Model Synthesis

Paul C. Merrell

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2009

Approved by:

Dinesh Manocha, Advisor

Jack Snoeyink, Reader

Benjamin Watson, Reader

Anselmo A. Lastra, Reader

Ming C. Lin, Reader

Abstract
Paul C. Merrell: Model Synthesis.
(Under the direction of Dinesh Manocha.)

Three-dimensional models are extensively used in nearly all types of computer graphics applications. The demand for 3D models is large and growing. However, despite extensive work in modeling for over four decades, model generation remains a labor-intensive and difficult process even with the best available tools.

We present a new procedural modeling technique called model synthesis that is designed to generate many classes of objects. Model synthesis is inspired by developments in texture synthesis. Model synthesis is designed to automatically generate a large model that resembles a small example model provided by the user. Every small part of the generated model is identical to a small part of the example model. By altering the example model, a wide variety of objects can be produced.

We present several different model synthesis algorithms and analyze their strengths and weaknesses. Discrete model synthesis generates models built out of small building blocks or model pieces. Continuous model synthesis generates models on set of parallel planes. We also show how to incorporate several additional user-defined constraints to control the large-scale structure of the model, to control how the objects are distributed, and to generate symmetric models. The generality of the approach will be demonstrated by showing many models produced using each approach including cities, landscapes, spaceships, and castles. The models contain hundreds of thousands of model pieces and are generated in only a few minutes.

Acknowledgments

Writing this thesis has been a long, but satisfying journey full of facinating, perplexing, and sometimes terribly frustrating problems. Through it all, I knew I could count on the support of friends and colleagues to boost my spirits.

I'm indebted first and foremost to my advisor Dinesh Manocha. I'm grateful for his steadfast support and for the many nights he helped me frantically finish papers before the deadline. More than anything, I'm grateful for the confidence he placed in me. Early on, there were so many reasons he could have doubted me and questioned the value of my work, but he believed in me and offered me the freedom to explore a subject that I'm most passionate about.

I'm thankful to Ming Lin, Ben Watson, Anselmo Lastra, and Jack Snoeyink for serving on the committee. Thanks to Anselmo for his abundant kindness and words of advice. Thanks to Ben for his procedural modeling expertise and for letting me renege on a promise I made to become his student at NC State. Thanks to Jack for his deep mathematical insights and for giving me lots of feedback.

Special thanks goes to Jess Martin who was the first person to show real enthusiasm and encouragement for my work. Thanks to Peter Wonka, Jeremy Wendt, and Phillipos Morodhai for reading some of my papers and suggesting revisions. Thanks to Vivek Kwatra for interesting conversations about texture synthesis and for providing Figure 2.10(c). Thanks to my officemates David Gallup, Brian Clipp, and Christian Lauterbach for being welcome distractions from the daily grind. Thanks to all the faculty, staff, and students at UNC for making my time here pleasant and fulfilling.

I'm deeply grateful to Mom, Dad, Brian, Christine, Douglas, Shannon, and Amy for their love and support. Even though my parents didn't understand a word of my

papers they read through them just to fix my grammatical and punctuational mistakes. I'm especially grateful to Dad for always being willing to listen to my ideas even the bizzare and incomprehensible ones. Thanks to Douglas for writing a powerful graphing program that I used in Figure 3.8.

Thanks to many friends especially in the Durham Third Ward for making life in Chapel Hill exciting and memorable.

Table of Contents

List of Tables

List of Figures

List of Symbols

E	Input Example Model
M	Generated Output Model
$n_x \times n_y \times n_z$	The length, width, and height of M
$n'_x \times n'_y \times n'_z$	The length, width, and height of E
K	The set of Possible Labels
k	The number of elements in K
$\hat{\imath}, \hat{\jmath}, \hat{k}$	Unit vectors in the x,y, and z directions
T_x, T_y, T_z	The Transition Matrices
$D_E(n_x, n_y, n_z)$	The number of solutions for a given size
$C^\star_{M_t}$	The ideal catalog of possible labels
C_{M_t}	The imperfect catalog of possible labels
$\exists$	There exists
$\forall$	For all
$\Rightarrow$	implies that
u	List of positions to update
$\cap, \cup, \wedge, \vee, \neg$	intersect, union, and, or, not
$m_x \times m_y \times m_z$	The size of the block to modify
R_i	The i-th possible region in M
$\oplus$	Minkowski Sum
V_i	The extent of object i
vf	The vertex figure
m	The number of distinct normals
n	The number of planes for each normal
$h_1, h_2, \ldots h_m$	The set of half-spaces or half-planes
$s_1, s_2, \ldots s_m$	The plane spacings
$\mathbf{n}_1, \mathbf{n}_2, \ldots \mathbf{n}_m$	The face normals
h_i^C	The complement of h_i
P_i	The set of points $\mathbf{x}$, where $E(\mathbf{x}) = i$

Chapter 1

Introduction

Three-dimensional geometric models are used to represent the shape and design of objects in nearly every type of computer graphics application including virtual environments, CAD/CAM, computer gaming, animated movies, and medical simulations. These applications require complex 3D models to be realistic and compelling. The demand for detailed 3D models is large and expanding. However, satisfying the demand for models is difficult. Realistic models often contain very complex and widely varying shapes and styles. Modeling can be tremendously time-consuming. For example, the urban models created for the movie *Superman Returns* took 15 man years to complete [40].

Modeling is a creative and artistic process. The objects being modeled may not be based upon real objects, but purely on an artist's imagination. Modeling involves many artistic and high-level design decisions. Decisions about the style and purpose of each object must be made to produce compelling models. Even though creative decisions are an integral part of modeling, in practice, users spend more effort on routine and tedious tasks.

Despite extensive work in geometric modeling for over four decades, it remains a labor-intensive and difficult process even with the best available tools. Current modeling tools are notoriously complex. Learning how to use them requires significant training

and even when the tools are mastered creating complex models is still difficult. With state of the art 3D CAD and modeling tools such as Autodesk's 3D Studio Max and Maya, the user can create simple geometric primitives and modify them using various transformations and geometric operations. Modeling complex environments such as cities or a landscapes requires creating and manipulating a huge number of primitives and can take many hours or days [40].

Fortunately, there are many reasons to believe that the modeling process can be greatly simplified and automated. Modeling involves many routine and repetitive tasks. Many of the objects in games, movies, and virtual environments contain repetitive and self-similar structures. Self-similarity is common in man-made objects and natural objects [38] (Figure 1.1). Self-similarity is often used to simplify and automate the modeling process. Automation is the goal of *procedural modeling* techniques. In procedural modeling, automatic procedures are used to generate models.

(a) Photograph of a Fern   (b) Photograph of Prague

Figure 1.1: Objects with repetitive or self-similar structures tend to be procedurally modeled more easily. Self-similarity is a common feature of both man-made and natural objects.

This thesis explores a new procedural modeling technique that is designed to apply broadly to many classes of objects. It is inspired by recent developments in the texture synthesis literature [15, 74]. Textures are loosely defined as images containing some type of repeated pattern. The goal of texture synthesis is to create a large texture

that resembles an example texture. For example, from the small example in Figure 1.2(a) a texture synthesis algorithm would generate the large texture in Figure 1.2(b). Texture synthesis is based upon the user specifying what the algorithm should generate by providing an example. Texture synthesis is one of many techniques which use this example-based principle. Example-based techniques are also used for generating high resolution images from low resolution images [18], for filtering images so they resemble a particular painting or drawing [24], for skinning [56], and for generating curves [25]. By using an example, the user can often specify what kind of results should be produced more easily and more intuitively. Example-based techniques often apply more generally to a wider variety of models than other methods [74]. For example, texture synthesis methods which use examples can generate a wider variety of textures than other methods such as Perlin noise [47]. Even though example-based techniques have been applied to many areas of graphics, their use in modeling has been limited [19].

(a) Example Input Texture   (b) Output Texture
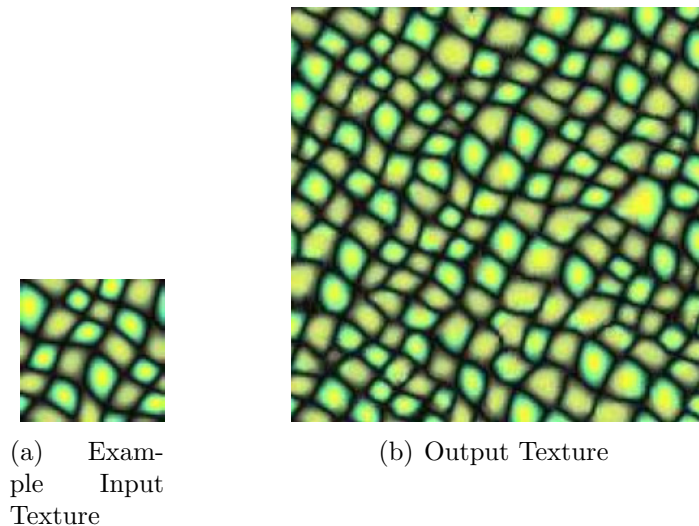
Figure 1.2: Texture synthesis algorithms take a small input example texture (a) and produce a new texture (b) that resembles it.

In an example-based modeling technique, the user would provide a small example model (Figure 1.3(a)) and then the algorithm would generate a larger model that resembles it (Figure 1.3(b)). This type of algorithm is called a *model synthesis* algorithm

because it is similar to texture synthesis. Designing this type of algorithm for 3D models is the central goal of this thesis.

(a) Example Input Model   (b) Synthesized Output Model
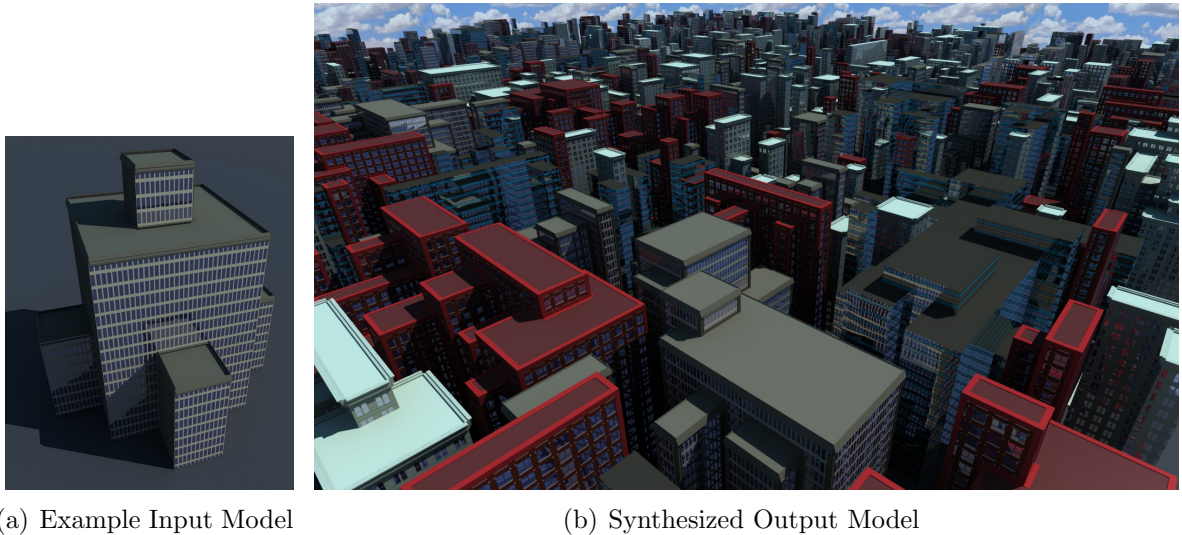
Figure 1.3: Model Synthesis Input and Output

1.1   Thesis Statement

We introduce an example-based procedural modeling algorithm that allows a 3D modeler to generate a variety of complex and rich environments relatively quickly and easily. Our model synthesis algorithm can efficiently generate large models containing flat polyhedral shapes common in architecture.

1.2   Thesis Goals

The thesis has four main goals:

- **To develop algorithms that generate 3D models resembling an input model.**

- **To analyze the strengths and limitations of such algorithms including their time and memory requirements.**

- **To demonstrate of the generality of these algorithms by modeling many diverse complex objects and environments.**

- **To impose several additional user-defined constraints on the generated model.**

This thesis focuses on one central problem of generating a new 3D model that resembles a given input model. Models may resemble one another for a variety of reasons and so the notion of resemblance needs to be defined more precisely. A similar issue is encountered in the texture synthesis literature. Many texture synthesis algorithms are based on the principle that two textures resemble one another, if the patches of texture they contain are similar. That is, if every small patch in one texture is similar or identical to a small patch in the other. The same principle could be applied to 3D models. Models resemble one another if every small part of one model is identical to some part of the other.

We consider two different model synthesis problems: discrete model synthesis and continuous model synthesis. In discrete model synthesis, the user divides an input model into discrete building blocks called *model pieces* shown in Figure 1.4(a). The model pieces are also called *labels*, since every point in a 3D array is labeled according to which model piece occupies it. Discrete model synthesis is simpler than continuous model synthesis and is discussed first.

The goal of model synthesis is to generate a new model in which each pair of adjacent labels exactly matches a pair of adjacent labels in the input model. This is called the *adjacency constraint*. The effect of the constraint is illustrated in Figure 1.4. Figure 1.4(b) satisfies the constraint, but Figure 1.4(c) violates it. The adjacency constraint ensures that all of the model pieces fit together seamlessly and that the new model

(a) Model divided into *model pieces*

(b) A Seamless Model

(c) A Model with Many Conflicts
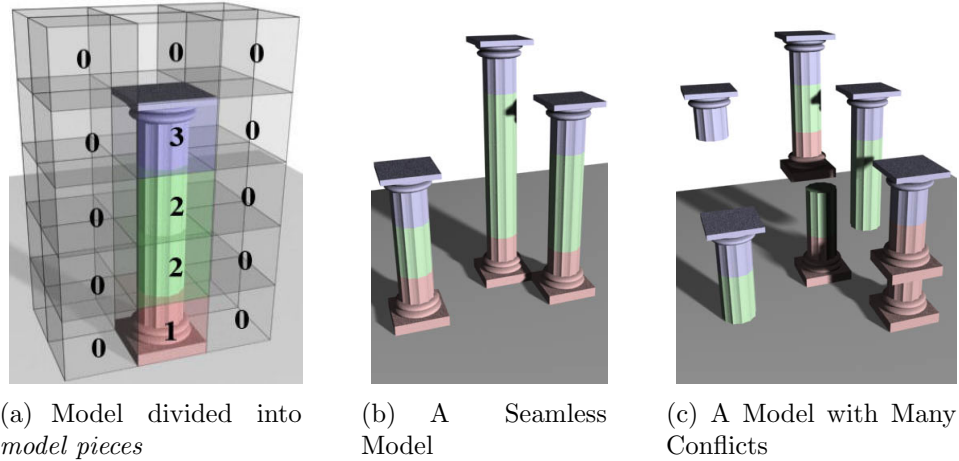
Figure 1.4: In discrete model synthesis, the user divides the model into model pieces (a). The goal is to generate a new model whose pieces fit seamlessly (b). If the pieces do not fit together, the model (c) does not resemble the input.

resembles the input. By always satisfying this constraint, model synthesis improves upon texture synthesis. Current texture synthesis algorithms do not always satisfy the constraint. They may generate textures containing parts that do not fit together properly and conflict with each other. These conflicts occur because existing texture synthesis algorithms only check the local neighborhood around a pixel when it is added. The model synthesis algorithm searches for possible conflicts globally, so it can find and avoid conflicts between the labels more effectively. This global search is particularly valuable for model synthesis, but it is also useful for texture synthesis. The search is performed using a catalog of possible labels that could be added. An example of this catalog is shown in Figure 1.5. Each label corresponds to a model piece. Labels are removed from the catalog, if they conflicts with the current model. Each removal may cause other adjacent labels to be removed. The removals may propagate through the array. So a possible conflict in one of the locations may cause labels to be removed in a distant location. When labels are added into the model, they are only selected from the catalog to avoid possible conflicts.

(a) Example Model, E (b) Incomplete Model M (c) Catalog of labels to add, C_M
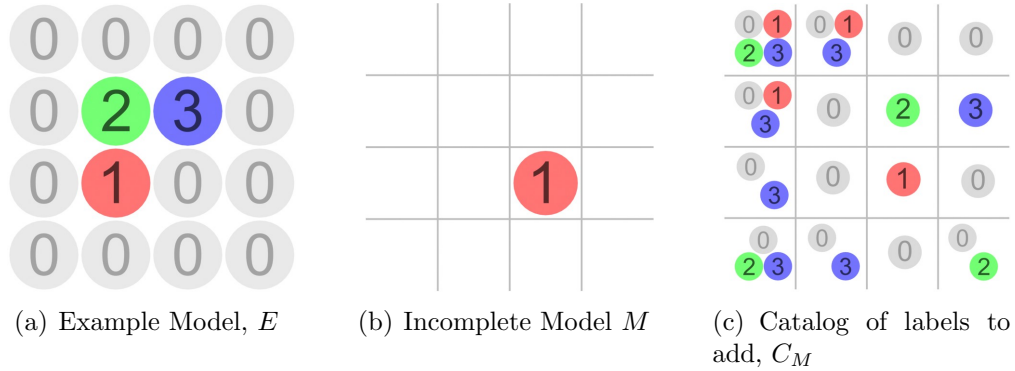
Figure 1.5: For the example model E (a) and the incomplete model M(b), a catalog of possible assignments is computed (c).

However, this global search may not be able to detect every conflict. In fact, detecting all conflicts when generating a large model can be extremely difficult. To show this, we present an NP-completeness proof in Theorem 3.3.5. An undetected conflict can cause the catalog to become empty. This is a serious problem since then there would no labels to choose from and the adjacency constraint would be violated. If the catalog becomes empty, the algorithm in its initial form fails. To handle failures, we introduce a second algorithm. The second algorithm is based upon the observation that the initial algorithm succeeds much more frequently when generating small models. The second algorithm creates large models in small parts. If a failure occurs when creating one of the small parts, the algorithm backtracks slightly and continues.

In summary, model synthesis improves upon texture synthesis in two key ways. First, it uses a global search to find and avoid conflicts and second, it creates the model in parts. With these improvements, it can generate models where all of the model pieces fit together seamlessly.

The discrete and the continuous model synthesis problems are both solved by using a global search and by modifying in parts. The key difference between them is that discrete model synthesis represents the models as an array of labels as shown in Figures 1.4(b) and 1.5. Discrete model synthesis assumes that the user provides an example model

that has been decomposed into discrete model pieces that fit on a grid. The algorithm works well if it is given a good example model. But providing the example model can be difficult since many models do not fit naturally onto a grid. If the model does not fit well on a grid, then model synthesis cannot generate interesting new variations similar to the input. The algorithm generates only exact copies of the input. This problem is caused by using discrete model pieces. An algorithm that does not use discrete model pieces could overcome this problem. Rather than trying to assign labels to every discrete point on a grid, a better goal would be to assign labels to every point in 3D space i.e. continuous model synthesis. The goal is still to assign labels that satisfy an adjacency constraint, but the points are now in the continuous domain. Discrete and continuous model synthesis share many of the same concepts. Both methods use a catalog of possible labels, but the catalog is much more difficult to compute in the continuous case. The continuous domain includes an infinite number of points, so the catalog may contain an infinite number of possible assignments and the catalog is recorded geometrically rather than in an array. We propose several different ways of computing this catalog, but some of them are too difficult to implement. One way to greatly simplify the continuous problem is to assume that the faces of the output lie on a set of planes parallel to the input. This assumption imposes an additional constraint on the output which can interfere with the result in a few cases.

An overview of the continuous model synthesis algorithm in is shown in Figure 4.10. Starting with the input example shape shown in Figure 1.6(a), we create sets of lines parallel to the edges of the input as shown in Figure 1.6(c). These lines divide the plane into an arrangement of faces, edges, and vertices. Each face, edge, and vertex is associated with a set of acceptable neighborhoods or labels that satisfy the adjacency constraint. The set of possible labels could be computed by dividing the input model along parallel lines as shown in Figure 1.6(b). The remaining steps of the algorithm are the same as discrete model synthesis. A catalog of possible labels is maintained

to search globally for potential conflicts and the model can be modified in parts. The algorithm generates an output model satisfying the adjacency constraint such as Figure 1.6(d).

(a) Example Model E

(b) Parallel lines dividing E

(c) Parallel lines diving up the plane.

(d) Output Model
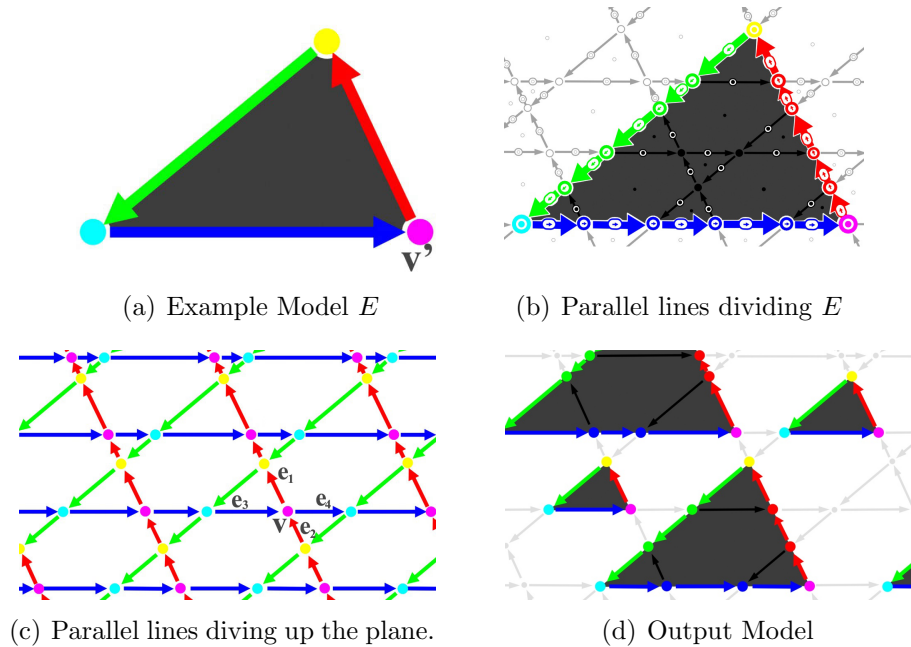
Figure 1.6: Continuous Model Synthesis Overview. Lines parallel to the input shape (a), divide the plane into faces, edges, and vertices (c). The output shape (d) is formed within the parallel lines. The set of acceptable vertex and edges labels in the output (d) can be found by dividing the input along parallel lines (b).

Overall, model synthesis offers many benefits. Most other procedural modeling techniques that are targeted to a specific type of object, but model synthesis can generate a wide variety of objects and environments including cities, landscapes, plants, fractal structures, castles, cathedrals, spaceships, roller coasters, oil platforms, building interiors and more. A few examples are shown in Figure 1.7. In each case, the only user input is a simple example model.

The primary goal of both discrete and continuous model synthesis is to satisfy the adjacency constraint, but many additional user-defined constraints should be used to create more realistic models. The user might have a floor plan or a general idea of what the model should look like on a macroscopic scale. The user might want to create models

9
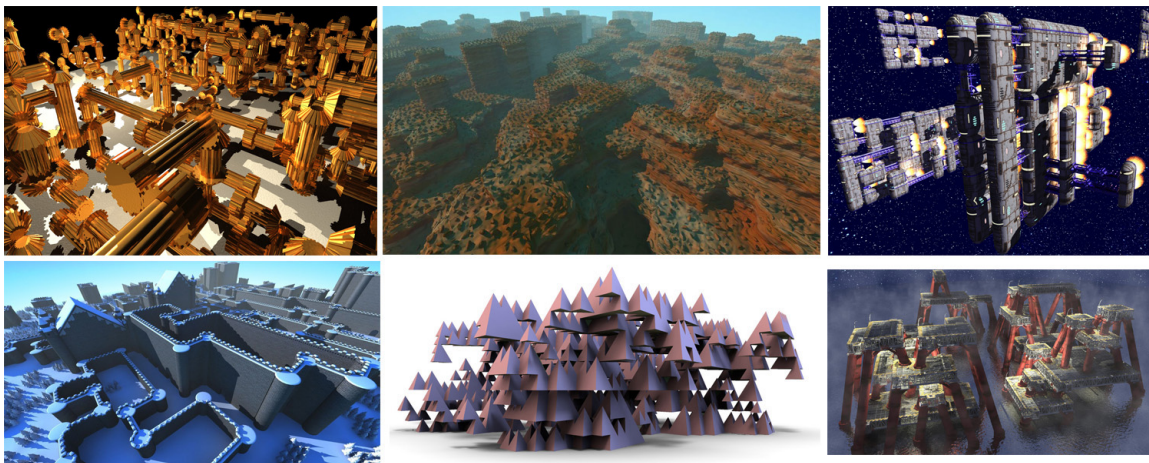
Figure 1.7: Examples of the wide variety of shapes that our model synthesis algorithm can generate including machinery, landscapes, spaceships, castles, fractals, and oil platforms.

that are symmetric. These constraints and many others can be imposed on the models with our framework.

1.3 Chapter Overview

The rest of the thesis is organized as follows. Chapter 2 surveys related work in procedural modeling and texture synthesis. Section 2.3 discusses differences between textures and 3D models that explain why texture synthesis techniques generate 3D models less effectively.

Chapter 3 discusses discrete model synthesis. The problem is formally described in Section 3.1. It is shown that the number of possible solutions may grow exponentially with the output size. An algorithm for finding solutions is described in Section 3.3 and its time complexity is analyzed. Unfortunately, this algorithm fails to complete properly in some cases. The catalog may become empty and the algorithm cannot continue. Section 3.3.5 shows with an NP-completeness proof that in some cases these failures are unavoidable for any polynomial-time algorithm unless $P = NP$. Section 3.3.6 describes an improved algorithm that lowers the frequency of failures and properly handles them

if they occur.

Chapter 4 discusses continuous model synthesis. Section 4.1 explains why continuous model synthesis is needed by discussing the limitations of discrete model synthesis. Several different approaches are introduced. Sections 4.3 and 4.4 describe two different approaches that could be used, but these approaches have several serious implementation issues. For example, one approach has not been implemented because it requires exact and robust 3D Boolean operations and 3D Minkowski sum computations. Section 4.5 describes a more practical approach that is much simpler to implement because it assumes that the models are generated on sets of parallel planes. This parallel plane assumption introduces some limitations which are also discussed. Also, other constraints beyond the adjacency constrain are described for controlling the output more effectively.

Model synthesis is compared with texture synthesis in Section 5.1 and with other procedural modeling techniques in Section 5.2. Model synthesis is also compared with formal grammar and a close relationship between them is established.

Chapter 6 summarizes the main points of the thesis and discusses exciting possibilities for future research.

Chapter 2

Related Work and Background

This chapter discusses work related to model synthesis in the fields of procedural modeling and texture synthesis. Section 2.3 discusses differences between texture synthesis and model synthesis and how texture synthesis might be extended to 3D modeling and why a new algorithm is needed for model synthesis.

2.1 Procedural Modeling

Many procedural modeling techniques have been developed over the last few decades. These techniques as a group have a great amount of variety in the approach they take. Most techniques are targeted at modeling a specific type of object or environment. Early techniques based on fractal geometry achieved some success modeling natural landscapes [17]. A connection between landscapes and fractal geometry was observed in the 70s [38]. Mandelbrot observed that a record of Brownian motion over time resembles an outline of jagged mountain peaks. Models of landscapes can be further improved by considering how landscapes erode over time [42].

There also is a long history of modeling plants procedurally. Many plant modeling techniques use a formal grammar call an L-system. L-systems were proposed by Lindenmayer as a general framework for describing plant growth and plant models [35, 52]. An L-system is a parallel rewriting system. L-systems can be extended made to consider

how plants interact with their environment as they grow [39]. Many techniques also use information supplied by the user to influence the shape of the plant models such as positional information [53], sketches of plants [6], or photographs [54, 61].

Many techniques are designed targeted specifically for modeling urban models procedurally [69, 65]. Like many plant modeling techniques, some urban modeling techniques use L-systems. L-systems have been used to generate road networks from elevation and population density data and to generate buildings on parcels of land between the roads [44]. Other grammars have been introduced specifically for modeling architecture. The architect, Stiny [1971] introduced shape grammars as a tool for analyzing and designing architecture. Shape grammars remained largely a conceptual tool [59, 16] until Wonka and others introduced a related group of grammars called split grammars [75]. Split grammars operate by splitting shapes into smaller components and can generate highly detailed models of architecture. Split grammars were further developed by Müller et al. [40] who include shape operations for mass modeling and for aligning many parts of a building's design together. Their method can generate both the large-scale layout of a city as well as many geometric details within each building to produce a highly complex and realistic city. Tools have also been developed to edit these grammars visually using a GUI [36] and for deriving grammars automatically from images of facades [41]. A method developed by Aliaga et al. [1] constructs grammars from photographs with the user guiding the creation and subdivision of an initial 3D model.

Figure 2.1: Procedurally Generated Buildings created using Müller et al. [40]

13

Another group of techniques focuses more heavily on the 2D layouts of cities than on the 3D shapes of the buildings. Chen et al. [5] allow users to edit a city's street layout interactively using tensor fields. Aliaga et al. [2] generate street layouts using an example-based method. This is particularly relevant as their method combines elements of texture synthesis and procedural modeling. The streets are generated like other procedural modeling techniques and then an image of the city seen from above is generated like a texture using texture synthesis. A related area of research is urban simulation. Much of the research into urban simulation is conducted outside of computer graphics where the purpose is not to model and render cities, but to understand how various factors influence a cities development and growth over time [63, 67]. However, area of research is certainly relevant to computer graphics and several authors have incorporated aspects of urban simulation into their methods to produce more realistic models of cities [70, 33]. Their methods simulate part of a city's economy and generate street layouts and zone the land area for different economic activities.

There are also other techniques designed to model much smaller structures than cities. Legakis et al. [34] propose a method for automatically embellishing 3D surfaces with various cellular textures including bricks, stones and tiles. Cutler et al. [11] developed a method for modeling layered, solid models with an internal structure. Their method can modify models by simulating various physical processes such as erosion and fractures. Another method has been developed to model truss structures by optimizing the locations and strengths of beams and joints that support bridges, tower platforms, and other objects [58]. Pottmann et al. [49, 50] have developed algorithms based on discrete differential geometry that determine how to arrange beams and glass panels so they form in the shape of a given freeform surface and satisfy various geometric and physical constraints.

Another way to model objects is to combine together parts of existing models interactively [19]. In this method, the user can search through a large database of 3D models

to find a desired part, then cut the part out from the model, and stitch various parts together to create a new object.

2.2 Texture Synthesis

Although model synthesis is designed for procedural modeling, the algorithm itself has more in common with texture synthesis. The field of texture synthesis has seen many exciting new developments over the past decade. This section surveys these developments and explain their relationship to model synthesis. A more detailed survey is given in [74].

2.2.1 Markov Random Fields

Textures are often described as Markov Random Fields [15, Zhu et al., 48, 43, 74]. Markov Random Fields have a set of random variables X_i. In this case, each random variable represents the color of a pixel. Each pixel i has a set of neighbors surrounding it called N_i. It is often assumed that only the neighbors of pixel i determines its value. This assumption is called the Markov locality property. Stated more formally, the color of pixel i is conditionally independent of the pixel colors outside N_i, given the pixel colors inside N_i. This means the probability of X_i having the color x_i has the following property $\forall x_1, x_2, \ldots$

$$P[X_i = x_i \mid \forall j \neq i, X_j = x_j] = P[X_i = x_i \mid \forall j \in N_i, j \neq i, X_j = x_j] \qquad (2.1)$$

Many texture synthesis algorithm use this assumption [74].

2.2.2 Texture Synthesis Algorithms

Over the past decade, the field of texture synthesis has seen a proliferation of new algorithms and new ideas. Many of these algorithms were influenced by a seminal paper written by Efros and Leung [1999]. Their algorithm is remarkably simple and produces good results. Their algorithm generates textures by adding pixels individually. To determine which pixel should be added at a given point, a small neighborhood around the point is compared against every neighborhood in the example texture. The purpose of the comparisons is to find which neighborhood matches the neighborhood around the insertion point the closest. The quality of each match is evaluated using a sum of squared differences. Figure 2.2 shows a set of close matches for a given neighborhood. A neighborhood is a close match if it matches to within a certain percentage of the closest match. From among every close neighborhood, one is randomly selected and its central pixel is added into the new texture. Every pixel of the texture is added this way. Efros and Leung's method [1999] is one of the simplest texture synthesis algorithms. It generally produces good results, but it does have some failure cases. The algorithm is slow because computing an exhaustive nearest neighborhood search is expensive.

(a) Input Example Texture

(b) Synthesizing a Pixel
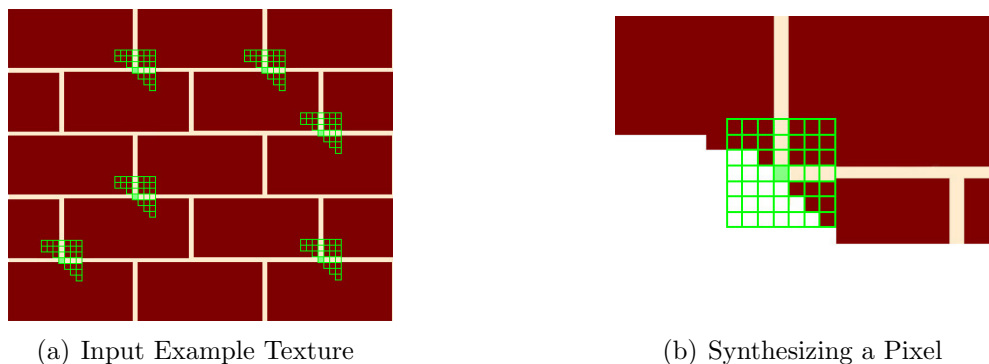
Figure 2.2: Illustration of Efros and Leung's Algorithm [1999]. To determine which pixel to insert at a given point, the neighborhood around the point is compared against other neighborhoods in the example texture. The pixel to insert is randomly selected from among the closely matching neighborhoods.

In Efros and Leung's method, the textures are typically generated by starting from

the center and adding pixels going out in concentric rings. However, by adding pixels in a different order, the speed of the algorithm can be improved as shown by Wei and Levoy [2000]. They developed a similar algorithm that adds the pixels in scan line order. Suing this order, the method can be acceleration using tree-structured vector quantization.

There are several other ways to accelerate texture synthesis. Several acceleration techniques are based on the observation that groups of neighboring pixels in the input are likely to be grouped together in the output. This is known as coherence. Coherence is used in several methods to improve the performance [62, 3]. Coherence can be used to compute approximate nearest-neighborhood matches very quickly to be used in interactive editing tools [3]. Another acceleration strategy is instead of adding pixels individually, they can be added in large groups or patches. Patches of texture rarely fit together seamlessly, but an optimal cut can be made between the patches so they fit together without any noticeable seams as shown in Figure 2.3. The cuts can be made either by using dynamic programming [14] or by using graph cuts [29]. Another approach to texture synthesis is to optimize a global energy function using an expectation maximization algorithm [30].

(a) Patches Overlap   (b) Patches are Stitched Together
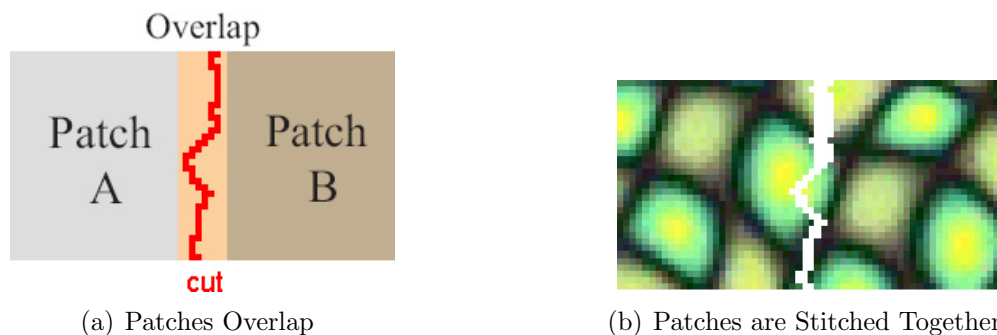
Figure 2.3: In patch-based texture synthesis, different patches are copied and overlapped slightly. An optimal cut is computed between the two patches.

Textures are primarily used for texture mapping 3D models. Ordinarily, textures are synthesized onto a flat texture map which is then applied to a curved surface, but this is not the most direct approach. Instead, the textures could be synthesized directly

onto the curved surfaces. This can be accomplished with an orientation field covering the surface [64, 72]. The orientation field specifies the direction of the texture and plays the same role as the rows and columns of an array of pixels.

What is novel about model synthesis is that it extends texture synthesis to 3D modeling. It is not simply an extension into three dimensions since this has been done before. Some methods have given a third temporal dimension to textures so that they move and change over time [71, 29]. This is especially useful for creating moving images of fire, smoke, and running water. These dynamic textures could also be described using linear dynamical systems [12].

Another reason to extend texture synthesis into three dimensions is to create solid textures [45, 46]. Solid textures are an alternative to 2D texture maps. They describe many natural materials such as wood and stone more accurately than 2D texture maps and they avoid the task of parameterizing the object's surface which can be difficult. Until recently, solid textures were only created procedurally, but now texture synthesis can be used [23, 28]. Solid texture synthesis starts like ordinary texture synthesis from an input of a 2D texture, but the goal is to produce a 3D solid texture which certain properties. It should be possible to slice the solid texture open to reveal a pattern on the slice that resembles the input texture. Such a solid texture could be generated by optimizing a global energy function [28] like some 2D texture synthesis algorithms [30].

Texture synthesis has also been used to generate what are called geometric textures [4, Zhou et al.] which are a combination of texture mapping and modeling. They are used like texture maps to apply patterns to objects, but these patterns actually change the shape of the object itself. This is useful for creating objects with bumps or dimples (Figure 2.4(a)) or objects that are made out of chain mail (Figure 2.4(b)). Texture synthesis has also been used to generate hair in different styles [68] and to create 3D models of terrain using real elevation data as the example [76]. Texture synthesis has also been combined with certain elements of procedural modeling to create 2D arrangements

of objects [26].

Lagae, Dumont, and Dutré [2005] developed a method called geometry synthesis
which resembles model synthesis in some ways. Their method also extends texture
synthesis into three dimensions for procedural modeling. Their algorithm takes an input
model and computes a 3D array of signed distance values. This array is used as the
example and a new array is generated using a standard texture synthesis technique.
However, texture synthesis methods have difficulty with many inputs when they are
extended into modeling. This is discussed in Section 2.3 in detail. This method is
applied to models that have regular patterns as shown in Figures 2.4(c) and 2.4(d).

(a) Geometric Textures   (b) Mesh Quilting

(c) Chain Mail   (d) Grid
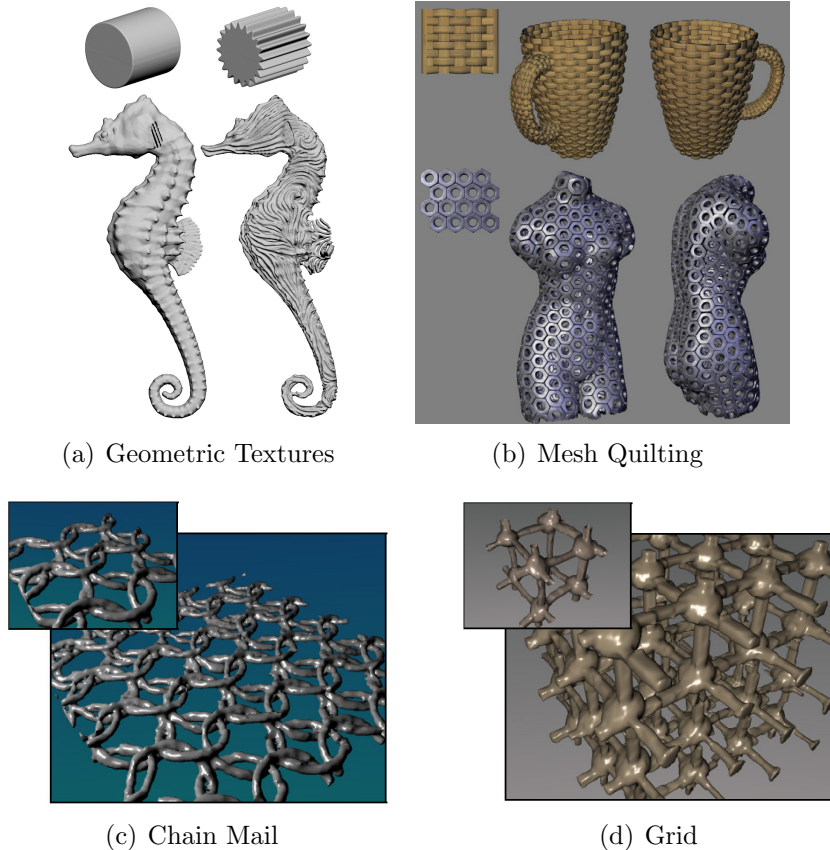
Figure 2.4: Result from Geometric Textures (a) [4], Mesh Quilting (b) [Zhou et al.] and
Geometry Synthesis (c & d) [32]. Each method extends texture synthesis in some way to
generate models. The models have patterns that resemble textures. Geometry synthesis
generates models from examples. The examples are shown in the insets (c & d).

Some texture synthesis techniques use tiles to accelerate the algorithm. Tiles are particularly relevant to model synthesis since model pieces are essentially 3D tiles. Most of these methods use Wang tiles. Wang tiles were studied initially by mathematicians interested in aperiodic tiling [10], but they have also been applied to texture mapping and texture synthesis [Stam, 8, 31]. The 3D counterpart of a Wang tile is a Wang cube. Wang cubes have been used to model asteroid fields [57] and render volume data [37]. However, we show that it is often difficult to apply Wang tiles and Wang cubes to modeling later in Section 5.1.1.

A few other advancements in texture synthesis should be mentioned. A multiscale algorithm has been used to generate extremely high resolution textures [21]. The texture synthesis process can be inverted to find a small representative example texture from a large texture [73]. Texture synthesis can also be used to complete a missing part of an image [9, 13, 60, 22]. This is especially useful for removing objects from images without leaving holes. A few of the image-completion techniques change choosing the order in which the pixels are added to improve the results [9, 60].

A related technique called context-based surface completion [55] completes models that contain regions with missing surface information or holes. Surface completion fills in any missing regions with surfaces that resemble the rest of the model. In this case, the rest of the model effectively acts as the example.

2.3 Differences between Textures and Models

Texture synthesis and model synthesis have similar goals. However, textures and models differ in several important ways that affect how they are generated. An obvious difference is that textures have two spatial dimensions while models have three, but most texture synthesis algorithms can easily be extended to operate in three dimensions. In fact, texture synthesis algorithms have frequently been used to create three-dimensional solid

textures [28] or textures with a temporal dimension [71, 12, 29].

Textures are typically represented as arrays of pixels which store red, green, and blue color values. Models are typically represented using geometric shapes such as polygonal meshes or NURBS. Texture synthesis methods could be directly used for modeling if the models were represented in an array. The elements of the array would be small building blocks called *model pieces*. Each model piece would contain textured geometric shapes within a cubic volume of space as shown in Figure 2.5. Model pieces are similar to texture patches or to tiles that some texture synthesis methods use [7]. The model pieces are created by the user. The user could break an existing model down into model pieces or could start by creating the model pieces and then build the model up with them. Most model pieces should be in the model multiple times. Otherwise, the model is not self-similar and the algorithm may not create interesting new variations off the original model. However, it can sometimes be difficult to create the model so that the model pieces repeat. These difficulties are described in Section 4.1. To overcome them, we introduce continuous model synthesis which does not use model pieces in Chapter 4.
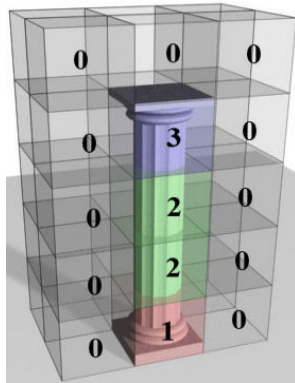
Figure 2.5: Model constructed out of model pieces.

Each model piece corresponds to a label. Labels are assigned to every point in a 3D array. The labels are numbered $0, 1, \ldots, k-1$ where k is the total number of model pieces. The label 0 is typically reserved for the empty space model piece.

Model synthesis differs from texture synthesis because model pieces differ from pixel

colors in several ways. First, colors can easily be blended together. Mixing red and yellow, produces orange. But model pieces are not so easily blended. The top of a pyramid cannot easily be blended with the bottom of a sphere. Averaging the labels 1 and 3 does not produce the label 2. Another difference is that colors can easily be positioned in a color space where similar colors are grouped together in the space. But model pieces are different. The label 1 is not necessarily closer to label 2 than to label 15. Model synthesis is in some sense stricter than texture synthesis. In texture synthesis, if a color is close to the right value that often is good enough, but in model synthesis, that is not good enough. For example, suppose some model pieces contain flat squares. The model pieces fit together perfectly, if both squares are at exactly the same height, but if one square is slightly higher then there is a noticeable hole in between them. Minor changes to the model pieces can produce large errors.

There are other important differences between textures and models that go beyond differences in the number of dimensions they use or between pixels and model pieces. We demonstrate these other differences by running texture synthesis algorithms on 2D shapes commonly found in models. Let us consider as an input two of the simplest shapes: a triangle and a rectangle (Figure 2.6).
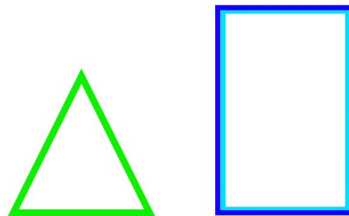
Figure 2.6: Typical Shapes used in Modeling

These shapes are common in geometric models. They could represent the floor plan of a triangular or a rectangular building. The shapes can be rasterized and their image can be used as an input into a texture synthesis algorithm, but surprisingly, texture

synthesis algorithms have difficulty even with these simple shapes. Although the shapes in Figure 2.6 are typically used in modeling, they differ from typical textures used in texture mapping or texture synthesis. A few representative examples of textures used in texture synthesis are shown in Figure 2.7 for comparison. Each texture was generated using a texture synthesis method by Kwatra et al. [2005].

(a)   (b)   (c)   (d)
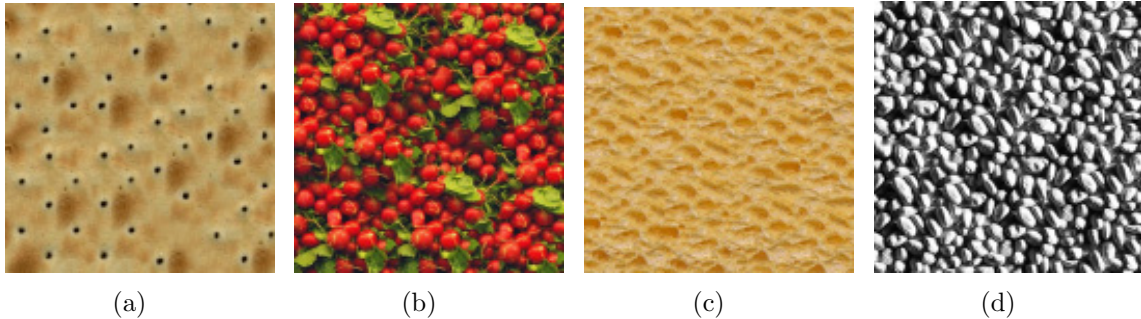
Figure 2.7: A few representative textures typically used in texture synthesis.

One difference between a typical textures and a typical shape used in modeling is they have different types of boundaries. 3D models typically represent objects with a few sharp well-defined boundaries between their interior and exterior. (One possible exception might be a puff of smoke.) On the other hand, a texture has soft or hard boundaries where its image intensity changes. The change could be soft and gradual or a hard edge. Textures typically have many edges rather than a few prominent ones. For example, Figure 2.6 only has seven edges, while Figure 2.7(d) has many more.

In order to better understand why texture synthesis algorithms have trouble with the shapes in Figure 2.6, let us examine the synthesis process of a typical texture synthesis algorithm. Let us examine Efros and Leung's method [1999]. Suppose that about halfway through the algorithm, it has produced a half-finished result as shown in Figure 2.8(b). The bottom half is finished, but the upper half has not been determined. The next step of the algorithm is to determine the pixel color at pixel **c**. This is accomplished by finding pixels with neighborhoods similar to pixels in the input in Figure 2.8(a). Let

us examine two of the alternatives: pixel **a** or pixel **b**.

(a) Input Shapes (b) Half-finished Output (c) Pixel **b** Chosen (d) Pixel **a** Chosen (e)
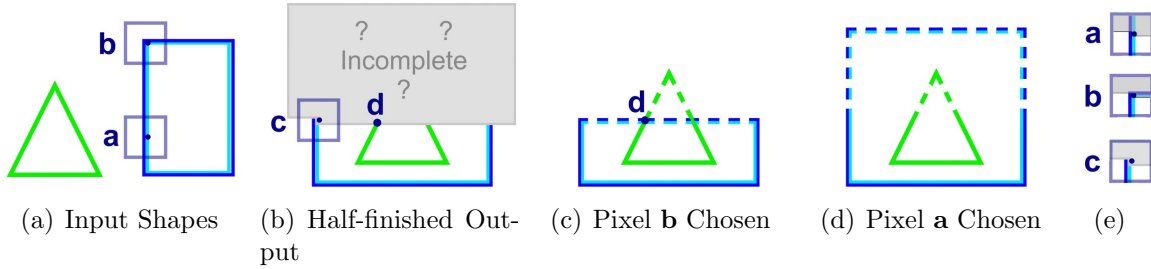
Figure 2.8: Some texture synthesis methods have difficulty with input (a). After half of the texture has been created (b), a decision is being considered at pixel **c**. If pixel **b** from the input is used, the texture can not be completely properly (c). If pixel **a** is used, it can be completed (d). The bottom halves of the neighborhoods around pixels **a**, **b**, and **c** are the same (e).

