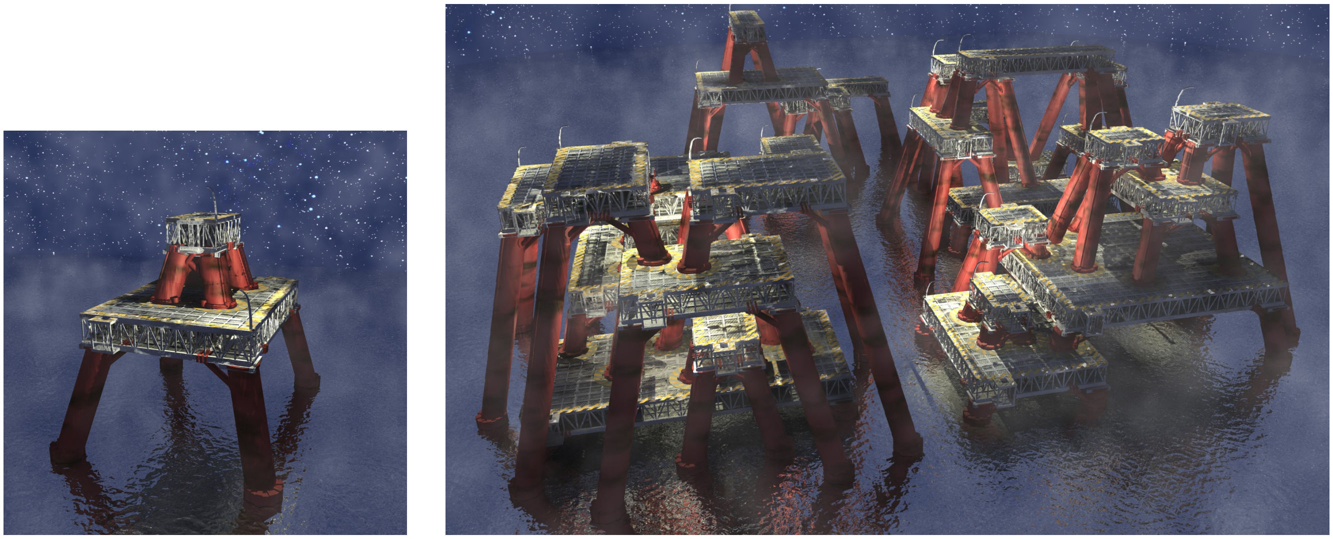
Constraint-Based Model Synthesis

Paul Merrell *   Dinesh Manocha
University of North Carolina at Chapel Hill



(a) Example Model

(b) Generated Model

Figure 1: *(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.*

Abstract

We present a method for procedurally modeling general complex 3D shapes. Our approach is targeted towards applications in digital entertainment and gaming and can automatically generate complex models of buildings, man-made structures, or urban datasets in a few minutes based on user-defined inputs. The algorithm attempts to generate results that resemble a user-defined input model and that satisfy various dimensional, geometric, and algebraic constraints. These constraints are used to capture the intent of the user and generate shapes that look more natural. We also describe efficient techniques to handle complex shapes and demonstrate their performance on many different types of models.

CR Categories: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—;

Keywords: model synthesis, procedural modeling

1 Introduction

Creating 3D digital content for computer games, movies, and virtual environments is an important and challenging problem. It is challenging because realistic scenes often require very complex and widely varying shapes and styles. Consider the problem of generating a realistic model of outdoor scenes. Different applications may require many different types of models such as buildings, oil platforms, spacecrafts, roller coasters, and other man-made structures. To generate such models, current CAD or authoring systems need a large amount of user input and manipulation and do not provide intuitive interfaces.

There is extensive literature on procedural modeling techniques which are designed to automatically or semi-automatically generate complex models. These include techniques based on shape grammars, scripting languages, L-systems, fractals, or solid texturing. These approaches have been used to generate many complex shapes, but each method is mainly limited to a specific class of models or requires considerable user input or guidance.

In this paper, we address the problem of generating complex models using model synthesis. Model synthesis is a simple technique proposed by [Merrell 2007; Merrell and Manocha 2008] to automatically generate complex shapes. The model synthesis algorithm accepts a simple 3D shape as an input and then generates a larger and more complex model that resembles the input. An example of this is shown in Figure 1.

