The Soft Grid

by

Ari Kardasis

MArch, Princeton University 2009
ScB, Mathematics Brown Univerity 2001

SUBMITTED TO THE DEPARTMENT OF ARCHITECTURE IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE IN ARCHITECTURE STUDIES
AT THE
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

JUNE 2011

Signature of Author: ____
Department of Architecture
May 20, 2011
Certified by: ____
Terry Knight
Professor of Design and Computation
Thesis Supervisor
Accepted by:____
Takehiko Nagakura
Associate Professor of Design and Computation
Chair of the Department Committee on Graduate Students

Reader:

Athanassios Economou
Visiting Associate Professor

The Soft Grid

by

Ari Kardasis

MArch, Princeton University 2009
ScB, in Mathematics Brown Univerity 2001

Submitted to the Department of Architecture
on May 20, 2011 in Partial Fulfillment of the
Requirements for the Degree of Master of Science in
Architecture Studies

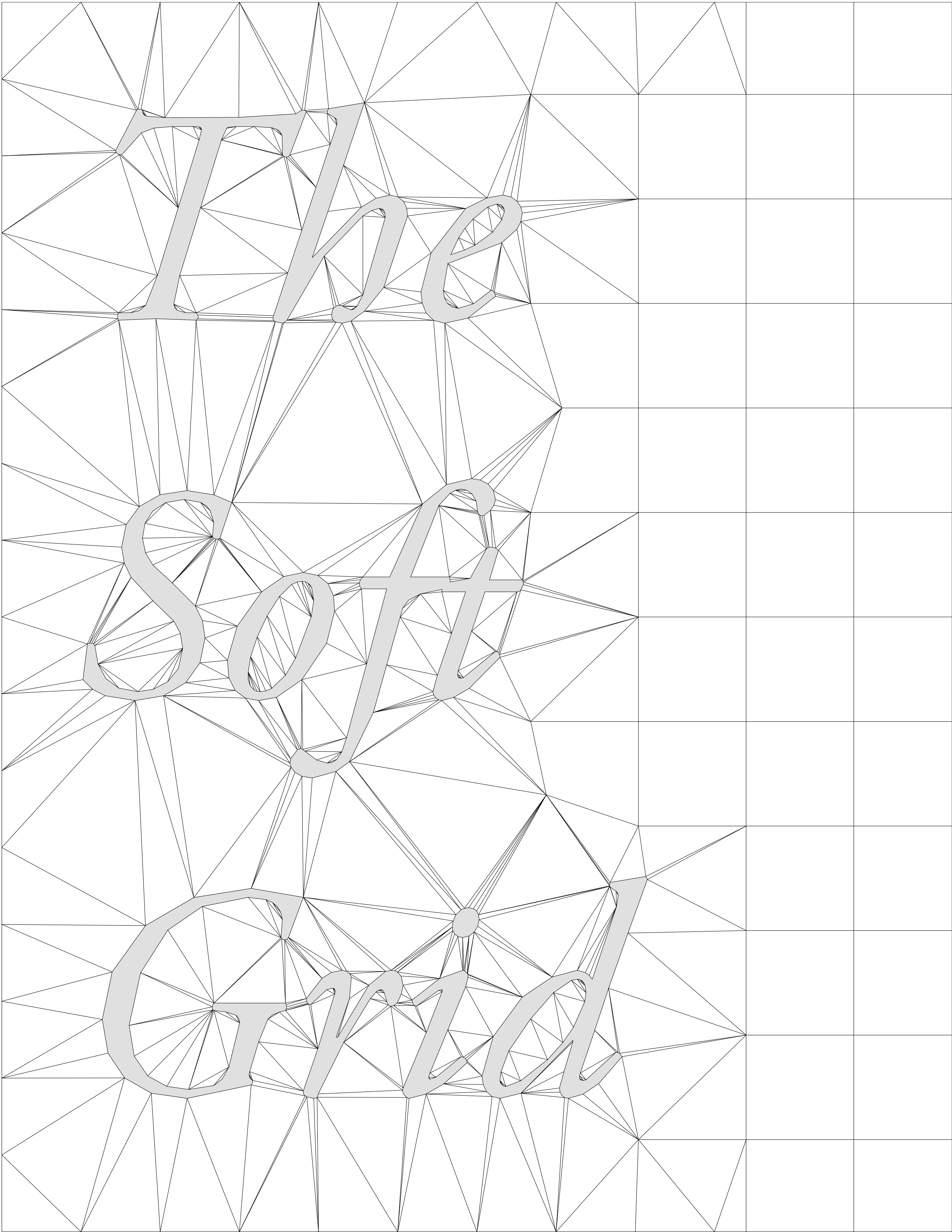
ABSTRACT

The grid in architecture is a systematic organization of space. The means that architects use to organize space are, almost by definition, rigid and totalizing. The Cartesian grid, which will serve as the antagonist of the soft grid, is geometrically and topologically unyielding on both the local and global scales. There are, however, alternatives to such hard grids. Through a series of studies, this thesis will catalog and analyze the soft girds, i.e. those that are adaptive, variable, scalable, asymmetrical and entropic.
Computational tools in architecture have, in recent years, enabled designers to manage geometries that until now have been realizable only by analog means. The instrumental capacity for complex designs has lead to increased demand for soft gridding systems as is evidenced by the profusion of Voronoi diagrams, pixelations, distorted grids and Danzer tilings in student and conceptual work. However, the built scale of such projects is rarely beyond installation largely because of the difficulty in managing spatial organizations that are not essentially Cartesian. This thesis will lay the groundwork for a systematic understanding of the possibilities of soft grids while providing much of the computational tools to generate and manage specific examples.

Thesis Supervisor: Terry Knight
Title: Professor of Design and Computation

Acknowledgements:

First, a great thanks to my adviser, Terry Knight and my reader, Thanos Economou. Additionally, I'd like to thank Erik Demaine, Neil Gershenfeld, Daniel Rosenberg, Duks Koschitz, Daniel Cardoso and Kaustuv DeBiswas for intellectual guidance as well as The Design Computation Group and the MIT community at large. This thesis would not have happened without the conversations with and general influence of my peers including Steffen Reichert, Skylar Tibbits, Xiaoji Chen, Grant Alford, William Patera, Moritz Kassner, Carl Lostritto and many more. Finally, I want to thank my parents Nick and Arline Kardasis without whom none of this would be possible.



The Soft Grid

1| Introduction

The grid

A grid is a systematic organization of space. Whether planometric, graphic, crystallographic or structural, a grid is a geometrical abstraction that allows designers to give structure to space. In the context of design, a grid is an explicit tool that affords the designer a geometrical system within which to work and provides hierarchy, adjacency and continuity across a particular design. This work will focus primarily on the use of two dimensional grids in architecture as a means of structuring plans, sections and three dimensional organizations.

Grids occupy the space between geometry and topology. While populated with shapes, they maintain the graph-like relationships of adjacency, vertex-adjacency, etc. It is this duality that makes the grid such a powerful tool for architectural design, encompassing elements of organization as well as form. Traditionally, grids maintain a uniformity from one area of the organization to another. This consistent format which will be discussed more in chapter 4 is the result of the technology used to generate such grids, namely, architectural drafting equipment such as the right angle and straight edge.

This Purpose of this Thesis

The purpose of this thesis is three-fold. First, it is an exploration of what a soft grid is and a search for a definition. Second, it is a survey of existing soft girds with significant contributions particularly in the area of parallelepiped packing. Third, it is meant to act as a design guide for students, practitioners and teachers interested in exploring alternative systems for spatial organization. The sum of these efforts is a codification of a strong trend in architectural schools and practices to design using systems that are free from the the totalizing mentality of the Modernist grids that have dominated the field for the last several decades. With the advent of easily accessible scripting environments (both visual and textual), the designer's ability to create complex, intricate and customized grids has exploded in recent years. The Soft Grid captures this idea.

In searching for a description of an idea which is self contradictory and elusive, it is fair to say that results are themselves difficult to nail down. In this spirit, the following chapters, with their graphical and descriptive content, will give the reader an intuition for a notion rather than a definitive description of a complete concept. The analyses are critical and at no point should softness be taken to be a blanket assertion of quality. Like any tool, a soft grid is quite capable of being misused and, at the end of the day, the designer is responsible for his/her own decisions.

Chapters 1 through 4 are meant as an introduction to the idea of grids, both soft and hard with a brief survey of the historical uses of grids in architecture. Here, the primary terms of the thesis, namely 'grid', 'soft' and 'hard', will be defined and expanded on using examples from art and architecture as a means of contextualizing the ideas in the following chapters.

Chapters 5 through 10 are organized into six categories of soft grids: High-symmetry, Local forcing, Global systems, Voronoi diagrams, Subdivision schemes and Parallelepiped packing. In no way do these classifications cover the entire space of soft grids nor are they independent from one another. The boundaries of what qualifies as a soft grid are themselves soft and, because of this, no totalizing classification scheme can exist. However, the survey is designed to take a representative sample of the field and the material that has been added is meant to advance the principles that have been seen in the observed cases.

Finally, this thesis serves as a tool for anyone looking for a soft gird to advance a design project of their own. I will discuss issues such as how to grid oddly shaped regions which are directly relevant to the design process and will give guidance to those seeking alternatives to hard grids.

Moreover, the typological analysis of the major classes that I list will provide a basis for comparison and a way to move forward with this fundamental choice in the design process.

In combining these agendas, I hope to bring clarity and insight to an otherwise uninvestigated design tool which is in need of codification in a way that is simultaneously academic and applicable to design practice. This thesis does not purport to settle any issue with finality but instead to shed light on an emerging collection of ideas and practices that, taken as a whole, constitute a comprehensive addition to the design field at large.

2| Context

A Brief History of the Grid

The grid in architecture is a device which is pervasive throughout history but whose manifestation changes significantly through the ages. As far back as antiquity, and further, buildings have been designed and built using rectilinear systems with regular spacing. But the post facto recognition of a grid does not imply the conscious deployment of a grid in the design process.
It wasn't until the early Modern era that girds were employed *explicitly* in the service of design.
In the 17th century, Descartes posited the numerical treatment of the rectilinear geometry that had been used for millennia to describe space rather than the implicit geometric mapping. This philosophical and mathematical idea translated into rigorous design principles in the 18th century most notably at L'École des Beaux-Arts. This is not to say that the grid was geometrically absent before the 18th century but rather that its use was not itself the object of scrutiny.
In the late 18th and early 19th century, Jean-Nicolas-Louis Durand brought further rigor to the practice of gridding in architecture with a series of lectures describing organization as a practice

for design.[1] Following Durand, the 19th century saw a profusion of grids on all scales including the 1811 plan for New York's rectangular grid, Haussmann's organizational system of Paris, early skyscrapers, and the factory designs of industrialization. Grids in architecture arose because designers were dealing with larger and more complex systems which needed an organizational system to be maintained.

The early 20th century is marked architecturally and artistically with an explosion of grids. Perhaps most prominent among the grids in art are those of Piet Mondrian who explored the grid for its own sake in his series of compositions (Figure 1). In fact, these grids introduce a kind of softness through the adjustment of the overall system. This is a local forcing as we will discuss in chapter 6. These principles were translated into the design community by the De Stijl movement and the Bauhaus. The grid's lineage in modern architecture continues from there including, most prominently, the work of Mies van der Rohe and LeCorbusier.
Buckminster Fuller made a significant contribution to the conceptualization of space with his trademark geodesic structures. The building of these overtly non-rectilinear structures highlighted the potential for alternative organizations in the built form. Fuller's reconsideration of the way things are designed and made was ultimately not a material or structural argument, but an organizational one played out in his innovative and novel grids.

In the 1950's John Hejduk introduced the nine-square grid[2] as a tool specifically as an aid for design and not an element of the built form. By calling into question the grid as a tool for design and using it as a teaching technique, Hejduk brought to light the importance of the grid as a fundamental aspect of design practice.
In the 1960's and 70's, the grid made a resurgence in the conversations of architecture and art. Le Corbusier expanded the idea of the grid as an infinite overlay of space with his design for the Venice hospital (Figure 2) which proposed an unbounded rectilinear organization. Bernard Tschumi's Parc de la Villette similarly highlighted the gridded organization of space. Artists like Agnes Martin and Sol Lewitt explored the grid at its most fundamental level, using methodical grid-making as a form of visual art itself.

Hannah B Higgins, in her book *The Grid Book*[3] takes a much broader approach to the study of grids than I will here but she nonetheless manages to identify some essential qualities from art,

1 Durand

2 Love / Caragonne

3 Higgins

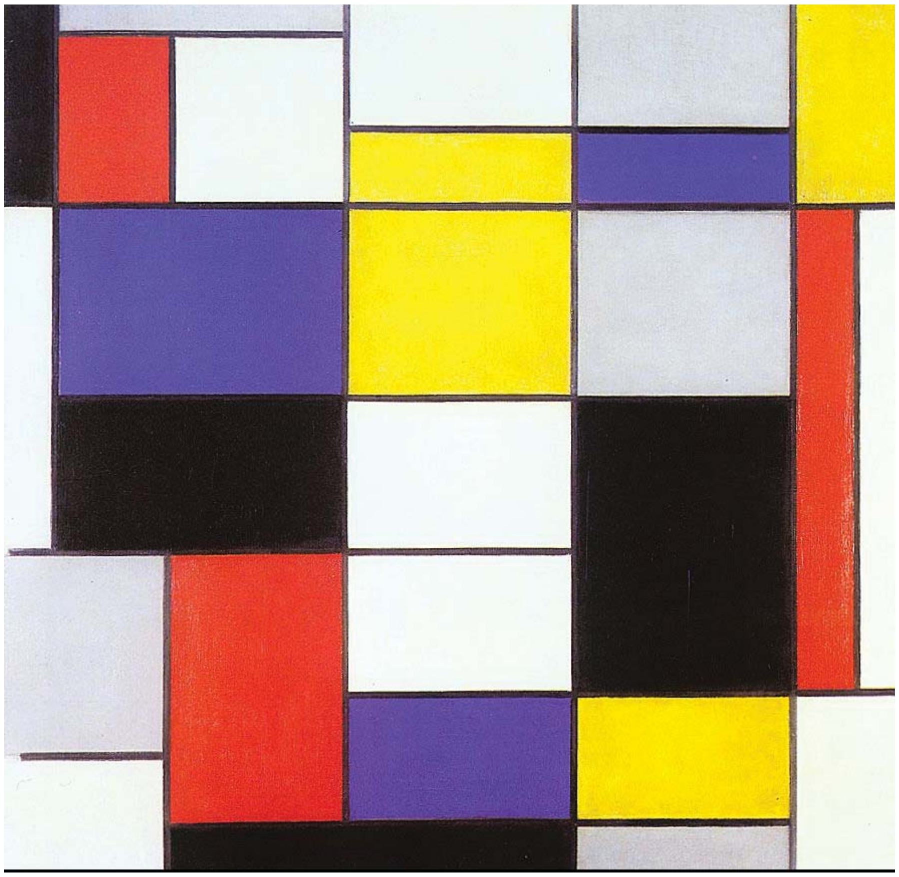


Figure 1. Piet Mondrian. Composition A. 1923
witcombe.sbc.edu/modernism-b/artsake.html

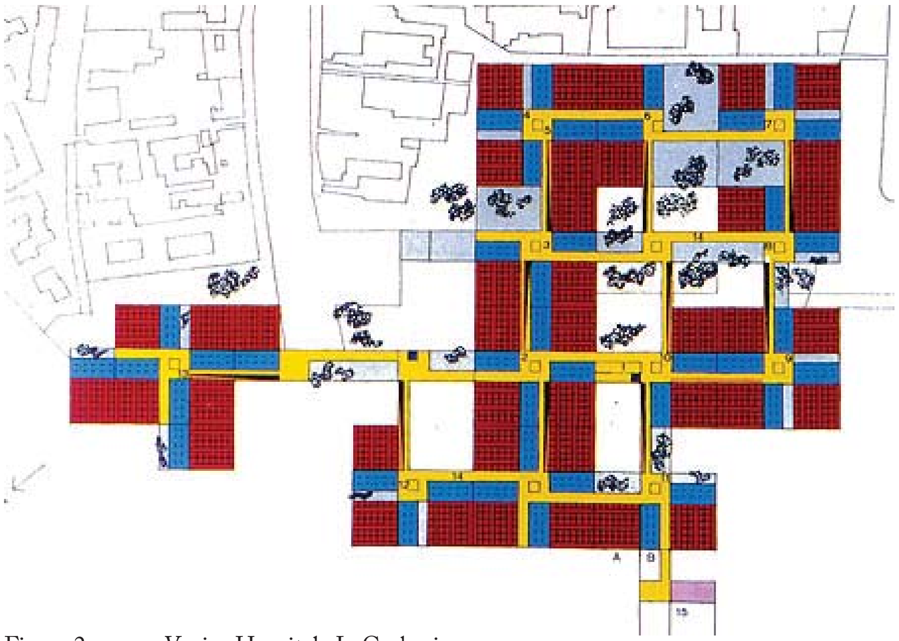


Figure 2. Venice Hopsital. LeCorbusier
www.revistaau.com.br/arquitetura-urbanismo/182/imprime134779.asp

graphic design, mathematics, technology and more. While her definitions differ from mine[1], her treatment of the network and the fractional dimension grids is especially relevant to the idea of softness.

Contemporary Condition

The impetus for the study of soft grids in this thesis comes from the profusion of organizational techniques facilitated by the introduction of computational tools into the architectural design process. In architecture schools and in practice, there is a search for grids which are variable, dynamic and otherwise soft, but a lack of the specific knowledge required to implement them. As a design guide, the following survey will introduce some of the concepts to achieve these grids while maintaining a critical eye toward the value of such systems.

Much of the desire for soft grids comes from the introduction of advanced geometry and scripting practices into the design community by small firms like Aranda/Lasch[2] and TheVeryMany[3] but also the large-scale geometrically-advanced practices like Gehry Partners and Zaha Hadid Architects. These geometrical practices are driving not only complexity but novel spatial systems for its understanding.
Among the most explored soft grids that have resulted from recent technologies is the Voronoi diagram which, although rarely built, has seen a huge amount of popularity in student and conceptual architecture projects in the first decade of the 21st century. The Voronoi diagram will be discussed more in chapter 8, but it is worth noting here that it is a strong motivating factor for this study. Additionally, the Danzer tiling is becoming increasingly prevalent in certain practices. This will come up again in chapter 9.

1 While Higgins' grids are often organizational, she also includes chapters on the brick and the box, which generate grids in their assemblage, not as design tools or a means of structuring space.

2 Ben Aranda and Crish Lasch

3 Marc Fornes, Skylar Tibbits, Mathew Staudt

3| The Soft Grid

A soft grid is a way of organizing space that is adaptive, variable, entropic, asymmetrical or scalable. The term is intentionally enigmatic as it arises at the border of the rigid and the malleable and its definition is somewhat elusive because of this contradiction. The intent is to introduce a quality (softness) into a concept which is strongly opposed to that quality. This is a precarious balance and we will see in chapter 8 how grids can be too soft.

There is no single qualitative or quantitative metric for softness. One of the primary results of this thesis is a discussion of what softness is and the reader will have a sense, albeit tacit, of what makes a grid soft as well as why and when this is a valuable quality. The five attributes of softness listed below should not be taken as a complete description, nor should they be understood as completely independent of one another. (e.g. we cannot have entropy without variability.) The sum total of these descriptors taken together should give an overall sense of what it means for a grid to be soft.

Adaptivity
A grid can be said to be adaptive if it reacts, either locally or globally, so some enforced condition. This can be a manual change made by a designer or an algorithmic process for organizing a given space with given conditions. In this way, the grid adapts to it's environment whether this is a one-time a priori adaptation or an interactive process of adaptivity, in which the designer is changing the grid as it is being used.

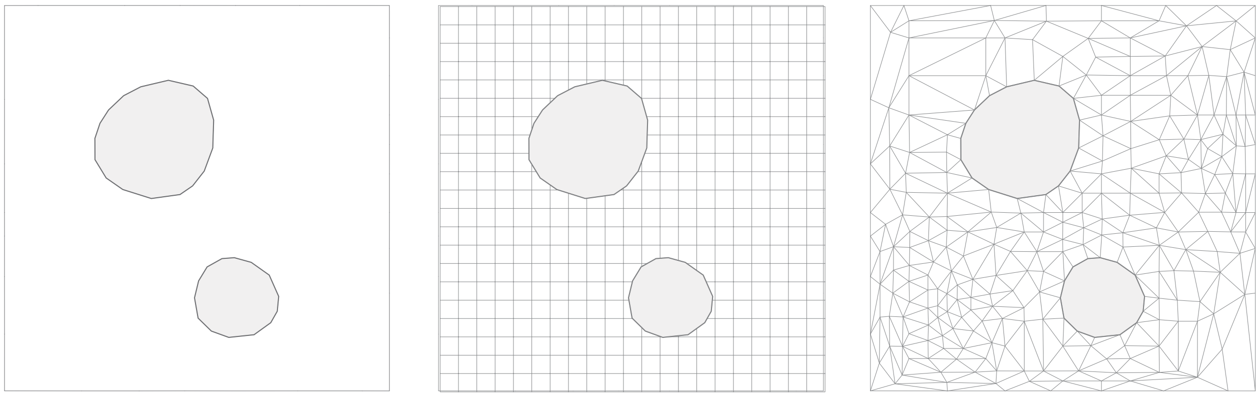


Figure 3. The enforced condition, an imposed unadaptive grid and an adaptive grid

Variability
Perhaps the most evident attribute that lends softness to a grid is variability. This can be seen by observing the hard grids and noting that all geometry and topology is constant throughout which is to say that there is no variability at all. Variability may be the most fundamental aspect of softness for without it, any grid is quite hard.

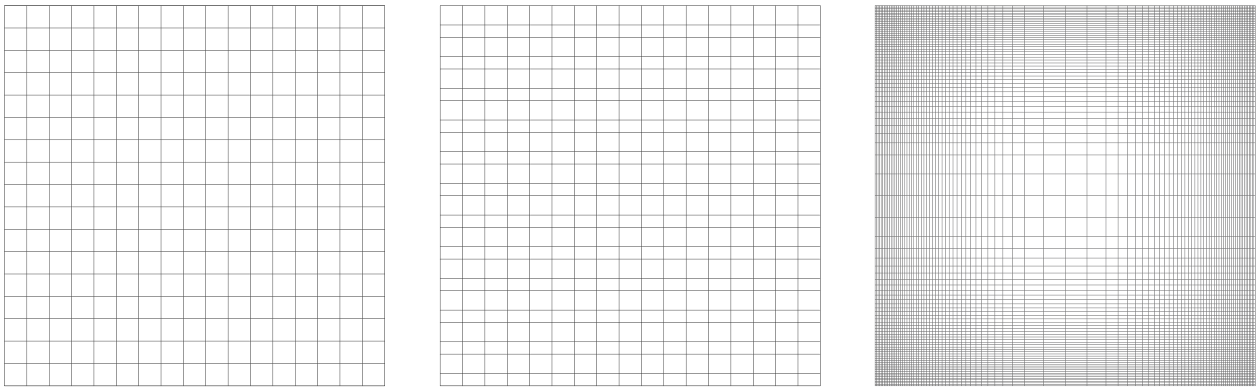


Figure 4. The grid on the left has no variability. The grid in the middle has very little and the one on the right has a lot.

Entropy

Borrowing from information theory, entropy is a measure of the surprise[1] when one gains a new piece of information. This idea has vast consequences but in the context of the soft grid, it is meant to express how 'knowable' a grid is. If one were to slowly reveal a grid and that grid were hard in the sense that it was the same everywhere and had no choices to be made, one would fail to be surprised by the successive revelations. On the other hand, if the successive exposures of the cells gave new information to the observer, she would be surprised by the content on view. In this sense, we can say that the grid is entropic in that it contains a great deal of information.

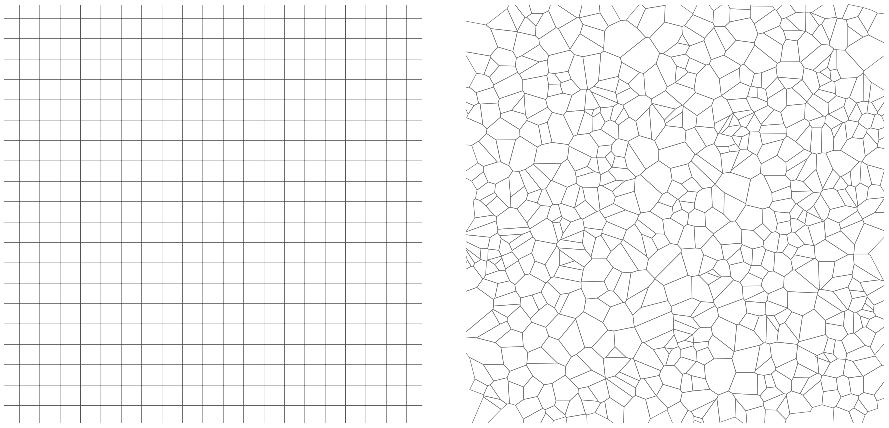


Figure 5. Zero entropy in the hard grid and high entropy in the random Voronoi diagram.