Each alternative looks equally suitable if we only examine the neighborhood surrounding pixel **c**. In fact, the neighborhoods around pixels **a**, **b**, and **c** have identical bottom halves (Figure 2.8(e)). Since Figure 2.8(b) is unfinished, we have no information about what is above or directly to the right of pixel **c**. Locally, both choices appear to be equally good, but in practice, they are not. If pixel **b** is chosen, the algorithm will eventually fail. Once pixel **b** is chosen, the texture cannot be completed by only using neighborhoods from the input (Figure 2.8(a)). There is only one way to complete the rectangle which is shown in Figure 2.8(c), but the rectangle is bound to intersect the triangle's edge at pixel **d**. Edges should not intersect, because the input in Figure 2.8(a) does not contain any intersecting edges. If pixel **a** is chosen, the texture can be completed successfully as shown in Figure 2.8(d). But there is no way of knowing that **a** is a better choice than **b** by only looking at local neighborhoods.

The choice of inserting the value at **b** into pixel **c** is unacceptable because of the triangle's edge at pixel **d**. The value at pixel **d** influences the value at pixel **c**, even though these pixels are far from one another. In fact, even if Figure 2.8(b) were scaled up a hundred times, pixels **c** and **d** would still influence one another across an even larger distance. This example demonstrates a problem common to many texture synthesis al-

24

gorithms which only examine local neighborhoods when making their decisions. At first glance, the fact that pixel **d** influences pixel **c** while it is outside the local neighborhood of **c** might appear to violate the locality assumption in Equation 2.1. But Equation 2.1 assumes the entire neighborhood $N_{\mathbf{c}}$ surround pixel **c** is know, but in Figure 2.8(b), the values of the pixels directly above pixel **c** are unknown. So the locality assumption is not violated.

Figure 2.8 illustrates a single error that a texture synthesis algorithm could make, but there are many more chances to make errors when synthesizing a large texture or model as shown in Figure 2.9. Results from a different input shape are shown in Figure 2.10 in which numerous shapes fail to close.

(a)
Input
Shape

(b) Efros and Leung, 1999
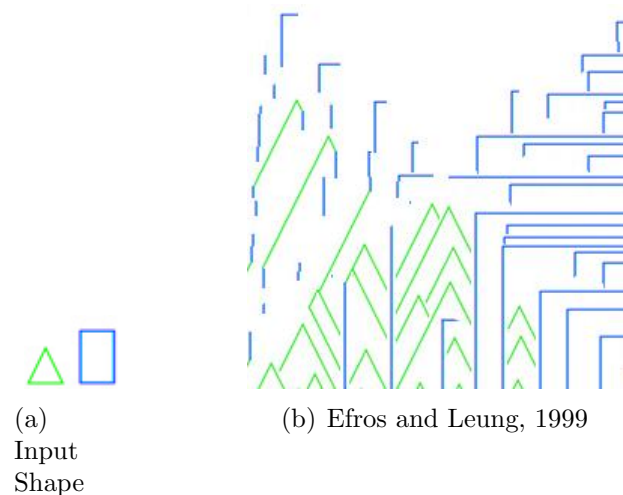
Figure 2.9: Texture synthesis methods have difficulty with the triangle and rectangle input (a). Results from a texture synthesis method is shown (b).

Other texture synthesis techniques work differently from Efros and Leung's method. Patch-based methods copy patches of texture together with some overlap and then cut and stitch the overlapping parts together. An optimal cut is computed using dynamic programming [14] or graph cuts [29]. These techniques work fine on typical textures shown in Figure 2.3, but work poorly for an input like Figure 2.6. Two patches are copied from Figure 2.6 and placed into Figure 2.11 which shows the two patches cannot

(a)
Input
Shape

(b) Efros and Leung, 1999

(c) Kwatra et al., 2005
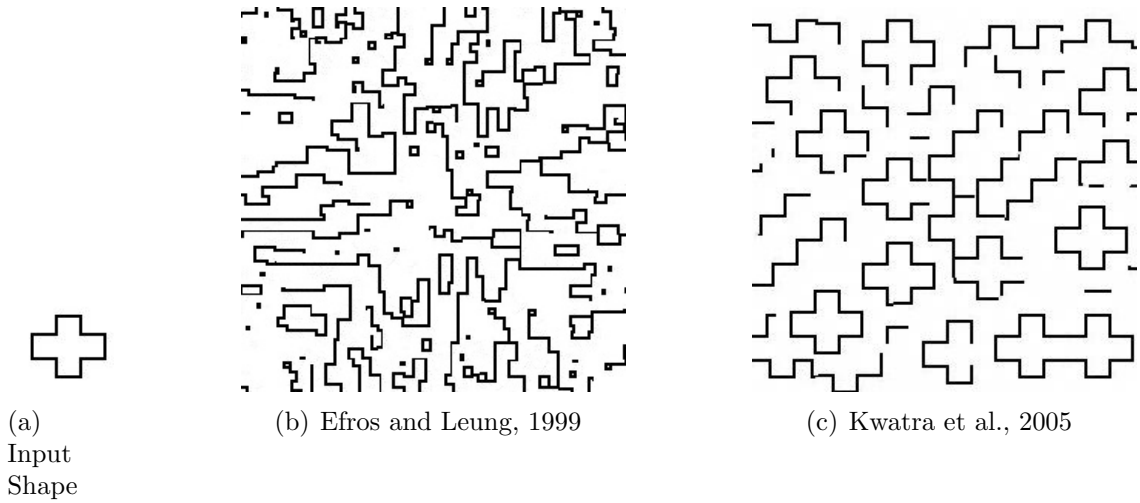
Figure 2.10: Texture synthesis methods have difficulty with a cross-shaped input (a). Results from two different texture synthesis methods are shown (b,c).

be stitched together. Most patches copied from Figure 2.6 would have the same problem.

(a) Patches are Stitched Together

(b) Overlapping Patches
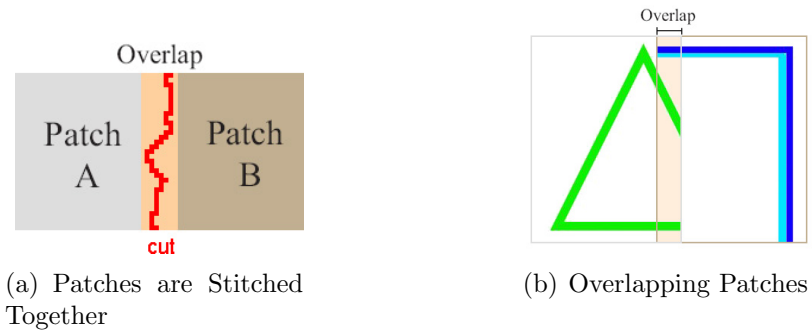
Figure 2.11: In patch-based texture synthesis, different patches are copied and over-lapped slightly. A optimal cut is supposed to be made, but none of the cuts are good.

Patch-based methods work on typical textures, because the textures can be stitched together without great difficulty. This means that a pixel's value has little influence on the values of pixel in distant locations, since there is always a way to stitch the patches together so the distant pixel values fit together. So the same assumption lies beneath Efros and Leung's algorithm, patch-based algorithms, and many others. These methods assume that each pixel value only has a local influence. This assumption is valid for many textures, but not valid for many shapes used into modeling as shown in 2.9(b),

2.10(b), and 2.11. So texture synthesis algorithms can not easily be extended to work on 3D models, not because they are three-dimensional, but because models are structured differently from textures.

Model synthesis is focused on a slightly different problem from texture synthesis. Both methods attempt to generate an output that resembles an input, but the goal of model synthesis is concentrated more on the local structure. At first, this may seem counter-intuitive. While the goal of model synthesis is more local than texture synthesis, the algorithm itself introduces a global search for conflicts. However, this seeming contradiction is explained by Figure 2.8 which demonstrates that it is necessary to look outside the local neighborhood even to satisfy a local constraint. Model synthesis is focused on ensuring that each model piece fit together seamlessly with its immediate neighbors.

Chapter 3

Discrete Model Synthesis

In this chapter, we first given a formal definition of the model synthesis problem in Section 3.1. Then we analyze the problem and discuss how many solutions exist in Section 3.2. In Section 3.3, we introduce an algorithm for solving the problem. In Section 3.4, we show results from the algorithm and in Section 3.5, we discuss related problems that can be solved with the algorithm.

3.1   Problem Definition

Model Definition   Discrete models are represented as three-dimensional arrays of labels where each label corresponds to a model piece. The algorithm uses two discrete models: the input example model E and the output model M. Each model has a finite length, width, and height. Let $n_x \times n_y \times n_z$ be the size of the output M and $n'_x \times n'_y \times n'_z$ be the size of the input E. Every point within the bounds of the model maps to a particular label. Each label is represented by an integer. Let K be the set of possible labels in the input and output models and k be the number of labels in K, $k = |K|$. The set K typically consists of every integer from 0 to $k - 1$. The input and output models are mappings between a point within their bounds to a label $E, M : \mathbb{Z}^3 \rightarrow K$. The models are functions that return which set of objects are located at each point.

Let $\hat{\imath}, \hat{\jmath}$, and $\hat{k}$ be unit vectors in the x, y and z directions respectively.

Consistency Definition   The model M is *consistent* with E, if for all points $\mathbf{x} \in \mathbb{Z}^3$ within M and for all axis-aligned unit vectors $\hat{\mathbf{d}} \in \{\hat{\imath}, \hat{\jmath}, \hat{k}\}$, there exists a point $\mathbf{x}' \in \mathbb{Z}^3$ within E such that

$$
\begin{aligned}
M(\mathbf{x}) &= E(\mathbf{x}') \\
M(\mathbf{x} + \hat{\mathbf{d}}) &= E(\mathbf{x}' + \hat{\mathbf{d}}).
\end{aligned}
\tag{3.1}
$$

The primary goal of model synthesis is to generate a model M is consistent with E. For a given input E, this set of equations 3.1 acts as a constraint on M and is called the *adjacency constraint*.

The adjacency constraint can be expressed in a slightly different form that is often more convenient. This expression uses three binary $k \times k$ matrices T_x, T_y, and T_z which are called transition matrices. Let b and c be two labels, $0 \le b, c < k$. The transition matrices T_x, T_y, and T_z are defined as

$$
\begin{aligned}
T_x[b, c] &= \begin{cases} 1, & \exists \mathbf{x}' | E(\mathbf{x}') = b \text{ and } E(\mathbf{x}' + \hat{\imath}) = c \\ 0, & \text{otherwise} \end{cases} \\[2mm]
T_y[b, c] &= \begin{cases} 1, & \exists \mathbf{x}' | E(\mathbf{x}') = b \text{ and } E(\mathbf{x}' + \hat{\jmath}) = c \\ 0, & \text{otherwise} \end{cases} \\[2mm]
T_z[b, c] &= \begin{cases} 1, & \exists \mathbf{x}' | E(\mathbf{x}') = b \text{ and } E(\mathbf{x}' + \hat{k}) = c \\ 0, & \text{otherwise} \end{cases}
\end{aligned}
\tag{3.2}
$$

The adjacency constraint is equivalent to the statement that for all points $\mathbf{x} \in \mathbb{Z}^3$ within M

$$T_x[M(\mathbf{x}), M(\mathbf{x} + \hat{\imath})] = 1$$

$$\wedge T_y[M(\mathbf{x}), M(\mathbf{x} + \hat{\jmath})] = 1 \tag{3.3}$$

$$\wedge T_z[M(\mathbf{x}), M(\mathbf{x} + \hat{k})] = 1.$$

These equations assume that the models are three-dimensional, but nearly the same set of equations could be applied to two-dimensional models. With 2D models, the z coordinate can be ignored and we can set $n_z = 1$. A few examples of 2D models are illustrations in Figure 3.1. 2D models are easier to illustrate and visualize on paper, so 2D models are often used to illustrate properties of full 3D model synthesis. Fortunately, many of the properties of model synthesis are identical for two, three, and higher-dimensions. However, one-dimensional model synthesis is often the exceptional case. One-dimensional model synthesis does not share many of the properties of higher-dimensional version as discussed in Sections 3.3.5 and 5.2.

(a) Empty Space Model, E_0 (b) Checkerboard Model, E_1 (c) Line Model, E_2 (d) Non-Self Similar Model, E_3
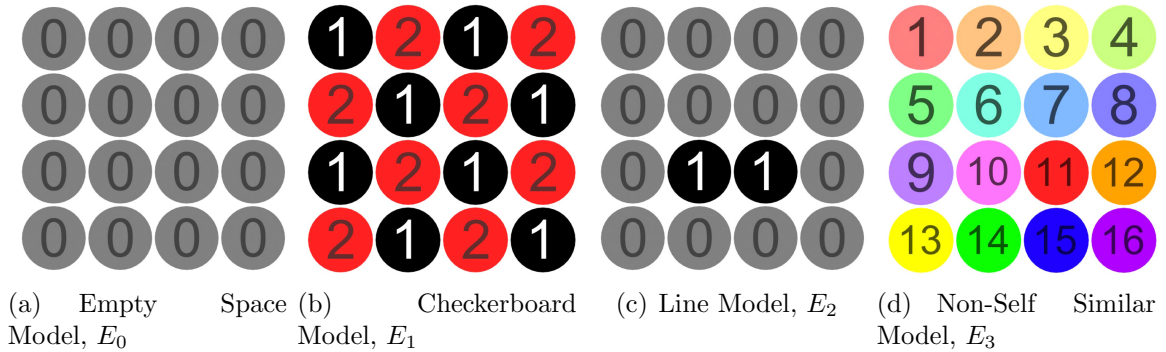
Figure 3.1: Examples of two-dimensional models

3.2   Bounds on the Number of Consistent Solutions

Definition For a given input model E, let $D_E(n_x, n_y, n_z)$ be the total number of models M consistent with E of size $n_x \times n_y \times n_z$.

This section discusses how $D_E(n_x, n_y, n_z)$ varies with n_x, n_y, and n_z. Many examples are given in 2D. In these cases, we assume that $n_z = 1$.

The function $D_E(n_x, n_y, n_z)$ could be zero. If the input model E_3 in Figure 3.1(d) is used, $D_{E_3}(5, 5, 1) = 0$. No consistent models larger than E_3 exist because none of the labels in E_3 repeat. The input model E_3 is not self-similar at all. This demonstrates why self-similarity is so important to model synthesis. Without self-similar input models, no consistent solution except the original model exists. The model E_3 is an unusual model because it does not contain any empty space. Empty space labels are adjacent to themselves in all directions. If the label 0 represents empty space, then $T_x(0, 0) = T_y(0, 0) = T_z(0, 0) = 1$. If E contains a labels that is adjacent to itself in all directions, then $D_E(n_x, n_y, n_z) > 0$ since a model that contains only this label is consistent.

Sometimes $D_E(n_x, n_y, n_z)$ is a constant nonzero value that does not depend on n_x, n_y, and n_z. For example, the empty space model E_0 in Figure 3.1(a), is consistent only with another empty space model $D_{E_0}(n_x, n_y, 1) = 1$ for all n_x and n_y. The checkboard model E_1 in Figure 3.1(b), is consistent only with two models, i.e. $D_{E_1}(n_x, n_y, 1) = 2$.

For many input models, $D_E(n_x, n_y, n_z)$ increases exponentially with n_x, n_y, and n_z. Suppose that E contains some empty space that is labeled zero. A set of points H is *enclosed by empty space* if every point adjacent to H, but not inside H is labeled zero. There are sets of points enclosed by empty space in Figures 3.1(c) and 3.2.

Theorem 3.2.1. If the input model E contains a set of points H that is enclosed by empty space and H has a length, width, and height of $h_x \times h_y \times h_z$, then $D_E(n_x, n_y, n_z) = \Omega\left(2^{\frac{n_x n_y n_z}{h_x h_y h_z}}\right)$.
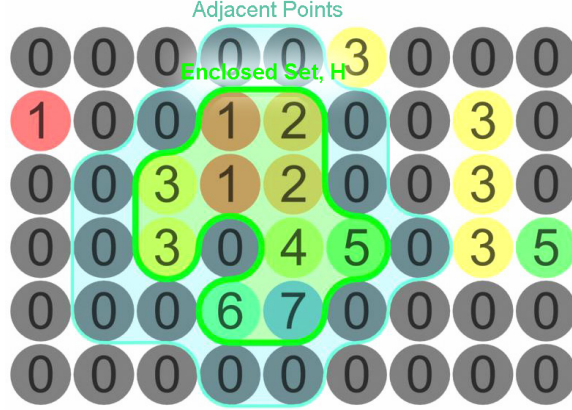
Figure 3.2: A set of points H is enclosed by empty space. Every point adjacent to H is labeled zero and the zero label is adjacent to itself in all directions $T_x(0,0) = T_y(0,0) = 1$.

Proof. A completely empty model that contains only empty space is consistent. If the model is empty except for a copy of the set H copied, it is also consistent. H could be copied into an otherwise empty model many times. As long as the copies do not overlap or touch, the generated model is consistent. A pair of copies needs only one row or one column of empty space separating them. An $n_x \times n_y \times n_z$ model M can contain $\lfloor \frac{n_x}{h_x+1} \rfloor \lfloor \frac{n_y}{h_y+1} \rfloor \lfloor \frac{n_z}{h_z+1} \rfloor$ copies of H. If some of these copies were excluded from M as shown in Figure 3.3, M would still be consistent. In fact, M is consistent whether or not each copy is included M. Therefore, simply by including or excluding particular copies, $2^{\lfloor \frac{n_x}{h_x+1} \rfloor \lfloor \frac{n_y}{h_y+1} \rfloor \lfloor \frac{n_z}{h_z+1} \rfloor}$ different models consistent with E can be constructed. $D_E(n_x, n_y, n_z) = \Omega\left(2^{\frac{n_x n_y n_z}{h_x h_y h_z}}\right).$ $\qquad\square$

Remark: For the purposes of this proof, a set of points was copied and pasted in different ways to create different consistent models. But it would be a mistake to conclude that this is the only way that the output models can vary. The output models can combine different parts of the input model E in much more complicated and interesting ways. Also, there probably exists a tighter lower bound on $D_E(n_x, n_y, n_z)$ than Theorem 3.2.1.

Figure 3.3: A model that satisfies the adjacency constraint can be created by copying and pasting a set of points H enclosed by empty space. Each copy can be included or excluded independently. This model is large enough to contain 8 copies, so at least $2^8 = 256$ model consistent with Figure 3.2 exist.

Theorem 3.2.2. $D_E(n_x, n_y, n_z) \leq k^{n_x n_y n_z}$.

Proof. Each point has k possible labels and there are $n_x n_y n_z$ points. □

3.3 The Discrete Model Synthesis Algorithm

3.3.1 Overview

The goal of model synthesis is to generate a model M that is consistent with E. In our algorithm, M is generated by assigning a label to each point individually. An *assignment* is a pairing $(\mathbf{x}, b)$ of a point $\mathbf{x}$ and a label b. The generated model M changes over time as these assignments are added. Let M_t be the model M at a given time step t. At each time step, a single assignment is added to M. For example, if the label b was assigned to the point $\mathbf{x}'$ at time t, then M would change from $M_t(\mathbf{x}') = -1$ to $M_{t+1}(\mathbf{x}') = b$.

Initially, every point in M is unlabeled. If $\mathbf{x}$ is an unlabeled point, then $M(\mathbf{x}) = -1$. So initially, $M_0(\mathbf{x}) = -1$ for every point $\mathbf{x}$. Labels are assigned until every point is labeled. When every point is labeled M is *complete*. A complete model is consistent with E, if it satisfies the adjacency constraint in Equation 3.3. An incomplete model is *consistent*, if it can be completed so that it satisfies the adjacency constraint. More formally, an incomplete model M is consistent if there exists a consistent and complete model M' such that for every point $\mathbf{x}$, $M(\mathbf{x}) \neq -1 \Rightarrow M(\mathbf{x}) = M'(\mathbf{x})$.

Every time a label is assigned, there is a risk that the assignment may cause M_t to become inconsistent. This risk could be eliminated if we could construct a catalog of possible assignments to add to M. The catalog $C_M^\star$ is a catalog that stores exactly which labels can be assigned to M without causing M to become inconsistent. We define $C^\star$ as

$$C_{M_t}^\star(\mathbf{x}, b) = \begin{cases} 0, & M_t(\mathbf{x}) \text{ is unassigned and if } M_{t+1}(\mathbf{x}) \text{ is set to } b, M_{t+1} \text{ is inconsistent} \\ 1, & M_t(\mathbf{x}) \text{ is unassigned and if } M_{t+1}(\mathbf{x}) \text{ is set to } b, M_{t+1} \text{ is consistent} \\ 0, & M_t(\mathbf{x}) \text{ is assigned and } M_{t+1}(\mathbf{x}) \neq b \text{ or } M_t \text{ is inconsistent} \\ 1, & M_t(\mathbf{x}) \text{ is assigned and } M_{t+1}(\mathbf{x}) = b \text{ and } M_t \text{ is consistent} \end{cases}$$

$$(3.4)$$

The assignment $(\mathbf{x}, b)$ is in the catalog if $C^{\star}_{M_t}(\mathbf{x}, b) = 1$. If we assign only labels that are in the catalog $C^{\star}_{M_t}$, then M_t will remain consistent until it has been completed. Each time a label is assigned the catalog may need to be updated. So the overall strategy of our algorithm is to pick a point, assign the point a label from the catalog, then update the catalog, and repeat until M is complete. The algorithm is described in more detail in Algorithm 3.1.

Algorithm 3.1 begins by counting the number of distinct labels k in the input E in the function FindK and by computing the transition matrices according to Equation 3.3 in the function FindTransitionMatrices. The catalog initially contains every possible assignment $(\mathbf{x}, b)$, so lines 3-5 set every value in the catalog to 1. The main loop (lines 6-18) goes through every point in M, selects a label b from the catalog (line 9), assigns b to the output M (line 13), and then updates the catalog (line 14).

Unfortunately, for some inputs E computing $C^{\star}$ is NP-hard. This is shown in Section 3.3.5. Therefore, it is not always possible to compute $C^{\star}$ in polynomial time unless $P = NP$. So we introduce in Section 3.3.2 an alternative catalog called C. The catalog C can be computed more easily than $C^{\star}$ (Section 3.3.3), but the catalog C is imperfect. It may or may not be equal to the ideal catalog $C^{\star}$ may or may not be equal. Several cases where they are equal are discussed in Section 3.3.7 and several cases where they are not are discussed in Section 3.3.4. If C is not equal to the ideal catalog $C^{\star}$, there is a chance that M_t may become inconsistent. If M_t is inconsistent, eventually C will be become empty, $C_{M_t}(\mathbf{x}, b) = 0$ for all assignments $(\mathbf{x}, b)$, and the failure case will be returned (line 11). In order to handle the possible failure cases, we introduce several changes to the algorithm in Section 3.3.6.

3.3.2 The Catalog C

The catalog C is an imperfect approximation to the ideal catalog $C^{\star}$. Computing $C^{\star}$ is NP-hard (Section 3.3.5), so C is introduced as an alternative that is easier to compute.

Algorithm 3.1 Discrete Model Synthesis Algorithm

Input: An Example Model, E, and an output size $n_x \times n_y \times n_z$
Output: A synthesized model M satisfying the adjacency constraint

1: $k \leftarrow$ FindK(E) // Count the number of labels
2: $T \leftarrow$ FindTransitionMatrices(E) // Compute the Transition Matrices

3: **for** all points of $\mathbf{p}, b$ **do** // Include all assignments in the catalog
4:     $C[\mathbf{p}, b] \leftarrow 1$
5: **end for**

6: **for** $p_x = 1$ to n_x **do** // Loop through every point
7:   **for** $p_y = 1$ to n_y **do**
8:     **for** $p_z = 1$ to n_z **do**
9:       **if** $C[\mathbf{p}, b] = 0$ for all b **then** // Check if the catalog is empty
10:         **return** failure
11:       **else**
12:         Select any value of b for which $C[\mathbf{p}, b] = 1$ at random
13:         $M[\mathbf{p}] \leftarrow b$
14:         $C \leftarrow$ UpdateC$(C, \mathbf{p}, b, T, k)$
15:       **end if**
16:     **end for**
17:   **end for**
18: **end for**

19: **return** M

The problem of finding consistent models is an example of a contraint satisfication problem. The model synthesis problem is similar to many well-known problems such as Boolean satisfiability and Sudoku. These problems are often solved by assigning values to some of the variables, quickly testing if is possible to complete the solution, and then backtracking if necessary. Model synthesis is solved similarly by assigning values to some points in M and then quickly testing if it is possible to complete M by checking if the catalog C_M is empty. Some limited backtracking may be necessary as is discussed in Section 3.3.6.

Each point has neighbors in the positive and negative x, y, and z directions, $\pm\hat{\imath}, \pm\hat{\jmath}$, and $\pm\hat{k}$ which makes six neighbors in total. The adjacency constraint applies to all six neighbors. Given a set of possible labels at any point $\mathbf{x}$, our goal is to determine which labels could be assigned to its neighbor $\mathbf{x} + \mathbf{d}$ where $\mathbf{d} \in \{\pm\hat{\imath}, \pm\hat{\jmath}, \pm\hat{k}\}$. Suppose that

b and c are labels, $\mathbf{x}$ is the labeled b, and $\mathbf{x} + \mathbf{d}$ is labeled c. To determine if b and c satisfy the adjacency constraint, one of the transition matrices is used. The constraint is satisfied if $T[b, c] = 1$ where T is the appropriate transition matrix based on the direction $\mathbf{d}$. If $\mathbf{d} = \hat{\imath}$, then $T = T_x$. If $\mathbf{d} = -\hat{\imath}$, then $T = T_x^T$ since when the matrix is transposed the roles of b and c are switched. If $\mathbf{d}$ is equal to $\hat{\jmath}, -\hat{\jmath}, \hat{k}$, or $-\hat{k}$, then T is equal to T_y, T_y^T, T_z, or T_z^T respectively.

The catalog C contains a list of acceptable labels at each point. The label c is only acceptable at $\mathbf{x} + \mathbf{d}$, if there exists a label b that is acceptable at point $\mathbf{x}$ meaning $C_{M_t}(\mathbf{x}, b) = 1$ and that can be adjacent to c meaning $T[b, c] = 1$.

$$C_{M_t}(\mathbf{x} + \mathbf{d}, c) = 1 \Rightarrow \exists b | C_{M_t}(\mathbf{x}, b) = 1 \text{ and } T[b, c] = 1 \tag{3.5}$$

Its contrapositive is given as:

$$\nexists b | C_{M_t}(\mathbf{x}, b) = 1 \text{ and } T[b, c] = 1 \Rightarrow C_{M_t}(\mathbf{x} + \mathbf{d}, c) = 0. \tag{3.6}$$

Additionally, we know that only one label may occupy a given point:

$$M_t(\mathbf{x}) = b \text{ and } c \neq b \Rightarrow C_{M_t}(\mathbf{x}, c) = 0. \tag{3.7}$$

Statement 3.6 is a direct consequence of the adjacency constraint and Statement 3.7 expresses an occupancy constraint. The catalog $C_{M_t}(\mathbf{x}, b)$ is defined as the binary function that maximizes $\sum_{\mathbf{x}} \sum_b C_{M_t}(\mathbf{x}, b)$ while satisfying Statements 3.6 and 3.7. Statements 3.6 and 3.7 are also true for the ideal catalog $C^\star$.

Algorithm 3.2 describes in detail how C is computed. Figure 3.4 shows an example of this computation. Figure 3.4(a) shows the example model. Suppose the size of M is $n_x \times n_y = 4 \times 4$. Initially, M_0 is empty and C_{M_0} contains all four possible labels in each of the 4×4 positions. Suppose that a '1' label is assigned to a point in M as shown in Figure 3.4(b). That point is now reserved exclusively for the '1' label. No

other labels may occupy that point according to Statement 3.7. So the other labels are removed from the catalog as shown in Figure 3.4(c). Figures 3.4(c-f) show the catalog C in various stages while as it is being computed. Labels are repeatedly removed from C_{M_t} according to Statement 3.6. Each time a label is removed, the algorithm checks if other labels need to be removed. The point that is currently being checked is marked 'v'. The other points that need to be checked are marked 'u' in Figures 3.4(c-f). Eventually, every point is checked and every assignment that violates Statements 3.6 and 3.7 is removed. The result is shown in Figure 3.4(f). We prove that Statements 3.6 and 3.7 always hold in Theorem 3.3.1.

(a) Example Model, E (b) Incomplete Model M_t (c) C_{M_t} as it is computed

(d) C_{M_t} (e) C_{M_t} (f) Final value of C_{M_t}
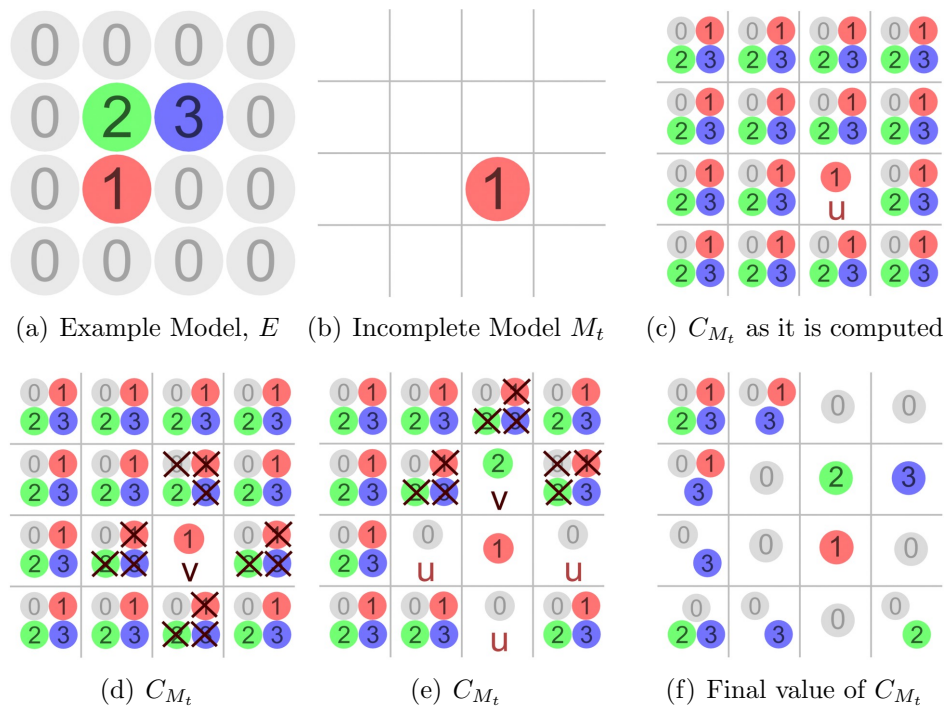
Figure 3.4: For the example model (a) and the incomplete model M_t(b), a catalog of possible assignments to make is calculated in several steps (c-f).

Theorem 3.3.1. Algorithm 3.2 updates the function $C_{M_t}(\mathbf{x}, b)$ so that Statements 3.6 and 3.7 are true when Algorithm 3.2 returns.

Proof. Statement 3.7 holds due to lines 5-9 of Algorithm 3.2. These lines are executed

after every new assignment in line 13 of Algorithm 3.1 since immediately afterwards Algorithm 3.2 called. Lines 5-9 guarantee that Statement 3.7 holds.

We use induction on the time step of the algorithm to prove that Statement 3.6 holds. The inductive hypothesis is that at each step Statement 3.6 is true at point $\mathbf{x}$, if $\mathbf{x}$ is not in the stack u. The basis case is the first time the algorithm is executed. Initially, none of the points in the model are labeled and $C_{M_t}(\mathbf{x}, b) = 1$ for all assignments $(\mathbf{x}, b)$, so the left side of the condition in Statement 3.6 is always false in the basis case. Assuming the inductive hypothesis is true before each line is executed, it can only become false if $C_{M_t}(\mathbf{x}, b)$ changes or if $\mathbf{x}$) is not in the stack u. $C_{M_t}(\mathbf{x}, b)$ changes only in line 7 of Algorithm 3.2 and in line 4 of Algorithm 3.3, but in both cases u has already been pushed onto the stack u immediately before $C_{M_t}(\mathbf{x}, b)$ changes. So the inductive hypothesis is true, since $\mathbf{x}$ is on the stack. The point $\mathbf{x}$ is only popped off the stack on line 17 of Algorithm 3.2. Lines 11-16 guarantee that Statement 3.6 holds. Each of the six lines 11-16 imposes Statement 3.6 on the point and each of its six neighbors by calling Algorithm 3.3. Algorithm 3.3 directly uses Statement 3.6 in Line 2. Algorithm 3.2 does not return until the stack u is empty. When Algorithm 3.2 returns, Statement 3.6 is satisfied at every point in C_{M_t}. $\qquad\square$

In Theorem 3.3.1, we showed that Algorithm 3.2 computes C so that it satisfies Statements 3.6 and 3.7. Next, we prove that every label removed by Statements 3.6 and 3.7 should be removed because it does not belong in $C^\star$.

Theorem 3.3.2. For any assignment $(\mathbf{x}, b)$

$$C_{M_t}(\mathbf{x}, b) = 0 \Rightarrow C^\star_{M_t}(\mathbf{x}, b) = 0 \tag{3.8}$$

Proof. We prove this by induction. The inductive hypothesis is that Statement 3.8 is true at every step of the algorithm. The basis case is the initial state of the algorithm.

Algorithm 3.2 UpdateC($C, \mathbf{p}, b, T, k$)
___
Input: A 4D array of possible labels C, a point $\mathbf{p} = (p_x, p_y, p_z)$, a label b, a set of transition matrices $T = \{T_x, T_y, T_z\}$, and the number of distinct labels k.

Output: The 4D array C is updated to reflect assigning label b to point $\mathbf{p}$.

1: push($u, \mathbf{p}$) // u is a stack of points to update.

2: **for** $c = 0$ to $k - 1$ **do** // Since label b is assigned to $\mathbf{p}$, remove all other labels.
3:    **if** $c \neq b$ **then**
4:       $C[\mathbf{p}, c] \leftarrow 0$
5:    **end if**
6: **end for**

7: **while** u is not empty **do** // Update the six closest neighbors
8:    $\mathbf{v} \leftarrow$ pop(u)
9:    $(C, u) \leftarrow$ UpdateNeighbor($C, u, k, \mathbf{v}, \hat{\imath}, T_x$)
10:   $(C, u) \leftarrow$ UpdateNeighbor($C, u, k, \mathbf{v}, -\hat{\imath}, T_x^T$)
11:   $(C, u) \leftarrow$ UpdateNeighbor($C, u, k, \mathbf{v}, \hat{\jmath}, T_y$)
12:   $(C, u) \leftarrow$ UpdateNeighbor($C, u, k, \mathbf{v}, -\hat{\jmath}, T_y^T$)
13:   $(C, u) \leftarrow$ UpdateNeighbor($C, u, k, \mathbf{v}, \hat{k}, T_z$)
14:   $(C, u) \leftarrow$ UpdateNeighbor($C, u, k, \mathbf{v}, -\hat{k}, T_z^T$)
15:   $u[\mathbf{v}] = 0$
16: **end while**

17: **return** C
___

Initially, $C_{M_t}(\mathbf{x}, b)$ is set to 1 for all assignments, so Statement 3.8 is always true, since the left side of the conditional is always false. Assuming by induction that Statement 3.8 is true, it could become false only when $C_{M_t}(\mathbf{v} + \mathbf{d}, c)$ is set to zero on line 4 of Algorithm 3.3. Line 4 is executed only if line 2 determines that adding the assignment $(\mathbf{v} + \mathbf{d}, c)$ would cause M to become inconsistent. Line 2 determine this because in order to be consistent, $(\mathbf{v} + \mathbf{d}, c)$ must have an adjacent assignment $(\mathbf{v}, b)$ for which two criteria are met: (1) $T[b, c] = 1$ and (2) $C_{M_t}(\mathbf{v}, b) = 1$. Criterion (1) follows directly from the adjacency constraint. Criterion (2) follows from the inductive hypothesis, since $C_{M_t}(\mathbf{v}, b) = 0 \Rightarrow C_{M_t}^{\star}(\mathbf{v}, b) = 0$, the assignment $(\mathbf{v} + \mathbf{d}, c)$ cannot rely on the assignment $(\mathbf{v}, b)$ if using $(\mathbf{v}, b)$ would cause M_t to become inconsistent. $\quad\square$

Algorithm 3.3 UpdateNeighbor($C, u, k, \mathbf{v}, \mathbf{d}, T$)
_____
Input: A 4D array of possible labels C, a stack u recording which points need updating, the number of distinct labels k, two vectors $\mathbf{v}$ and $\mathbf{d}$, and a transition matrix T.
Output: The array C and the stack u are updated properly at the point $\mathbf{v} + \mathbf{d}$.

1: **for** $c = 0$ to $k - 1$ **do** // Check if each label c belongs in the catalog at $\mathbf{v} + \mathbf{d}$
2:    **if** $C[\mathbf{v} + \mathbf{d}, c] = 1$ and $\nexists b | C[\mathbf{v}, b] = 1$ and $T[b, c] = 1$ **then**
3:      push$(u, \mathbf{v} + \mathbf{d})$
4:      $C[\mathbf{v} + \mathbf{d}, c] \leftarrow 0$
5:    **end if**
6: **end for**

7: **return** (C, u)
_____

3.3.3 Time and Space Complexity

Theorem 3.3.3. Algorithm 3.1 takes $\Theta(k^2 n_x n_y n_z)$ time and $\Theta(k n_x n_y n_z)$ space.

Proof. There are $n_x n_y n_z$ points each with k possible labels, so there are $k n_x n_y n_z$ possible assignments. If the algorithm is successful, one assignment is left at each point. If it fails no assignments are left. So line 7 of Algorithm 3.2 and line 2 of Algorithm 3.3 are both executed at least $(k - 1) n_x n_y n_z$ times. Each line is executed at most $k n_x n_y n_z$ times. Line 2 of Algorithm 3.3 checks k different labels each time.

The only data structures that require substantial amounts of memory are the arrays C and u. The array C requires $k n_x n_y n_z$ bits and the array u requires $n_x n_y n_z$ bits. $\square$

3.3.4 Failure Cases

Theorem 3.3.2 shows that $C_{M_t}(\mathbf{x}, b) = 0 \Rightarrow C^{\star}_{M_t}(\mathbf{x}, b) = 0$, but the converse may not be true. It is possible that $C^{\star}_{M_t}(\mathbf{x}, b) = 0$ and $C_{M_t}(\mathbf{x}, b) = 1$. C_{M_t} may contain an assignment $(\mathbf{x}, b)$ which does not belong in the ideal catalog, i.e. $C^{\star}_{M_t}$. This is demonstrated in the example shown in Figure 3.5. In the input model shown in Figure 3.5(a), every '2' label has a '6' label two spaces to its right. In order for M to be consistent, every '2' label in M must be connected to a '6' label two spaces to its right. What makes this case particularly interesting is that the '2' and the '6' label can be connected through two

possible paths.

Figure 3.5(b) shows an incomplete model M_t which includes a '2' label. Two spaces to the right of the '2' label, only the '6' label is present in the ideal catalog $C^\star_{M_t}$ (Figure 3.5(c)), but C_{M_t} (Figure 3.5(c)) contains other labels there such as the '1' label. The '1' label is included in C_{M_t} because this label satisfies Statements 3.6 and 3.7. This is illustrated by Figures 3.5(h) and 3.5(i) with show two paths of assignments satisfying Statement 3.6 connecting from the '2' label to the '1' label. Since the '1' label is in C_{M_t}, it could be selected at the next time step. If it is selected, the computations shown in Figures 3.5(e-g) will occur. In Figure 3.5(e), the labels '3' and '7' are above the label '2', but in Figure 3.5(f) both of these labels are eliminated from two different directions which eliminates every possible label there. Once every possible label at a point have been eliminated, the model is clearly inconsistent and the catalog eventually becomes completely empty (Figure 3.5(g)). An empty catalog is recognized as a failure on line 10 of Algorithm 1.

If M_t becomes inconsistent, then eventually the catalog C_{M_t} will become empty. Sometimes the catalog does not become empty until several more labels are added. In the example shown in Figure 3.5, C_{M_t} became empty when M_t became inconsistent. If this always happened, then failures could be handled simply by backtracking one time step whenever they occurred. But C_{M_t} does not always become empty when M_t becomes inconsistent as is illustrated by Figure 3.6. In this case, there is no clear indication that M_t has become inconsistent. When the algorithm eventually fails, it is hard to know where the error occurred or how far it is necessary to backtrack.

The example model in Figure 3.6(a) contains a pair of '2' labels connected by a path. In order to be consistent, M must contain pairs of '2' labels connected by paths that are two spaces wide. The incomplete model M_t in Figure 3.6(b) is inconsistent because it contains three '2' labels and there is no way to group these three labels into pairs. But C_{M_t} is not empty as shown in Figure 3.6(c). C_M will become empty after some time
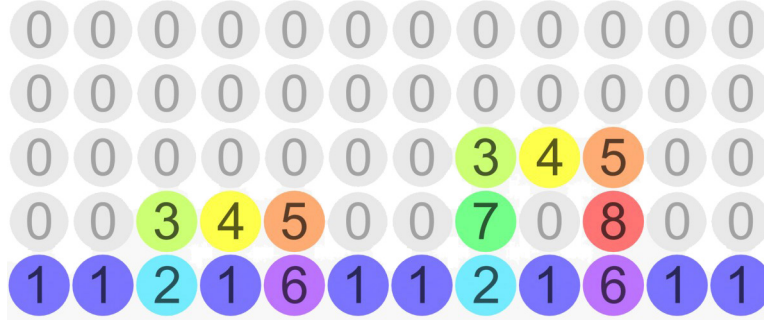
(a) Example Model, E
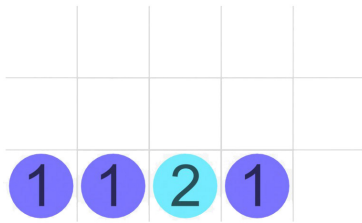
(b) Incomplete Model M_t
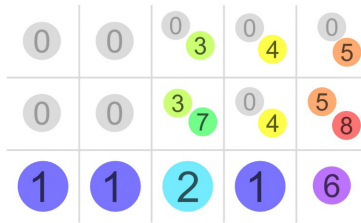
(c) $C^{\star}_{M_t}$
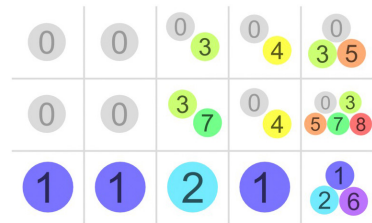
(d) C_{M_t}
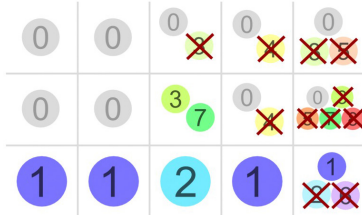
(e) $C_{M_{t+1}}$
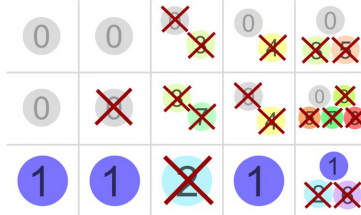
(f) $C_{M_{t+1}}$
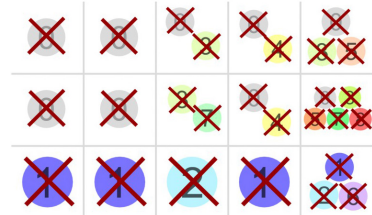
(g) $C_{M_{t+1}}$ Failure
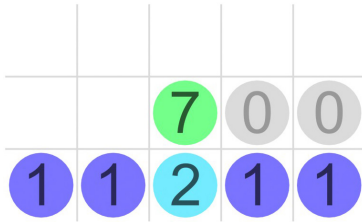
(h) A path satisfying Statement 3.6 exists between two inconsistent labels.
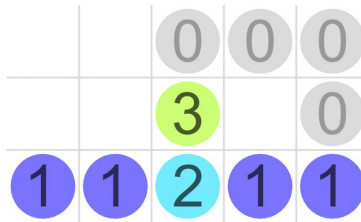
(i) A path satisfying Statement 3.6 exists between two inconsistent labels.

Figure 3.5: Example showing that C_{M_t} is imperfect. $C_{M_t} \neq C^{\star}_{M_t}$. For the example model E (a), the incomplete model M_t (b), has an ideal catalog $C^{\star}_{M_t}$ (c) and an imperfect catalog C_{M_t} (d). The '1' label in the bottom right corner should not be added to M. If it is added, the resulting computation of $C_{M_{t+1}}$ is shown (e-g). Eventually, $C_{M_{t+1}}$ becomes empty (g). The presence of the '2' label eliminates any possibility of a '1' label in the bottom right corner, but the '1' label is not removed from C_{M_t} because there are paths satisfying Statement 3.6 between the '1' and the '2' label (h & i).

(a) Example Model, E
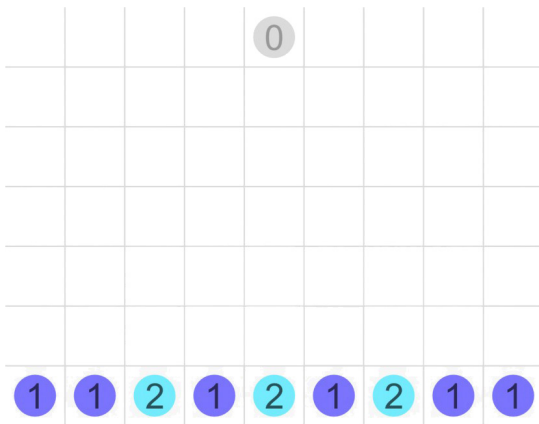
(b) Incomplete Model, M_t
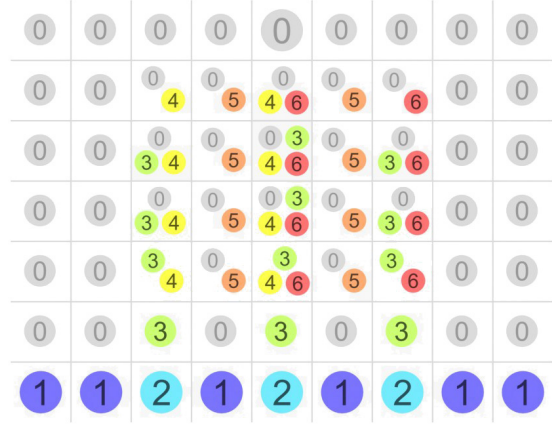
(c) C_{M_t}

Figure 3.6: Example showing that the algorithm may not fail immediately after M_t becomes inconsistent. For the example model E (a), the model M_t is inconsistent (b). To be consistent, the '2' labels must come in pairs, but there are only three points with '2' labels. Since M_t is inconsistent, eventually the catalog C_M will become completely empty, but this does not happen immediately (c).

steps, but when it does, it may not be obvious when the error occurred that caused the model to first become inconsistent [1].

3.3.5 Computing $C^\star$ is NP-hard

Theorem 3.3.4. Deciding if an incomplete model M_t is consistent is NP-complete.

---

[1]One possible objection to the example in Figure 3.6(b) is that the labels have not been assigned in scan line order since the top row contains a '0' label. But it would not be difficult to construct a more complicated example that would illustrate the same result in scan line order.

Proof. This problem is in NP since a consistent model could be guessed and then verified in polynomial time by checking the adjacency constraint at every point. To show that the problem is NP-hard, we reduce a known NP-complete problem the Planar 3-SAT problem to it. The Planar 3-SAT problem is to decide if it is possible to satisfy a Boolean formula written in conjunctive normal form that has three literals per clause and that can be drawn as a planar graph. An example is shown in Figure 3.7(a). The literals are connected by wires into three-input OR gates. One of the three inputs must have a true value for the Boolean formula to be satisfied.

In order to reduce the Planar 3-SAT problem to a model consistency problem, we construct a model like the one in Figure 3.7(c) which resembles the planar graph. Each literal and the wires coming out of each literal are enclosed by a group of labels. The transition matrices T are chosen so that all the labels inside the enclosure must TRUE or FALSE and so that the TRUE and FALSE labels can never touch one another. To be consistent, everything inside each enclosure must be completely TRUE or completely FALSE. The wires may have NOT gates attached to them and the wires meet at OR gates.

