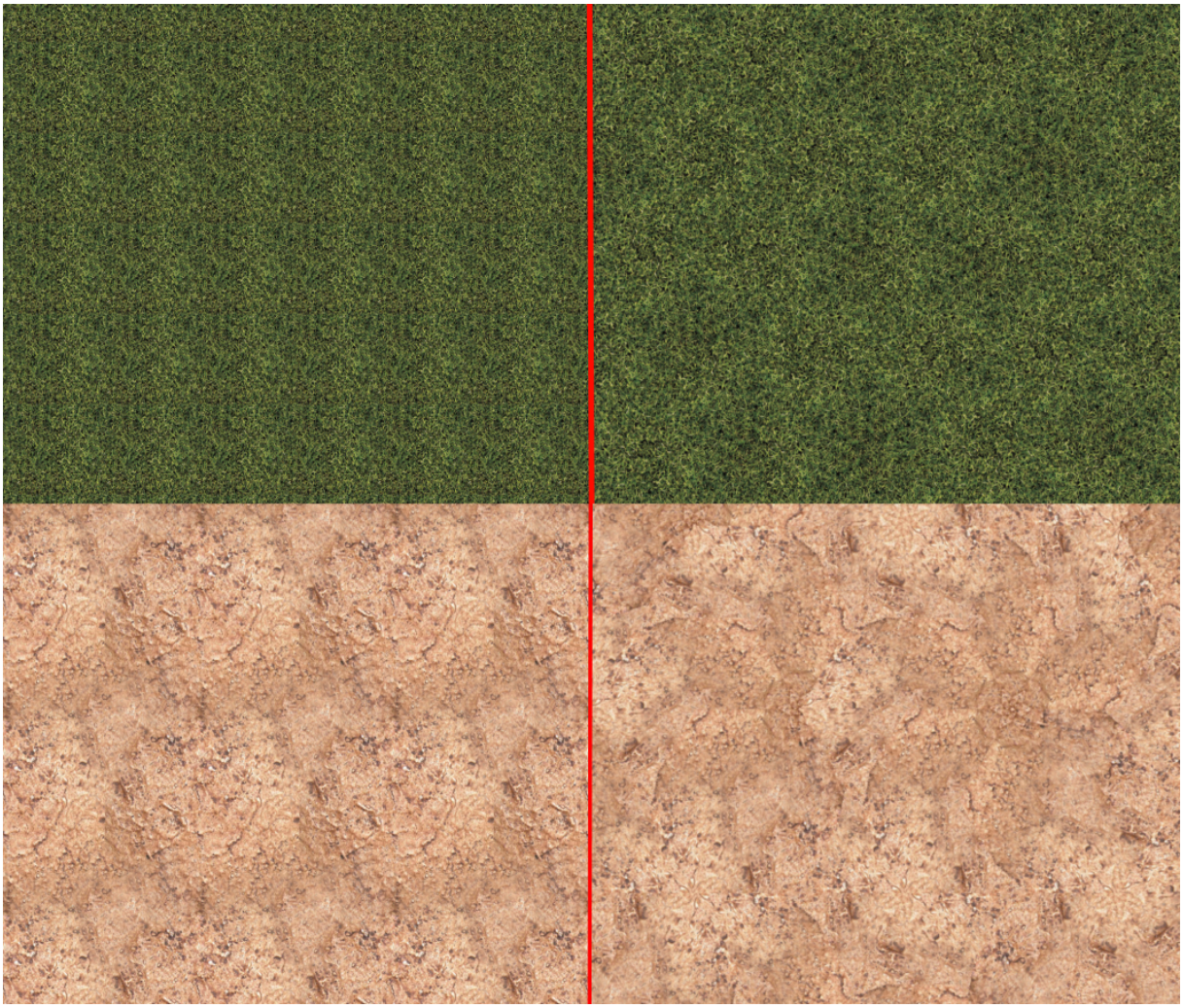

Figure 3.16: Comparison of different texture tilings. The images on the left side were created by stitching together the same texture in x- and y-direction while the images on the right side were created with the 5-way Inductive Rotation Method. Images on the left side contain artifacts resulting from the periodicity of the pattern. While the right side also contains artifacts, they are less apparent due to their aperiodic nature.

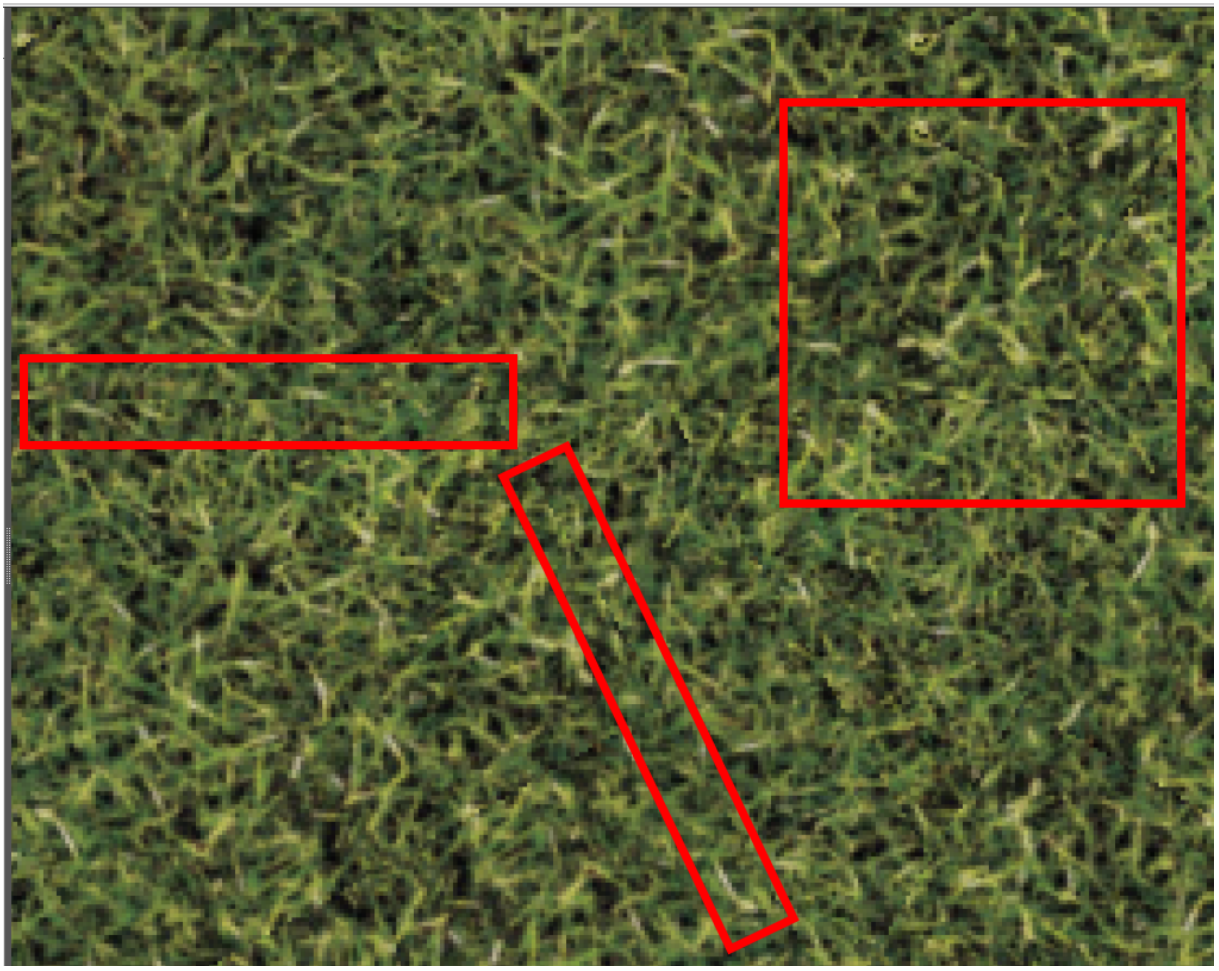

Figure 3.17: A grass-pattern created with the 5-way Inductive Rotation Method. At a closer zoom level texture artifacts from overlapping tiles, marked in red, become more apparent.

CHAPTER 4

Implementation

This chapter gives an overview of the development process and provides details on the implementation of the algorithms introduced in the previous chapter. The first section describes the technological requirements for the Inductive Rotation Framework which influenced the decisions concerning software technology. The second section gives an overview of Direct Compute 5.0, a technology that the algorithms use for parallel processing on the GPU. Section three describes the implementation details of the algorithms in depth. The third section is further structured into subsections containing all details about generation, rendering and editing of IR-Patterns. An overview of the iterative development process is provided in section 4. Section 5, which also serves as documentation, visualizes the Inductive Rotation Framework's software architecture and points out the functionalities of the important classes of the framework. The last section contains screen-shots and a description of the Inductive Rotation Framework's user interfaces.

4.1 Technological Requirements for the Framework

For the implementation of the Inductive Rotation Framework the following technological requirements had to be considered:

- Hofstetter uses a Windows operating system.
- A rapid implementation process has to be ensured.
- GPU debugging features are needed.
- GPGPU support is needed.
- A modern user interface library is preferred.
- An IDE with an integrated UI-designer is necessary.

Since Hofstetter Kurt uses Windows 7 and 8 as operating systems it was not necessary to develop the Inductive Rotation Framework as a multi-platform solution. For Windows the following reasons make the Microsoft .NET Framework an attractive development platform: It supports software developers with libraries for frequently performed tasks, it has a stable and relatively bug free code base, all of its standard libraries are very well documented and the .NET Framework is already installed on most systems running a Windows OS.
For developing software with the .NET Framework, Microsoft offers the Visual Studio (2013) IDE. Visual Studio includes many tools which simplify software development and contains advanced debugging features, including a Direct X Shader debugger.
The IDE supports two different project types for developing user interfaces with integrated WYSIWYG editors. Developers can choose to either work with Windows Forms projects or WPF (Windows Presentation Foundation) projects if a GUI is needed. WPF is based entirely on vector graphics and provides a clear separation of design and development [Neta]. Windows Forms, the predecessor of WPF, follows several obsolete paradigms and will not be enhanced by new features in the near future [All].
For software development with the .NET Framework, Microsoft offers different languages. According to the Tiobe Index [Sof15], the most popular languages supported by the .NET Framework are, in descending order, C++, C# and Visual Basic (.NET).
In contrast to Visual Basic (.NET), C# offers a subjectively more pleasing C-style syntax. To access Direct X functionality via the .NET Framework, either the library Slim Dx or the library Sharp Dx can be used. Slim Dx and Sharp Dx both wrap native Direct X calls. Differences exist only concerning documentation, where Slim DX's documentation is slightly more comprehensive.

Based on these considerations, the following technologies were chosen for the development of the Inductive Rotation Framework:

- Windows 7/8
- .NET Framework 4.5 /C#
- Visual Studio 2013
- Windows Presentation Foundation
- Direct X(Slim DX)

Architectural Considerations

Two different paradigms for the development of WPF applications exist. Developers can either use the classical, event based programming paradigm or apply the Model View ViewModel architectural pattern [Netb]. Model View ViewModel (MVVM) is a specialization of Model View Controller (MVC) architecture and directly supported by WPF. MVVM follows a separation of concerns approach and allows programmers to

develop tightly coupled, change resistant code by partitioning the software's architecture into View, Model and ViewModel classes. This architecture supports a work flow that needs less coordination of programmers and designers through the loose coupling of user interface code and the code for domain logic [Netd]. The MVVM pattern also increases application testability by allowing programmers to instance classes for UI logic independently of UI technology. However, the MVVM pattern also tends to introduce more complexity [Gos] which leads to longer development cycles. The higher complexity results from the architectural structure which demands more code for algorithmically trivial parts of the project.
To ensure fast development cycles an event based implementation was chosen for the Inductive Rotation Framework. However, the Inductive Rotation Framework applies a ViewModel approach which separates UI logic from domain logic by structuring the source code into different projects.

4.2 An Introduction to Direct Compute

In graphics applications the processing power of the GPU is normally dedicated to perform operations on vertices, fragments or geometry. Using the GPU for tasks which are typically executed on the CPU is referred to as General Purpose Computing on Graphics Processing Units (short GPGPU). Some of the advantages of GPGPU computing can be summarized as follows:

- GPUs are optimized for massive parallel computing, allowing the graphics programmer to utilize a vast number of threads for parallel computations.
- Results of computations performed on the GPU are stored directly in GPU memory. This eliminates the need to upload buffers from the CPU's RAM into the GPU memory which is a comparatively slow operation. Figure 4.1 depicts the differing access speeds.
- GPGPU algorithms can be implemented in a scalable fashion which makes them easy to adapt for future generations of graphics hardware.

Historically, the first successful GPGPU application, for the computation of matrix LU factorizations was developed in 2005 [DWL+12]. At that time using GPGPU was a cumbersome endeavor due to the fact that algorithmic problems had to be reformulated in terms of graphics primitives. The desire to avoid these transformations led to the creation of several API's, which were intended to simplify GPGPU programming. The first popular GPGPU framework was NVIDIA's CUDA. CUDA works only with vendor specific (NVIDIA) hardware. Nowadays the most popular GPGPU standard is OpenCL, which is maintained by the Khronos Group [Gro]. OpenCL is supported by all major platforms and graphics-card vendors. Direct Compute is a different API which is part of Direct X since version 10. However, in version 10 Direct Compute has limited features. The API supports GPGPU on all major vendor's graphics cards.

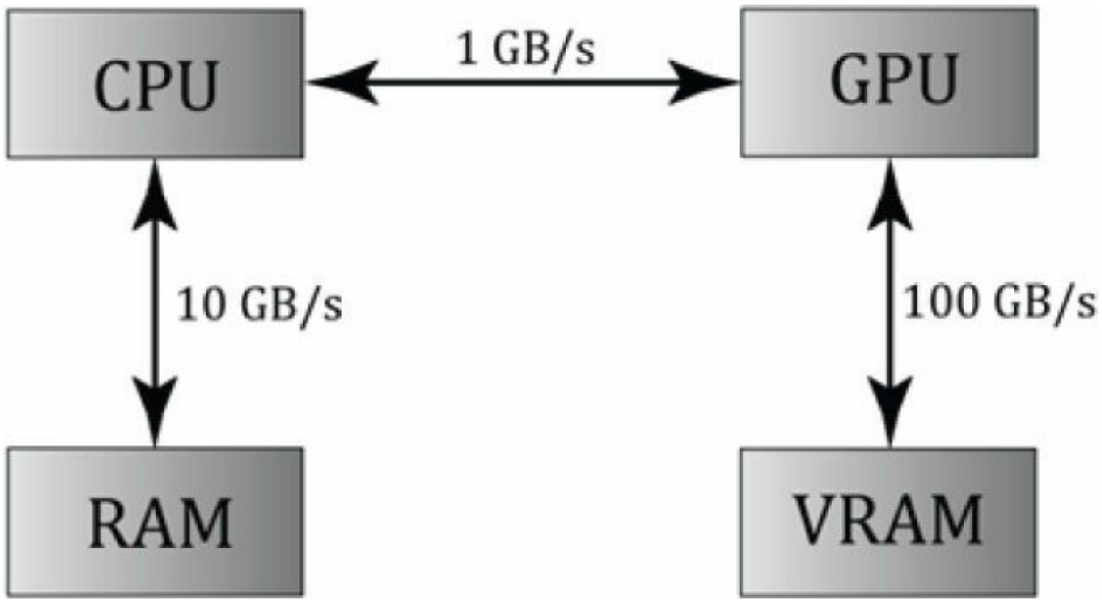

Figure 4.1: Different orders of magnitude for memory access. While the CPU can access its RAM at a relatively high rate, transferring data between the CPU and the GPU is slow. The GPU on the other hand can access its memory at a very rate. [Lun12]

Direct Compute uses so called compute shader functions which enable algorithm developers to access the GPU. compute shader functions are not directly part of the graphics pipeline [Lun12], which implies that they are not executed when a frame is rendered (like vertex, pixel or domain shaders). Instead, these programs are triggered separately by calling the appropriate Direct X API command.

Modern GPUs possess multiple processing units also called multiprocessors. Each of these multiprocessors contains several groups of streaming processors which are able to execute multiple threads at once. Direct Compute algorithms are executed in multiple thread groups. Each of these thread groups is dedicated to a single streaming processor. Multiple thread groups can be assigned to a single multiprocessor but it is not possible to split a thread group amongst different streaming processors. Threads belonging to the same group can be synchronized and have access to shared memory. According to Fung [Fun10] for an optimal distribution at least two thread groups should be assigned to a single streaming multiprocessor to avoid stalls.

Each thread group consists of n threads. The number of thread groups which are executed simultaneously is directly specified by the programmer. Direct Compute starts execution when the Dispatch command is called. The Dispatch command accepts three parameters, x, y and z which represent the dimensions of a three- dimensional grid of thread groups. Figure 4.2 depicts such a grid of thread groups. The number of threads in each thread group is also defined as a three-dimensional grid. The maximum number of threads in a Direct Compute 5.0 thread group is 1024. The number of thread groups

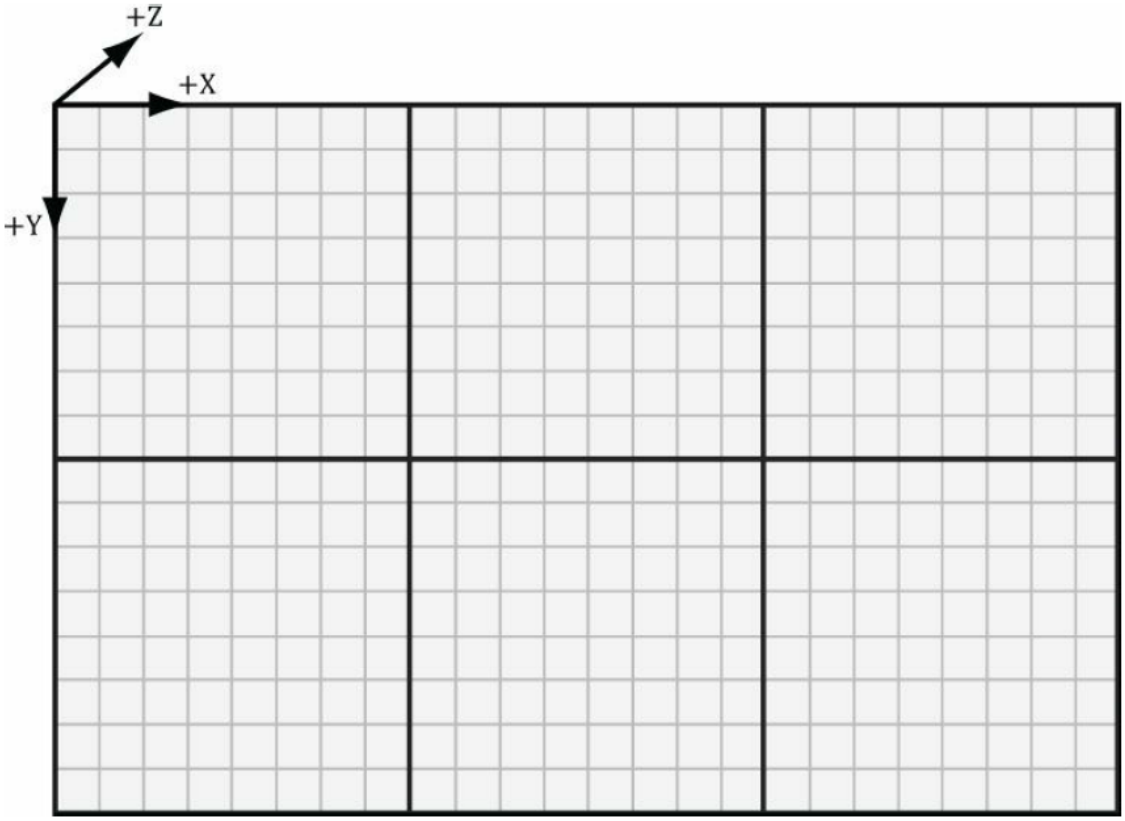

Figure 4.2: This figure shows the alignment of thread groups and their respective threads. Direct Compute dispatches thread groups in a three-dimensional grid with x, y and z dimensions. The threads of each thread-group are also aligned in a three-dimensional grid. The figure shows a grid of 6 thread-groups (2 in x and 3 in y direction) with 64 threads (8 in x and 8 in y direction) in each thread group. The number of threads in a group should be a multiple of the wavefront- or warp-size. [Lun12]

has to be chosen before compiling the compute shader function.
While NVIDIA graphics hardware combines threads into so called warps with 32 threads each, ATI combines threads into wavefronts with 64 threads each. All threads in a warp or wavefront are executed simultaneously on different streaming processors. While it is possible to define thread groups with thread counts that do not represent a multiples of the warp or wavefront sizes this is not advisable for performance reasons [Lun12].
Direct Compute 5.0 also offers new RWStructuredBuffer<T> objects, which permit compute shader functions write access to GPU buffers. However, compute shader functions can also use StructuredBuffer<T> objects or constant buffers if they only need read access. Additionally, Direct Compute 5.0 supports new input semantics such as SV_GroupID and SV_GroupIndex which provide information about a thread's group-id and its position in the thread group.