Asymmetry

Symmetry exists in many different forms but the way that it is being used here is primarily in relation to symmetries in the plane, specifically, wallpaper symmetry.[2] Simply put, symmetry is the capacity for a grid to look like itself when moved in some way. This movement can be a translation, rotation, reflection or glide[3] but it can also be a scaling if we are discussing fractal grids.

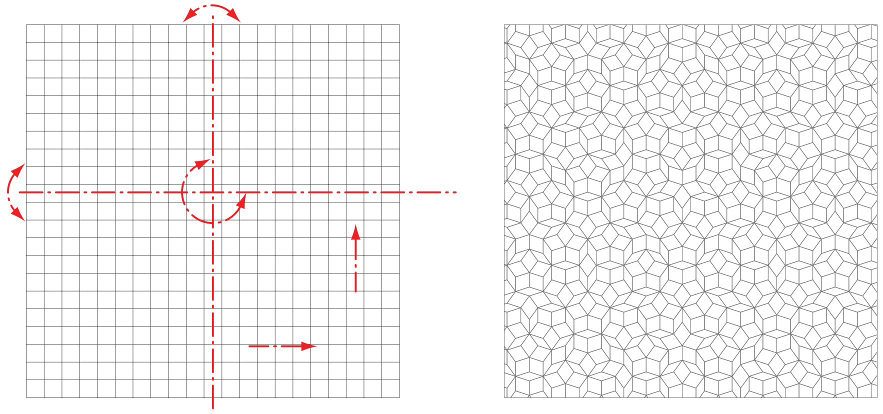


Figure 6. The square grid has translational, rotational and mirror symmetry while the Penrose tiling has none of these.

1 Shannon, etc

2 See March and Steadman and Grünbaum and Shephard for further discussions on symmetry.

3 A glide symmetry is a translation and reflection along the same axis. See March and Steadman p 60

Scalability

Scalability is the ability of a grid to operate across scales. Simply put, it is a variation in the size rather than in the shape of the elements of a grid. The idea of a fractal[1] is essentially infinite scalability where a grid can operate at every scale from the gigantic to the minute. Being scalable can mean having scaling symmetry and therefore puts this attribute in opposition to asymmetry when the symmetry in question is scalar.

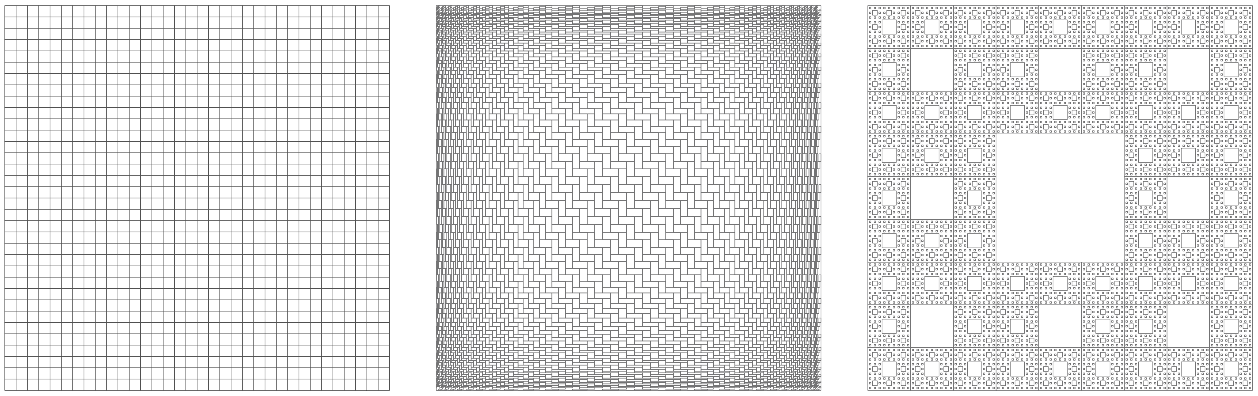


Figure 7. The square grid operates at only one scale while the deformed grid in the middle has many and the Sierpinski carpet on the right works at an infinite number of scales.

1 Mandelbrot

4| Hard Grids

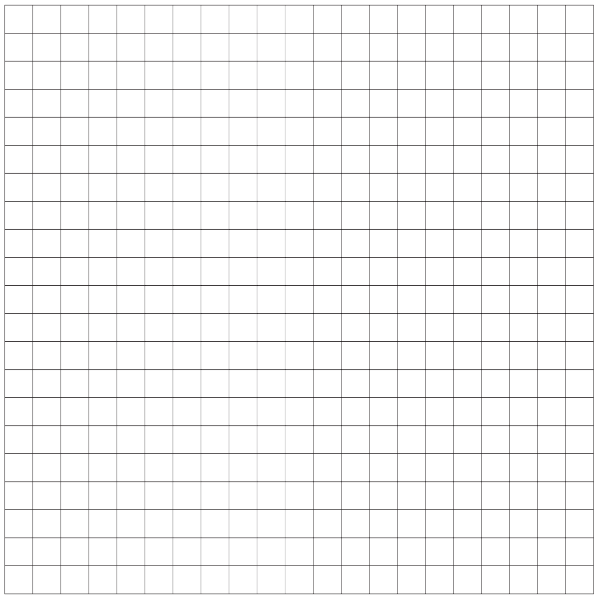


Figure 8. The Square Cartesian Grid is highly uniform.

The Antagonists

Before delving into the wide variety of soft grids, it is worthwhile to establish a set of observations about our antagonists, the hard grids. Just as soft grids are variable, adaptive, scalable, asymmetric and entropic, their counterparts are uniform, static, unscalable, symmetrical and highly ordered. From a mathematical perspective, hardness can be thought of as sameness across the grid. Sameness of cells, edges, vertices etc. is apparent for square grids as well as triangular and hexagonal. These three grids constitute absolute hardness. In the next chapter, I will discuss rectangular and skew grids as well as regular tilings, which demonstrate a great deal of rigidity and structure but are not as hard as the three in this chapter.

Square Cartesian Grid

Although there is no softest grid, the opposite is not true. I will argue that there is in fact a hardest grid and it is the Square Cartesian Grid . While detractors may argue that the square grid can lend a great deal of softness to a design process, I assert that it is through the various softening processes that it gains these properties, not from its essential nature.
The square grid is generated by evenly spaced lines in two orthogonal (or perpendicular) directions, the spacing in one direction being the same as in the other. This simple construction yields a uniform array of square cells which demonstrate a zero on the scale of adaptivity, variation, asymmetry, scalability and entropy. It is the canonical grid in architecture and has been used for millennia because of its ease of management and construction. It is strongly associated with the Cartesian coordinate system whereby points in the plane can be located by two numbers (x and y) each of which has no effect on the other. The grid lines are the places at which one of the coordinates is an integer and the intersections of those lines are where they are both integers.
Unlike the triangular and hexagonal grids, the square grid has 4-fold symmetry around its cell vertices[1]. This is deeply tied to the 90° angles at the corners and to the overall orthogonal nature of the square grid. It is this separation of space into two totally independent directions that makes the square grid that much harder than the other two discussed in this section.

Equilateral Triangular Grid

The equilateral triangular grid, much like the square, is generated by sets of evenly spaced parallel lines. The difference being that the triangular grid is composed of three sets of lines which are at 120° angles to one another, removing the 4-fold rotational symmetry and introducing 6-fold symmetry around the vertices and 3-fold symmetry around the centers of the cells (Figure 10). In a sense, the triangular grid (as well as the hexagonal) are harder than the square in that they have more rotational symmetries althought in the case of the triangular grid, an alternating orientation of the cells creates some small amount of softness.

1 March and Steadman p. 56. I will discuss symmetry in more detail in the next chapter.

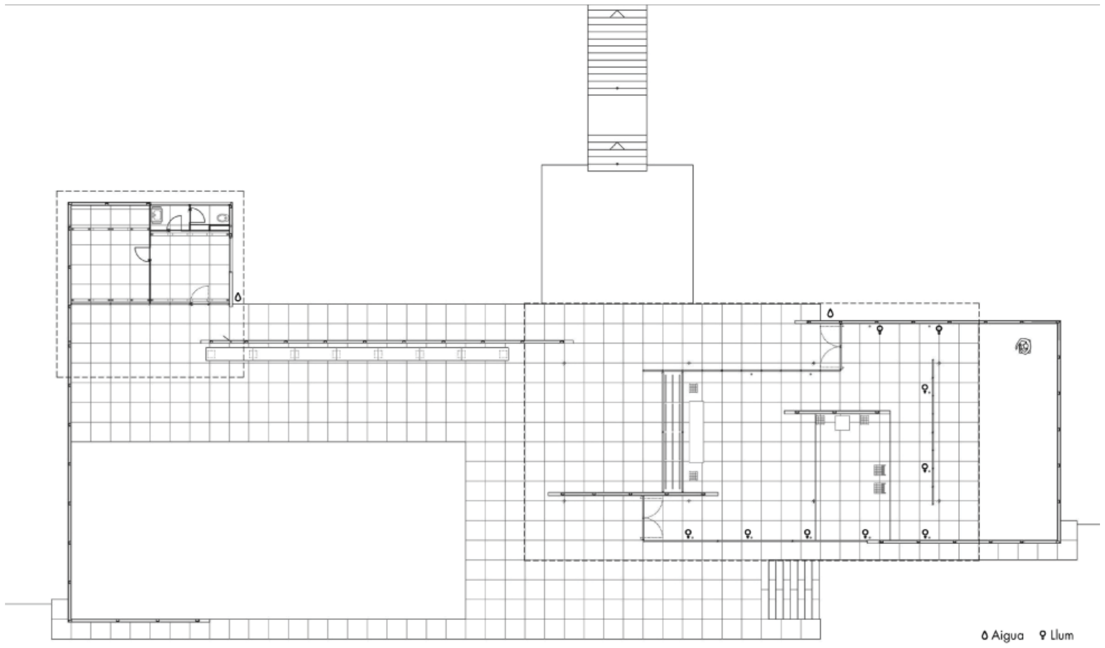


Figure 9. The Barcelona Pavillion. Ludwig Mies van der Rohe.
The tiling pattern is a clear manifestation of the square grid on which the design was made.

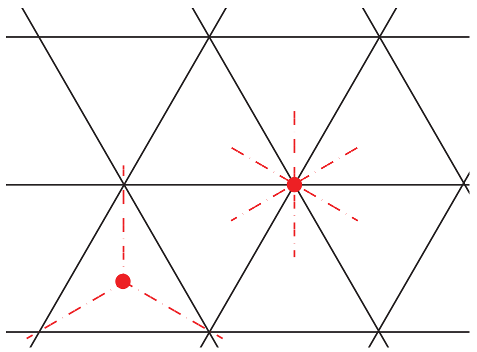


Figure 10. Symmetries in the triangular grid

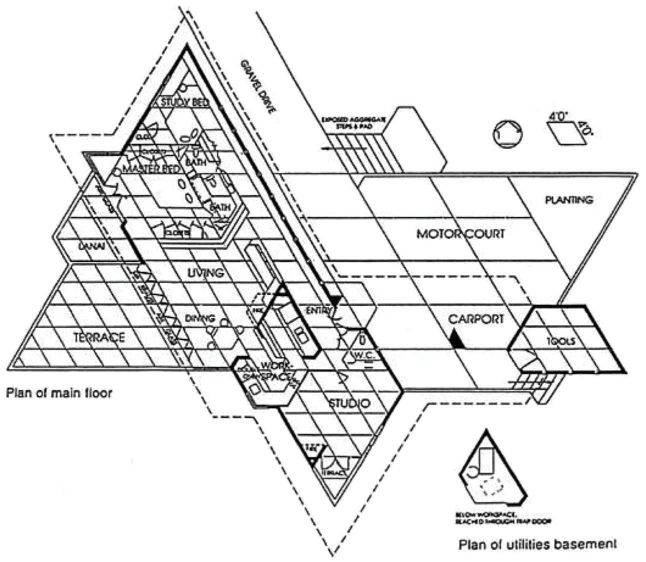


Figure 11. The Frank Lloyd Wright House in Ebsworth Park.

Hexagonal Grid

The hexagonal grid, often referred to as a honeycomb, is dual[1] to the triangular grid (Figure 13) and exhibits all of the same symmetries as it. Unlike the previous two hard grid types, the hexagonal grid is not constructed from continuous straight lines but instead exhibits a more cell-like organization characterized by trinodal[2] vertices and large (120°) angles. The hexagonal grid has a certain biological look because of the relationship to the Voronoi diagram[3] and is, at first glance, less hard looking than the other two hard grids in that it lacks the linearity and continuity that they have.

Hexagonal grids are rarely used in architecture and tend to be something of a novelty when they are. There is a strong aesthetic that the hexagonal grid carries with it (Figure 12) and its use tends to imply a connection to nature despite the strong hardness of the grid. This is partly due to Voronoi diagram connection but also a cultural bias relating such grids to an era of late modernism marked by a return to nature.

1 For a definition and discussion of duality, see Grünbaum and Shepard p.97.

2 Each vertex is at the intersection of three edges.

3 See chapter 8.

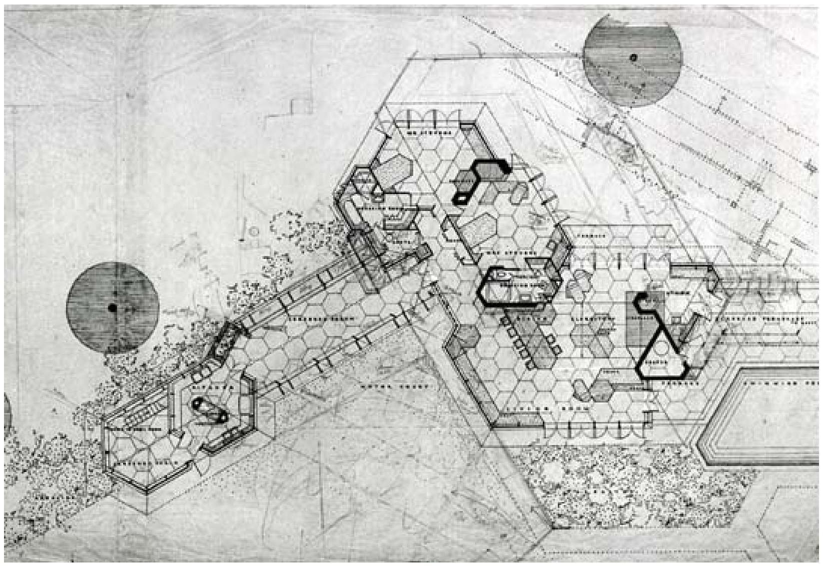


Figure 12. Auldbrass. Frank Lloyd Wright

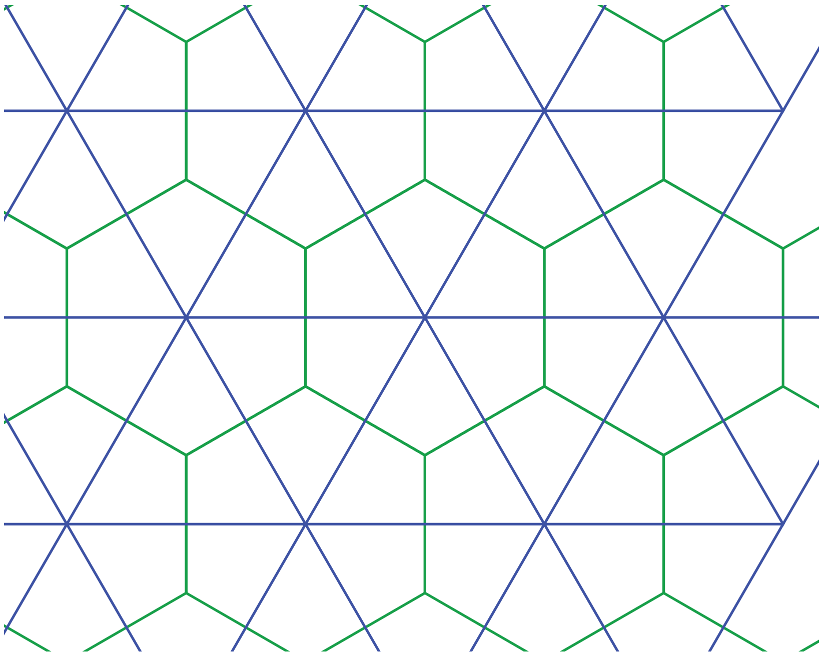


Figure 13. The duality of the triangular and hexagonal grids.

5| High-Symmetry Grids

After the hard grids described in the previous chapter, the high-symmetry grids are the most rigid and familiar grids. They are characterized by translational symmetry, which gives them very low variation, adaptivity, scalability, asymmetry and entropy, although in each of these categories they do (or can) display a small amount of softness over the hard grids. Beyond the translational symmetry, high-symmetry grids are often associated with the wallpaper symmetries[1] which include rotation, reflection and glide symmetries.

Carsten Nicolai's[2] vast and exhaustive study of every imaginable grid near the hard end of the spectrum is a beautiful collection.[3] Grünbaum and Shephard provide a highly detailed and robust mathematical exploration of tilings of all sorts with a great deal of formalism and rigor.

1 See March and Steadman p. 64

2 Nicolai

3 Nicolai also looks at quasiperiodic tilings of 5-, 7-, 8-, 9- and 12- fold symmetry, which will be discussed further in chapter 9.

The tartan grid[1] (Figure 15) is closely related to the square grid but has a syncopated spacing of the grid lines yielding an opportunity to insert walls between the narrowly spaced lines and open space in the larger space. In a sense, this is an adaptation of the square grid to accommodate the thickness of walls, which would be on the grid lines of the square grid. Trying to place walls on the infinitesimally small lines leads to inevitable difficulty and the tartan grid is the logical solution to this problem. In this way, the tartan grid is highly adaptive because it responds to a specific need.

The street grid in the city of Barcelona (Figures 16-17) is an example of high-symmetry grid which, like the tartan grid, permits a space between the primary elements. Here, the edges are thickened as in the tartan grid but also the vertices are expanded into rotated squares to allow the intersections to open up and allow for better traffic flow.

1 See Stiny and Mitchell.

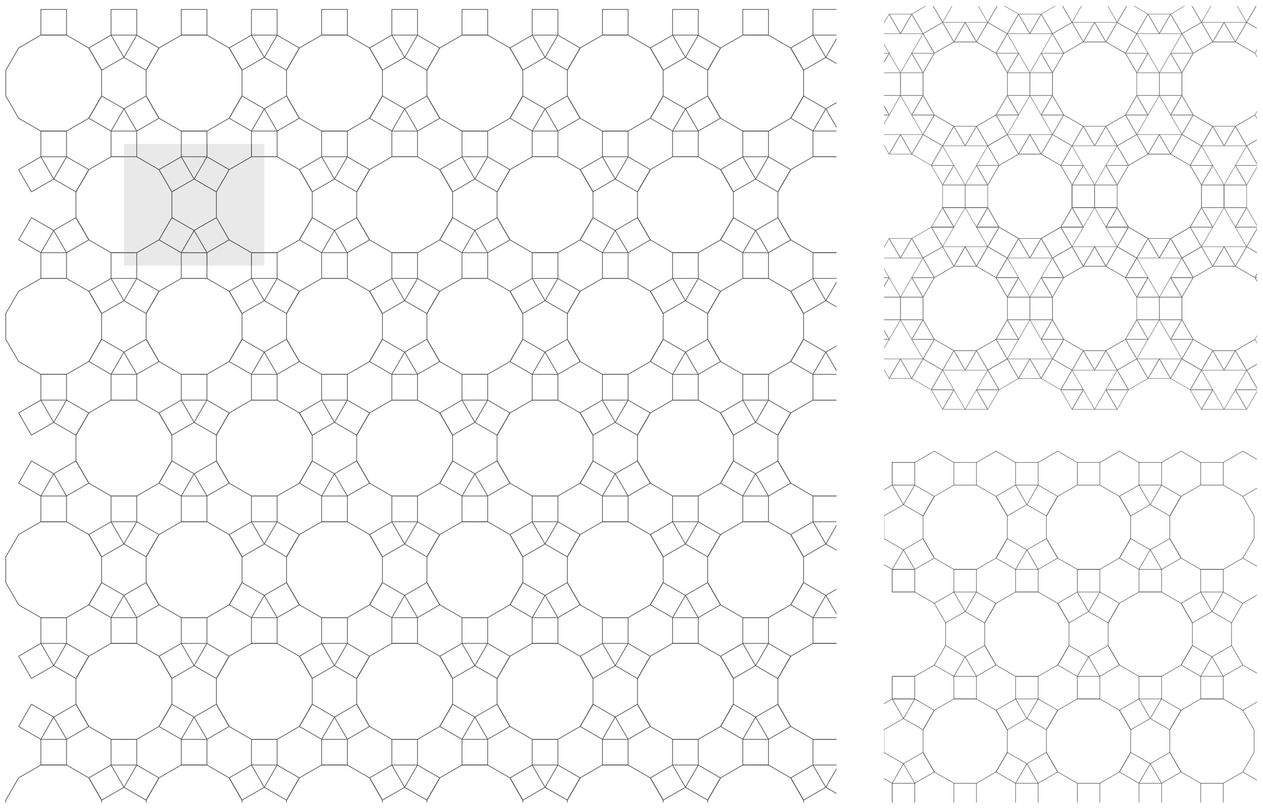


Figure 14. Examples of grids produced with regular polygons which demonstrate a high degree of symmetry.

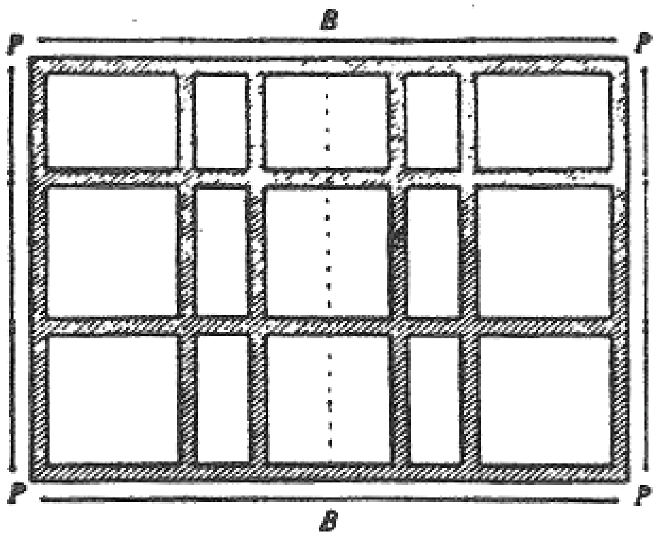


Figure 15. The tartan grid of Palladio's Villla Malcontenta.

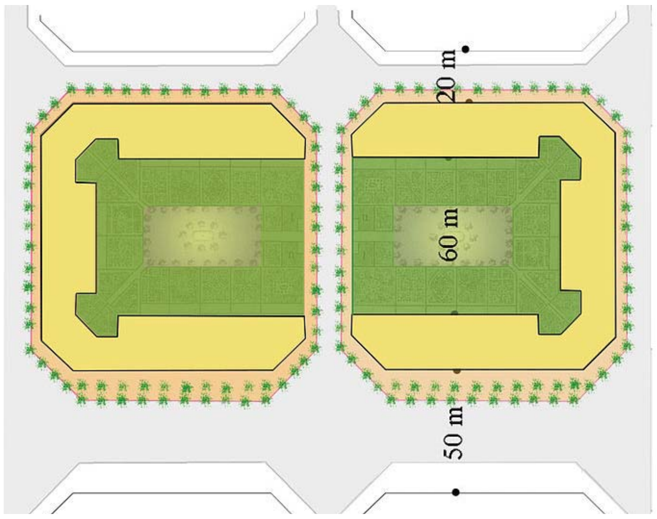


Figure 16. Blocks from the Barcelona street grid.

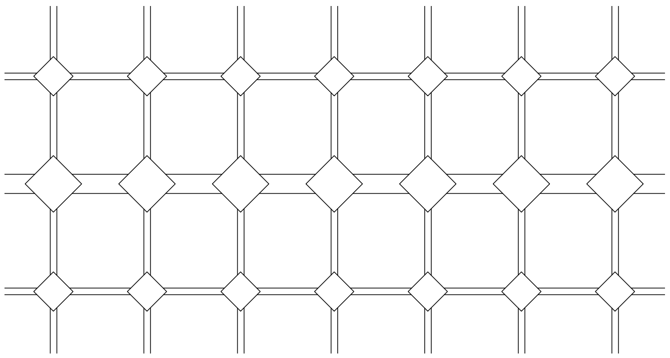


Figure 17. The Barcelona street grid abstracted.