With an appropriate set of transition matrices, we can create NOT gates and OR gates. A NOT gate is created by placing a particular label on the wire. This label only has two possible labels to its right which are shown as blue squares with arrows. The label with the arrow facing down has a TRUE label beneath it and a FALSE label above it. The second label with the arrow facing up has a FALSE label beneath it and a TRUE label above it. In both cases, the value on the wire is negated and so this functions as a NOT gate. An OR gate is created by placing a particular label where the three wires meet. This label always has one of three possible labels beneath it which are shown in yellow with arrows. For each of these possible pieces, the TRUE label must be found in the direction the arrow points. The other two directions may have either TRUE or FALSE labels. The model can be completed consistently only if a TRUE label is found
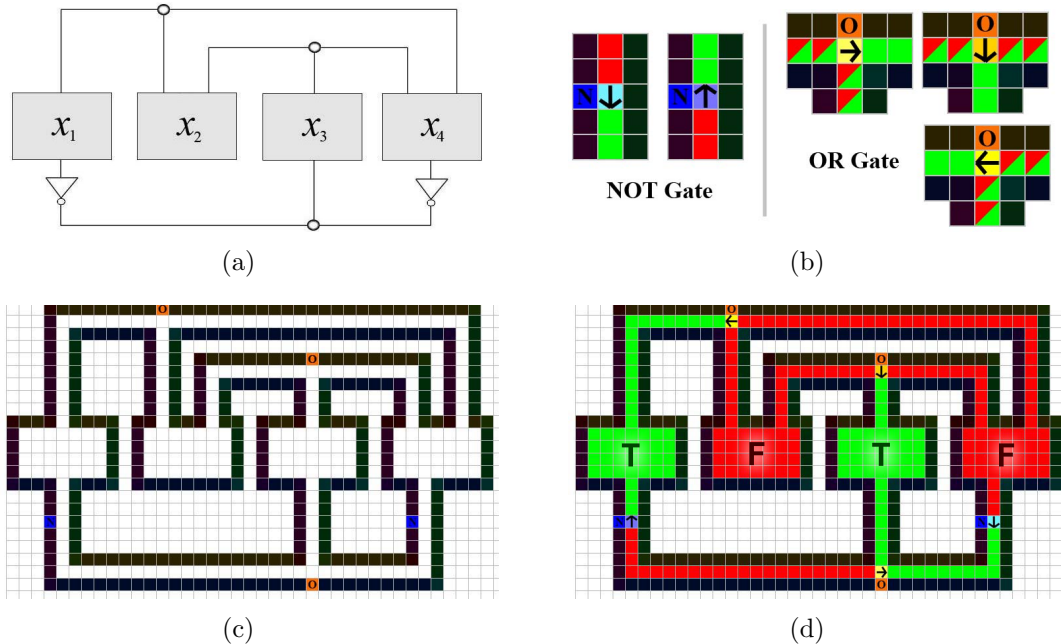
Figure 3.7: (a) A Planar 3-SAT Problem $(x_1 \vee x_2 \vee x_4) \wedge (x_2 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee x_2 \vee \neg x_4)$, (b) Possible configurations of a NOT and OR gate created from different labels, (c) An Equivalent model synthesis problem, (d) A model synthesis solution. Green = TRUE, Red = FALSE

at one of the three incoming wires.

A solution to the Planar 3-SAT problem exists if and only if it is possible to complete this 2D model such that all the OR gates have at least one TRUE value and the values are negated at the NOT gates. The Planar 3-SAT problem is reduced to the model consistency problem in polynomial time. □

Theorem 3.3.5. Computing $C^\star$ is NP-hard.

Proof. Computing $C^\star_{M_t}$ would immediately solve the NP-complete problem of deciding if M_t is consistent. Since M_t is consistent if and only if $C^\star$ is not empty. □

Remark: We intentionally prove these theorems using 2D models to show that they apply not only to model synthesis, but also to texture synthesis. This result easily extends to three and higher dimensional models, but not to one-dimensional models. For 1D models, $C^\star = C$, so it is not NP-hard to compute $C^\star$ since it can be computed

using Algorithm 3.2.

3.3.6   Modifying in Parts

In Section 3.3.4 have shown that the catalog C is imperfect and Theorem 3.3.5 shows that unless $P = NP$ the ideal catalog $C^\star$ can not be calculated in polynomial time. Because C is imperfect, Algorithm 3.1 may fail. To handle failures, we introduce several changes to the algorithm in this section

The difficulty of an NP-hard problem depends greatly on the size of the input. If the input is moderately large, an NP-hard problem may be intractable, but if the input size is tiny, these problems can easily be solved. In the case of model synthesis, the difficulty of the problem depends on the size of M_t, $n_x \times n_y \times n_z$. Deciding if M_t is consistent or not is intractable only when M_t is large. The size of M_t greatly affect the probability of failure in Algorithm 3.1 fails. Figure 3.8 shows the success rates of Algorithm 3.1 for several different input models as n_x and n_y are varied. The success rate can become quite small when M_t is large.

The relationship between the model size and the success rate suggests a strategy for handling failures. Rather than trying to generate a large model entirely in a single pass as Algorithm 3.1 does, the model could be created in small sections or blocks. We propose an algorithm that first finds an initial solution and then modifies that solution in small blocks. This strategy depending upon first finding at least one large initial model that is consistent with E, but such a model usually is easy to find. If E contains any empty space, then containing only empty space labels is consistent with E. While the empty space model is consistent, it is one of the least interesting consistent solutions. So the rest of the algorithm is designed to modify the current solution in small blocks.

Our algorithm is described in pseudocode in Algorithm 3.4 and Figure 3.9 shows an example of how the algorithm operates. First, an initial trivial solution is computed in lines 3-5 (Figure 3.9(a)). Then different parts of the solution are modified in overlapping

47
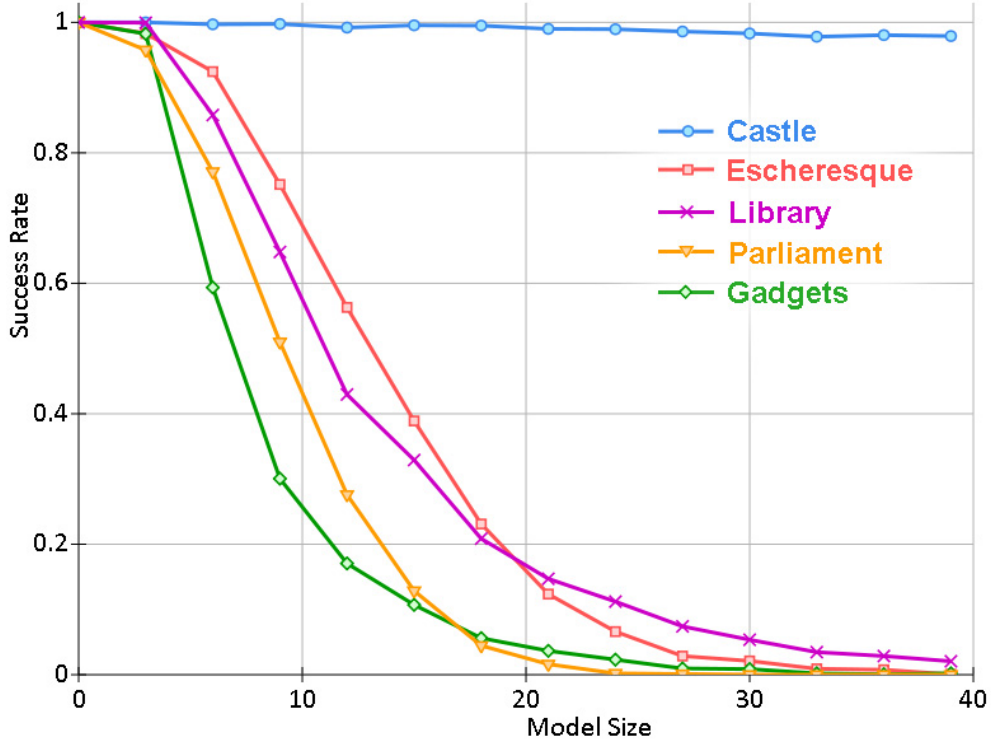
Figure 3.8: The success rate of Algorithm 3.1 for different example models as the model size varies. n_z is held constant at $n_z = 10$, while the values of n_x and n_y are equal to the x-coordinate. The example models are shown in Section 3.4

blocks. Let $m_x \times m_y \times m_z$ be the size of the block that is modified. Lines 6-8 cycle through every block. The start of the current block is determined by the value of $\mathbf{q} = (q_x, q_y, q_z)$. The function inBlock($\mathbf{p}, \mathbf{q}$) returns true if a point $\mathbf{p}$ is inside the current block given by $\mathbf{q}$. Lines 9 - 29 are almost exactly the same as lines 3 - 18 of Algorithm 3.1. The only different is that this algorithm does not run Algorithm 3.1 on all of M, but on a block of M. Like Algorithm 3.1, every assignment is initially included in the catalog (lines 9-11). Within the modifying block, random assignments are selected from the catalog (line 21) and then the catalog is updated (line 26). Outside of the block, the labels are not modified. The labels outside the block have already been determined. Only the labels inside the block change (Figure 3.9(b)). The labels inside the block must agree with the labels outside the block. So the labels outside the block are used to update

48

the catalog C in lines 12-17 (Figure 3.9(c)). Assignments are selected from the catalog just like Algorithm 3.1 (Figure 3.9(d)) and then the process is repeated on subsequent modifying blocks (Figure 3.9(e) and 3.9(f)).

If the block size $m_x \times m_y \times m_z$ is small, the method is likely to succeed, but it is still possible that during each iteration M_t might become inconsistent and that a failure may occur (line 23). If a failure occurs, the model M reverts back to its previous state before it was modified. So before modifying M, the values of M are stored in another array M_0 (line 13). If a failure occurs, M reverts back to M_0 on lines 31-33.

If the block size is equal to the model size $m_x \times m_y \times m_z = n_x \times n_y \times n_z$, then Algorithm 3.4 is equivalent to Algorithm 3.1.

(a) Initial Trivial Solution   (b) Pick a Block to Modify   (c) Compute C_M

(d) Select Labels from C_M (e) Pick a New Block, Compute C_M (f) Fill New Block
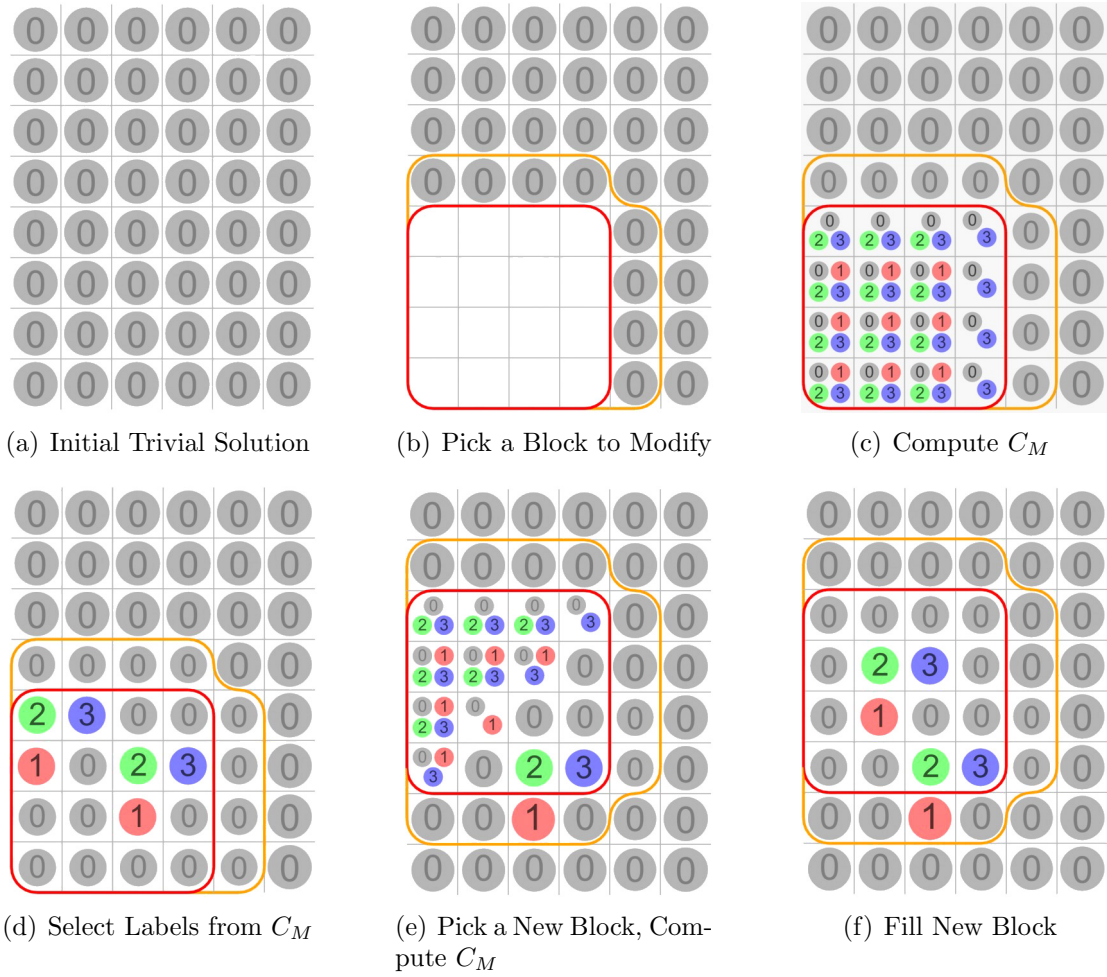
Figure 3.9: Example illustrating how parts of the model are modified. The example model is take from Figure 3.4(a). First, an initial trivial solution is found (a), then a block of this solution is modified. The labels in this block are removed (b), C_M is computed (c), and the labels are assigned to each point individually until every point has a label (d). Then a new block to modify is selected (e) and the process repeats (f).

Algorithm 3.4 Final Discrete Model Synthesis Algorithm

Input: An Example Model, E, and an output size $n_x \times n_y \times n_z$
Output: A synthesized model M satisfying the adjacency constraint

1: $k \leftarrow$ FindK(E) // Count the number of labels
2: $T \leftarrow$ FindTransitionMatrices(E) // Compute the Transition Matrices

3: **for** all points **p do** // Create Initial Solution
4:    $M[\mathbf{p}] \leftarrow 0$
5: **end for**

6: **for** $q_x = 0$ to $2n_x/m_x$ **do** // Loop through each block
7:   **for** $q_y = 0$ to $2n_y/m_y$ **do**
8:     **for** $q_z = 0$ to $2n_z/m_z$ **do**
9:       **for** all values of $\mathbf{p}, b$ **do**
10:         $C[\mathbf{p}, b] \leftarrow 1$
11:       **end for**

12:       **for** all points **p do** // Put every assignment in the catalog
13:         $M_0[\mathbf{p}] = M[\mathbf{p}]$ // Save the current value of M
14:         **if** not insideBlock($\mathbf{p}, \mathbf{q}$) **then** // Add everything outside the current block.
15:           $C \leftarrow$ updateC($C, \mathbf{p}, M[\mathbf{p}], T, k$)
16:         **end if**
17:       **end for**

18:       failed $\leftarrow 0$
19:       **for** all points **p** while not failed **do** // Run Algorithm 3.1 within the block.
20:         **if** insideBlock(**p**,**q**) **then**
21:           **if** $C[\mathbf{p}, b] = 0$ for all b **then** // Check if the catalog is empty
22:             failed $\leftarrow 1$
23:           **else**
24:             Select any value of b for which $C[\mathbf{p}, b] = 1$ at random
25:             $M[\mathbf{p}] \leftarrow b$
26:             $C \leftarrow$ UpdateC($C, \mathbf{p}, b, T, k$)
27:           **end if**
28:         **end if**
29:       **end for**

30:       **if** failed **then** // If M becomes inconsistent, restore its previous value
31:         **for** all points **p do**
32:           $M[\mathbf{p}] \leftarrow M_0[\mathbf{p}]$
33:         **end for**
34:       **end if**
35:     **end for**
36:   **end for**
37: **end for**

38: **return** M

Algorithm 3.5 inBlock($\mathbf{p} = (p_x, p_y, p_z), \mathbf{q} = (q_x, q_y, q_z)$)

Input: Two points $\mathbf{p}$ and $\mathbf{q}$

Output: True if $\mathbf{p}$ is inside the block with a corner at the point $\left(\frac{q_x m_x}{2}, \frac{q_y m_y}{2}, \frac{q_z m_z}{2}, \right)$

1: **return**

$$q_x m_x / 2 \leq p_x < q_x m_x / 2 + m_x \text{ and}$$
$$q_y m_y / 2 \leq p_y < q_y m_y / 2 + m_y \text{ and}$$
$$q_z m_z / 2 \leq p_z < q_y m_z / 2 + m_z$$

Block Size Trade Offs

Algorithm 3.4 has the same time complexity as Algorithm 3.1 which was shown in Section 3.3.3 to be $\Theta(n_x n_y n_z)$. Algorithm 3.4 improves upon Algorithm 3.1. Given the same inputs, Algorithm 3.1 often repeatedly fails to produce a large model, but Algorithm 3.4 can do it successfully. As the block size $m_x \times m_y \times m_z$ decreases, the failure rate also decreases, but this low failure rate comes with a trade off. Algorithm 3.1 might only rarely produce a consistent model, but there is always a tiny chance that a given model M consistent with E could be produced. Algorithm 3.4 might not be able to produce some consistent models. An example of this is illustrated in Figure 3.10. The example model E in Figure 3.10(a) contains a single rectangle. Figures 3.10(b) - 3.10(f) show how to produce a long thin rectangle by modifying several different blocks of the model in turn, but the rectangle cannot get much larger than this. Algorithm 3.4 cannot generate the large square in Figure 3.10(h) unless the block size is large since Figure 3.10(g)'s rectangle cannot be widened. As $m_x \times m_y \times m_z$ decreases the number of models that could be produced using Algorithm 3.4 may also decreases. As $m_x \times m_y \times m_z$ increases, the failure rate may increase. In the most extreme case, where $m_x = m_y = m_z = 1$, Algorithm 3.4 can not fail, but it also can not produce interesting models since it would only create models with empty space in them. In practice, a value of $m_x = m_y = m_z = 10$, often produces interesting models with a reasonably low failure rate. This value is used in most of our results.

Implementation Details

When implementing Algorithm 3.4, a few minor changes are recommended. It is not necessary to compute C_{M_t} across the entire model. It is only necessary to compute C_{M_t} within the modifying block plus its immediate neighbors. So the array C can be a relatively small $m_x + 2 \times m_y + 2 \times m_z + 2 \times k$ array rather than the $n_x \times n_y \times n_z \times k$ array used in Algorithm 3.4. Making C smaller saves memory and computation time.

(a) Example Model E (b) Initially Empty Model M about to be modified. (c) Block in red area is modified. (d) New block is about to be modified.

(e) Block in red is modified. (f) A long rectangle is produced after many modifications. (g) The rectangle can not be widened by modifying in the red block. (h) This large square can not be produced with a small red block.
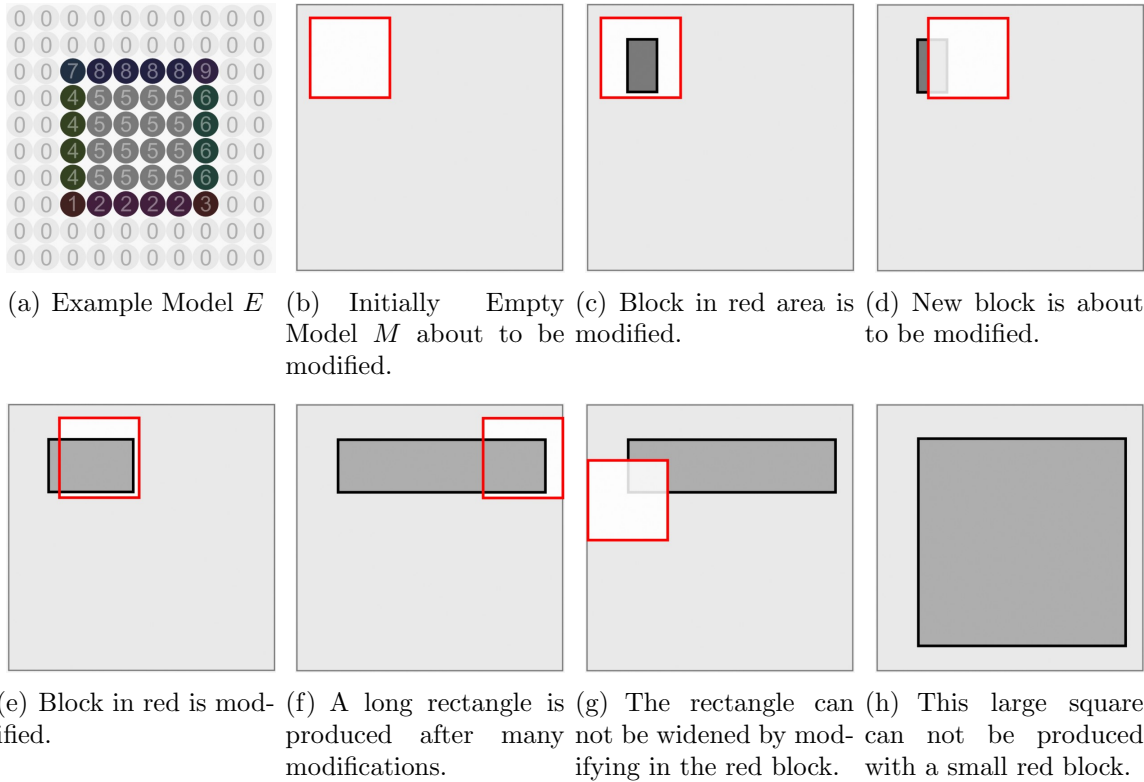
Figure 3.10: Example demonstrating that some consistent models cannot be produced if the block size is small. The example model is a rectangle (a). Different blocks can be modified in turn (b-f) to produce a long thin rectangle (f), but it cannot be widened (g) to produce the large square (h).

All the model synthesis algorithms create model satisfying the adjacency constraint. The adjacency constraint allows objects to simply stop at the boundaries of the model as shown in Figure 3.11. The models in Figure 3.11(b) and 3.11(c) are consistent with the model in Figure 3.11(a), but the objects they contain are cut off where the model ends. These models contain only part of a house. They may not contain a ground plane at the bottom (Figure 3.11(c)) or empty sky at the top (Figure 3.11(b)). All of these problems can be addressed by labeling the boundaries of the model in a preprocessing step. The bottom boundary is given ground plane labels and the top, left, right, front, and back boundaries of the model are labeled as empty space. With empty space and a ground plane along the boundaries of the model, the algorithm can only generate

complete objects as shown in Figure 3.11(d).

(a) Example Model, E (b) Possible Output M

(c) Possible Output M (d) Output with correct boundaries
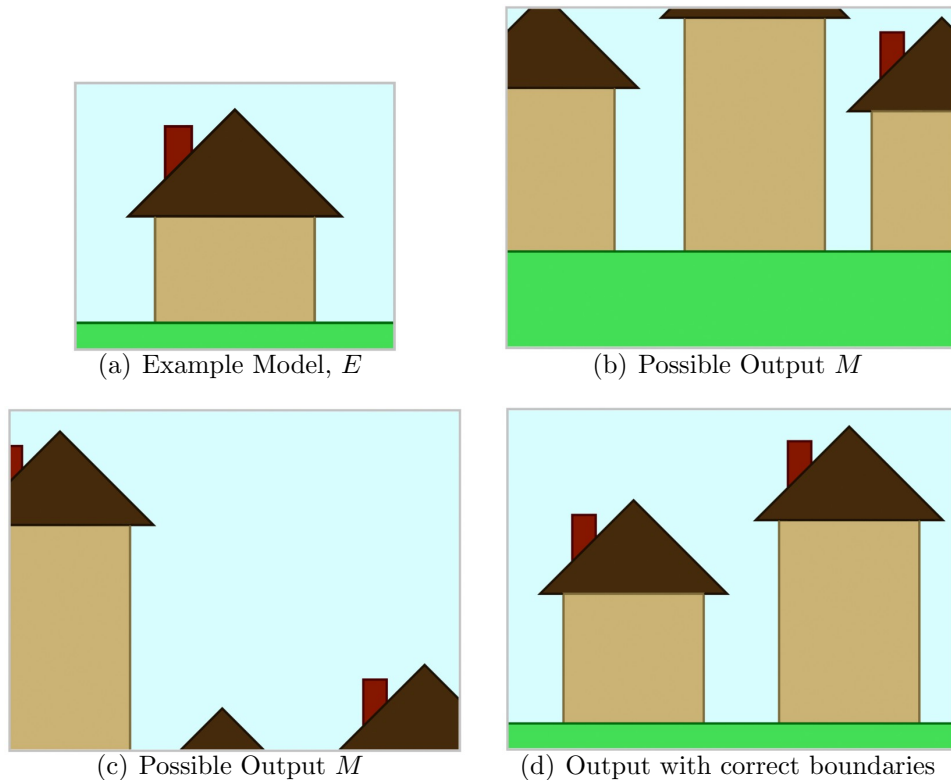
Figure 3.11: Example illustrating the problems that may occur at the boundaries of the model. The models in (b-d) are consistent with the example model in (a), but the houses could be cut off at the boundary of the model (b & c) and there may be no ground plane (c). These problems can be addressed by initially assigning proper values to the boundaries (d).

3.3.7   Infallible Cases where $C = C^\star$

Figure 3.8 shows that the success rate depends upon which input model E is used. In fact, Algorithms 3.1 and 3.4 never fail when given some input models. If Algorithm 3.1 cannot fail when given E, then E is called an *infallible* model. The models in Figures 3.1(a), 3.1(b), 3.1(c), and 3.4(a) are infallible. These models are all quite simple, but more complicated infallible example models are given in the results (Section 3.4). When E is infallible, Algorithm 3.2 computes the ideal catalog $C = C^\star$. The block size $m_x \times m_y \times m_z$ does not need to be small anymore to avoid failures. It should be as

large as possible so that the algorithm has a chance of producing every consistent model. Maximizing $m_x \times m_y \times m_z$ in Algorithm 3.4 is equivalent to simply using the original Algorithm 3.1.

It is much easier to prove that a fallible model is fallible than to prove that an infallible model is not. We can prove that a fallible model is fallible by finding a single failure. But we cannot prove that an infallible model is infallible by the absence of any observed failures since they may exist unobserved.

In order to prove that some models E are infallible, we introduce a new algorithm. It is described in Algorithm 3.6. We introduce Algorithm 3.6 because it allows us to prove that E is infallible in some cases. If Algorithm 3.6 does not fail on E then Algorithm 3.1 cannot fail either and by definition E is infallible. It is often easier to prove that Algorithm 3.6 does not fail than to prove that Algorithm 3.1 does not. We do not recommend using Algorithm 3.6 since it is less powerful than Algorithm 3.1. Algorithm 3.6 may fail on some input models E that Algorithm 3.1 would not.

Algorithm 3.6 Modified Model Synthesis Algorithm using the Regions R_i
___
Input: An Example Model, E, and an output size $n_x \times n_y \times n_z$
Output: A synthesized model M satisfying the adjacency constraint

Algorithm 3.6 is exactly the same as Algorithm 3.1 except line 14 can be omitted and line 9 should be changed to

$b \leftarrow \text{ComputeC\_Ri}(\mathbf{p}, M, \mathbf{r}, T, k)$

The only difference between Algorithm 3.1 and 3.6 is in how the catalog is computed. Algorithm 3.6 computes a catalog like C_{M_t}, but it does not use the entire model M_t. It only uses a small region R around the insertion point. R has a size of $r_x \times r_y \times r_z$ and the values it contains are copied directly from M_t

$$R(\mathbf{x}) = M(\mathbf{x} - \mathbf{x}') \tag{3.9}$$

for some point $\mathbf{x}'$. A catalog C_R can be compute using only the labels of M_t in R.

Algorithm 3.7 ComputeC_Ri($\mathbf{p}, M, \mathbf{r}, T, k$)

Input: A point $\mathbf{p} = (p_x, p_y, p_z)$, a model M, a region size $\mathbf{r} = (r_x, r_y, r_z)$, a set of transition matrices T, and the number of distinct labels k

Output: A label selected only by using a small $r_x \times r_y \times r_z$ region R_i of the model M.

 1: **for** all values of $\mathbf{v}, b$ **do** // Include all assignment into C.
 2:    $C[\mathbf{v}, b] \leftarrow 1$
 3: **end for**

 4: **for** $v_x = p_x - 1$ to $p_x + r_x - 1$ **do** // Update C only with the small region around v
 5:    **for** $v_y = p_y - 1$ to $p_y + r_y - 1$ **do**
 6:       **for** $v_z = p_z - 1$ to $p_z + r_z - 1$ **do**
 7:          **if** $M[\mathbf{v}] \neq -1$ **then**
 8:             $C \leftarrow \text{UpdateC}(C, \mathbf{v}, M[\mathbf{v}], T, k)$
 9:          **end if**
10:       **end for**
11:    **end for**
12: **end for**

13: **return** a randomly selected value of b for which $C[\mathbf{p}, b] = 1$

Algorithm 3.6 picks assignments from C_R rather than from C_{M_t}.

Theorem 3.3.6. If Algorithm 3.6 cannot fail when given input E and when $r_x, r_y, r_z \geq 2$, then E is infallible.

Proof. The catalog C_{M_t} only shrinks over time t as more assignments are added to M. $C_{M_t}(\mathbf{x}, k) = 0 \Rightarrow C_{M_{t+1}}(\mathbf{x}, k) = 0$. If assignments were removed from M then the catalog could only get larger. R is simply M_t with some of the assignments missing. Every assignment in R is in M_t, so C_R cannot be larger than C_{M_t} meaning that

$$C_{M_t}(\mathbf{x}, k) = 1 \Rightarrow C_R(\mathbf{x} + \mathbf{x}', k) = 1. \tag{3.10}$$

Lemma 3.3.1. *If, for a given E, $C_R(\mathbf{x} + \mathbf{x}', k) = 1 \Rightarrow C^\star_{M_t}(\mathbf{x}, k) = 1$ for all $\mathbf{x}, k$, and t, then E is infallible.*

Proof. $C_{M_t}(\mathbf{x}, k)$ has only two possible values. In either case, $C_{M_t}(\mathbf{x}, k) = C^\star_{M_t}(\mathbf{x}, k)$. From Statement 3.10 we know that, $C_{M_t}(\mathbf{x}, k) = 1 \Rightarrow C_R(\mathbf{x} + \mathbf{x}', k) = 1 \Rightarrow C^\star_{M_t}(\mathbf{x}, k) = 1$.

From Theorem 3.3.2 we know that $C_{M_t}(\mathbf{x}, k) = 0 \Rightarrow C^\star_{M_t}(\mathbf{x}, k) = 0$. So $C_{M_t}(\mathbf{x}, k) = C^\star_{M_t}(\mathbf{x}, k)$ which make E infallible. □

Algorithm 3.6 selects assignments from C_R. If the catalog C_R becomes empty for any region R, then Algorithm 3.6 fails. If Algorithm 3.6 does not fail, then C_R is not empty for any R region. If C_R is not empty, then within R the adjacency constraint is satisfied between adjacent assignments. Every pair of adjacent assignments is found in some region R if $r_x, r_y, r_z \geq 2$. If Algorithm 3.6 succeeds, then the generated model M satisfies the adjacency constraint and is consistent by definition. If Algorithm 3.6 cannot fail for input E no matter which assignments are selected at random from C_R, then $C_R(\mathbf{x}, k) = 1 \Rightarrow C^\star_{M_t}(\mathbf{x}, k) = 1$ since every assignment picked from C_R produces a consistent model. By Lemma 3.3.1, if Algorithm 3.6 cannot fail, then Algorithm 3.1 cannot fail either. □

We introduce Algorithm 3.6 because it is easier to prove that Algorithm 3.6 does not fail than that Algorithm 3.1 does not fail, because M_t can be any size and has an infinite number of possible values, but the region R has a small finite size. We can exhaustively list every possible value of R_i, if $r_x \times r_y \times r_z$ is small. But the list can be quite long even when $r_x \times r_y \times r_z$ is only moderately large. The length of the list depends on $D_E(r_x, r_y, r_z)$ which can grow exponentially according to Theorem 3.2.1. If $r_x \times r_y \times r_z$ is large, the list may become unmanageably long. To make the proof simpler, $r_x \times r_y \times r_z$ should be as small as possible (but never smaller than $2 \times 2 \times 2$). However, sometimes Algorithm 3.6 fails when $r_x \times r_y \times r_z$ is too small, but always succeeds when $r_x \times r_y \times r_z$ is slightly larger.

To prove that Algorithm 3.6 does not fail for a given E, we use induction on the time step t. The inductive hypothesis is that C_{R_i} is not empty for any region R_i at time step t. The goal is to prove that this is true at the next time step. Let us consider a specific example model. Let E be the example model shown in Figure 3.12 (it was previously shown in Figure 3.1(c)) and let $r_x \times r_y = 2 \times 2$. The model M_t and each of
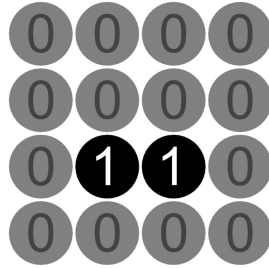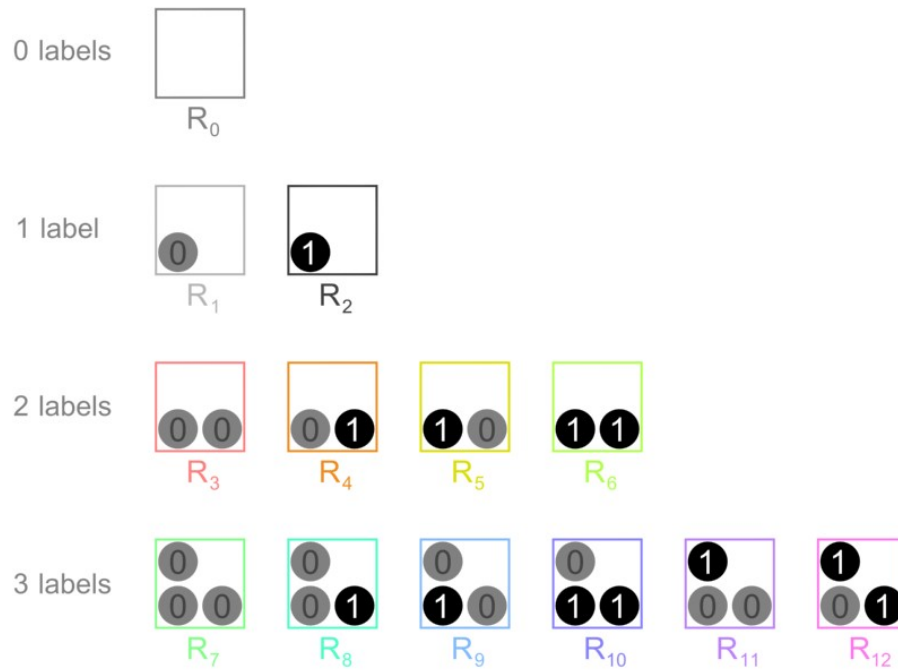
Figure 3.12: Line Example Model

the R_i regions are filled in scan line order from left to right and then bottom to top. Every possible region with three or fewer labels is shown below.

The region $R' = $ is not included since $C_{R'}$ is empty which would violate the inductive hypothesis. Let us consider the region R_9 in particular. We can prove that if we pick an assignment from C_{R_9} that none of the catalogs C_{R_i} become empty in the next time step. C_{R_9} is

Either the '0' or the '1' label could be selected at random from C_{R_9} and R_9 could have either R_3 or R_4 region to its right. So there are four cases to consider:
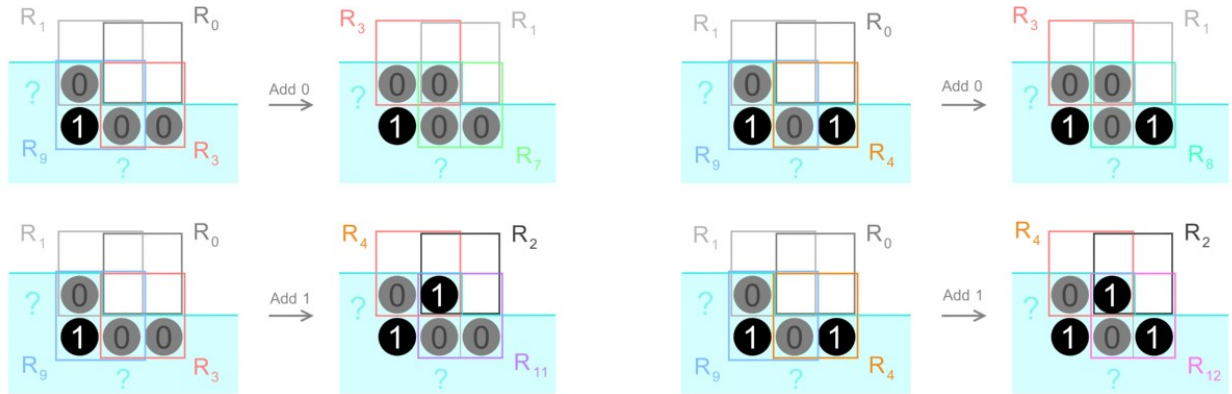
Figure 3.13: No matter what is added to a R_9 region, none of the neighboring catalogs become empty.

Figure 3.13 shows that no matter which assignment is chosen out of R_9 none of the catalogs C_{R_i} become empty and Algorithm 3.6 can not fail. But Figure 3.13 only checks the R_9 region, by verifying that Algorithm 3.6 does not fail on any of the regions $R_1, R_2, \ldots, R_{12}$, we can prove that Algorithm 3.6 never fails on the example E in Figure 3.12. The remainder of the proof is to enumerate all possible cases, but we omit the rest because it is long and tedious. It would be even longer if $r_x \times r_y \times r_z$ or k was larger. Fortunately these proofs can be automated.

In the previous example, we assumed that $r_x \times r_y = 2 \times 2$. When $r_x \times r_y = 2 \times 2$, the proof can be simplified. Algorithm 3.6 never fails if for all $a, b,$ and c

$$T_x(a,b) = 1 \text{ and } T_x(a,c) = 1 \Rightarrow \exists d | T_x(c,d) = 1 \text{ and } T_x(b,d) = 1 \qquad (3.11)$$

Equation 3.11 can be used to prove that the example models in Figures 3.1(a), 3.1(b), and 3.12 are infallible.

Unfortunately, Algorithm 3.6 does fail sometimes even when Algorithm 3.1 would not. If we use the example model from Figure 3.4(a) and if we assume that $r_x \times r_y = 2 \times 2$, Algorithm 3.6 might fail. It could fail when the following two regions are beside one another
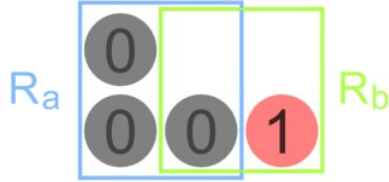
C_{R_a} is computed as
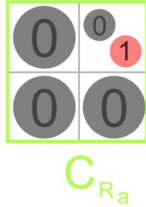
If the label '1' is added, the result is
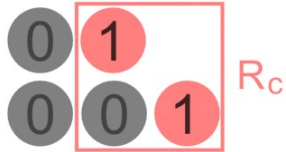
but C_{R_c} is empty since the bottom '1' label requires have a '2' label above it while the other '1' label requires a '0' label to its right. When C_{R_c} becomes empty, Algorithm 3.6 fails. Although Algorithm 3.6 may fail when $r_x \times r_y = 2 \times 2$, it cannot fail when $r_x \times r_y = 3 \times 2$. The 3×2 region R_d is shown below combines the R_a and R_b regions from the $r_x \times r_y = 2 \times 2$ case. C_{R_d} is computed as
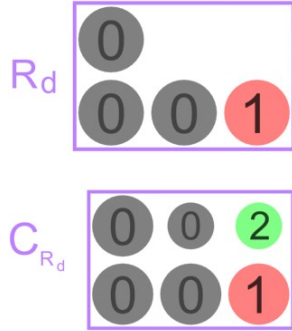
so when the $r_x \times r_y = 3 \times 2$, it is no longer possible to add a '1' label as it was in the $r_x \times r_y = 2 \times 2$ case. A full proof that Algorithm 3.6 does not fail on Figure 3.4(a) when $r_x \times r_y = 3 \times 2$ exists, but it is omitted because if its length.

3.3.8   Converting to an Infallible Model

The difference between an infallible model and a fallible model can be subtle. This is demonstrated by considering the example models in Figure 3.5(a) and Figure 3.3.8 which we will call E_F and E_I respectively. E_I is identical to E_F except E_F has a '0' label in place of the '9' label in E_I. We proved that E_F is fallible in Section 3.3.4 and we can show that E_I is infallible using Algorithm 3.6 with $r_x \times r_y = 2 \times 2$ or with Equation 3.11. We can convert between the two models using a function f define as
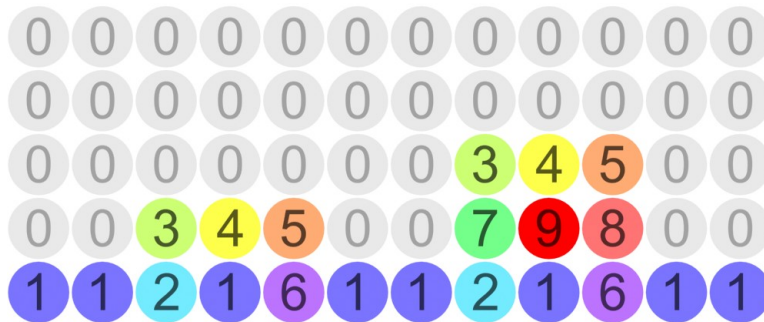
Figure 3.14: An infallible model E_I identical to the fallible model in Figure 3.5(a) except the '9' label is a '0' label in Figure 3.5(a).

$$f(b) = \begin{cases} 0, & b = 9 \\ b, & b \neq 9 \end{cases} \tag{3.12}$$

So $f(E_I(\mathbf{x})) = E_F(\mathbf{x})$ or this could be written simply as $f(E_I) = E_F$. The model synthesis algorithm generates similar models from both example models. In fact, the every model generated from E_I can be converted into a model generated from E_F using the function f.

$$\{M|M \text{ is consistent with } E_F\} = \{f(M)|M \text{ is consistent with } E_I\} \tag{3.13}$$

This suggests another strategy for handling fallible models. For any fallible model E_F, if we can find an infallible model E_I and a function f that satisfies Equation 3.13, then Algorithm 3.1 could be executed on E_I and the model it produces M could be converted to $f^{-1}(M)$ which is consistent with E_F. Algorithm 3.1 would never fail and it could produce every possible model consistent with the fallible model E_F.

Another example of applying this strategy is shown in Figures 3.6(a) and 3.3.8. Figure 3.6(a) shows a fallible model E_F and Figure 3.3.8 shows an infallible model E_I. The two models satisfy Equation 3.13 where

$$f(b) = \begin{cases} b, & b < 10 \\ b - 10, & b \geq 10 \end{cases} \tag{3.14}$$

While this strategy is effective on Figures 3.5(a) and 3.6(a), it may be difficult to apply this strategy more generally to every fallible model E_F. We do not know a general procedure for finding an infallible model satisfying Equation 3.13. We do not know there even exists an infallible model satisfying Equation 3.13 in every case. They exist for Figure 3.5(a) and 3.6(a), but they are two of the simplest fallible models. One may

Figure 3.15: An infallible model similar to the fallible model in Figure 3.6(a).

not exist for a more complicated example model such as the one behind Theorem 3.3.5. These are important topics for future research.

The success of this strategy for the models in Figures 3.5(a) and 3.6(a) demonstrate that Algorithms 3.1 and 3.6 can be improved although how exactly to do it is not clear except for these two specific models.

3.3.9 Summary

Model synthesis generates models that satisfy the adjacency constraint (Equation 3.1), which are called consistent models. The number of model consistent with a given example model E may grow exponentially number with the output size $n_x \times n_y \times n_z$ (Section 3.2). The model synthesis algorithm maintains a catalog C_{M_t} of possible labels that can be added into the model (Section 3.3.2). The catalog can be computed and every assignment can be selected from C_{M_t} in $\Theta(n_x n_y n_z)$ time. The catalog is imperfect because it may contain assignments that would cause the model to become inconsistent if they were used (Section 3.3.4). A perfect catalog cannot be computed in polynomial time unless $P = NP$ (Section 3.3.5). Model synthesis is less likely to fail when $n_x \times n_y \times n_z$ is small. Using this observation, we introduce a new algorithm that modifies an existing solution in small parts (Section 3.3.6). This new algorithm fails less, but may not be able

	Output Size $n_x \times n_y \times n_z$ pieces	# labels k	Time - Displayed Model (min)	Time (min) $40 \times 40 \times 10$
Parliament	$50 \times 50 \times 10 = 25K$	95	4.0	2.6
Castle	$50 \times 50 \times 10 = 25K$	62	0.6	0.4
Escheresque	$60 \times 60 \times 10 = 36K$	107	4.3	1.5
City at Night	$80 \times 80 \times 10 = 64K$	120	1.2	0.3
Canyon	$120 \times 60 \times 10 = 72K$	53	1.3	0.3
Forest	$50 \times 50 \times 12 = 30K$	33	0.5	0.3
Gadgets	$30 \times 30 \times 10 = 9K$	43	0.3	0.6
Library	$35 \times 40 \times 10 = 14K$	71	1.3	1.3

Table 3.1: The output model sizes, number of labels k, and computation times for each of the models shown in the results. To compare the computation times more easily, the last column shows computation times for a $40 \times 40 \times 10$ output model.

to generate every possible consistent model. We show the original algorithm (Algorithm 3.1) never fails on certain example models (Section 3.3.7). It is sometimes possible to use a model that never fails in place of a model that does (Section 3.3.8).

3.4 Results

Figures 3.16 through 3.23 show many different models each generated using a different example models, including cities with different architectural styles, plants, terrain, castles, and building interiors. The generated models are fairly large and would take a great amount of effort to model manually without model synthesis.

Table 3.1 shows for each results, the size of the output, the number of label k, and the computation time. Two columns of computation times are shown. The first column shows how much time was spent computing the model shown in Figure 3.16 - 3.23. However, the computation times are difficult to compare using this column since they depend on the output size which varies between each model. So we include a second column which shows how much time was spent generating a $40 \times 40 \times 10$ model.

Model synthesis takes less time when the input model is infallible as defined in Section 3.3.7. It generally takes less time when the success rate of the input E is high.

The success rate of Algorithm 3.1 is shown in Figure 3.8. Figure 3.8 does not graph the success rates of the canyon model, the city at night model, or the forest model because it would graph a perfect 100% success rate. No failures have ever been observed for the canyon or the city at night models even when generating huge models[2]. Since the canyon and the city at night never cause Algorithm 3.1 to fail, it was used to generate those results. The other models cause Algorithm 3.1 to fail, so Algorithm 3.4 was used with $m_x \times m_y \times m_z = 10 \times 10 \times 10$ to produce those results.

The computation times are a few minutes at the most which is relatively short and insignificant compared to the modeling process as a whole. Modeling is often a long and difficult. Even the example models, which are intended to be simple, each took several hours to create. Each model also needs textures to look realistic. Finding and applying good texture maps by itself can take a few hours. Finally, after the model has been generated, it needs to be rendered. Since each output model contains thousands of model pieces and the model pieces each contain many polygons, a high-quality rendering can also take a lot of time. The overall time spent involved in other tasks make running the model synthesis algorithm is trivial by comparison.

Model synthesis can also be used to light environments containing a large number of lights. This is done by including model pieces that have lights in them. In Figure 3.19, thousands of street lights and car lights were generated using model synthesis.