Different procedural modeling techniques require varying degrees of user input. Using a high degree of user input has both advantages and disadvantages. Without sufficient user input, the result gener-

*http://gamma.cs.unc.edu/synthesis

ated by the algorithm may be too random and some parts of the result may turn out to be different from what the user intended. With too much user input, the time required to adjust it could overwhelm the user. Ideally, the user could choose to provide any amount of input and the algorithm should be able to adjust accordingly. The user input can often be specified in the form of a constraint on the output. Any output that satisfies all of the user's constraint is acceptable. Prior work in model synthesis [Merrell and Manocha 2008] uses a minimal amount of user input in the form of a single adjacency constraint and may not give the user enough control over the result.

We present a novel model synthesis algorithm which enables the user to specify geometric constraints that give the user greater control over the results. We use dimensional, incident, algebraic, and connectivity constraints that have been used in CAD/CAM, geometric modeling, and robotics. The constraints are specified between a set of geometric objects and their features. These include spatial and logical constraints such as incidence, tangency, perpendicularity, and metric constraints such as distance, angle, etc. We use these constraints to capture the user's intent, to prevent objects from becoming unnaturally large or small, to generate more complex shapes, and to manage the objects' spatial distribution.

In order to satisfy the constraints, we represent local neighborhoods of the objects using Boolean expressions. The Boolean expressions are used to compute how different vertices, edges and faces of the synthesized model connect together. Furthermore, we present a scheme to incorporate dimensional and algebraic constraints into our model synthesis algorithm.

Like most procedural modeling techniques, our algorithm is primarily designed to work on objects that are self-similar. We demonstrate its ability to generate models of buildings, man-made structures, city streets, plumbling, etc. In practice, our algorithm works well on man-made structures rather than organic curved shapes.

The rest of the paper is organized as follows. We give a brief survey of prior work on procedural modeling and geometric constraint systems in Section 2. Section 3 gives a brief overview of model synthesis and the constraints used by our algorithm. The overall constraint-based algorithm is described in Section 4 and we highlight its performance in Section 5. We analyze its performance and discuss some of its limitations in Section 6.

2 Related Work

A wide variety of procedural modeling techniques have been designed to model specific types of objects or environments [Ebert et al. 2002]. Plants have been generated using L-systems [Měch and Prusinkiewicz 1996; Prusinkiewicz et al. 2001]. Terrain has been modeled using fractals [Musgrave et al. 1989]. Split grammars have been used to create architectural buildings [Müller et al. 2006; Wonka et al. 2003]. Other techniques have been developed to create truss structures [Smith et al. 2002], layered solid models [Cutler et al. 2002], freeform buildings [Pottmann et al. 2007], and cellular textures of bricks and masonry on building exteriors [Legakis et al. 2001]. However, each of these methods is designed to model a specific class of objects and may require the user to specify many production rules. Shape grammars were derived automatically by Müller et al. [2007] from images of facades. Complex architecture has also been modeled by reshaping and combining existing textured models [Cabral et al. 2009].

Model synthesis was initially proposed by Merrell [2007] and later extended to handle non-axis-aligned objects [Merrell and Manocha 2008]. Model synthesis is inspired by texture synthesis. Both these methods are designed to take a small sample as an input example and generate a larger result that resembles the input example. Texture synthesis methods have become increasingly sophisticated [Efros and Leung 1999; Wei and Levoy 2000; Kwatra et al. 2003; Han et al. 2008; Kopf et al. 2007]. Another example-based technique is designed specifically for generating urban layouts [Aliaga et al. 2008].

Model synthesis relies on finding repeated geometric patterns. Many methods have been developed to find patterns in 3D models [Mitra et al. 2006].

There is rich literature in solid modeling on designing shapes that satisfy various geometric, parametric or variational constraints [Bouma et al. 1995; Ault 1999]. There is also considerable work on solving geometric constrained systems and some excellent surveys are available [Hoffmann et al. 1998; Kramer and Qh 1992]. Geometric constraints are widely used in computer aided engineering applications [Hoffmann and Rossignac 1996] and also arise in many geometric modeling contexts such as virtual reality, robotics, molecular modeling, computer vision, etc. These constraints are used to incorporate relationships between geometric entities and features and thereby capture the intent of the designers. Our formulation of various constraints is similar, though our approach to satisfy these constraints during model synthesis is different. Besides geometric constraints, silhouette-based constraints are also used to model freeform objects using sketch-based interfaces [Igarashi et al. 1999; Nealen et al. 2007].

3 Method

In this section, we give a brief overview of model synthesis and the constraints used in the algorithm.