6| Local Forcing

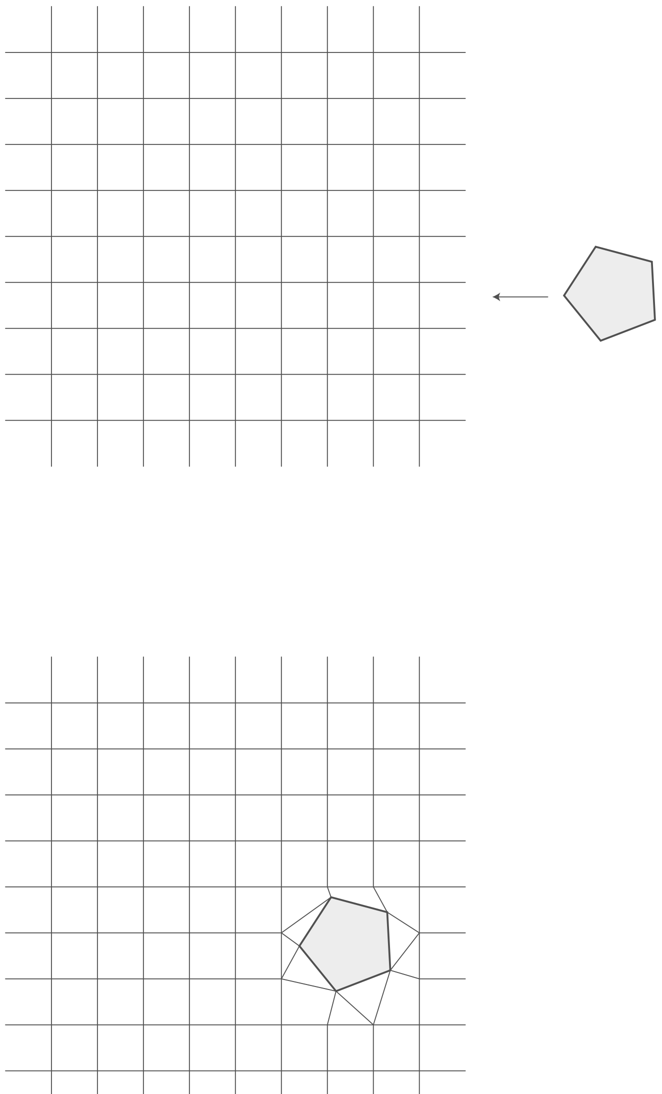


Figure 18. Local forcing.

A first step in generating softness in a grid is to start with a hard grid and to make local adjustments to fit some organizational requirement. Local forcing is a design move made to meet a specific need after a grid has been established. This approach can be interpreted as breaking the rules that of a particular grid and although that grid can be hard, the same logic can be applied to grids that are soft in various ways.

Grids generated by local forcing necessarily show a great deal of adaptivity. As far as the other criteria for softness, the definitions are not always applicable because of the fact the local forcing is dependant on the initial grid. The specific techniques for generating local forcing grids are wide ranging and difficult to fully categorize. The ideas of local stretching and local meshing will be discussed but it should be born in mind that this is by no means a complete description of the possibilities of local forcing.

Local Stretches

Applying a point-wise interruption to an otherwise hard grid can be extended to the construction of a grid based on a set of points or a set of obstacles or interruptions such as, in the case of architecture, a street, a river or a tree. In the case of a point-set, we can look at algorithms for charges in a field (Figure 19) and laminar flow (Figure 20). These types of local forcing allow the grid to expand or contract at certain specified points in order to accommodate design elements that may not fit into an existing hard grid. Clearly, the systems are highly adaptive but the other soft properties that are introduced are done so only at a local level, leaving the bulk of the grid relatively hard.

Local Meshing

A variety of meshing algorithms and techniques exist to grid a region which will not adhere to hard gridding techniques. This is a form of local forcing in that the grid will adapt locally to the space that it is responsible for organizing. This topic will be expanded on in the section on surface meshing in the next chapter.
For obstacles, this can be accomplished using meshing algorithms from computational geometry[1] as a way of subdividing the region after it has been interrupted (Figure 21). Here we see a square grid and a set of six circles which do not fit nicely into the organization. In this case, a triangulation of the surroundings is being used to organize the interstitial space in a way that makes it usable. This is a relatively simple meshing but they can be highly involved and there is a large body of material on the subject.

1 Berg et al.

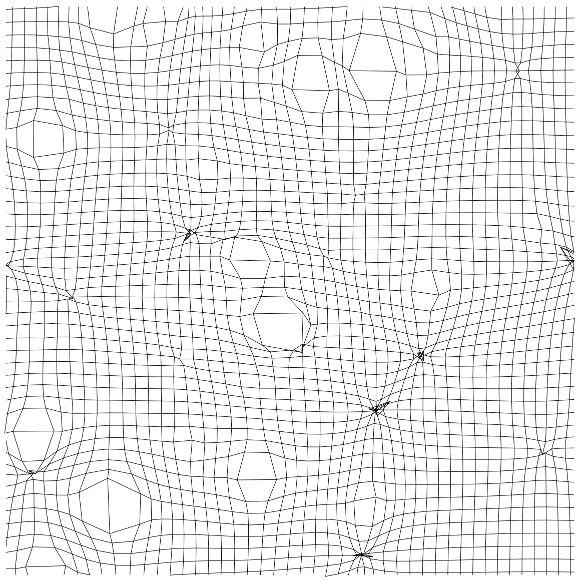


Figure 19. A discrete stretching and contracting. See appendix for code.

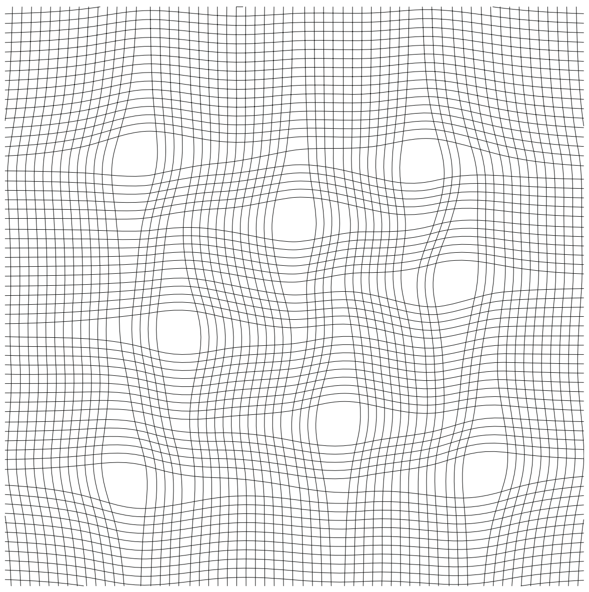


Figure 20. A continuous laminar flow local forcing. See appendix for code.

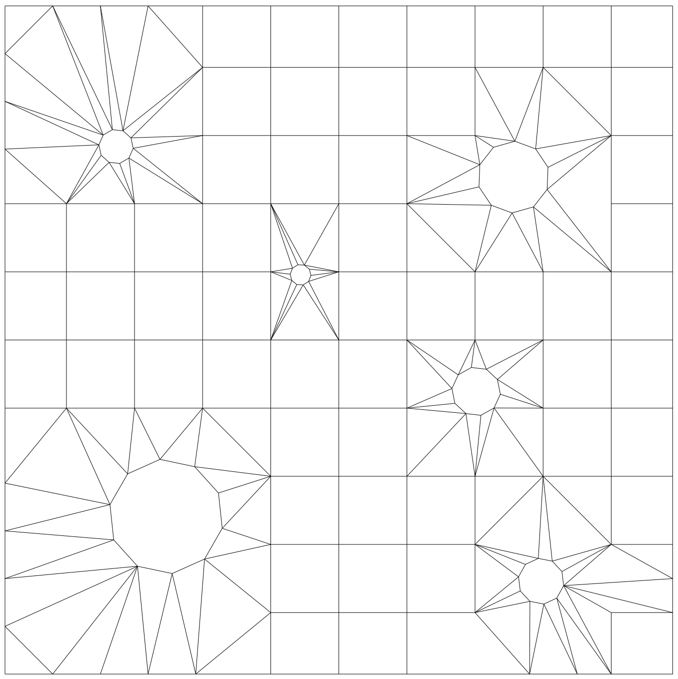


Figure 21. A local forcing based around six circles.

7| Global Systems

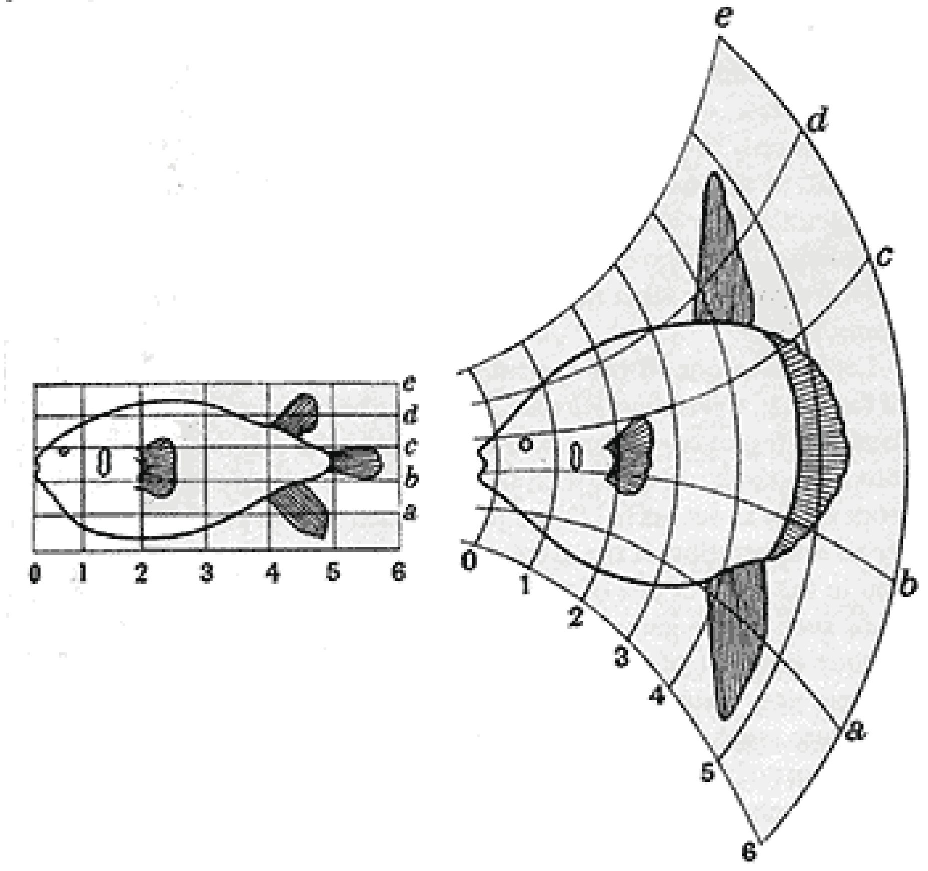


Figure 22. D'Arcy Thompson's global gridding system.

After local forcing, one might consider the grid in its entirety as the subject of softening instead of approaching it at the scale of the vertex, cell or edge. D'Arcy Thompson's treatment (Figure 22) of the continuously distorted square grid[1] is a beautiful introduction to the logic of non-linear transformations. In his case, the grids are being used as an analytical tool for morphology. In addition to these transformations, this chapter will discuss grids induced by global functions in the plane as well as surface tesselations which is a important not only in the fabrication of geometry but also for computational purposes as in Finite Element Analysis[2].
Global distortions are easily generated by many contemporary software packages so their use has become quite common in architectural practice and speculation. NURBS surfaces are parameterized by u-v coordinates which are a means of stretching the square grid onto a geometry in order to impart on it an organizational system for design or fabrication. Adaptive meshing techniques decompose a surface based on its intrinsic geometry rather than forcing on it a foreign coordinate system.

1 Thompson pp 268-325

2 Hughes

Continuous Planar Distortions

The continuous distortions presented here are a mapping from the plane to itself. In mathematical notation, we can say:

$$D : \mathbb{R}^2 \to \mathbb{R}^2$$

Such that the mapping does not overlap itself. This condition is called injectivity i.e., there do not exist x, y $\in \mathbb{R}^2$ such that D(x) = D(y). We also require that this mapping be continuous, so that it does not jump quickly from one place to another. What this means is that the distortion bends and stretches the plane into a new configuration without making any overlaps or jumps.

As shown in Figures 24-25, we can compress certain regions all the way into individual points. This gives a topological change that is most familiar in polar coordinates or in the standard grid applied to a sphere, namely, latitude and longitude, as we will discuss further in the section on surfaces. Any grid can be used as the initial organization as is shown in Figure 26.

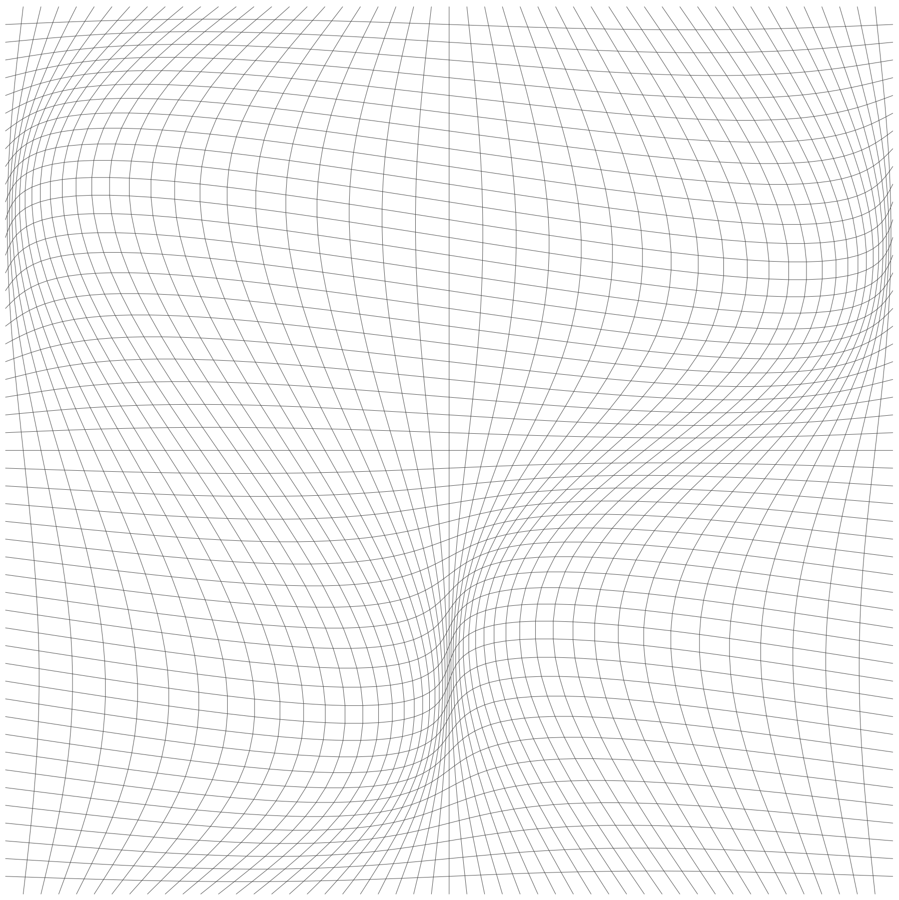


Figure 23. A continuous planar distortion. See appendix for code.

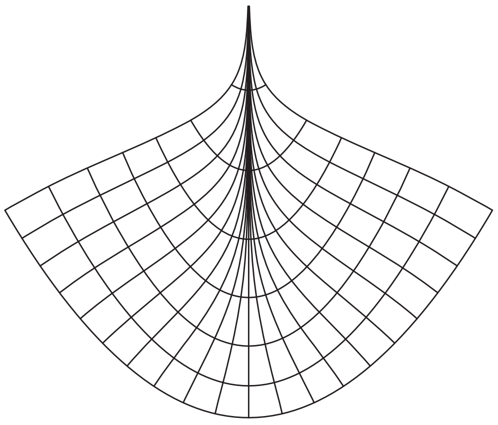


Figure 24. Continuous planar distortion.

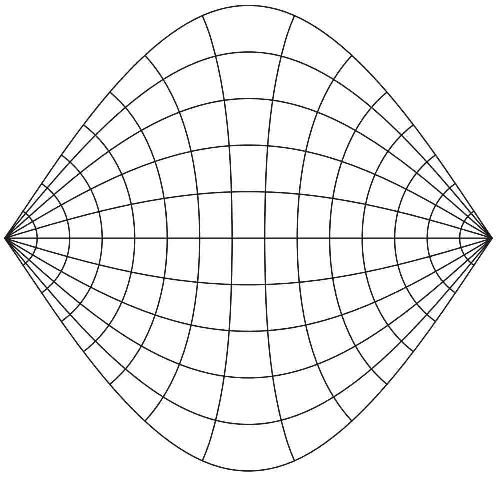


Figure 25. Continuous planar distortion.

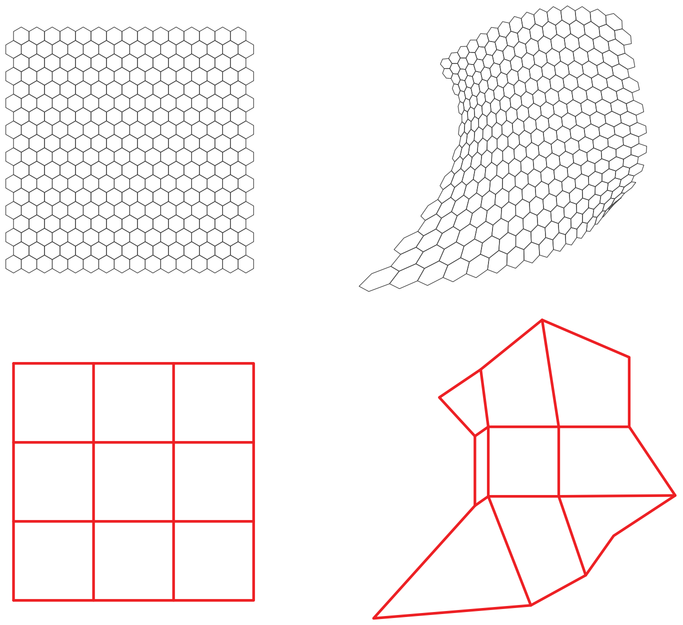


Figure 26. A continuous planar distortion of the hexagonal grid.

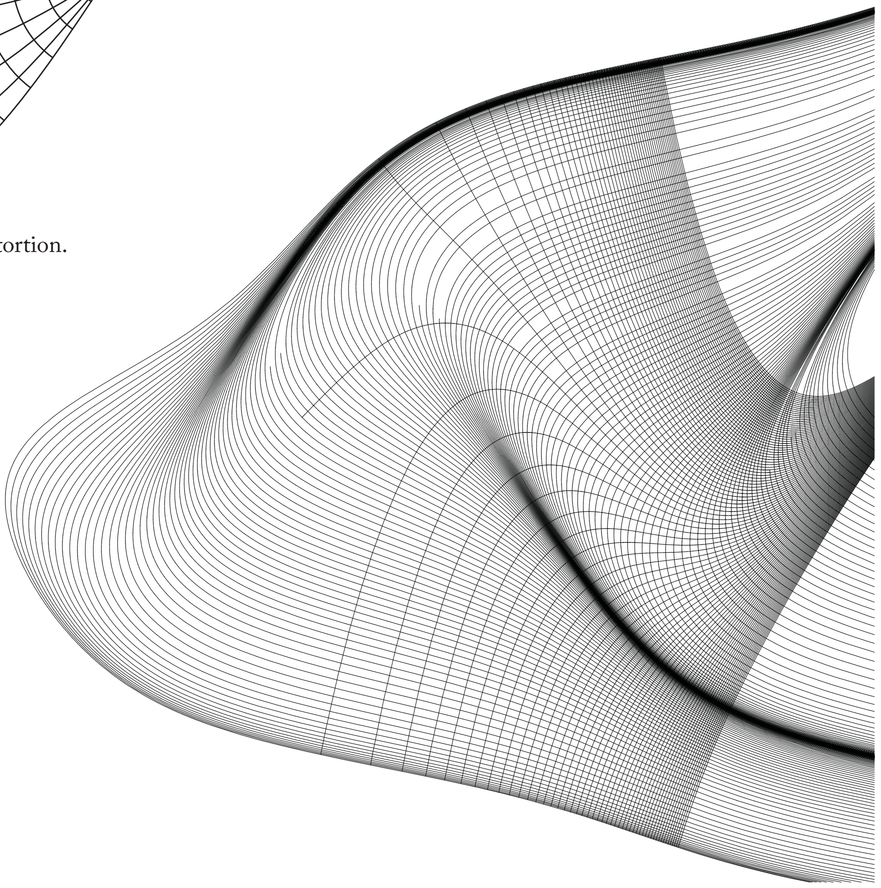


Figure 27. A large scale planar distortion.

Surface Distortions

Just as a grid can be used to map the plane onto itself, so too can a plane be mapped onto some other surface. This technique is most commonly used in computational practices with NURBS surfaces, which are parameterized using u-v coordinate systems, a direct extension of the square grid in the plane. This transformation has a deep relationship to the surface in question as can be seen in the images of the torus and the sphere in Figures 28-29.

In the case of the sphere (Figure 28), we see that there are two points (the north and south poles)[1] at which many lines converge. This is a result of the non-Euclidean geometry of the sphere that allows many seemingly parallel lines to intersect in one point. The same phenomenon occurs in a distortion of the plane when we look at the polar coordinates, which is the interpretation of the cardinal directions when one is at the north or south pole.

In a very different way, the topology of the torus is apparent in its canonical gridding (Figure 29) which does not have any points that are fully collapsed but instead wraps around in two directions, one going around the hole in the middle and one wrapping the thin tube of the torus.
In this way, the grid is highly specific to the topology of the surface, meaning that bending or stretching the surface (without closing up holes) can be achieved with the same grid.[2]

Surface distortions are in common use in today's architectural practice (Figure 30) but they are not a new device. As far back as antiquity in many cultures, domes, arches and other curved surface geometries in architecture have been realized with the help of a surface distortion grid such as is the case in the Pantheon in Rome (Figure 31) which uses a spherical mapping to organize the small vaults in the interior dome[3].

1 The poles of Earth are an artifact of the gridding system that is applied to it. Cardinal directions (north, east, south, west) were originally a local square grid on a plane derived from the movement of the sun in the sky, which is a result of the spinning of the orb. This leads to the poles lying on the axis of rotation.

2 For a good introduction to topology and geometry and the relationship between the two see Reid and Szendroi

3 The dome of the Pantheon is, in fact, spherical.

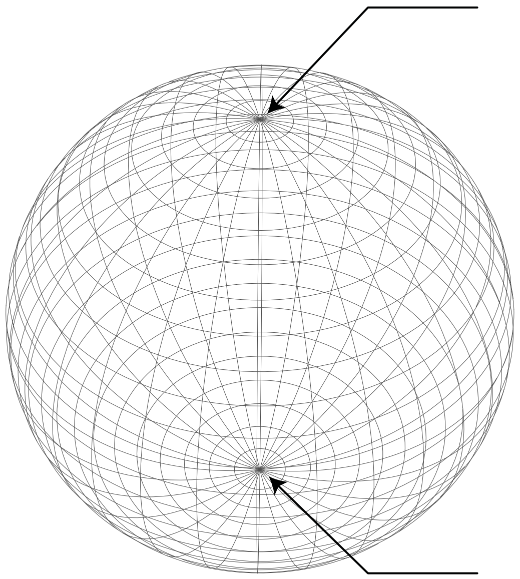


Figure 28. The sphere with the canonical latitude and longitude grid.

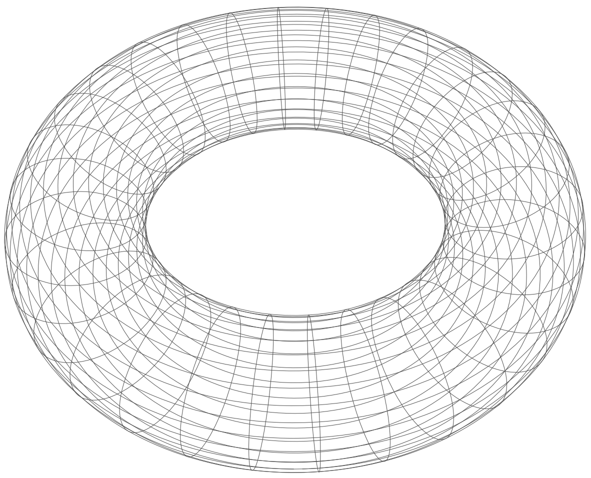


Figure 29. A torus with a grid that wraps in two directions

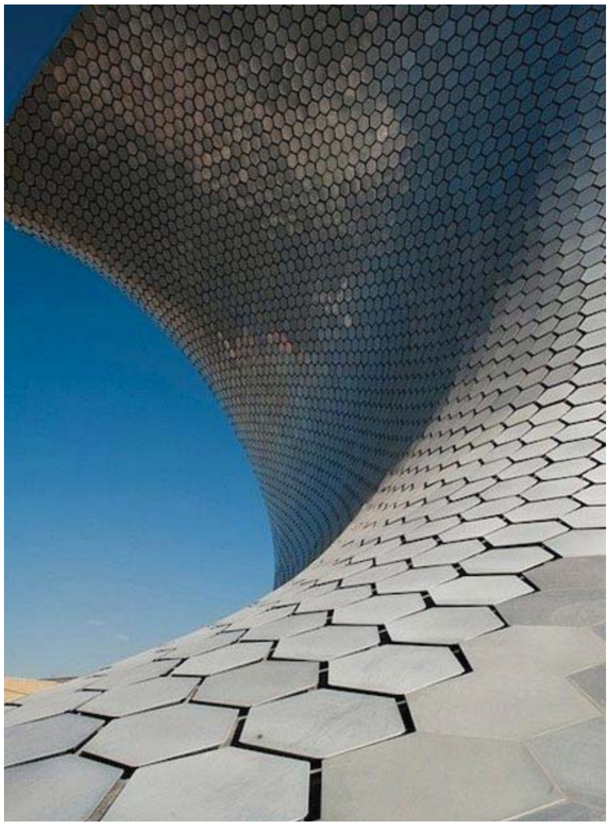


Figure 30. Museo Soumaya by FREE Fernando Romero EnterprisE.