---

[2]Failures have been observed for the forest model, but only for large models $50 \times 50 \times 10$.
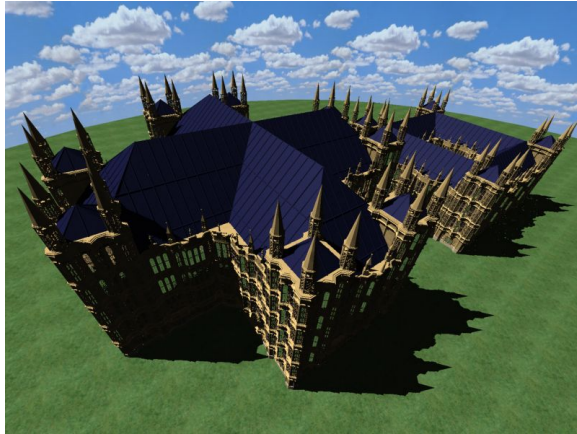
(a) Parliament Example Model
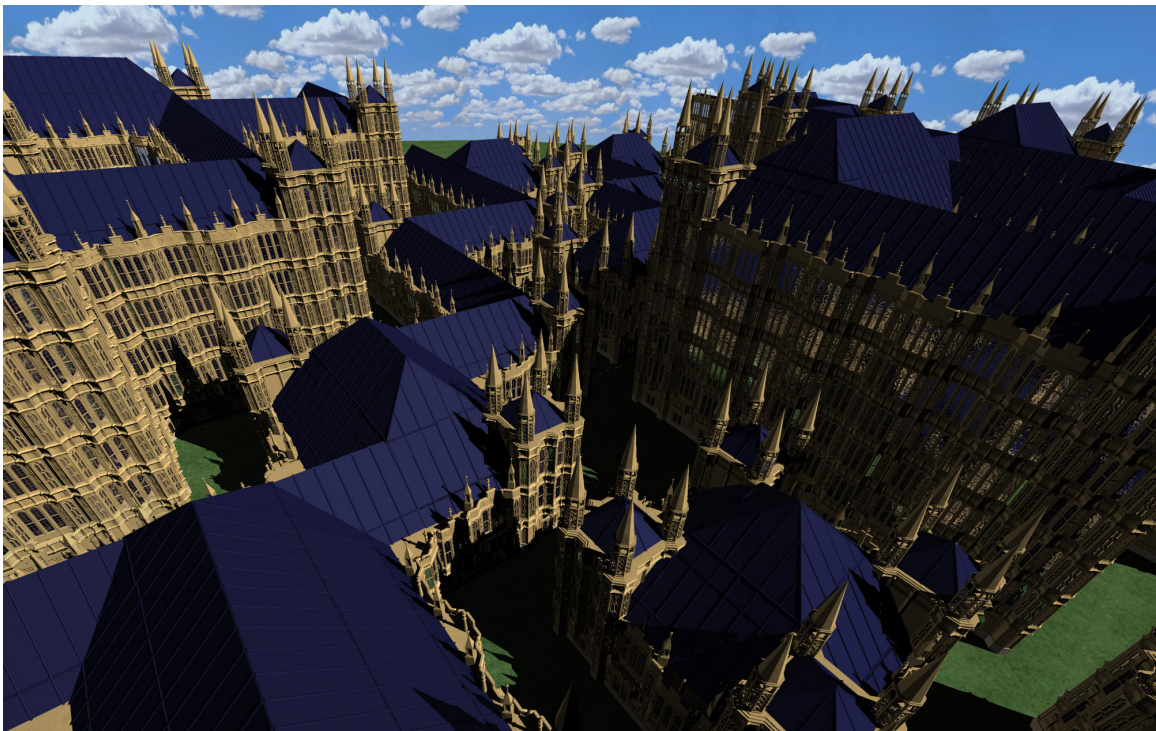
(b) Synthesized Model

Figure 3.16: Given two buildings (a), model synthesis produces a cluster of buildings (b).
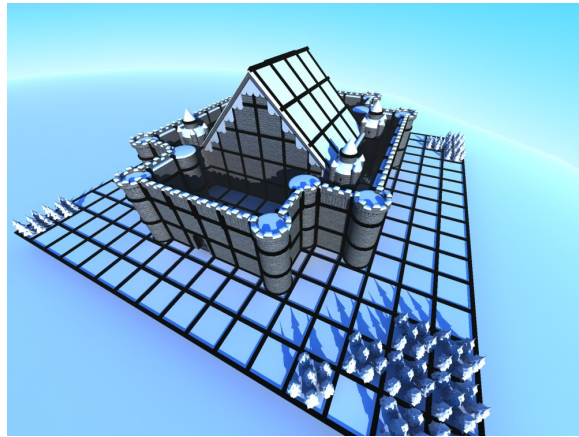
(a) Castle Example Model
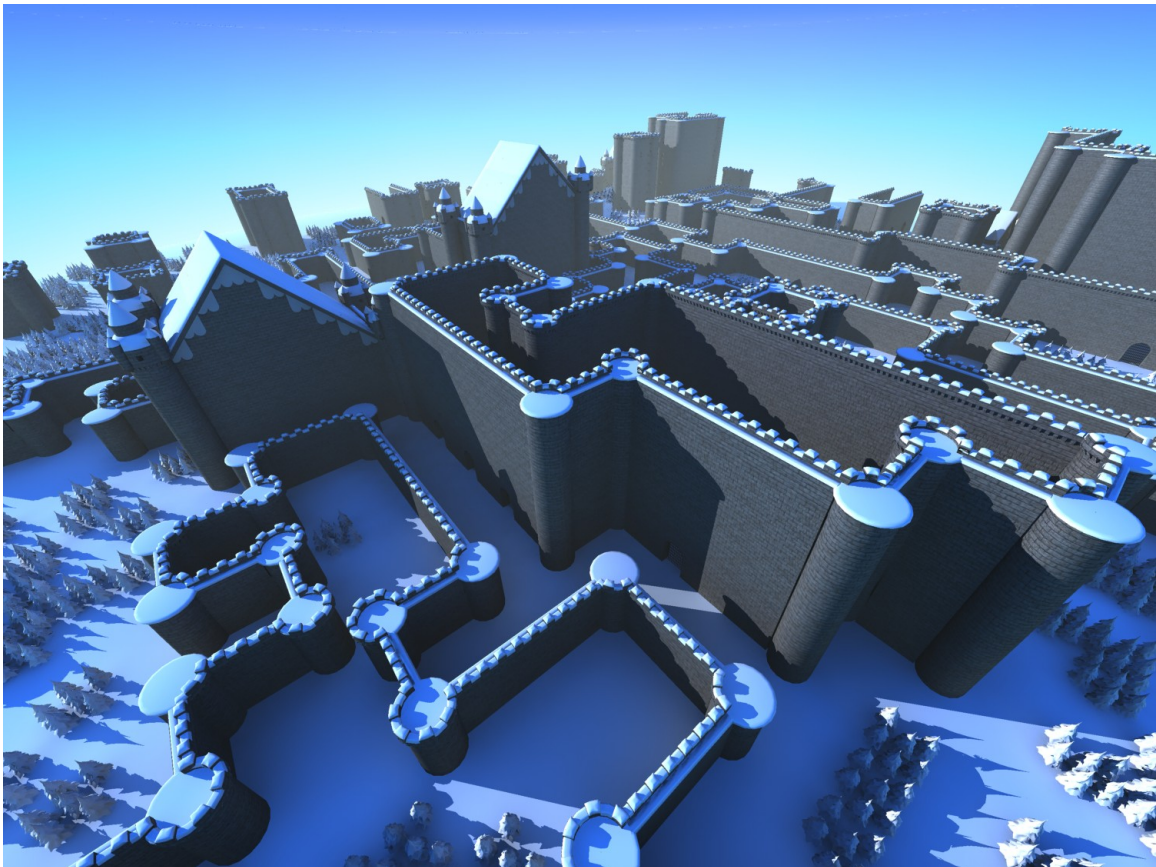
(b) Synthesized Model

Figure 3.17: Given a castle wall (a), model synthesis produces many fortifications (b). Grid lines are drawn in the example (a) to show how the model is divided into pieces.
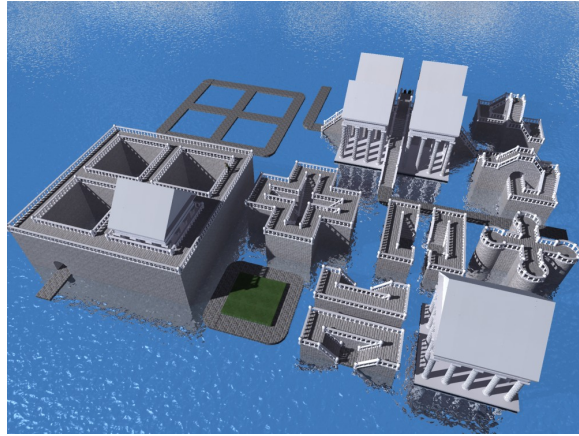
(a) Escheresque Example Model
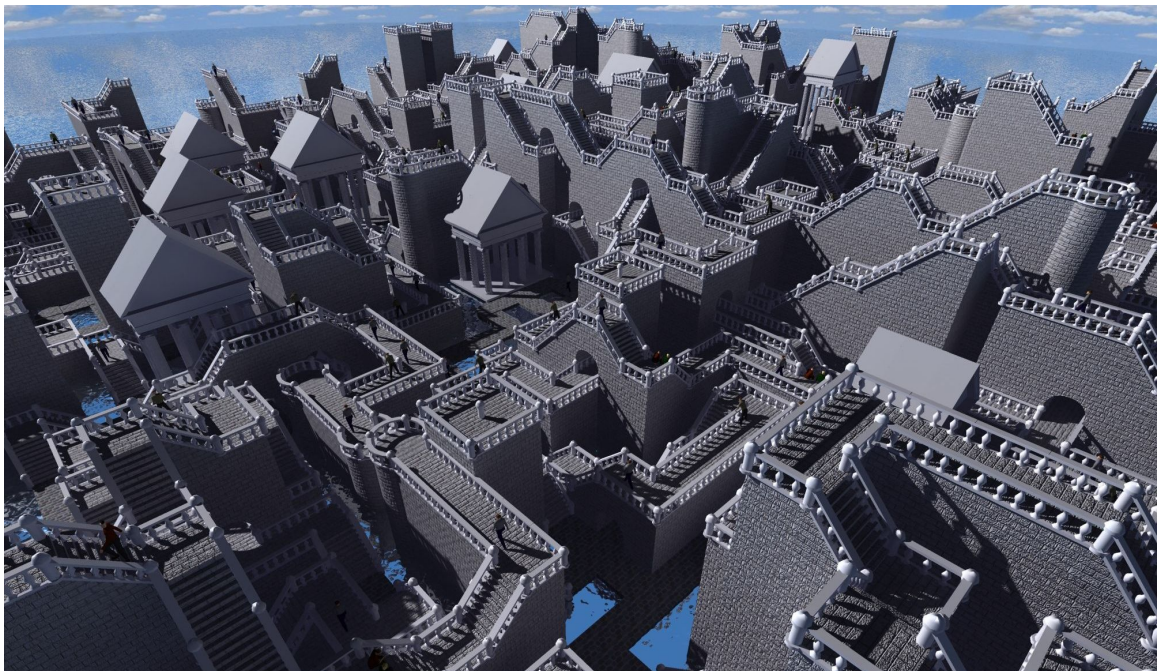
(b) Synthesized Example Model

Figure 3.18: Escheresque Result. Long windy paths are created through arches and over bridges.
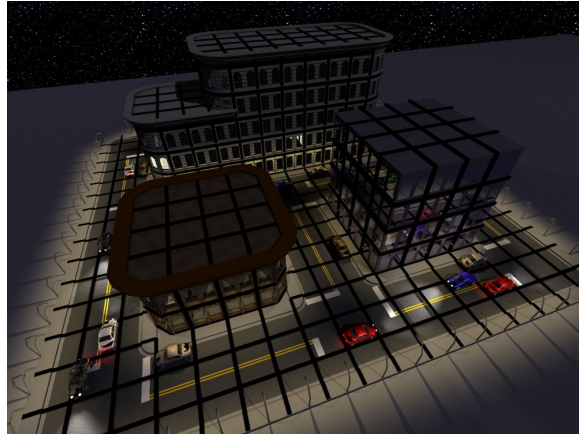
(a) City Example Model

(b) Synthesized Model

Figure 3.19: Given a few buildings (a), model synthesis produces a city (b). Grid lines are drawn in the example model. This input model is believed to be infallible. Unlike most of the other models, Algorithm 3.1 has never failed on this input model even for huge output models. In the other models, Algorithm 3.1 nearly always fails even on only moderately large output models. In addition to all of the geometric shapes, thousands of street lights and car light are generated automatically.
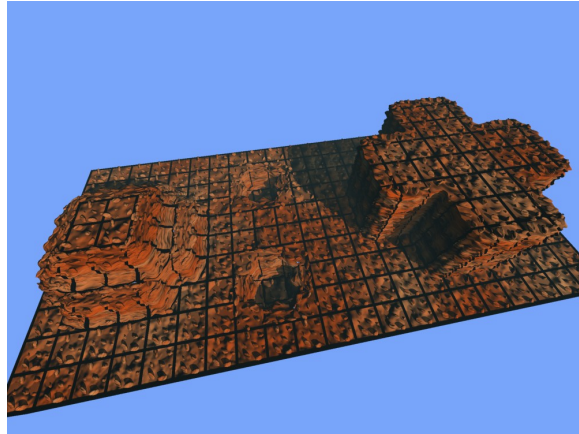
(a) Canyon Example Model
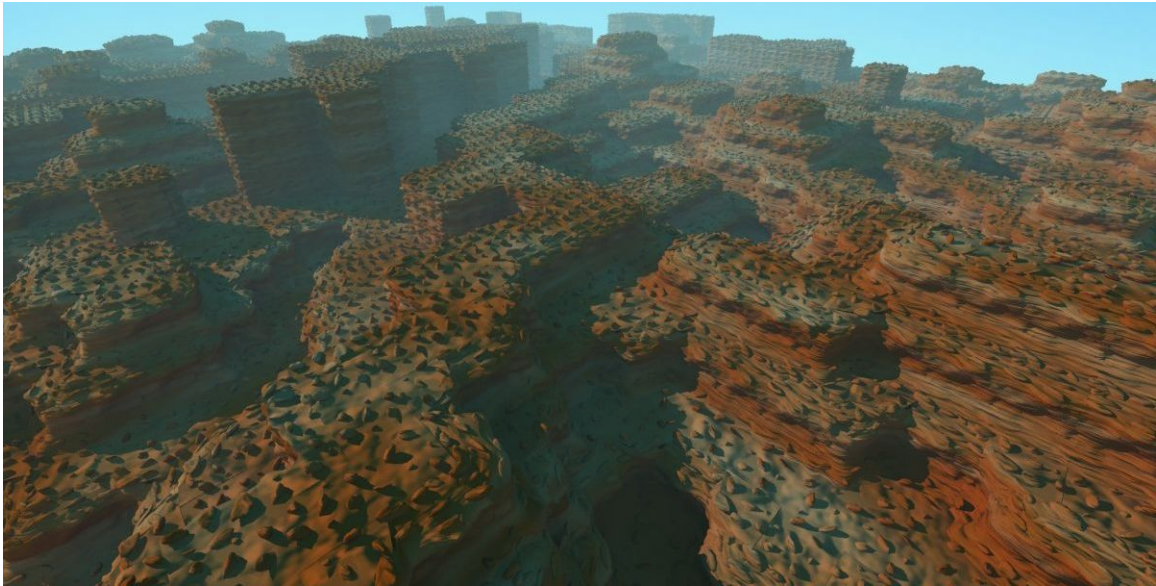
(b) Synthesized Model

Figure 3.20: Given a patch of land (a), model synthesis produces a landscape (b). Grid lines are drawn in the example model.
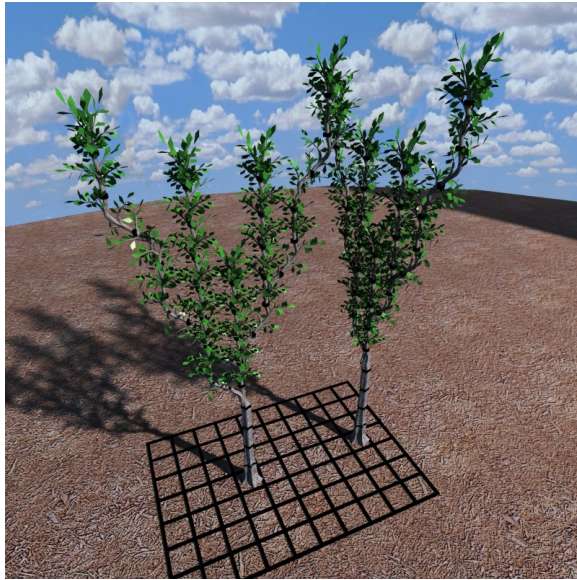
(a) Tree Example Model
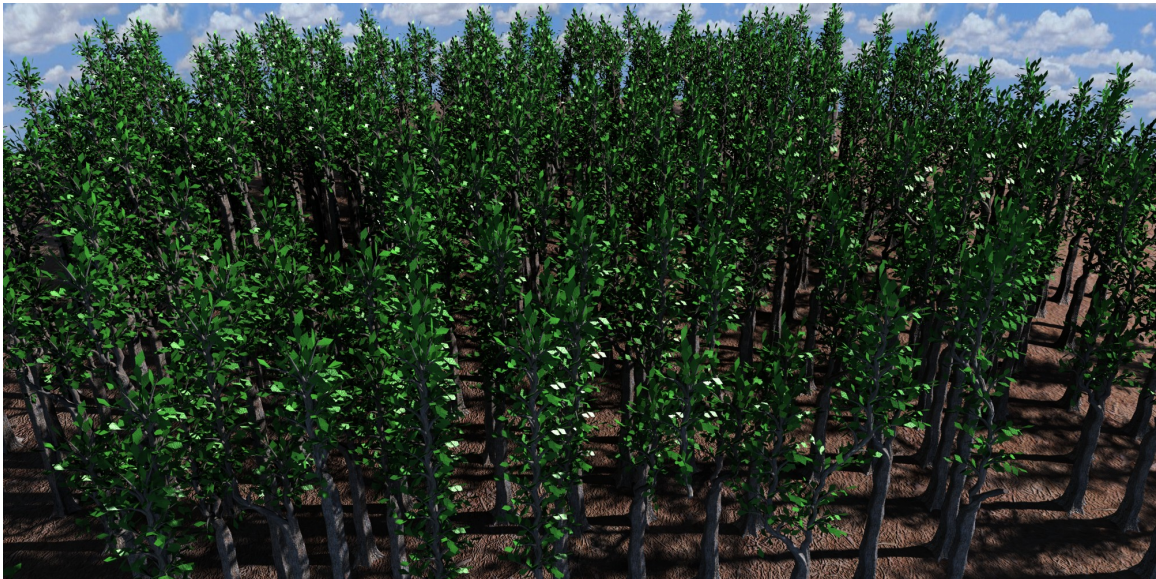
(b) Synthesized Model

Figure 3.21: Given two trees (a), model synthesis produces a forest (b). Grid lines are drawn in the example model.
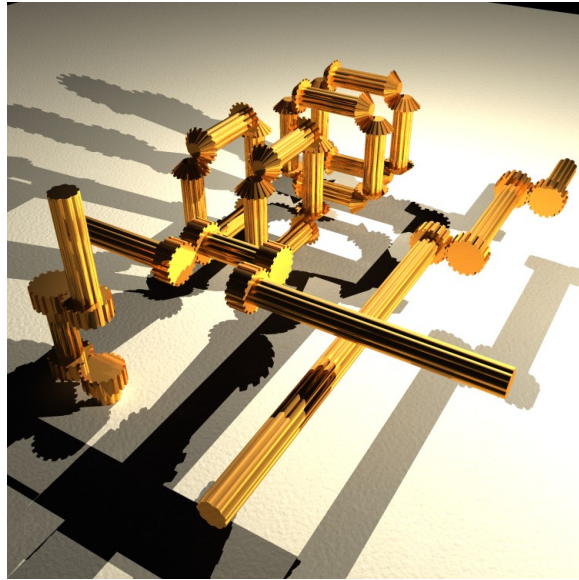
(a) Gadgets Example Model
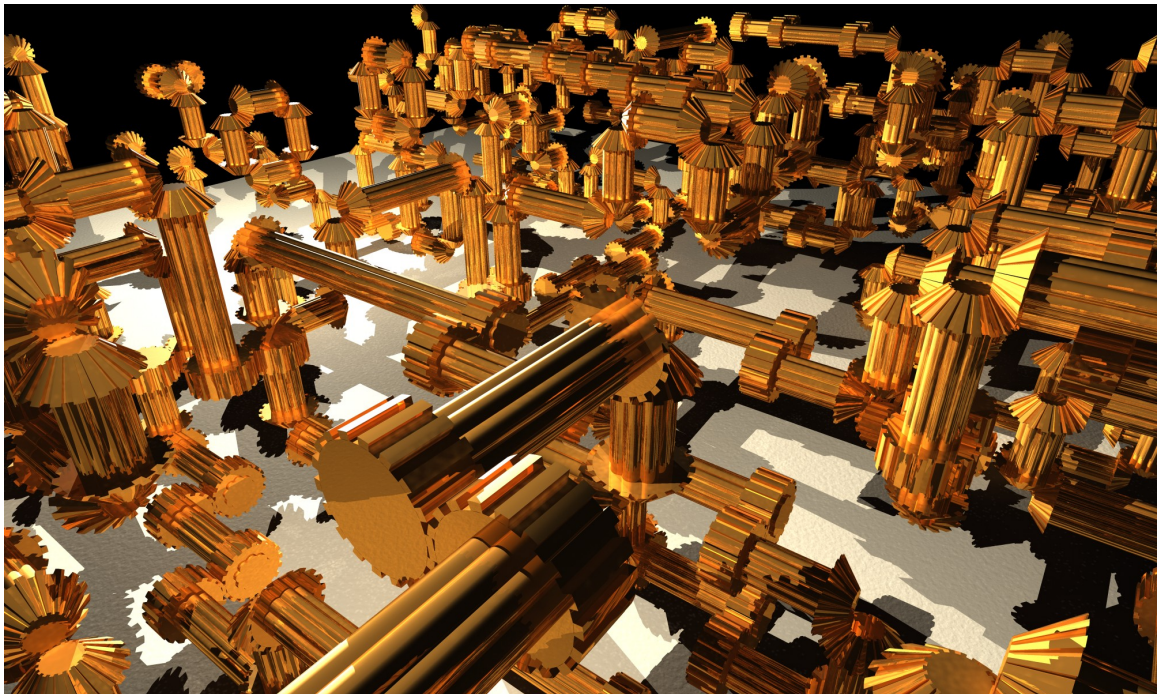
(b) Synthesized Model

Figure 3.22: Given a few rotating gears (a), model synthesis generates complex machinery (b).
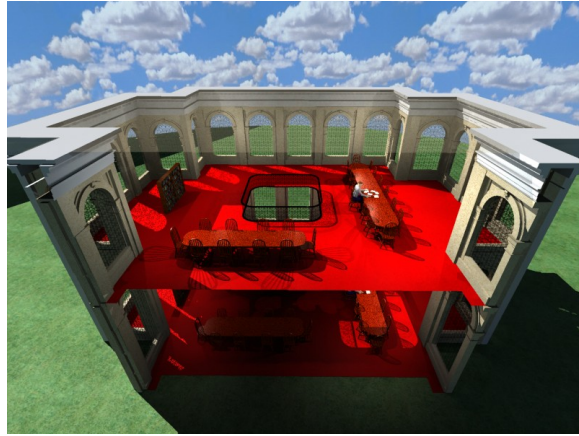
(a) Library Example Model
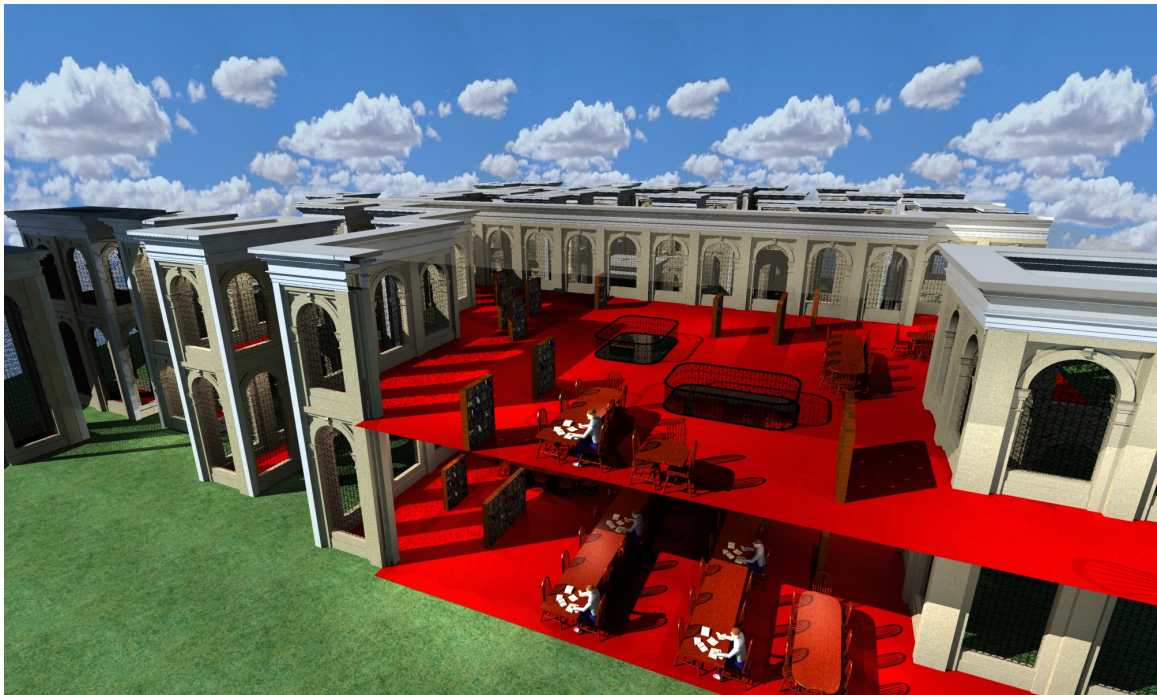
(b) Synthesized Model

Figure 3.23: Given the interior and exterior of a building (a), model synthesis produces several buildings (b). Sections of the buildings are cut away and the roof is made transparent to show the interior more clearly.

3.5 Variants of Model Synthesis

Model synthesis can be altered and extended in several ways. The grid can be altered, additional constraints such as symmetry can be added, model synthesis can be extended into four-dimensions to create models that change over time.

3.5.1 Modifying the Grid

In all of the examples, we have shown the model pieces are cubes in 3D or squares in 2D, but we could easily use rectangles or parallelograms instead. Also, hexagons could be used and then each position would have six neighbors rather than four in 2D. Model synthesis could be extended into cylindrical coordinates (r, θ, z). In this case, each position would have neighbors in the $\pm r, \pm \theta$, and $\pm z$ directions and the model pieces would need to be rotated and scaled depending upon their position. Extensions are also possible into spherical or toriodal coordinates.

The transition matrices T_x, T_y, T_z are used to describe the adjacency constraint between neighboring points in x, y, and z directions. But we can easily use a transition matrix describe a constraint between any two points. The points could be diagonal neighbors, so that each point would have 8 neighbors in 2D or 26 neighbors in 3D. Model synthesis could easily be extended so that the adjacency constraint not only applied to immediate neighbors such as $\mathbf{x}$ and $\mathbf{x} + \hat{\imath}$, but less immediate neighbors like $\mathbf{x}$ and $\mathbf{x} + 2\hat{\imath}$.

3.5.2 Symmetry

Transition matrices can be used to impose constraints on positions that are not close spatially. A good example of this is symmetry. Suppose we would like to create a model with reflective symmetry about the plane $x = s$ for any integer s. The model is symmetric if for any point (x, y, z), its mirror image appears at the point $(2s - x, y, z)$.

Each model piece may have another model piece that is its mirror image. Let $f(b)$ be a function that returns the mirror image of b. For the model M to be symmetric, $M(x, y, z) = b \Leftrightarrow M(2s - x, y, z) = f(b)$. A symmetric model M is constructed by applying a transition matrix T_s between the points (x, y, z) and $(2s - x, y, z)$ where $T_s[b, c] = 1 \Leftrightarrow c = f(b)$. Applying this transition matrix ensures that the model is symmetric. This transition matrix is used in addition to the normal transition matrices that define the adjacency constraint in Equation 3.3. By using both sets of transition matrices, the algorithm will create models that are both symmetric and consistent. Other types of symmetry such as translational and rotational symmetry can be applied in a similar process. Figure 3.24 shows models with different types of symmetry that are all based off the example model in Figure 3.18(a).

3.5.3   Other Constraints

Other constraints can be used to control the large-scale structure of the output. The user might have a general idea of where certain types of objects should appear. Each label normally has an equal chance of being selected at each point. But the probability of selecting each label could easily be modified so that it is higher or lower depending on if the user does or does not want it to appear within a particular area. The user could even set some of the probabilities to be zero in some places. If the probability of label b at point $\mathbf{x}$ is set to zero, it can be removed by setting $C_{M_t}(\mathbf{x}, b) = 0$ and then that removal may be propagated as normally done using Algorithm 3.2. By changing these probabilities, cities and other structures can be created in the shape of various symbols and other objects. We can also generate multiple outputs, evaluate how well they match the user's desired goal, and then select the best output. Figure 3.25 shows models that are constrained to be in the shape of different symbols.
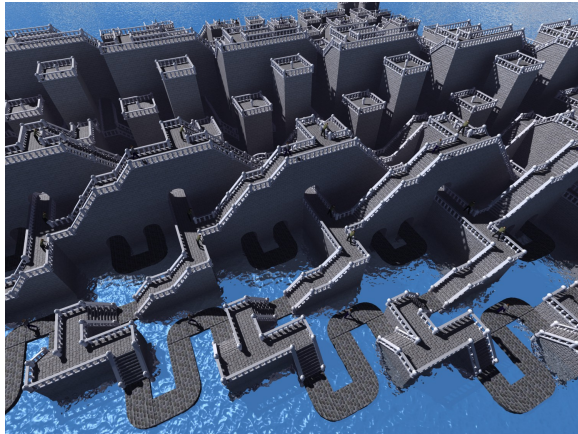
(a) Translational Symmetry
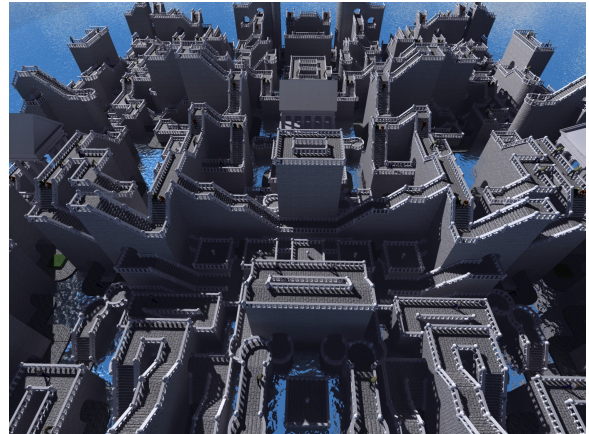
(b) Reflectional Symmetry
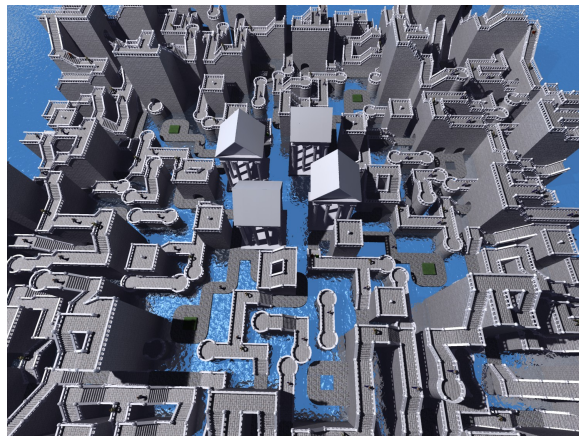
(c) Rotational Symmetry

Figure 3.24: Three symmetric models based off the example model from Figure 3.18(a).
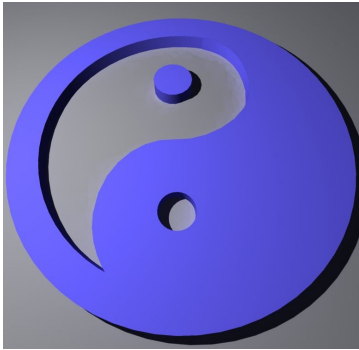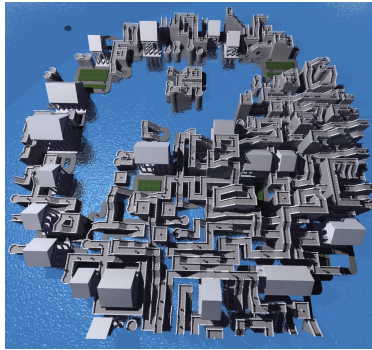
(a) Yin and Yang (b) Escheresque Model (c) Tree Model
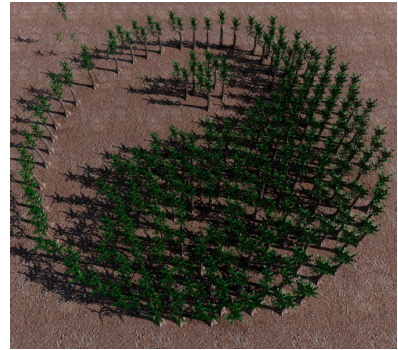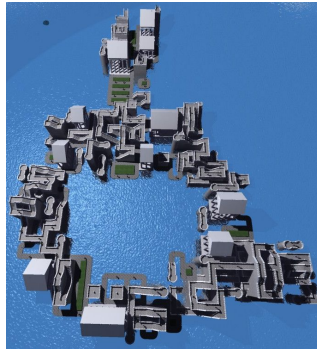
(d) Wheelchair (e) Escheresque Model (f) Tree Model
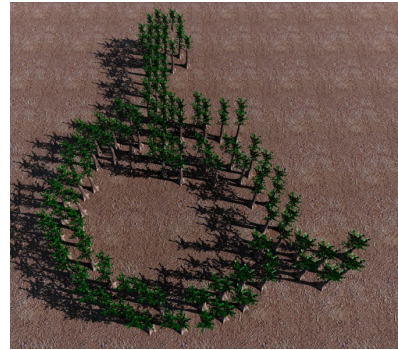
Figure 3.25: Models that are constrained to be in the shape of various symbols. This uses the example models in Figures 3.18(a) and 3.21.

3.5.4 Higher-Dimensional Models

Model synthesis can also be extended to four-dimensional and higher-dimensional models. By adding a time dimension, we can generate time-varying models. Time-varying models are created much like ordinary 3D models. The user provides a time-varying example model constructed out of time-varying model pieces, and the algorithm synthesizes a time-varying model that resembles the input. Each point in the 4D model not only has spatial neighbors, but also temporal neighbors and the adjacency constraint applies to spatial and temporal neighbors. Extending model synthesis into higher dimensions requires only a few changes to the algorithm, but time-varying model pieces can be harder to visualize. Figure 3.26 depicts a model of a red ball traveling to the right. The model has two spatial and one temporal dimension. The model is shown in two different ways. It is shown first with multiple exposures of the ball shown overlapping each other and second as a 1D slice that changes over time. Four model pieces are used as shown in Figure 3.26(a): one empty model piece, one where the ball travels out of the square disappearing into the right boundary, one where the ball moves from the left boundary to the right, and one where the ball emerges from the left boundary. These four model pieces are used to create a time-varying model of a moving ball shown in Figure 3.26(b).
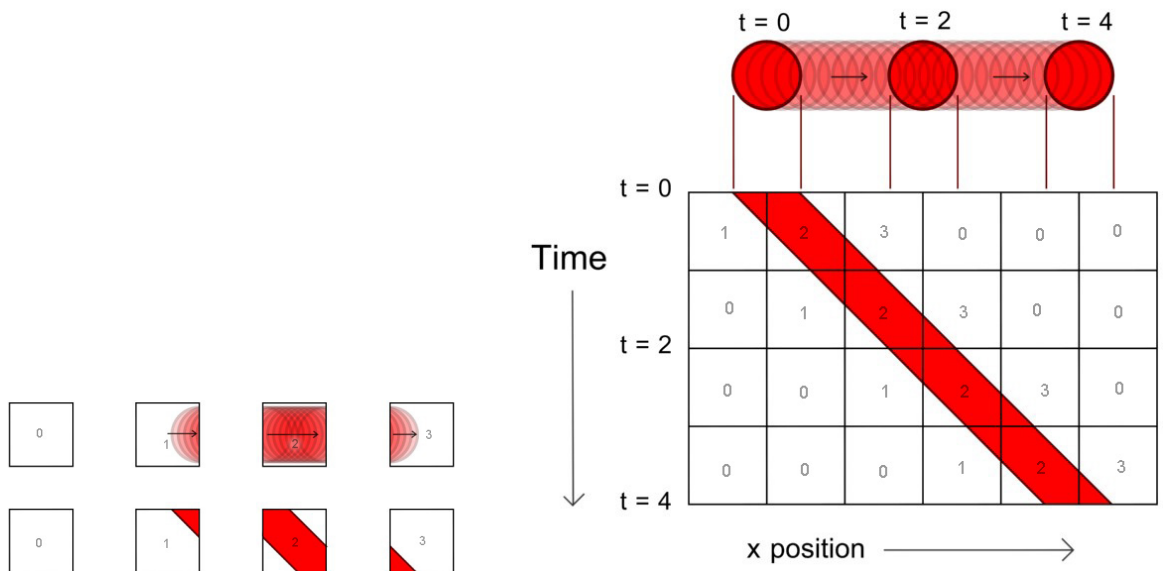
(a) Four Time-Varying Model Pieces. The top row shows multiple exposures of a moving ball and the bottom row shows a 1D slice of the ball moving over time.

(b) A Time-Varying Model of a rolling ball. The bottom grid shows how a 1D slice changes over time.

Figure 3.26: Model synthesis can be extended into higher dimensions to create models that change over time. Time-varying model pieces (a) are used to create a time-varying model (b).

Chapter 4

Continuous Model Synthesis

4.1 Limitations of Discrete Model Synthesis

Discrete model synthesis can sometimes be difficult to use because the example model
must be decomposed into model pieces. This task may not be difficult when the objects
fit naturally on a grid like many architectural models, but it is difficult when the objects
do not fit on a grid.

An example is given in Figure 4.1(a) which shows a triangle that is not aligned to
the grid. This triangle can be divided into model pieces using a grid as shown in Figure
4.1(b). In fact, any shape can be divided into model pieces, but the problem is that
almost all model pieces appear only once. By including these model pieces, the model
is not self-similar and only exact copies of the input model are allowed into the output
model according to the adjacency constraint. Without self-similarity, model synthesis
cannot produce interesting new variations of the input. Notice that self-similarity is lost
only when the triangle is divided along the grid lines. The original triangle is self-similar.
The points on its edges are identical to each other on a microscopic scale.

If the shapes do not align to the grid, one possible solution is to align the grid to the
shapes. We could apply an affine transformation to the grid as mentioned in Section
3.5.1 and as shown in Figure 4.1(c). By using this grid to divide up the input, the model

pieces repeat as shown in Figure 4.1(d). Because they repeat, model synthesis does more than simply generate copies of the original triangle, it generates triangles with different sizes. This grid works much better. However, we cannot align the grid to every input model. If the input model contained a regular pentagon or if contained two triangles with different orientations, then we could not align the grid completely with the input model.

(a) A triangle not aligned with the grid

(b) The triangle can be decomposed into model pieces, but they are almost all unique.

(c) The grid can be skewed to fit this triangle.

(d) With an aligned grid, the same few model pieces repeat.
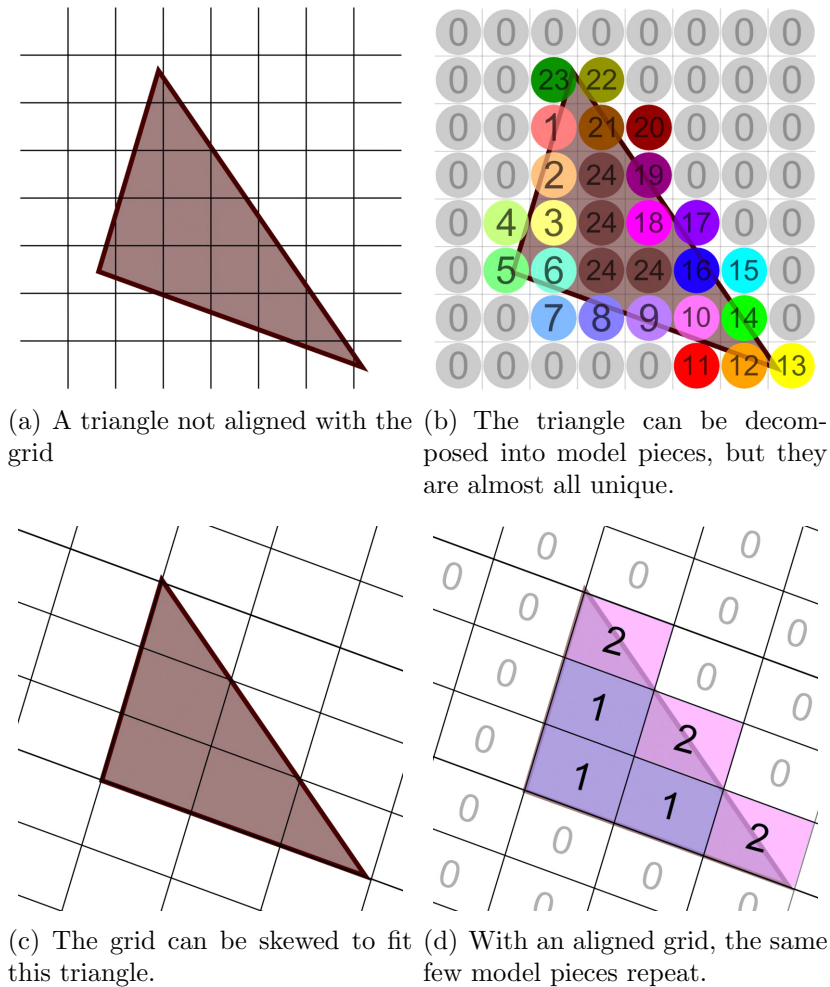
Figure 4.1: Discrete model synthesis may not work well on models that are not aligned with the grid (a). The model can be divided into pieces; (b) discrete model synthesis will simply generate exact copies of the input. The grid can be transformed so it is aligned with the triangle (c). This work much better (d), but such a transformation only works on a few shapes.

A different problem with discrete model synthesis is illustrated in Figure 4.2. The

model in Figure 4.2 contains axis-aligned objects stacked on top of one another. If we divide the model using the grid shown in Figure 4.2(a) only the blue objects produce repeating model pieces. The purple objects repeat, but not along with the grid. This problem can be addressed by choosing a smaller better grid size. By shrinking the grid height in Figure 4.2(a) by a factor of $\frac{1}{5}$ in 4.2(b) the blue and purple objects both have repeating model pieces. Most objects can be fitted onto a grid, if the grid size is sufficiently small. If the objects do not fit exactly, they can be scaled slightly. Figure 4.3 shows another example of how the grid size can be decreased to create repeating model pieces. Decreasing the grid size is a useful strategy for fitting the objects to a grid. However, decreasing the grid spacing, also increases the computation time. The entire process of fitting the objects onto a grid can be both difficult for the user and difficult to automate.
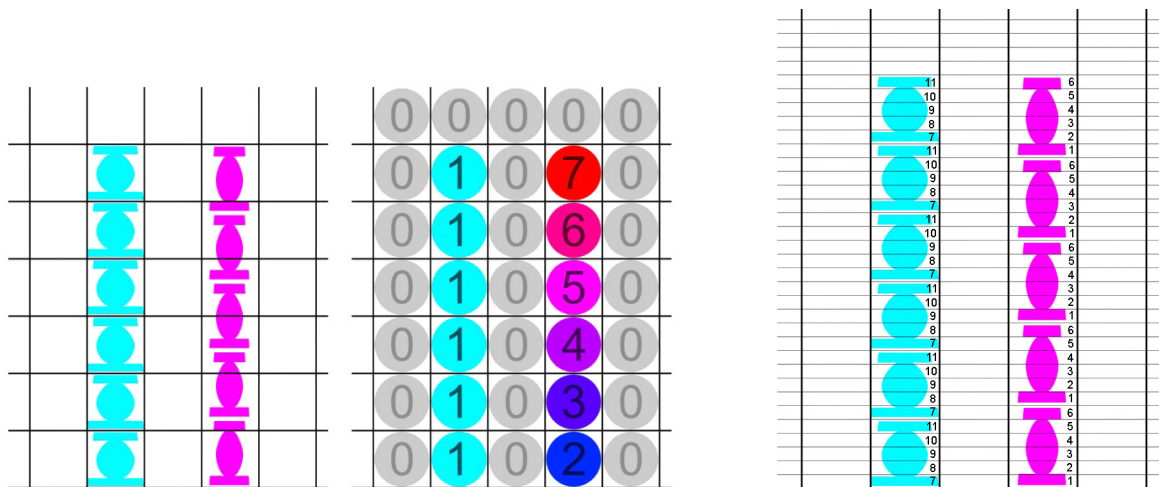
(a) An input model is divided into model pieces. Many unique model pieces are produced when the shapes are not aligned to the grid.

(b) Both shapes can be aligned to the grid if a smaller grid is used.

Figure 4.2: Even though the same purple shape is copied many times in (a), the model pieces are all different. Sometimes a smaller grid can be used to produce models pieces that repeat, as shown in (b).
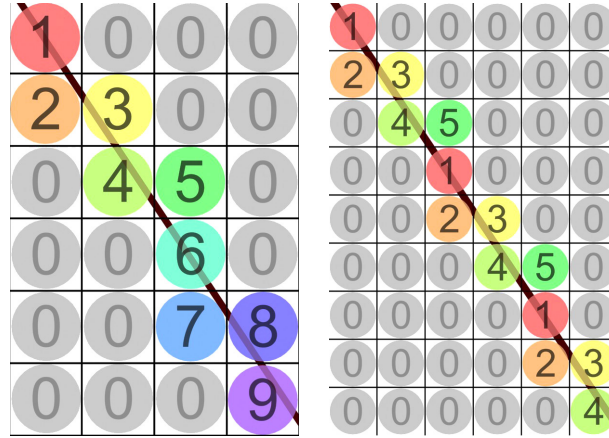
(a) If a line does not fit on the grid, all the model pieces are unique.

(b) A different grid spacing produces repeating model pieces.

Figure 4.3: Using a smaller grid size, may improve the results by creating repeating model pieces.

4.2 The Continuous Model Synthesis Problem

As shown in Figures 4.2 and 4.3 some of the limitations discussed in Section 4.1 can be handled by using smaller model pieces. The limitations in Section 4.1 are caused because objects that are self-similar lose their self-similarity when divided into model pieces on a grid. Points that are self-similar, lose their self-similarity when they are combined with all the other points within a cell. None of the limitations in Section 4.1 would apply if we dealt with individual self-similar points or model pieces that are individual points. Point-sized model pieces could be used to represent many geometric shapes such as planes edges, and vertices that have no thickness to them.

Geometric shapes are represented more easily in the continuous domain with real-valued coordinates $\mathbf{x} \in \mathbb{R}^3$. Many shapes are difficult to represent using discrete model synthesis since discrete models use integer-valued coordinates $\mathbf{x} \in \mathbb{Z}^3$. The alternative is to use a continuous model which is defined as a mappings from $\mathbb{R}^3 \rightarrow K$ where K is the set of possible labels. The goal of continuous model synthesis is to generate models that satisfy the adjacency constraint. The adjacency constraint is slightly different in

the continuous domain.

The adjacency constraint states that for every point $\mathbf{x}$ in M, there must exist a point $\mathbf{x}'$ in E that is has the same local neighborhood. In Figure 4.4, the points $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$, $\mathbf{d}$, and $\mathbf{e}$ have the same local neighborhoods as the points $\mathbf{a}'$, $\mathbf{b}'$, $\mathbf{c}'$, $\mathbf{d}'$, and $\mathbf{e}'$.

(a) Example Shape $E(\mathbf{x})$ (b) Generated Output Shape $M(\mathbf{x})$
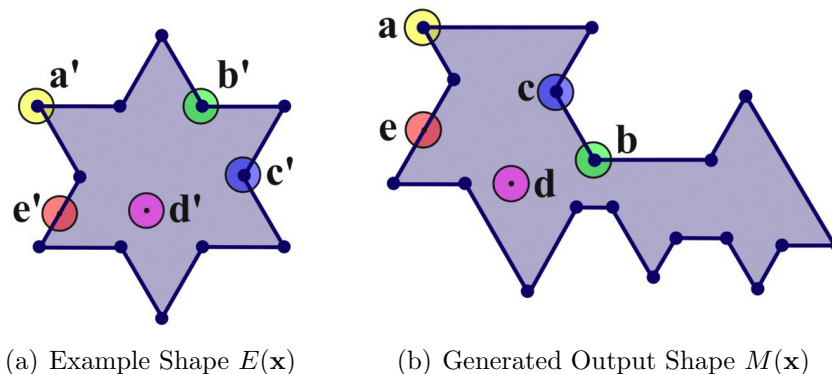
Figure 4.4: The Continuous Adjacency Constraint. For each selected point $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$, $\mathbf{d}$, and $\mathbf{e}$ in M, a point with the same local neighborhood $\mathbf{a}'$, $\mathbf{b}'$, $\mathbf{c}'$, $\mathbf{d}'$, and $\mathbf{e}'$ exists in the example model E.

More formally, the model M is *consistent* with E if for all points $\mathbf{x}$, $\exists \epsilon > 0, \mathbf{x}' \in \mathbb{R}^3$ such that for all small vectors $\mathbf{d}$ where $||\mathbf{d}|| < \epsilon$,

$$M(\mathbf{x} + \mathbf{d}) = E(\mathbf{x}' + \mathbf{d}) \tag{4.1}$$

This is the continuous adjacency constraint. It is similar to the discrete adjacency constraint in Equation 3.1.

In this chapter, we discuss the problem of generating consistent continuous models in detail. A general solution to this problem would overcome the limitations of discrete model synthesis discussed in Section 4.1. However, the continuous model synthesis problem is far more difficult than the discrete problem because a solution assigns labels to an infinite set of points. In order to address this problem, we introduce three different approaches in Sections 4.3, 4.4, and 4.5. Each approach has important limitations. We have been unable to produce models using the first two approaches except in simple

cases. The first approach has unresolved theoretical problems. The second approach is extremely difficult to implement because it requires exact and robust 3D Boolean operations and Minkowski sum computations. Even though these approaches are unsuccessful, there is value in discussing them. Negative results tend to be overlooked, but discussing them can be valuable [51, 27]. First, it would be difficult to explain the design of the successful algorithm in Section 4.5 without understanding some of the issues with the alternatives. Second, it may be possible to overcome these issues and develop a better algorithm with these ideas.

4.3   Point-Sized Model Pieces

In this section, we discuss an approach to continuous model synthesis that is quite similar to discrete model synthesis. It would follow a similar procedure and use a catalog C_{M_t} of possible labels like discrete model synthesis. The key difference is that the model pieces or labels are assigned to individual points in the continuous domain. For example, the triangle shown in Figure 4.5(a) could be constructed using 8 labels. One label for each vertex and each edge, plus one for the triangle's interior and one for the empty space. Like discrete model synthesis, labels would be assigned to points in an incomplete model M_t and then a catalog C_{M_t} would be computed where $C_{M_t}(\mathbf{x}, b) = 1$ if the label b could be added at point $\mathbf{x}$. But unlike the discrete case, C_{M_t} can not be stored as an array. The catalog C_{M_t} may contain an infinite number of points $\mathbf{x} \in \mathbb{R}^3$ and C_{M_t} would be stored as a union of geometric shapes. Figure 4.5(b) shows an incomplete model M_t and Figure 4.5(c) shows C_{M_t}. Computing C_{M_t} is not trivial. We do not discuss how to compute C_{M_t} because the purpose of this section is to demonstrate that this approach has several problems with it including that M may become inconsistent even when C_{M_t} is computed correctly.