4.3 Implementation Details

This section explains all implementation details of the algorithms presented in 3.4 in depth. The Inductive Rotation Framework allows the user to select one of these algorithms for pattern generation. The first subsection contains implementation details for the sprite based algorithm from Parzer, which has been reimplemented in the Inductive Rotation Framework. The second subsection describes parallel algorithms which apply GPGPU methods for pattern generation. The third subsection provides details on the implementation of the tile substitution algorithm. The fourth subsection describes how the patterns are rendered and the last subsection outlines details of the pattern editor.

4.3.1 The Sprite Based Approach

The implementation of the sprite based approach follows the idea outlined in Section 3.2.2. To generate an Inductive Rotation Pattern this algorithm first creates two lists which contain the center coordinates and rotations of each tile. Tile rotations are stored as a list of integers according to Expression 3.1. To render the pattern using graphics hardware, this representation must be converted into textured triangles. To render the tiles in a back to front order the sequence of triangles also needs to be reversed before they are copied into a vertex buffer and uploaded to the graphics hardware.

Pattern Generation

When the algorithm starts, the two lists (a list of two-dimensional vectors and a list of integers) which represent the pattern's coordinates and rotations are initialized with the vector $(1, 1)$ and the integer 0 respectively. This implies that the first tile of the pattern will not be rotated and that it is centered at the coordinates $(1, 1)$. The algorithm performs the following steps iteratively to generate an Inductive Rotation Pattern, until the requested number of iterations has been reached:

To execute a single iteration step the pivot of the rotation is determined first. To this end Expression 3.2 is used to generate pivots for the 3- and 5-way IR-Method and Expression 3.8 is used to generate pivots for the 2-way rotation method. After determining the pivot, the algorithm continues by rotating each coordinate in the coordinate list around the pivot. All coordinates are first rotated by the smallest angle defined by the corresponding Inductive Rotation Method and stored in a temporary list. The procedure then rotates all coordinates of the coordinate list around the second smallest angle and appends the resulting coordinates to the temporary list. The algorithm continues with the next smallest angle until all coordinates contained in the coordinate list have been rotated by all angles defined by the corresponding rule.
The list of integers is updated in a similar fashion. Each number stored in the integer list is simply incremented by $m = 1$ and then a modulus division by p_r is performed with each result. The resulting numbers are then also appended to a temporary list. This procedure is repeated for $p_r - 2$ times and in each subsequent iteration the number m is

incremented by 1. After each of these steps the results are appended to the temporary list. Finally, the temporary list is appended to the list of integers.
This completes the necessary steps for a single iteration. The procedure is also depicted as pseudo code in Listing 4.1 for better understanding.

```
1  void CreateIteration()
2   {
3     Vector2 pivot = pivotFactory.FindPivot(iterations);
4
5     List<Vector2> sprites = new List<Vector2>();
6     List<int> rotations = new List<int>();
7
8     float increment = (float)(2 * Math.PI / pr);
9     float angle = increment;
10
11    for (int i = 1; i < pr; i++)
12    {
13      for (int j = 0; j < coords.Count; j++)
14      {
15        sprites.Add(rotateAround(coords[j], pivot, -angle));
16        rot.Add((rotations[j] + i) % pr);
17      }
18      rotationAngle += increment;
19    }
20
21    coords.AddRange(sprites);
22    rotations.AddRange(rot);
23
24    iterations++;
25
26  }
```

Listing 4.1: C# code that creates the new tiles for an iteration of the IR algorithm. The lists for the already existing tiles coords and rotations are extended when the function is called. The FindPivot function returns the pivot for the current iteration. The rotateAround function contained in the inner loop accepts a point to rotate, a point to rotate around and a rotation angle as parameters.

Upon completion the next iteration is executed if the requested amount of iterations has not yet been reached. To reduce computation times for subsequent executions of the

algorithm the lists are kept in memory. This is only possible if the user does not select a different Inductive Rotation Method between the calls (i.e. if p_r does not change).
Before the pattern can be rendered (as described in Section 4.3.2), the list of center coordinates and integers (rotations) must be converted into quads which are represented by two triangles. The sprite based as well as the parallel algorithm use a mask texture and texture transparency to create the shapes (star, hexagon) for 2-way and 5-way Inductive Rotation Methods. The position of each (quad shaped) tile's corners is found by adding different offset vectors, which depend on the tile's rotation, to the center coordinate of a tile. This concept is depicted by Figure 4.3.

These offset vectors are computed only once and in a way such that each tile has a width w and height h of two units. When the offset vectors are computed and stored in the multidimensional array, a second multidimensional array containing the texture coordinates of the corners that correspond to offset vectors is also created. the Hofstetter defines special proportions for 5-way and 2-way Inductive Rotation Methods, such that the height of each tile corresponds to $h = \sqrt{3}$ for the 5-way Inductive Rotation Method and $h = \frac{4}{\sqrt{3}}$ for the 2-way Inductive Rotation Method. To take this into account the offset vectors are modified such that they match these definitions. After computing the offset vectors they are stored in an array. To get the correct offset for a tile, the array is accessed at the index that represents the tile's rotation. After computing the corner and texture coordinates of each tile and uploading them to the vertex and texture coordinate buffers the pattern is ready to be rendered.

4.3.2 Pattern Rendering

After generating an Inductive Rotation Pattern with the above algorithm, or as described later with the algorithms below, the pattern is rendered. A vertex shader and a pixel shader are used to perform alpha blending, camera movement, stenciling operations and aspect ratio transformations. The pixel shader optionally also combines the tile textures with a stencil texture [Netc] that hides a specific region of the tile's texture or sets it to a semi transparent level. This method is necessary to achieve the different tile shapes (hexagon, star) for the 2-way and 5-way Inductive Rotation Methods. If desired, the pixel shader can also replace the hidden texture part by white color. This feature is necessary to achieve the same results as the "Irrational Image Generator" from Parzer [Par13].

4.3.3 Parallel Approach

The implementation of this algorithm is based on the definitions presented in Section 3.4.1. It uses the same data representation as the sprite based approach from the previous section: Tile rotations are represented by a list of integers and only the center coordinates of the tiles belonging to a pattern are stored. However, the expressions introduced in Section 3.4.1 create the basis to parallelize the computations needed to obtain these data. All necessary calculations are performed by Direct Compute 5.0 directly on graphics hardware. Two variations of this algorithm are presented in the following section: A parallel algorithm which computes the pattern representation once and then stores it

Figure 4.3: The sprite based algorithm as well as the parallel algorithm compute the corner coordinates of each tile by adding offset vectors to the previously computed center coordinate. The offset vectors are computed in advance and stored in a multidimensional array V. To create a tile, the algorithm accesses the array at the correct indices and adds the offset vector to the tile center to get to a corner coordinates of the tile. Texture coordinates of each corner are stored in a multidimensional array similarly such that the index of each offset vector a corresponds to the index of the texture coordinate located at the same corner the offset vector points to. The number of offset vectors and texture coordinates which are stored depends on the Inductive Rotation Method. The figure shows the offset vectors which are necessary to compute the corner coordinates for a tile when the 3-way Inductive Rotation Method is used. For each of the four possible rotations of the textured tile, four different offset vectors (drawn in green, black, blue and orange) need to be stored. However, since two triangles are used to construct a tile some of the vectors and texture coordinates are stored redundantly. In the image these are always the top left and bottom right vectors shown in the tiles.

in GPU memory and another version of this algorithm which splits pattern generation into multiple passes. While the first version increases pattern generation speed compared to the sprite based approach, the latter version permits rendering larger patterns while maintaining a constant memory footprint.

Mapping Rotations to Integers

The implementation of the parallel algorithm follows the implementation of Parzer's [Par13] sprite based algorithm and uses the same representation for tile-coordinates and rotations. Similarly to Parzer's algorithm it represents each tile only by its rotation and center and applies Expression 3.1 to compute the rotation matrices/angles in each step. To work, the parallel algorithm, however, needs a scheme to directly map the index of a tile to the integer that represents a tile's rotation.
The indexing scheme from Table 3.1 can be used in the following way to directly map a tile index to the integer that represents its rotation:
The integer representing the rotation of tile is obtained by summing the indices of the rotation matrices used to compute the center coordinate of the tile and performing a modulo division by p_r.
For the tile with index 22 in the example above this results in $2 + 1 + 1 \mod 3 = 1$ (see Table 3.1), meaning that the tile's rotation is represented by the integer 1. The corresponding angle is computed by applying Expression 3.1.

Pattern Generation

To generate Inductive Rotation Patterns with the GPGPU algorithm, several input and output buffers are set up in advance. First of all, an input buffer holding the pivots for the algorithm is generated. The content of this buffer needs to be regenerated if p_r changes. Next, an input buffer containing powers of p_r is created. This buffer is necessary since the internal pow HLSL function yields incorrect results for large numbers for unknown reasons. For example, calculating 6^8 results in the number 1679617 on the GPU, while performing the same calculation with the 32-bit float data type provided by the .NET Framework yields 1679616. Despite the inaccuracies being very small, they lead to significant computational errors which produce incorrect visual results.
Another buffer, containing the precomputed offset vectors for the corners of each tile, is created next. Computing these vectors on the GPU is possible but the Inductive Rotation Framework re-uses the already existing code of the sprite based approach to this end. Finally, the last input buffer which contains four integer variables representing the current run (described later), p_r and the number of thread-groups in x- and y-direction (the algorithm uses a 2-D grid for thread group layout) is generated.
Besides the input buffers, the algorithm also needs two output buffers for vertices and texture coordinates. These buffers, however, do not need CPU access since they are only used for passing data to the vertex and pixel shaders for rendering. Figure 4.4 provides a detailed overview of all necessary buffers for the parallel algorithm. To compute an

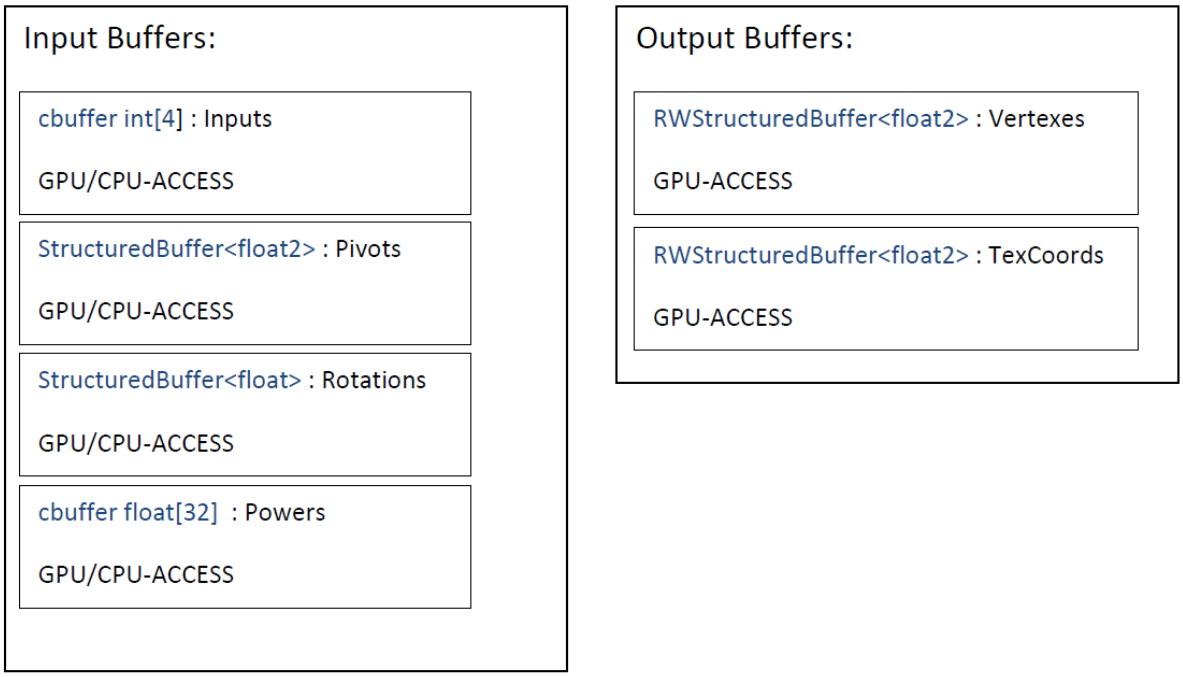

Figure 4.4: A diagram depicting the input and output buffers for the parallel algorithm which are used by the compute shader function. The input buffers, which also need CPU access, contain only a very small amount of data. The output buffers contain the vertices of the tiles. These buffers, in contrast to all previous methods, need only GPU access.

Inductive Rotation Pattern with the parallel algorithm some other definitions need to be established:

$$i_{max} = t_{max} * x_{max} * y_{max} \tag{4.1}$$

$$i_{tile} = t_{index} + tg_x * t_{max} + x_{max} * tg_y * t_{max} + r_{spilt} * i_{max} \tag{4.2}$$

$$i_{vbuf} = (i_{max} - 1 - t_{index} - tg_x * t_{max} - x_{max} * tg_y * t_{max}) * 6 \tag{4.3}$$

Expression 4.1 evaluates the number of threads on the GPU. Expression 4.2 maps thread-ids to tile indices and Expression 4.3 is used to compute the starting index in vertex and texture coordinate buffers. The variables and constants in the expressions above have the following meaning:

- t_{max} is the number of threads running in a thread group. The current implementation uses a constant value 1024 for t_{max} (which is the maximum of possible threads).

- x_{max} and y_{max} represent the maximal number of thread groups in x and y dimensions.
- i_{max} is the number of threads in all groups.
- i_{tile} is the index of the tile to which the thread id gets mapped to.
- i_{vbuf} is the start index in the vertex and texture buffer where the tile corner coordinates and texture coordinates will be stored.
- t_{index} is the integer index of the thread in a thread group. This value is obtained by the shader semantic SV_GroupIndex. In the current implementation the variable has a value ranging from 0 to 1023.
- tg_x and tg_y are the coordinates of the threads group in the thread-group grid. These values are obtained by the shader semantic SV_GroupThreadID, an integer vector with three dimensions. In the current implementation however, the z coordinate is not used (and therefore always set to one).
- r_{spilt} is an integer value which is read from the constant buffer. This value can be used to split pattern generation into multiple passes. Only the parallel LOD algorithm described later makes use of this variable.

To generate an Inductive Rotation Pattern each thread performs the following tasks to generate the corner coordinates of a tile:

1. Map the id of the current thread to a tile index by Expression 4.2. Figure 4.5 visualizes this process.
2. Compute the tile's center coordinate by a chain of transformations as described in Section 3.4.1 by using the tile index.
3. Map the id of the current thread to a starting index in the (output) vertex buffer by applying Expression 4.3. This expression maps tiles to the vertex and texture buffer in a back to front order, which is necessary for rendering the pattern later.
4. Compute the integer that represents the rotation of the current tile as described in Section 4.3.3 by using the tile index. Use this index to access an the array of offset vectors and texture coordinates similar to the sprite based algorithm (see Figure 4.3).
5. Use the center coordinate and the offset vectors to compute the tile's corner positions and store them them as six consecutive entries which represent two triangles in the output vertex buffer, beginning at the index computed in the previous step. This is explained in more detail by the example by Figure 4.6.
6. Generate the texture coordinates for the vertices and store them in the texture coordinate buffer. This works similar to filling the vertex buffer in the previous step.

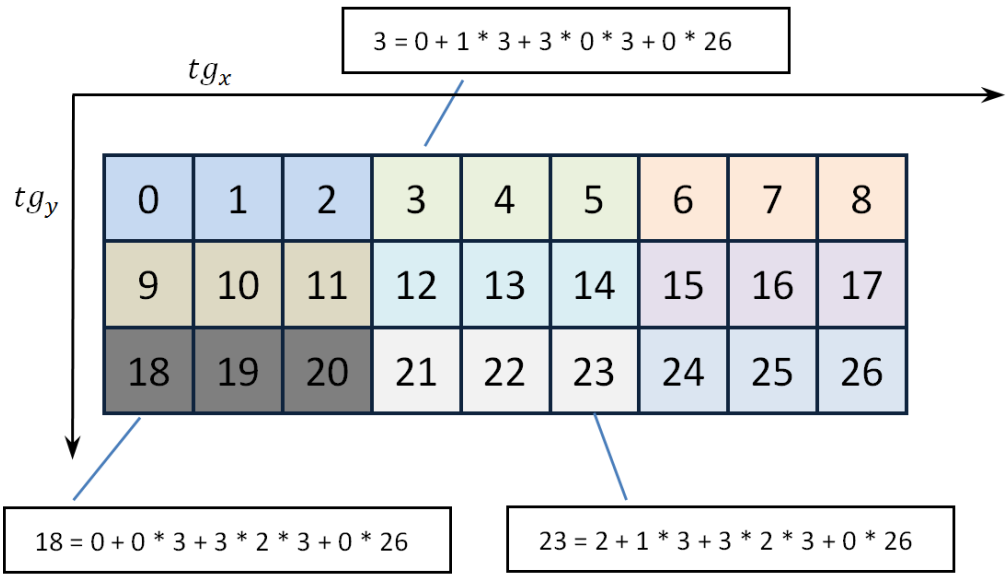

Figure 4.5: To create a mapping between the current thread and a tile index, Expression 4.2 is applied. Thread groups are depicted as boxes in different colors. There are $x_{max} = 3$ thread groups in the x-dimension and $y_{max} = 3$ thread groups in the y-dimension in this example. Each of the thread groups contains three threads in the example ($t_{max} = 3$). The parameter r_{spilt} is set to zero.

After the execution of all threads the vertex and texture buffers are merged and copied to the vertex shader's input buffer for rendering. Listing 4.2 and 4.3 show the compute shader functions for computing a tile's rotation and position by its index. These functions are executed by each thread to create the corner coordinates of each tile and store them. The steps for computing the tile coordinates for a pattern are the same for both versions of the compute shader algorithms that are presented in the next section. However, the way in which the pattern is stored and rendered differs for both algorithms.

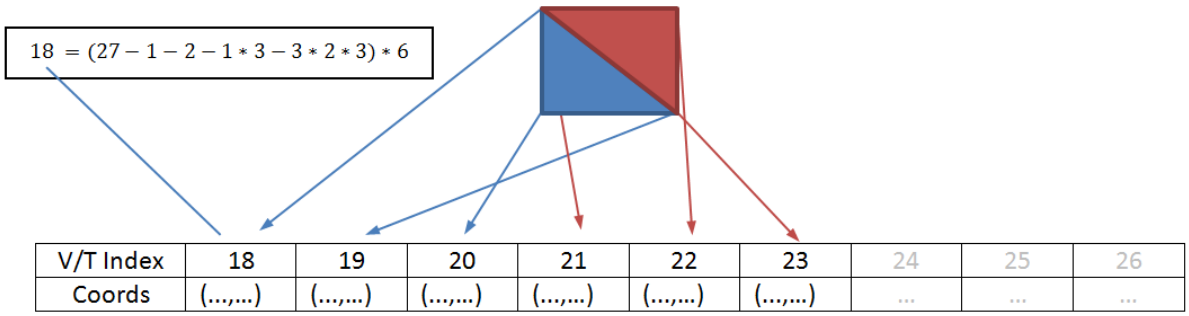

Figure 4.6: This figure explains how a specific thread is mapped to vertex and/or texture buffers by Expression 4.3. The example uses the same parameters as in Figure 4.5: $x_{max} = 3$, $y_{max} = 3$ and $t_{max} = 3$. This yields the index 18 in the vertex or texture buffer for a thread with the parameters $t_{index} = 2$, $tg_x = 1$, $tg_y = 2$ and $r_{spilt} = 0$. The Inductive Rotation Pattern is stored in a back to front order this way, which is necessary for rendering.

```
1  int ComputeRotationDesc(int index, int pr)
2  {
3    int finalRotation = 0;
4
5    int divisor = pr;
6
7    while (index > 0)
8    {
9      finalRotation += index % divisor;
10     iteration /= pr;
11   }
12
13   finalRotation %= pr;
14
15   return finalRotation;
16
17 }
```

Listing 4.2: The compute shader Function for finding the rotation of a tile with a given index. The first parameter is the index of the tile and the second parameter is p_r. The function returns an integer representing the rotation of the tile.

```
1
2  float2 ComputeTileCoord(int index, int pr)
3  {
4    int pivotOffset = 0;
5
6    float2 tileCoord = float2(1, 1);
7
8    float increment = 2 * PI / pr;
9
10
11   while (index > 0)
12   {
13     pivotOffset++;
14     float rotationAngle = ComputeRotationDesc(index, pr);
15     rotationAngle *= increment;
16     tileCoord = RotateAroundPoint(tileCoord,
17                                   pivotBuffer[pivotOffset],
18                                   -rotationAngle);
19     index /= pr;
20   }
21
22
23   return tileCoord;
24
25 }
```

Listing 4.3: The compute shader function for finding the center coordinate of a tile. The function accepts the index of a tile and p_r as parameters and returns a 2-D vector representing the center of the tile. The function successively rotates a tile until it has been placed at the desired position. The pivotBuffer array used in the code contains the pre-computed pivots of the pattern.