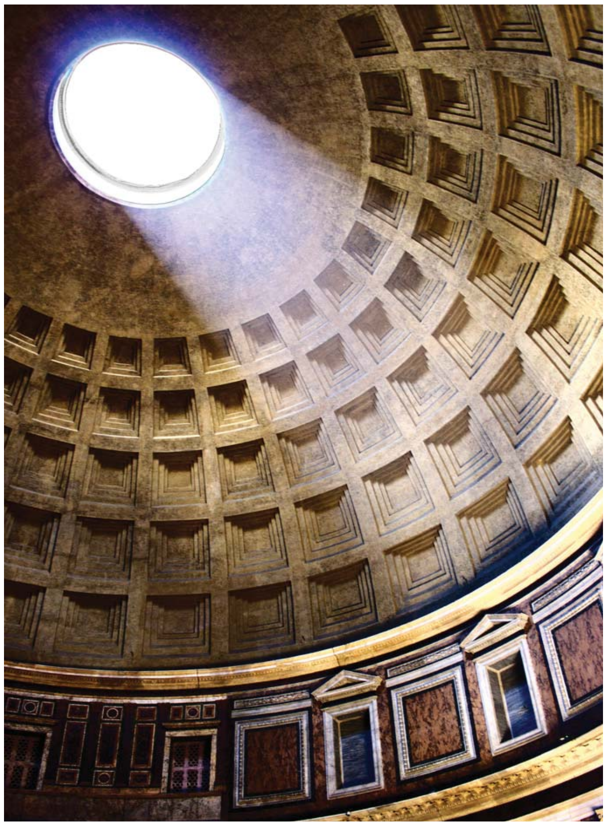


Figure 31. The Pantheon, Rome.
Photo by : Kristoffer Winther Balling.

Surface Meshing

Just as grids can have a deep relationship with the topology of a surface, so too can they relate to the geometry of a surface in the sense of *intrinsic* properties. An intrinsic property is one that appears only locally like the curvature at a point. If we cut away an area of a surface, it maintains its intrinsic properties there unlike the topological properties which exist only in relation to the whole surface.

In the examples in Figures 32-33, a grid is applied to a surface in the form of a triangular mesh. In some cases, this type of triangulation is done to an existing surface as a way of representing it in a computational context or fabricating it, but the meshes can also be a consequence of the construction or calculation of the surface itself. The SoaPy algorithm for computing minimal surfaces generates the surface using a dynamically generated mesh.[1] It simulates the behavior of soap films on wires by calculating surfaces with zero mean curvature or minimal area. This is achieved by repeatedly minimizing the curvature on the mesh and refining it. The mesh refinement takes into account the intrinsic properties in order to find an optimal configuration that suits the surface with the greatest precision possible.

This type of procedure for grid generation is extremely powerful and is commonly used not only in architectural projects but also in computer graphics, gaming and a wide range of other digital environments. It is highly adaptive in that the grid is a result of that which it is gridding. It is asymmetrical, as much as the surface is, but not very entropic as it is quite predictable. It is variable, again, to the extent that the surface is and it is highly scalable as it creates a finer grain to map finer geometry.

1 see http://fab.cba.mit.edu/classes/MIT/864.11/people/Ari_Kardasis/Soapy/index.html

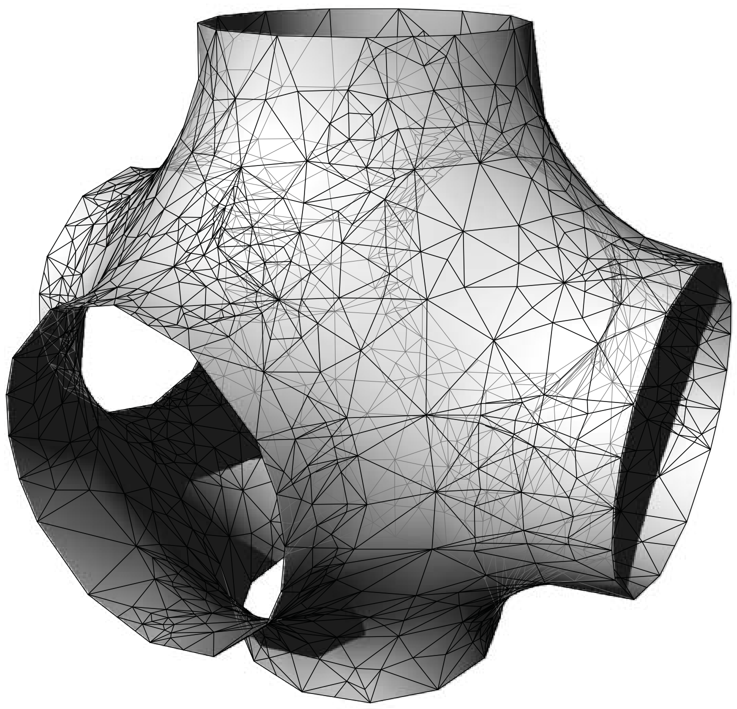


Figure 32. A mesh generated for a minimal surface by the software package SoaPy, by the author.

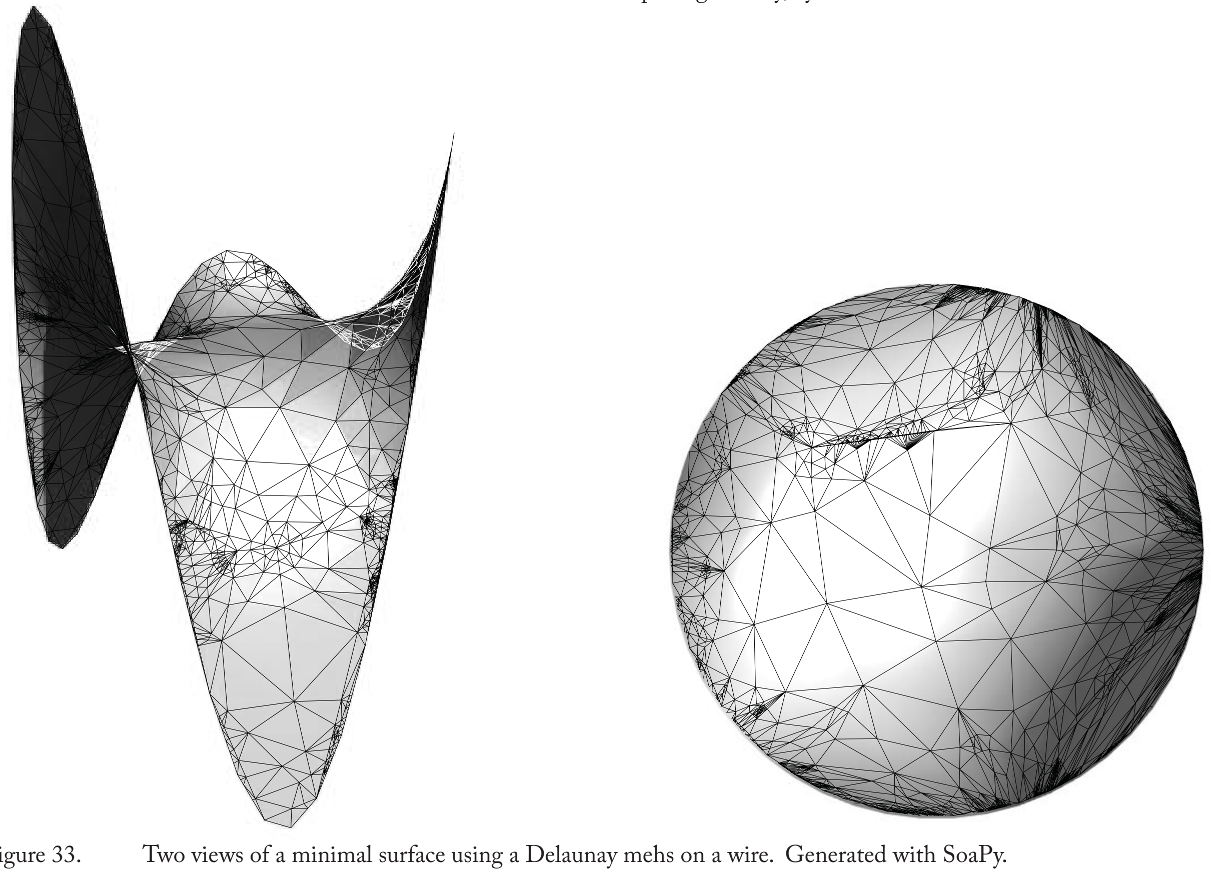


Figure 33. Two views of a minimal surface using a Delaunay mehs on a wire. Generated with SoaPy.

8| Voronoi Diagrams

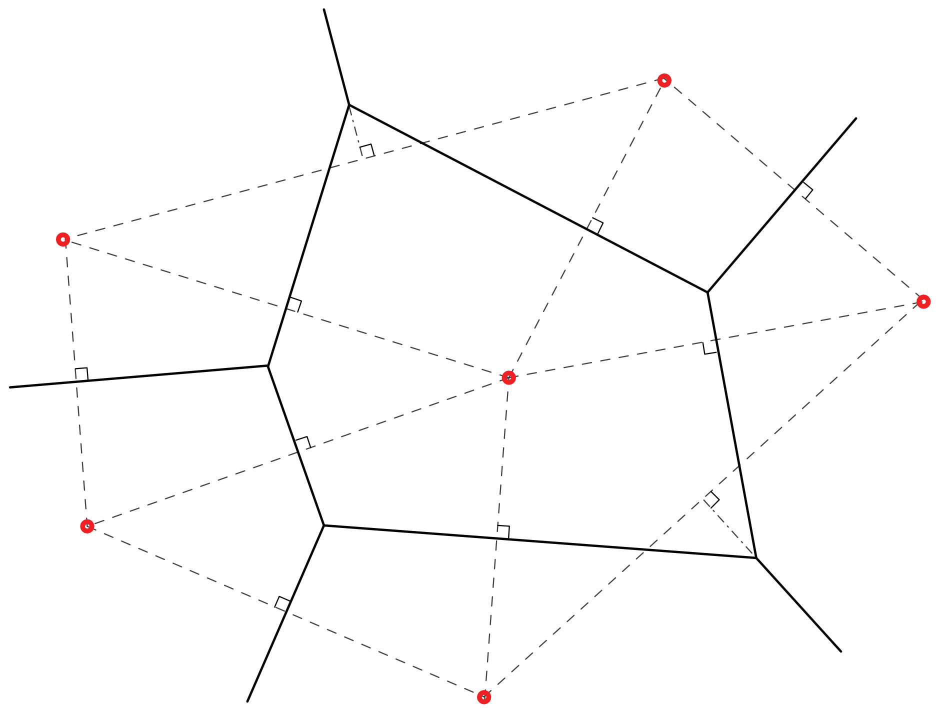


Figure 34. The Voronoi diagram marks the regions in a plane which are closest to a point. This can be generated by drawing the perpendicular bisectors of the lines that connect the points, which are the Delaunay triangulation .

Voronoi diagrams are often presented in conjunction with their relationship to nature. Butterfly wings, leaf venation, cell structures, bubbles, etc, are all examples of naturally occurring Voronoi diagrams. This is however, only a very small part of the story. Any point set, in any number of dimensions, will generate a Voronoi diagram and only a small subset of these are those that are found in nature. The natural patterns arising from Voronoi diagrams, while beautiful, tend to be only moderately structured and lack the hierarchy necessary for design. These are the same versions borrowed by architects and due to their lack of order, direction or scale, they fail to be capable of imposing the will of the designer on a project. In short, they are too soft.
But the depth of the Voronoi diagram does not stop at the mimicry of natural phenomena. We must not forget that a square Cartesian grid is the Voronoi diagram of a square lattice of points. Similarly, rectangular grids are the Voronoi diagram of a rectangular lattice and hexagonal grids are the Voronoi diagram of a triangular lattice and vice versa.

Random Arrangements

The Reichert Interestingness Curve (Figure 35), which estimates the intellectual value of the Voronoi diagram over time, is marked by two major features. The first, a sharp spike at the beginning corresponding with the initial intrigue associated with the Voronoi diagram, is primarily associated with Voronoi diagrams generated from randomly placed points in the plane. Because there is little to no overall structure from these types of grids, they lack the organizational capacity of many of the other grids discussed in this thesis.
Because of this lack of structure, the randomly generated Voronoi diagram is in many ways *too soft*. In opposition to the hard grids, they have 100% variability, asymmetry, scalability and entropy. Adaptivity is a different issue but they can be adapted to nearly any situation although not while maintaining randomness. The one structural property that they do maintain is trinodal vertices[1].

1 Three edges meeting at each vertex.

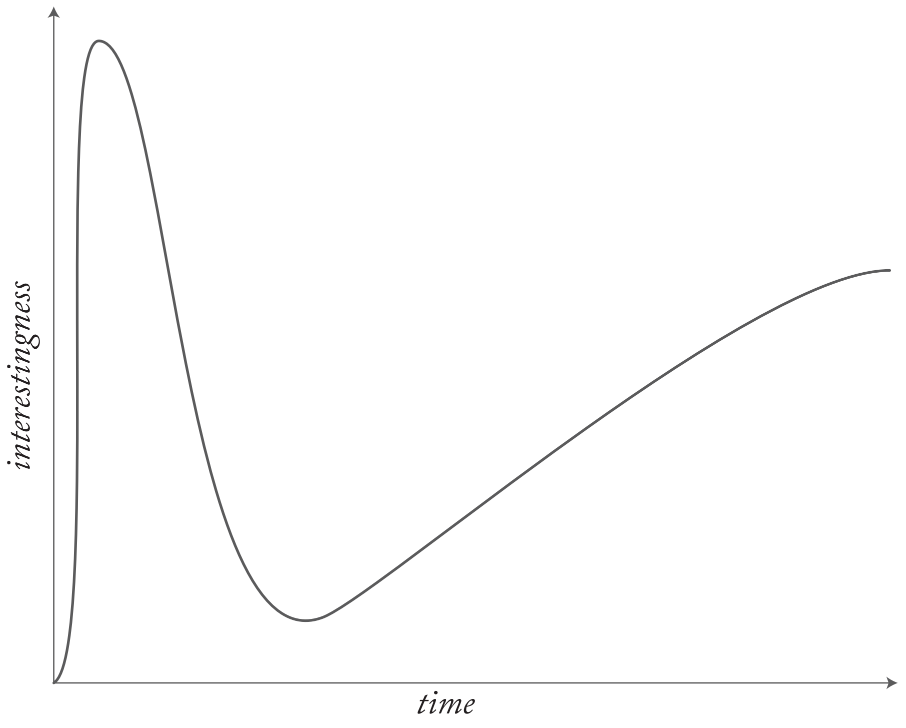


Figure 35. The Reichert Interestingness Curve. Steffen Reichert.

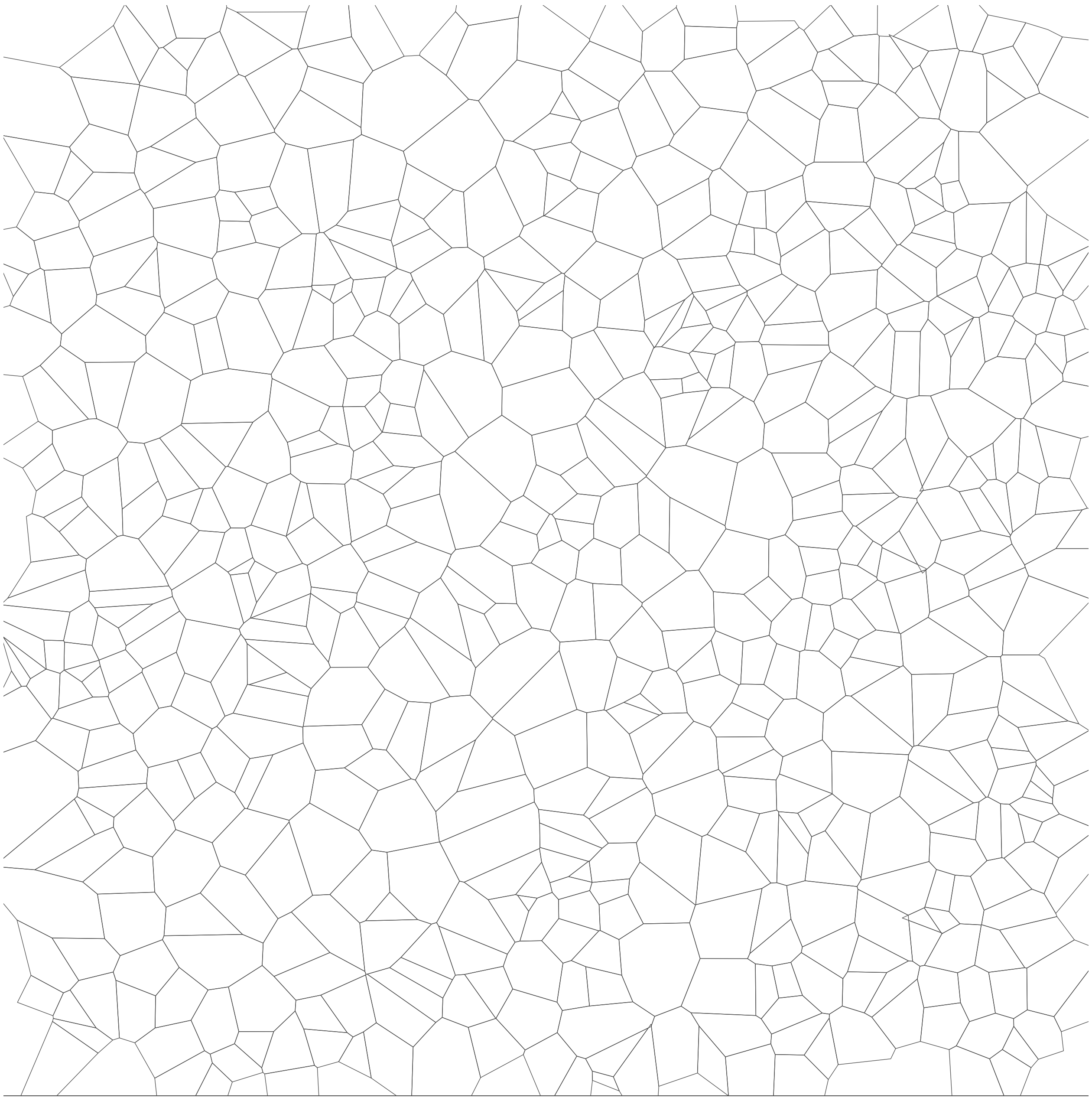


Figure 36. A Voronoi Diagram generated from 500 random points

Structured Arrangements

The story with Voronoi diagrams does not end with random arrangements. In fact, any set of points can be used as input to generate grids in the plane and the second feature of the Reichert Interestingness Curve is largely due to the realization that Voronoi diagrams can be controlled in highly specified ways. Figure 37[1] shows how square and hexagonal grids are in fact special cases of the Voronoi diagram and that this distinction need not be absolute. The power of Voronoi diagram is in its ability to morph, shift and in fact mimic other grids but with control.

The fact that Voronoi diagrams cover the space of the hard grids gives them the ability to make transitions from one type of grid to another. Figure 38 shows the continuous transformation from a square grid to a hexagonal one. The use of voronoi diagram to stitch together hard grids is quite powerful. In design situations, there is often the case where disparate grids intersect leaving an unresolved edge condition between them. This also applies to the design problem of a continuous scale shift (Figure 39) where a system has to be maintained but multiple local conditions must be met.

Figure 40 shows a grid generated from a phylotaxis pattern[2]. This particular grid displays some important qualities that the Voronoi diagram affords. We can see an overall radial organization with one central cell. As the spiral moves outward, the local conditions shift from a rectangular grid, to a hexagonal one, to square, back to hexagonal and then to rectangular again. This is a complex system of local organization bleeding continuously into one another based on a separate global organization, all of which come together to make an intricate grid.

3D

Voronoi diagrams can also be used to grid three dimensional space although the applications to design are not as apparent as they are in the planar case. Spatial organizations like these are often refered to as *foams* in material science and their structural and physical properties are quite incredible[3].