One serious issue arises because we cannot reevaluate C_{M_t} every time a label is

(a) Example Shape E (b) New Incomplete Shape M_t (c) Catalog C_{M_t}

(d) A Consistent Model using Parts of an Extended Bottom Edge

(e) Inconsistent model after Extending the Bottom Edge
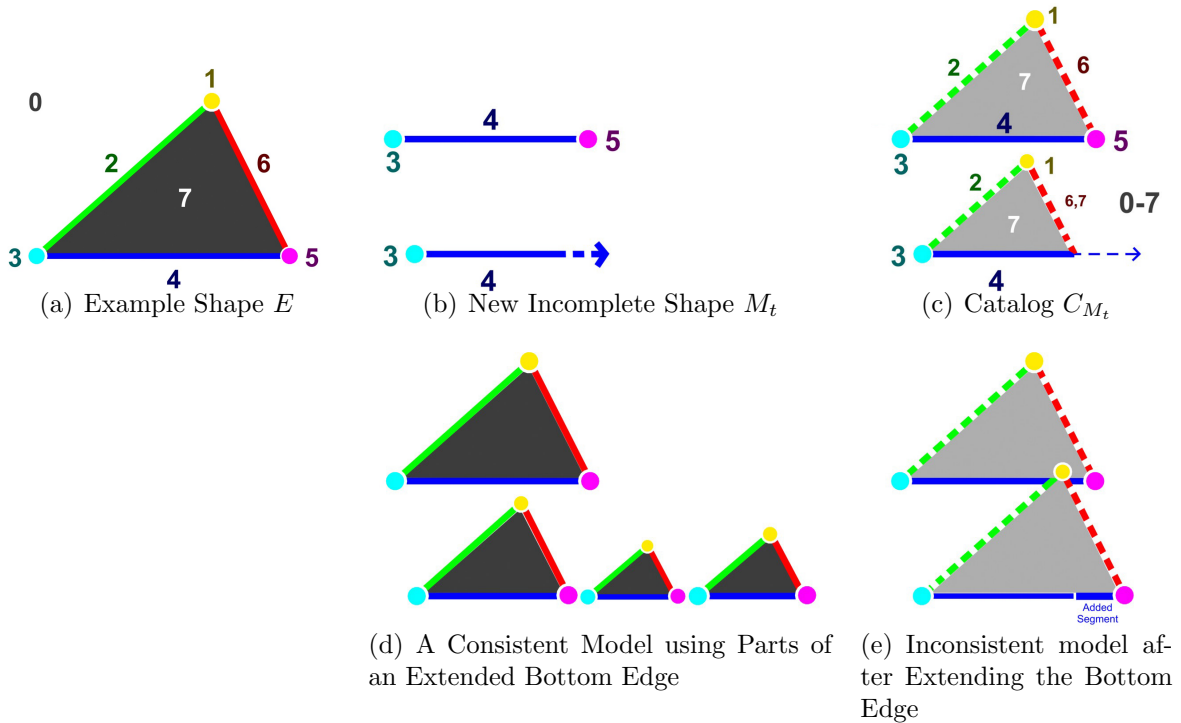
Figure 4.5: An approach that only uses point-sized model pieces has difficulties. For a triangular example shape (a), a new incomplete shape is being generated with two edges (b). The catalog C_{M_t} (c) shows that the bottom edge can be extended indefinitely, but if it is extended (e) an inconsistent model is created. The catalog is correct since consistent models (d) can be created with those labels.

assigned to a point in M_t. Since an infinite number of point assignments are required to create a single edge, edges are created by adding line segments instead of points into M_t. But adding line segments can cause M_t to become inconsistent as shown in Figure 4.5(e) since the bottom triangle cannot be completed without intersecting the top triangle. The source of the problem is that the catalog C_{M_t} is based on adding labels at individual points, but we are trying to add line segments. Labels can be assigned to each individual point in M_t without difficulty as Figure 4.5(d) demonstrates. The model becomes inconsistent only if the entire line segment is added to M_t at once, but they cannot be added individually.

One possible solution to this problem is to effectively create a small buffer region around each edge and each vertex by giving the edges and vertices a small finite size.

The buffer regions could be tiny, but as long as the line segments assigned to M_t are given in equally small increments, then the problems shown in Figure 4.5 can be avoided. This solution is discussed in Section 4.4.

4.4 Discrete and Point-Sized Model Pieces Using Minkowski Sums

We first discuss in Section 4.4.1 a method for creating objects with a fixed finite size within continuous model synthesis. This is useful not only for dealing with the issues raised in Section 4.3, but also because it allows the user to precisely control the dimensions of the objects. Then in Section 4.4.2 the form of the catalog of possible labels C_{M_t} and in Sections 4.4.3, 4.4.4, and 4.4.5 how to compute it in certain cases. Section 4.4.6 shows why C_{M_t} is extremely difficult to implement in some cases. This algorithm is the most general and powerful algorithm in this thesis, but there are so many issues implementing it that it has not been implemented.

4.4.1 Discrete Objects in Continuous Model Synthesis

Let us consider a one-dimensional model $E(x)$ that contains a discrete fixed-size objects and is defined as
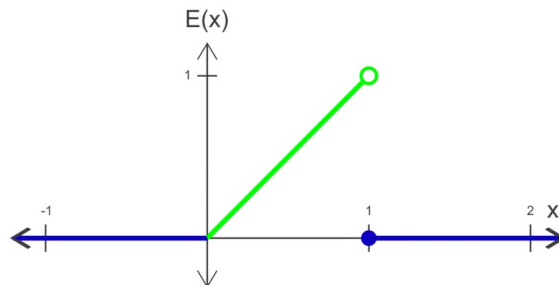
$$E(x) = \begin{cases} x, & 0 \le x < 1 \\ 0, & \text{otherwise} \end{cases} \tag{4.2}$$

In $E(x)$, all the points from $0 < x < 1$ are uniquely labeled and only the 0 label appears at multiple points. The continuous adjacency constraint (Equation 4.1) implies that $\forall y, d \mid 0 < y < 1$ and $0 < y + d < 1$

$$M(x) = y \Leftrightarrow M(x + d) = y + d \tag{4.3}$$
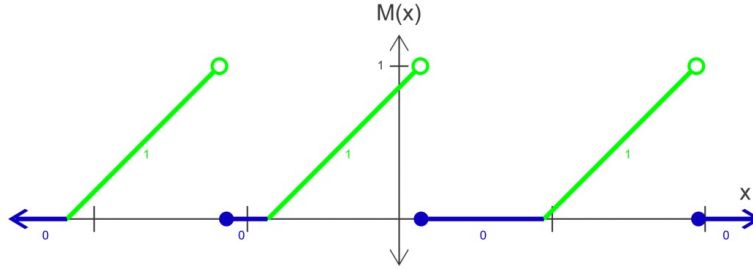
Figure 4.6: A model $M(x)$ consistent with $E(x)$. $M(x)$ contains copies of a discrete object.

If any nonzero label is present, then they all are. The labels are grouped into an indivisible unit or object. Each nonzero label represents a point within a discrete object. Only exact copies of the object appear in the new model M as shown in Figure 4.6. Each copy of the object has its own local coordinate system. In fact, the value of the local coordinate is equal to the value of $E(x)$ and $M(x)$. A one-dimensional local coordinate system is sufficient for a one-dimensional object, but 2D or 3D objects need a 2D or 3D coordinate system as shown in Figure 4.7. The model may contain more than one type of object. So each point should map to two quantities: an object type and a local coordinate. If J is the total number of object types, then the 3D continuous models E and M are defined as mappings $E, M : \mathbb{R}^3 \rightarrow [0, \ldots, J-1] \times \mathbb{R}^3$. The 1D model defined in Equation 4.2 does not conform to this definition because it does not record the object

type, but we can alter it record this

$$E(x) = \begin{cases} (1, x), & 0 \le x < 1 \\ (0, 0), & \text{otherwise} \end{cases} \qquad (4.4)$$

This model contains two different types of objects. Object 1 is a discrete object one-unit wide. Object 0 is the empty space. Empty space is different because it is not represented in discrete units. $M(x)$ can have any amount of empty space no matter how big or how small. Another difference is that empty space has translational symmetry, but discrete objects usually do not. Empty space is symmetric because every point within a volume of space is the same, but discrete objects are not symmetric because their shape varies according to position. A volume of empty space is essentially the same point copied over an entire volume.

Each object type has a smallest indivisible unit. For object 1 in Equation 4.4, it is a one-unit wide object. For empty space, it is a single point of empty space. The set of points in objects j's indivisible unit is called V_j or the extent of object j. In $E(x)$ in Equation 4.4, V_0 is a point $V_0 = \{0\}$ and V_1 is a line segment $V_1 = [0, 1)$. In a three-dimensional model, V_j could be a point, a line segment, a polygon, or a polyhedron. A two-dimensional model is shown in Figure 4.7 which contains a pillar. The pillar's top and bottom are called objects 1 and 3. Both objects have a fixed width and height and their extents V_1 and V_3 are both rectangles. The extent of empty space is a point $V_0 = \{0\}$. The pillar's middle section has an extent of a line segment since it is a combination of a discrete object along its x-coordinate and a symmetric object along its y-coordinate. It has a fixed width, but it can have any height. It is symmetric along the y-coordinate, but not along the x-coordinate.
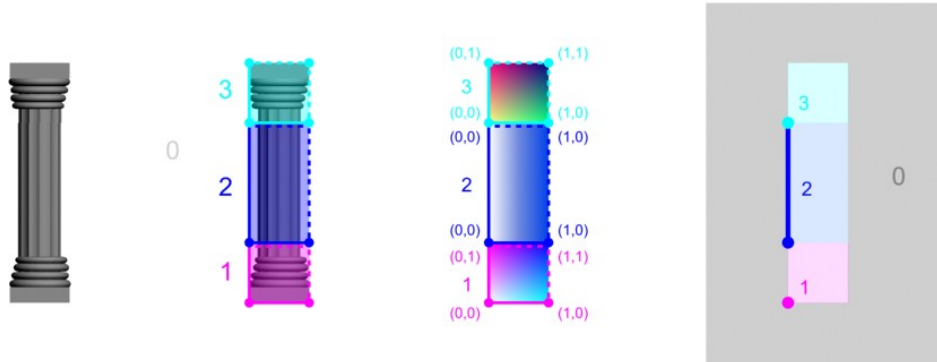
(a) The Example Model in Various Forms
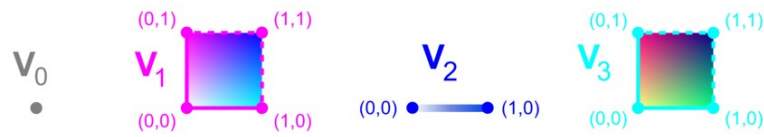
(b) The Extents of the Objects
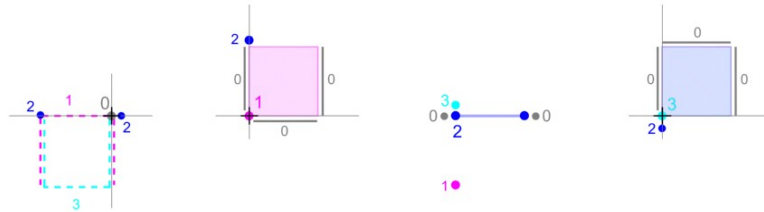
(c) Each object has a set of other objects that must be present to satisfy the adjacency constraint.

Figure 4.7: A 2D Model with several sub-objects. The top and bottom of the pillar are discrete objects with a 2D local coordinate system. The middle part is symmetric in the Y direction. Empty space is labeled 0 and is symmetric in all directions. Each object has an extent and a set of other objects that must be present to satisfy the adjacency constraint. Solid lines in (c) indicate that every point must be present to satisfy the adjacency constraint. Dashed lines in (c) indicate that only one point from the object needs to be present.

The extents of the objects V_j are indivisible. If any part of these units are present in E, then the entire unit is. If this statement is true for E, then it is also true for M by the adjacency constraint. This means that for every $\mathbf{p}$ inside object j, $\forall \mathbf{p} \in V_j$

$$M(\mathbf{x}) = (j, \mathbf{0}) \Leftrightarrow M(\mathbf{x} + \mathbf{p}) = (j, \mathbf{p}) \tag{4.5}$$

and the same statement holds for E. This is a more general version of Equation 4.3.

4.4.2 The Catalog of Possible Labels, C_{M_t}

The basic approach is similar to discrete model synthesis. It is to construct a catalog C_{M_t} of which labels can be added into M_t and then assign labels from the catalog. So $C_{M_t}(\mathbf{x}, b) = 1$ if the label $b = (j, \mathbf{p})$ can be assigned at point $\mathbf{x}$. The catalog cannot be represented with a discrete array since $\mathbf{x}, \mathbf{p} \in \mathbb{R}^3$. Instead, it is stored as a collection of geometric shapes. The catalog C_{M_t} is a function of six real numbers since $\mathbf{x}$ and $\mathbf{p}$ are 3D points, but it does not need to be represented as a collection of six-dimensional shapes. Fortunately, it can be represented perfectly as a collection of three-dimensional shapes. This follows from Equation 4.5 which implies that for all $\mathbf{p} \in V_j, C_{M_t}(\mathbf{x}, (j, \mathbf{p})) = C_{M_t}(\mathbf{x} - \mathbf{p}, (j, \mathbf{0}))$. So we only need to record the value of $C_{M_t}(\mathbf{x}, (j, \mathbf{0}))$ which is a 3D function, since all the other values of $C_{M_t}(\mathbf{p}, (j, \mathbf{p}))$ can be derived from it. To simplify the notation, $C_{M_t}(\mathbf{x}, (j, \mathbf{0}))$ is written as $C_{M_t}(\mathbf{x}, j)$.

The catalog C_{M_t} is defined for discrete model synthesis using Statements 3.7 and 3.6. Statement 3.7 expressed the occupancy constraint and Statement 3.6 expresses the adjacency constraint. We derive similar statements for continuous model synthesis. The occupancy constraint means that only one label may occupy any point. It is derived as $\forall \mathbf{p} \in V_i, \mathbf{q} \in V_j$

$$M_t(\mathbf{x}) = (i, \mathbf{0}) \quad \Rightarrow \quad M_t(\mathbf{x} + \mathbf{p}) = (i, \mathbf{p}) \tag{4.6}$$

$$\Rightarrow \quad M_t(\mathbf{x} + \mathbf{p}) \neq (j, \mathbf{q}) \vee (j, \mathbf{q}) = (i, \mathbf{p}) \tag{4.7}$$

$$\Rightarrow \quad M_t(\mathbf{x} + \mathbf{p} - \mathbf{q}) \neq (j, \mathbf{0}) \vee (j, \mathbf{q}) = (i, \mathbf{p}) \tag{4.8}$$

$$\Rightarrow \quad C_{M_t}(\mathbf{x} + \mathbf{p} - \mathbf{q}, j) = 0 \vee (j, \mathbf{q}) = (i, \mathbf{p}) \tag{4.9}$$

Statements 4.6 and 4.8 follow directly from Statement 4.5. Statement 4.7 follows since $(i, \mathbf{p})$ and $(j, \mathbf{q})$ can not occupy the same point $\mathbf{x} + \mathbf{p}$ unless they are exactly the same label. Statement 4.9 follows since $C_{M_t}(\mathbf{x}, j)$ should be 0 if $M_t(\mathbf{x}) \neq (j, \mathbf{0})$. Statement 4.9 is part of the definition of C_{M_t}. It means that each time a label $(i, \mathbf{0})$ is assigned to M_t that is $M_t(\mathbf{x}) = (i, \mathbf{0})$, we remove from C_{M_t} a set of points since it is true $\forall \mathbf{p} \in V_i, \mathbf{q} \in V_j$. The set to remove is expressed as using a Minkowski sum operation $\oplus$ which is defined between any two sets A and B where

$$A \oplus B = \{a + b | a \in A \text{ and } b \in B\} \tag{4.10}$$

Statement 4.9 can be written using a Minkowski sum as

$$M_t(\mathbf{x}) = (i, \mathbf{0}) \quad \wedge i \neq j \quad \Rightarrow C_{M_t}(\mathbf{x} + \mathbf{r}, j) = 0, \quad \forall \mathbf{r} \in V_i \oplus -V_j \tag{4.11}$$

$$M_t(\mathbf{x}) = (i, \mathbf{0}) \quad \quad \Rightarrow C_{M_t}(\mathbf{x} + \mathbf{r}, i) = 0, \quad \forall \mathbf{r} \in V_i \oplus -V_i - \{\mathbf{0}\} \tag{4.12}$$

This is the continuous version of the occupancy constraint. The discrete version was described in Statement 3.7.

In addition to the occupancy constraint, the adjacency constraint must also be checked within C_{M_t}. There is no reason to verify the adjacency constraint within each object. The constraint is satisfied assuming Statement 4.5 is true. The adjacency con-

straint only needs to be verified on the boundaries of the object. Let ∂V_i be the set of points on object i's boundary. Let i and j be two objects. A transition function $T(i, \mathbf{p}, j, \mathbf{q})$ is defined for point on the objects' boundaries for $\mathbf{p} \in \partial V_i, \mathbf{q} \in \partial V_j$. By definition, it is equal to one iff two labels are allowed to touch, which means that they were touching in the example model E

$$T(i, \mathbf{p}, j, \mathbf{q}) = \begin{cases} 1 & , \exists \mathbf{x} | E(\mathbf{x} - \mathbf{p}) = (i, \mathbf{0}) \text{ and } E(\mathbf{x} - \mathbf{q}) = (j, \mathbf{0}) \text{ and } (i, \mathbf{p}) \neq (j, \mathbf{q}) \\ 0 & , \text{otherwise} \end{cases}$$

$$(4.13)$$

According to the adjacency constraint, a label $(i, \mathbf{p})$ is only allowed into M if it touches another label $(j, \mathbf{q})$ allowed by the transition function. So for all $\mathbf{p} \in \partial V_i, \mathbf{q} \in \partial V_j$,

$$M(\mathbf{x}) = (i, \mathbf{p}) \Rightarrow \exists (j, \mathbf{q}) | T(i, \mathbf{p}, j, \mathbf{q}) = 1 \textbf{ and } M(\mathbf{x} - \mathbf{q}) = (j, \mathbf{0}) \qquad (4.14)$$

$$M(\mathbf{x}) = (i, \mathbf{0}) \Rightarrow \exists (j, \mathbf{q}) | T(i, \mathbf{p}, j, \mathbf{q}) = 1 \textbf{ and } M(\mathbf{x} + \mathbf{p} - \mathbf{q}) = (j, \mathbf{0}) \qquad (4.15)$$

Its contrapositive is given as

$$\nexists (j, \mathbf{q}) | T(i, \mathbf{p}, j, \mathbf{q}) = 1 \textbf{ and } M(\mathbf{x} + \mathbf{p} - \mathbf{q}) = (j, \mathbf{0}) \Rightarrow M(\mathbf{x}) \neq (i, \mathbf{0}) \qquad (4.16)$$

This can be turned into a statement about C_M

$$\nexists (j, \mathbf{q}) | T(i, \mathbf{p}, j, \mathbf{q}) = 1 \textbf{ and } C_M(\mathbf{x} + \mathbf{p} - \mathbf{q}, j) = 1 \Rightarrow C_M(\mathbf{x}, i) = 0 \qquad (4.17)$$

94

The catalog C_M is defined as the largest set of points for which Statements 4.12 and 4.17 are true. The catalog C_M has many of the same properties as it does in the discrete case.

Under the adjacency constraint, each object in M can only touch objects that it had touched in E. Any assignment that violates this constraint is removed from the catalog. We consider three different cases in Sections 4.4.3 - 4.4.5. First, how to ensure that a certain discrete object touches another discrete object. Second, how to ensure that a symmetric object touches a discrete object. Third, how to ensure that a discrete object touches a symmetric object. In each case, C_{M_t} is computed differently. A detailed example of how to compute C_{M_t} is illustrated in Figure 4.9. We explain only how to compute C_{M_t} in a few simple cases. A full explanation of how to compute C_{M_t} in every case would be long and unnecessary because our goal is simply to demonstrate that computing C_{M_t} can be enormously difficult. We can demonstrate this using only a few simple cases. Computing C_{M_t} requires computing many 3D Boolean operations and many 3D Minkowski sums. These operations must be computed robustly or else C_{M_t} may contain huge errors. Exact solutions are difficult to compute because of numerical errors and degeneracies despite extensive research on these operations [66]. Implementing this algorithm would be exceptionally difficult. This is discussed more in Section 4.4.6.

C_{M_t} is much easier to compute in discrete model synthesis, but the overall approach is similar in both discrete and continuous model synthesis. Each time a label is assigned the model, assignments may be removed from C_{M_t}. Each removal is computed using a Boolean difference operation. Each removal may cause further removals, but eventually the removals will stop. Then the process repeats, additional labels are assigned from the catalog, and the catalog is recomputed.
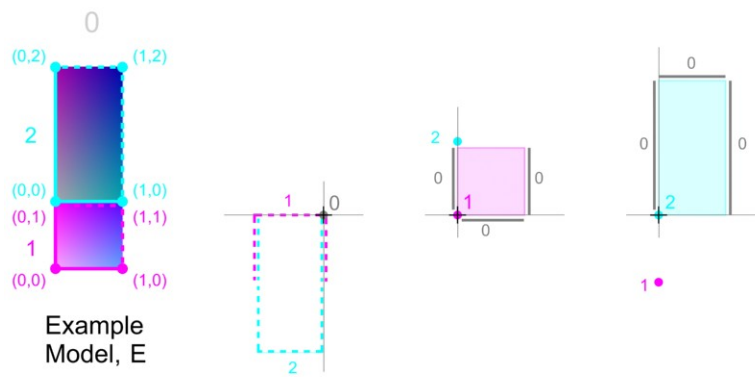
Figure 4.8: Example Model of Two Discrete Objects. From the example model, a set of labels that need to be present to satisfy the adjacency constraint are shown for 0, 1, and 2 labels. We only keep track of where the origins of each object should be in relation to each other. These relationships are used extensively in Figure 4.9 to compute C_{M_t}. Solid lines mean that all of the labels need to be present. Dashed lines mean only one point from the line needs to be present. Object 1 is always directly below object 2 and surrounded by empty space.

Example Model, E

New Incomplete
Model, M

Initially, C_{M_t} begins with all possible
assignments. Then regions of C_{M_t}
are removed using the occupancy
and adjacency constraints.

C_{M_t} Initially

0,1,2

The figures on both pages examine
how C_{M_t} changes after a single
assignment is made to the model M.
The assigment (1,0) is added at the
origin.

Step 1:

First, the occupancy constraint is
applied. Since object 1 was added
several Minkowski sums are computed
between V and $-V_0$, $-V_1$, and $-V_2$.
Each of these regions is removed from
C_{M_t}.

$$- \quad V_1 \quad \oplus \quad -V_2 \quad = \quad - \quad V_1 \oplus -V_2$$

$$- \quad V_1 \quad \oplus \quad -V_0 \quad = \quad - \quad V_1 \oplus -V_0$$

$$- \left(V_1 \quad \oplus \quad -V_1 \quad - \quad \{0\} \right) = - \quad V_1 \oplus -V_1 - \{0\}$$

Step 2:

Next, we begin applying the adjacency
constraint. The 2 label (blue) must
have a 1 label (purple) beneath it. So
once a region of 1 labels are removed,
a region of 2 labels must also be
removed.

$$- \quad 1 \quad \oplus \quad = \quad - \quad 2$$

$$V_1 \oplus -V_1 - \{0\}$$

Each of the four regions above is
subtracted from C_{M_t}. The result is
shown to the right. The calculations
continue on the next page.

C_{M_t}

0,1

2

∅

1

0

0,1

0,1,2

Figure 4.9: Computing C_{M_t}

97

Step 3:

The remaining 2 labels of C_{M_t}
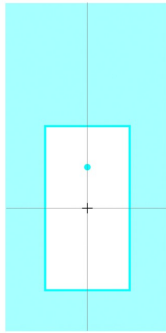
A 0 label needs either a 2 or a 0 label beneath it. Region F contains those points which might have 2 labels beneath them.
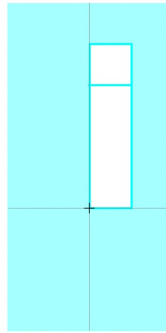
To find points that might have 0 labels beneath them, we trace rays up from Region F. Regions such as $V_1 \oplus -V_0$ which cannot contain 0 labels act as obstacles.

$\oplus$ =

Region F

Ø

Region F

Region A

0
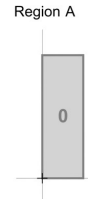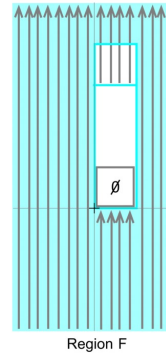
Region A contains all the eliminated 0 labels.

Step 4:

The 2 object needs 0 labels above it and to its left and right. After the 0 labels in Region A are removed, the 2 labels in Region B are too. The 1 labels in Region C are removed for similar reasons.

0

Region A

$\oplus$ =

2

Region B

Step 5:

0

Region A

$\oplus$ =

1
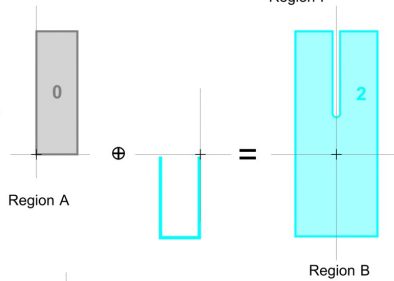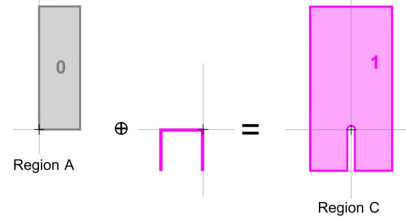
Region C

Step 6:

The removal of Regions B and C cause further removals. Object 2 must be directly above object 1. The 1 labels in Region D and the 2 labels in Region E are removed.

2

Region B

$\oplus$ =

1

Region D

Step 7:

1

Region C

$\oplus$ =

2
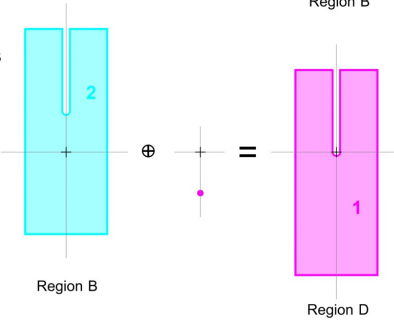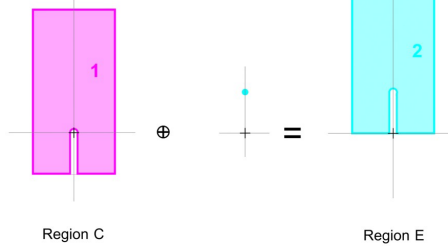
Region E

Regions A,B,C,D, and E are subtracted from the previously computed value of C_{M_t}. All the labels that would violate the occupancy and adjacency constraints have been removed from C_{M_t}.

0,1

2

1

Ø

0

0,1

0,1,2

Previous value of C_{M_t}

- Regions A,B,C,D,E =

0,1

2

Ø

1

0

0,2

0,1,2

Final value of C_{M_t}

98
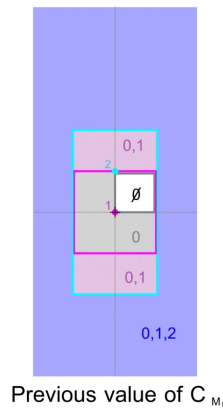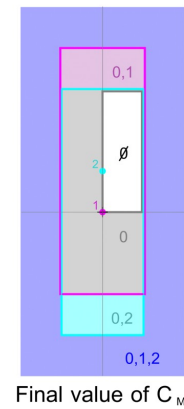
4.4.3 Two Discrete Objects

To simplify the problem, we only consider cases in which every label on object i's boundary only touches one other label. This means that

$$T(i, \mathbf{p}, j, \mathbf{q}) = T(i, \mathbf{p}, k, \mathbf{r}) = 1 \Leftrightarrow (j, \mathbf{q}) = (k, \mathbf{r}). \tag{4.18}$$

Statement 4.18 is not always true, since there could be multiple copies of object i and different copies could touch different objects.

We first discuss how to update C_{M_t} in the case where two discrete objects touch, such as in Figure 4.8. Suppose that the labels $(i, \mathbf{p})$ and $(j, \mathbf{q})$ touch one another so that $T(i, \mathbf{p}, j, \mathbf{q}) = 1$. Using Statements 4.17 and 4.18 we find that for all $\mathbf{x}$

$$C_{M_t}(\mathbf{x} + \mathbf{p} - \mathbf{q}, j) = 0 \implies C_{M_t}(\mathbf{x}, i) = 0 \tag{4.19}$$

This statement is true for any two labels that touch (assuming Statement 4.18). We have only considered two labels that touch $(i, \mathbf{p})$ and $(j, \mathbf{q})$, but the objects touch at many points. We could repeat the same analysis on other points, but the result be the same. For example, suppose $(i, \mathbf{p}')$ and $(j, \mathbf{q}')$ are two other labels from the same two objects which touch each other. From equation 4.5, we know that $\mathbf{p}' - \mathbf{q}' = \mathbf{p} - \mathbf{q}$, so we derive the same result as Statement 4.19.

The catalog C_{M_t} is stored as a collection of geometric shapes. Let $C_{M_t}^i = \{\mathbf{x} | C_{M_t}(\mathbf{x}, i) = 1\}$. If a set of points R is removed from $C_{M_t}^j$, then according to Statement 4.19 we should remove from $C_{M_t}^i$ the set R translated by $\mathbf{q} - \mathbf{p}$. Several examples of this computation are shown in Figure 4.9. In Step 6 of Figure 4.9, removing Region B from $C_{M_t}^2$ means Region D should be removed from $C_{M_t}^1$ where Region D is a translated copy of Region B. A similar computation is performed in Steps 2 and 7 of Figure 4.9.

4.4.4 Discrete Object Touching a Symmetric Object

A discrete object could touch a symmetric object such as empty space. Suppose that the discrete object i touches the symmetric object j over a set of points P. So for all $\mathbf{p} \in P$, $T(i, \mathbf{p}, j, \mathbf{0}) = 1$ and we assume that so Statement 4.18 is true. Using Statements 4.17 and 4.18 we find that for all $\mathbf{p} \in P$

$$C_{M_t}(\mathbf{x} + \mathbf{p}, j) \implies C_{M_t}(\mathbf{x}, i) = 0 \tag{4.20}$$

This statement implies that any time a set of points R is removed from $C_{M_t}^j$, then the set $R \oplus -P$ should also be removed from $C_{M_t}^i$. This removal occurs in Steps 4 and 5 of Figure 4.9. The set R is the set of points removed from $C_{M_t}^0$. In Step 4, the set P is the set of points on the boundary of object 2 that touch empty space which is shown as Region A. The Minkowski sum of Region A and $-P$ is subtracted from $C_{M_t}^2$ and is called Region B. In Step 5, Region C is computed similarly for object 1.

4.4.5 A Symmetric Object Touching a Discrete Object

Statement 4.20 does not explain how $C_{M_t}^i$ affects $C_{M_t}^j$. This is difficult to compute because j is a symmetric object which can be adjacent to itself. This means that $C_{M_t}^j$ which we are trying to compute depends upon $C_{M_t}^j$. We need to keep track of three different sets of points. First, there is a set of points that are already know to violate the occupancy constraint (Equation 4.12) or the adjacency constraint (Equation 4.1). For example in Step 3 of Figure 4.9, $i = 2$ and $j = 0$ and the set $V_1 \oplus -V_0$ is know to violate the occupancy constraint. Second, there is a set of points for which the occupancy and adjacency constraints are known to be satisfied. This set is shown as Region F in Figure 4.9. A point $\mathbf{x}$ is in the set if there exists a point $\mathbf{p}$ such that

$$T(i, \mathbf{p}, j, \mathbf{0}) = 1 \text{ and } C_{M_t}(\mathbf{x} - \mathbf{p}, i) \Rightarrow C_{M_t}(\mathbf{x}, j) = 1 \tag{4.21}$$

So the first set of points must not belong in $C_{M_t}^j$ and the second set of points must belong in $C_{M_t}^j$. Any point $\mathbf{x}$ outside of these two sets belong in $C_{M_t}^j$ only if it is possible to trace a ray from a point in the second set to $\mathbf{x}$ without intersecting the first set. At these points, the adjacency constraint is satisfied because a symmetric object can be adjacent to itself.

4.4.6   Difficulties with this approach

Sections 4.4.1 - 4.4.5 are meant to given an overview of the approach without delving into the details which are quite complicated. Figure 4.9 shows how many steps are involved even in a very simple example. This example model simply contains one discrete object stacked on top of another. Most models are much more complicated than this. In a more complicated example, different copies of a discrete object might touch different objects, so that Statement 4.18 may be false or a single object might touch multiple objects on different parts of its face. These more complicated input models can still be handled just using Boolean operations and Minkowski sums, but many more of these operations would be necessary in these cases. Constructing fairly simple models requires hundreds of Boolean operations and Minkowski sums.

Despite considerable effort, every attempt we have made to implement this algorithm has only achieved limited success. The algorithm relies heavily on 3D Boolean and 3D Minkowski sum operations. This is the source of the problem. Boolean operations and Minkowski sums are important topics in many areas of computer graphics unrelated to procedural modeling. These operations have been studied intensively, but they are still enormously difficult to implement exactly and robustly for general 3D models [66]. There are many ways to approximate these operations, but approximate solutions may introduce huge errors since each set that is removed causes other sets to be removed. The operations all build on top of one another, so small errors can easily expand into large errors.

To simplify the problem, we attempted to implement the algorithm in two dimensions and a separately attempted to implement it on axis-aligned boxes since Boolean operations and Minkowski sums are much easier to compute in these particular cases. However, other implementation issues came up. Many mathematical details which are often overlooked become important when implementing the algorithm. For example, suppose an object occupies all the points from 0 to 1. It may not seem important, but it could be important to know if the object occupies the points at 0 or the point at 1. Tiny details like this become important and make the algorithm difficult to implement. Another example is shown in Figure 4.7, the label 3 is directly above the label 2 an infinitesimally small distance. In this case, the occupancy constraint is difficult to implement.

A common assumption that is made to simply Boolean operations is that the operations are regularized, meaning that any isolated vertices, edges, and faces can be removed, but regularization is unacceptable for this algorithm since it relies on isolated vertices. Many points in the catalog depend on isolated vertices. If they are removed, C_M would become empty and the algorithm would fail.

Certainly, it may be possible to overcome these implementation issues in the future. One idea is to find a way to approximate the Boolean operations and Minkowski sums without introducing errors into the algorithm. If the implementation issues could be overcome, the resulting algorithm would be the most powerful algorithm described in this thesis.

4.5   The Continuous Model Synthesis Algorithm

We can construct a much simpler continuous model synthesis algorithm, but this algorithm can not generate every consistent output model. It generates output models in which every face lies on a predetermined set of planes or in 2D that all its edges lie on a

set of parallel lines. This is called the parallel plane assumption in 3D and the parallel line assumption in 2D. An example is shown in 2D in Figure 4.10. Given an input shape (see Figure 4.10(a)), a set of parallel lines are created (Figure 4.10(c)), and the edges of the output model all lie on one of these lines 4.10(d)). The 3D case is similar. In 3D, the faces of the output model all lie on one of a set of planes. The edges of the output model all lie on the lines where two planes intersect and the vertices are located at the intersection of three planes. The number of possible vertex locations stored in C_{M_t} is finite because of the parallel plane assumption which greatly simplifies the algorithm. Another reason the algorithm is simpler is that each edge of the output is created from a finite set of line segments and each face is created from a finite set of polygons. In constrast, the sets of possible vertex and edge locations were infinite in the approach in Section 4.4 which used operations involving finite sets including Boolean operations and Minkowski sums. The parallel plane assumption simplies the algorithm, but also may limit in some ways which models can be produced. The algorithm shares some limitations of discrete model synthesis, but it also overcomes many of them. The limitations are discussed more in Section 4.8.

Figures 4.10 and 4.11 both give an overview of the algorithm. Each figure uses different input shapes. The algorithm is described in pseudocode as Algorithm 4.1. First, sets of lines are created (Figures 4.10(c) & 4.11(b)) that are parallel to the input example shape (Figures 4.10(a) & 4.11(a)). These lines divide the plane into faces, edges, and vertices. Each face, edge, or vertex is eventually assigned a label. Each label corresponds to a neighborhood that satisfies the adjacency constraint (Equation 4.1). Computing the set of all possible labels is described in Sections 4.5.1 and 4.5.3. From this set, only certain labels can be applied to each particular edge or vertex. Which labels can be used is discussed in Sections 4.5.2 and 4.5.4. If a vertex is assigned a particular label, then any edge incident to the vertex must have a label that agrees with the vertex's label. This is discussed in Section 4.5.5. Sections 4.5.1 - 4.5.5 discuss Steps

(a) Example Shape $E(\mathbf{x})$

(b) Parallel lines dividing $E(\mathbf{x})$

(c) Parallel lines diving up the plane.
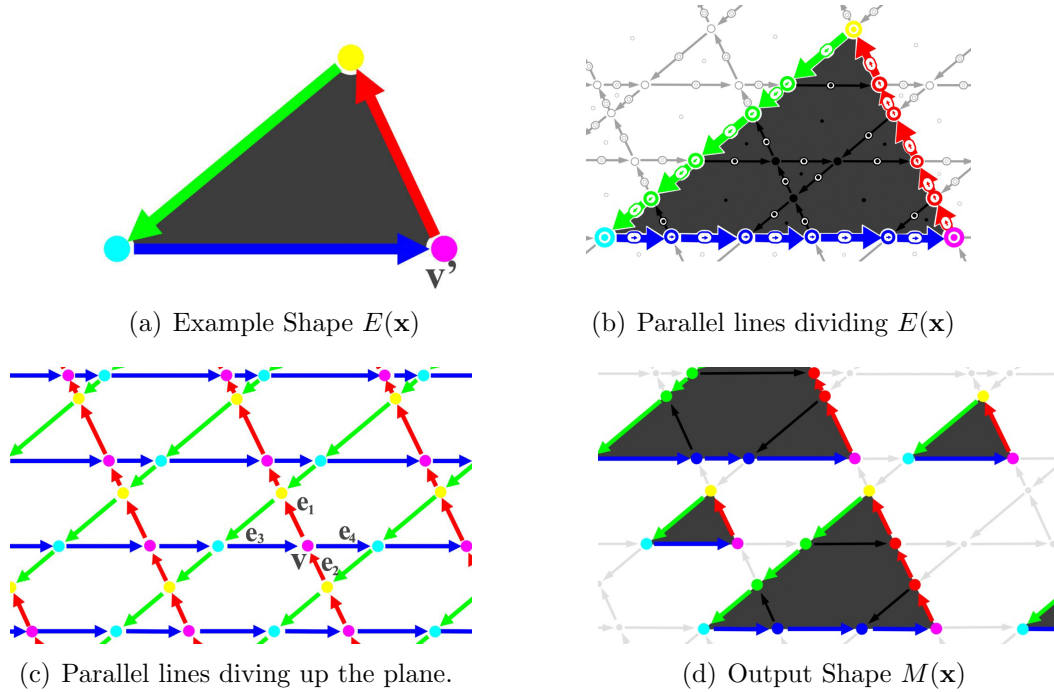
(d) Output Shape $M(\mathbf{x})$

Figure 4.10: Lines parallel to the input shape (a), divide the plane into faces, edges, and vertices (c). The output shape (d) is formed within the parallel lines. The set of acceptable vertex and edges labels in the output (d) can be found by dividing the input along parallel lines (b).

1-4 of the algorithm. The rest of the algorithm (Steps 5-10) is described in Section 4.5.6 and is almost identical to the discrete model synthesis algorithm. All possible labels are placed into a catalog C_{M_t}. Labels are assigned to each edge and each vertex and invalid labels are removed until an acceptable output shape is generated like the shapes shown in Figures 4.10(d) and 4.11(c).

4.5.1 The Set of Possible Labels in 2D

This section explains how to find all possible neighborhoods that satisfy the adjacency constraint in a 2D model for Step 2 of Algorithm 4.1. First, we need a way to identify every local neighborhood.

The example model E may contain many different types of objects. Let P_j be the set of point where object j is present $P_j = \{\mathbf{x}|E(\mathbf{x}) = j\}$. We assume that for all j,

(a) Example Shape $E(\mathbf{x})$

(b) Parallel lines diving up the plane.

(c) Acceptable Output Shape $M(\mathbf{x})$
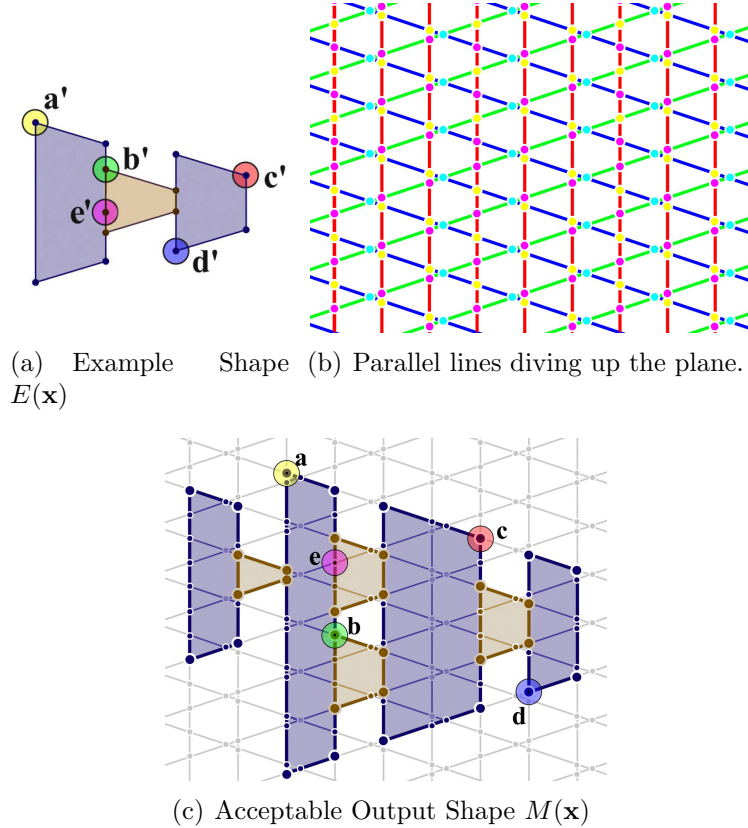
Figure 4.11: (a) From the input shape E, (b) sets of lines parallel to E intersect to form edges and vertices. (c) The output shape is formed within the parallel lines. For each selected point $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$, $\mathbf{d}$, and $\mathbf{e}$ in M, there are points $\mathbf{a}'$, $\mathbf{b}'$, $\mathbf{c}'$, $\mathbf{d}'$, and $\mathbf{e}'$ in the example model E which have the same neighborhood. The models E and M contain two different kinds of object interiors shown in blue and brown.

P_j is a polyhedron in the 3D case and a polygon in the 2D case. Let b be the open ball with unit radius centered at the origin. For a point $\mathbf{x} \in \mathbb{R}^2$ we choose $\epsilon > 0$ sufficiently small and intersect the open ball with radius ϵ around $\mathbf{x}$ with a polyhedron P_j: $N_\epsilon^j(\mathbf{x}) = (\mathbf{x} + \epsilon \cdot b) \cap P_j$. The enlarged version of the neighborhood is sometimes called the vertex figure vf

$$\mathrm{vf}^j \mathbf{x} = \bigcup_{\lambda > 0} \lambda \cdot \left(N_\epsilon^j(\mathbf{x}) - \mathbf{x} \right) \tag{4.22}$$

If for every point $\mathbf{x}$ in M, there is a point $\mathbf{x}'$ in E such that $\mathrm{vf}^j \mathbf{x} = \mathrm{vf}^j \mathbf{x}'$ for all j, then

Algorithm 4.1 Overview of Continuous Model Synthesis

1: Create planes parallel to E. Create vertices where three planes intersect. Create edges where two do.
2: Create a list of all acceptable labels (Sections 4.5.1 & 4.5.3).
3: Create a list of acceptable labels for each edge and each vertex (Sections 4.5.2 & 4.5.4).
4: Determine which edge labels can be adjacent to each vertex label (Section 4.5.5).

5: Main Loop (Section 4.5.6)
6: **while** there exists an unassigned vertex v **do**
7:     Randomly select a label b from C_M.
8:     Assign label b to vertex v.
9:     Remove from C_M all labels that are incompatible with this assignment.
10: **end while**
---

the adjacency constraint is satisfied.

We first discuss 2D model synthesis, since it is easier to explain and illustrate. We discuss the 3D case in Section 4.5.3. In 2D, every edge of the example shape has a line parallel to it that intersects the origin. The line divides the plane into two half-planes. To the line's left is one half-plane h_i and to its right is h_i's complement h_i^C. The input shape has a finite number of edges, so we can enumerate every half-plane $h_1, h_2, \ldots, h_m$ where m is the number of edges with a distinct slope. By combining these half-planes using Boolean operations, the vertex figures of every point $\mathbf{x}$ in $E(\mathbf{x})$ can be described.

Points outside the polygon P_j which do not touch its boundary have a vertex figure equal to the empty set. Points in the interior of P_j have a vertex figure equal to the entire plane $\mathbb{R}^2$. Points touching an edge of P_j have one of the half-planes as their vertex figure. The vertex figure of a point at a vertex of P_j uses the two half-spaces that are parallel to its incident edges. If its interior angle is less than $180°$, then the vertex figure is the intersection of the two half-planes as shown by the three vertices in Figure 4.12. If the angle at the vertex is a reflex angle (i.e. greater than $180°$), then the vertex figure is a union of the half-planes as shown in Figure 4.13.

This shows how to write a Boolean expression describing how the neighborhood surrounding any given point intersects a single polygon. But each neighborhood could
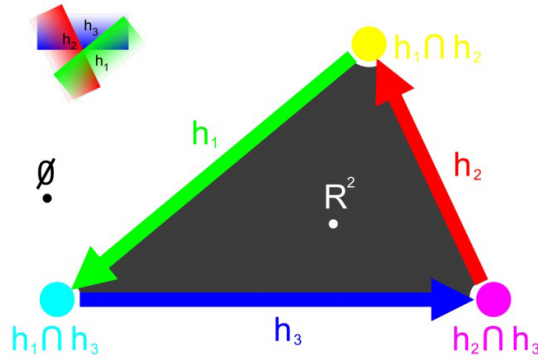
Figure 4.12: Vertex figures of various points on a triangle described as Boolean expressions of half-planes. Eight possible labels are identified $\emptyset, \mathbb{R}^2, h_1, h_2, h_3, h_1 \cap h_2, h_1 \cap h_3$, and $h_2 \cap h_3$.
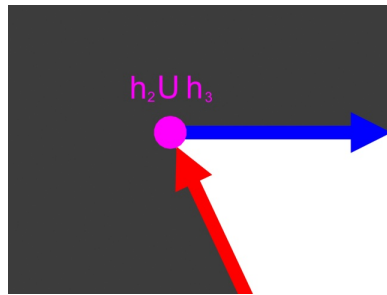
Figure 4.13: Vertex figure of a concave vertex.

intersect multiple objects as shown in Figure 4.14. When multiple objects intersect, we can compute a Boolean expression for each object separately and then combine all them all together. Let k_1 and k_2 be two types of objects and let b_1 and b_2 be two Boolean expressions. Each Boolean expression describes the set of points in a local neighborhood that are inside of a specific object. We use the notation $k_1 \cdot b_1 + k_2 \cdot b_2$ to describe a neighborhood which contains the object k_1 at the points b_1 and object k_2 at b_2. For example, $1 \cdot h_3^C + 2 \cdot h_1 \cap h_3$ describes a neighborhood where an edge h_3^C of object 1 touches a vertex of $h_1 \cap h_3$ of object 2. This notation is only used when there are multiple objects.
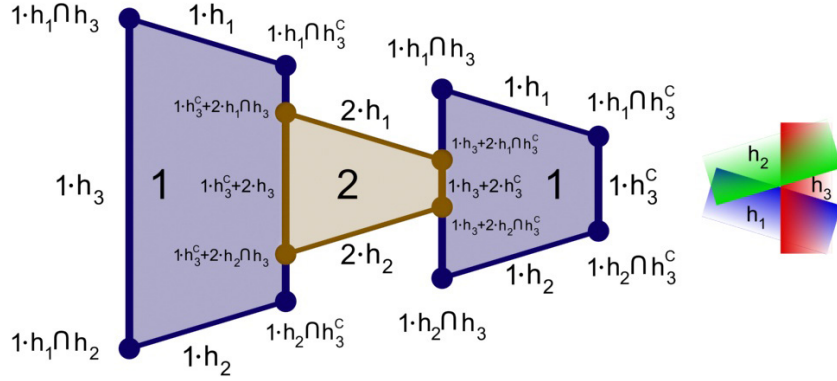
Figure 4.14: Boolean expressions using half-spaces describing the neighborhoods of various points on a shape with two different object types.

4.5.2 The Set of Labels for Each Vertex and Edge in 2D

Each label represents a possible local neighborhood. Every label must satisfy both the adjacency constraint and the parallel line assumption. The set of labels that satisfy the adjacency constraint were discussed in Section 4.5.1. This section discusses applying the parallel line assumption in Step 3 of Algorithm 4.1 to only allow certain labels at each point. Many of the labels specify that an edge of the polygon intersect the point where the label is assigned, but edges are not allowed to intersect every point under the parallel line assumption. They can only intersect points on one of the lines. Whenever a label contains the half-space h_i, that labels marks the position of an edge parallel to h_i. For example, the label h_1 marks the position of an edge parallel to h_1 and the label $h_1 \cap h_2$ marks the position of two edges that intersect at a vertex. There are three different types of points to consider:

1. Points inside the faces of Figure 4.10(c) do not lie on any of the parallel lines, so their vertex figures are either $\emptyset$ or $\mathbb{R}^2$ since they cannot use any of the half-planes. Each face is entirely inside or outside the polygon.