Versions of the Parallel Algorithm

There exist two variations of the parallel algorithm that use the procedure described above to generate Inductive Rotation Patterns:

The first implementation executes the algorithm with a grid of thread groups containing 43 thread groups in x-dimension and 43 thread groups in y-dimension which facilitates the creation of a total of $43 * 43 * 1024 = 1983376$ tiles. This number is sufficient for 10 iterations with the 3-way Method, 8 iterations for the 5-way Method and 13 iterations for the 2-way Method. The complete pattern is created immediately when

p_r is changed and stored directly in a vertex buffer in the GPU's memory where it is kept for rendering until a different IR-Method is selected.

The second implementation is a more complex version of the parallel algorithm. To enable the visualization of larger patterns than with the previous methods a different way to view patterns was implemented. The pattern is first rendered at a high resolution into a $8192 * 8192$ texels wide texture from a specific point of view. The position, size and zoom level for this texture are stored in an object that is further called "texture camera". The user sees only a part of this texture but views it at a very high resolution (how high depends on the zoom level). The zoom level, position and dimensions of the user's view port are called the "render camera" furthermore. The user can navigate freely in the large texture until the "render camera" hits a boundary of the "texture camera". If this happens the pattern is rendered with updated locations of the "texture" and "render cameras". To enable zooming, the Inductive Rotation Pattern is also rendered with updated camera settings after each fifth zoom steps of the "render camera" (This is the default value. The value can be changed through the options dialog of the application.). The overall concept of the algorithm is also depicted in Figure 4.8.
By saving intermediate results to a texture the total amount of GPU memory for the algorithm is bounded, a difference to all non-parallel methods which run out of memory after a certain amount of iterations. This new approach allows the artist to generate about 12 times larger patterns than with the previous methods. However, the total size of patterns is still limited by the amount of time it takes to render them to the texture.

The steps to generate the pattern when the pattern needs to be recreated can be summarized as follows:

1. Render the pattern into the large texture that is about four times larger than the view port.

2. Execute a compute shader pass to compute the first part of the pattern. This is done by setting the variable $r_{split} = 0$ in Expression 4.2

3. Render the pattern to the initially created texture as described in Section 4.3.2.

4. Increment r_{split} by one and execute step 2 and 3. Do this until the complete pattern has been rendered into the texture.

5. Finally, display the pattern to the user by rendering the texture onto a quad.

To split the computation of the pattern into multiple passes, the value of the variable r_{split} in Expression 4.2 is varied from index 0 to an index which depends on the number of iterations and p_r. Figure 4.7 shows the patterns that result from varying variable r_{split} from 0 to an end index 3 for a 3-way Inductive Rotation Pattern. To find the number of passes r_{max} (the maximum for r_{split}) to generate an Inductive Rotation Pattern,

Expression 4.4 is applied. In Expression 4.4 l repreesnts the currently selected iteration level and i_{max} is the maximal number of threads in all groups.

$$r_{max} = \frac{p_r^l}{i_{max} + 1} \tag{4.4}$$

The current implementation uses $16 * 16$ thread groups with 1024 threads each to compute a total of 262144 tiles in an iteration of the algorithm. The "texture camera" uses a texture with a size of $8192 * 8192$ texels. This is about four times larger than the "render camera's" view port. Larger textures and thread numbers decreased the performance on the tested hardware.

4.3.4 The Substitution Tiling Approach

When the algorithm is initialized, mappings between the sub-tiles $T_1 ... T_4$ described in Section 3.4.2 and the prototile texture are created. This is performed by splitting the texture of the prototile into four equally sized sub squares and assigning each sub square to a sub-tile T_i. The algorithm described in Section 3.4.2 is then applied to create the pattern.
To render the tiles after the pattern has been created, the rotations which are stored as integers are converted to angles by Expression 3.1. Figure 4.9 compares a pattern created with the substitution tiling approach to a pattern which was created with the parallel algorithm. The Inductive Rotation Pattern is contained in the tiling and marked with red boxes for better visibility. There is no simple method to remove the tiles which do not belong to the Inductive Rotation Pattern. Additionally, this method can only be applied to square shaped tiles (and textures).

4.3.5 Pattern Editing

To edit prototile textures, a prototile editor GUI was created. After the user completes an operation in the prototile editor window, the manipulated texture is uploaded to the GPU's memory and the pattern is updated. This is implemented by an event which is triggered after finishing an image manipulation command (rotate, translate, draw,...). The prototile editor window uses the WriteableBitmapEx library to enable fast bitmap manipulation [Wri]. All paint commands which use the mouse are encapsulated in State Objects [GHJV95]. This approach simplifies handling of state specific information and collocates code belonging to certain states to the same classes. Commands which are executed by menus and/or shortcuts implement the IPaintCommand interface. This interface is an implementation of the Command Pattern [GHJV95] which simplifies the source code for undo and redo operations. The application maintains two command stacks (done/undone) for this purpose. To conserve memory, the paint commands store only pixel differences when applying any of the operations. Rotation and translation of the image in the canvas is performed by manipulating the transformation matrices of WPF controls. The prototile editor also contains an experimental animation feature which is based on WPF's (storyboard) animation system.

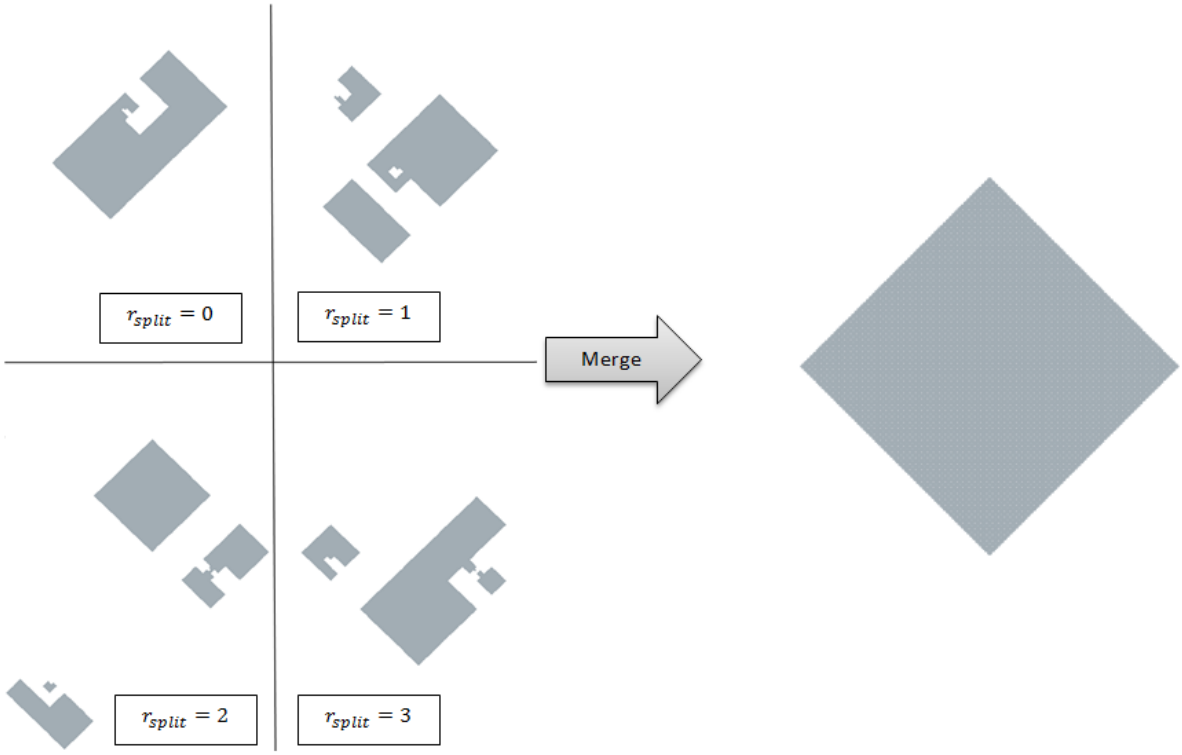

Figure 4.7: Pattern rendering is split into multiple passes with the parallel LOD approach. This figure shows the first four passes for a 3-way Inductive Rotation Pattern. The number r_{spilt} determines which tiles are rendered by applying Expression 4.2. These partial patterns were rendered with 1893376 tiles per pass. To render a composite pattern, the algorithm first computes the necessary number of tiles for the pattern and renders each partial pattern to the same texture in a back to front order. This means the pattern with the highest number r_{split} is rendered first.

Figure 4.8: The figure shows the concept of the parallel LOD algorithm. The “texture camera”, marked with a red boundary, views a part of the pattern which was rendered to a large (e.g., 8192 * 8192 texels) texture in advance. If the user moves the “render camera” out of the boundaries (into the orange area) or the maximum zoom level is exceeded, the texture is recreated with updated “texture camera” and “render camera” locations. The view port’s size is about 1/4 of the size large texture without zooming.

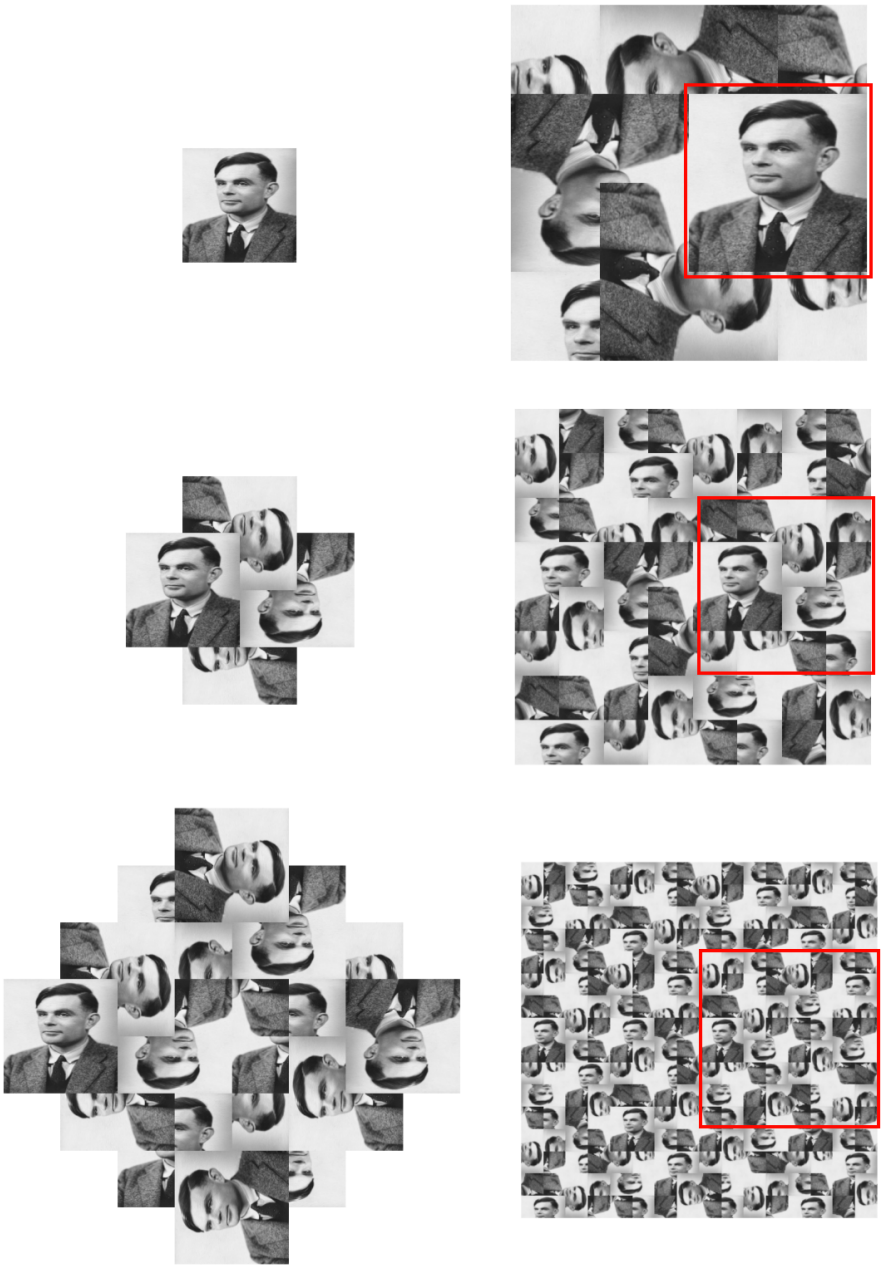

Figure 4.9: Comparison of the resulting pattern of the substitution tiling approach shown in the right column to a pattern created with the parallel method in the left column. The substitution method creates a tiling of the plane, containing the Inductive Rotation Pattern. [Bri]

4.4 Development Process

To ensure an efficient development phase, development was performed iteratively and with a user centered approach. This means that after a first analysis and implementation phase, test and development cycles were done in parallel. After each cycle, the desired features from Hofstetter's tests were implemented into the next version of the Inductive Rotation Framework.

As a first step a simple prototype, **V 0.1** containing an implementation of the sprite based algorithm (from Section 3.2.2) was developed. **V 0.1** contains no graphical user interface. However, its GUI was used until the end of the development phase for shader debugging and experimenting with variations of the Inductive Rotation Method .
With the first prototype built, it was possible to further research the structure of the generated patterns and develop an idea for the parallel algorithm from the last section.

V 0.2 includes the basic version of the parallel algorithm. **V 0.2** also possesses a basic user interface which is largely based on the "Irrational Image Generator"'s GUI.

In the next iteration step **V 0.3** which contains two new, major features was developed. Firstly, to improve Hofstetter's work flow, the Inductive Rotation Framework was enhanced with the prototile editor UI (described in Section 4.6). Secondly, **V 0.3** contains the algorithm that produces patterns with the substitution method.

In the next test cycle all features which were necessary to produce exactly the same patterns as "The Irrational Image Generator" were added and lead to **V 0.4**. These features include an option to replace transparent areas by white color and the option to apply mask textures for the creation of hexagon and star shapes. Additionally a bookmarking feature (described in Section 4.6) that can be used to store viewpoints and fly back to them later, was implemented for this version.

Version **0.5** extended the prototile editor with duplication and reflection features (see Figure 4.16), with an option for printing the pattern and with an option for saving pattern images at different resolutions.

The final version of the Inductive Rotation Framework is called "Inductive Rotation - by Hofstetter Kurt". The implementation contains all features of the previous versions with an improved user interface, redo operations, a paint bucket tool (to fill areas containing similar colors), the option to copy images from the clipboard, support for drag and drop and many other UI improvements.

4.5 Software Architecture

The code was structured into five different projects to achieve maximum code re-usability and to follow the principles of "Don't repeat yourself" (DRY) [Ven]:

- Inductive Rotation
- Inductive Rotation User Interface
- Inductive Rotation WPF User Interface
- Prototile Editor User Interface
- Graphics Utilities

The Inductive Rotation project is the core piece of the software. It contains code for generating Inductive Rotation Patterns, different camera classes, code for the bookmarking feature and pivot creation. Both user interfaces, described in Section 4.6, inter operate with this project. This section describes the responsibilities of the project's classes in detail and serves as documentation.
The Inductive Rotation User Interface project implements a simple UI, which offers no widgets. The Inductive Rotation WPF User Interface project contains all code for the WPF GUI. This GUI is completely operable by widgets, while important features can also be accessed by keyboard shortcuts.
The Prototile Editor User Interface project implements the prototile editor and the Graphics Utilities projects contains classes and support methods used in all other projects.

The Inductive Rotation Project

Figure 4.10 gives an overview of the classes in the Inductive Rotation project. The core class of the framework is the abstract PatternRenderer class. It exposes a Render Method that is used by either one of the user interfaces to render the current pattern in the drawing loop of the application. Furthermore, this class exposes public properties to access TextureHandler and Camera classes as well as a property to set the number of iterations. Implementations of this abstract class are provided by the CSPatternRenderer class which is an implementation of the parallel algorithm described in Section 3.4.1, the IterativePatternRenderer class which implements the sprite based algorithm from Section 3.2.2 and the SPatternRenderer class which realizes the substitution algorithm from Section 3.4.2. To a achieve clean, modular class design the framework provides the IMultiModeRenderer interface which is implemented by all rendering classes that support multiple Inductive Rotation Methods (i.e., 2-way/3-way/5-way).

Texture operations are handled by classes implementing the ITextureHandler interface. A basic implementation of this interface is the SimpleTextureHandler class, which contains methods for uploading textures, controls texture transparency (on/off) and texture aspect ratio settings. A part of the logic for enabling and disabling texture

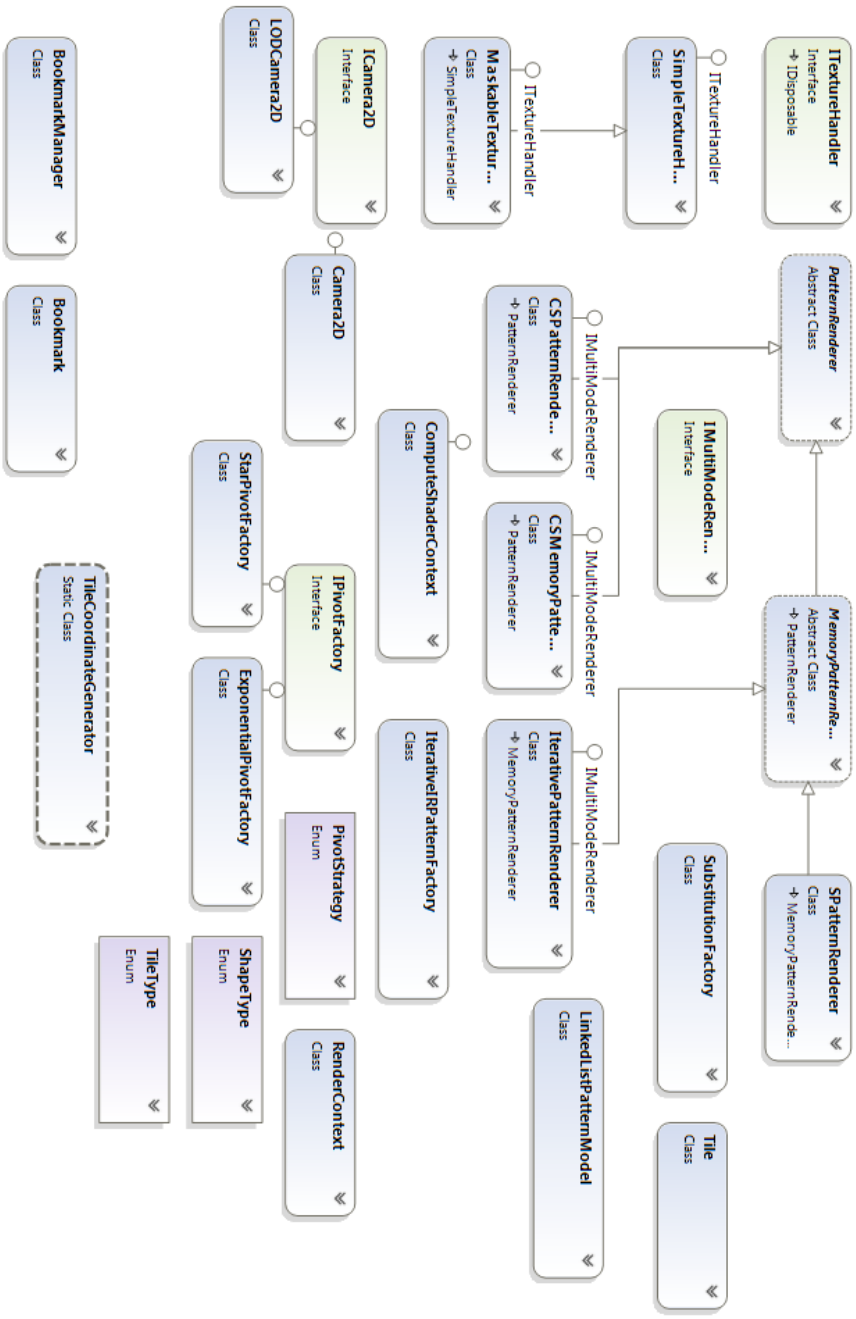

Figure 4.10: A class diagram showing the Inductive Rotation project. The abstract PatternRenderer class is implemented by all algorithms described in Section 4.3. This chart was generated with Microsoft Visual Studio 2013.