1 See appendix for code.

2 Points are generated using Fermat's spiral: $r = \pm\sqrt{\theta}$

3 Weaire & Hutzler

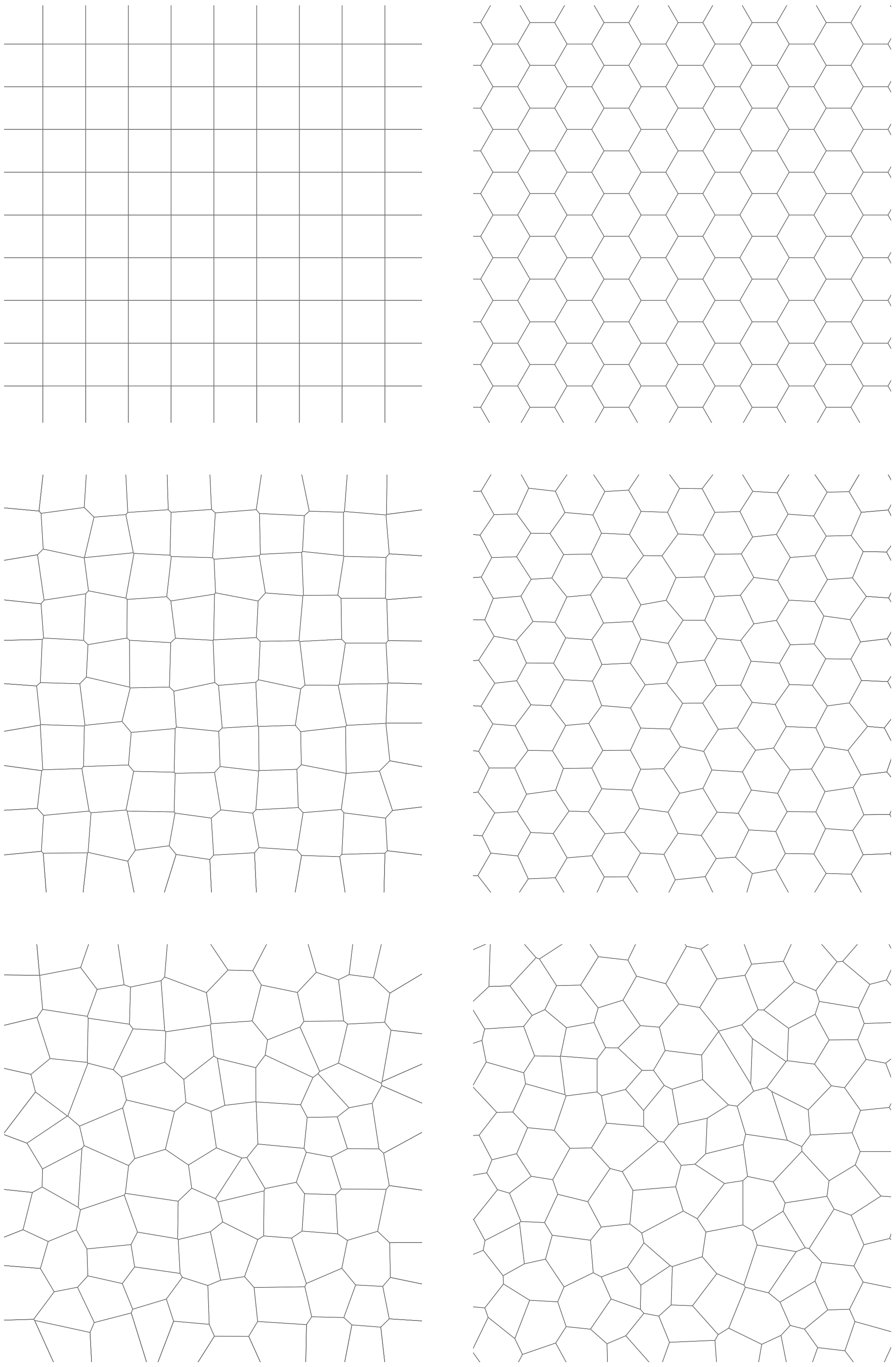


Figure 37. The degeneration of a square and a hexagonal grid into randomized Voronoi Diagrams

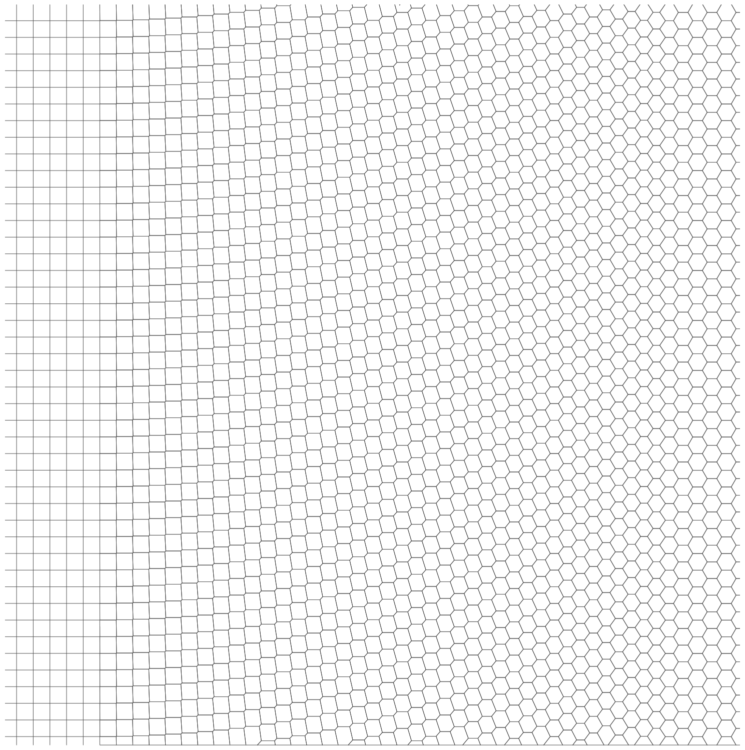


Figure 38. The continuous change from a square to a hexavonal grid using Voronoi diagram

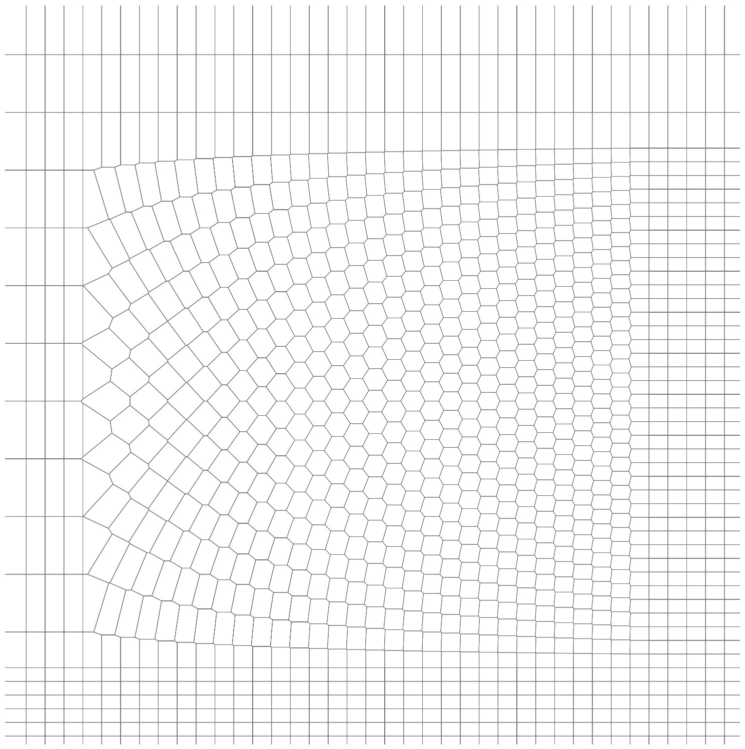


Figure 39. An intruption in a hard grid using the fact that Voronoi diagrams can stitch together the square and rectangular.

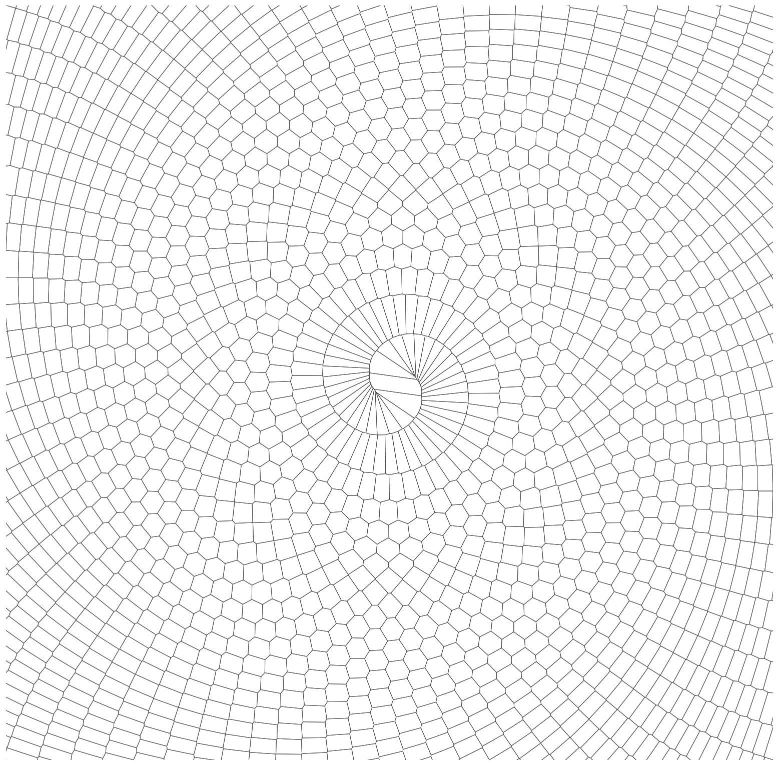


Figure 40. A Voronoi Diagram generated from a phylotaxis pattern.

Figure 41. Soap bubbles creating a 3D Voronoi diagram
http://www.thepigeon.org.uk/?p=1

9| Subdivision Schemes

A subdivision scheme is a technique for gridding that starts with some unit or cell and recursively cuts it up into smaller pieces, eventually giving sufficient structure to the space to warrant a grid. Much like Voronoi diagrams, subdivision schemes can be used to make the hard grids (square, rectangular, triangul etc.) thereby making them a prime candidate adapting hard grids to soft.

In general, subdivision schemes afford a high degree of scalability above all. They can also be highly adaptive[1], variable and to a lesser extent entropic and asymmetrical. Because of the recursive nature of the generation process, computational algorithms lend themselves quite well to subdivision schemes which can require a lot of numerical calculation. For this reason, many of the soft grids in contemporary practice are generated with subdivision schemes.

1 Charalampous

Danzer Tilings

Danzer tilings have been used in several architectural and design projects, most notably the Watercube for the Beijing 2008 Olympics (Figure 43) but also a fair number of student and speculative projects from geometry minded designers such as the Grotto project by Aranda/Lasch (Figure 44). The Danzer subdivision scheme is a three dimensional example of an aperiodic tiling[1], which means that a finite number of tiles can be arranged in an infinite number of ways without ever repeating the same configuration. This property leads to infinite variation with a finite number of unit types, a property that is appealing when one needs to fabricate the units. In a thesis from the Bartlett School in 2007, Charis Charalampous extended Danzer tilings to be a responsive subdivision capable of altering the density depending on conditions in space.

L-System and Fractal Grids

L-Systems, first explored by Aristid Lindenmayer, are a powerful mathematical tool that can be applied to generate a wide range of subdivision grids. The Danzer tilings as well as the Penrose tiling (see below) can be generated using this clever, bottom-up, approach. The most common implementation of L-Systems is the fern type drawings (Figure 42), which display much of the fractal and biomimetic properties seen in many of the L-Systems geometries.

1 See Grünbaum & Shephard pp. 520 - 604

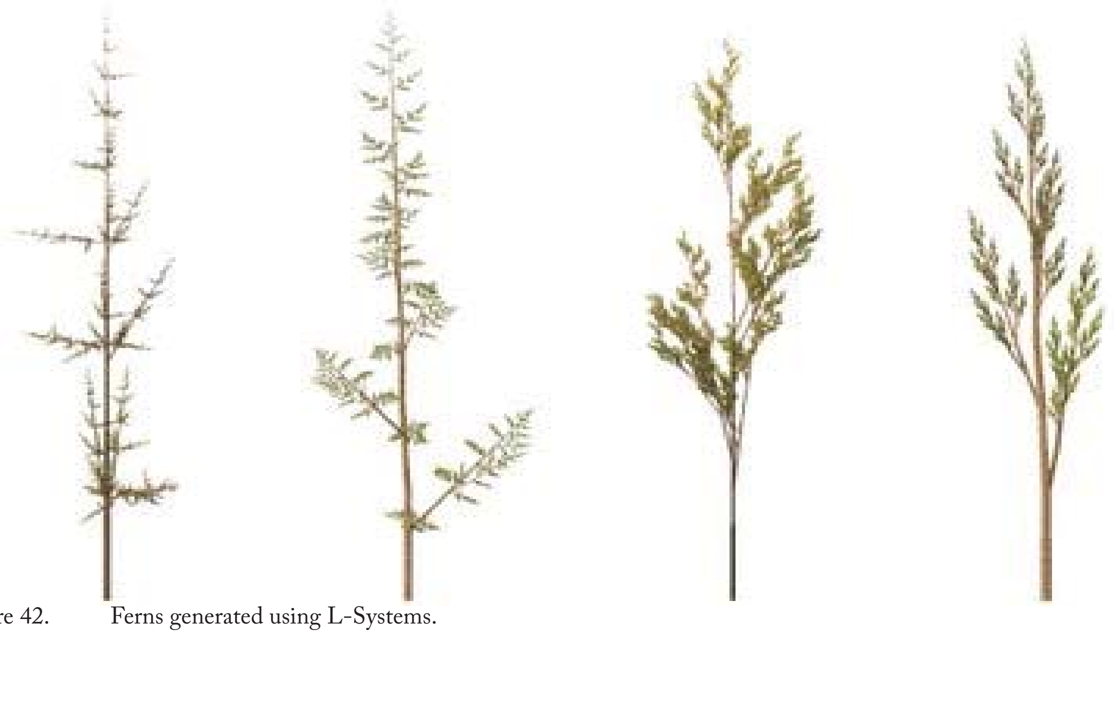


Figure 42. Ferns generated using L-Systems.

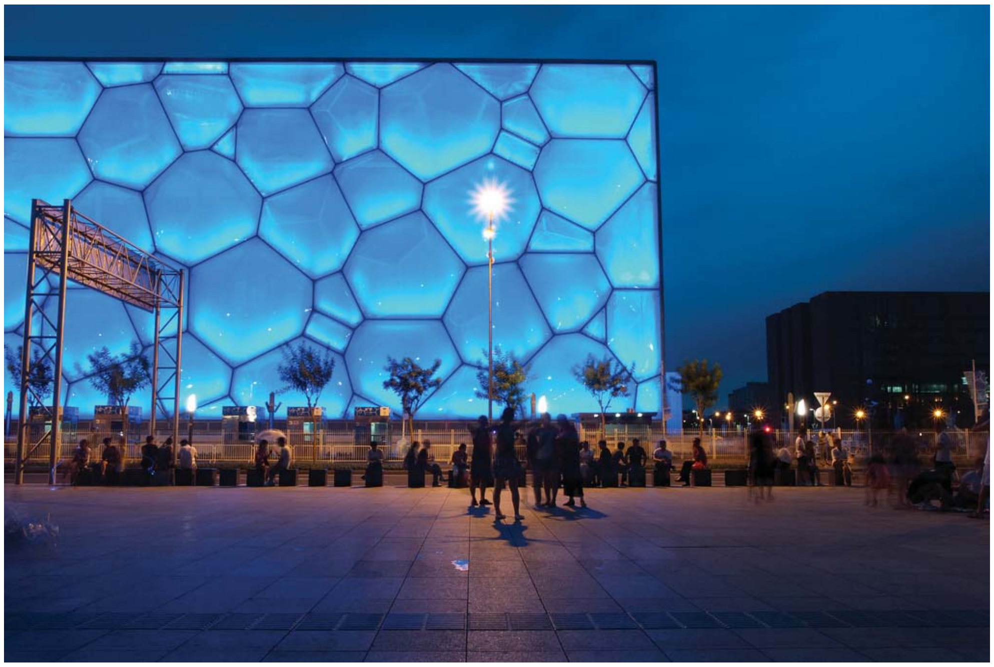


Figure 43. The Water Cube by PTW Architect.
Photo from Flickr user **inkelv1122**
http://www.flickr.com/photos/inkelv1122/4865475411/sizes/l/in/photostream/

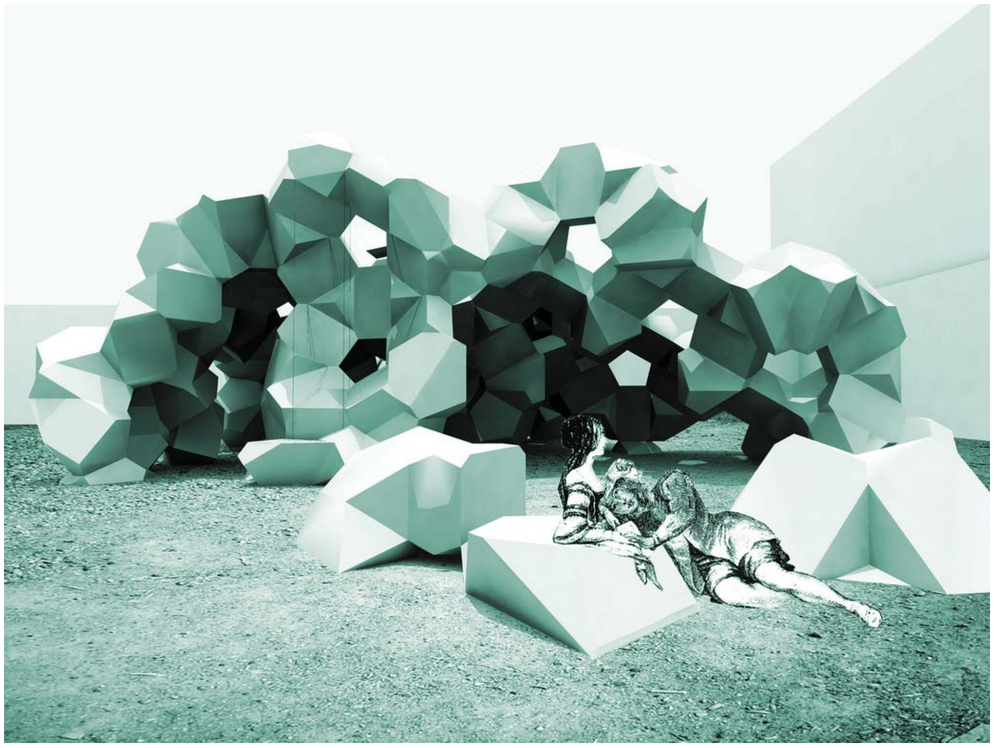


Figure 44. The Grotto. An entry into PS1's young architects competition.
Aranda/Lasch

Penrose Subdivision Scheme

Roger Penrose's discovery in the 1970's of a set of aperiodic[1] tiles (the Penrose Tiles) has been thoroughly studied by mathematicians over the past four decades. Although the resulting grids can be generated in a number of different ways[2], Penrose originally used a subdivision scheme which we will show here[3]. The following description is outlined visually in Figure 49.

First, we have six *prototiles,* a star, a 'half pentacle' a rhombus and three regular pentagons. Although the pentagons are idential in shape, they have different labels, or rules, that determine how they will be subdivided with each step.
Each of these prototiles has a rule that determines how it is to be divided into smaller versions of the same prototiles. After this division, we have a new set of smaller tiles and we can again subdivide those according to the same rules. This procedure can be repeated indefinitely, generating not a set of tiles, but a true grid which organizes space in a systematic way. The resulting grid does not have any translational symmetry and has 5-fold rotational symmetry around only one point.

Furthermore, the six prototiles described can each be divided into a set of rhombuses whose angles are multiples of $\pi/5$ (or 36°). There are only two rhombuses that satisfy this property so the entire aperiodic tiling that Penrose created can be built with only two tiles. This property makes the tiling especially appealing for fabrication purposes as it is only slightly more difficult to manufacture than a perfectly hard grid! (Figure 45)

In their 2007 paper[4], Peter Lu and Paul Steinhardt discuss the techniques used by medieval Islamic architects and designers to generate aperiodic Penrose-like tilings at the Darb-i Imam shrine in Isfahan, Iran (Figure 51). These types of grids were known long before the 'discovery' by Penrose in the 1970's. In that paper, they discuss yet another technique for generating such grids.

1 For definitions and explainations of aperiodic, see Grünbaum and Shepard p 520.

2 I will discus one of the other approaches in the next chapter.

3 This method is adapted, and corrected, from Grünbaum and Shepard.

4 Lu and Steinhardt

Figure 45. Roger Penrose on a Penrosed tiled floor.

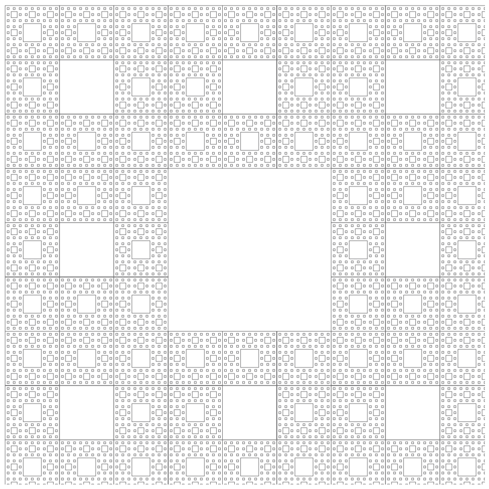


Figure 46. A Sierpinski carpet.

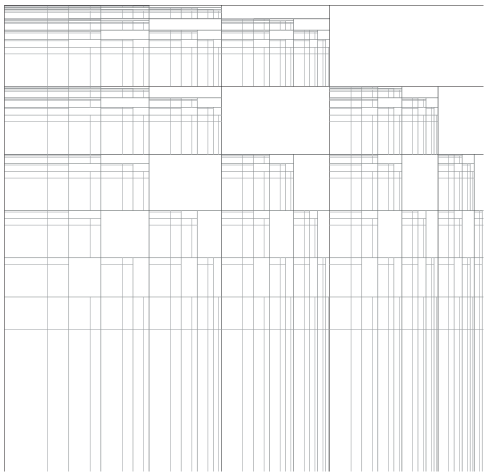


Figure 47. An L-System based on proportional subdivision.

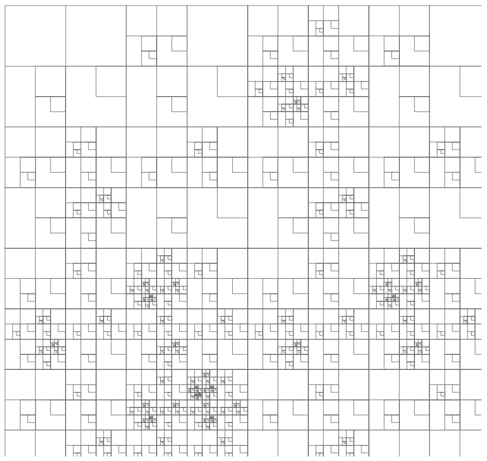


Figure 48. An L-System grid based on repeated scaling

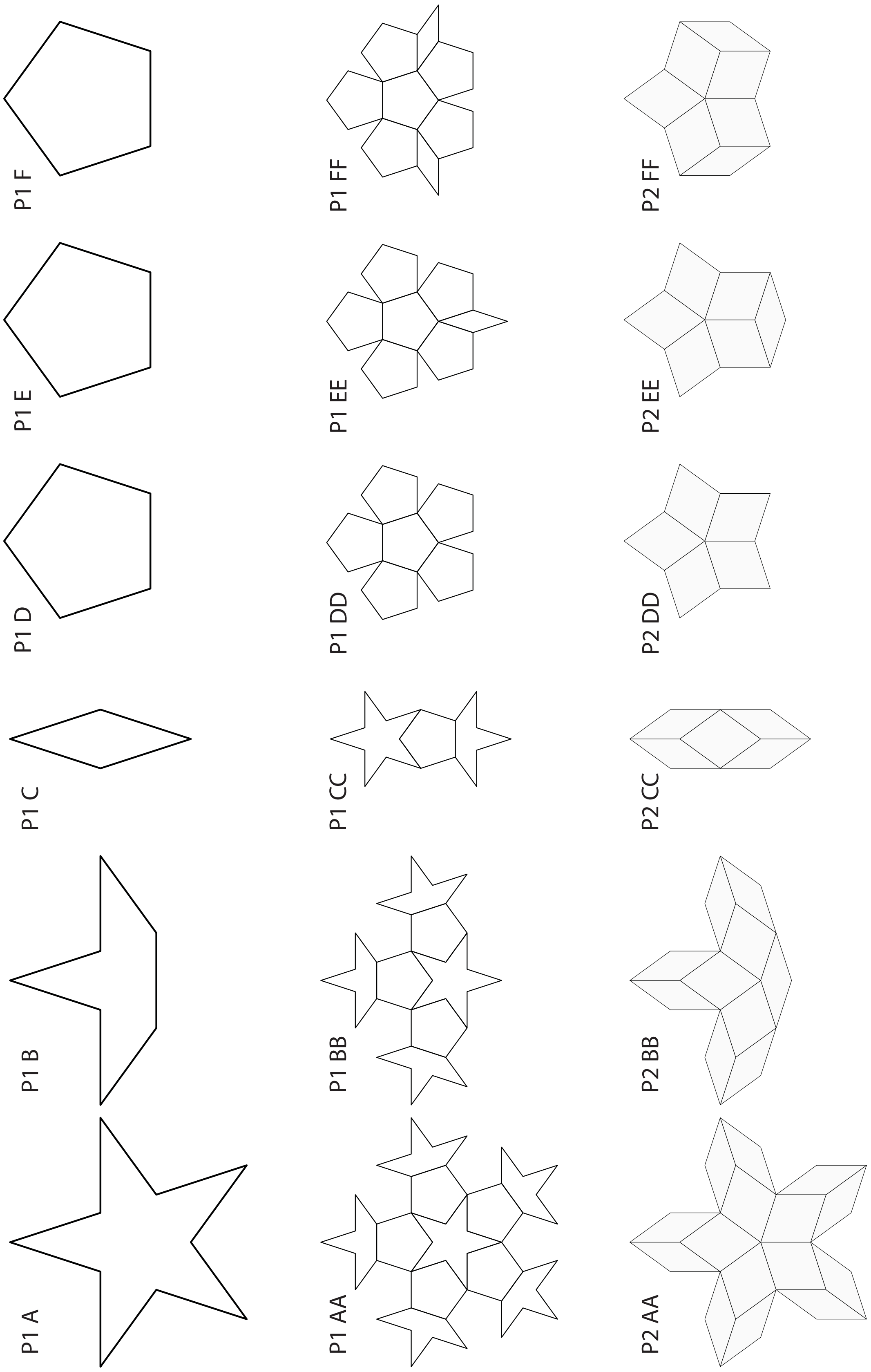


Figure 49. The Penrose subdivision scheme.

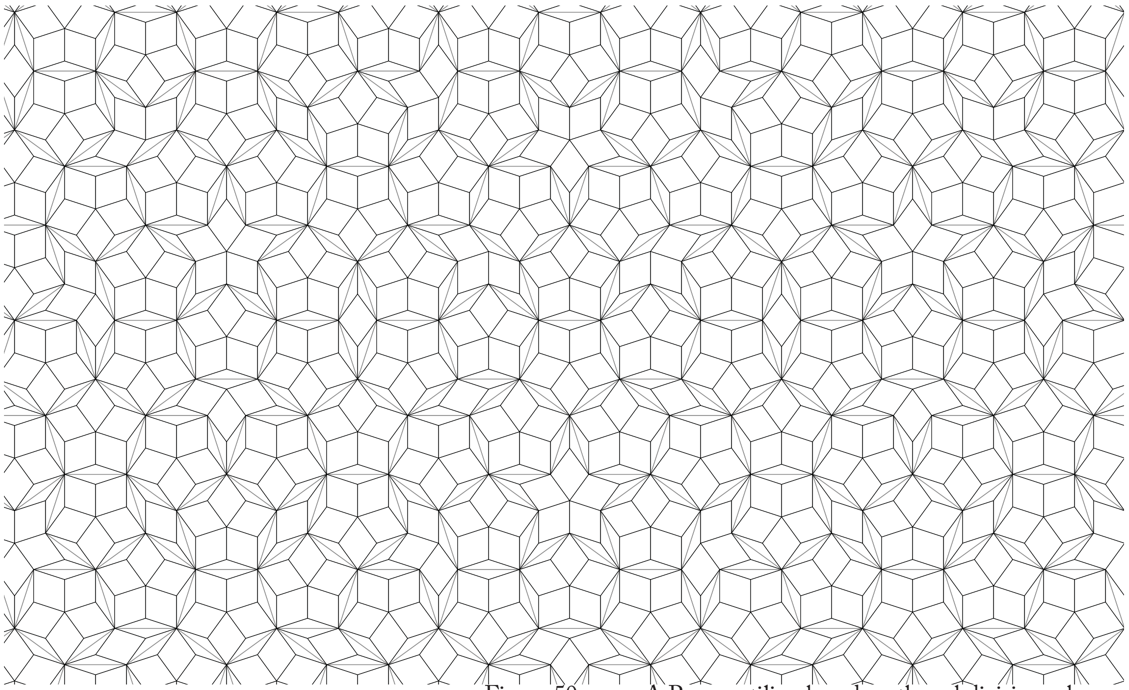


Figure 50. A Penrose tiling based on the subdivision scheme.