2. Points on the edges of Figure 4.10(c) lie on one of the parallel lines and must either be on an edge of the polygon or entirely inside or outside a polygon. If the edge is

parallel to the half-plane h_i, then they must have vertex figures of $\emptyset, \mathbb{R}^2, h_i$, or h_i^C.

3. The vertices of Figure 4.10(c) lie on two of the parallel lines can be entirely inside or outside a polygon or on an edge or vertex of the polygon. Suppose that the vertex lies at the intersection of two lines parallel to h_1 and h_2, then its vertex figure could be any combination of these half-planes: $\emptyset, \mathbb{R}^2, h_1, h_1^C, h_2, h_2^C, h_1 \cap h_2, h_1 \cap h_2^C, h_1^C \cap h_2, h_1^C \cap h_2^C, h_1 \cup h_2, h_1 \cup h_2^C, h_1^C \cup h_2$, or $h_1^C \cup h_2^C$.

The labels can only be used if they satisfy both the parallel plane assumption and the adjacency constraint. For example, Figure 4.12 identifies eight labels that satisfy the adjacency constraint, but a horizontal edge of Figure 4.10(c) can be assigned three of the eight labels according to the parallel line assumption. These three labels are shown in Figure 4.15(a-c).

(a) Exterior, $\emptyset$ (b) Interior, $\mathbb{R}^2$ (c) Edge, h_3 (d) Edge, h_3^C
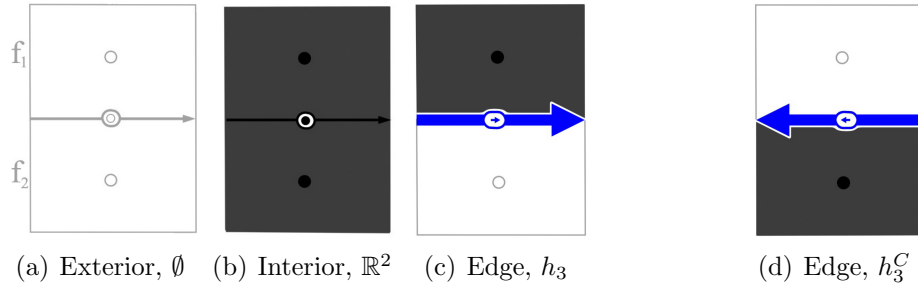
Figure 4.15: Possible labels of a horizontal edge. Only (a-c) are found in the example shape (Figure 4.10(b)).

The label h_1^C can not be assigned to a horizontal edge the example model (Figure 4.10(b)) do not contain any neighborhoods with the inside of a polygon beneath a horizontal edge.

The vertices of Figure 4.10(c) that are located at the intersection of lines parallel to h_2 and h_3 can be assigned five of the eight labels identified in Figure 4.12: $\emptyset, \mathbb{R}^2, h_2, h_3$, and $h_2 \cap h_3$.
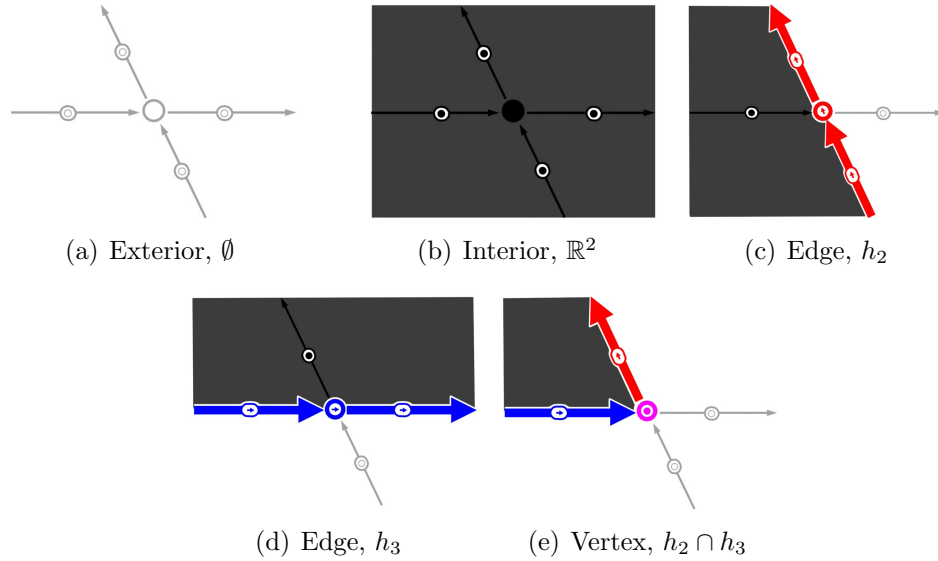
Figure 4.16: The five possible labels of vertex v in Figure 4.10(c). These are the only labels are found in the example shape (Figure 4.12) which only use h_2 and h_3.

4.5.3 Set of Possible Labels in 3D

For 3D models, each vertex figure is equal to a Boolean expression of half-spaces. In the 3D case, we will use $h_1, h_2, \ldots, h_m$ to denote the half-spaces. Each half-space is parallel to one of the input faces. The number of half-spaces m is equal to the number of distinct normals of the input polyhedra. The half-spaces are used to describe local neighborhoods. Each neighborhood corresponds to a label. Different labels are described using half-spaces as shown in Figure 4.17.

This section explains how to compute the set of all possible labels in 3D. This computation is Step 2 of Algorithm 4.1. Each possible label describes a neighborhood around a point $\mathbf{p}$ in the input model E. We explain how to describe the neighborhood around each point $\mathbf{p}$ in E. To construct a complete list of possible labels use the following procedure on each on each vertex, edge, face, and solid

1. If $\mathbf{p}$ is outside the polyhedra P_i, then $\emptyset$ describes the neighborhood around $\mathbf{p}$. If $\mathbf{p}$ is in the interior of P_i, then $\mathbb{R}^3$ describes the neighborhood.

Half-Spaces

$h_1 \cap h_3$

$h_1 \cap h_2 \cap h_3$

$h_1 \cup h_2 \cup h_3$

$h_1 \cup (h_2 \cap h_3)$

$h_1 \cup (h_2 \cap h_3^C)$

$(h_1 \cap h_2) \cup$
$(h_1 \cap h_3)$

$(h_1 \cap h_2^C) \cup$
$(h_1 \cap h_3)$
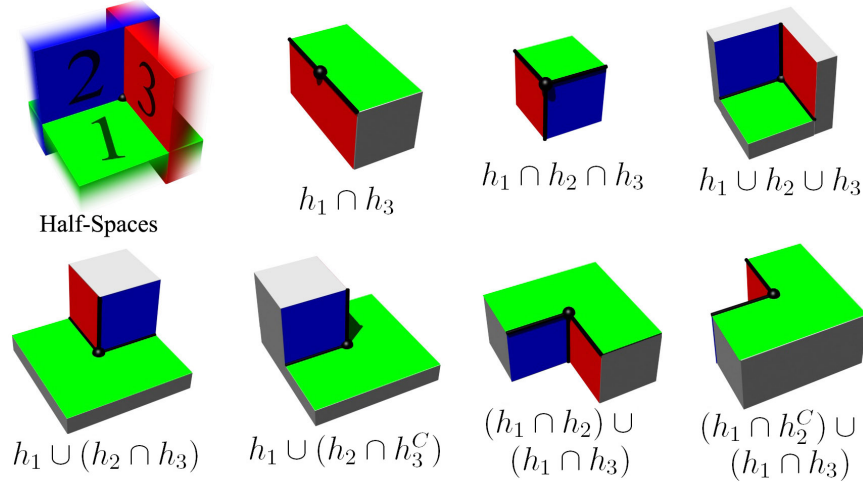
Figure 4.17: One edge and six vertex labels are described using Boolean expressions of three half-spaces. In our algorithm, every neighborhood is represented by a Boolean expression.

2. If **p** lies on a face, then suppose that h_i is the half-space parallel to the face. If the face's normal points away from h_i then h_i describes the neighborhood around **p**, otherwise h_i^C does.

3. If **p** lies on an edge, combine the expressions of the two faces it touches. For example, if they are h_i and h_j, then the neighborhood around **p** is described by $h_i \cup h_j$ if the edge has a reflex angle and $h_i \cap h_j$ if it does not.

4. If **p** corresponds to a vertex, then the procedure is more complex. Let us find every face that intersects **p**. Each face is on a plane. Let us use all of the planes the faces are on to divide the space into cells. An example of this is shown in Figure 4.18. Each cell is the intersection of several half-spaces. Every cell has points in the neighborhood of **p**. For each cell, we determine if the points within the cell and within the neighborhood of **p** are inside or outside the polyhedron E. We take the union of all cells that have points inside the polyhedron and this is the Boolean expression that represents the neighborhood surrounding **p**. Each cell is the intersection of several half-spaces and so the neighborhood at **p** is represented

as a union of intersections. These expressions can often be simplified using rules of Boolean algebra such as $(h_i \cap h_j) \cup (h_i \cap h_j^C) = h_i$. Simplified Boolean expressions for various labels are shown in Figures 4.17 and 4.18.
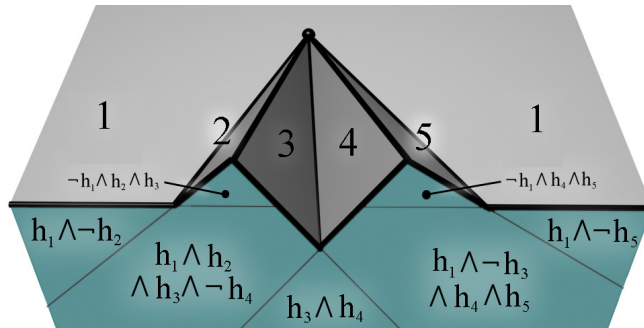
Figure 4.18: Even complex vertices can be described using a Boolean expression of half-spaces. We can find the faces that intersect the vertex and use all of the planes the faces are on to divide up the space into cells that are the intersection of several half-spaces. The cells labeled in the figure are part of the interior of the polyhedron. The label can be described as the union of all the regions inside the polyhedron. The expression for this vertex when simplified is $(h_3 \cap (h_1 \cup h_2)) \cup (h_4 \cap (h_1 \cup h_5))$.

A tetrahedron such as the one in Figure 4.19(a) has 16 possible labels: $\emptyset, \mathbb{R}^3, h_1, h_2, h_3, h_4, h_1 \cap h_2, h_1 \cap h_3, h_1 \cap h_4, h_2 \cap h_3, h_2 \cap h_4, h_3 \cap h_4, h_1 \cap h_2 \cap h_3, h_1 \cap h_2 \cap h_4, h_1 \cap h_3 \cap h_4$, and $h_2 \cap h_3 \cap h_4$.

(a) Input Model $E(\mathbf{x})$ (b) Parallel planes based on E
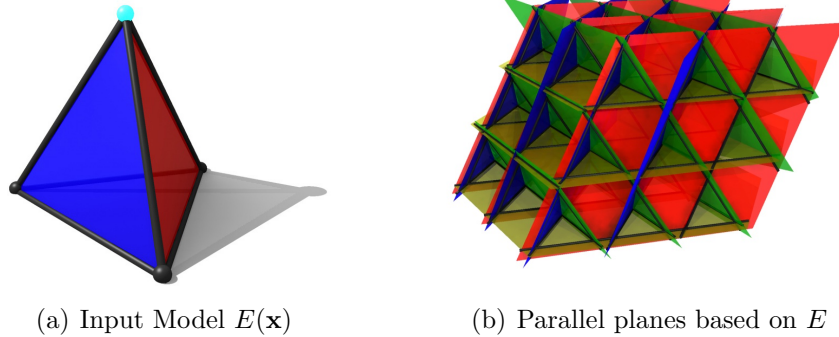
Figure 4.19: 3D Case. Planes parallel to the faces of the input divide space into solid regions, faces, edges, and vertices in which the output is created.

4.5.4 The Set of Labels for Each Vertex and Edge in 3D

A point can only be assigned a label that includes h_i in its expression if the point intersects a plane that is parallel to h_i. In the previous section, 16 possible labels were listed for a tetrahedron (Figure 4.19(a)). Of those 16 labels, only 9 labels can be assigned to a vertex that intersects planes parallel to h_1, h_2, and h_3. These labels are shown in Figure 4.20.

4.5.5 Evaluating Boolean Expressions along Edges

Sections 4.5.1 - 4.5.4 discuss how to construct a list of possible labels for each edge and each vertex. This section discusses the relationship between adjacent labels. If a label is assigned to a vertex, each adjacent edge must have a certain label. We discuss how to find this label which is Step 4 of 4.1.

The labels are described as Boolean expressions of half-spaces. Each half-space h_i has a characteristic function $h_i(\mathbf{x})$ that evaluates to 1 if $\mathbf{x}$ is inside the half-space and

113

(a) Exterior, $\emptyset$ (b) Interior, $\mathbb{R}^3$ (c) Face Labels h_1, h_2, and h_3

(d) Edge Labels $h_1 \cap h_2, h_2 \cap h_3$, and $h_1 \cap h_3$ (e) Vertex, $h_1 \cap h_2 \cap h_3$
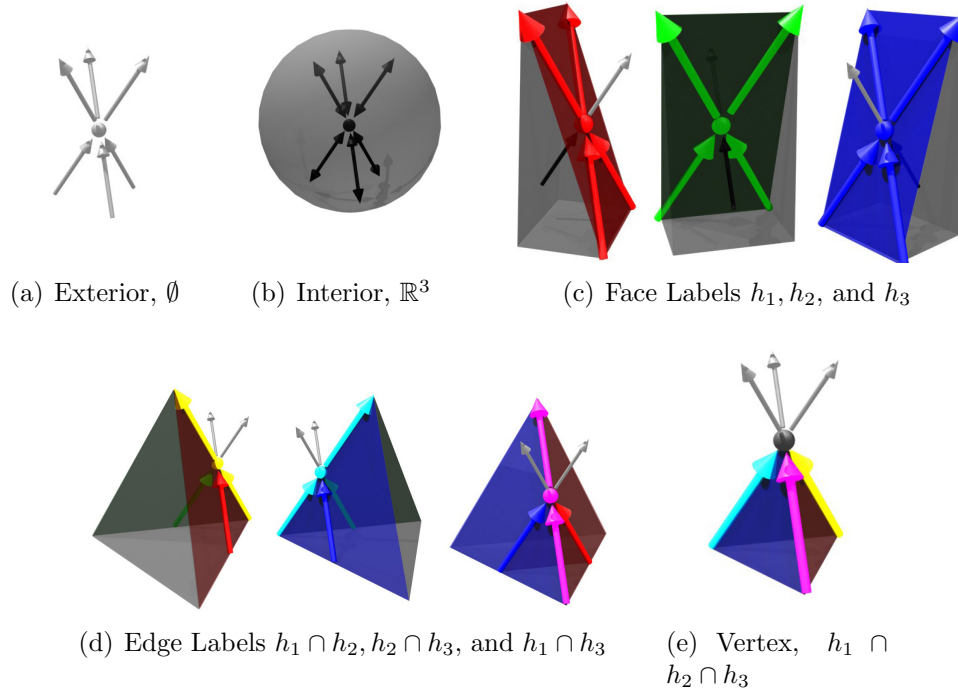
Figure 4.20: The possible labels of a 3D vertex found in the input model.

to 0 if $\mathbf{x}$ is in the opposite half-space. However, another third possibility is that $\mathbf{x}$ could intersect the plane dividing the two half-spaces. In this case, $h_i(\mathbf{x})$ is not evaluated as 0 or 1, but is left as a symbol h_i. This symbolic representation provides a convenient way to determine how the labels connect together. The characteristic functions of the sets $h_1 \cap h_2$ and $h_1 \cup h_2$ are $h_1(\mathbf{x}) \wedge h_2(\mathbf{x})$ and $h_1(\mathbf{x}) \vee h_2(\mathbf{x})$ respectively.

How the function $h_i(\mathbf{x})$ gets evaluated depends upon if point $\mathbf{x}$ intersects the plane parallel to the half-space. To keep track of point-plane intersections, we use subscripts. In this notation, the point $\mathbf{x}_{12} = \mathbf{n}_1 \times \mathbf{n}_2$ is on the planes parallel to h_1 and h_2 where $\mathbf{n}_1$ and $\mathbf{n}_2$ are the normal vectors for those planes.

The Boolean expressions describe many different neighborhoods or labels, including vertices, edges, and faces. When we evaluate the expression at a point $\mathbf{x}$, essentially we get a new expression that describes what we would encounter if we were to travel away from a neighborhood in the direction of $\mathbf{x}$ a small distance. If the expression evaluates to 0, we have traveled into empty space. If it evaluates to 1, we have traveled into an

114

object's interior. If it evaluates to h_1, we have traveled onto a face. If it evaluates to $h_1 \wedge h_2$ or $h_1 \vee h_2$, then we have traveled onto an edge. We determine if two labels can be next to one another by evaluating each label in opposite directions and then check if their evaluations are identical. For example, in Figure 4.21, the label $(h_1 \wedge h_2) \vee (h_1 \wedge h_3)$ evaluates in the down direction to $h_2 \vee h_3$. Any label that evaluates to $h_2 \vee h_3$ in the up direction can be beneath $(h_1 \wedge h_2) \vee (h_1 \wedge h_3)$ including the three labels shown beneath it in Figure 4.21.

The Boolean expressions may contain more than one object type. In this case, we evaluate each object type separately and combine the results. For example, the expression $1 \cdot (h_1 \wedge \neg h_2) + 3 \cdot h_2$ is used to describe a neighborhood in which an edge $h_1 \wedge \neg h_2$ of object 1 touches a face h_2 of object 3. If we evaluate this expression at the point $\mathbf{x}_{23}$ and if $h_1(\mathbf{x}_{23}) = 1$, then we would compute $1 \cdot h_1(\mathbf{x}_{23}) \wedge \neg h_2(\mathbf{x}_{23}) + 3 \cdot h_2(\mathbf{x}_{23}) = 1 \cdot \neg h_2 + 3 \cdot h_2$. This means that if we travel in the direction $\mathbf{x}_{23}$ we will encounter two faces that touch each other, one face from object 1 and the other from object 3.
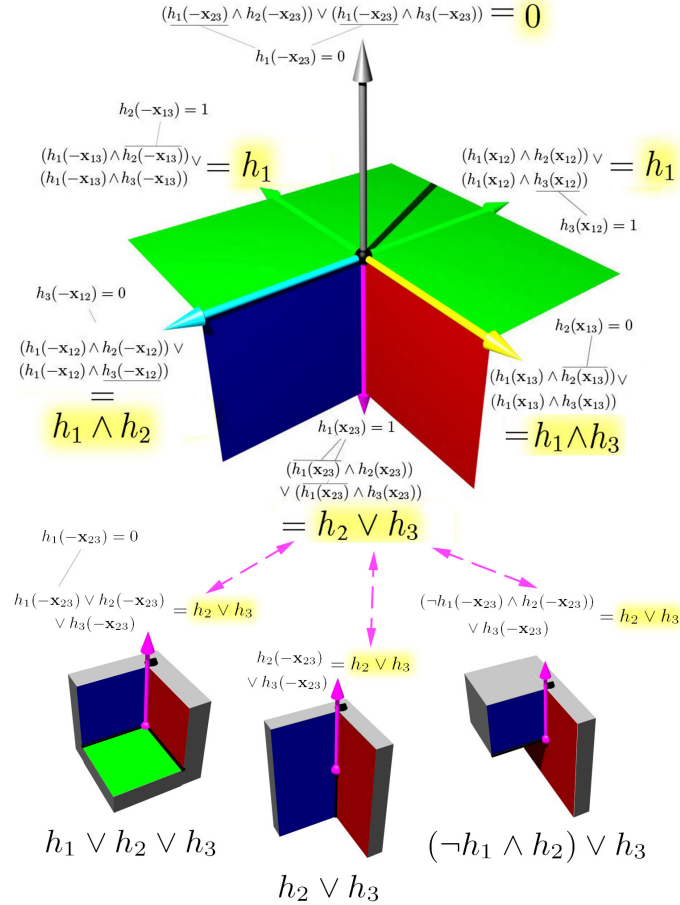
Figure 4.21: We can evaluate the labels to figure out which labels can be adjacent to them in various directions. This shows the label $(h_1 \cap h_2) \cup (h_1 \cap h_3)$ evaluated in six directions. In this figure, we assume that the direction $\mathbf{x}_{23}$ points down and is inside the h_1 half-space so that $h_1(\mathbf{x}_{23}) = 1$ and $h_1(-\mathbf{x}_{23}) = 0$. In the $\mathbf{x}_{23}$ direction, the label $(h_1 \cap h_2) \cup (h_1 \cap h_3)$ evaluates to $h_2 \cup h_3$ which is shown as the magenta edge. The only labels that can be directly beneath the label $(h_1 \cap h_2) \cup (h_1 \cap h_3)$ are labels which share the same $h_2 \cup h_3$ edge. We test every label to see which ones evaluate to $h_2 \cup h_3$ in the up direction which is $-\mathbf{x}_{23}$. Three examples of acceptable labels are shown: $h_1 \cup h_2 \cup h_3$, $h_2 \cup h_3$, and $(\neg h_1 \cap h_2) \cup h_3$. More than three acceptable labels exist.

4.5.6   Assigning Consistent Labels

The remaining steps in the continuous model synthesis algorithm (Lines 5-9 of Algorithm 4.1) follow essentially the same procedure as discrete model synthesis. The algorithm uses a catalog of possible labels C_{M_t}. Initially, C_{M_t} contains every possible assignment of labels to each edge and edge vertex as computed in Sections 4.5.1 - 4.5.4. Assignments are gradually removed from C_{M_t} until each edge and vertex is assigned only one valid label. Once all the assignments have been made, M has been computed.

Initially, no labels are assigned and C_M is full of many possible labels. At each edge or vertex, we randomly select a single label from among the choices found in C_M. The selected label is assigned to M and then C_M is updated.

An overview of the algorithm is given in Figure 4.22. Figure 4.22(a) depicts the initial configuration of C_M. Initially, every face, edge, and vertex has all possible labels present. In this example, we take a set of edges and vertices which are highlighted in yellow and assign labels to them. These assignments reduce the set of possible adjacent labels. From Section 4.5.5, we know that only certain labels are allowed to be neighbors and so we find assignments that disagree with their neighbors and remove them. The result after the first removal is shown in Figure 4.22(b). Removing these assignments reduce the set of possible labels in other edges and vertices, which compels us to remove more labels. We continue to remove labels until there are none left that need removing as shown in Figure 4.22(c). We continue to pick and assign labels from C_M and then update C_M. This continues until every edge and vertex has been assigned a single label. The shape produced in Figure 4.22(c) achieves the desired result of a triangle that is similar to the input triangle in Figure 4.10(a) and satisfies the adjacency constraint everywhere.

This procedure has much in common with discrete model synthesis including the possibility of the failure cases. It is possible that the algorithm may make an incorrect assignment that causes the list of possible assignments C_M to become empty. In this

case, the algorithm has not computed a valid and consistent set of assignments. We discuss these types of failures in Sections 3.3.4-3.3.7, so we do not revisit this issue in any detail. If a failure occurs, we use the strategy introduced in Section 3.3.6 which is instead of creating the whole modeling space in one pass, we modify small blocks of the model as shown in Figure 4.23. The algorithm is much less likely to make an incorrect assignment if it is only modifying a small part of the model. Empirically, we have found that our algorithm often succeeds when modifying a volume of 10 x 10 x 10 or smaller (in units of plane spacings). Sometimes the algorithm works when modifying huge volumes, since there are infallible input models which never cause failures such as those shown in Figures 4.35, 4.28 - 4.32. But even in the worst case, there is never a possibility that an over-constrained input would cause an immediate failure since at least one solution exists for all inputs. A solution must only satisfy the adjacency constraint. The input model trivially satisfies the constraint as do stretched copies of the input. A solution can always be found, although it may be necessary to modify the output in small blocks to find it.

4.5.7 Time and Space Complexity

The time and space complexity of continuous model synthesis depends upon the number of distinct face normals and the number of parallel planes created for each normal. The number of normals is called m and the number of planes for each normal is called n. The value of n depends on the volume of the output model and how far apart the planes are spaced. The output volume and the plane spacing are both specified by the user. Increasing the output volume has the same effect as decreasing the plane spacing. Wherever three planes intersect, a vertex is created. The number of vertices is less than or equal to $\binom{m}{3}n^3$. The number of vertices is lower when some sets of three planes do not intersect. For example, if the input model is a $m-1$ sided prism, then number of vertices is $\binom{m-1}{2}n^3$. The number of vertices is proportional to the time and space complexity of
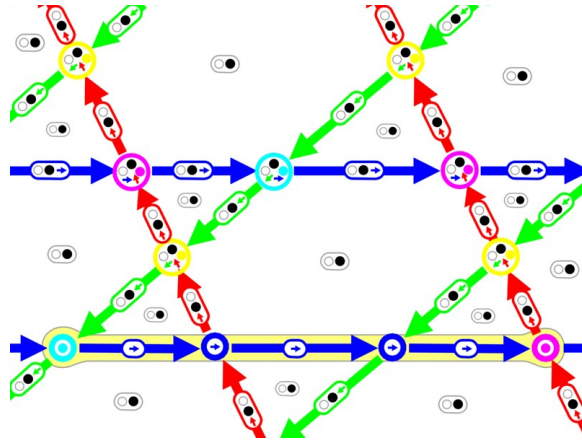
(a) We begin by allowing all possible labels to be at every face, edge, and vertex. Then we assign a few labels to the highlighted locations.
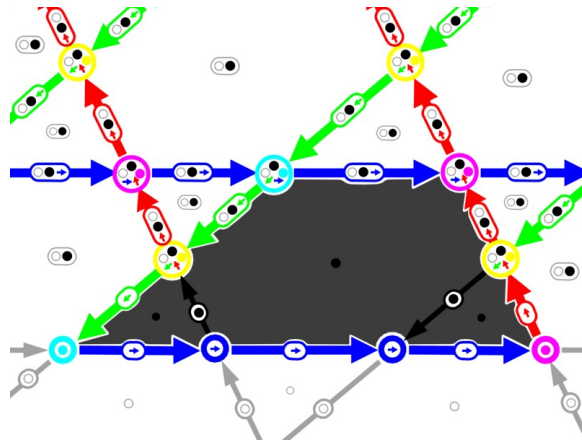
(b) Using Section 4.5.5, we remove adjacent labels that are incompatible with the previously assigned labels.
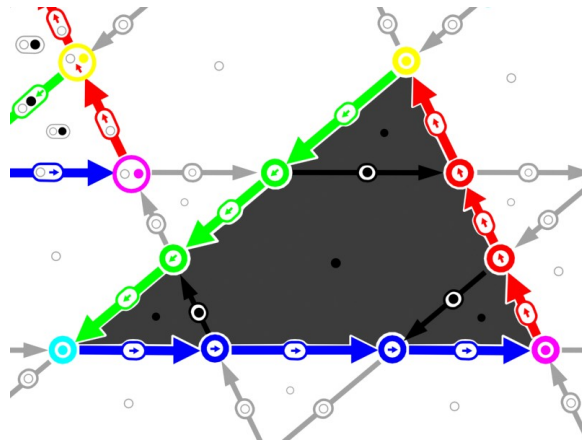
(c) After labels are removed, their removal causes other labels to be removed. The result after all incompatible labels have been removed is a triangle.

Figure 4.22: The evolution of the list of possible labels C_{M_t} over time.

(a) Initial model with exterior assignments.

(b) Clear assignments in part of the model.

(c) Modify part of the model.

(d) Clear a new part.

(e) Modify new part.
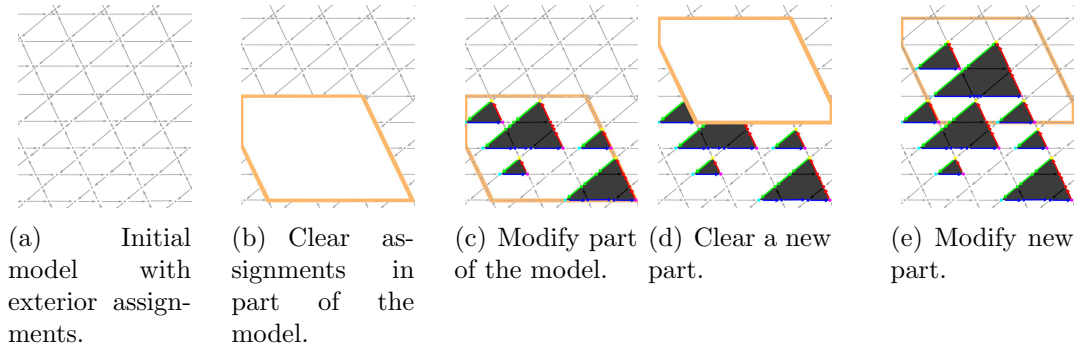
Figure 4.23: This figure shows how the model can be created by only modifying part of the model at once. Each part is modified so that it is consistent with the rest of the model along the border. This is similar to Figure 3.9 in the discrete case.

the algorithm. The number of vertices, the time complexity, and the space complexity are all $O(m^3n^3)$ and $\Omega(m^2n^3)$.

4.5.8 Spacing the Planes

As Section 4.5.3 explains, under the parallel plane assumption, a vertex can only be assigned a particular label if it intersects a plane parallel to each of the half-spaces it used in the Boolean expression. Labels that use four or more half-spaces pose a problem, since these labels are only allowed at vertices that intersect four or more planes. Figures 4.24 and 4.18 show examples of labels that involve four or more half-spaces. The simplest example is the vertex at the top of a four-sided pyramid. This vertex is not a trihedral vertex. Trihedral vertices can easily be produce by the algorithm. Trihedral vertices only use three half-spaces; three half-spaces require three planes to intersect; and those three planes must intersect at some point. But four planes may not intersect at a point. In these cases, we need to choose the plane spacing so that do four planes intersect at a point.

We first discuss how to compute all the locations where three planes intersect and then discuss how to ensure that a fourth plane intersects the same locations. Let $\mathbf{n}_1, \mathbf{n}_2$, and $\mathbf{n}_3$ be the normals of three sets of planes. Within each set, all the planes are parallel

$h_1 \cap h_2 \cap h_3 \cap h_4$

$1 \cdot h_1 \cap h_2 \cap h_3$
$+ 2 \cdot h_2 \cap h_3^C \cap h_4$

Figure 4.24: Examples of neighborhoods that involve more than four half-spaces

and evenly spaced. Let s_1, s_2, and s_3 be the spacing between the planes within each of the three sets. The first and second sets of planes intersect at along lines that point in the $\mathbf{n}_1 \times \mathbf{n}_2$ direction. If $\mathbf{p}$ is a point on the line, then the point $\mathbf{p}'$ is also on the line if $\mathbf{p}' - \mathbf{p} = (\mathbf{n}_1 \times \math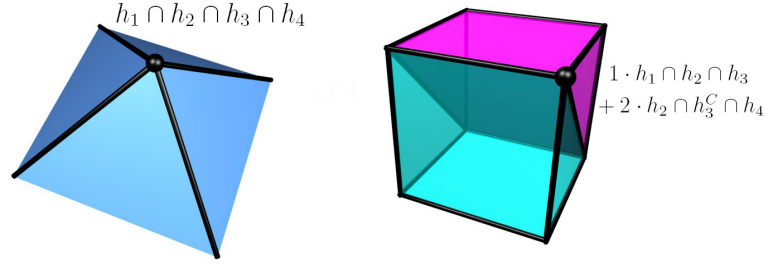bf{n}_2)t$ for any scalar t. If $\mathbf{p}$ intersects a plane from the third set, then $\mathbf{p}'$ also does if $\mathbf{n}_3 \cdot (\mathbf{p}' - \mathbf{p}) = c_3 s_3$ for some integer $c_3 \in \mathbb{Z}$. Solving for t, we find that $t = \frac{c_3 s_3}{\mathbf{n}_3 \cdot (\mathbf{n}_1 \times \mathbf{n}_2)}$ and therefore,

$$\mathbf{p}' = \mathbf{p} + c_3 s_3 \frac{\mathbf{n}_1 \times \mathbf{n}_2}{\mathbf{n}_3 \cdot (\mathbf{n}_1 \times \mathbf{n}_2)} \tag{4.23}$$

for some $c_3 \in \mathbb{Z}$. This gives us a set of points along the $\mathbf{n}_1 \times \mathbf{n}_2$ direction where the three planes intersect. The same argument can also be applied to the directions given by $\mathbf{n}_1 \times \mathbf{n}_3$ and $\mathbf{n}_2 \times \mathbf{n}_3$. Three planes intersect at the points

$$\mathbf{p}' = \mathbf{p} + \frac{c_1 s_1 \mathbf{n}_2 \times \mathbf{n}_3}{\mathbf{n}_1 \cdot (\mathbf{n}_2 \times \mathbf{n}_3)} + \frac{c_2 s_2 \mathbf{n}_1 \times \mathbf{n}_3}{\mathbf{n}_2 \cdot (\mathbf{n}_1 \times \mathbf{n}_3)} + \frac{c_3 s_3 \mathbf{n}_1 \times \mathbf{n}_2}{\mathbf{n}_3 \cdot (\mathbf{n}_1 \times \mathbf{n}_2)} \tag{4.24}$$

for any $c_1, c_2, c_3 \in \mathbb{Z}$. Each different integer value of (c_1, c_2, c_3) gives us a different equation and intersection point. The resulting intersection points form a 3D lattice. The three planes always intersect regardless of how they are spaced, but it is more difficult to ensure that four planes intersect.

If $\mathbf{p}$ intersects a plane from the fourth set, then $\mathbf{p}'$ also does if $\mathbf{n}_4 \cdot (\mathbf{p}' - \mathbf{p}) = c_4 s_4$ for some integer $c_4 \in \mathbb{Z}$. The point $\mathbf{p}'$ intersects the fourth set of planes if

$$s_4 = s_1 \frac{c_1 \mathbf{n}_4 \cdot (\mathbf{n}_2 \times \mathbf{n}_3)}{c_4 \mathbf{n}_1 \cdot (\mathbf{n}_2 \times \mathbf{n}_3)} + s_2 \frac{c_2 \mathbf{n}_4 \cdot (\mathbf{n}_1 \times \mathbf{n}_3)}{c_4 \mathbf{n}_2 \cdot (\mathbf{n}_1 \times \mathbf{n}_3)} + s_3 \frac{c_3 \mathbf{n}_4 \cdot (\mathbf{n}_1 \times \mathbf{n}_2)}{c_4 \mathbf{n}_3 \cdot (\mathbf{n}_1 \times \mathbf{n}_2)} \tag{4.25}$$

for some $c_4 \in \mathbb{Z}$. By solving Equation 4.25, four planes can intersect at multiple points. Equation 4.25 represents multiple equations that need to be solved since each combination of c_1, c_2, and c_3 results in another equation. If we solve Equation 4.25 for the $(c_1, c_2, c_3, c_4) = (1, 0, 0, 1)$ and $(0, 1, 0, 1)$ and $(0, 0, 1, 1)$, then it will hold for any combination of c_1, c_2, and c_3. This gives us three equations and four unknowns s_1, s_2, s_3, and s_4. By solving for these three linear equations, we produce a 3D lattice of points where a non-trihedral vertex label may appear. However, this only takes care of a single non-trihedral vertex label. There may be more non-trihedral vertices in the input and they would each require more equations to be solved. There are even more difficult vertex labels to handle such as the vertex shown in Figure 4.18 which involves five half-spaces. These require solving more linear equations.

In the end, we may have an underconstrained or an overconstrained set of linear equations. An overconstrained set of equations occurs when the input model does not fit well on within a lattice. One example of an input shape that produces overconstrained equations is a five-sided pyramid. These overconstrained equations can be handled in several ways. One approach is to add many more planes, but this increases the computational cost of the overall algorithm. Another approach might be to modify the normals just enough so that the shapes better fit on a lattice, but not so much that the normals significantly change the results. A third option is to ignore a few of the equations. In this case, non-trihedral vertices will be generated at fewer locations, but this might be adequate to produce a good final result. Investigating these options is an important topic for future research.

4.6 Additional User-Defined Constraints

Sections 4.3 - 4.5 present algorithms for generating models that satisfy the adjacency constraint defined in Section 4.2, but there are other user-defined constraints that can be added to give the user more control over what is generated, including:

Dimensional Constraints: Many objects have predetermined dimensions. Cars, road lanes, and chairs have a certain width. Stair steps and building floors have a certain height. Bowling balls and pool tables have a predetermined size. Without constraining the dimensions of the objects, the synthesis algorithm could easily generate roads too narrow to drive across, steps too tall to walk up, ceilings too close to the ground, and bowling balls too large to bowl. Dimensional constraints allow the user to fix the dimensions of the objects so that they are always sized realistically.

Algebraic Constraints: Some objects do not have predetermined dimensions, but instead must satisfy an algebraic relationship between their dimensions. An example might be that an object's length must be twice its height. These constraints are especially useful for curved objects.

Connectivity Constraint: Many objects look unnatural if they are not connected to a larger whole. One example is a road network. An isolated loop of road looks unrealistic when it is disconnected from the all the other roads. All of the roads in a city are normally connected by some path. This defines a connectivity constraint which can be used to eliminate the possibility of an isolated loop and create fully connected roads.

Large-Scale Constraints: The user might have a floor plan or a general idea of what the model should look like on a macroscopic scale. For example, the user might want to build a city with buildings arranged in the shape of a circle or a triangle or a symbol. The user can generate such a model by using large-scale constraints. These constraints are specified on a large volumetric grid where each voxel records which objects should appear within it.

4.6.1 Dimensional Constraints

We would like to give the user greater control over the dimensions of the output. The user should be able to control if an object can scale in a particular direction. For example, a user might specify that a road must have a particular width. Along its width, the road can not scale, but along its length, the road can scale to any length. The ability to fix the dimensions of some objects is important for creating realistic models.

Dimensional constraints are already included in the approach described in Section 4.4 using Minkowski sums, but they are missing from the algorithms in Section 4.5. Since the objects are created on sets of evenly spaced planes, the lengths of object is an integer multiple of the plane spacing. If the user wants to constrain the dimensions of an object to a non-integer multiple like 1.5-plane spaces, this can be a problem. One solution is to space the planes more closely. If they are spaced twice as close, an object that was 1.5-plane spaces wide would become three planes wide which better because it is a round number. This is exactly the strategy that was used for discrete model synthesis in Figure 4.2(b). Often there is an even simpler solution since objects with dimensional constraints are often next to objects without them and the two objects can be attached together to produce a round number. For example, it might be possible to combine an object 1.5 spaces wide with 0.5 spaces of empty space to produce an object two plane spaces wide which is better because it is a round number.

Even though objects may be two, three or more plane-spaces wide, we only need to consider the issue of how to ensure that an object is exactly one-space wide since we can easily create objects exactly two or three spaces wide simply by attaching a few one-space wide objects together.

Figure 4.25 shows an example of how this constraint is imposed. The objects can never grow wider than one plane-space if every time they intersect a plane they stop. In order to stop their growth, we disallow all labels in which the object passes through the plane. The object passes through the plane if $h_2 \cap B \neq \emptyset$ and $h_2^C \cap B \neq \emptyset$ where

h_2 is the half-space parallel to the plane and B is the Boolean expression describing the label. By removing all label where $h_2 \cap B \neq \emptyset$ and $h_2^C \cap B \neq \emptyset$, we constrain the objects to be exactly one plane-space wide.
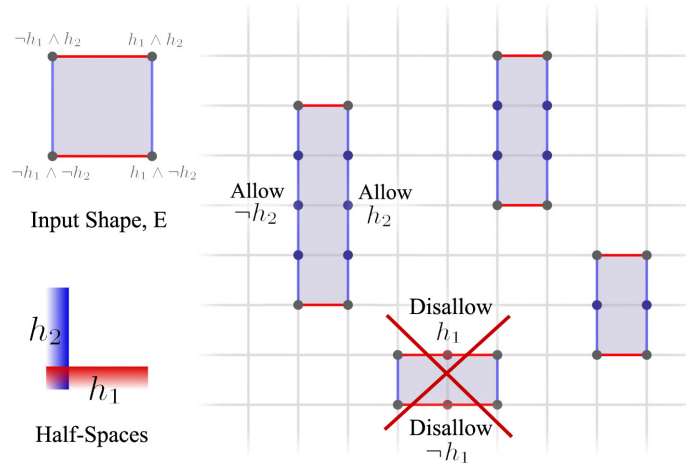
Figure 4.25: Dimensional Constraint. To create objects that are only one plane space wide horizontally, we disallow any labels which pass through the vertical h_2 planes such as h_1 or $\neg h_1$. This ensure that the generated model satisfies the dimensional constraint.

4.6.2 Connectivity Constraints

In many applications, it is important to control how the objects connect together. For example, the connections are important when creating cities with roads. In most cities, one could choose any two points on a road map and find a path that connects them. However, the model synthesis algorithms could generate isolated loops or cycles of road networks that are not connected to each other. We can address this problem by changing the order in which the labels are assigned. First, a starting location is chosen at random and an object (e.g. road) is created there. Then the roads are all grown out from this initial seed. This means that we only assign road labels to vertices that are already next to a road. By growing out from a single seed, the generated roads are fully connected.

A fully connected object is just one of several options to consider. One alternative to use the original algorithm which does not use seeds and assigns the labels in any order.

This can be useful when the user wants to create more isolated objects. A third option fits in between the other two. The user might not want everything to be connected, but might not want many small isolated objects either. The user may want a few large isolated objects. To accomplish this goal, everything could be grown from a few seeds, instead of just one.

4.6.3 Large-Scale Constraints

We would also like to give the user more control over the large-scale structure of the output. The user might have a general idea of where certain types of objects should appear. Each object has a particular probability that it will appear at any location in space. Generally, we assign each label an equal probability of being chosen, but we could easily modify the probabilities so that they are higher for those objects the user wants to appear within certain areas. The user could even set some probabilities to be zero at some places. If a label's probability drops to zero, we can remove it entirely and then propagate the removal as usually done when assigning labels (see Section 4.5.6). By changing these probabilities, we can create cities and other structures in the shape of various symbols and other objects. We can also generate multiple outputs, evaluate how well they match the user's desired goal, and select the best output. Figure 4.26 shows several models that were constructed in the shape of the letters "GPM".

4.6.4 Algebraic Constraints and Bounding Volumes

As Section 4.5.7 explains, handling curved input models with many distinct normals is computationally expensive because of the large number of planes and vertices that would need to be created. However, the number of distinct normals can be greatly reduced by using bounding boxes or other bounding volumes in place of complex objects. The algorithm could proceed using the bounding volumes in place of the input model and once the output model M is generated complex objects can be substituted back into M.
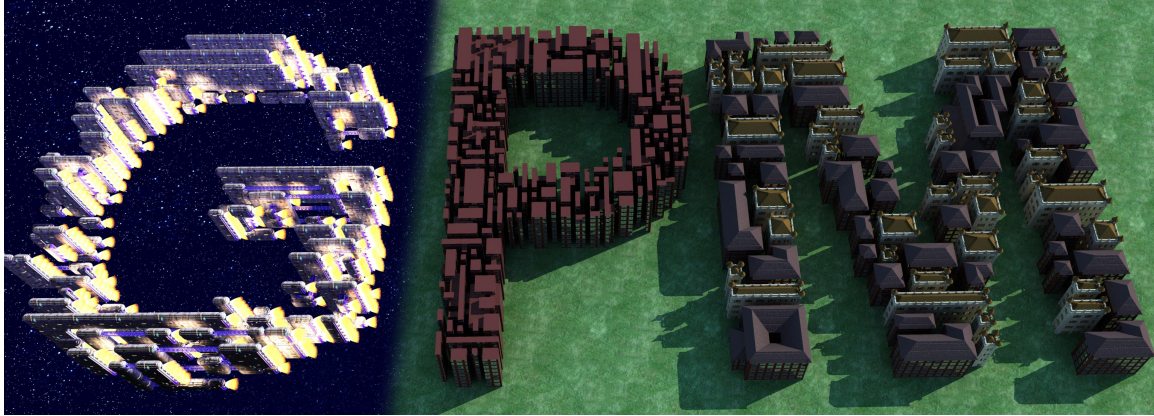
Figure 4.26: Large-scale constraints are used to build spaceships in the shape of the letter 'G', rectangular buildings in the shape of the letter 'P', and buildings from Figure 4.36 in the shape of the letter 'M'.

When using bounding volumes, the user might want to constrain the dimensions of the objects in a slightly different way. The original algorithm had no dimensional constrains. The object's dimensions could scale freely in all directions. In Section 4.6.1, we discussed how to fix the size in each direction. A third option is to let an object scale, but to require that it must scale uniformly in two or three directions. This is useful for objects in bounding volumes. For example, the cylinder in Figure 4.27 only remains cylindrical if its x and y coordinates scale uniformly $s_x = s_y$. It is free to stretch along the z-coordinate by any amount. In order to get its x and y coordinates to scale equally, we can place a bounding box around the cylinder and the cut the box into two halves along the diagonal creating two triangular prisms shown in Figure 4.27. Since model synthesis scales triangular objects uniformly in two dimensions, the output will be scaled identically in x and y, $s_x = s_y$ and the cylinder can be substituted back in the shape.

The user may want to be even more restrictive and require the scalings to be uniform in all the directions $s_x = s_y = s_z$. For example, the dome in Figure 4.27 remains spherical only when scaled uniformly. This can be accomplished by placing a bounding box around the sphere and cutting off a tetrahedron as shown in Figure 4.27. Since

127

(a) Input   (b) Output with Bounding Volumes

(c) Output without Bounding Volumes
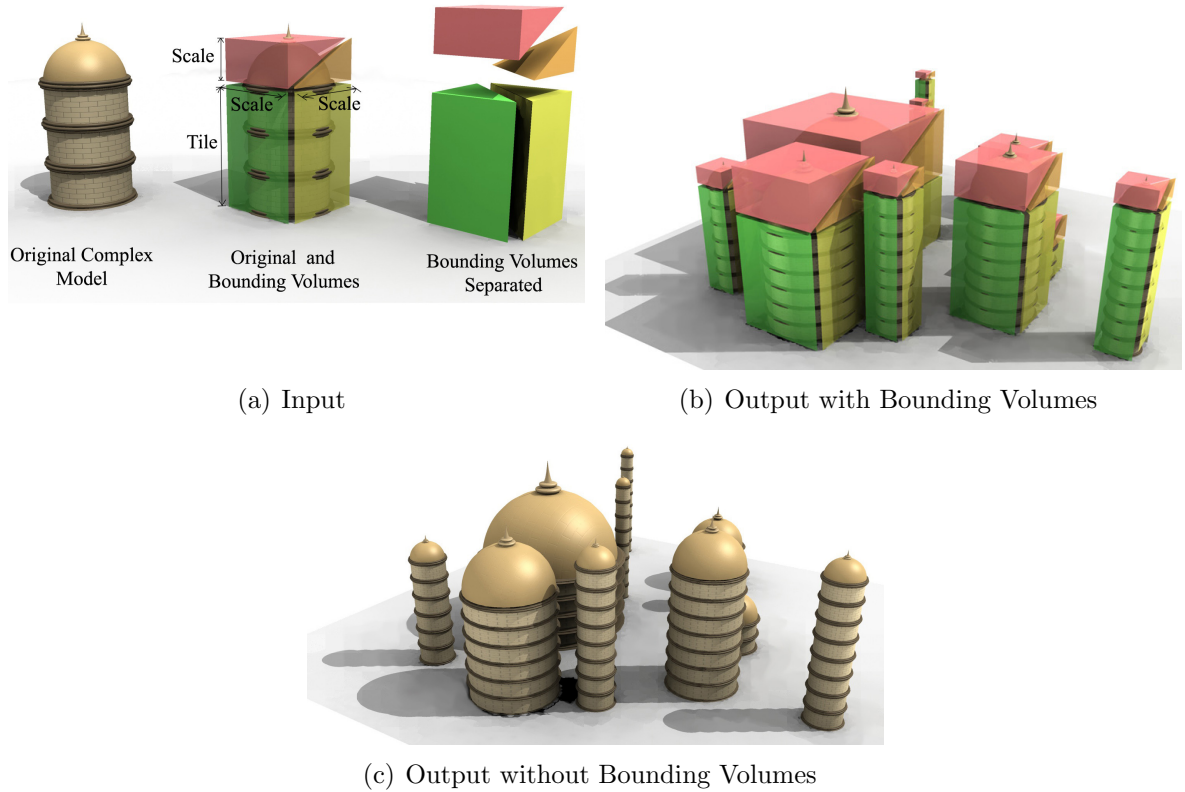
Figure 4.27: Because model synthesis is inefficient on curved models bounding volumes are used to simplify the geometry (a). The bounding boxes are cut into two objects, so the dome will scale uniformly in all directions and the cylinder will scale uniformly in x and y. The output is generated and the complex original shapes are substituted back in (b,c). Some of the corners of the box intersect five faces.

model synthesis scales tetrahedra uniformly in all directions, the output will contain only uniformly scaled copies of the bounding box.

4.7 Results

Figures 4.26 - 4.40 show a wide variety of different models generated using our algorithm. The generated models are large and detailed and would be difficult and time-consuming to model them manually using a CAD or authoring system. Each output model was generated in less than two minutes as shown in Table 4.1. The total time spent by the human user is also short. The user only supplied the size of the output and the example

models which are quite simple. In order to demonstrate how simple the input models can be, they were create using only a few dozen polygons as shown in Table 4.1, but they could be much larger. They were manually created in a few minutes using 3D Studio Max. All of the displayed images include artistic decorations to the vertices and edges of the models and some include complex objects that were substituted back from bounding volumes. The polygon count does not include any of these decorations. The exact same algorithm generated all the models shown in Figures 4.26 - 4.40 without changing anything except the input model, the output size, and a few user-defined constraints. The output models do not simply contain copies of the input model, but contain interesting new features not found in the input.