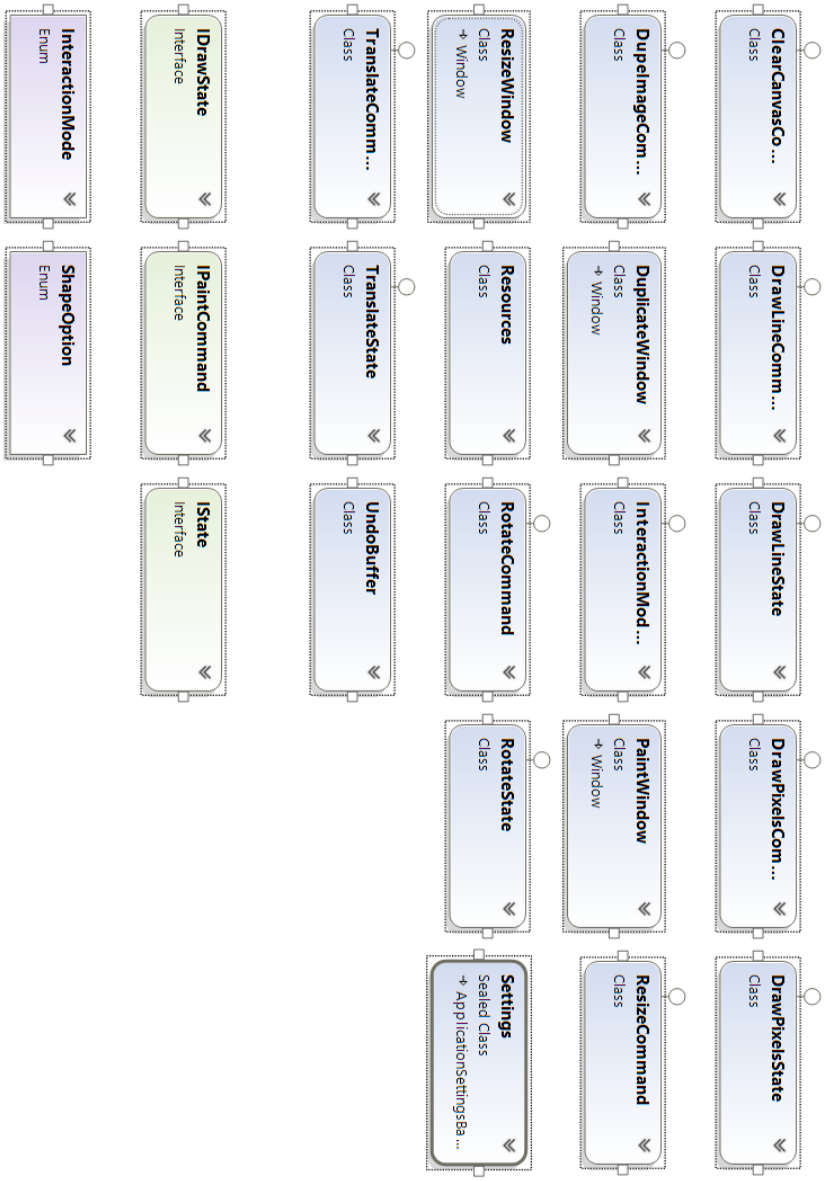

Figure 4.11: A class diagram showing the prototile editor project. The PaintWindow class contains the UI for the prototile editor. The State and Command patterns from Gamma et al. [GHJV95] are implemented by classes which derive from the IPaintCommand and IState interfaces. Classes which's names contain either Command or State derive from these interfaces and implement specific drawing commands and states. The buffer class encapsulates logic for undo and redo operations. This chart was generated with Visual Studio 2013.

transparency is also implemented via pixel shader operations for efficiency reasons. The MaskableTextureHandler class extends basic texture handling logic by the already described masking feature.

The ICamera2D interface provides a contract for classes that implement 2-D cameras. The Inductive Rotation Framwork contains the Camera2D implementation which supports zooming and panning operations. A more sophisticated implementation of this interface is given by the LODCamera2D class. This class is used in conjunction with the CSPatternRenderer class for the LOD approach described in Section 4.3.3. Another important interface is IPivotFactory. Classes that implement this interface control the pivot generation strategy. The classes ExponentialPivotFactory and StarPivotFactory implement this interface and use the formulas presented in Section 3.2.2 for pivot generation.

The Prototile Editor Project

Figure 4.11 depicts the classes from the prototile editor project. PaintWindow is the project's main class. This window used to display the prototile texture and provides access to methods for editing and painting via menus. The implementation is split into the PaintWindow.xaml file and the PaintWindow.cs file to separate design and implementation. The PaintWindow class also provides a callback interface which enables other classes to register for updates to the texture. The IDrawState interface and its implementations (...State) define the State Pattern [GHJV95] whose classes encapsulate logic for paint operations and state management. The IPaintCommand interface and all of its implementations (...Command) are responsible for encapsulating the logic for redo and undo operations [GHJV95].

Other projects

The rest of the code is structured into three projects: Two user interface projects and a utility project. The Graphics Utilities project contains all code which is used by more than one project. It is further structured into Math, Slim DX, WPF and WriteableBitmapUtils namespaces. The two UI libraries contain mostly event handling logic and (xaml) GUI definitions.

4.6 User Interfaces

The Inductive Rotation Framework provides two user interfaces. The first user interface, a development UI, contains no widgets and is controllable by keyboard commands only. Hofstetter does not use this interface. Instead the UI serves for testing and experimenting. The UI compiles and loads in a short amount of time, and can be used to test new features quickly. This project was also used for debugging vertex, pixel and compute shaders, since this is not possible with WPF GUIs due to technical limitations. The non-graphical UI furthermore can also generate patterns that do not strictly follow the Inductive Rotation

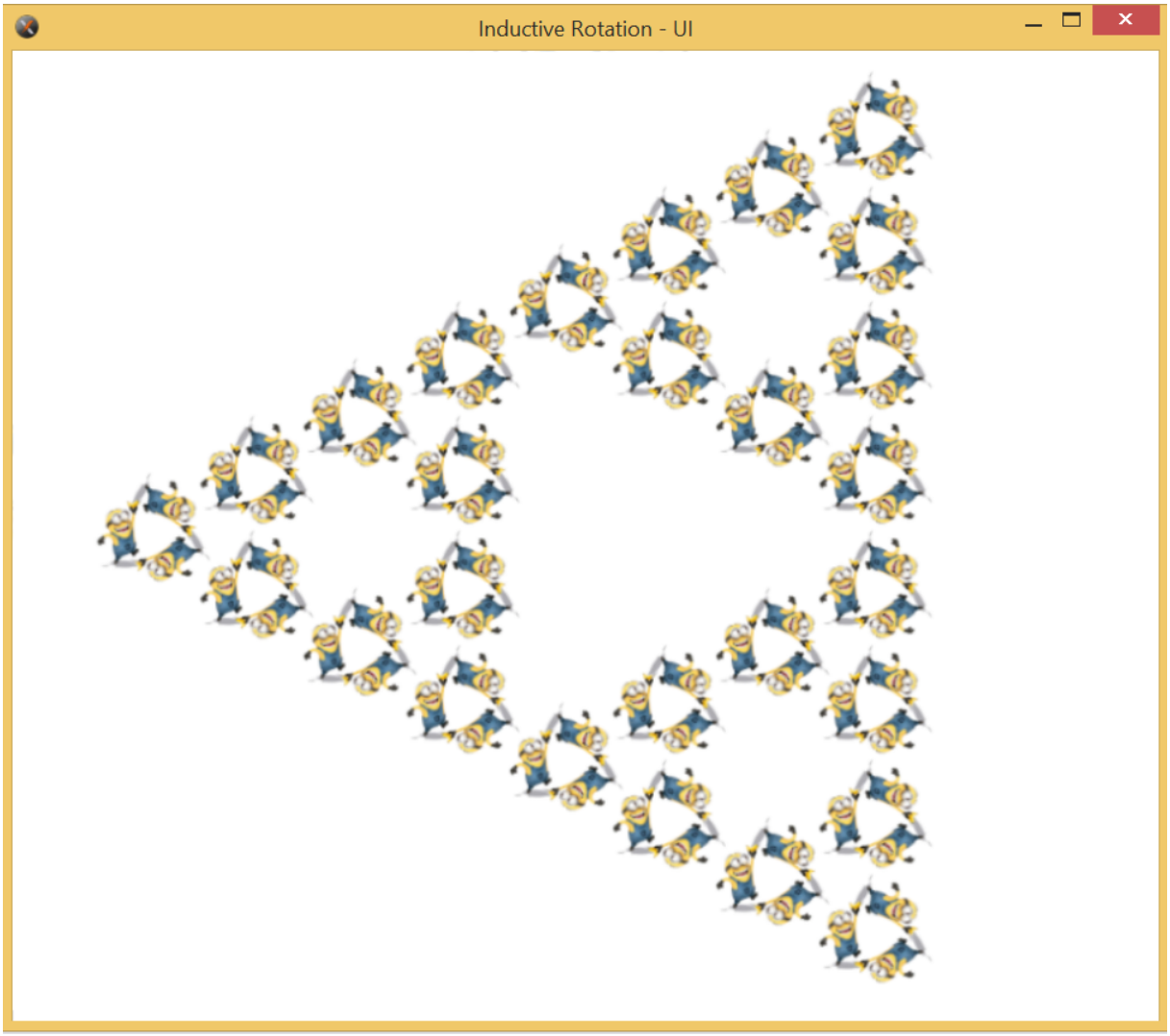

Figure 4.12: A screen shot showing the development UI. In contrast to the WPF GUI designed for Hofstetter, this UI is used for shader debugging and performing experimental trials such as stacking banana minions in a Sierpinski triangle. [Chi].

Method. Any rotation about an arbitrary fraction of 360° is possible. Additionally this interface allowed us to experiment with different pivot generation strategies. Figure 4.12 depicts this user interface.

The second UI, the WPF GUI, which is depicted in Figure 4.13 was created with WPF and tailored to meet Hofstetter's needs. All features of the program are controllable by widgets. The controls which Hofstetter uses frequently are located in the main toolbar from where they can be accessed easily. The main toolbar is comprised of a check-box for switching the alpha channel of the texture on and off, a check box for enabling and disabling aspect ratio settings and a check box that enables or disables the texture-mask. The toolbar also contains a drop-down menu for changing the Inductive Rotation Method, a slider for selecting the number of iterations. To reduce the amount of controls in the main toolbar labels showing the camera position and zoom level were positioned at the bottom of the window. Access to the prototile editor window, which is described later in this section, is provided by a button with a pen symbol. Less frequently used features were put into different drop down menus contained in the main-menu bar. The main menus are labeled File, Edit, Options and Bookmarks. The File menu allows the user to load new prototiles, provides access the quick save feature, which stores the contents of the view port in a "png" file. The menu also contains the Print feature which sends the pattern to a printer and the Save option that also stores the pattern in a "png" format but additionally allows the artist to select a resolution for the resulting image. The Edit menu contains an option to paste a new prototile texture into the window, which replaces the currently selected prototile and the Option menu provides access to the program's settings window, where different parameters can be adjusted. Furthermore, the options menu permits to switch the currently active render mode (sprite based, parallel, ...) via a sub-menu. The bookmarks menu allows the artist to store the current camera position and pattern parameters as well as the prototile texture as bookmarks. The Bookmark can be accessed at a later time to quickly navigate back to the stored position. Bookmarks can be deleted by simply right clicking and selecting the delete option. The Bookmark menu is depicted in Figure 4.6. The main view port of the window shows the generated pattern and provides navigational features via the mouse (zoom and drag). The prototile editor window, depicted in Figure 4.15, provides a simple texture editor. Similar to the main user interface, the window features a toolbar for frequently used operations as well as several drop down menus for functions that are used less frequently. Five different buttons allow the user to select an interaction mode, which is triggered when the user presses the left mouse button in the view port of the prototile editor window. The possible interaction modes include drawing lines, free-hand drawing, using a paint bucket (fill) and rotating and translating the textures. Each operation is previewed by a semi-transparent shape which shows the potential result of the operation. If the line-drawing mode is selected, the user can draw lines by pressing the left mouse button at the start position and then dragging the cursor to the desired end position. The line is drawn by releasing the mouse button. Translating the texture works similarly. Holding the left mouse button starts the translation and releasing the mouse button confirms the

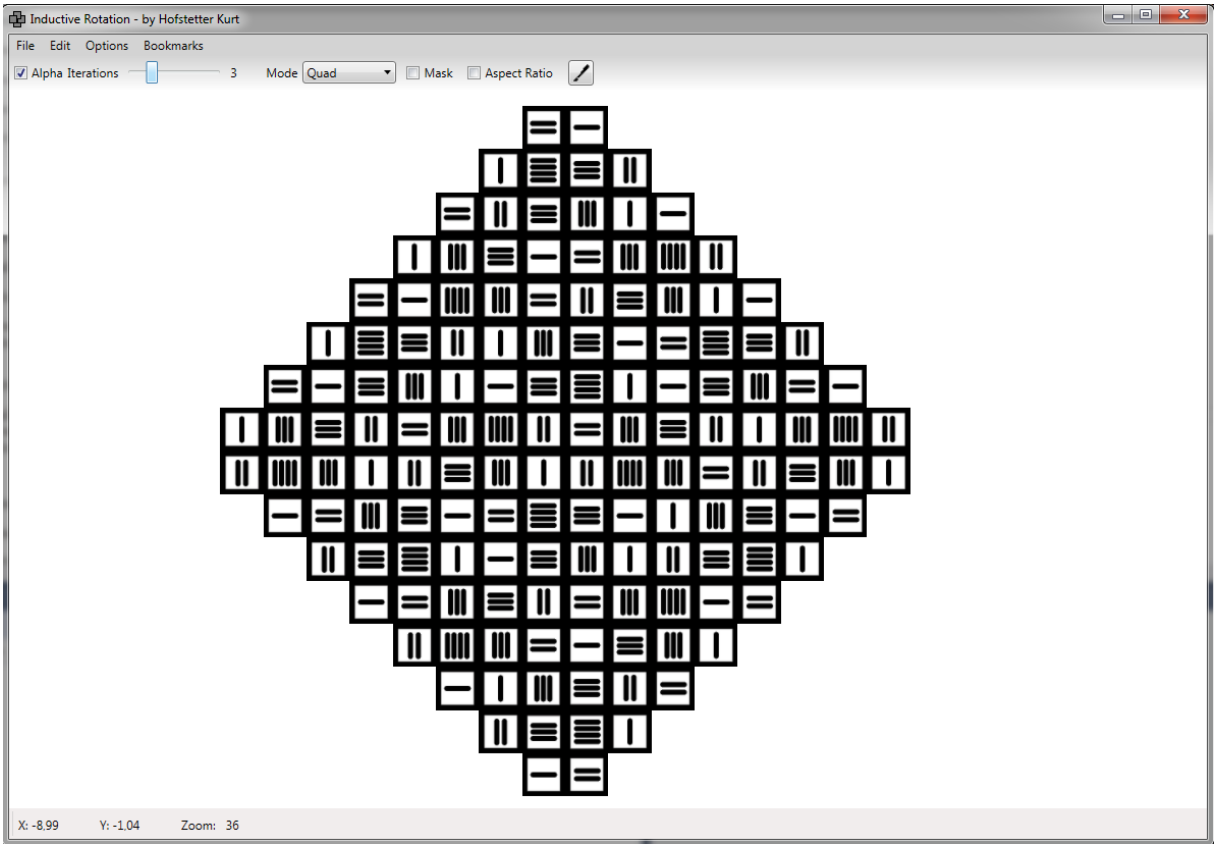

Figure 4.13: A screen shot showing the WPF GUI. Access to frequently used functions is provided by the toolbar at the top. The button with the pencil starts the prototile editor. Other options can be accessed by drop down menus. The user interface also supports loading textures from the clipboard or via drag and drop.

translation. All operations are aborted by pressing the right mouse button. Rotations are also started via the left mouse button. Another left click confirms the rotation. To enable the precise selection of a rotation angle, the operation can also be paused with the right mouse button. While paused, the user has different options. The rotation can be resumed, aborted or changed by typing the angle directly into a text box.
The drop-down menus for less frequently used functions are labeled File, Edit and Options. The File menu allows the user to create new textures with a specified width and height and permits loading and saving textures. The Edit menu provides access to Undo and Redo operations, the experimental animation feature, options to re-size and clear the texture as well as a special feature, requested by Hofstetter, which duplicates and/or reflect the texture. Possible results of this so called “Duplicate Prototile” feature are shown in Figure 4.16. The Options menu contains two features that can be enabled or disabled:

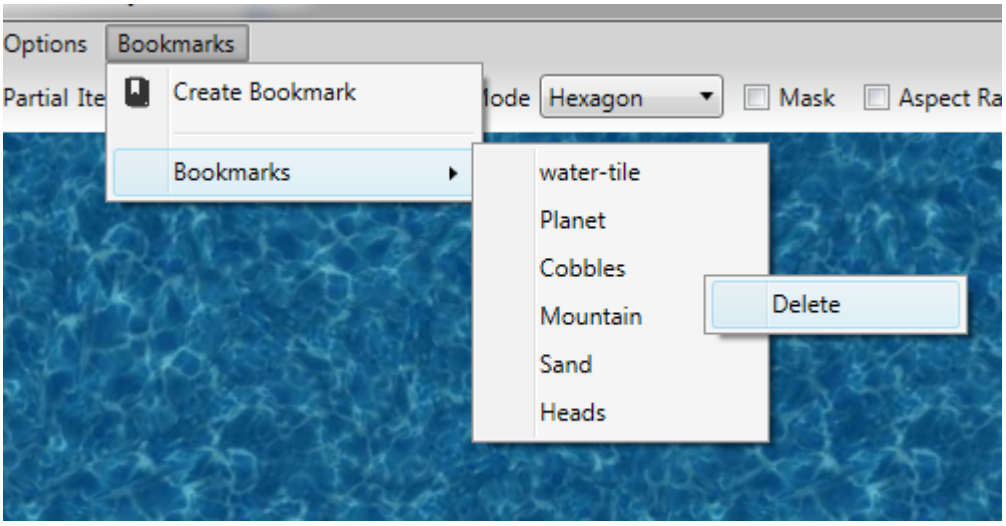

Figure 4.14: A screen shot showing the bookmarks feature. Each bookmark stores the current camera coordinates, render parameters and the active texture. By accessing the bookmark at a later time, the pattern is restored. Bookmarks are stored permanently in an XML file.

- "Continuous Update" is an experimental feature which immediately reflects texture changes in the pattern, instead of using the end-of-state (mostly mouse up) event which is the default. Currently this feature can only be used with small textures and a limited number of iterations.
- The other feature labeled "Increase Image Quality" improves the rendering quality of the final pattern. This increases the texture size by factor five and drastically slows down rendering. However, this feature compensates quality loss in several cases where WPF's control renderer fails to produce high quality images.

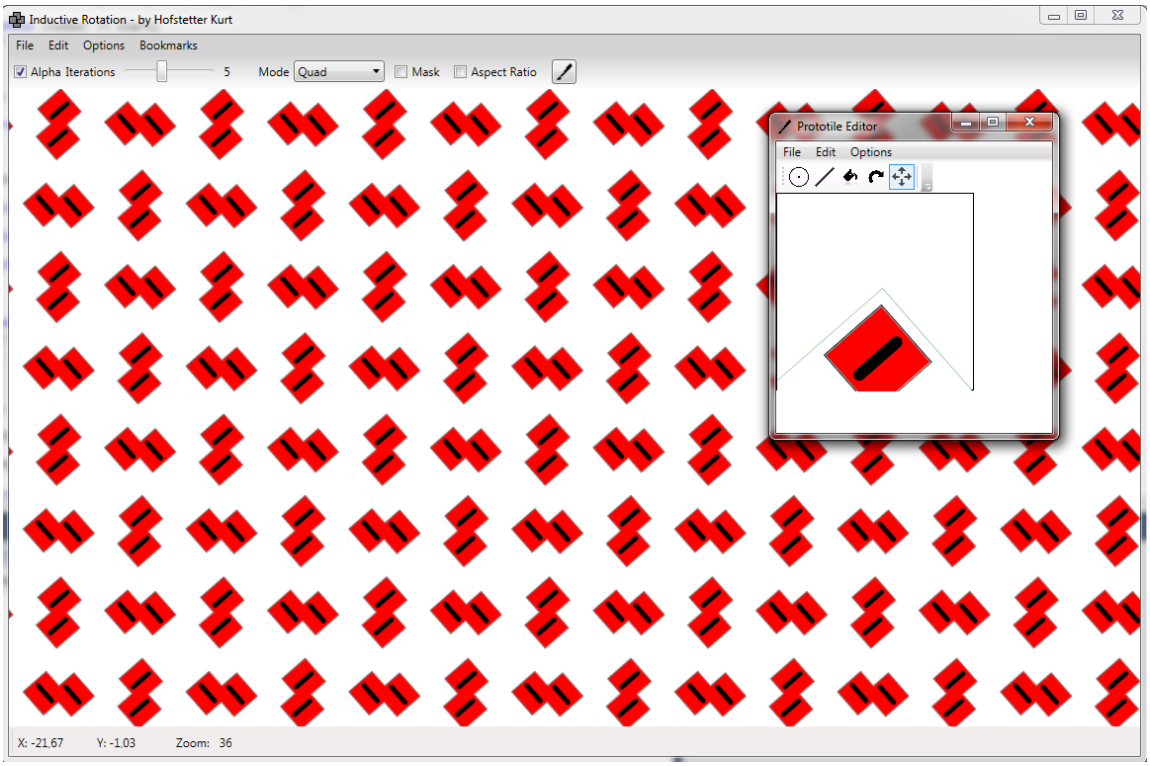

Figure 4.15: A screen shot of the prototile editor window. While drawing onto the tile texture or after translating or rotating the tile, the generated pattern in the main window reflects the changes to the texture.