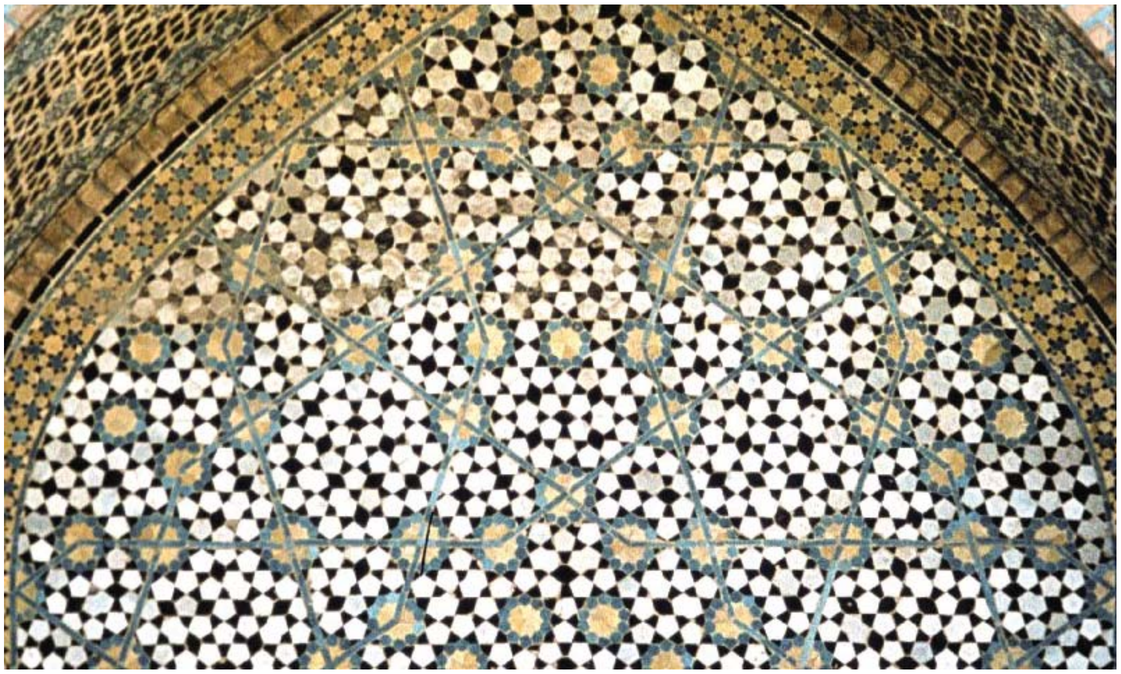


Figure 51. Darb-i Imam shrine in Isfahan, Iran

10| Parallelepiped Packing

This chapter is devoted to a series of computational design experiments performed by the author in the search for soft grids. A great deal of research into the aperiodic tilings revealed that the mathematics involved in generating Penrose tilings and other n-fold symmetrical tilings was extensible to a type of grid which was softer in many respects such as variation and adaptivity, while maintaining the requisite structure to be considered a grid.

In this chapter, the techniques used to generate these grids will be explained and some examples of the results will be shown. This being the final chapter of this thesis should not be taken as an indication that the p-ped packings presented here are in any way an ideal version of the soft grid or a culmination of the previous chapters. While there is a great deal of potential in these grids, they have not had the benefit of being tested in a design context and thus are immature as organizational tools. However, their benefits are apparent and while they meet the criteria of softness, they do not overextend it.

P-ped Packing of Order 3

The simplest parallelepiped[1] packing, other than the square, rectangular and parallelogram grids[2] is the one generated by the 60˚-120˚ rhombuses as detailed in Figure 52. The generation of such a grid can be thought of as a motif on a hexagonal lattice, as a packing of one tile oriented in three directions, or as three different p-ped cells each specified by two initial vectors chosen from three. It is this last definition that allows an extension into higher dimensions with a wider variety of p-peds.

Taking a relatively hard grid as a basis (Figure 53), we can successively soften it by determining larger scale regions (Figure 54), specifying more complex local conditions (Figure 55) and taking different initial cells as the generators for the entire grid (Figure 56). These steps will serve as the motivation for generating much more complex girdding systems that involve all of these together. One may note that the images on the opposite page can be taken as a two dimensional projection of three dimensional space. This is not a coincidence. The nature of these grids is that they are generated from three vectors in the plane. If these three vectors are taken to be projections from three dimensional space, our 3-D tuned brains will reconstruct that space. As we will see in the next section, using more than three vectors to start the process gives us what we may consider to be a glimpse into four, five, etc, dimensional space.[3]

1 The word parallelepiped is used here as an n-dimensional generalization of the parallelogram, particularly in two and three dimensions. This choice is made to for the sake of the 3D case.

2 Again, the hardest grids are a special case of these soft grids.

3 Talking about numbers of dimensions higher than three inevitably leads to a great deal of confusion. The simplest way to think about higher dimensional spaces is as a *numerical* extension of Cartesian space, i.e. (x, y, z) coordinates are three dimensional

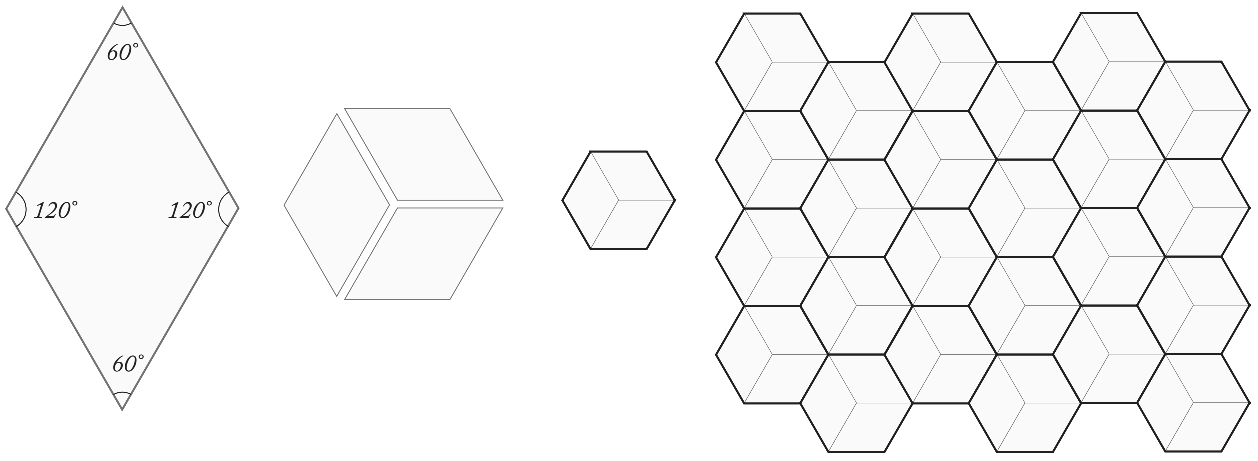


Figure 52. The construction of the 60˚-120˚ p-ped packing.

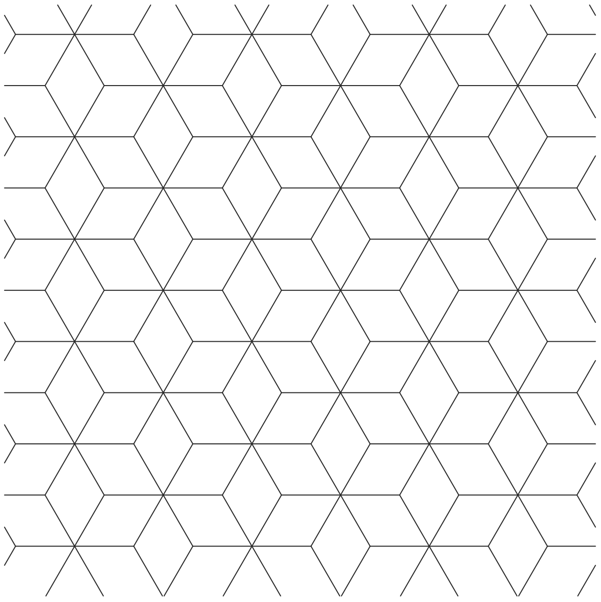


Figure 53. A hard grid built from three orientations of the same cell.

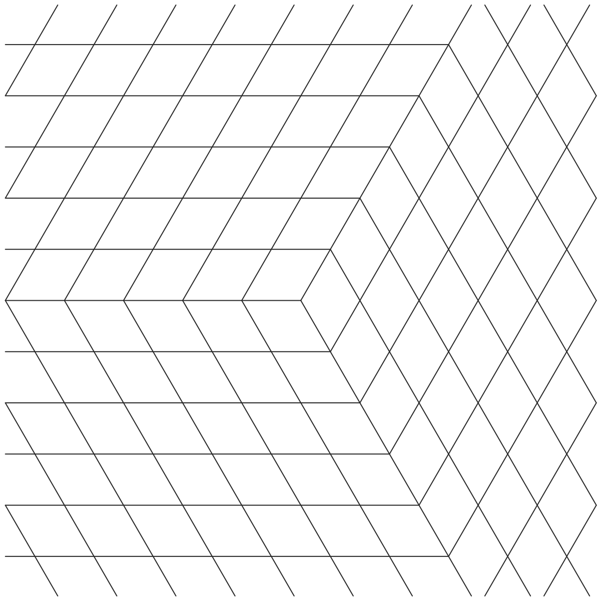


Figure 54. The same cells organized into three primary regions

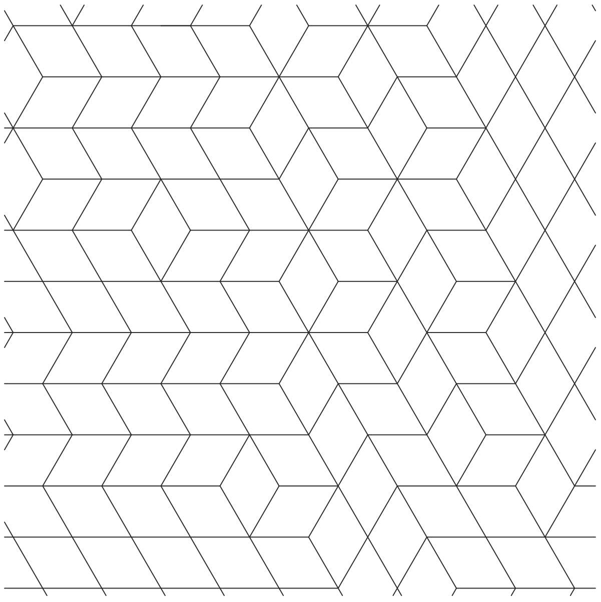


Figure 55. The organization of the cells can assume a much less regular form.

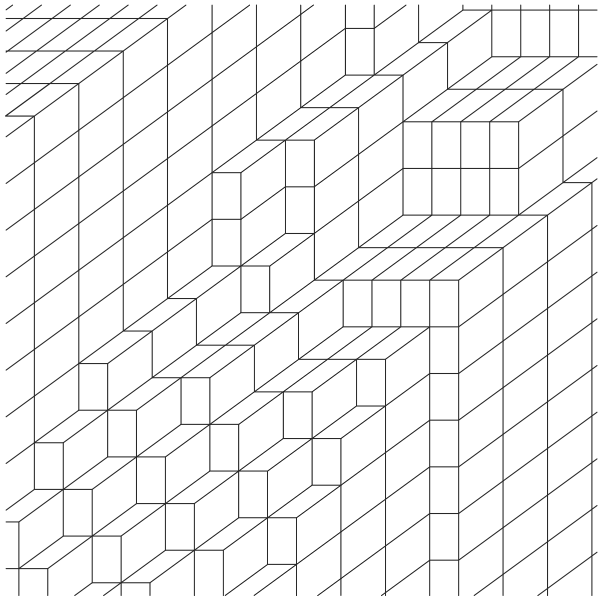


Figure 56. Altering the shape of the p-peds gives us a different organization entirely.

P-ped Packings of Order k>3

As mentioned in the previous section, the p-ped packings shown there are projections of three dimensional space into the two dimensional plane. In this section, we will extend this idea to higher dimensions and explore the consequences of four and five dimensional spaces and their projection into the plane.

Interestingly enough, the familiar 3-to-2 dimensional projection of a hyper-cubic lattice can be extended[1] to higher dimensions simply by adding numbers. It is through this method for generating p-ped grids that allows for a much greater degree of freedom in their softening. The Penrose subdivisions from the last section were generated by recursive subdivision of prototiles but the same grids can be generated using the cut-and-project method[2] which slices a lattice in k-dimensions and projects the remaining faces into the plane. This method can be seen in action in Figures 53-56 by noticing that those grids appear to be three dimensional latices.

Now if the original lattice is four dimensional, we can perform the same sort of cutting and projecting to get a p-ped packing with six different rhombuses.

There are six different rhombuses because we can take any two of the initial four directions and because :

$$\binom{4}{2} = 6$$

In three dimensions, there are three orthogonal directions (x, y and z) and four dimensional space is defined by having four directions[3] Thus, in three dimensions we have three rhombuses and in five dimentions, we have ten:

$$\binom{3}{2} = 3, \binom{5}{2} = 10$$

Using this logic, we can generate p-ped packings using any number of dimensions as the projecting space but the results provided by high dimensional projections are limited.

1 This extension is 'by analogy' or not depending on whether one considered 4, 5, 6, etc. dimensions to be something real or strictly a construction. In mathematics, n-dimensional space is as real as 2 or 3.

2 de Bruijn

3 The fourth is often called w.

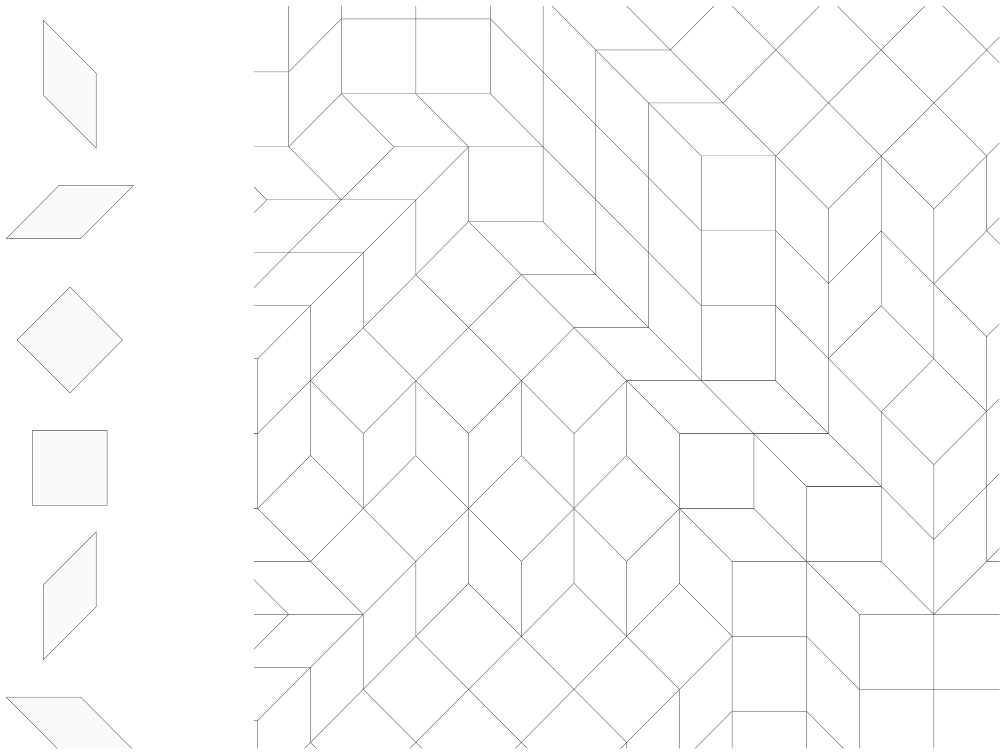


Figure 57. Instead of three types of cells, we can do the same with six.

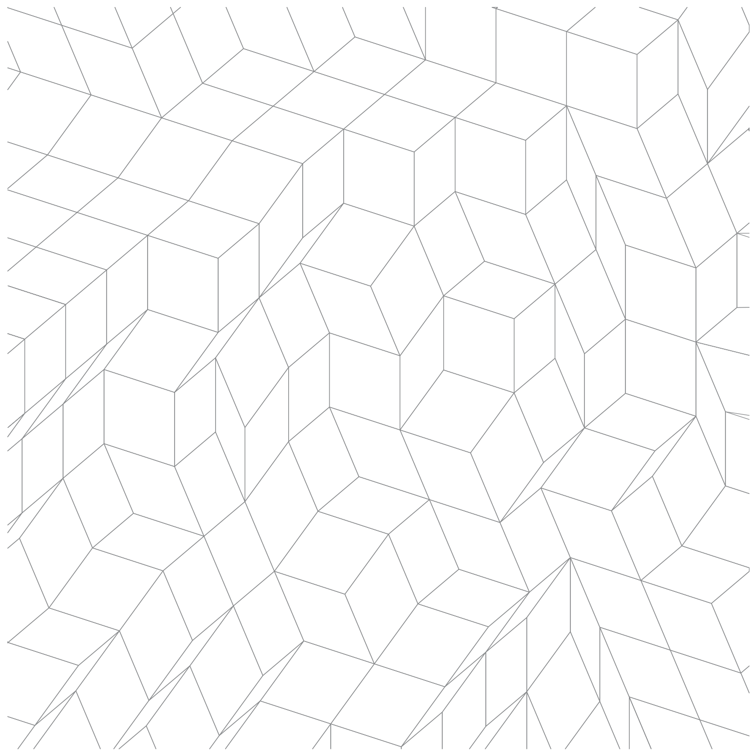


Figure 58. When we have five input vectors there are ten types of p-peds.

The Crawling Grid[1]

The two methods described so far to generate p-ped grids[2] are good for the mathematical production of aperiodic grids. However, generating worthwhile design grids requires a different technique all together. The manual propagation of such grids is extremely tedious and makes no sense as a design practice. The alternative is a customized algorithm for propagation which allows a user to vary parameters in real time as grids are generated to suite their needs.

Start by choosing a small number[3], k, of vectors in the two dimensional plane. Ideally, no two are at very small angles to one another because very narrow, acute p-peds will result. We then determine the possible combinations of any two of these vectors to produce a set of p-peds of which there are N where:

$$N = \binom{k}{2}$$

Then, starting with a single p-ped, attach adjacent p-peds growing outward and checking for collisions each time. This is the basic method employed for propagation but it does not fully describe the technique. At each step two choices must be made to completely determine the process. First, we need to decide which p-ped and on which face to place the next p-ped. Second we have to decide which p-ped to place there. Because we know the face on which we're placing, when we decide which to place, we only have to choose one new vector instead of two because one is already determined. Because we cannot pick the same vector twice, this leaves us with only k-1 choices instead of N for the next p-ped once we know where it is going. Once collisions are taken into account, there is very little choice left once a face for propagation has been chosen, making that choice the important one.
This process mimics the cut-and-project method although here we are first determining the projected vectors rather than the angle of cutting.[4] It allows a controlled propagation of the grid while maintaining organizational structure and allows for a great deal of local and global specificity.

1 The Python code for this algorithm along with mathematical infrastructure can be found in the appendix.

2 Penrose subdivision and the cut-and-projet method

3 Three to six seems ideal although there is nothing magial about this choice.

4 See de Bruijn for a detailed explanation of this.

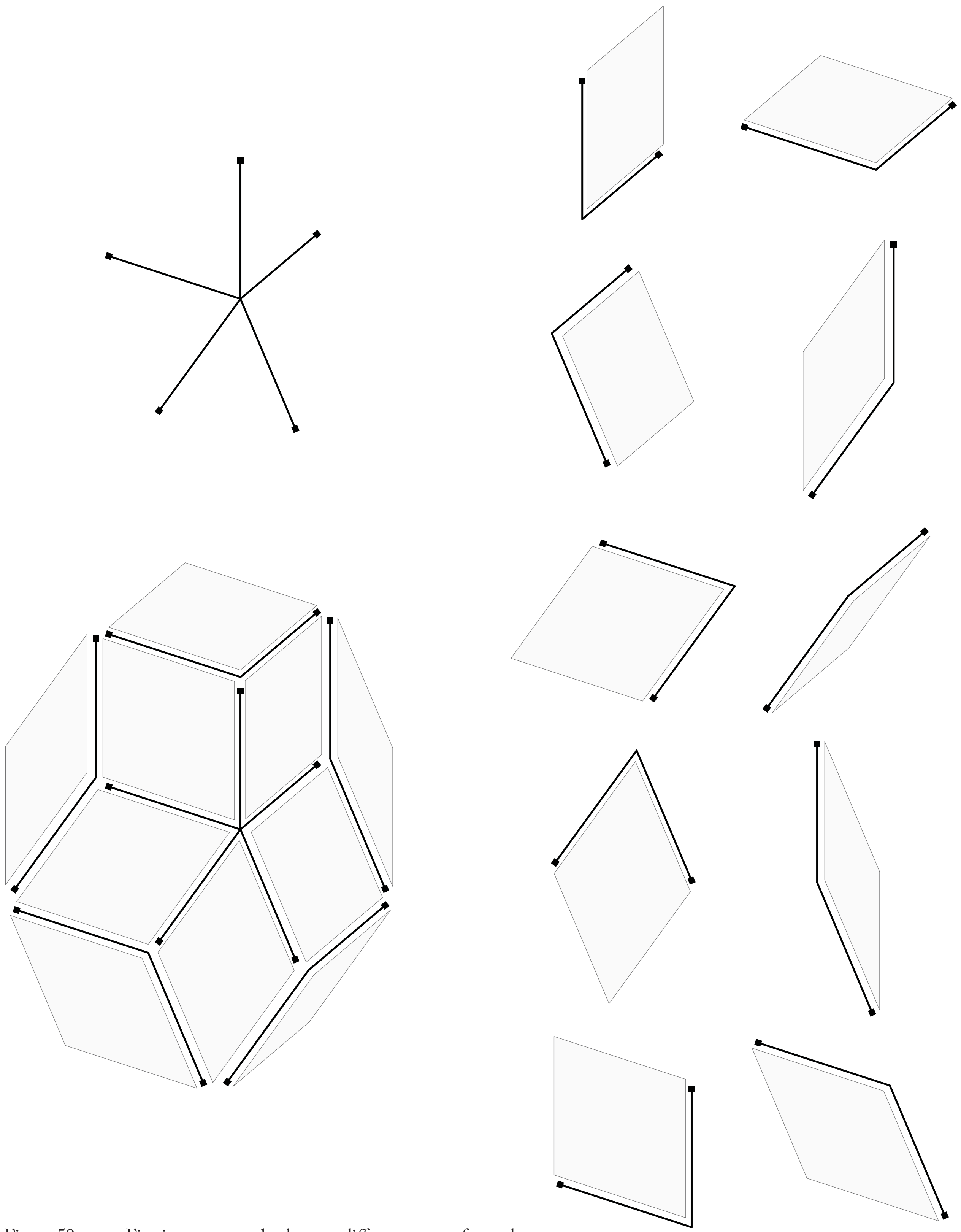


Figure 59. Five input vectors lead to ten different types of p-peds.