Figures 4.26, 4.27, 4.35 - 4.40 show a variety of models that were generated with additional user-defined constraints. Dimensional constraints are used in Figure 4.35 to give the platforms and beams a fixed thickness. They are also used to constrain the width of the road in Figure 4.38, the width of the spacecrafts in Figure 4.37, the size of the pipes in Figure 4.39, and the width of the roller coaster track in Figure 4.40. Connectivity constraints are used in Figures 4.35, 4.36, and 4.37 to grow the objects out from a few seeds. This controls the distribution of the objects and prevents them from being all crowded together. The roads in Figure 4.38 and the pipes in Figure 4.39 are fully connected to a single seed. In Figure 4.37, the parts of the spaceships are connected by beams which have gaps in between them. The gaps in the spaceship model demonstrate that model synthesis can generate shapes which have a high genus. Figure 4.29 also has a high genus. Bounding volumes were used in Figures 4.35 and 4.37 to generate curved objects.
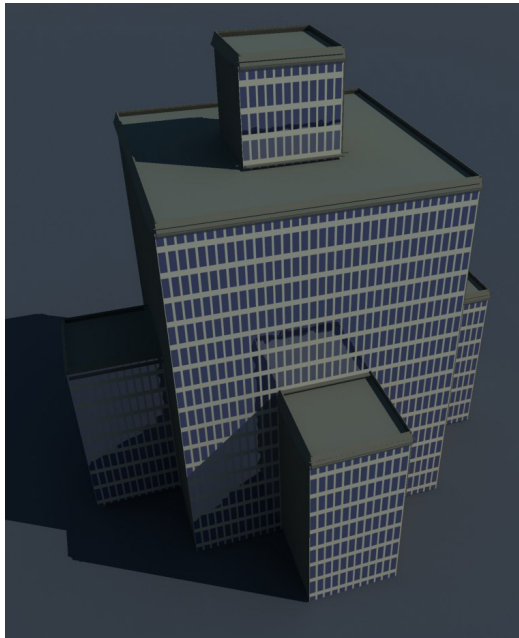
(a) Example Model
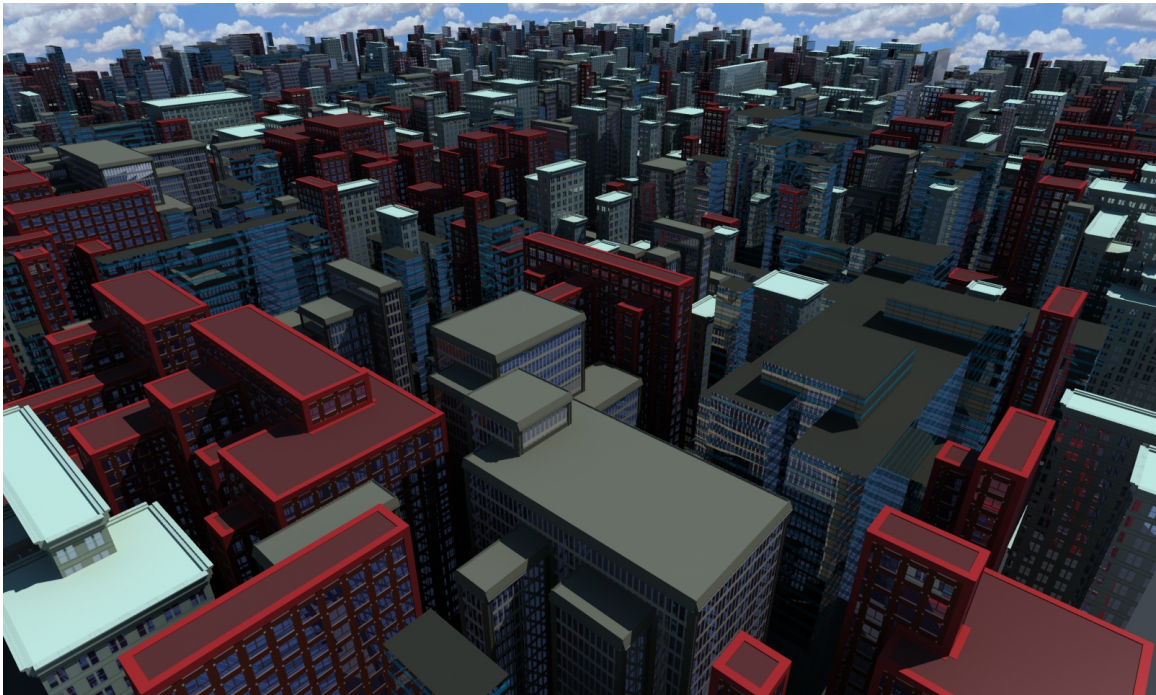
(b) Synthesized Model

Figure 4.28: Model synthesis is used to generate office buildings. The models shown in (b) are automatically generated from the example model (a). Different textures are applied to different buildings, but the shape of each building resembles the shape of the input. The output shapes were generated in under two minutes.
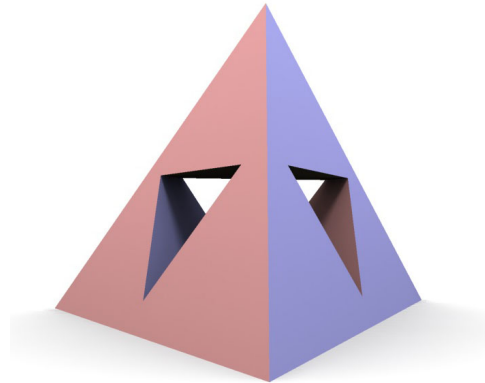
(a) Input Example
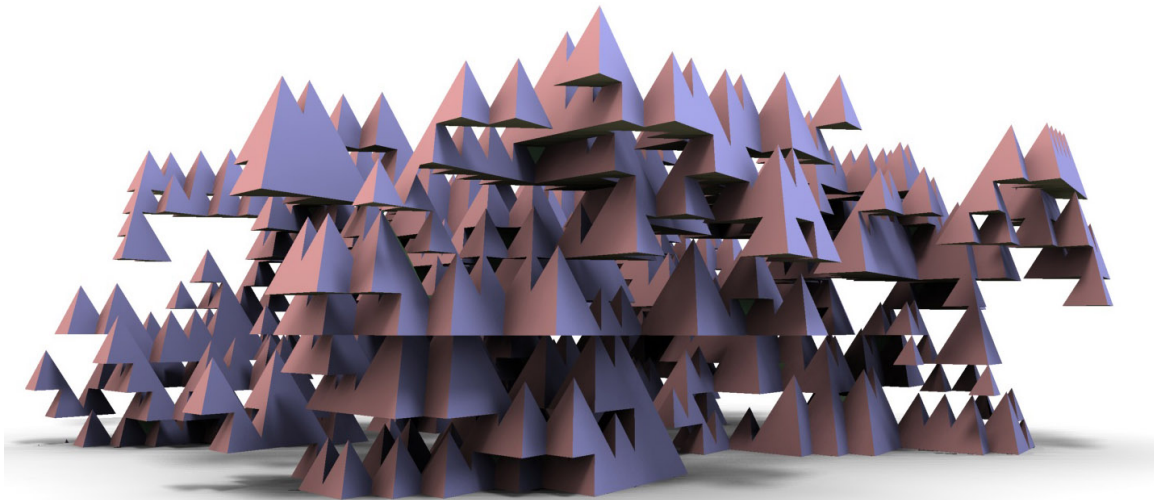
(b) Output Synthesized Model

Figure 4.29: The Sierpinski Tetrahedron (a) is used as an input to generate fractal structures (b).
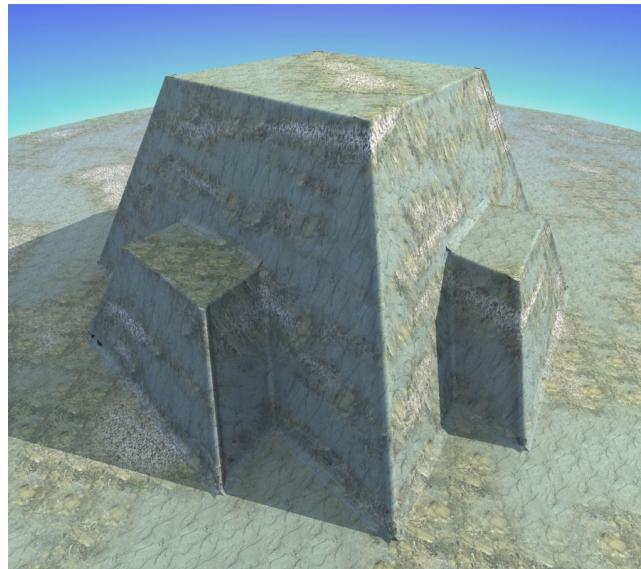
(a) Input Example
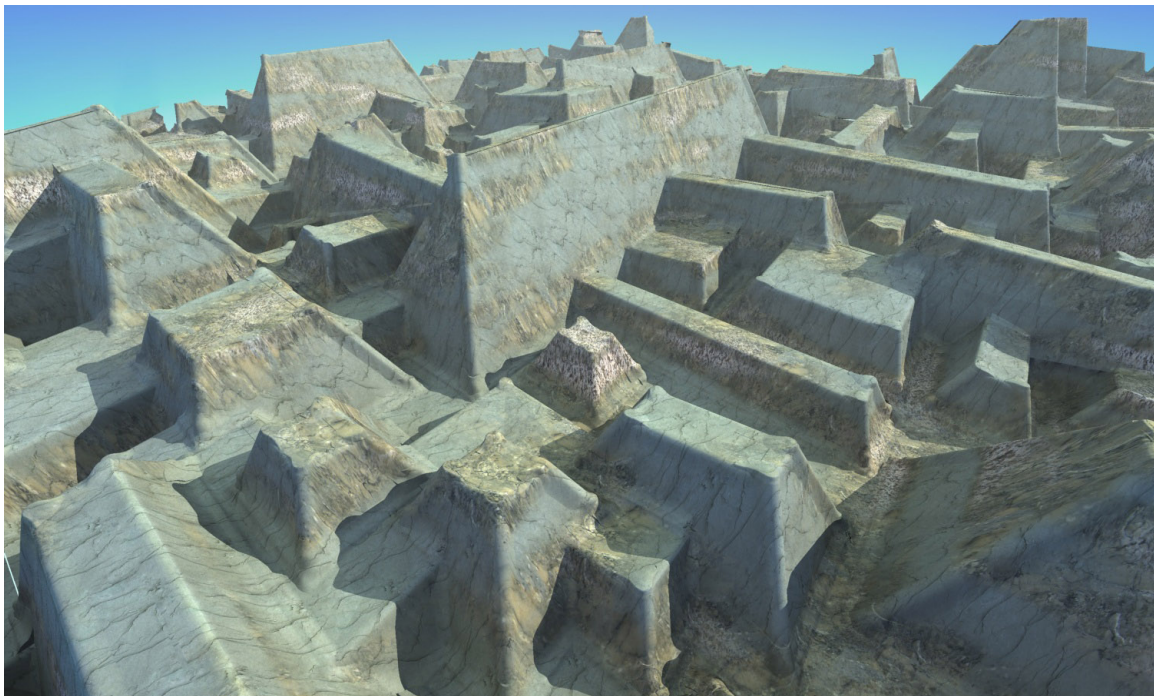
(b) Output Synthesized Model

Figure 4.30: From the example model (a), rocky terrain is generated (b).
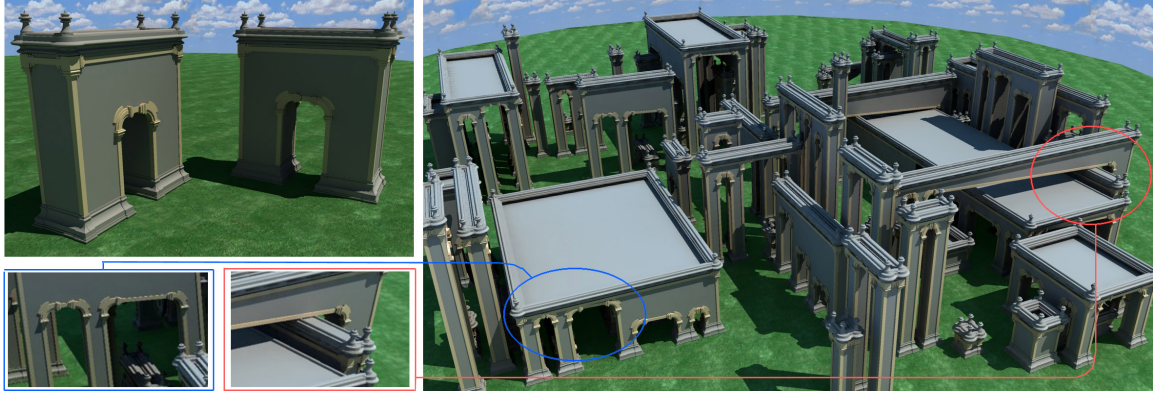
Figure 4.31: From the input example model (left) many arches are synthesized (right). The output contains interesting new variations not found in the input such as structures with multiple arches and arches passing over arches (insets).

4.8 Limitations

The continuous model synthesis algorithm has several important limitations. Some of these limitations are a product of the parallel plane assumption. The algorithm may be unable to produce good output models satisfying the parallel plane assumption. Other limitations are caused by the algorithm requiring excessively large amounts of memory or computation time in certain cases.

4.8.1 Limitations from the Parallel Plane Assumption

The parallel plane assumption makes the approach in Section 4.5 much easier to implement than the approach described in Section 4.4 which uses Minkowski sums. But it also is the source of many limitations since it forces the output model to be structured in a particular way.

Under this assumption, the models are all generated on sets of parallel planes and this may require the input shape to have a structure that fits on a grid. This is similar in some ways to the limitations with discrete model synthesis, but the limitations are less serious for continuous model synthesis. Continuous model synthesis can easily handle many of the input models that discrete model synthesis has trouble with such as the non-
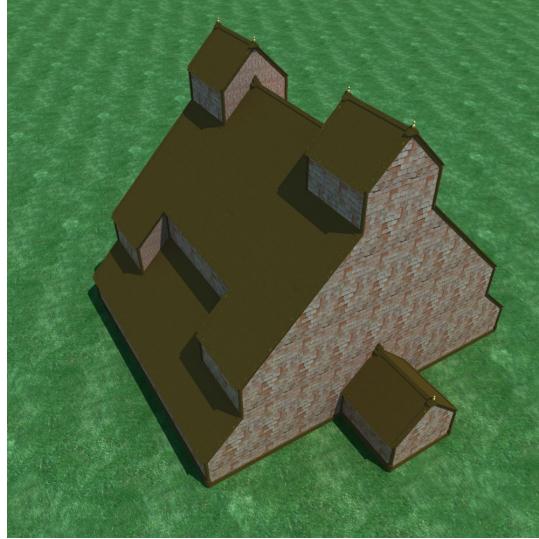
(a) Input Example Model
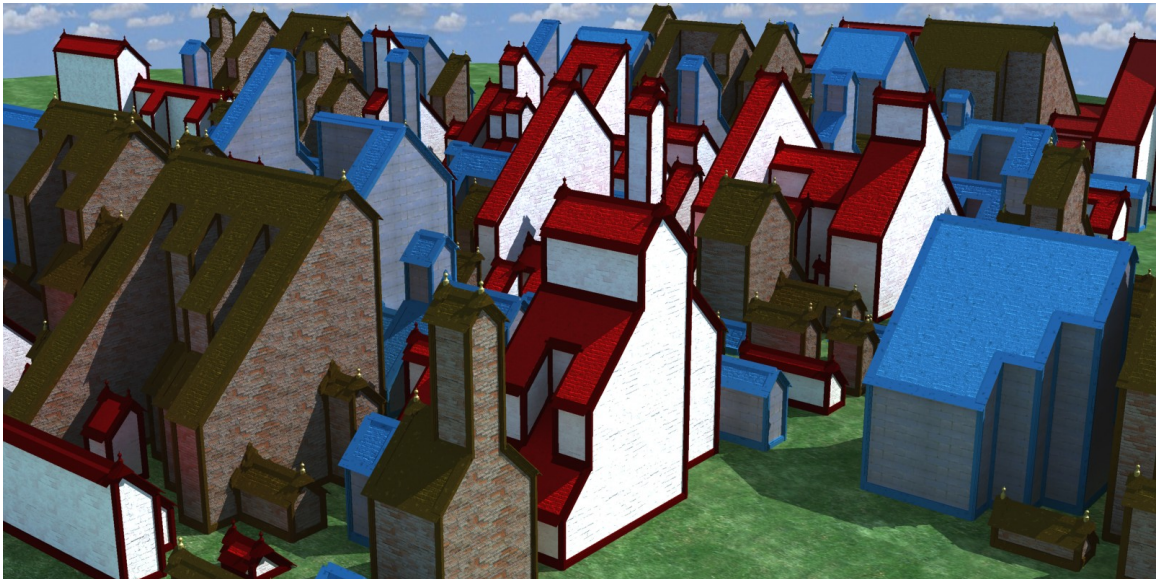
(b) Output Synthesized Model

Figure 4.32: From the input model (a), houses are automatically generated (b). Each house has a complex and unique roof structure.
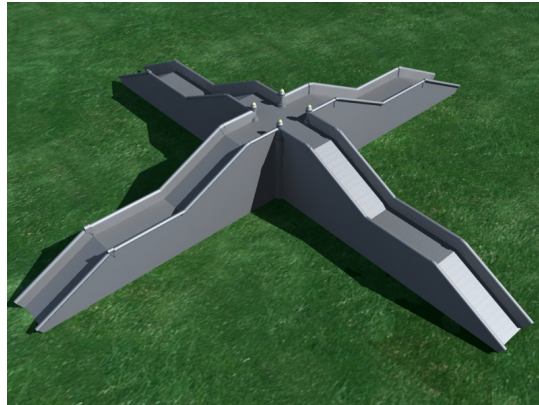
(a) Input Example Model
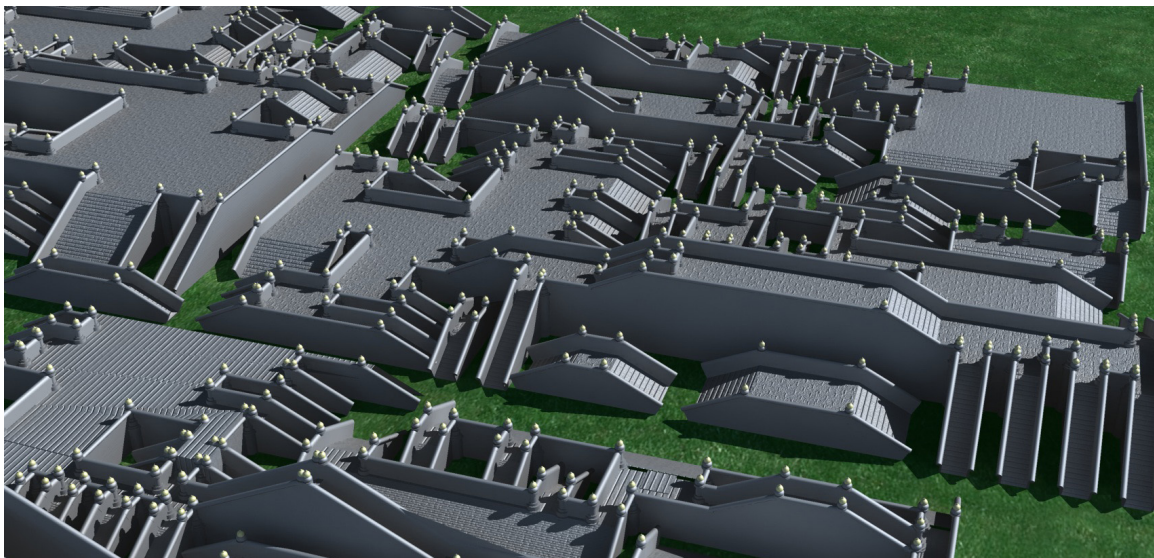
(b) Output Synthesized Model

Figure 4.33: From the input model (a), stairs are automatically generated (b).
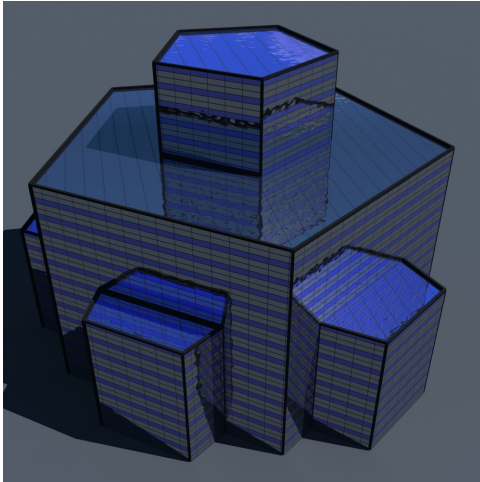
(a) Input Example Model
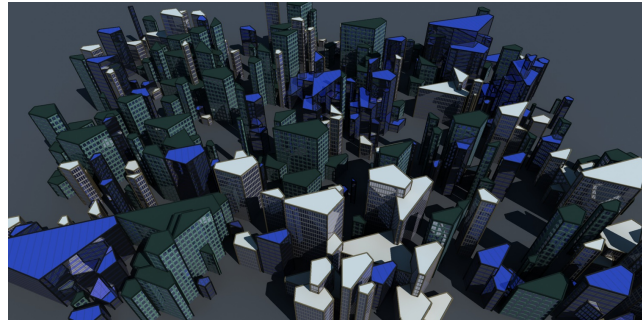
(b) Output Synthesized Model
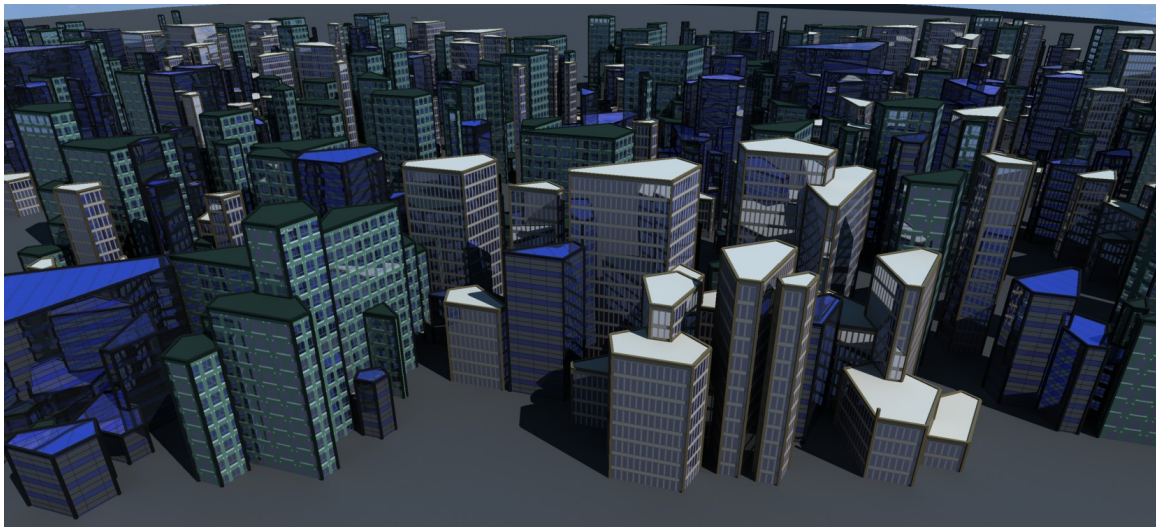
(c) Output Synthesized Model Different View

Figure 4.34: From the input example model (a) pentagonal buildings are synthesized (b,c). Most of the faces are not aligned with the axes. Most buildings are complex combinations of many pentagonal shapes and have a unique shape.
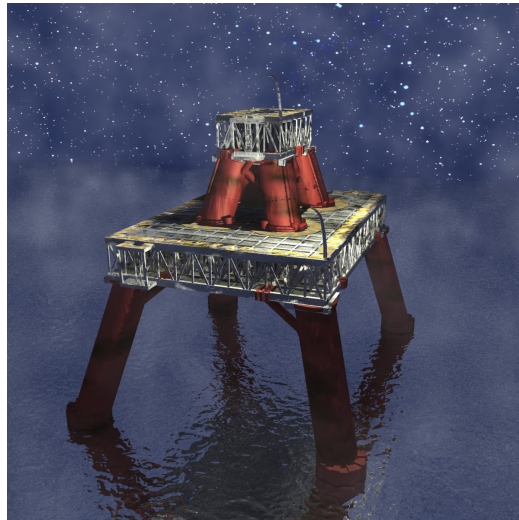
(a) Input Model
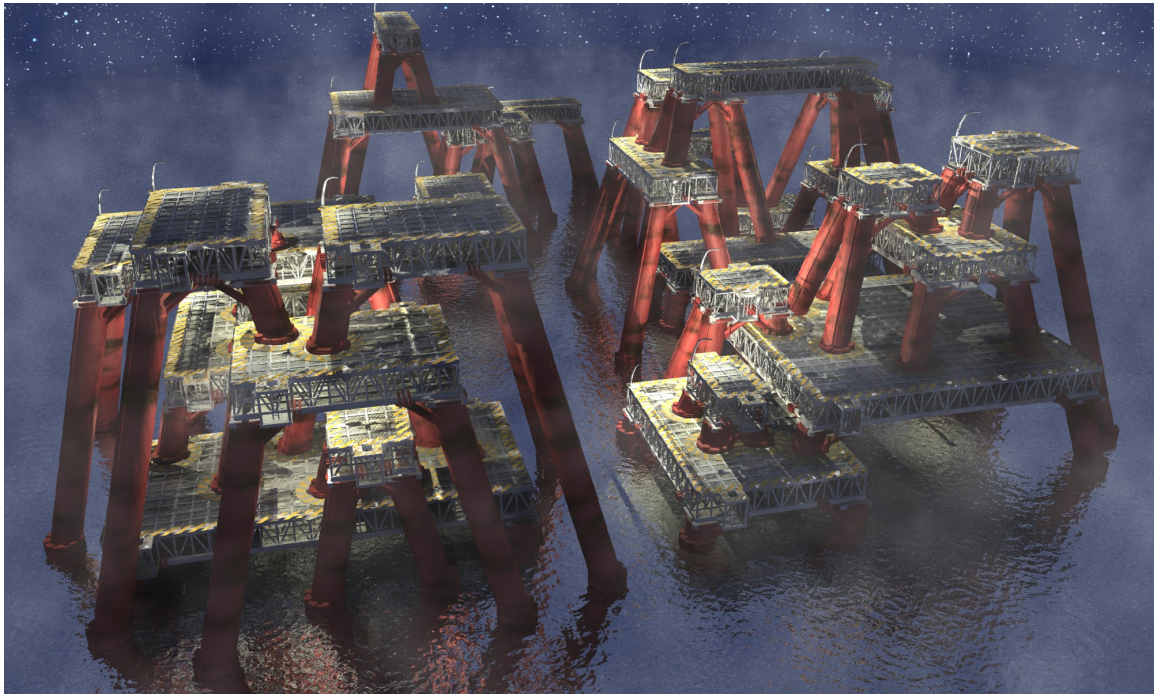
(b) Output Model

Figure 4.35: (a) From an example model specified by the user, (b) a model of several oil platforms is generated automatically by our algorithm. The shape of the output resembles the input and fits several dimensional and connectivity constraints. The height of the platform and the length and width of the beams are constrained to have a particular size. The shapes are constrained to be in four connected groups. Our algorithm can generate the new model in about half a minute.
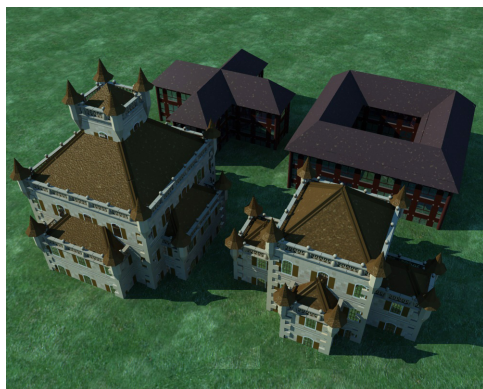
(a) Input Model

(b) Output Model

Figure 4.36: Many complex buildings (b) are generated from four simple ones (a). This model contains many non-trihedral vertices which are circled. These vertices are more difficult to create. The result also uses the connectivity constraint to space the buildings apart which gives the buildings more room to develop into more interesting shapes.
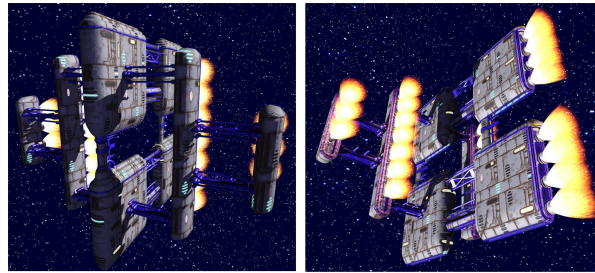
(a) Input Model from two viewpoints
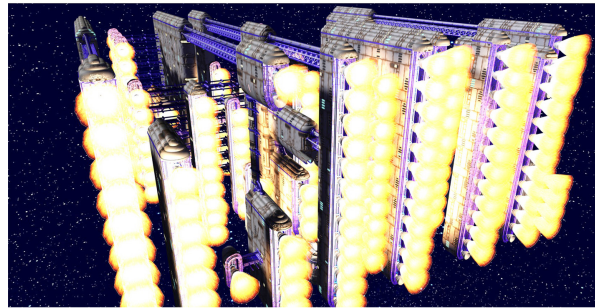
(b) Output without the Connectivity Constraint
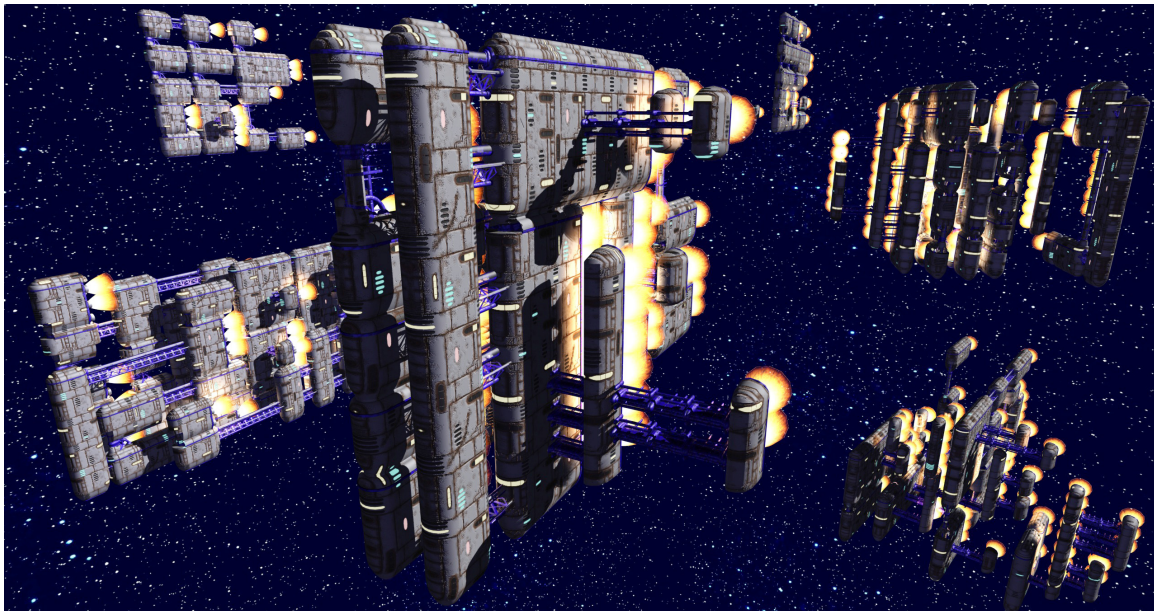
(c) Output with the Connectivity Constraint

Figure 4.37: A fleet of spaceships (b,c) is automatically generated from a simple spaceship model (a). Without the connectivity constraint several dozen small unconnected spaceships are generated (b), but they are all crowded together. With the connectivity constraint, six large spaceships are generated (c). Dimensional constraints are extensively used. They ensure that the rocket engines and other structures do not stretch unnaturally. The shape of the spaceships have a high genus because there are gaps in between the beams and parts of the spaceships.
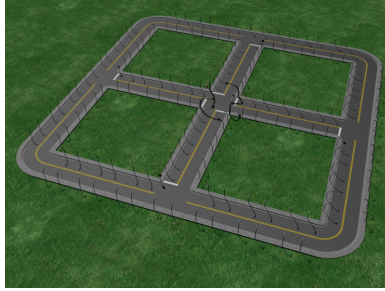
(a) Input
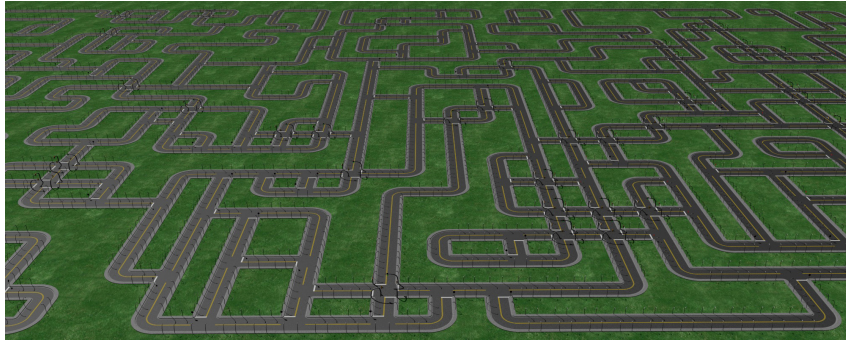
(b) Output

Figure 4.38: A large fully connected road network is generated (b) from a few streets (a) using the connectivity constraint. The dimensions of the roads are also constrained.

axis-aligned triangle in Figure 4.1(a). However, Section 4.5.8 explains that continuous model synthesis has trouble when the input shape contains non-trihedral vertices. It cannot generate non-trihedral vertices without changing the plane spacing. The plane spacing can be altered to accommodate some shapes, but not all shapes. Some shapes may produce an overconstrained set of equations when using Equation 4.25. Several strategies for dealing with this problem are discussed in Section 4.5.8, but each of them has downsides.

Another limitation is that the dimensional constraints in Section 4.6.1 must be in integral units of plane spacings. The plane spacings could be decreased like in Figure 4.2(b) to address this problem, but this may only approximately solve the problem. For example, if the purple objects in Figure 4.2(b) were as high as the blue objects multiplied by $\sqrt{2}$, then both objects could never both be on a grid no matter how tightly the plane
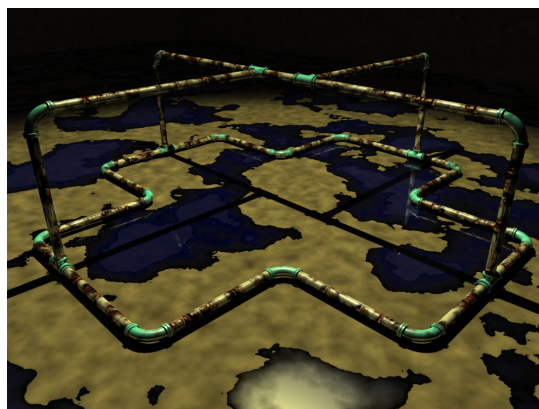
(a) Input Model
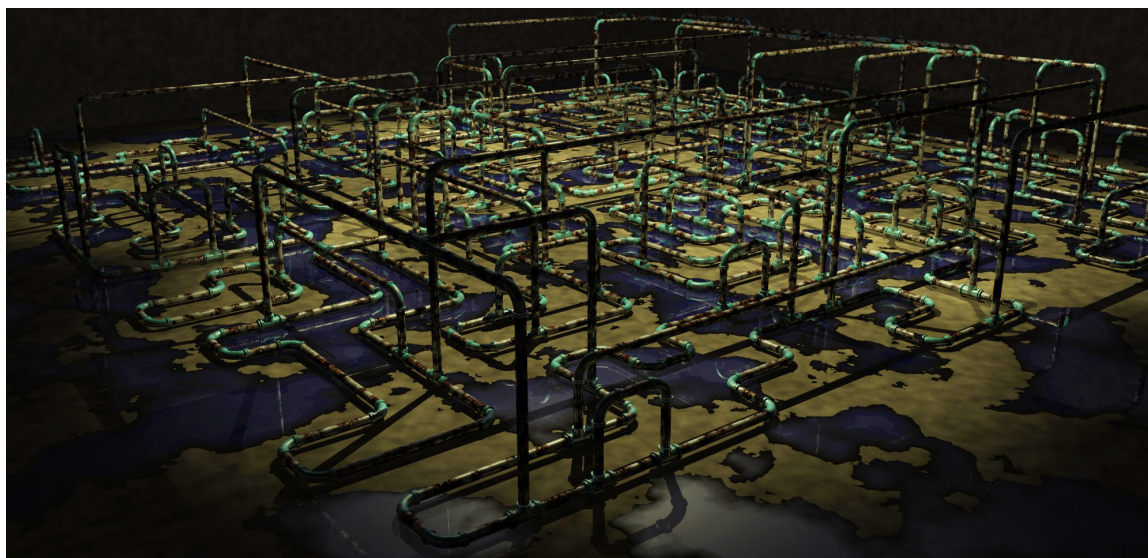
(b) Output Model

Figure 4.39: A complex network of pipes (b) is generated from a simple one (a). Dimensional constraints are used to keep the pipes a certain size.
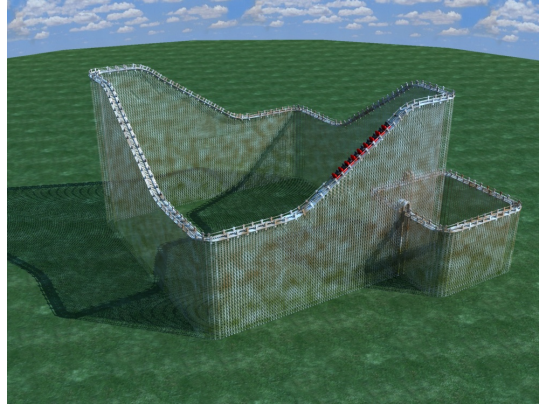
(a) Input Model
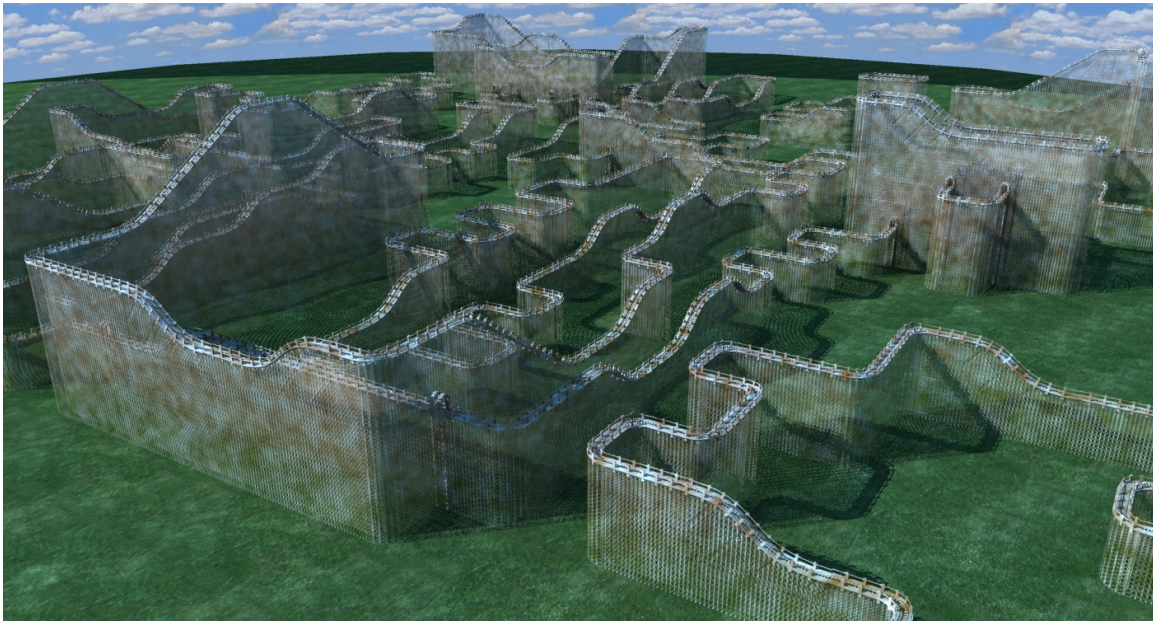
(b) Output Model

Figure 4.40: Several long roller coasters (b) are generated from one simple one (a). Dimensional constraints are used to keep the track a certain width.

	Input Size (polygons)	Output Size (polygons)	Time (minutes)
Skyscrapers	27	9,542	1.8
Terrain	22	922	0.5
Fractals	8	5,743	0.2
Arches	20	1,002	0.5
Houses	39	1,908	1.3
Pentagons	33	2,004	1.1
Oil Platform	60	1,377	0.5
Domes	21	324	0.1
Buildings	116	2,230	1.4
Spaceships	168	4,164	0.6
Roads	126	6,888	0.2
Plumbing	282	7,422	0.8
Roller Coaster	124	1,376	1.8
GPM	365	7,527	3.5

Table 4.1: Complexity of the input and output models and computation time for various results.

were spaced.

Finally, since the objects are produced on sets of parallel planes, the result may contain objects in neat regular patterns of rows and columns. Sometimes this is not very noticeable, but sometimes the result may look too neat and organized. It is possible to perturb the results in a post-processing step, so that all the objects are not so perfectly aligned by applying small translations or rotations to objects within the final mesh.

4.8.2 Limitations in Performance

Section 4.5.7 shows that the time and space complexity of continuous model synthesis is $O(m^3n^3)$, where m is the number of distinct face normals and n is the number of planes in each set of parallel planes. The dependence on n^3 is not particularly troubling since for a given plane spacing n^3 is proportional to the output volume. One might expect the time complexity to depend linearly on the size of the output. The dependence on m^3 is more troubling. Most models contain many distinct normals. Curved shapes are

often approximated using many polygons with many different normals. Because of the $O(m^3)$ dependence, model synthesis is better suited for objects with large flat surfaces such as many found in architecture. As explained in Section 4.6.4, the shapes can often be simplified to some degree using bounding volumes.

Another limitation is that it is difficult to generate objects at different scales. For example, it would be difficult to model a large building while also creating many architectural details. Even though the details could be generated by spacing the parallel planes more closely, the extra planes would consume much more time and memory.

Chapter 5

Comparison

5.1 Model Synthesis and Texture Synthesis

Model synthesis is based upon texture synthesis, but improves upon texture synthesis in several ways that make it more suitable for modeling. First, model synthesis maintains a catalog C_{M_t} which removes labels that would cause problems and second, model synthesis modifies the model in blocks as described in Section 3.3.6. With these improvements, consistent models are generated where all of the pieces fit together seamlessly. Figure 5.1 shows a direct comparison between texture synthesis algorithms and model synthesis. The example model in Figure 5.1(a) is one continuous path with no dead ends. Model synthesis generates creates closed paths without any dead ends as shown in Figure 5.1(d). The results from two existing texture synthesis algorithms [15, 30] are shown for comparison in Figures 5.1(b) and 5.1(c). The texture synthesis techniques do not produce consistent textures. Figure 5.2 shows another comparison between texture synthesis and model synthesis. Again texture synthesis does not produce result satisfying the adjacency constraint which means the shapes suddenly end. The reasons why texture synthesis does not produce consistent models is explained in Section 2.3. Figure 5.1(b) shows that Kwatra et al.'s method is better at reproducing the large-scale structure of the input texture. It should be possible to improve model synthesis by incorporating
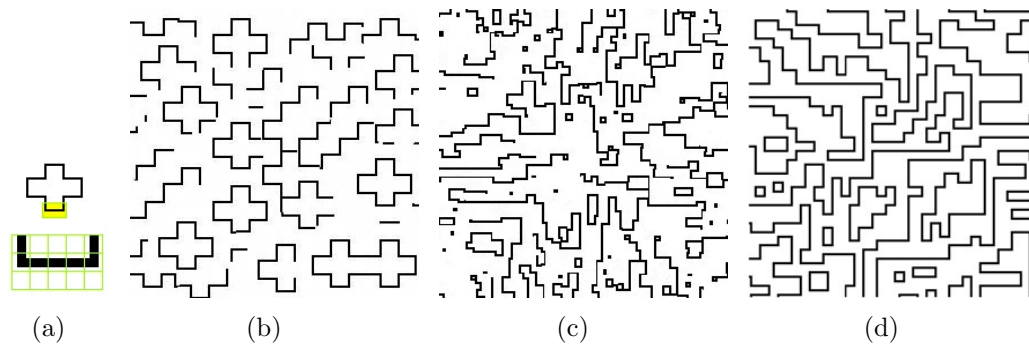
Figure 5.1: (a) Example Texture, (b) Kwatra et al. 2005, (c) Efros and Leung, 1999, (d) 2D Model Synthesis, Part of the example texture is magnified beneath the original to show the 4×4 pixel model pieces.
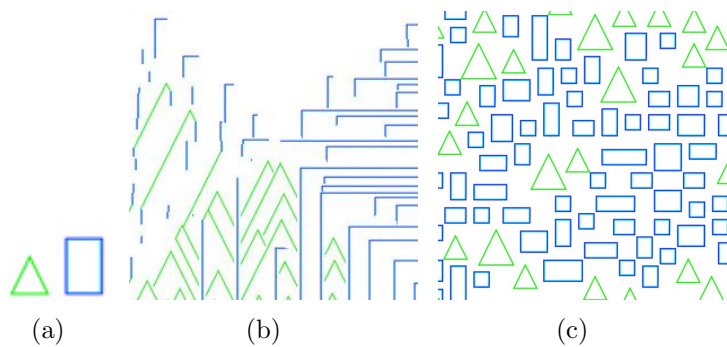
Figure 5.2: (a) Example Texture, (b) Efros and Leung, 1999, (c) 2D Model Synthesis

146

ideas from more sophisticated texture synthesis algorithms.

5.1.1   Comparison to Wang Tiles

The model pieces that discrete model synthesis uses are similar to Wang tiles which have been used for texture synthesis [7]. Wang tiles have colored edges. Two tiles can only be adjacent to each other if the edge they share is colored the same on both tiles. This constraint is very similar to the adjacency constraint. The method of Cohen et al. [7] tiles the plane with Wang tiles to synthesize a texture. Their method successfully tiles the plane in one pass. They essentially succeed in satisfying an adjacency constraint without any failures. This is possible because they explicitly assume that for any tile below and any tile to the left of the location there exists an acceptable tile for that location. This condition is exactly the same as the condition in Statement 3.11 for an infallible model. If Statement 3.11 is satisfied, then Algorithm 3.6 never fails even when $r_x \times r_y = 2 \times 2$. Furthermore, their tiling algorithm performs the same operations as Algorithm 3.6 when $r_x \times r_y = 2 \times 2$. Algorithm 3.6 fails more often the standard model synthesis Algorithm 3.1 as explained in Section 3.3.7, but every model synthesis algorithm succeeds if Statement 3.11 is satisfied. The Wang tile algorithm [7] ensures that Statement 3.11 by constructing the tiles in a particular way. The tiles are constructed so that Statement 3.11 is satisfied by copying different patches of texture and cutting and stitching them back together like a patch based texture synthesis algorithm [14]. However, as explained in Figure 2.8 and Section 2.3, this kind of cutting and stitching works well for textures, but not for models.

5.2   Model Synthesis and Grammars

This section explores how model synthesis compares with grammar-based algorithms used in procedural modeling. Section 5.2.1 describes the advantages and disadvantages

of using the different approaches. In Section 5.2.2, several results and theorems are presented to better explain the relationship between model synthesis and grammar-based algorithms. Specifically, Theorem 5.2.2 shows that there is a context-sensitive grammar that can generate consistent models. Theorem 5.2.4 shows that deciding whether an input string belong to a context sensitive language can be reduced to a problem of deciding if a model is consistent. Section 5.2.3 considers a specific problem involving closed paths and compares how it might be solved using a grammar with how it would be solved using model synthesis.

5.2.1 Comparison of Model Synthesis and Other Approaches

Model synthesis is not fully automatic. The user must perform several tasks. The difficulty of these tasks depends on the type of object that is being modeled. Objects that fit on a grid or contain mostly flat surfaces are relatively easy to generate using model synthesis. These types of objects are often man-made objects and are frequently found in the architectural domain. But other shapes are more difficult to generate using model synthesis including many natural and organic shapes. Organic shapes are difficult to generate with model synthesis because they do not fit on a grid and have many distinct normals. While model synthesis is not useful for generating every type of objects, it offers benefits over other procedural methods for many classes of objects. The user has a relatively simple and straightforward objective: to find or to create an example model and decompose it into model pieces, if discrete model synthesis is used.