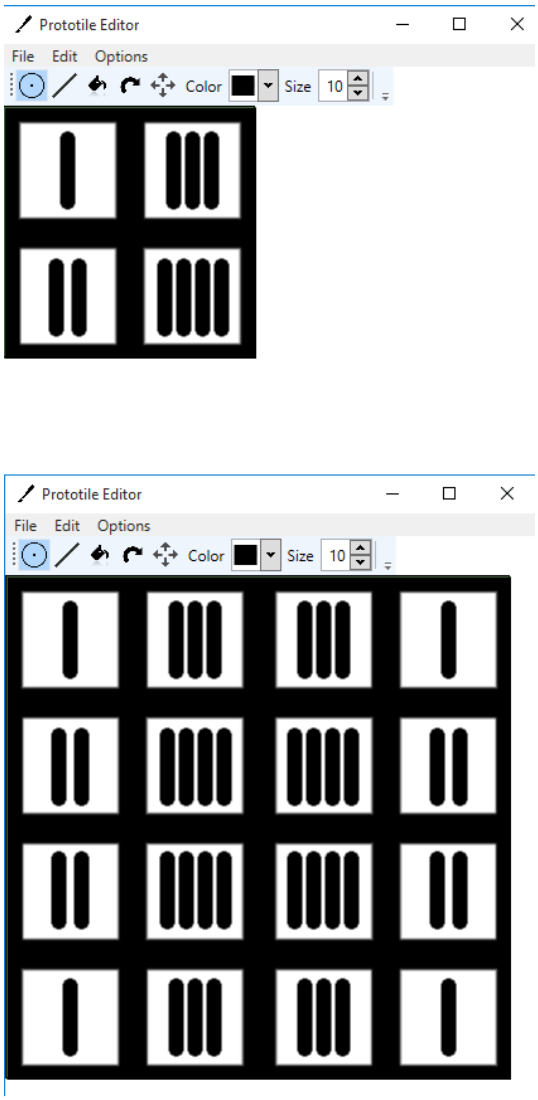

Figure 4.16: An image depicting the “Duplicate Prototile” feature of the prototile editor. The top window shows the prototile editor before the feature was activated. The bottom window shows the prototile editor after activating the feature. The tile from the top window has been copied and reflected for four times. Hofstetter uses reflected prototiles frequently in his pattern design process.

CHAPTER 5

Results

This chapter analyzes the algorithms and their implementation, provides user feedback, new patterns and ideas for future work. The first section of this chapter presents benchmarks of the algorithms from the previous section. These benchmarks measure pattern generation times and memory usage of the different algorithms and compare their performance. For a more comprehensive comparison, the algorithms from the chapter section are compared with Parzer's sprite based algorithm (written in C++) from Section 3.2.2. The second section reflects Hofstetter's personal experience with the Inductive Rotation Framework. The third section presents some new patterns that were generated with the Inductive Rotation Framework by Hofstetter. The last section discusses the results of the benchmarks section, the advantages and disadvantages of the different algorithms and provides ideas for improvements and future work.

5.1 Benchmarks

This section presents benchmarks for all algorithms described in the previous chapter. This section also provides updated benchmarks of Parzer's sprite based algorithm from Section 3.2.2. These were generated on the hardware described below with the software created in the course of Parzer's thesis [Par13]. All tables in this section show either execution times for GPU or CPU, depending on the algorithm, as well as VRAM (Video RAM) and RAM usage for all algorithms and Inductive Rotation Methods.

Benchmarks were conducted on a machine with the following hardware:

- Intel Core i5-2500 with 3.30 GHz and 4 cores
- 8 GB RAM
- AMD Radeon HD 6800 with 1024 MB (GDDR5) RAM

All benchmarks were performed with a 64 bit version of Windows 7 Home Premium. Time and RAM usage were measured with 32-bit "Release" versions of the software. Device creation code and shader compilers also use the release flag in these versions.

Since pattern creation and rendering is handled differently by the algorithms (sprite based/parallel LOD/parallel), the comparison is based on the total time each algorithm needs to create a pattern and upload its coordinates into GPU memory. This implies, that the rendering times are excluded in all tables. Please note, that the parallel LOD method marked with an asterisk(*) needs to regenerate the pattern in all situations described in Section 4.3.3.
CPU Times for C++ code were measured using the QTime class while the Stopwatch class was applied for analyzing CPU Times in the .NET Framework. GPU Times were measured with Direct X "time-stamp queries". Generation times for the sprite based approaches, the parallel LOD algorithm and the substitution algorithm for all different Inductive Rotation Methods are depicted in Tables 5.1, 5.2 and 5.3.
Tracking the exact memory usage in the .NET Framework is difficult. The reason for this is that the .NET Framework uses a garbage collection algorithm which facilitates programming but does not allow the developer to directly control memory management. The benchmarks in this section represent only an approximation of the algorithm's true memory usage (except for the C++ code).
Memory usage of the host machine was measured using the Sysinternals Process Explorer tool [Sys15]. GPU memory usage was measured using GPU-Z [Tec].

GPU-Z measures two types of GPU memory usage:

- Dedicated GPU memory usage (GPU memory with fast access times). In the columns titled VRAM this is the value on the left side of the plus sign.

- Dynamic GPU memory usage (memory of the host machine with slower access times). In the columns titled VRAM this is the value on the right side of the plus sign.

Memory usage for the sprite based algorithms and the substitution algorithm for all different Inductive Rotation Methods are depicted in Tables 5.5, 5.6 and 5.7. Both parallel algorithms use the same amount of memory independently of the selected parameters. These times are summed up in Table 5.8.
The parallel algorithm is set to compute a pattern with a fixed set of tiles for all three Inductive Rotation Method which means it produces only three different times which are summed up in table 5.1. The benchmarks were performed using a single pass with $43 * 43 = 1849$ thread groups and 1024 threads per group to compute the pattern (1983376 tiles in total). The benchmarks for the parallel LOD algorithm were generated by using $16 * 16 = 256$ thread groups and 1024 threads per group for pattern creation.

Iterations	sprite based (Parzer)	sprite based (.NET)	Substitution	Parallel LOD*
1	< 1 ms	< 1 ms	< 1 ms	7 ms
2	< 1 ms	< 1 ms	< 1 ms	7 ms
3	< 1 ms	< 1 ms	3 ms	7 ms
4	< 1 ms	< 1 ms	22 ms	7 ms
5	< 1 ms	1 ms	48 ms	7 ms
6	1 ms	4 ms	195 ms	7 ms
7	3 ms	24 ms	888 ms	7 ms
8	17 ms	113 ms	4140 ms	7 ms
9	67 ms	608 ms	-	15 ms
10	225 ms	2705 ms	-	41 ms
11	805 ms	-	-	149 ms
12	-	-	-	628 ms
13	-	-	-	2667 ms

Table 5.1: Measured times for generating 3-way Inductive Rotation Patterns . *This is the time for recomputing the LOD texture on the GPU, when moving the camera. This happens when the pattern is recomputed and introduces a lag of this duration. The lag occurs when the camera moves out of the LOD texture's boundary.

Iterations	sprite based (Parzer)	sprite based (.NET)	Parallel LOD*
1	< 1 ms	< 1 ms	6 ms
2	< 1 ms	< 1 ms	6 ms
3	< 1 ms	< 1 ms	6 ms
4	< 1 ms	1 ms	6 ms
5	1 ms	14 ms	6 ms
6	11 ms	86 ms	6 ms
7	59 ms	675 ms	12 ms
8	303 ms	4215 ms	48 ms
9	-	-	294 ms
10	-	-	1896 ms

Table 5.2: Measured times for generating 5-way Inductive Rotation Patterns . *This is the time for recomputing the LOD texture on the GPU, when moving the camera. This happens when the pattern is recomputed and introduces a lag of this duration. The lag occurs when the camera moves out of the LOD texture's boundary.

Iterations	sprite based (Parzer)	sprite based (.NET)	Parallel LOD*
1	< 1 ms	< 1 ms	9 ms
2	< 1 ms	< 1 ms	9 ms
3	< 1 ms	< 1 ms	9 ms
4	< 1 ms	< 1 ms	9 ms
5	< 1 ms	< 1 ms	9 ms
6	< 1 ms	< 1 ms	9 ms
7	< 1 ms	1 ms	9 ms
8	1 ms	5 ms	9 ms
9	4 ms	23 ms	9 ms
10	15 ms	101 ms	9 ms
11	43 ms	405 ms	9 ms
12	114 ms	1302 ms	30 ms
13	335 ms	3803 ms	72 ms
14	828 ms	-	210 ms
15	-	-	635 ms
16	-	-	2590 ms
17	-	-	8089 ms

Table 5.3: Measured times for generating 2-way Inductive Rotation Patterns . *This is the time for recomputing the LOD texture on the GPU, when moving the camera. This happens when the pattern is recomputed and introduces a lag of this duration. The lag occurs when the camera moves out of the LOD texture's boundary.

Inductive Rotation Method	Max Iterations	Time
2-way	13	48 ms
3-way	10	58 ms
5-way	8	71,8 ms

Table 5.4: GPU Times for the parallel algorithm.

Iterations	sprite based (Parzer)		sprite based (.NET)	
	RAM	VRAM	RAM	VRAM
1	68 MB	30 MB	36 MB	8 MB
2	68 MB	30 MB	36 MB	8 MB
3	68 MB	30 MB	36 MB	8 MB
4	68 MB	30 MB	36 MB	8 MB
5	68 MB	30 MB	36 MB	8 MB
6	68 MB	30 MB	36 MB	8 MB
7	68 MB	30 MB	37 MB	8 MB
8	70 MB	30 MB	40 MB	8 MB
9	73 MB	30 MB	46 MB	9 MB
10	81 MB	30 MB	71 MB	13 MB
11	106 MB	30 MB	122 MB	24 MB
12	191 MB	30 MB	274 MB	56 MB
13	372 MB	30 MB	775 MB	152 MB
14	840 MB	30 MB	-	-

Table 5.5: Total memory usage when creating patterns with the 2-way Method. The values in the columns titled VRAM represent the amount of dedicated video memory used by the algorithms while values in the RAM-column show their usage of host-machine memory.

Iterations	sprite based (Parzer)		sprite based (.NET)		Substitution	
	RAM	VRAM	RAM	VRAM	RAM	VRAM
1	68 MB	30 + 18 MB	37 MB	8 + 3 MB	40 MB	8 + 3 MB
2	68 MB	30 + 18 MB	37 MB	8 + 3 MB	40 MB	8 + 3 MB
3	68 MB	30 + 18 MB	37 MB	8 + 3 MB	40 MB	8 + 3 MB
4	68 MB	30 + 18 MB	37 MB	8 + 3 MB	40 MB	8 + 3 MB
5	68 MB	30 + 18 MB	37 MB	8 + 3 MB	41 MB	8 + 3 MB
6	69 MB	30 + 19 MB	38 MB	9 + 3 MB	50 MB	9 + 3 MB
7	72 MB	30 + 19 MB	46 MB	9 + 3 MB	80 MB	9 + 3 MB
8	83 MB	30 + 22 MB	70 MB	14 + 5 MB	202 MB	32 + 10 MB
9	128 MB	30 + 37 MB	160 MB	32 + 9 MB	605 MB	104 + 10 MB
10	308 MB	30 + 97 MB	504 MB	104 + 9 MB	-	-
11	828 MB	30 + 141 MB	-	-	-	-

Table 5.6: Total memory usage when creating patterns with the 3-way Method. The values in the columns titled VRAM represent the amount of dedicated video memory used by the algorithms while values in the RAM-column show their usage of host-machine memory.

Iterations	sprite based (Parzer)		sprite based (.NET)	
	RAM	VRAM	RAM	VRAM
1	68 MB	30 + 18 MB	36 MB	8 + 3 MB
2	68 MB	30 + 18 MB	36 MB	8 + 3 MB
3	68 MB	30 + 18 MB	36 MB	8 + 3 MB
4	68 MB	30 + 18 MB	36 MB	8 + 3 MB
5	71 MB	30 + 18 MB	41 MB	8 + 3 MB
6	78 MB	30 + 19 MB	60 MB	14 + 1 MB
7	127 MB	30 + 22 MB	180 MB	35 + 9 MB
8	386 MB	30 + 92 MB	791 MB	128 + 9 MB

Table 5.7: Total memory usage when creating patterns with the 5-way Method. The values in the columns titled VRAM represent the amount of dedicated video memory used by the algorithms while values in the RAM-column show their usage of host-machine memory.

Parallel		Parallel LOD	
RAM	VRAM	RAM	VRAM
38 MB	354 + 3 MB	38 MB	310 + 3 MB

Table 5.8: Total memory usage for the parallel algorithms. All parallel algorithms have a constant memory footprint. The values in the VRAM columns represent the usage of dedicated video memory of the graphics card. The values in the RAM columns represent the usage of host machine memory.

Discussion

The conclusion from the tables in this section is that the sprite based algorithms excel the parallel algorithms at fewer iterations. However, this speed bonus is not noticeable by the user when comparing the algorithms. The only computation which the parallel algorithm needs, happens when the user switches the Inductive Rotation Method. The maximum time the algorithm consumes for this task (71,8 ms) is almost not noticeable for the user, while the sprite based algorithms need more time to compute higher iteration numbers. This happens more often when the user switches between pattern modes and iterations, especially at higher iteration levels. The parallel LOD algorithm on the other hand allows the artist to create and navigate in patterns which are 12 times larger than with the previous methods before the algorithm gets too slow on the tested hardware. As described in Section 5.4, the parallel LOD algorithm still provides a lot of potential for future optimizations which could further improve the maximum pattern size and lower the overall pattern generation times. The Substitution algorithm provides no competitive results at the current stage of its implementation.

5.2 User Experience

The Inductive Rotation Framework was specifically designed for a single person, the artist Hofstetter Kurt. Hofstetter expressed the desire that the Inductive Rotation Framework will not be made publicly available and we will follow this request. This also means it was not possible to do research on user experience with a broad range of users. However, Hofstetter expresses his personal experience with the Inductive Rotation Framework as follows:

Hofstetter Kurt, May 2015:

"The Inductive Rotation Framework and its extensions provides tools to create IR-Patterns more easily by means of getting a better strategical knowledge to design prototiles. Using the embedded prototile editor to interactively create IR-Patterns, in particular getting immediate display response of any interaction (like editing by drawing, coloring and / or moving) opens not only a new way of pattern generation by means of intuition but also provides a better understanding how the tiles fit together and how they communicate with each other. Thus, in particular, the software gives new insights into the development of complex IR-Pattern-structures. Furthermore a new field of experimentation with dots, lines, curves, colors and movement comes into being. In this context I would like to note that such experiments were necessary for the creation of Oriental_P (see next section) *which is the result from an extensive process of interactively editing and shaping oriental ornaments. In addition, the immediate on screen response that follows any free-hand translation of prototiles provides a moving pattern experience, which stimulates to replace static prototiles with moving images. Interdisciplinary experiments with these kinds of patterns have already been started by Barbara Doser, an artist who creates abstract sequences of patterns for video-art."*

5.3 Inductive Rotation Images

This section depicts three Inductive Rotation Patterns which were created by Hofstetter with the Inductive Rotation Framework. Figures 5.1, 5.2 and 5.3 show different patterns created by Hofstetter with support of the prototile editor included in the Inductive Rotation Framework.

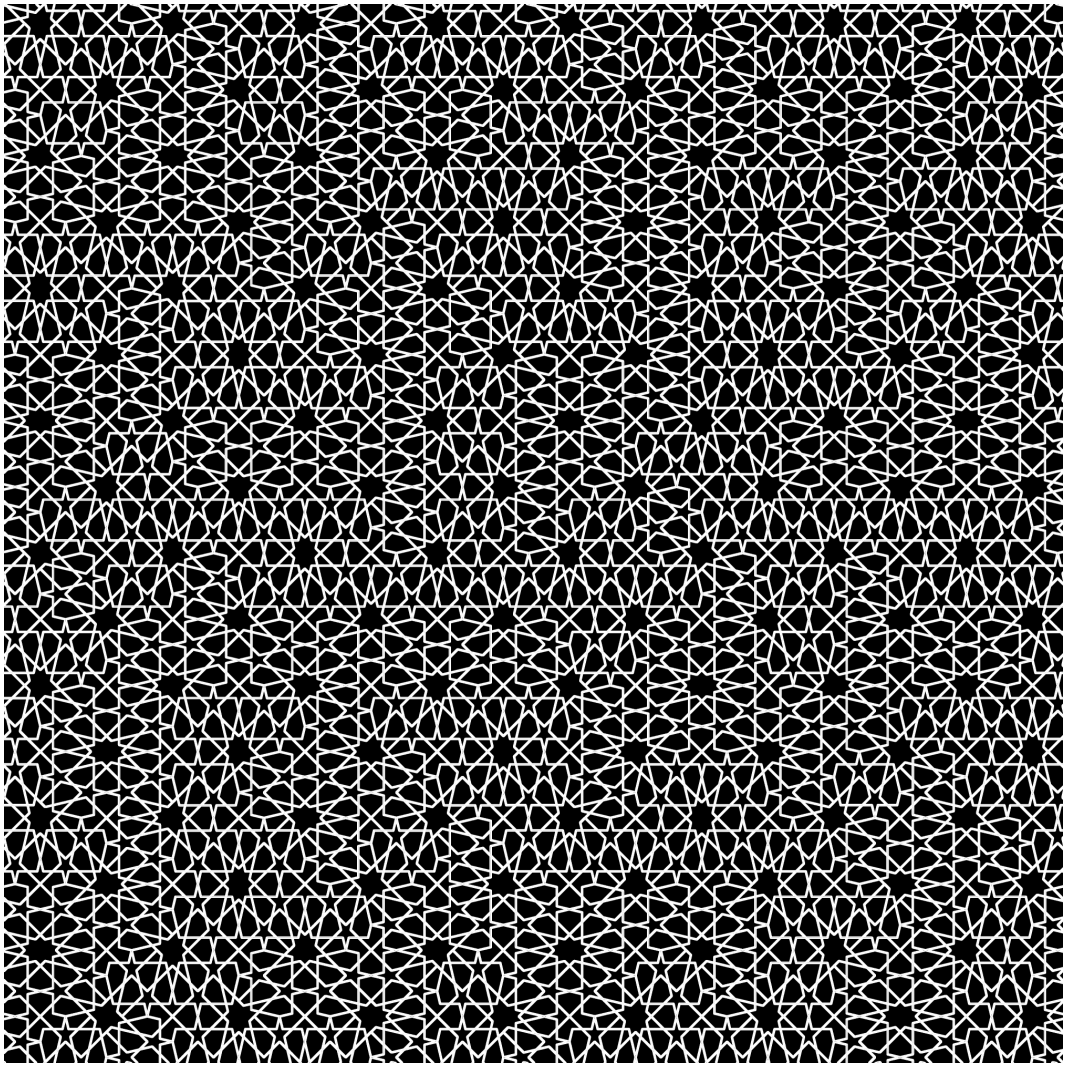

Figure 5.1: This pattern, called Oriental_P, was created by Hofstetter with the Inductive Rotation Framework.

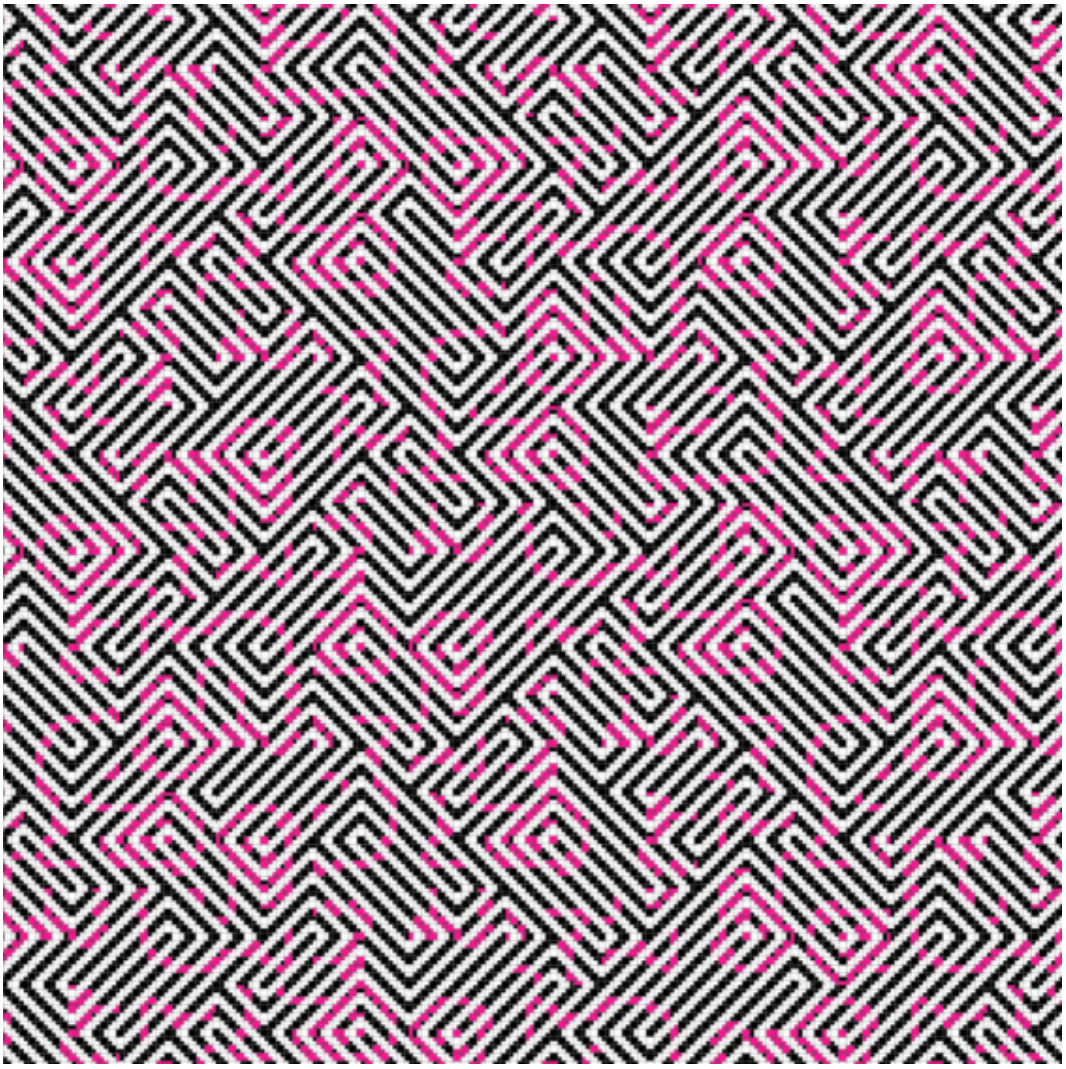

Figure 5.2: This pattern, called P_traces, was created by Hofstetter with the Inductive Rotation Framework.