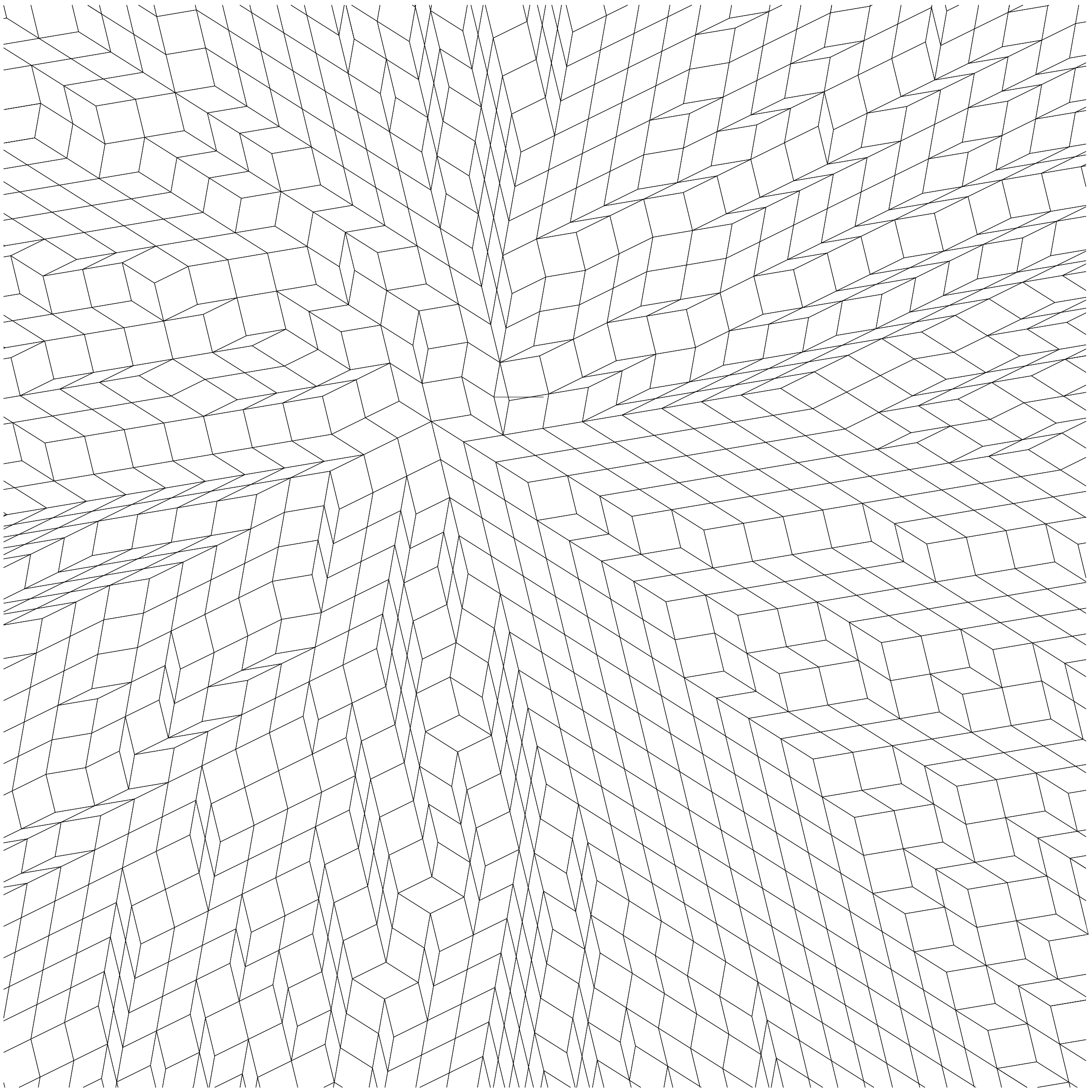


Figure 60. One result from the parallelepiped packing algorithm.

11| Conclusions

In Toto

The study of soft grids presented here has lead to two primary realizations. The first is the working definition of a soft grid in the form of the five attributes given in chapter 3. These, though presented at the beginning of the document, are actually a consequence of the research and not a presumption. The second is the typological classification of soft grids and the interrelation of these types to each other. Both of these results are presented as discrete, concrete concepts but in reality the attributes as well as the classes are imperfectly defined, overlapping and incomplete.

That being said, the goal of the research presented here was never a mathematical grade qualitative or quantitative definition of softness but rather an exploration and codification of an emerging set of ideas in architecture and design that, according to this author, are underexplored despite a growing profusion of use in the community. I hope that those who have read this thesis will have a foundation for a vocabulary and a basis for understanding soft girds as they come up in design work.

Self-contradiction

There is something in the name "Soft Grid" that speaks volumes about the nature of the project. It is the juxtaposition of the rigid and the flexible, the forgiving and the totalizing. Perhaps this dichotomy is the manifestation of a backlash against the Modernist notion organization compounded with the inherent need for systematization in design. Maybe it is a by-product of the democratizing power of computers and the irony that that produces with the systematic formalism of their architecture. Either way, this tension is a powerful component of the idea of the soft grid and has been a fruitful source of motivation.

Attributes

The five attributes that I have used as a metric for softness (Variability, Adaptivity, Asymmetry, Scalability and Entropy), have provided a means of discussing what makes a grid soft while simultaneously serving to define precisely what a soft grid is. Through the investigation of the grid types, the attributes have born out some of their weaknesses such as when they become non-applicable[1] or when they fail to recognize certain types of softness[2]. But they have also proved quite useful as a descriptive language for the grids that I've been studying.
The five descriptors are divisible into two groups, the chaotic and the ordering. The chaotic (entropy, variability and asymmetry) tend to reduce order in the grid, especially entropy. The ordering (adaptivity and scalability) tend to add more structure. It is a balance of these two that keeps the soft grids in the sweet spot between order and chaos, maintaining the self-contradiction.

Classification

The categories of grids that have been enumerated over the preceding chapters[3] are a rough outline of some of the soft grids circulating in the design world. Some of the specifications of these categories are more descriptive, such as local forcing and global systems, while some are generative such as Voronoi diagrams and subdivision schemes. The construction of a typology for a loosely defined set of things is a difficult task as the definition and the typological study tend to be reciprocal processes and each affects the other.

1 As in the case of local forcing.

2 P-ped packing.

3 Hard grids, High-symmetry, Local forcing, Global systems, Voronoi diagrams, Subdivision Schemes and P-ped packing.

Questions and future work

How can the attributes be refined and expanded to more completely define and describe soft grids?

The five attributes developed in this thesis are a good starting point in the description of a soft grid but they are somewhat redundant and fail to address certain aspects of softness that haven't been identified here. Also, they should to be subcategorized into groups like the chaotic and the ordered or the geometrical and the topological.

What other classifications and types of soft grids can be added to cover the field?

The classes of grids that have been enumerated here fall into two categories, the descriptive and the generative, and because of this, they do not make a coherent typology of soft grids. These two issues should be addressed separately and each should be expanded to include a wider array of classes.

Beyond the examples given, what are the applications and advantages of generating soft grids?

While the study of the use of soft grids in contemporary architecture are fruitful, more work needs to be done to understand the implications for design when the soft grids are brought to the design process as a precondition. This question could be answered in detail in the context of a class where students explicitly apply soft grids to design problems and scenarios.

How can these ideas be further developed into the study of a three-dimensional soft grid?

While this thesis mentions 3-D grids a few times, it does not address the field in general which is large and fruitful. Any one of the chapters of this thesis could be extended to three dimensions and expanded into a book. This is integrally related to the discussion on the implications for architecture.

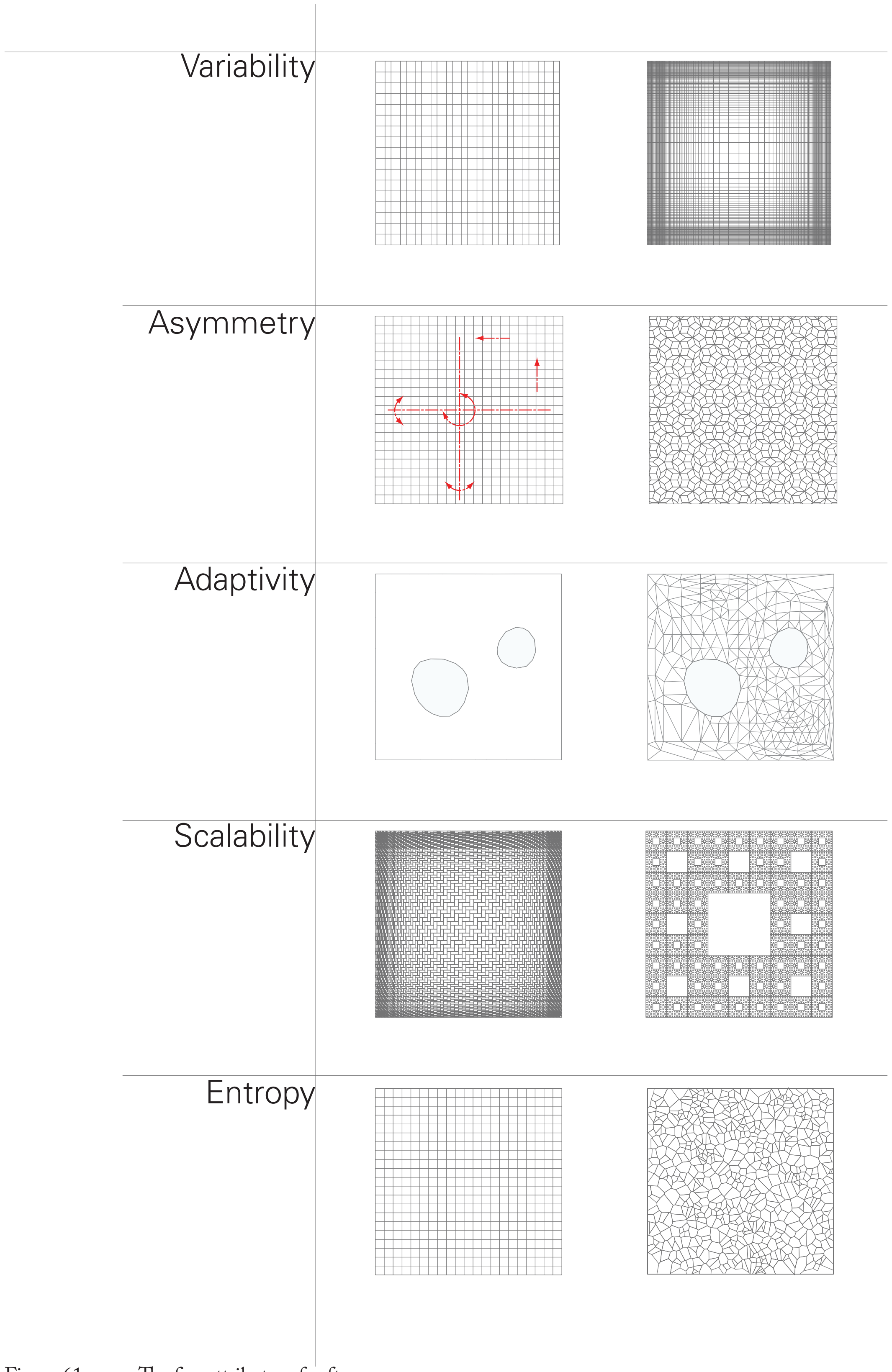


Figure 61. The five attributes of softness.

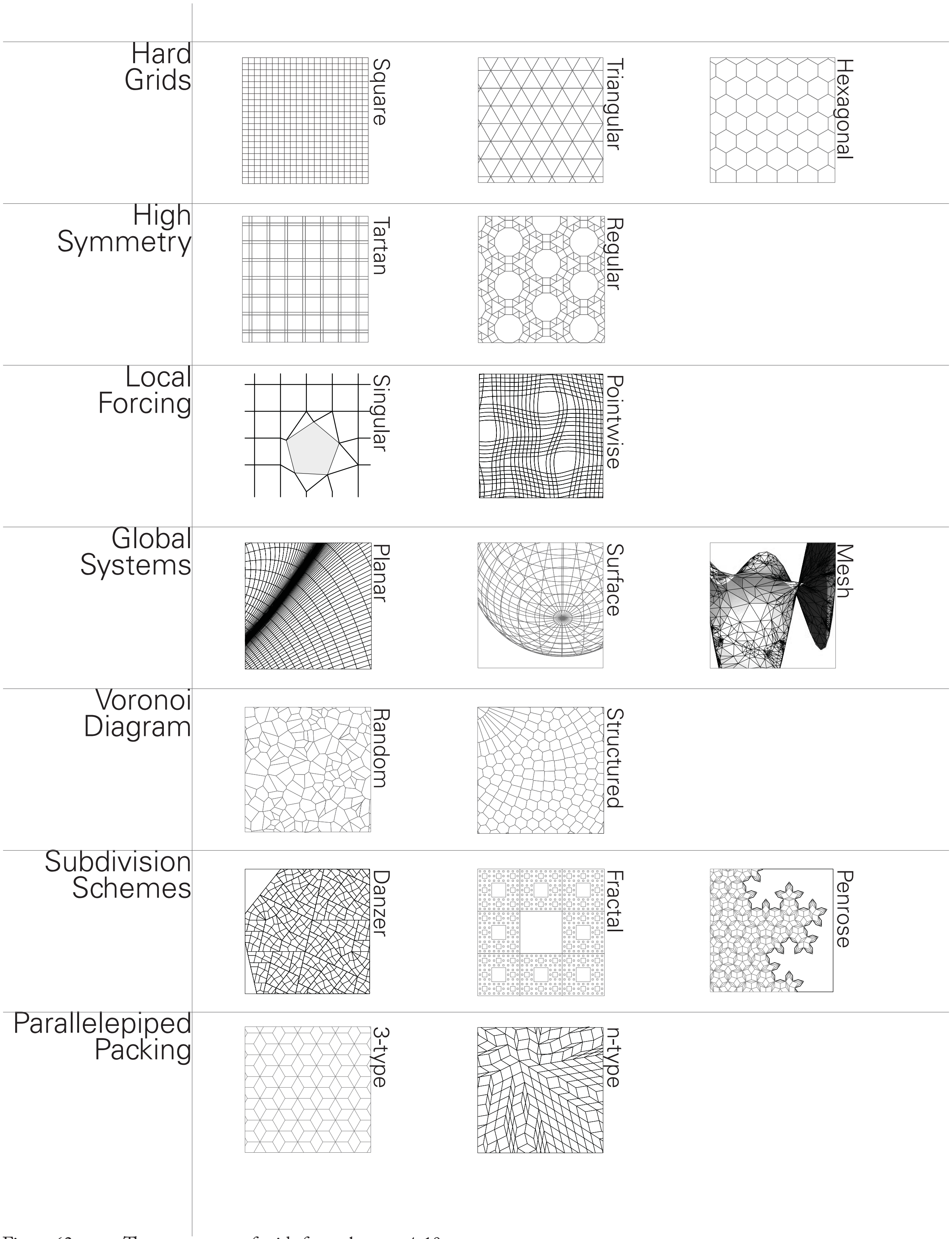


Figure 62. The seven types of grids from chapters 4-10

A| Appendix

Code

This section contains all of the code used to generate the figures in this document. Some code is nearly trivial while some is quite intricate and algorithmically difficult. Not all of what is presented here will stand alone because it was used in conjunction with numerical or geometrical data that cannot be included. In these cases, it is my hope that the interested reader will at least be able to extract the logic of the program even if it will not be executable.

In some cases, the code presented[1] was not the only tool used to generate the images such as is the case with the Voronoi diagrams which were generated using the algorithm packaged with the Grasshopper plug-in for Rhinoceros. The code for the p-ped packings is written in Python and uses PyGame's openGL pipeline for rendering.

1 generally Python, often in the context of McNeel's Rhinoceros

Figure 19 - Pointwise charges

```
import rhinoscriptsyntax as rs
import random
import math


def p_and_b_1():
    x_count = 50
    y_count = 50

    x_space = 1
    y_space = 1

    num_points = 50
    att_points = []
    att_weights = []

    falloff_distance = x_space*x_count / 5
    gravity_scale = 200

    rs.EnableRedraw(False)
    for i in range(num_points):
        x = x_count*x_space*random.random()
        y = y_count*y_space*random.random()
        att = rs.VectorAdd([x, y, 0], [0,0,0])
        weight = 2*random.random()-1
        att_points.append((att, weight))

    att_objs = rs.AddPoints([aw[0] for aw in att_points])
    for obj in att_objs:
        rs.ObjectLayer(obj, “Layer 01”)

    grid_points = {}
    for i in range(x_count+1):
        for j in range(y_count+1):
            grid_points[(i, j)] = rs.VectorAdd([i*x_space, j*y_space, 0],
[0,0,0])


    for i in range(x_count+1):
        for j in range(y_count+1):
            shift = rs.VectorAdd([0,0,0], [0,0,0])
            for att, weight in att_points:
                sub = rs.VectorSubtract(grid_points[(i, j)], att)
                uni = rs.VectorUnitize(sub)
                mag = rs.VectorLength(sub)
                shift = shift + (gravity_scale*weight/(mag + falloff_
distance)**2)*uni
            grid_points[(i,j)] = grid_points[(i,j)] + shift
            rs.AddPoint(grid_points[(i,j)])

    for i in range(x_count):
        for j in range(y_count+1):
            rs.AddLine(grid_points[(i,j)], grid_points[(i+1,j)])

    for i in range(x_count+1):
        for j in range(y_count):
            rs.AddLine(grid_points[(i,j)], grid_points[(i,j+1)])


    rs.EnableRedraw(True)

def p_and_b_2():
    x_count = 100
    y_count = 100

    x_space = 1
    y_space = 1

    spring_func = lambda p : 1 + (math.sin(0.5 * rs.VectorLength(rs.
VectorSubtract(p, [20,40,0]))))

    steps = 10
    k = 0.3

    rs.EnableRedraw(False)

    grid_points = {}
    for i in range(x_count+1):
        for j in range(y_count+1):
            grid_points[(i, j)] = rs.VectorAdd([i*x_space, j*y_space, 0],
[0,0,0])


    pt_objs = rs.AddPoints(grid_points)

    shifts={}
    for step in range(steps):
        for i in range(1, x_count):
            for j in range(1, y_count):
                #up
                shift = rs.VectorAdd([0,0,0], [0,0,0])
                midpoint = (grid_points[(i, j)] + grid_points[(i, j+1)])/2
                spring_val = y_space * (spring_func(midpoint))
                sub = grid_points[(i, j)] - grid_points[(i, j+1)]
                uni = rs.VectorUnitize(sub)
                d = spring_val - rs.VectorLength(sub)
                shift_up = k * d * uni

                #down
                midpoint = (grid_points[(i, j)] + grid_points[(i, j-1)])/2
                spring_val = y_space * (spring_func(midpoint))
                sub = grid_points[(i, j)] - grid_points[(i, j-1)]
                uni = rs.VectorUnitize(sub)
                d = spring_val - rs.VectorLength(sub)
                shift_down = k * d * uni


                #right
                shift = rs.VectorAdd([0,0,0], [0,0,0])
                midpoint = (grid_points[(i, j)] + grid_points[(i+1, j)])/2
                spring_val = x_space * (spring_func(midpoint))
                sub = grid_points[(i, j)] - grid_points[(i+1, j)]
                uni = rs.VectorUnitize(sub)
                d = spring_val - rs.VectorLength(sub)
                shift_right = k * d * uni

                #left
                shift = rs.VectorAdd([0,0,0], [0,0,0])
                midpoint = (grid_points[(i, j)] + grid_points[(i-1, j)])/2
                spring_val = x_space * (spring_func(midpoint))
                sub = grid_points[(i, j)] - grid_points[(i-1, j)]
                uni = rs.VectorUnitize(sub)
                d = spring_val - rs.VectorLength(sub)
                shift_left = k * d * uni

                shifts[(i, j)] = shift_up +shift_down +shift_left +shift_
right

        for i in range(1, x_count):
            for j in range(1, y_count):
                grid_points[(i,j)] = grid_points[(i,j)]+shifts[(i, j)]
        rs.EnableRedraw(True)
        rs.DeleteObjects(pt_objs)
        pt_objs = rs.AddPoints(grid_points.values())
        rs.EnableRedraw(False)

    for i in range(x_count):
        for j in range(y_count+1):
            rs.AddLine(grid_points[(i,j)], grid_points[(i+1,j)])

    for i in range(x_count+1):
        for j in range(y_count):
            rs.AddLine(grid_points[(i,j)], grid_points[(i,j+1)])

    rs.EnableRedraw(True)



p_and_b_2()
```

Figure 20 - Laminar flow

```
import rhinoscriptsyntax as rs
import random
import math

def acot(x):
    if x<0:
        return -math.atan(x)-math.pi/2
    else:
        return -math.atan(x)+math.pi/2

#pts = [[8* random.random() + 1, 8* random.random() + 1, 0] for i in
range(8)]
pts = rs.GetPoints(“Choose obstacles”)
obstacles = [rs.VectorAdd(pt, [0,0,0]) for pt in pts]
weights = [(1 + random.random())/50 for i in obstacles]
weights = [.2 for i in obstacles]
rs.AddPoints(obstacles)

stretch = .5
width = 1

rs.EnableRedraw(False)

for j in range(61):
    ystart = 10*j/60
    crv_pts = []
    for i in range(1001):
        x = 10*i/1000
        y = ystart
        for ob_index in range(len(obstacles)):
            yo = obstacles[ob_index][1]
            xo = obstacles[ob_index][0]
            w = weights[ob_index]
            y += w * acot(stretch*(ystart-yo)) / (((x-xo))**2+width)
        crv_pts.append((x,y,0))

    rs.AddPolyline(crv_pts)

for i in range(61):
    xstart = 10*i/60
    crv_pts = []
    for j in range(1001):
        y = 10*j/1000
        x = xstart
        for ob_index in range(len(obstacles)):
            yo = obstacles[ob_index][1]
            xo = obstacles[ob_index][0]
            w = weights[ob_index]
            x += w *  acot(stretch*(xstart-xo)) / ((y-yo)**2+width)
        crv_pts.append((x,y,0))

    rs.AddPolyline(crv_pts)

rs.EnableRedraw(True)
```

Figure 23 - Continuous plane distortion

```
import rhinoscriptsyntax as rs
import math

points = {}
lines = []
objPts = []

iMax = 50
jMax = 50

def x(i, j):
    return 2*math.pi*(i/iMax) + 0.9*math.sin(2*math.pi*i/iMax)*math.
sin(2*math.pi*j/jMax)

def y(i, j):
    return 2*math.pi*(j/jMax) + -0.3*math.sin(2*math.pi*j/jMax)*math.
sin(2*math.pi*i/iMax)

rs.EnableRedraw(False)
for i in range(iMax+1):
    for j in range(jMax+1):
        point3d = [x(i,j), y(i,j), 0]
        points[(i,j)] = rs.VectorAdd([0,0,0], point3d)
        objPts.append(rs.AddPoint(points[(i,j)]))

xmin = points[(0,0)].X
xmax = points[(0,0)].X
ymin = points[(0,0)].Y
ymax = points[(0,0)].Y
for pt in points.values():
    if pt.X<xmin:
        xmin = pt.X
    if pt.X>xmax:
        xmax = pt.X
    if pt.Y<ymin:
        ymin = pt.Y
    if pt.Y>ymax:
        ymax = pt.Y

for i in range(iMax):
    for j in range(1,jMax):
        lines.append(rs.AddLine(points[(i,j)], points[(i+1,j)]))
for i in range(1,iMax):
    for j in range(jMax):
        lines.append(rs.AddLine(points[(i,j)], points[(i,j+1 )]))

xScale = 10/(xmax - xmin)
yScale = 10/(ymax - ymin)

rs.ScaleObjects(objPts, [xmin, ymin, 0], [xScale, yScale, 1])
rs.ScaleObjects(lines, [xmin, ymin, 0], [xScale, yScale, 1])

rs.EnableRedraw(True)
```

Figure 37 - Voronoi degeneration

```
import rhinoscriptsyntax as rs
from random import *

pts = rs.GetObjects(“pts”)
for pt in pts:
    rs.MoveObject(pt, [.03*(random()-.5), .03*(random()-.5), 0])
```

Figures 58 & 60 - Penrose Subdivision

```
import rhinoscriptsyntax as rs
import math

rules = dict([[“P1 “+ x, “P1 “+ x + x] for x in [“A”, “B”, “C”, “D”, “E”, 
“F”]])
steps = 3
tt = ((1+math.sqrt(5))/2)**2

initial_block = rs.GetObjects(“select initial”, rs.filter.instance)

blocks = initial_block

def subdivide_group(all_blocks):
    new_blocks = []
    for block in all_blocks:
        type = rs.BlockInstanceName(block)
        xform = rs.BlockInstanceXform(block)
        new_block = rs.InsertBlock2(rules[type], xform)
        new_blocks.append(new_block)
    rs.DeleteObjects(all_blocks)
    return (rs.ScaleObjects(new_blocks, [0,0,0], [tt, tt, tt]))

step = 1

while step:
    rs.EnableRedraw(False)
    news = subdivide_group(blocks)
    rs.SelectObjects(news)
    blocks = rs.Command(“Explode”)
    blocks = rs.GetObjects(“Enter to repeat”, preselect=True)
    rs.EnableRedraw(True)
    step = rs.GetInteger(“0 to stop, 1 to continue”)
for i in range(30):
    for j in range(int(i) + 10):
          rs.AddPoint([i, (j)/(int(i) + 9), 0])
```

The remaining code is for the p-ped packing algorithm, presented by file name.