In contrast, the user's objective is less simple and straightforward for many existing procedural modeling techniques. Many techniques require the user to construct a grammar. Given the shape of an object the user wants to model, there may not be a straightforward procedure for constructing the rules of a grammar that could generate a similar shape. Grammars are constructed through some human ingenuity and through trail and error. The grammars themselves can be complicated, even when they describe

simple shapes. A few examples of complicated grammars are shown in Figure 5.3. It is difficult to understand what kind of shape the grammars in Figures 5.3(a) and 5.3(b) would produce just by looking at them.

```
PRIORITY 1:
1:   footprint ⤳ S(1r,building_height,1r) facades
       T(0,building_height,0) Roof("hipped",roof_angle){ roof }
PRIORITY 2:
2:   facades ⤳ Comp("sidefaces"){ facade }
3:   facade : Shape.visible("street")
       ⤳ Subdiv("X",1r,door_width*1.5){ tiles | entrance } : 0.5
       ⤳ Subdiv("X",door_width*1.5,1r){ entrance | tiles } : 0.5
4:   facade ⤳ tiles
5:   tiles ⤳ Repeat("X",window_spacing){ tile }
6:   tile ⤳ Subdiv("X",1r,window_width,1r){ wall |
       Subdiv("Y",2r,window_height,1r){ wall | window | wall } | wall }
7:   window : Scope.occ("noparent") != "none" ⤳ wall
8:   window ⤳ S(1r,1r,window_depth) I("win.obj")
9:   entrance ⤳ Subdiv("X",1r,door_width,1r){ wall |
       Subdiv("Y",door_height,1r){ door | wall } | wall }
10:  door ⤳ S(1r,1r,door_depth) I("door.obj")
11:  wall ⤳ I("wall.obj")
PRIORITY 3:
12:  roof ⤳ Comp("sidefaces"){ covering }
       Comp("sideedges"){ roofedge } Comp("topedges"){ roofedge }
13:  covering ⤳
       Repeat("XY",flatbrick_width,brick_length){ flatbrick }
       Subdiv("X",flatbrick_width,1r){ ε |
         Repeat("X",flatbrick_width){ roofedge } }
14:  roofedge ⤳
       Subdiv("Y",overlap,brick_length-2*overlap,1r){ ε |
         roundbrick | Repeat("Y",brick_length-overlap){ roundbrick } }
15:  flatbrick ⤳ S(1r,1r,flatbrick_height) T(0,0,-flatbrick_height)
       Rx(-3) I("flatbrick.obj")
16:  roundbrick ⤳ S(roundbrick_w,Scope.sy+overlap,roundbrick_h)
       T(-roundbrick_w/2,-overlap,-roundbrick_h)
       Rx(-3) I("roundbrick.obj")
```
(a) A split grammar

```
#define φ 137.5    /* divergence angle */
#define α₀ 5       /* direction change - no branching */
#define α₁ 20      /* branching angle - main axis */
#define α₂ 32      /* branching angle - lateral axis */
#define W 0.02     /* initial branch width */
#define VD 0.95    /* apex vigor decrement */
#define Del 30     /* delay */
#define LS 5       /* how long a leaf stays */
#define LP 8       /* full photosynthate production */
#define LM 2       /* leaf maintenance */
#define PB 0.8     /* photosynthates needed for branching */
#define PG 0.4     /* photosynthates needed for growth */
#define BM 0.32    /* branch maintenance coefficient */
#define BE 1.5     /* branch maintenance exponent */
#define N_min 25   /* threshold for shedding */
Consider: ?E[]!L   /* for context matching */
ω:   !(W,1,1)F(2)L(1,LP,0,0)A(1,0)[!(0,0,0)]!(W,0,1)

p₁:  A(vig,del) : del<Del → A(vig,del+1)
p₂:  L(vig,p,age,del) : (age<LS)&&(del<Del-1) → L(vig,p,age,del+1)
p₃:  L(vig,p,age,del) : (age<LS)&&(del==Del-1)
         → L(vig,p,age,del+1)?E(vig*0.5)
p₄:  L(vig,p,age,del) > ?E(r) : (age<LS) && (r*LP>=LM)
         && (del == Del) → L(vig,LP*r−LM,age+1,0)
p₅:  L(vig,p,age,del) > ?E(r) : ((age == LS)||(r*LP<=LM))
         && (del == Del) → L(0,0,LS,0)
p₆:  ?E(r) < A(vig,del) : r*LP−LM>PB {vig=vig*VD;}
         → /(φ)[+(α₂)!(W,−PB,1)F(vig)L(vig,LP,0,0)A(vig,0)
           [!(0,0,0)]!(W,0,1)]
         −(α₁)!(W,0,1)F(vig)L(vig,LP,0,0)/A(vig,0)
p₇:  ?E(r) < A(vig,del) : r*LP−LM > PG {vig=vig*VD;}
         → /(φ)−(α₀)[!(0,0,0)]
           !(W,−PG,1)F(vig)L(vig,LP,0,0)A(vig,0)
p₈:  ?E(r) < A(vig,del) : r*LP−LM <= PG → A(vig,0)
p₉:  ?E(r) → ε
p₁₀: !(w₀,p₀,n₀) > L(vig,p_L,age,del) [!(w₁,p₁,n₁)]!(w₂,p₂,n₂) :
         {w=(w₁^2+w₂^2)^0.5; p=p₁+p₂+p_L−BM*(w/W)^BE;}
         (p>0) || (n₁+n₂ >=N_min) → !(w,p,n₁+n₂)
p₁₁: !(w₀,p₀,n₀) > L(vig,p_L,age,del) [!(w₁,p₁,n₁)]!(w₂,p₂,n₂)
         → !(w₀,0,0)L%
```
(b) An L-system

(c) A building generated from the above split grammar
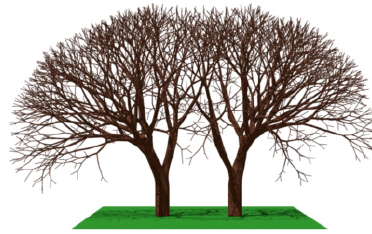
(d) A tree generated from the above L-system

Figure 5.3: Examples of grammars used in modeling. The building grammar (a,c) is from [Müller et al., 2006]. The tree grammar (b,d) is from [Měch and Prusinkiewicz, 1996]. It takes some effort to be able to understand and modify the grammar.

Model synthesis is easier to understand from a user's perspective. The user does not need to know anything about grammars or the inner mechanics of the algorithm itself in order to use it. Instead, the user only deals with the input and output models. Once

a suitable example model has been created it is easy to modify it as needed. Its parts can easily be rearranged using standard 3D modeling programs.

Most procedural modeling techniques are aimed at modeling specific classes of objects such as urban buildings [40], truss structures [58], fractals, and landscapes [42]. But many interesting structures lie outside of these classes of objects including spaceships, castles, oil platforms, plumbing, and roller coasters, just to name a few. Since model synthesis is a general technique, it is especially useful for modeling objects that cannot be generated easily using other techniques.

There is a close connection between model synthesis and context-sensitive grammars which is demonstrated by Theorems 5.2.2 and 5.2.4. These theorems show that both methods can be used to accomplish the same goal. The set of *consistent models* can be represented by a grammar, but it is different in several ways from grammars that are typically used in procedural modeling. Model synthesis uses model pieces which are volumetric blocks that keep track of the space in which they are embedded in. The location of empty space is recorded in model synthesis. Empty space is not explicitly recorded in most grammar-based techniques. It is determined by checking that all of the objects are absent. Model synthesis is also good at avoiding self-intersections which is part of the adjacency constraint. The grammars found in other techniques may need to be carefully constructed so that self-intersections do not occur. Another task that model synthesis is particularly good at is in creating closed paths. A way to create a closed paths using a grammar is discussed in Section 5.2.3, but it is non-trivial.

Model synthesis is good at creating geometric detail at a particular scale, but not at multiple scales. For example, it is difficult for model synthesis to create geometric detail at the scale of a building and at the scale of the building's window or door knob simultaneously. Other procedural methods [40, 39, 42] create geometric detail at multiple scales more easily.

5.2.2 Solving Equivalent Problems with Model Synthesis and Grammars

This section discusses how to use grammars to decide if a model is consistent. A model is consistent if it satisfies the adjacency constraint in Equation 3.3 defines a consistent model.) Since grammars operate on one-dimensional strings, we first discuss a method for converting a 3D model into a 1D string that a grammar could generate. Every label in the model is written down in row-column-vertical order. Labels in the bottom $z = 1$ positions are transcribed first, followed by the labels in the $z = 2$ position, etc. A special character '|' is inserted at the end of each row and another special character '||' is inserted when the height z changes. An example is shown in Figure 5.4.
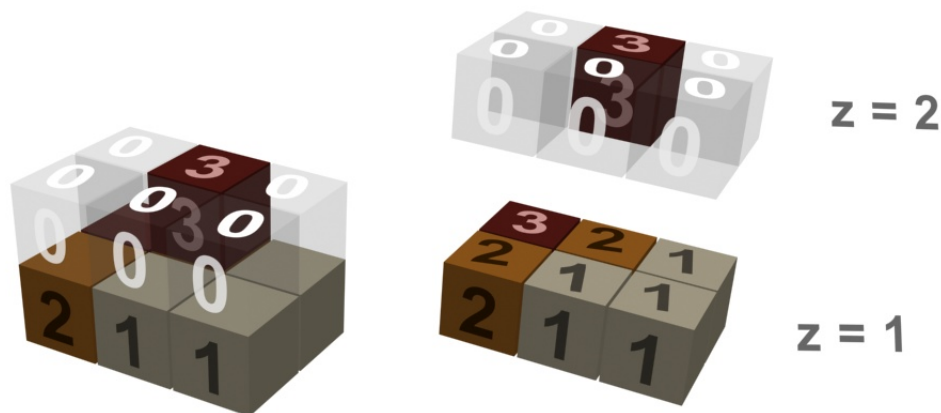
Figure 5.4: This 3D model is converted into the 1D string: 2 1 1 | 3 2 1 || 0 0 0 | 0 3 0.

Definition If the language L is the set of all models, in string form, consistent with the model E, then a grammar *generates E consistent models* if it generates L.

Theorem 5.2.1. For some input models E, there is no grammar generating E models that is context-free.

Proof. This can be shown by a counter example. The language $\{a^i b^i c^i | i \geq 1\}$ is the canonical example of a non-context-free language. A grammar is not context free if it

151

generates strings from this language. The model shown in Figure 5.5 is constructed, so that any model that is consistent with it will contain a string of labels of the form, $a^i b^i c^i$. When this model is used as the input model E, then there is no context-free grammar that can generate E-consistent models. □
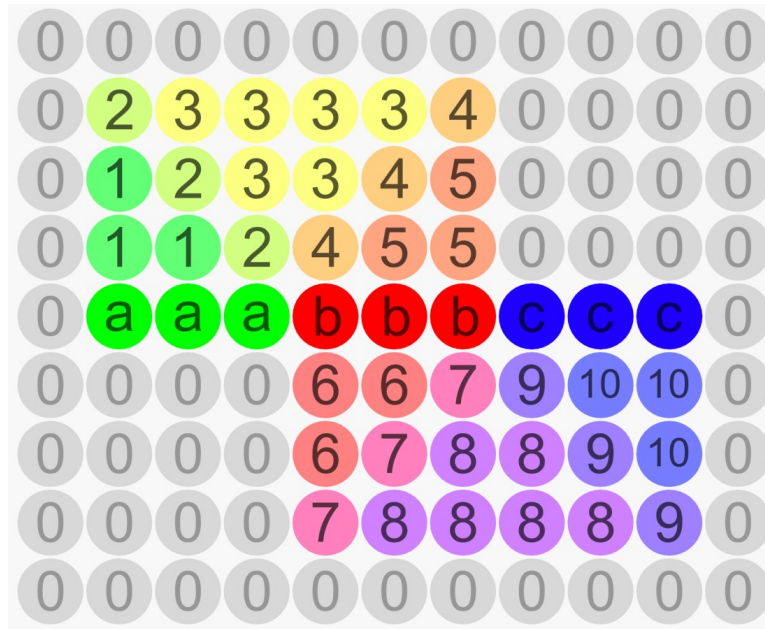
Figure 5.5: No context-free grammar can generate models consistent with this model.

Theorem 5.2.2. For all input models E, there is a context-sensitive grammar that generates E-consistent models.

Proof. Let L be the set of all models consistent with E. The grammar generating L is context-sensitive, if there is a linear-bounded automaton A that accepts L. A linear-bounded automaton (LBA) is a 5-tuple $A = (Q, \Sigma, \Gamma, q_0, \delta)$ where Q is a finite set of states, Σ and Γ are the input and tape alphabets, $q_0 \in Q$ is the initial state, and δ is a function determining the automaton's next move. A LBA is the same as a nondeterministic Turing machine except two extra tape symbols '<' and '>' mark the ends of the tape and the tape head may not travel past these two symbols.

A linear-bounded automaton A can be constructed that accepts L. The language L consists of models that satisfy the adjacency constraint (Equation 3.1) in the x, y, and z directions. The adjacency constraint is verified over a series of passes by A. It is verified in the x direction in the first pass and then in the y and z directions in subsequent passes. The automaton A begins with an initial state of $q_0 = q^x$ to indicate that it is currently verifying the x direction.

$$\text{Tape:} \quad < \; 2 \;\; 1 \;\; 1 \;\; | \;\; 3 \;\; 2 \;\; 1 \;\; || \;\; 0 \;\; 0 \;\; 0 \;\; | \;\; 0 \;\; 3 \;\; 0 \;\; >$$
$$\text{State:} \qquad q^x$$

The tape head travels from left to right. As it does, the state of A records the previous symbol that was visited. In the next time step, it acquires the state q_2^x.

$$< \; 2 \;\; 1 \;\; 1 \;\; | \;\; 3 \;\; 2 \;\; 1 \;\; || \;\; 0 \;\; 0 \;\; 0 \;\; | \;\; 0 \;\; 3 \;\; 0 \;\; >$$
$$q_2^x$$

$$< \; 2 \;\; 1 \;\; 1 \;\; | \;\; 3 \;\; 2 \;\; 1 \;\; || \;\; 0 \;\; 0 \;\; 0 \;\; | \;\; 0 \;\; 3 \;\; 0 \;\; >$$
$$q_1^x \qquad\qquad \vdots$$

The previous symbol on the tape corresponds in the 3D model to the label is in the $-x$ direction of the current label. These two labels are what is needed to verify the adjacency constraint in the x direction. If this constraint is ever violated, the model is rejected. Otherwise, the automaton continues to check other parts of the model. When the tape head reaches the end of the tape, the adjacency constraint has been completely verified in the x direction. Then the tape head moves back to the beginning and A requires the state q^y to indicate that it is verifying the adjacency constraint in the y direction.

The adjacency constraint is more difficult to verify in the y direction since adjacent labels in the 3D model are not adjacent to one another in the 1D string. The y-direction is checked in a series of passes. In the first pass, only the first symbol in each row is checked. In order to record which symbols have been checked, the automaton uses an

additional set of symbols $(\cdot, y)$. For example, the symbol $(2, y)$ is used to indicate that this position has been checked in the y direction and that the position was originally marked 2.

$$< \quad \underset{q^y}{2} \quad 1 \quad 1 \quad | \quad 3 \quad 2 \quad 1 \quad || \quad 0 \quad 0 \quad 0 \quad | \quad 0 \quad 3 \quad 0 \quad >$$
$$\vdots$$

$$< \quad (2,\text{y}) \quad 1 \quad 1 \quad | \quad \underset{q_2^y}{3} \quad 2 \quad 1 \quad || \quad 0 \quad 0 \quad 0 \quad | \quad 0 \quad 3 \quad 0 \quad >$$
$$\vdots$$

$$< \quad (2,\text{y}) \quad 1 \quad 1 \quad | \quad (3,\text{y}) \quad 2 \quad 1 \quad || \quad (0,\text{y}) \quad 0 \quad 0 \quad | \quad \underset{q_0^y}{0} \quad 3 \quad 0 \quad >$$

In the first pass, the automaton verifies that the adjacency constraint is satisfied in the y direction for the first symbol in each row. The second symbol in each row is checked next. These symbols are located by finding the first symbol b that has not been change to (b, y).

$$< \quad (2,\text{y}) \quad \underset{q^y}{1} \quad 1 \quad | \quad (3,\text{y}) \quad 2 \quad 1 \quad || \quad (0,\text{y}) \quad 0 \quad 0 \quad | \quad (0,\text{y}) \quad 3 \quad 0 \quad >$$
$$\vdots$$

$$< \quad (2,\text{y}) \quad (1,\text{y}) \quad 1 \quad | \quad \underset{q_1^y}{(3,\text{y})} \quad 2 \quad 1 \quad || \quad (0,\text{y}) \quad 0 \quad 0 \quad | \quad (0,\text{y}) \quad 3 \quad 0 \quad >$$
$$\vdots$$

The symbols that are third in their row are checked next and so on until the adjacency constraint has been checked along every column of the model in y direction. A similar method is used to check the model in the z direction. Another set of symbols $(\cdot, z)$, can be used to mark which positions have been checked in the z direction. If no violations of the adjacency constraint have been found in the x,y, and z direction, then the automaton accepts the model. $\qquad\square$

Theorems 5.2.1 and 5.2.2 apply to two, three, and higher dimensional versions of model synthesis, but one-dimensional model synthesis is an exception. A one-dimensional model is simply a string. For a 1D example model E, Theorem 5.2.1 is invalid since nothing like Figure 5.5 can be constructed with only one dimension. Although Theorem 5.2.2 is still true for 1D model synthesis, we can make a stronger statement involving a regular grammar.

Theorem 5.2.3. If E is a one-dimensional string, there is a regular grammar that generates E consistent models.

Proof. A regular grammar generating E-consistent models is constructed as follows. For each label, b of the model, the grammar contains b as a terminal symbol of the grammar and contains a variable called A_b. The rules of the grammar are constructed so that only one variable is present in the string at any given time. Each variable A_b, transforms into the label b at the next time step. The variable A_b could be replaced according to the rule $A_b \rightarrow b$ which would terminate the string or the string could continue according to the adjacency constraint. The label c can follow the label b iff $T[b, c] = 1$. So iff $T_x[b, c] = 1$, then $A_b \rightarrow bA_c$ is a rule of the grammar. This ensures that all strings that are produced satisfy Equation 3.3. $\square$

Theorem 5.2.4. The problem of deciding if a string is part of the context sensitive language L can be reduced to a problem of deciding if a model containing that string is consistent.

Proof. For every context-sensitive language L, there is a linear-bounded automaton that accepts L. It will be shown that all of the actions of a linear-bounded automaton $A = (Q, \Sigma, \Gamma, q_0, \delta)$ accepting a string can be described within a consistent model.

The transition matrices determine which labels can be next to one another in a consistent model. These matrices can be constructed so that the model is only consistent if each of its rows records the state of the A, the symbols recorded on the tape, and

155

location of the tape head. For example, suppose the problem is to determine if the string 'aabbcc' is in the language $\{a^i b^i c^i | i \geq 1\}$. The tape would initially contain this input string along with two symbols '<' and '>' to mark the start and end of the tape. The width of the model M_t is equal to the width of the tape. The first row of the model would contain the labels

$$\text{Row 1:} \quad < \quad (a, q_0) \quad a \quad b \quad b \quad c \quad c \quad >$$

The label (a, q_0) is used to indicate that the automaton A is in its initial state q_0 and the tape head is reading the 'a' symbol. For every state $q \in Q$ and every symbol in the tape alphabet $s \in \Gamma$, there is a label $(q, s) \in K$ where K is the set of labels. Suppose that when A is in state q_0 and is reading symbol 'a' that it responds by printing the symbol 'd' onto the tape, switching to state q_1, and remaining stationary. Then the model's next row would be

$$\text{Row 2:} \quad < \quad (d, q_1) \quad a \quad b \quad b \quad c \quad c \quad >$$

A transition matrix T_y can be constructed that would guarantee that this row would appear beneath row 1. Each symbol may repeat vertically $T_y[a, a] = T_y[b, b] = T_y[c, c] = T_y[<, <] = T_y[>, >] = 1$. The label (a, q_0) must be above (d, q_1), so $T_y[(a, q_0), s] = 1 \Leftrightarrow s = (d, q_1)$. The transition matrix is constructed to only allow one possible option at every location which is exactly the option that the automaton would choose. The tape head also could move to the left or to the right. Suppose that when the automaton is in state q_1 and is reading the symbol d, it moves to the right and switches to state q_2. This move occurs over two rows of the model

$$\text{Row 3:} \quad < \quad (d, q_2^r) \quad (a, q_2^{r'}) \quad b \quad b \quad c \quad c \quad >$$
$$\text{Row 4:} \quad < \quad d \quad (a, q_2) \quad b \quad b \quad c \quad c \quad >$$

The transition matrices are constructed so that only these labels may appear in rows 3 and 4.

156

$$T_y[(d, q_2^r), s] = 1 \Leftrightarrow s = d \tag{5.1}$$

$$T_y[(a, q_2^{r'}), s] = 1 \Leftrightarrow s = (a, q_2) \tag{5.2}$$

$$T_x[(d, q_2^r), s] = 1 \Leftrightarrow \exists t | s = (t, q_2^{r'}) \tag{5.3}$$

$$T_y[a, s] = 1 \Leftrightarrow s = a \text{ or } \exists q \in Q | s = (a, q^{r'}) \text{ or } \exists q \in Q | s = (a, q^{l'}). \tag{5.4}$$

For any linear bounded automaton, transition matrices can be constructed that will guarantee that a unique consistent model exists which reproduces the actions of the automaton. If the automaton reaches the accepting halt state h_a, then it can simply remain at that state for all of the remaining rows. On the other hand, if the automaton reaches the rejecting halt state h_r, then the model becomes inconsistent. The transition matrix is created such that h_r has nothing beneath it $\forall s, T_y[h_r, s] = 0$.

The problem of deciding if a string is part of the context-sensitive language L can be decided by reducing it to a consistency problem in several steps. First, the incomplete model M is created by placing the input and the initial state q_0 in Row 1 of M and leaving the rest of the model empty. The transition matrix is created so that it reproduces that actions of the automaton accepting L.

□

Remark: A more general question to consider is if deciding if a string is in a recursively enumerable language can be reduced to a consistency problem. Recursively enumerable languages are accepted by Turing machines. Turing machines allow the tape to be infinitely long. Extending the proof to reproduce a Turing machine, would require an infinitely wide model and Algorithm 3.2 would never finish computing.

In the proof, the constructed model is two-dimensional. The proof extends to higher-dimensional models, but not to one-dimensional models.

5.2.3 Generating Closed Paths with Grammars

To better understand how model synthesis differs from typical grammars used in modeling, it is instructive to consider a specific example such as the cross-shaped path shown in Figure 5.6. Earlier, this example was shown to cause difficulty in some texture synthesis algorithms (Figure 2.10). The path forms a closed loop and all of its edges are at right angles. Similar closed paths having only right angles can be generated by a grammar. Right-angled paths could be created using turtle graphics with only four basic commands move up u, move down d, move left l, and move right r. These four commands are the terminal symbols of the grammars. The rest of the symbols of the grammar are non-terminals.
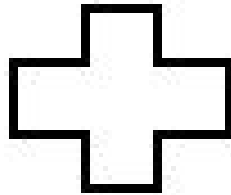
Figure 5.6: An example of a closed path.

The grammar is designed to first allow a random walk using right-angled paths and then end by closing the loop. No matter where the random walk ends, the loop can be closed if the position of the start relative to the end is recorded.

The current direction of the turtle is given by the variables U' (heading up), D' (heading down), L' (heading left), and R' (heading right). Initially, we assume the turtle is headed up U'. The turtle has three options. It could continue going up or it could turn left or turn right. These actions are accomplished using the following rules:

$$U' \rightarrow uU'D \qquad \text{Continue going up}$$

$$U' \rightarrow uL'D \qquad \text{Turn left}$$

$$U' \rightarrow uR'D \qquad \text{Turn right}$$

All three rules include a move up u command at the front and a D variable at the end. The unprimed variables at the end are used to close the loop. The variables at the end are always in the opposite direction of the terminals at the front. This is later used to close the loop, since every time the turtle goes up it must eventually come down an equal distance. When traveling left, right, or down, there are three options available for each:

$$
\begin{array}{lll}
L' \rightarrow lL'R & R' \rightarrow rR'L & D' \rightarrow dD'U \\
L' \rightarrow lU'R & R' \rightarrow rU'L & D' \rightarrow dL'U \\
L' \rightarrow lD'R & R' \rightarrow rD'L & D' \rightarrow dR'U
\end{array}
$$

An example of several rules chosen at random is shown in Figure 5.7. Notice that the strings never contain more than one primed variable, the primed variable is always centered in the string, and there are always the same number of u symbols as D symbols. Two more production rules are used to end the random walk while traveling either left or right:

$$L' \rightarrow lR$$

$$R' \rightarrow rL$$

159
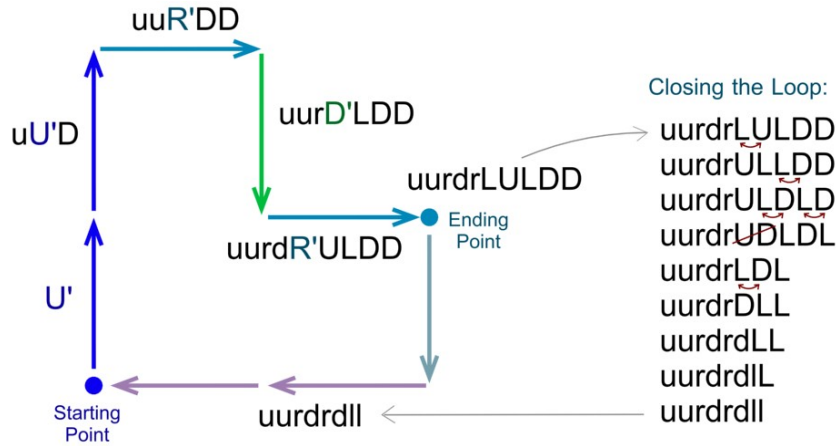
Figure 5.7: An example of a closed path generated by a grammar.

In order to close the loop, the turtle next moves up or down back to the original vertical position, then left or right back to the original horizontal position. The variables at the end of the string are sorted so that all of the U and all of the D variables come in front of all of the L and R variables. This is done through several production rules which swap a U or D variable in front of a L or R variable:

$$LD \to DL \qquad LU \to UL \qquad RD \to DR \qquad RU \to UR$$

Also, the up U and the down D moves cancel each other out as well as the L and R moves:

$$UD \to \lambda \qquad DU \to \lambda \qquad LR \to \lambda \qquad RL \to \lambda$$

where λ is an empty string.

Finally, several rules are used to convert the variables $U, D, L,$ and R into the terminals u, d, l and r to close the loop.

160

$$\{u, l, r\}U \rightarrow \{u, l, r\}u$$

$$\{d, l, r\}D \rightarrow \{d, l, r\}d$$

$$\{u, d, l\}L \rightarrow \{u, d, l\}l$$

$$\{u, d, r\}R \rightarrow \{u, d, r\}r$$

The grammar could generate paths that self-intersect like the path in Figure 5.8.
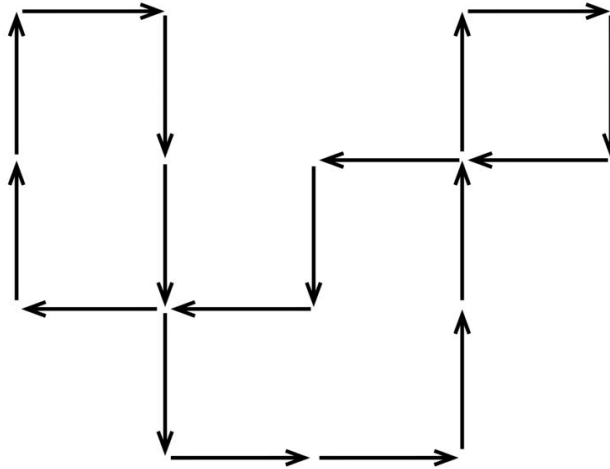
Figure 5.8: A self-intersecting closed path.

There does not appear to be a straightforward way to modify the grammar to generate paths that never intersect. Self-intersections occur in many kinds of grammar-based modeling techniques. Model synthesis is particularly well-suited for handling self-intersections. Self-intersections are eliminated by the adjacency constraint.

This example demonstrates that designing a grammar to generate a specific type of shape like a closed path can consume significant time and energy. These types of paths can be created relatively easily using model synthesis by dividing Figure 5.6 into a few model pieces.

5.3　Parallel Polygons

The continuous adjacency constraint (Equation 4.1) is related to some ideas from computational geometry which involve the concept of parallel polygons. Two polygons are defined to be parallel if they have the same number of sides and if their sides can be put into a one to one correspondence so that corresponding sides are parallel [20]. Any two parallel polygons are consistent with one another according to Equation 4.1.

Chapter 6

Conclusion

Model synthesis is the first example-based procedural modeling technique. Because it uses examples, it applies more generally to a much wider variety of objects than other procedural modeling techniques. Model synthesis is inspired by texture synthesis, but improves upon it in several ways. First, it uses a catalog of possible labels to identify and avoid labels that would cause the objects to self-intersect or to suddenly end or to produce other problems. potential problems early on. Second, it modifies the model in parts. With these two improvements, model synthesis can create large models that satisfy an adjacency constraint. Models that satisfy the constraint are seamless. All of their parts fit together as neatly and smoothly as they do in the original models. On the other hand, texture synthesis algorithms often do not produce seamless models.

Discrete model synthesis can be hard to use since it requires the model to be decomposed into model pieces that fit on a grid. Continuous model synthesis is introduced to accept more general input models. Several different approaches for continuous model synthesis are described, but some are too difficult to implement. The easiest approach to implement assumes that the faces of the output are all on sets of parallel lines. In addition to the adjacency constraint, several more constraints are introduced to give the user more control over the result.

6.1 Future Work

Several exciting possibilities for future research come to mind. It is likely that with future research, many of the limitations of continuous model synthesis can be overcome. One of the most important problems is that the time and memory requirements increase greatly with the number of distinct normals in the input model and also when trying to create large-scale structures with small-scale detail. It is likely that the performance can be improved since model synthesis creates a lot of vertices and edges that are not used in the final output model. They might end up on a face or in empty space. The difficulty is that when the algorithm starts, it is not clear which vertices and edges are needed and which are not. It may be possible to start with fewer vertices and edges, but add them in later as needed. There also may be a way to approximate curved objects with a few flat surfaces that work with model synthesis. Flat objects only need to be bent slightly to create curved object. Finding a way to use small deformations within texture synthesis or model synthesis could potentially be very valuable.

Other limitations discussed in Section 4.8 involve handling non-trihedral vertices and dimensional constraints that are not in integer multiples of the plane spacings. One solution is to find a way to implement or approximately implement the approach in Section 4.4 that uses Minkowski sums. There also may be a way to slightly alter the face normals, so that non-trihedral vertices fit on a grid or to create the meshes with trihedral vertices and then fuse the trihedral vertices together to create non-trihedral vertices.

The adjacency constraint assumes that every neighborhood of M is a translated copy of some neighborhood of E, but another interesting problem would be to allow rotated copies or scaled copies or even copies with an affine transformation.

Another interesting topic for future research is to find a way to more easily identify infallible models and to find ways of converting from fallible models to infallible models as described in Section 3.3.7.

Bibliography

[1] Daniel G. Aliaga, Paul A. Rosen, and Daniel R. Bekins. Style grammars for interactive visualization of architecture. *IEEE Transactions on Visualization and Computer Graphics*, 13(4):786–797, 2007.

[2] Daniel G. Aliaga, Carlos A. Vanegas, and Bedřich Beneš. Interactive example-based urban layout synthesis. *ACM Trans. Graph.*, 27(5):1–10, 2008. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/1409060.1409113.

[3] Connelly Barnes, Eli Shechtman, Adam Finkelstein, and Dan B Goldman. PatchMatch: A randomized correspondence algorithm for structural image editing. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 28(3), August 2009.

[4] Pravin Bhat, Stephen Ingram, and Greg Turk. Geometric texture synthesis by example. In *SGP '04: Symposium on Geometry processing*, pages 41–44, New York, NY, USA, 2004. ACM. ISBN 3-905673-13-4. doi: http://doi.acm.org/10.1145/1057432.1057437.

[5] Guoning Chen, Gregory Esch, Peter Wonka, Pascal Müller, and Eugene Zhang. Interactive procedural street modeling. *ACM Trans. Graph.*, 27(3), 2008.

[6] Xuejin Chen, Boris Neubert, Ying-Qing Xu, Oliver Deussen, and Sing Bing Kang. Sketch-based tree modeling using markov random field. *ACM Trans. Graph.*, 27(5): 1–9, 2008. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/1409060.1409062.

[7] Michael F. Cohen, Jonathan Shade, Stefan Hiller, and Oliver Deussen. Wang tiles for image and texture generation. *ACM Trans. Graph.*, 22(3):287–294, 2003. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/882262.882265.

[8] Michael F. Cohen, Jonathan Shade, Stefan Hiller, and Oliver Deussen. Wang tiles for image and texture generation. In *SIGGRAPH '03*, pages 287–294, New York, NY, USA, 2003. ACM. ISBN 1-58113-709-5. doi: http://doi.acm.org/10.1145/1201775.882265.

[9] A. Criminisi, P. Prez, and K. Toyama. Region filling and object removal by exemplar-based inpainting. *IEEE Trans. Image Processing*, 13(9):1200–1212, 2004.

[10] Karel Culik, II. An aperiodic set of 13 wang tiles. *Discrete Math.*, 160(1-3):245–251, 1996. ISSN 0012-365X. doi: http://dx.doi.org/10.1016/S0012-365X(96)00118-5.

[11] Barbara Cutler, Julie Dorsey, Leonard McMillan, Matthias Müller, and Robert Jagnow. A procedural approach to authoring solid models. *ACM Trans. Graph.*, 21(3):302–311, 2002. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/566654.566581.

[12] Gianfranco Doretto, Alessandro Chiuso, Ying Nian Wu, and Stefano Soatto. Dynamic textures. *Int. J. Comput. Vision*, 51(2):91–109, 2003. ISSN 0920-5691. doi: http://dx.doi.org/10.1023/A:1021669406132.

[13] Iddo Drori, Daniel Cohen-Or, and Hezy Yeshurun. Fragment-based image completion. *ACM Trans. Graph.*, 22(3):303–312, 2003. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/882262.882267.

[14] Alexei A. Efros and William T. Freeman. Image quilting for texture synthesis and transfer. In *SIGGRAPH '01*, pages 341–346, New York, NY, USA, 2001. ACM. ISBN 1-58113-374-X. doi: http://doi.acm.org/10.1145/383259.383296.

[15] Alexei A. Efros and Thomas K. Leung. Texture synthesis by non-parametric sampling. In *IEEE International Conference on Computer Vision*, pages 1033–1038, Corfu, Greece, September 1999.

[16] U Flemming. More than the sum of parts: the grammar of queen anne houses. *Environment and Planning B: Planning and Design*, 14(3):323–350, May 1987.

[17] Alain Fournier, Don Fussell, and Loren Carpenter. Computer rendering of stochastic models. *Commun. ACM*, 25(6):371–384, 1982. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/358523.358553.

[18] William T. Freeman, Thouis R. Jones, and Egon C Pasztor. Example-based super-resolution. *IEEE Comput. Graph. Appl.*, 22(2):56–65, 2002. ISSN 0272-1716. doi: http://dx.doi.org/10.1109/38.988747.

[19] Thomas Funkhouser, Michael Kazhdan, Philip Shilane, Patrick Min, William Kiefer, Ayellet Tal, Szymon Rusinkiewicz, and David Dobkin. Modeling by example. *SIGGRAPH '04*, 2004.

[20] Leonidas Guibas and John Hershberger. Morphing simple polygons. In *SCG '94: Proceedings of the tenth annual symposium on Computational geometry*, pages 267–276, New York, NY, USA, 1994. ACM. ISBN 0-89791-648-4. doi: http://doi.acm.org/10.1145/177424.177987.

[21] Charles Han, Eric Risser, Ravi Ramamoorthi, and Eitan Grinspun. Multiscale texture synthesis. In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, pages 1–8, New York, NY, USA, 2008. ACM. doi: http://doi.acm.org/10.1145/1399504.1360650.

[22] James Hays and Alexei A Efros. Scene completion using millions of photographs. *ACM Transactions on Graphics (SIGGRAPH 2007)*, 26(3), 2007.

[23] David J. Heeger and James R. Bergen. Pyramid-based texture analysis/synthesis. In *SIGGRAPH '95*, pages 229–238, New York, NY, USA, 1995. ACM. ISBN 0-89791-701-4. doi: http://doi.acm.org/10.1145/218380.218446.

[24] Aaron Hertzmann, Charles E. Jacobs, Nuria Oliver, Brian Curless, and David H. Salesin. Image analogies. In *SIGGRAPH '01*, pages 327–340, New York, NY, USA, 2001. ACM. ISBN 1-58113-374-X. doi: http://doi.acm.org/10.1145/383259. 383295.

[25] Aaron Hertzmann, Nuria Oliver, Brian Curless, and Steven M. Seitz. Curve analogies. In *EGRW '02: Proceedings of the 13th Eurographics workshop on Rendering*, pages 233–246, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association. ISBN 1581135343. doi: http://dx.doi.org/10.1145/364338.364405. URL `http://dx.doi.org/10.1145/364338.364405`.

[26] Takashi Ijiri, Radomr Mech, Takeo Igarashi, and Gavin Miller. An example-based procedural system for element arrangement. *Comput. Graph. Forum*, 27:429–436, 2008.

[27] Jonathan Knight. Negative results: Null and void. *Nature*, 422:554–555, April 2003.

[28] Johannes Kopf, Chi-Wing Fu, Daniel Cohen-Or, Oliver Deussen, Dani Lischinski, and Tien-Tsin Wong. Solid texture synthesis from 2d exemplars. *ACM Trans. Graph.*, 26(3):2, 2007. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/1276377. 1276380.

[29] Vivek Kwatra, Arno Schödl, Irfan Essa, Greg Turk, and Aaron Bobick. Graphcut textures: image and video synthesis using graph cuts. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pages 277–286, New York, NY, USA, 2003. ACM. ISBN 1-58113-709-5. doi: http://doi.acm.org/10.1145/1201775.882264.

[30] Vivek Kwatra, Irfan Essa, Aaron Bobick, and Nipun Kwatra. Texture optimization for example-based synthesis. *SIGGRAPH '05*, 2005.

[31] Ares Lagae and Philip Dutré. An alternative for wang tiles: colored edges versus colored corners. *ACM Trans. Graph.*, 25(4):1442–1459, 2006. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/1183287.1183296.

[32] Ares Lagae, Olivier Dumont, and Philip Dutre. Geometry synthesis by example. In *SMI '05: Proc. of Shape Modeling and Applications 2005*, pages 176–185, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2379-X. doi: http://dx.doi.org/10.1109/SMI.2005.24.

[33] Watson B. Wilensky U. Tisue S. Felsen M. Moddrell A. Lechner, T. Procedural modeling of land use in cities. Technical report, tech. report NWU-CS-04-38, Dept. Computer Science, Northwestern Univ., 2004.

[34] Justin Legakis, Julie Dorsey, and Steven Gortler. Feature-based cellular texturing for architectural models. In *SIGGRAPH '01*, pages 309–316, 2001. ISBN 1-58113-374-X. doi: http://doi.acm.org/10.1145/383259.383293.

[35] Aristid Lindenmayer. Mathematical models for cellular interactions in development i. filaments with one-sided inputs. *Journal of Theoretical Biology*, 18(3):280–299, March 1968.

[36] Markus Lipp, Peter Wonka, and Michael Wimmer. Interactive visual editing of grammars for procedural architecture, August 2008. ISSN 0730-0301. URL `http://www.cg.tuwien.ac.at/research/publications/2008/LIPP-2008-IEV/`. Article No. 102.

[37] Aidong Lu, David S. Ebert, Wei Qiao, Martin Kraus, and Benjamin Mora. Volume illustration using wang cubes. *ACM Trans. Graph.*, 26(2):11, e 07. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/1243980.1243985.

[38] Benoit B. Mandelbrot. *The Fractal Geometry of Nature*. W. H. Freeman, August 1982.

[39] Radomír Měch and Przemyslaw Prusinkiewicz. Visual models of plants interacting with their environment. In *SIGGRAPH '96*, pages 397–410, 1996. ISBN 0-89791-746-4. doi: http://doi.acm.org/10.1145/237170.237279.

[40] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural modeling of buildings. *ACM Trans. Graph.*, 25(3):614–623, 2006. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/1141911.1141931.

[41] Pascal Müller, Gang Zeng, Peter Wonka, and Luc Van Gool. Image-based procedural modeling of facades. *ACM Trans. Graph.*, 26(3):85, 2007. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/1276377.1276484.

[42] F. K. Musgrave, C. E. Kolb, and R. S. Mace. The synthesis and rendering of eroded fractal terrains. In *SIGGRAPH '89*, pages 41–50, 1989. ISBN 0-201-50434-0. doi: http://doi.acm.org/10.1145/74333.74337.

[43] Rupert Paget and I. D. Longstaff. Texture synthesis via a noncausal nonparametric multiscale markov random field. *IEEE Transactions on Image Processing*, 7:925–931, 1998.

[44] Yoav I. H. Parish and Pascal Müller. Procedural modeling of cities. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 301–308, New York, NY, USA, 2001. ACM. ISBN 1-58113-374-X. doi: http://doi.acm.org/10.1145/383259.383292.

[45] Darwyn R. Peachey. Solid texturing of complex surfaces. In *SIGGRAPH '85*, pages 279–286, New York, NY, USA, 1985. ACM. ISBN 0-89791-166-0. doi: http://doi.acm.org/10.1145/325334.325246.

[46] Ken Perlin. An image synthesizer. In *SIGGRAPH '85*, pages 287–296, New York, NY, USA, 1985. ACM. ISBN 0-89791-166-0. doi: http://doi.acm.org/10.1145/325334.325247.

[47] Ken Perlin. An image synthesizer. *SIGGRAPH Comput. Graph.*, 19(3):287–296, 1985. ISSN 0097-8930. doi: http://doi.acm.org/10.1145/325165.325247.

[48] Kris Popat and Rosalind W. Picard. Novel cluster-based probability model for texture synthesis, classification, and compression. In *In Visual Communications and Image Processing*, pages 756–768, 1993.

[49] Helmut Pottmann, Yang Liu, Johannes Wallner, Alexander Bobenko, and Wenping Wang. Geometry of multi-layer freeform structures for architecture. *SIGGRAPH '07*, 2007. URL `http://www.geometrie.tugraz.at/wallner/parallel.pdf`.

[50] Helmut Pottmann, Alexander Schiftner, Pengbo Bo, Heinz Schmiedhofer, Wenping Wang, Niccolo Baldassini, and Johannes Wallner. Freeform surfaces from single curved panels. *ACM Trans. Graph.*, 27(3):1–10, 2008. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/1360612.1360675.

[51] Lutz Prechelt. Why we need an explicit forum for negative results - announcement of the forum for negative results (fnr), 1997.

[52] Przemyslaw Prusinkiewicz, Aristid Lindenmayer, and James Hanan. Development models of herbaceous plants for computer imagery purposes. *SIGGRAPH Comput. Graph.*, 22(4):141–150, 1988. ISSN 0097-8930. doi: http://doi.acm.org/10.1145/378456.378503.

[53] Przemyslaw Prusinkiewicz, Lars Mündermann, Radoslaw Karwowski, and Brendan Lane. The use of positional information in the modeling of plants. In *SIGGRAPH '01*, pages 289–300, 2001. ISBN 1-58113-374-X. doi: http://doi.acm.org/10.1145/383259.383291.

[54] Long Quan, Ping Tan, Gang Zeng, Lu Yuan, Jingdong Wang, and Sing Bing Kang. Image-based plant modeling. *ACM Trans. Graph.*, 25(3):599–604, 2006. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/1141911.1141929.

[55] Andrei Sharf, Marc Alexa, and Daniel Cohen-Or. Context-based surface completion. *SIGGRAPH '04*, pages 878–887, 2004. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/1015706.1015814.

[56] Xiaohan Shi, Kun Zhou, Yiying Tong, Mathieu Desbrun, Hujun Bao, and Baining Guo. Example-based dynamic skinning in real time. In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, pages 1–8, New York, NY, USA, 2008. ACM. doi: http://doi.acm.org/10.1145/1399504.1360628.

[57] Peter Sibley, Philip Montgomery, and G. Elisabeta Marai. Wang cubes for video synthesis and geometry placement. ACM SIGGRAPH 2004 Poster Compendium, August 2004.

[58] Jeffrey Smith, Jessica Hodgins, Irving Oppenheim, and Andrew Witkin. Creating models of truss structures with optimization. *ACM Trans. Graph.*, 21(3):295–301, 2002. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/566654.566580.

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

[59] G Stiny and W J Mitchell. The palladian grammar. *Environment and Planning B: Planning and Design*, 5(1):5–18, January 1978.

[60] Jian Sun, Lu Yuan, Jiaya Jia, and Heung-Yeung Shum. Image completion with structure propagation. In *SIGGRAPH '05*, pages 861–868, New York, NY, USA, 2005. ACM. doi: http://doi.acm.org/10.1145/1186822.1073274.

[61] Ping Tan, Gang Zeng, Jingdong Wang, Sing Bing Kang, and Long Quan. Image-based tree modeling. In *SIGGRAPH '07: ACM SIGGRAPH 2007 papers*, page 87, New York, NY, USA, 2007. ACM.

[62] Xin Tong, Jingdan Zhang, Ligang Liu, Xi Wang, Baining Guo, and Heung-Yeung Shum. Synthesis of bidirectional texture functions on arbitrary surfaces. In *SIGGRAPH '02*, pages 665–672, New York, NY, USA, 2002. ACM. ISBN 1-58113-521-1. doi: http://doi.acm.org/10.1145/566570.566634.

[63] Paul Torrens, David, and Sullivan. Cellular automata and urban simulation: where do we go from here? *Environment and Planning B: Planning and Design*, 28(2): 163–168, March 2001.

[64] Greg Turk. Texture synthesis on surfaces. In *SIGGRAPH '01*, pages 347–354, New York, NY, USA, 2001. ACM. ISBN 1-58113-374-X. doi: http://doi.acm.org/10. 1145/383259.383297.

[65] Carlos Vanegas, Daniel Aliaga, Peter Wonka, Pascal Müller, Paul Waddell, and Benjamin Watson. Modeling the appearance and behavior of urban spaces. In *Eurographics 2009, State of the Art Report, EG-STAR*. Eurographics Association, 2009.

[66] Gokul Varadhan and Dinesh Manocha. Accurate minkowski sum approximation of polyhedral models. *Graph. Models*, 68(4):343–355, 2006. ISSN 1524-0703. doi: http://dx.doi.org/10.1016/j.gmod.2005.11.003.

[67] Paul Waddell. Urbansim: Modeling urban development for land use, transportation and environmental planning. *Journal of the American Planning Association*, 68: 297–314, 2002.

[68] Lvdi Wang, Yizhou Yu, Kun Zhou, and Baining Guo. Example-based hair geometry synthesis. *ACM Trans. Graph.*, 28(3):1–9, 2009. ISSN 0730-0301. doi: http://doi. acm.org/10.1145/1531326.1531362.

[69] Benjamin Watson, Pascal Müller, Oleg Veryovka, Andy Fuller, Peter Wonka, and Chris Sexton. Procedural urban modeling in practice. *IEEE Comput. Graph. Appl.*, 28(3):18–26, 2008. ISSN 0272-1716. doi: http://dx.doi.org/10.1109/MCG.2008.58.

[70] Basil Weber, Pascal Mueller, Peter Wonka, and Markus Gross. Interactive geometric simulation of 4d cities. *Computer Graphics Forum*, April 2009.

[71] Li-Yi Wei and Marc Levoy. Fast texture synthesis using tree-structured vector quantization. In *SIGGRAPH '00*, pages 479–488, 2000. ISBN 1-58113-208-5. doi: http://doi.acm.org/10.1145/344779.345009.

[72] Li-Yi Wei and Marc Levoy. Texture synthesis over arbitrary manifold surfaces. In *SIGGRAPH '01*, pages 355–360, New York, NY, USA, 2001. ACM. ISBN 1-58113-374-X. doi: http://doi.acm.org/10.1145/383259.383298.

[73] Li-Yi Wei, Jianwei Han, Kun Zhou, Hujun Bao, Baining Guo, and Heung-Yeung Shum. Inverse texture synthesis. In *SIGGRAPH '08*, pages 1–9, New York, NY, USA, 2008. ACM. doi: http://doi.acm.org/10.1145/1399504.1360651.

[74] Li-Yi Wei, Sylvain Lefebvre, Vivek Kwatra, and Greg Turk. State of the art in example-based texture synthesis. In *Eurographics 2009, State of the Art Report, EG-STAR*. Eurographics Association, 2009. URL `http://www-sop.inria.fr/reves/Basilic/2009/WLKT09`.

[75] Peter Wonka, Michael Wimmer, François Sillion, and William Ribarsky. Instant architecture. In *SIGGRAPH '03*, pages 669–677, 2003. ISBN 1-58113-709-5.

[76] Howard Zhou and Jie Sun. Terrain synthesis from digital elevation models. *IEEE Transactions on Visualization and Computer Graphics*, 13(4):834–848, 2007. ISSN 1077-2626. doi: http://dx.doi.org/10.1109/TVCG.2007.1027. Member-Turk, Greg and Member-Rehg, James M.

[Zhou et al.] Kun Zhou, Xin Huang, Xi Wang, Yiying Tong, Mathieu Desbrun, Baining Guo, and Heung-Yeung Shum. Mesh quilting for geometric texture synthesis. In *SIGGRAPH '06, year = 2006, isbn = 1-59593-364-6, pages = 690–697, location = Boston, Massachusetts, doi = http://doi.acm.org/10.1145/1179352.1141942, publisher = ACM, address = New York, NY, USA,*.

[Zhu et al.] Song Chun Zhu, Yingnian Wu, and David Mumford. Filters, random fields and maximum entropy (frame) - towards a unified theory for texture modeling. *International Journal of Computer Vision*, 27:1–20.