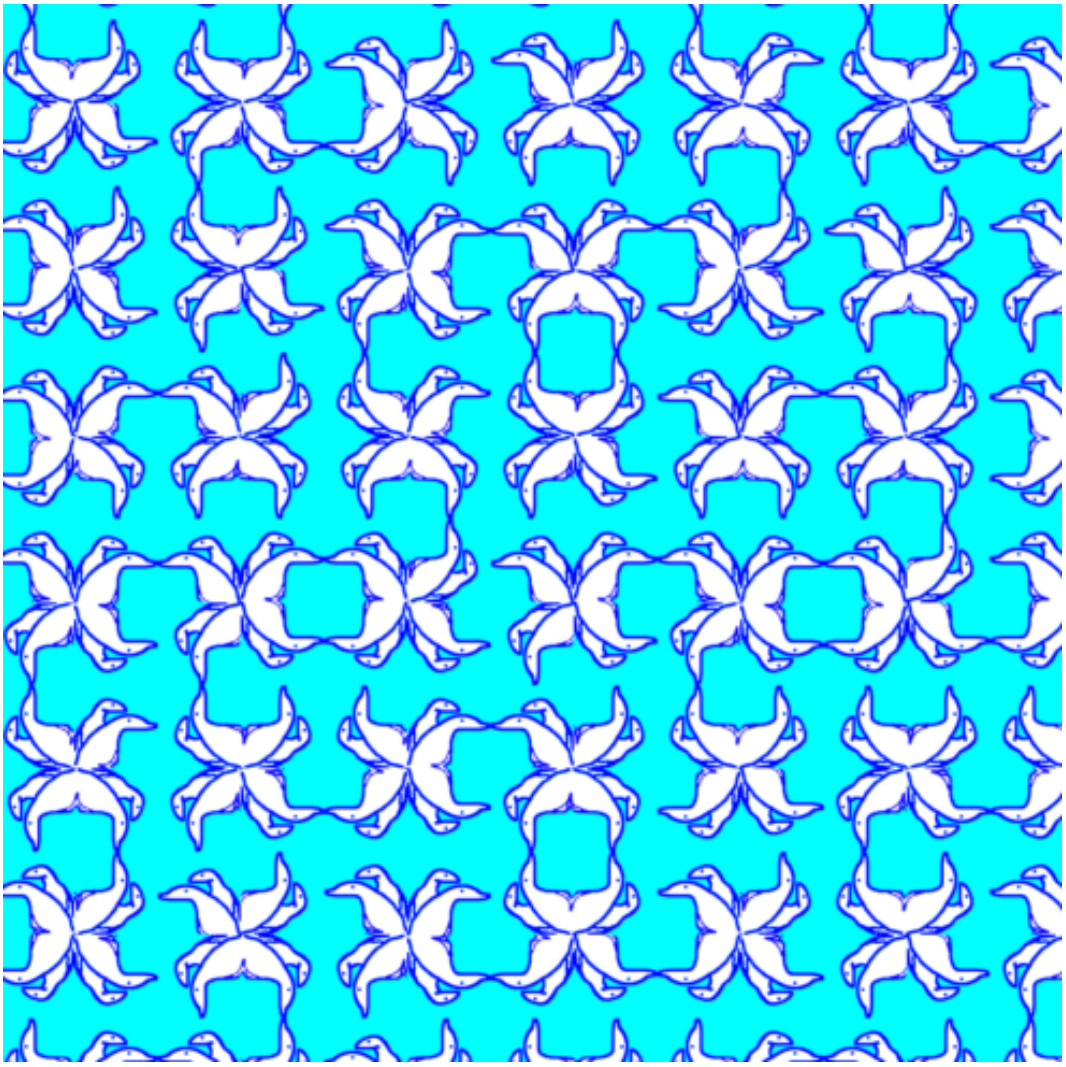

Figure 5.3: This pattern, called 8_bears, was created by Hofstetter with the Inductive Rotation Framework.

5.4 Comparison and Conclusions

The fundamental algorithm, described in Section 3.4.1, which was devised in the scope of this thesis has proven useful for the parallelization of pattern generation. Section 3.4.1 provides the mathematical basis for a variety of possible implementations. Two possible implementations were presented in Section 4.3.3.
The simple parallel algorithm produces large patterns significantly faster than the iterative C++ version. The parallel LOD algorithm on the other hand provides a way for visualizing about 12 times larger patterns than before. The algorithm, however, involves a recreation

of the pattern in all cases described in Section 4.3.3 and still offers many possibilities for improvement, yet the implementation of this features lies beyond the scope of this thesis.:

- Currently, the LOD implementation is very simple. If the LOD level switches, the whole pattern has to be recomputed. This can be improved by exploiting the nature of the pattern: If the LOD level switches, the algorithm could determine the position of the view port in the pattern and compute only the visible parts. To this end the view port could be recursively tested against a tree structure, containing oriented bounding boxes (OBB), which enclose the whole pattern.
- Instead of computing the rotation matrices/angles in every thread, as the current algorithm does, the matrices can be computed in advance and stored in a cache for re-use. This will increase the pattern-generation speed especially for higher tile indices.
- The current implementation maps the computation of a tile's coordinates to a single thread. This is not optimal, since the threads that compute coordinates of tiles with higher indices need to execute more operations than the threads that compute tile coordinates with lower indices. Using a smarter distribution strategy would also improve the speed of the algorithm.
- The number of threads and thread-groups should be optimized for different hardware.

While the parallel algorithms improve the aspects of pattern generation mentioned above, the parallel (as well as the sprite based) algorithm are limited to generating patterns on a single layer. Hence it is not possible to visualize "hidden layers", which store the overlapping pattern parts, like the grid based algorithm from Section 3.2.1 does. Furthermore, producing different tile shapes has to be achieved by applying mask textures if using the parallel or sprite based approaches, while the grid based algorithm supports different tile shapes out of the box.

The practical usefulness of the substitution algorithm for artistic purposes is questionable because of the following facts:

- The method is only applicable for 3-way patterns.
- The IR-Pattern cannot be easily isolated from the rest of the tiling.
- Textures with an aspect ratio different from 1:1 (square-textures) cannot be displayed easily.

The prototile editor and the other new features proved very useful to Hofstetter. From a technical point of view, the prototile editor's performance bottleneck is uploading textures to the GPU. This is a very expensive operation which needs to be performed after each texture manipulation. If editing smaller patterns the parallel implementation provides better performance (This is due to the fact that the pattern does not have to

be recreated, if the texture changes) than the parallel LOD algorithm. Interestingly, if working with huge patterns (e.g., eight iterations for 5-way patterns), the parallel LOD algorithm performs better on the tested hardware (less lagging). The reason for this is that the parallel LOD algorithm limits the usage of GPU memory.

CHAPTER 6

Summary

To create artistic patterns the artist Hofstetter Kurt developed the Inductive Rotation Method. This method uses one of several rules, defined by Hofstetter, to copy and rotate a prototile in order to create a pattern. Pattern creation is executed iteratively and the number of tiles raises exponentially with this method. The resulting patterns exhibit a nonperiodic structure. Hofstetters Inductive Rotation is reviewed in Section 6.1. The previously existing algorithms for the creation of Inductive Rotation Patterns [Par13], which are reviewed in Section 6.2, were not parallelizeable and the UI of the predecessor software was limited to pattern generation. Recent research [Hofb] has pointed out that 3-way Inductive Rotation Patterns can also be created by applying tile substitution.
In the course of this thesis a new framework for the creation of Inductive Rotation Patterns was developed. For this Inductive Rotation Framework a new, parallelizeable algorithm has been implemented. The mathematical core of this algorithm, a sophisticated indexing scheme, is reviewed in Section 6.3 . This scheme can be applied in different ways. Section 6.3 also describes the tile substitution method. The different implementations of these algorithms are described in Section 6.4.

6.1 Inductive Rotation

This section reviews the definition of the Inductive Rotation Method [Hofb, Kur]. The method is applied to a textured prototile. Inductive Rotation copies and rotates the tile around different pivots in several iterations. The rotation angles the Inductive Rotation Method uses are different fractions of 360°. Hofstetter defines three different Inductive Rotation Methods:

- The 3-way Inductive Rotation Method copies and rotates square shaped prototiles. This is done three times in each iteration, with rotation angles of $\frac{1}{4}$,$\frac{2}{4}$ and $\frac{3}{4}$ of 360°. The pivot for this operation always has $y = 0$ and its x coordinate is the rightmost point inside the pattern.

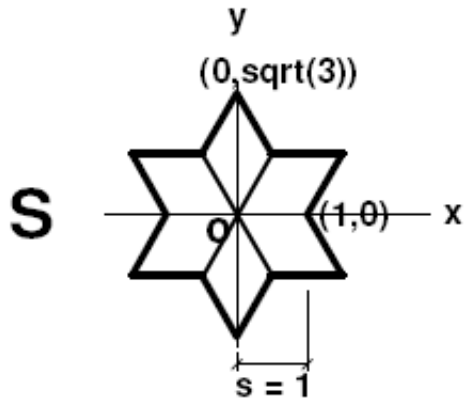

Figure 6.1: A figure showing the position of the first pivot for the creation of 2-way Inductive Rotation Patterns. The image also depicts the proportions of star-shaped prototiles defined by Hofstetter. Image courtesy of Hofstetter [Hofa]

- The 5-way Inductive Rotation Method copies and rotates a hexagon shaped prototile. This is done five times in each iteration, with rotation angles of $\frac{1}{6}$, $\frac{2}{6}$, $\frac{3}{6}$, $\frac{4}{6}$ and $\frac{5}{6}$ of 360°. The pivot for this operation always has $y = 0$ and its x coordinate is the rightmost point inside the pattern. This method uses a hexagon with equal side lengths as prototile.

- The 2-way Inductive Rotation Method copies and rotates a star shaped prototile with six spikes. This is done two times in each iteration, with rotation angles of $\frac{1}{3}$ and $\frac{2}{3}$ of 360°. The pivot for this method is chosen in a different manner. While in the first rotation the corner point of the first star between the two right apices (shown in Figure 6.1) is selected, for subsequent iterations the corner point of the star which is positioned at the rightmost and lowest position in the pattern is used. This method uses a star shaped prototile which is depicted in Figure 6.1.

6.2 Previous Work

Two algorithms were previously developed for the creation of Inductive Rotation Patterns by Parzer [Par13]. The first algorithm used a grid of either triangles or squares to represent the pattern and stores the tile rotations in a multidimensional array as integer numbers. To perform the necessary copy and rotate steps, the algorithm uses several expressions (see reference [Par13]) to compute the new positions and rotations in the array in each iteration. To render the pattern, the algorithm transforms the numbers stored in the grid into rotated and translated triangles or quads. The grid based algorithm also stores the hidden portions of the grid in different layers of the array to display them, if desired.

The grid based approach has several drawbacks which result in rendering artifacts for some prototiles. Another limit of the algorithm is that the prototile shapes are also limited to hexagons, stars and squares [Par13]. These facts led to the development of the sprite based approach which also excels the grid based approach in speed and memory usage. Parzer's sprite based approach stores the sprite representation of each tile in a list. The algorithm works iteratively. In the first iteration it rotates all tiles stored in the list around the pivot and appends the new tiles to the list. The procedure is repeated with the list that was created in the first iteration for further iterations if the desired size for the Inductive Rotation Pattern has not yet been reached. An important fact which has to be considered when displaying the pattern to the user is that the tiles have to be rendered in a back to front order, starting at the end of the list.

6.3 Algorithms

This section summarizes all new algorithms for Inductive Rotation Pattern generation which were developed and implemented for the Inductive Rotation Framework. The first algorithm provides a way to parallelize Inductive Rotation Pattern generation and the second algorithm is based on the substitution method of Frettlöh and Hofstetter [FH15].

6.3.1 Parallel Approach

The parallel approach is a scheme to compute the position and rotation of each tile by using a series of concatenated transformation matrices. This approach can be executed in parallel. The approach also removes a disadvantage of the previous methods, i.e., the necessity to store the tile coordinates of each iteration in order to compute the coordinates for subsequent iterations. The algorithm creates patterns by applying multiple series of matrix transformations which place the tiles of an Inductive Rotation Pattern at their designated positions in the pattern. Each series of matrix multiplications $c_{tile} = c_{init}^T B_{31} B_{21} B_{11} ... B_{lk}$ places the prototile with the starting position c_{init} at its designated position in the pattern c_{tile}. Each matrix B_{lk} represents a rotation around a pivot. To find the B_{lk} matrices that place a tile in the pattern, all translation matrices T_l, that represent the translations from the tile's center to the pivots and the rotation matrices R_k for the rotations around the points need to be found. These matrices are then used to construct the matrix B_{lk} by applying Expression 6.1.

$$B_{lk} = T_l^{-1} R_k T_l \tag{6.1}$$

The different angle's for each rotation matrix R_k is represented by Expression 6.2. The expression maps the integer k to a rotation angle for matrix R_k.

$$r(k) = k * \frac{2 * \pi}{p_r} \tag{6.2}$$

To determine the rotation angle which is needed to construct a matrix R_k and to determine the matrices T_l for the pivots, an indexing scheme is established. The indexing scheme

uses the same idea as the algorithm for converting numbers between different bases of numeral systems. Interestingly, this conversion scheme corresponds naturally to the way Inductive Rotation Patterns are constructed. The idea of the scheme is depicted by the example in Table 6.1.
To determine the indices for the matrices B_{lk} which are needed to construct the chain of rotations for a tile in an Inductive Rotation Pattern, the index of the tile sought is successively divided by p_r. Hereby the remainder of each division is kept and the order of the division operations is noted. The example in Table 6.1 shows this procedure for a

Number	Operation	k (Remainder)	l (Order)
22	$22/3 = 7$	1	1
7	$7/3 = 2$	1	2
2	$R : 2$	2	3

Table 6.1: An example on how to compute the indices k and l for the rotation and translation matrices. In this specific example the matrices for the tile with index 22 are sought. To find the indices k the starting number is successively divided by powers of p_r and the remainders of these divisions are kept. The indices l are found by simply numbering the calculations bottom up.

tile with index 22 using the 2-way Inductive Rotation Method. The results in the table are interpreted from bottom up to construct the chain of multiplications from left to right. For the index in this example the results is the chain of matrix multiplications $B_{32}B_{21}B_{11}$. In the case that the remainder of a division step equals zero (i.e., $k = 0$), the matrix B_{lk} of this step is replaced by the Identity-Matrix.
Each rotation matrix R_k for the B_{lk} matrices is constructed from a rotation angle by applying Expression 6.2 with index k as argument. Each translation matrix T_l is constructed by applying a corresponding pivot rule. For 3-way and 5-way Inductive Rotation Patterns, the translation matrix T_l is constructed by Expression 6.3. Pivots for 3-way and 5-way Inductive Rotation Patterns were already described by Parzer [Par13] in this way.

$$\vec{p}_{pivot}(l) = (2^l, 1) \tag{6.3}$$

A closed expression for the pivots of 2-way Inductive Rotation Patterns, however, was not given by Parzer who used a search method to find the pivots for each iteration. To find a closed expression for these pivots, the positions that Parzer's software determines for 2-way Inductive Rotation pivots were examined. While at the current stage of work no geometrical meaning of the following expressions can be given, they are able to reproduce all the pivot coordinates from Parzer's software in a coherent way and also produce correct looking results for iteration levels that exceed the capacity of Parzer's software (more than 14 iterations). Expression 6.4 computes the x-coordinate and Expression 6.5 computes the y-coordinate of a pivot for a 2-way Inductive Rotation Pattern at an iteration level l. A vector with the x- and y-coordinate of a pivot for a 2-way Inductive

Rotation Pattern at iteration level l is computed by Expression 6.6.

$$x(l) = \begin{cases} \frac{5}{3} & \text{if } l = 1 \\ \frac{5}{3} + \left[\sum_{i=0}^{l-2} 3^{\lfloor \frac{i}{2} \rfloor} * ((i \bmod 2) + 1)\right] & \text{if } l > 1 \end{cases} \tag{6.4}$$

$$y(l) = \begin{cases} \left(\sum_{i=1}^{l-1} 3^{\lfloor \frac{i-1}{2} \rfloor} * (i \bmod 2)\right) * p_{projy} & \text{if } l \geq 2 \\ 1 & \text{if } l < 2 \end{cases} \tag{6.5}$$

$$\vec{p}_{pivot}(l) = (x(l), y(l)) \tag{6.6}$$

With the established indexing scheme described above and the different Expressions for finding the rotation matrices R_k and the Expressions for finding the pivot's translation matrices T_l, the position of each tile can now be determined independently without knowing the exact position of any other tile in the pattern. Two possible implementations of this scheme are presented in section 6.4.

6.3.2 Substitution Tiling Approach

Another algorithm for generating Inductive Rotation Patterns is presented by Frettlöh and Hofstetter [FH15]. This algorithm, however, cannot directly generate Inductive Rotation Patterns. The mathematical theory, developed by Freetloeh et al. [FH15] , produces only 3-way Inductive Rotation Patterns that are contained in larger tilings. Therefore this method creates, in contrary to the previous methods, tilings of the plane in accordance with the mathematical definition, with no overlapping tiles. The tilings, which are created by this method do not represent Inductive Rotation Patterns themselves, but they contain Inductive Rotation Patterns. The idea of this approach is to apply a rule-based tile-substitution scheme, consisting of four different rules, to create Inductive Rotation Patterns. The substitution scheme is depicted in Figure 6.2. Additionally each of the four different tiles identified by T_1, T_2, T_3 and T_4 needs to be mapped to a sub square of the prototile of an Inductive Rotation Pattern for visualization. A pattern is represented by squares of 1/4 the size of the prototile with this approach. The theory works only with squares. To apply the substitution tiling approach, objects which store the properties of each tile are defined first. Each instance of these tile objects stores a tile's background-texture, its rotation, represented as an integer and its corner vertices.

To create a pattern by tile substitution the following simple steps are executed:

1. Create a set of four tiles, also called seed tiles. Create a linked list data structure and append the seed tiles. Each of the seed tiles has 1/4 of the prototile's size.

2. In an iteration step: Replace all tiles according to the given substitution rules (T_1 to T_4). The rotation of each replaced tile is kept.

3. Perform another iteration if the desired pattern size has not yet been reached.

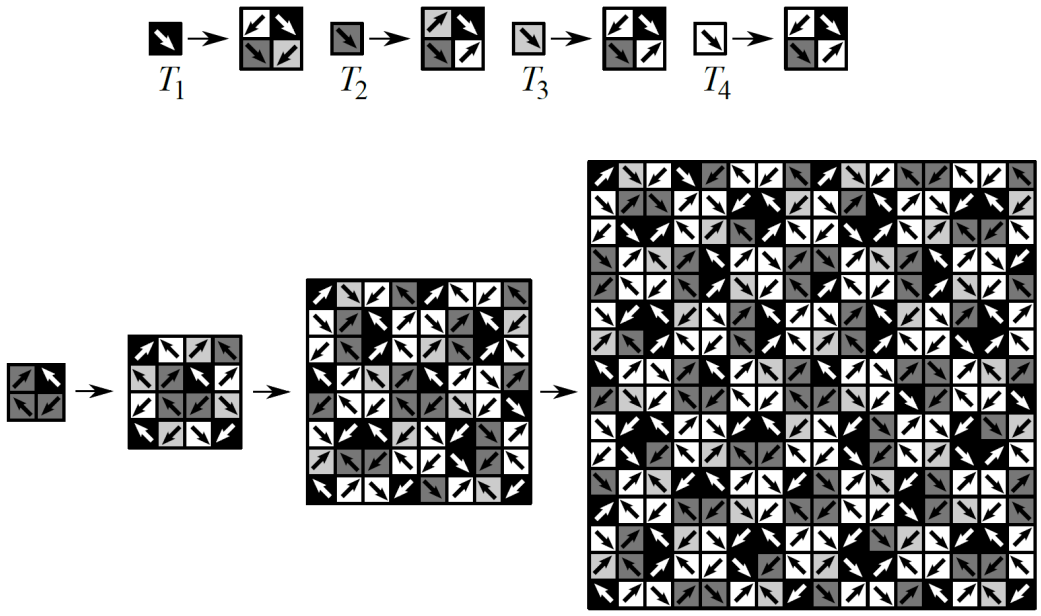

Figure 6.2: 3-way IR-Patterns can be created by applying substitution rules. Starting with four initial seed tiles, the rules T_1, T_2, T_3 and T_4 are applied to replace each tile in every step while keeping the rotation of the original tile, by rotating the tile given by the rule accordingly. After three steps this yields the tiling to the right. [FH15].