aperiodic_2D_propagate.py

```
import math
import random
from vector import *

class Pped:
    max_strikes = 10
    def __init__(self, corner, v1, v2, parent, parent_spot, prop):
        random_neighbor_order = False
        self.prop = prop
        self.v1 = v1
        self.v2 = v2
        n=self.prop.n
        greyval = (1 - float(v1*v2)/(n-1)**2)*.7
        self.color = [greyval, greyval, greyval]
        #strikes is a counter to identify troublesome pped’s
        self.strikes = 0
        self.lats = [corner,
                    lat_bump(corner, [v1]),
                    lat_bump(corner, [v1, v2]),
                    lat_bump(corner, [v2])]
        self.verts = map(prop.lat_to_coords, self.lats)

        self.edges = [[self.lats[0], self.lats[1]],
                     [self.lats[1], self.lats[2]],
                     [self.lats[2], self.lats[3]],
                     [self.lats[3], self.lats[0]]]

        self.center = (self.verts[1] + self.verts[3])/2
        self.mag_square = self.center.dot(self.center)
        neigh = []
        for i in range(4):
            n1 = range(prop.n)
            if i in (0,2):
                n1.remove(v1)
            else:
                n1.remove(v2)
            neigh.append(n1)

        self.parent = parent
        self.neighbors = []
        for n1 in neigh:
            self.neighbors.append([randomize(n1), None])
        if parent:
            self.parent_spot = parent_spot
            self.neighbors[parent_spot] = [[], parent]

        if random_neighbor_order:
            self.neighbor_order = randomize(range(len(self.neighbors)))
        else:
            self.neighbor_order = range(len(self.neighbors))
    def relocate(self):
        self.verts = map(self.prop.lat_to_coords, self.lats)
        self.center = (self.verts[1] + self.verts[3])/2


    def intersects(self, other):
        #self.prop.renderer.draw(red = [self, other])
        if (self.center-other.center).mag() > self.prop.max_dist:
            return False
        if other.is_in(self.verts[0]):
            return True
        if self.is_in(other.verts[0]):
            return True
        for e0 in self.edges:
            for e1 in other.edges:
                if self.prop.lines_intersect(e0, e1):
                    return True
        #raw_input(“no touches”)
        return False

    def square_to_origin(self):
        return dot(self.base, self.base)

    def is_in(self, pt):
        if (self.verts[1]-self.verts[0]).dot(pt-self.verts[0]) > 0:
            return False
        if (self.verts[2]-self.verts[1]).dot(pt-self.verts[1]) > 0:
            return False
        if (self.verts[3]-self.verts[2]).dot(pt-self.verts[2]) > 0:
            return False
        if (self.verts[0]-self.verts[3]).dot(pt-self.verts[3]) > 0:
            return False
        return True

    def remove_neighbor(self, neigh):
        for n in self.neighbors:
            if n[1]:
                self.prop.add_to_queue(n[1])
            if n[1]==neigh:
                n[1]=None

    def o_2r_plus_current(self):
        c = self.center
        cc = self.prop.current_cell.center
        r =  c.mag()
        r_c = (c-cc).mag()
        return 2*r + r_c

    def o_mouse(self):
        x = self.prop.renderer.plane_x
        y = self.prop.renderer.plane_y
        mouse = Vector([x,y,0])
        return (mouse - self.center).mag() + 10*self.center.mag()

def lat_bump(lat, indices):
    res = list(lat)
    for i in indices:
        res[i] += 1
    return tuple(res)

def lat_thump(lat, indices):
    res = list(lat)
    for i in indices:
        res[i] -= 1
    return tuple(res)

def ccw(A,B,C):
      return (C[1]-A[1])*(B[0]-A[0]) > (B[1]-A[1])*(C[0]-A[0])

def intersect(A,B,C,D):
    return ccw(A,C,D) != ccw(B,C,D) and ccw(A,B,C) != ccw(A,B,D)

def randomize(lst):
    n = len(lst)
    if n == 1:
        return lst
    r = random.randint(0, n-1)
    elem = lst[r]
    del lst[r]
    return [elem] + randomize(lst)

class Aperiodic_2D_propagate:
    def __init__(self, vects, renderer):
        self.renderer = renderer
        self.vects = vects
        self.n = len(self.vects)
        self.tree = []
        self.queue = []
        for v in self.vects:
            if v[0]<0 : v = -v
        vects.sort(key = lambda v : v[1] / v[0])
        max_dist=0
        for i in range(self.n):
            for j in range(i+1):
                dif = self.vects[i]-self.vects[j]
                if dif.mag() > max_dist:
                        max_dist =  dif.mag()
        self.max_dist = max_dist + 0.0001

        i = random.randint(0, self.n - 1)
        j = random.randint(0, self.n - 2)
        if j >= i:
            j += 1

        i, j = sorted([i,j])     #this just makes i the smaller of the two

        self.root = Pped(tuple([0] * self.n), i, j, None, None, self)
        self.queue = [self.root]
        self.cells = [self.root]
        self.current_cell = None
        self.orderer = Pped.o_2r_plus_current
        self.order_index=1
        self.orderer = Pped.o_2r_plus_current

    def nudge_vects(self):
        for i in range(len(self.vects)):
            self.vects[i] += Vector([0.05*(random.random()-.5), 0.05*(random.
random()-.5), 0])
        for cell in self.cells:
            cell.relocate()

    def nudge_one_vect(self, i, axis, direction):
        “””axis ==0 -> x, axis==1 -> y
        direction = +1 or -1”””
        if axis==0:
            self.vects[i] += Vector([direction*0.05,0, 0])
        elif axis==1:
            self.vects[i] += Vector([0, direction*0.05, 0])
        for cell in self.cells:
            cell.relocate()

    def lines_intersect(self, e0, e1):
        if e0[0] in e1 or e0[1] in e1:
            return False
        return intersect(self.lat_to_coords(e0[0]),
                self.lat_to_coords(e0[1]),
                self.lat_to_coords(e1[0]),
                self.lat_to_coords(e1[1]))

    def kill_cell(self, c):
        if c.parent:
            self.add_to_queue(c.parent)

        for n in c.neighbors:
            if n[1]:
                self.add_to_queue(n[1])
                if n[1].parent != c:
                    n[1].remove_neighbor(c)
                else:
                    self.kill_cell(n[1])

        while c in self.cells:
            self.cells.remove(c)
        while c in self.queue:
            self.queue.remove(c)

        for n in c.neighbors:
            if n[1]:
               self.add_to_queue(n[1])


    def lat_to_coords(self, lat):
        res = Vector([0,0,0])
        terms =  [l*v for l, v in zip(lat, self.vects)]
        for t in terms:
            res += t
        return res

    def next_ordering(self):
        self.order_index = (self.order_index + 1) % 2
        if self.order_index == 0:
            self.orderer = Pped.o_2r_plus_current
        if self.order_index == 1:
            self.orderer = Pped.o_mouse

    def add_to_queue(self, c):
        if not (None in [n[1] for n in c.neighbors]):
            return
        if c in self.queue:
            return
        self.queue.append(c)
        self.queue.sort(key = self.orderer)

    def try_neighbor(self, i, v):
        v_try = self.current_cell.neighbors[i][0].pop()
        if i == 0:
            v = self.current_cell.v1
            if v < v_try:
                lat_try = lat_thump(self.current_cell.lats[0], [v_try])
                v1, v2 = sorted([v, v_try])
                cell_try = Pped(lat_try, v1, v2, self.current_cell, 2, self)
            else:
                lat_try = self.current_cell.lats[0]
                v1, v2 =sorted([v, v_try])
                cell_try = Pped(lat_try, v1, v2,  self.current_cell, 3, self)
        if i == 1:
            v = self.current_cell.v2
            if v < v_try:
                lat_try = lat_thump(self.current_cell.lats[1], [v_try])
                v1, v2 =sorted([v, v_try])
                cell_try = Pped(lat_try, v1, v2, self.current_cell, 2, self)
            else:
                lat_try = self.current_cell.lats[1]
                v1, v2 =sorted([v, v_try])
                cell_try = Pped(lat_try, v1, v2, self.current_cell, 3, self)
        if i == 2:
            v = self.current_cell.v1
            if v < v_try:
                lat_try = self.current_cell.lats[3]
                v1, v2 =sorted([v, v_try])
                cell_try = Pped(lat_try, v1, v2, self.current_cell, 0, self)
            else:
                lat_try = lat_thump(self.current_cell.lats[3], [v_try])
                v1, v2 =sorted([v, v_try])
                cell_try = Pped(lat_try, v1, v2, self.current_cell, 1, self)
        if i == 3:
            v = self.current_cell.v2
            if v < v_try:
                lat_try = self.current_cell.lats[0]
                v1, v2 =sorted([v, v_try])
                cell_try = Pped(lat_try, v1, v2, self.current_cell, 0, self)
            else:
                lat_try = lat_thump(self.current_cell.lats[0], [v_try])
                v1, v2 =sorted([v, v_try])
                cell_try = Pped(lat_try, v1, v2, self.current_cell, 1, self)

        self.cells.append(cell_try)
        self.add_to_queue(cell_try)

        shapes_intersect = False
        for enum_other in self.cells:
            other = enum_other
            if other != cell_try:
                if cell_try.intersects(other):
                    self.kill_cell(cell_try)
                    shapes_intersect = True
                    return True

        if not shapes_intersect:
            #Add the try_cell to the tree
            self.current_cell.neighbors[i][1] = cell_try

            #check for adjacencies
            for other in self.cells:
                for my_edge in cell_try.edges:
                    for other_edge in other.edges:
                        if my_edge[0]==other_edge[1] and my_edge[1]==other_
edge[0]:
                            my_index = cell_try.edges.index(my_edge)
                            other_index = other.edges.index(other_edge)
                            cell_try.neighbors[my_index][1] = other
                            other.neighbors[other_index][1] = cell_try
                            #midpoint = (self.lat_to_coords(my_edge[0]) +
self.lat_to_coords(my_edge[1]))/2
            return False

    def queue_step(self):
        self.current_cell = self.queue[0]

        for j in range(len(self.current_cell.neighbors)):
            neighbor_i = self.current_cell.neighbor_order[j]
            if self.current_cell.neighbors[neighbor_i][1] == None:
                if neighbor_i in (0,2):
                    v_adj = self.current_cell.v1
                else:
                    v_adj = self.current_cell.v2

                shapes_intersect = True
                while self.current_cell.neighbors[neighbor_i][0]:
                    shapes_intersect = self.try_neighbor(neighbor_i, v_adj)

                    if not shapes_intersect:
                        break
                if shapes_intersect:
                    self.kill_cell(self.current_cell)
                    break

        if self.current_cell in self.queue:
            self.queue.remove(self.current_cell)
```

render.py

```
SCREEN_SIZE = (1030, 640)

from math import radians

from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *
from OpenGL.GLE import *
from  vector import *
import pygame
from pygame.locals import *
import random
from aperiodic_2D_propagate import *
import time

class Renderer:
    def init(self):
        glEnable(GL_DEPTH_TEST)

        self.zpos = 5
        self.tx, self.ty  = (0,0)

        glShadeModel(GL_FLAT)
        glClearColor(1.0, 1.0, 1.0, 0.0)

        glEnable(GL_COLOR_MATERIAL)

        glEnable(GL_LIGHTING)
        glEnable(GL_LIGHT0)
        glLight(GL_LIGHT0, GL_POSITION,  (0, 0, -1, 0))

    def resize(self, width, height):
        glViewport(0, 0, width, height)
        glMatrixMode(GL_PROJECTION)
        glLoadIdentity()

        gluPerspective(45.0, float(width)/height, .1, 1000.)
        glMatrixMode(GL_MODELVIEW)
        glLoadIdentity()
        gluLookAt(0,0,5, 0,0,0, 0,1,0)

    def export(self):
        t = time.time()
        f = open('./output/' + str(t) +'.stl', 'w')
        for cell in self.propagator.cells:
            f.write('C' + str(cell.center) + 'V' + str(cell.verts) + "\n")
        f.close()


    def draw(self, red =[], draw_tree=False, draw_dual=True, draw_
cells=False):
        glLoadIdentity()
        glTranslate(self.zpos*self.tx/500., self.zpos*self.ty/500., - self.
zpos)
```

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
#glLight(GL_LIGHT0, GL_POSITION,  (0, 0, 1, 0))

#mouse tracker
glPushMatrix()
glTranslatef(self.plane_x, self.plane_y, 0)
glutSolidSphere(0.1,32,32)
glPopMatrix()

p = self.propagator
if draw_cells:
    for cell in p.cells:
        if False: #cell==p.root:
            glLineWidth(3)
            glColor3f(0.0, 0.0, 1.0)
            glBegin(GL_LINE_LOOP)
            for v in cell.verts:
                glVertex3f(v[0], v[1], 0)
            glEnd()

            glColor3f(0.0, 0.0, 0.0)
            glBegin(GL_QUADS)
            for v in cell.verts:
                glVertex3f(v[0], v[1], 0)
            glEnd()

        else:
            if cell in red:
                glColor3f(1.0, 0.0, 0.0)
                glBegin(GL_QUADS)
                for v in cell.verts:
                    glVertex3f(v[0], v[1], 0)
                glEnd()
            else:
                glLineWidth(1)
                glColor3f(0.0, 0.0, 0.0)
                glBegin(GL_LINE_LOOP)
                for v in cell.verts:
                    glVertex3f(v[0], v[1], 0)
                glEnd()

                if cell in p.queue:
                    glColor3f(0.3, 1.0, 0.3)
                else:
                    glColor3f(*cell.color)
                glBegin(GL_QUADS)
                for v in cell.verts:
                    glVertex3f(v[0], v[1], 0)
                glEnd()

if draw_tree:
    glLineWidth(2)
    glColor3f(1.0, 0.0, 0.0)
    for cell in p.cells:
        if cell.parent:
            glBegin(GL_LINES)
            glVertex3f(cell.center[0], cell.center[1], .001)
            glVertex3f(cell.parent.center[0], cell.parent.center[1],
.001)
                    glEnd()

        if draw_dual:
            glLineWidth(2)
            glColor3f(0.0, 0.0, 1.0)
            for cell in p.cells:
                for n in cell.neighbors:
                    if n[1]:
                        glBegin(GL_LINES)
                        glVertex3f(cell.center[0], cell.center[1], .001)
                        glVertex3f(n[1].center[0], n[1].center[1], .001)
                        glEnd()


        pygame.display.flip()

    def run(self):
        glutInit()
        pygame.init()
        screen = pygame.display.set_mode(SCREEN_SIZE,
HWSURFACE|OPENGL|DOUBLEBUF)

        self.init()
        self.resize(*SCREEN_SIZE)

        clock = pygame.time.Clock()
        glMaterial(GL_FRONT, GL_AMBIENT, (0.1, 0.1, 0.1, 1.0))
        glMaterial(GL_FRONT, GL_DIFFUSE, (1.0, 1.0, 1.0, 1.0))

        vects = [Vector([random.random(),random.random(),0]) for i in
range(4)]
        num_vects = 5
        vects = []
        for i in range(num_vects):
            r = 1+random.random()
            theta = -math.pi/2 + i*math.pi/num_vects + random.random()*math.
pi*0.3/num_vects
            vects.append(Vector([r*math.cos(theta), r*math.sin(theta), 0]))
        self.propagator = Aperiodic_2D_propagate(vects, self)
        self.propagator.next_ordering()

        going = True
        filming = False
        do_draw_cells = False
        do_draw_tree = False
        do_draw_dual = False
        random_nudging = False
        move = False
        nudge_axis = frame_no = 0
        nudge_index = 0

        count =0
        while True:
            #Calculate mouse possition
            mouse_x, mouse_y = pygame.mouse.get_pos()
            model = glGetDoublev(GL_MODELVIEW_MATRIX)
            proj  = glGetDoublev(GL_PROJECTION_MATRIX)
            view  = glGetIntegerv(GL_VIEWPORT)
            mouse_z = glReadPixels(mouse_x, mouse_y, 1, 1, GL_DEPTH_COMPO-
NENT, GL_FLOAT);

            self.plane_x = self.tx + (mouse_x-SCREEN_SIZE[0]/2.0) * self.
zpos/SCREEN_SIZE[0]
            self.plane_y = self.ty - (mouse_y-SCREEN_SIZE[1]/2.0) * self.
zpos/SCREEN_SIZE[1]

            for e in pygame.event.get():
                if e.type == QUIT:
                    sys.exit()
                elif e.type == KEYDOWN and e.key == K_ESCAPE:
                    sys.exit()
                elif e.type == KEYUP and e.key == K_ESCAPE:
                    return
                elif e.type == KEYUP and e.key == K_SPACE:
                    going = not going
                elif e.type == KEYUP and e.key == K_e:
                    self.export()
                elif e.type == KEYUP and e.key == K_f:
                    filming = not filming

                #Drawing
                elif e.type == KEYUP and e.key == K_c:
                    do_draw_cells = not do_draw_cells
                elif e.type == KEYUP and e.key == K_t:
                    do_draw_tree = not do_draw_tree
                elif e.type == KEYUP and e.key == K_d:
                    do_draw_dual = not do_draw_dual

                elif e.type == KEYUP and e.key == K_o:
                    self.propagator.next_ordering()

                #Nudging of vectors
                elif e.type == KEYUP and e.key == K_r:
                    random_nudging = not random_nudging
                elif e.type == KEYUP and e.key == K_n:
                    nudge_index = (nudge_index + 1) % num_vects
                elif e.type == KEYUP and e.key == K_RIGHT:
                    self.propagator.nudge_one_vect(nudge_index, 0, 1)
                elif e.type == KEYUP and e.key == K_LEFT:
                    self.propagator.nudge_one_vect(nudge_index, 0, -1)
                elif e.type == KEYUP and e.key == K_UP:
                    self.propagator.nudge_one_vect(nudge_index, 1, 1)
                elif e.type == KEYUP and e.key == K_DOWN:
                    self.propagator.nudge_one_vect(nudge_index, 1, -1)

                elif e.type == MOUSEBUTTONDOWN:
                    if e.button == 4: self.zpos = max(1, self.zpos-1)
                    elif e.button == 5: self.zpos += 1
                    elif e.button == 1: rotate = True
                    elif e.button == 3: move = True
                elif e.type == MOUSEBUTTONUP:
                    if e.button == 1: rotate = False
                    elif e.button == 3: move = False
                elif e.type == MOUSEMOTION:
                    i, j = e.rel
                if move:
                    self.tx += i
                    self.ty -= j

            if filming:
                print “Frame Number”, frame_no
                frame_no += 1
                pygame.image.save(screen, “png/”+str(count).zfill(5) +’.png’)
            if going:
                pygame.time.wait(50)
                self.propagator.queue_step()

                #print “Cells: “, len(self.propagator.cells)
                if random_nudging:
                    self.propagator.nudge_vects()

            self.draw(draw_cells = do_draw_cells,
                      draw_tree = do_draw_tree,
                      draw_dual = do_draw_dual,
                      red=[self.propagator.current_cell])
            count += 1

r = Renderer()
r.run()
```

vector.py

```
import math
class Vector:
    def __init__(self, coords):
        self.coords = coords

    def __repr__(self):
        return self.coords.__repr__()

    def __add__(self, other):
        return Vector([a+b for a,b in zip(self.coords, other.coords)])

    def __sub__(self, other):
        return Vector([a-b for a,b in zip(self.coords, other.coords)])

    def __mul__(self, scale):
        return Vector([scale*a for a in self.coords])

    def __div__(self, scale):
        return Vector([a/scale for a in self.coords])

    def __rmul__(self, scale):
        return self*scale

    def __getitem__(self, index):
        return self.coords[index]

    def dot(self, other):
        return sum([a*b for a,b in zip(self.coords, other.coords)])

    def mag(self):
        return math.sqrt(self.dot(self))
```

B| Bibliography

Aranda, Benjamin, and Lasch, Chris. Pamphlet Architecture 27: Tooling. 1st ed. Princeton Architectural Press, 2005.

de Berg, Mark, M. van Krefeld, M. Overmars, and O. Schwarzkopf. Computational Geometry: Algorithms and Applications, Second Edition. 2nd ed. Springer, 2000.

de Bruijn, N G. "Algebraic theory of Penrose's nonperiodic tilings of the plane. I, II." Nederl. Akad. Wetensch. Indag. Math. 43, no. 1 (1981): 39-66.

Caragonne, Alexander. The Texas Rangers: Notes from the Architectural Underground. The MIT Press, 1995.

Charalampous, Charis. "Responsive spatial growth of the Danzer packing". London, England: Bartlett School of Graduate Studies University College London, 2007.

Dimitrie, Stefanescu. "f* Voronoi." Improved, October 28, 2010. http://improved.ro/blog/2010/10/f-voronoi/.

Durand, Jean-Nicholas-Louis. Precis of the Lectures on Architecture: With Graphic Portion of the Lectures on Architecture. 1st ed. Getty Research Institute, 2000.

Dyer, Ramsay, Hao Zhang, and Torsten Möller. "Delaunay mesh construction." In Proceedings of the fifth Eurographics symposium on Geometry processing, 273–282. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2007. http://portal.acm.org/citation.cfm?id=1281991.1282027.

Evans, Robin. The Projective Cast: Architecture and Its Three Geometries. The MIT Press, 1995.

Giedion, Siegfried. Space, Time and Architecture: The Growth of a New Tradition, Fifth Revised and Enlarged Edition. Harvard University Press, 1967.

Grünbaum, Branko, and G. C. Shephard. TILINGS AND PATTERNS. 1st ed. W.H. FREEMAN, 1990.

Higgins, Hannah B. The Grid Book. The MIT Press, 2009.

Hughes, Thomas J. R. The Finite Element Method: Linear Static and Dynamic Finite Element Analysis. Dover Publications, 2000.

Janot, C. Quasicrystals: A Primer. 2nd ed. Oxford University Press, USA, 1997.

Kobayashi, Kei, and Kokichi Sugihara. "Crystal Voronoi diagram and its applications." Future Generation Computer Systems 18, no. 5 (April 2002): 681-692.

Kramer, Peter. "Gateways towards quasicrystals." 1101.0061 (December 30, 2010). http://arxiv.org/abs/1101.0061.

Ledieu, J., R. McGrath, R. D. Diehl, T. A. Lograsso, D. W. Delaney, Z. Papadopolos, and G. Kasner. "Tiling of the fivefold surface of Al70Pd21Mn9." Surface Science 492, no. 3 (October 20, 2001): L729-L734.
Lim, Jin-young, and Kengo Kuma. "New types of geometry." Space, no. 479 (October 2007).

Love, Timothy. "Kit-of-Parts Conceptualism." Harvard Design Magazine 19 (Fall / Winter 2004 2003).

Lu, Peter J., and Paul J. Steinhardt. "Decagonal and Quasi-Crystalline Tilings in Medieval Islamic Architecture." Science 315, no. 5815 (February 23, 2007): 1106 -1110.

Mandelbrot, Benoit B. The Fractal Geometry of Nature. W. H. Freeman, 1982.

March, Lionel, and Philip Steadman. The Geometry of Environment: An Introduction to Spatial Organization in Design. The MIT Press, 1974.

March, Lionell. "A Class of Grids." Environment and Planning B 8, no. 3 (1981): 325-332.

Minsky, Marvin. The Society of Mind. Pages Bent. Simon & Schuster, 1988.

Mitchell, William J. The Logic of Architecture: Design, Computation, and Cognition. The MIT Press, 1990.

Nicolai, Carsten. Grid Index. Die Gestalten Verlag, 2009.

Okabe, Atsuyuki, Barry Boots, Kokichi Sugihara, Dr Sung Nok Chiu, and Sung Nok Chiu. Spatial Tessellations: Concepts and Applications of Voronoi Diagrams. 2nd ed. Wiley, 2000.

Pottmann, Helmut. "Geometry and new and future spatial patterns." Architectural design 79, no. 6 (November 2009): 60-65.

Prusinkiewicz, Przemyslaw, and James Hanan. Lindenmayer Systems, Fractals, and Plants. Springer, 1989.

Reid, Miles, and Balazs Szendroi. Geometry and Topology. Cambridge University Press, 2005.

Rowe, Colin. The Mathematics of the Ideal Villa and Other Essays. The MIT Press, 1982.

Shannon, Claude E. "Prediction and Entropy in Printed English." The Bell System Technical Journal 30 (January 1951): 50-64.

Shubnikov, A. Symmetry In Science and Art. 1st ed. Springer, 1995.

Stiny, George, and William J. Mitchell. "The Palladian Grammar." Environment and Planning B 5 (1978): 5-18.

Thompson, D'Arcy Wentworth. On Growth and Form: The Complete Revised Edition. Revised. Dover Publications, 1992.

Weaire, Denis, and Stefan Hutzler. The Physics of Foams. Oxford University Press, USA, 2001.

Williamson, Jack H. "The Grid: History, Use, and Meaning." Design Issues 3, no. 2 (October 1, 1986): 15-30.

Wittkower, Rudolf. Architectural Principles in the Age of Humanism, 2nd Edition. 2nd ed. Academy Press, 1998.

Wolfram, Stephen. A New Kind of Science. 1st ed. Wolfram Media, 2002.