To execute an iteration step, each element of the linked list data structure is visited and the following operations are performed:

1. Inspect the tile type (T_1, T_2, T_3 or T_4) and select a rule from the rule set.

2. Create four new tiles, according to the selected rule.

3. Inspect the rotation of the current tile that is being replaced.

4. If the rotation of the tile in the rule does not match the rotation of the tile to be replaced but it has the same color it has to be aligned first. To align it, the left and right part of the specific rule are rotated until the left side of the rule matches the rotation of the tile to be replaced.

5. Replace the current tile in the linked list data structure with the sub-tiles.

6.4 Implementation

The Inductive Rotation Framework implements four different algorithms for creating Inductive Rotation Patterns. The user can switch between the different pattern generation algorithms interactively. The first algorithm is an implementation of Parzer's sprite based pattern creation algorithm [Par13]. The algorithm has been reimplemented in C#. The other three, newly implemented algorithms are described in the next sections.

6.4.1 Parallel Approach

Two different versions of the parallel approach were implemented as Direct Compute algorithms. Both algorithms map the computation of each tile's coordinates to single threads which are executed in parallel on the GPU by Direct Compute. The threads of a Direct Compute application are stored in three-dimensional grids. Each of this grids is called a thread group. Thread groups themselves are also stored in three-dimensional grids. These relations are depicted in Figure 6.3.

To map a tile id to a thread, our implementation uses Expression 6.7 and 6.8. Expression 6.7 computes the total number of threads the implementation and Expression 6.8 maps a tile index to the each different thread.

$$i_{max} = t_{max} * x_{max} * y_{max} \tag{6.7}$$

$$i_{tile} = t_{index} + tg_x * t_{max} + x_{max} * tg_y * t_{max} + r_{spilt} * i_{max} \tag{6.8}$$

In the expressions above the variables have the following meaning:

- t_{max} is the maximal number of threads running in a thread group. The current implementation uses a constant value 1024 for t_{max} (which is the maximum of possible threads).
- x_{max} and y_{max} represent the maximal number of thread groups in x and y dimensions.
- i_{max} is the number of threads in all groups.
- i_{tile} is the index of the tile to which the thread id gets mapped to.
- t_{index} is the integer index of the thread in a thread group. This value is obtained by the shader semantic SV_GroupIndex. In the current implementation the variable has a value ranging from 0 to 1023.
- tg_x and tg_y are the coordinates of the threads group in the thread-group grid. These values are obtained by the shader semantic SV_GroupThreadID, an integer vector with three dimensions. In the current implementation however, the z coordinate is not used (and therefore always set to one).

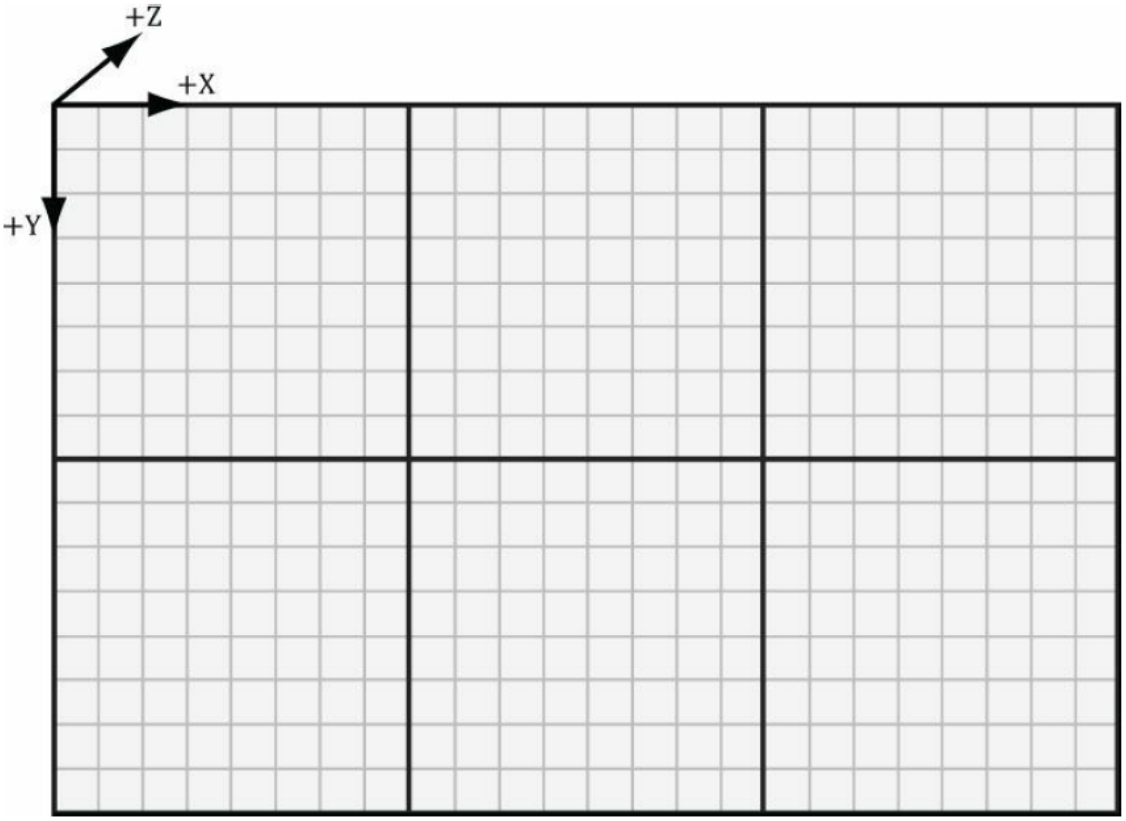

Figure 6.3: This figure shows the alignment of thread groups and their respective threads. Direct Compute dispatches thread groups in a three-dimensional grid with x, y and z dimensions. The threads of each thread-group are also aligned in a three-dimensional grid. The figure shows a grid of 6 thread-groups (2 in x and 3 in y direction) with 64 threads (8 in x and 8 in y direction) in each thread group. The number of threads in a group should be a multiple of the wavefront- or warp-size. [Lun12]

- r_{spilt} is an integer value which is read from the constant buffer. This value can be used to split pattern generation into multiple passes. Only the parallel LOD algorithm described later makes use of this variable.

These parameters allow the programmer to change the number of thread groups dynamically and to split pattern generation into multiple passes. Figure 6.4 shows how tiles indexes are mapped to thread ids by Expression 6.8 for an example with the parameters $x_{max} = 3$, $y_{max} = 3$, $t_{max} = 3$ and $r_{spilt} = 0$.

As already mentioned above, the parallel algorithm was implemented in two different variations. The first implementation uses the parameters $t_{max} = 1024$, $x_{max} = 43$, $y_{max} = 43$ and $r_{split} = 0$ for the grid of thread groups. This generates a total of 1983376 tiles which is enough for either 10 iterations for a pattern with the 3-way Method, 8

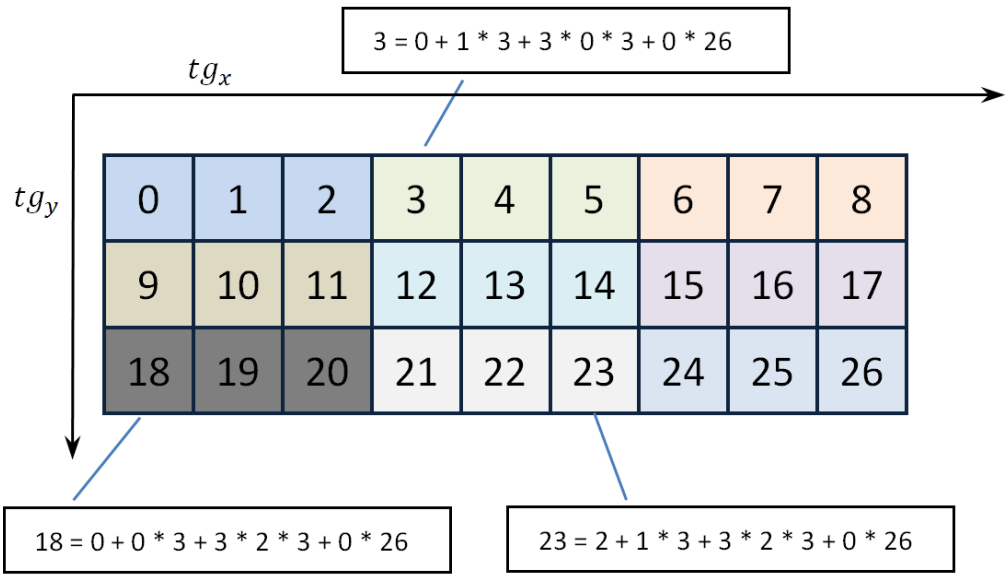

Figure 6.4: To create a mapping between the current thread and a tile index Expression 4.2 is applied. Thread groups are depicted as boxes in different colors. There are $x_{max} = 3$ thread groups in the x-dimension and $y_{max} = 3$ thread groups in the y-dimension in this example. Each of the thread groups contains three threads in the example ($t_{max} = 3$). The parameter r_{split} is set to zero.

iterations for a pattern with the 3-way Method or 13 iterations for a pattern the 2-way Method.
The second implementation uses a more sophisticated approach. Since the size of the vertex buffer is limited, the previous approaches can only produce patterns with a number of tiles that fit into the vertex buffer. To render patterns that are too large to fit into the vertex buffer, the rendering process is split into different passes and the intermediate results are stored in a texture. After each rendering pass, the results of the pass are merged with the texture. This way rendering large Inductive Rotation patterns can be split into multiple passes. This concept is also depicted in Figure 6.5. A large texture (the implementation uses 8192 * 8192 texels) is used to store the pattern. Only a part of this texture (about 1/4 at the regular zoom level) is shown to the user. The user can navigate in the texture and zoom in or out. Figure 6.6 illustrates this. In the cases where the user navigates out of the boundaries of the created texture, the pattern is recreated and rendered into the texture from a different point of view, with the pattern texture centered at the position of the user's view port. Handling texture zoom works similarly. After a, user definable, maximal number of zoom in or zoom out operations the Inductive Rotation Pattern is also recreated and rendered at the new position of the user's view port to avoid blurry looking tiles. With this approach we were able to generate patterns

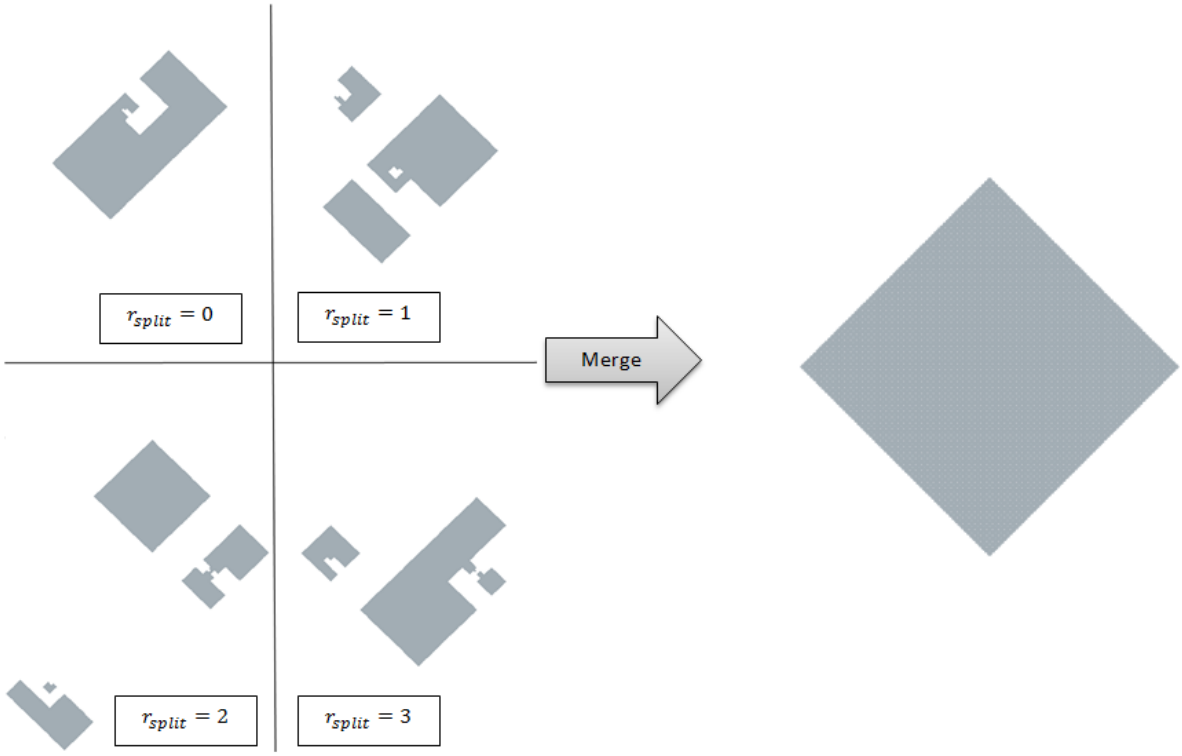

Figure 6.5: Pattern rendering is split into multiple passes. This figure shows the first four passes for a 3-way Inductive Rotation Pattern. The number r_{spilt} determines which tiles are rendered by applying Expression 4.2. These partial patterns were rendered with 1893376 tiles per pass. To render a composite pattern, the algorithm first computes the necessary number of tiles for the pattern and renders each partial pattern to the same texture in a back to front order. This means the pattern with the highest number r_{split} is rendered first.

Figure 6.6: The figure shows the concept of the parallel LOD algorithm. A part of the pattern is stored in a large (e.g., 8192 * 8192 texels) texture marked with the red boundary. The user sees only a part of the pattern, marked as "Camera Viewport". If the user's view port moves out of the boundaries (into the orange area) or the maximum zoom level is exceeded, the texture is recreated with updated view port locations. The view port's size is about 1/4 of the size large texture without zooming.

with a sizes that are about 12 times larger than patterns generated with the previous methods. Larger patterns need more time to be recreated, which induces serious lags at the events described above (user leaves texture/zoom level exceeded).

6.4.2 Substitution Tiling Approach

The substitution tiling approach is implemented by first creating mappings between the sub-tiles $T_1 ... T_4$ described in section 6.3 and the prototile texture. The prototile texture is split into four equally sized sub squares to this end. Each sub square is then assigned to a sub-tile T_i. Then the substitution tiling approach described in 6.3 is applied to create

the pattern. After executing an arbitrary number of replacement steps, the resulting tiling that contains the Inductive Rotation Pattern is rendered. Figure 6.7 compares an Inductive Rotation Pattern to a tiling that was generated with the substitution tiling approach. The pattern on the left side was generated with the parallel algorithm while the tiling on the right side was generated with the substitution tiling approach. The area marked in red contain the Inductive Rotation Pattern.

6.5 User Interfaces

The Inductive Rotation Framework can be accessed by two different user interfaces. The first user interface is keyboard-controlled and was mainly used in the development phase. It is mentioned here, however, because it can produce patterns that do not strictly follow Hofstetter's Inductive Rotation Method. The second user interface, which is depicted in Figure 6.8, contains widgets to control the generation of Inductive Rotation Patterns and several display settings. Drop down menus allow the user to switch between different render modes, access settings, paste textures from other programs and to save the image displayed in the view port. To interactively design tiles, the user can either use custom image editing software or the integrated prototile editor. When custom image editing software is used, the pattern will update when the prototile texture is saved to disk. To use the integrated prototile editor the user clicks a button with a pen symbol at the center of the top toolbar in Figure 6.8.

When the user clicks the pen symbol, the prototile editor window which shows the prototile's texture opens. Figure 6.9 shows the WPF GUI with the open prototile editor window. The user can perform different image editing operations in the prototile editor which are then directly reflected in the Inductive Rotation Pattern. This allows the user to create Inductive Rotation Patterns intuitively. Figure 6.9 shows a pattern that has been created with the prototile editor by drawing and rotating a square in a few simple steps. The prototile editor offers multiple basic commands like drawing, a fill function and texture translation and rotation.

6.6 Method Comparison

The performance for the algorithms was compared using the following hardware:

- Intel Core i5-2500 with 3.30 GHz and 4 cores
- 8 GB RAM
- AMD Radeon HD 6800 with 1024 MB (GDDR5) RAM

CPU Times for C++ code were measured using the QTime class while the Stopwatch class was applied for analyzing CPU Times in the .NET Framework. GPU Times were measured with Direct X "time-stamp queries". Memory usage of the host machine was measured using the Sysinternals Process Explorer tool [Sys15]. GPU memory usage was

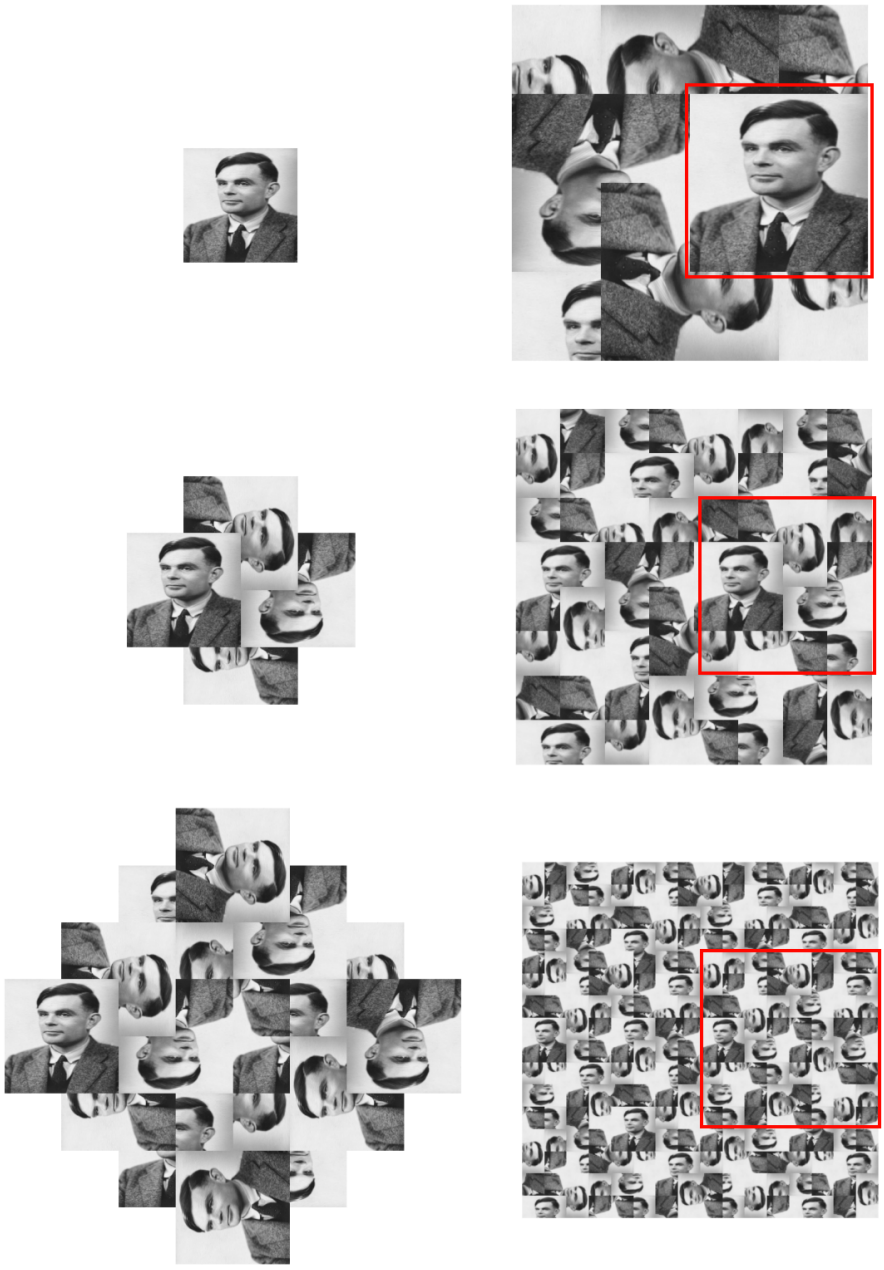

Figure 6.7: Comparison of the resulting pattern of the substitution tiling approach shown in the right column to a pattern created with the parallel method in the left column. The substitution method creates a tiling of the plane, containing the Inductive Rotation Pattern. [Bri]

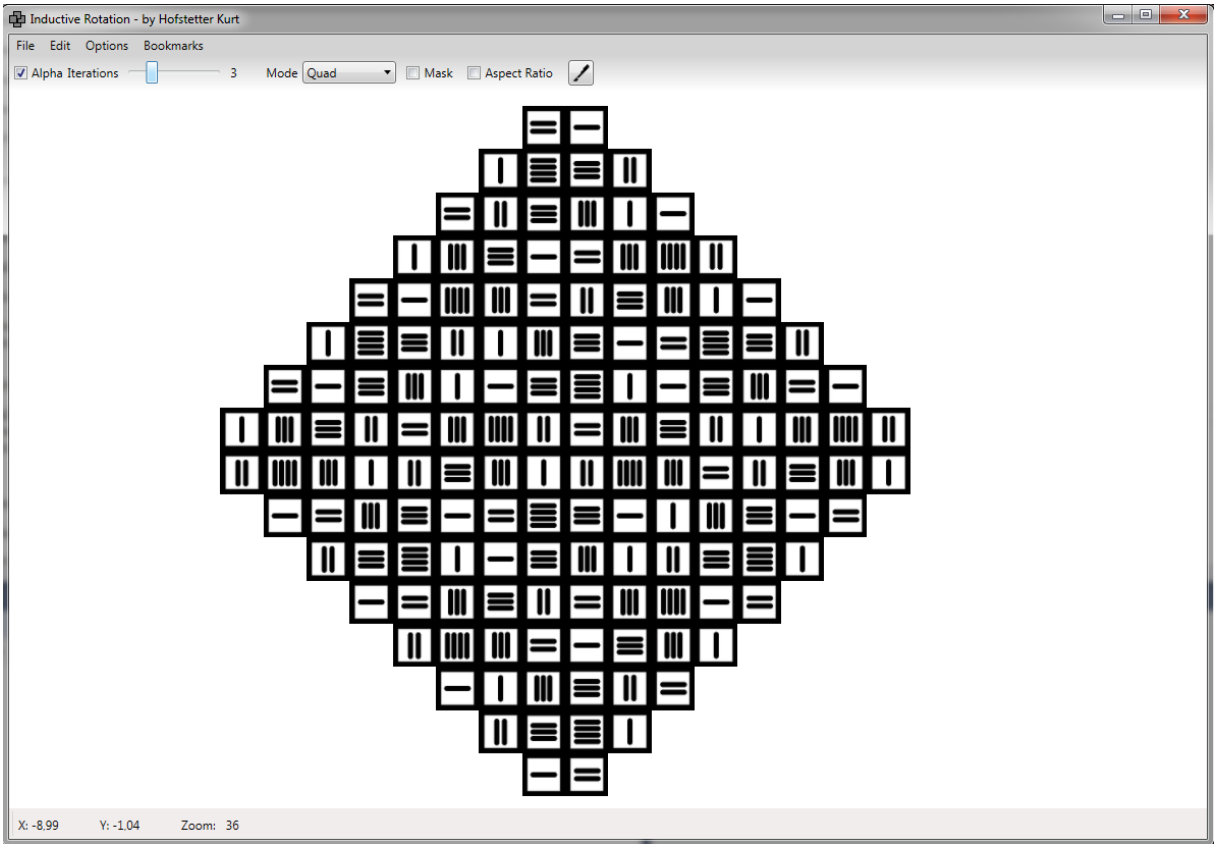

Figure 6.8: A screen shot showing the WPF GUI. Access to frequently used functions is provided by the toolbar at the top. The button with the pencil starts the prototile editor. Other options can be accessed by drop down menus. The user interface also supports loading textures from the clipboard or via drag and drop.

measured using GPU-Z [Tec].
The sprite based algorithm from Parzer, which was developed in C++, excels both the sprite based and the substitution approach, which are developed in C# in terms of generation time and memory usage. The algorithm's single iteration steps, however, cannot be directly compared to the pattern generation method of the parallel algorithm, which always generates a fixed number of (1983376) tiles. The time that the sprite based algorithm in C++ needs for generating this number of tiles was compared to the times that the parallel algorithm needs for this task in Table 6.6. The parallel LOD algorithm, which has to regenerate the pattern at the different events described above, provides the user lag free navigation in 2-way patterns for up to sixteen iterations. In 3-way Patterns the navigation works lag free for up to twelve iterations and in 5-way patterns for up to 9 iterations. Viewing larger patterns works, but introduces subjectively too much lag at the events described in Section 6.4, when the pattern is recreated.

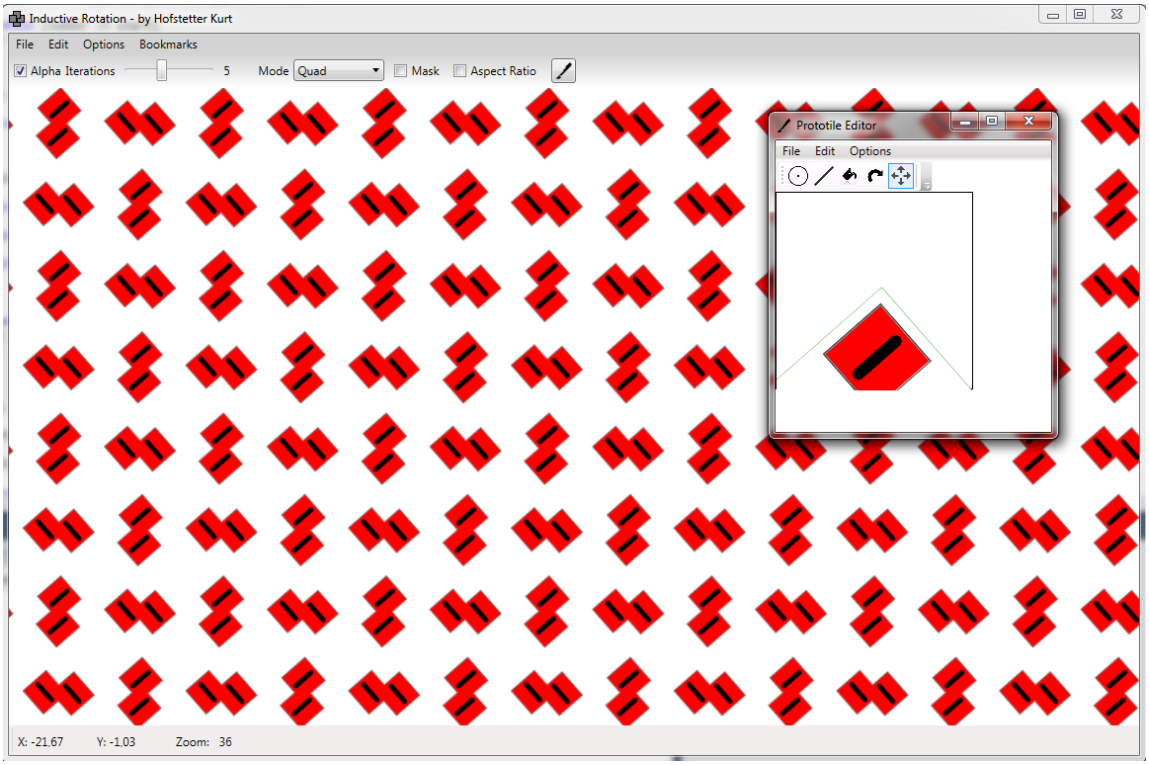

Figure 6.9: A screen shot of the prototile editor window. While drawing onto the tile texture or after translating or rotating the tile, the generated pattern in the main window reflects the changes to the texture.

Inductive Rotation Method	Max Iterations	parallel	sprite based (C++)
2-way	13	48 ms	335 ms
3-way	10	58 ms	225 ms
5-way	8	71,8 ms	303 ms

Table 6.2: The pattern generation times for the parallel algorithm compared to the pattern generation times of the sprite based algorithm. The parallel algorithm always generates a fixed amount of tiles. The time that the algorithm needs to create the pattern, however, is not noticeable by the user.

Video-RAM (VRAM) usage of both parallel algorithms is about 400 MB at all times. Both algorithms also use about 40 MB of system RAM at all times. The RAM usage of all other algorithms grows exponentially.

6.7 Discussion and Future Work

Two new algorithms for the generation of Inductive Rotation Patterns were presented in the previous sections. While the substitution approach does not have any practical advantages for generation of Inductive Rotation Patterns, it is the only approach that generates tilings in a mathematical sense. This, however, presents a disadvantage since the Inductive Rotation Pattern is contained in this tiling and there is no simple way to isolate the Inductive Rotation Pattern from the tiling.
The indexing scheme that the parallel approach uses provides a promising mathematical basis for a variety of possible parallel generation algorithms. Two implementations called the parallel and the parallel LOD algorithm were presented in the previous sections. The parallel approach generates a large number of tiles in a short amount of time as shown in the previous section. The parallel LOD algorithm enables the user to visualize about 12 times larger patterns than with the previous approaches.
The maximum pattern size for Inductive Rotation Patterns, however, is still limited. The pattern size and generation speed could probably be increased by the following improvements in future implementations:

- Currently, the LOD implementation is very simple. If the LOD level switches, the whole pattern has to be recomputed. This can be improved by exploiting the nature of the pattern: If the LOD level switches, the algorithm could determine the position of the view port in the pattern and compute only the visible parts. To this end the view port could be recursively tested against a tree structure, containing oriented bounding boxes (OBB), which enclose the whole pattern.

- Instead of computing the rotation matrices/angles in every thread, as the current algorithm does, the matrices can be computed in advance and stored in a cache for re-use. This will increase the pattern-generation speed especially for higher tile indices.

- The current implementation maps the computation of a tile's coordinates to a single thread. This is not optimal, since the threads that compute coordinates of tiles with higher indices need to execute more operations than the threads that compute tile coordinates with lower indices. Using a smarter distribution strategy would also improve the speed of the algorithm.

- The number of threads and thread-groups should be optimized for different hardware.

Bibliography

[All] Jonathan Allen. A WPF Q & A. http://www.infoq.com/news/2014/04/WPF-QA. Accessed: 2015-01-30.

[Bar88] Michael Barnsley. *Fractals Everywhere*. Academic Press Professional, Inc., San Diego, CA, USA, 1988.

[Bey] Wolfgang Beyer. File:mandel zoom 00 mandelbrot set.jpg. http://commons.wikimedia.org/wiki/File:Mandel_zoom_00_mandelbrot_set.jpg?uselang=de. Accessed: 2015-03-30.

[Bri] Matt Brice. Alan Turing: Imitating genius. http://www.valuewalk.com/2015/02/alan-turing-imitating-genius/. Accessed: 2015-03-30.

[Bro] Cameron Browne. Pythagoras Tree. http://www.cameronius.com/graphics/box-trees-figures/box-trees-fig-2.jpg. Accessed: 2015-01-30.

[Che10] Paul Chew. Voronoi Diagram / Delaunay Triangulation. https://www.cs.cornell.edu/home/chew/Delaunay.html, 2010. Accessed: 2015-01-30.

[Chi] Chiquita. Galleries. http://minionslovebananas.com/galleries-images-wallpapers/. Accessed: 2015-01-30.

[Cla] Bryan Clair. Hyperbolic Geometry. http://euler.slu.edu/escher/index.php/Hyperbolic_Geometry. Accessed: 2015-01-30.

[Coma] Wikimedia Commons. File:tiling regular 3-6 triangular.svg. http://commons.wikimedia.org/wiki/File:Tiling_Regular_3-6_Triangular.svg. Accessed: 2015-01-30.

[Comb] Wikimedia Commons. File:tiling semiregular 3-3-3-3-6 snub hexagonal.svg. http://en.wikipedia.org/wiki/File:Tiling_Semiregular_3-3-3-3-6_Snub_Hexagonal.svg. Accessed: 2015-01-30.

[CSHD03] Michael F. Cohen, Jonathan Shade, Stefan Hiller, and Oliver Deussen. Wang tiles for image and texture generation. *ACM Trans. Graph.*, 22(3):287–294, July 2003.

[Dry15] Scot Drysdale. Voronoi diagrams: Applications from archaology to zoology, February 2015. Accessed: 2015-01-30.

[Dun07] Douglas Dunham. An algorithm to generate repeating hyperbolic patterns. *ISAMA, Texas A and M*, May 2007.

[DWL+12] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. From Cuda to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Comput.*, 38(8):391–407, August 2012.

[Esc] Maurits Cornelis Escher. Circle Limit IV. http://www.mcescher.com/gallery/mathematical/circle-limit-iv/. Accessed: 2015-01-30.

[Fal03] Kenneth Falconer. *Fractal Geometry: Mathematical Foundations and Applications*. Wiley, 2 edition, November 2003.

[FH15] Dirk Frettlöh and Kurt Hofstetter. Inductive rotation tilings. *Proceedings of the Steklov Institute of Mathematics*, 288(1):247–258, 2015.

[For86] Steven Fortune. A Sweepline Algorithm for Voronoi Diagrams. In *Proceedings of the Second Annual Symposium on Computational Geometry*, SCG '86, pages 313–322, New York, NY, USA, 1986. ACM.

[Fou] Fractal Foundation. Fractal applications. http://fractalfoundation.org/OFC/OFC-12-1.html. Accessed: 2015-01-30.

[Fun10] James Fung. Direct Compute Lecture Series 210: GPU Optimizations and Performance. http://channel9.msdn.com/Blogs/gclassy/DirectCompute-Lecture-Series-210-GPU-Optimizations-and-Performance, 2010. Accessed: 2015-01-30.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[GKS90] Leonidas J. Guibas, Donald E. Knuth, and Micha Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. In *Proceedings of the Seventeenth International Colloquium on Automata, Languages and Programming*, pages 414–431, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[Gla04] Steven Glanville. Texture bombing. In Randima Fernando, editor, *GPU Gems*, pages 323–338. Addison-Wesley, 2004.

[Gos] Joe Gossman. Advantages and disadvantages of m-v-vm. http://blogs.msdn.com/b/johngossman/archive/2006/03/04/543695.aspx. Accessed: 2015-01-30.

[Gro] The Khronos Group. The open standard for parallel programming of heterogeneous systems. https://www.khronos.org/opencl/. Accessed: 2015-01-30.

[GS86] Branko Grünbaum and Geoffrey Colin Shephard. *Tilings and Patterns*. W. H. Freeman & Co., New York, NY, USA, 1986.

[Hil] Raymond Hill. Javascript implementation of Steven J. Fortune's algorithm to compute voronoi diagrams. http://www.raymondhill.net/voronoi/rhill-voronoi.html. Accessed: 2015-05-08.

[HKL+99] Kenneth E. Hoff, III, John Keyser, Ming Lin, Dinesh Manocha, and Tim Culver. Fast Computation of Generalized Voronoi Diagrams Using Graphics Hardware. In *ACM SIGGRAPH 1999 Papers*, SIGGRAPH '99, pages 277–286, New York, NY, USA, 1999. ACM.

[Hofa] Kurt Hofstetter. The inductive rotation by hofstetter kurt. http://hofstetterkurt.net/NIP/inductive-rotation-description.pdf. Accessed: 2015-03-30.

[Hofb] Kurt Hofstetter. NEW IRRATIONAL PATTERNS. http://hofstetterkurt.net/ip. Accessed: 2015-03-30.

[Hut81] John Hutchinson. Fractals and Self-Similarity. *Indiana University Mathematics Journal*, 30(5):713–747, 1981.

[Joy] David E. Joyce. Hyperbolic tesselations. http://aleph0.clarku.edu/~djoyce/poincare/PoincareApplet.html. Accessed: 2015-01-30.

[KCODL06] Johannes Kopf, Daniel Cohen-Or, Oliver Deussen, and Dani Lischinski. Recursive Wang Tiles for Real-time Blue Noise. In *ACM SIGGRAPH 2006 Papers*, SIGGRAPH '06, pages 509–518, New York, NY, USA, 2006. ACM.

[Kle04] Rolf Klein. *Algorithmische Geometrie*. Springer, 2004.

[Kur] Hofstetter Kurt. Introduction. http://hofstetterkurt.net/InductiveRotation/Introduction.pdf. Accessed: 2015-06-30.

[Lag09] Ares Lagae. *Wang Tiles in Computer Graphics*. Synthesis Lectures on Computer Graphics and Animation. Morgan & Claypool Publishers, San Rafael, CA, USA, March 2009. Editor: Brian A. Barsky, Series ISSN: 1933-8996 (print) 1933-9003 (electronic), Volume: 4, Number: 1.

[LRP95] John Lamping, Ramana Rao, and Peter Pirolli. A focus+context technique based on hyperbolic geometry for visualizing large hierarchies. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '95, pages 401–408, New York, NY, USA, 1995. ACM Press/Addison-Wesley Publishing Co.

[Lun12] Frank Luna. *Introduction to 3D Game Programming with Direct X 11*. Mercury Learning & Information, USA, 2012.

[Man82] Benoit B. Mandelbrot. *The fractal geometry of nature*. W.H. Freeman, 1 edition, August 1982.

[Mar09] Maurice Margenstern. Navigation in tilings of the hyperbolic plane and possible applications. *CoRR*, 2009.

[MCK13] Matthias Müller, Nuttapong Chentanez, and Tae-Yong Kim. Real time dynamic fracture with volumetric approximate convex decompositions. *ACM Trans. Graph.*, 32(4):115:1–115:10, July 2013.

[Neta] Microsoft Developer Network. Introduction to WPF. https://msdn.microsoft.com/en-US/en-en/library/aa970268%28v=vs.110%29.aspx. Accessed: 2015-01-30.

[Netb] Microsoft Developer Network. The MVVM pattern. https://msdn.microsoft.com/en-us/library/hh848246.aspx. Accessed: 2015-01-30.

[Netc] Microsoft Developer Network. What is a Stencil Buffer. https://msdn.microsoft.com/en-us/library/bb976074.aspx?f=255&MSPPError=-2147217396. Accessed: 2015-01-30.

[Netd] Microsoft Developer Network. What is databinding? https://msdn.microsoft.com/en-us/library/ms752347.aspx#what_is_data_binding. Accessed: 2015-01-30.

[Pal] Luis Blackaller Kyle Buza Justin Mazzola Paluska. Aperiodic Tilings. http://black.mitplw.com/tiles/aperiodic.html. Accessed: 2015-01-30.

[Par13] Simon Parzer. Irrational image generator. Master's thesis, Institute of Computer Graphics and Algorithms, TU Wien, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, June 2013.

[Pic09] Clifford A. Pickover. *The Math Book*. Sterling Publishing Company, Inc., 2009.

[SH75] Michael Ian Shamos and Dan Hoey. Closest-point problems. In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*, SFCS '75, pages 151–162, Washington, DC, USA, 1975. IEEE Computer Society.

[Sip] Sebastian Sippl. Inductive rotation - Animated pattern. https://www.youtube.com/watch?v=zP-H99P7lnA&feature=youtu.be. Accessed: 2015-05-16.

[Sof15] Tiobe Software. TIOBE index for may 2015. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, 2015. Accessed: 2015-05-25.

[Sta97] Jos Stam. Aperiodic texture mapping. Technical report, European Research Consortium for Informatics and Mathematics (ERCIM), 1997.

[SW] Matthew Szudzik and Eric W. Weisstein. Parallel Postulate. From mathworld–a wolfram web resource. http://mathworld.wolfram.com/HyperbolicGeometry.html. Accessed: 2015-01-30.

[Sys15] Sysinternals. Process Explorer. https://technet.microsoft.com/en-us//sysinternals/bb896653.aspx, 2015. Accessed: 2015-01-30.

[Tec] TechPowerUp. Techpowerup gpu-z. http://www.techpowerup.com/gpuz/. Accessed: 2015-05-30.

[Tra] Christoph Traxler. Classic Fractals by L-Systems. http://www.cg.tuwien.ac.at/courses/Fraktale/PDF/fractals8.pdf. Accessed: 2015-03-30.

[Tut15] Tutorialspoint. Number System Conversion. http://www.tutorialspoint.com/computer_logical_organization/number_system_conversion.htm, 2015. Accessed: 2015-06-30.

[Ven] Bill Venners. Orthogonality and the dry principle. http://www.artima.com/intv/dry.html. Accessed: 2015-01-30.

[Vor08] Georges Voronoi. Nouvelles applications des paramétres continus à la théorie des formes quadratiques. Deuxième mémoire. Recherches sur les parallélloèdres primitifs. *Journal für die reine und angewandte Mathematik*, 134:198–287, 1908.

[Wei04] Li-Yi Wei. Tile-based texture mapping on graphics hardware. In *ACM SIGGRAPH 2004 Sketches*, SIGGRAPH '04, pages 67–, New York, NY, USA, 2004. ACM.

[Wri] WriteableBitmapEx. Writeablebitmapex. http://writeablebitmapex.codeplex.com/. Accessed: 2015-01-30.