3.1 Notation

Points and vectors are written in bold face, $\mathbf{x} \in \mathbb{R}^3$. Lower-case letters not in bold face are generally used to denote scalar variables, but there are a few exceptions. The variable h is used to denote the set of points within a half-space. The upper-case letters, E and M are used to denote the models. The model E is the input example model provided by the user. The model M is the new model generated by the algorithm. Each model is a set of closed polyhedra. The models E and M and the half-spaces h_i are represented in two different ways. A half-space h_1 could be represented as a set of points h_1 or as the characteristic function of that set $h_1(\mathbf{x})$ where $h_1(\mathbf{x}) = 1$ if $\mathbf{x} \in h_1$, otherwise $h_1(\mathbf{x}) = 0$. The complement of the half-space h_1 is written as either the set h_1^C or the function $\neg h_1(\mathbf{x})$.

3.2 Background

Our algorithm builds upon earlier work in model synthesis. In this section, we give a brief overview of a previous model synthesis algorithm [Merrell and Manocha 2008]. The user provides an example model as the input. The example model is a set of polygons that form closed polyhedral objects. Model synthesis generates a new model M that resembles the example model E. In earlier work, it was assumed that the input was a single object, but we allow multiple objects in E. Let n be the number of different objects in E. We consider the example model to be a function $E(\mathbf{x})$ of a point in space $\mathbf{x}$ where $E(\mathbf{x}) = k$ if $\mathbf{x}$ is inside an object of type k where $1 \leq k \leq n$. If $\mathbf{x}$ is not inside any of the objects, then $E(\mathbf{x}) = 0$. The function $M(\mathbf{x})$ is similarly defined for the new model M.

In the prior model synthesis algorithm, the output model only needed to satisfy a single constraint called the adjacency constraint.

The adjacency constraint is defined on neighborhoods. A neighborhood around a point is just a set of points near it. In Figure 2, the neighborhoods surrounding the points $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$, $\mathbf{d}$, and $\mathbf{e}$ exactly match the neighborhoods surrounding the points $\mathbf{a}'$, $\mathbf{b}'$, $\mathbf{c}'$, $\mathbf{d}'$, and $\mathbf{e}'$. The neighborhoods around a point $\mathbf{x}$ matches the neighborhoods around the point $\mathbf{x}'$ if there exists $\epsilon > 0$ such that for all vectors δ where $||\delta|| < \epsilon$

$$M(\mathbf{x} + \delta) = E(\mathbf{x}' + \delta). \qquad (1)$$

The adjacency constraint states that for every point $\mathbf{x}$ there exists a point $\mathbf{x}'$ whose neighborhood matches according to equation 1. This constraint ensures that every neighborhood of of M is the same as a neighborhood of E.

Figure 2 gives an overview of our approach. Starting with the input example shape $E(\mathbf{x})$ shown in Figure 2(a), the algorithm creates sets of parallel lines (or parallel planes in 3D) as shown in Figure 2(b). The output shape is generated on these sets of parallel lines. One possible output shape is shown in Figure 2(c). The lines (or planes in 3D) intersect at vertices.

Each vertex has a set of acceptable neighborhoods that match the input according to Equation 1. The vertex could be outside $M(\mathbf{x})$, inside it, or on its boundary which could be an edge, a face, or a vertex of the output shape. Each possible neighborhood is represented by a different possible state. One state might be a neighborhood which is on a face. A neighborhood on a face that has a different normal would be a different state. There are other states for neighborhoods on edges or vertices. Every neighborhood that is different according to Equation 1 is a different possible state. Several states are shown in Figures 3 and 4. Two states can be at adjacent vertices in Figure 2 if they have similar features along the direction in which they connect. For example, Figure 4 shows three states that could be beneath a particular state because they all share a vertical magneta edge that can connect the two states. Adjacent states which do not share common features conflict because they can not connect together.

After creating the planes, the next part of the algorithm is to assign states to each vertex without assigning two adjacent states that conflict. We keep track of a list of every possible state that could be assigned to each edge and each vertex. This list is long initially, but it gets shorter as we assign more states. Each assigned state is associated with a set of states that could be adjacent to it. Neighboring states outside this set conflict with the assigned state and get removed from the list. This removal may, in turn, expose other conflicting states which are also removed. This process is repeated until no more states need to be removed. We continue to assign states to each vertices and then update the list of possible states until every vertex has been assigned a single state.

3.3 Geometric Constraints

Our approach uses several geometric constraints to capture the user's intent and to control the shape of the synthesized model. To describe different constraints, we borrow terminology from the solid modeling and CAD literature [Ault 1999].

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.

Incidence Constraints: Prior model synthesis techniques are limited to shapes which have only trihedral vertices. Trihedral vertices are vertices which are incident to three faces. As a result, there are many simple shapes such as a pyramid or an octahedron that previous model synthesis techniques can not generate. To generate such shapes, we use additional incidence constraints.

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 in some way. This defines a connectivity constraint which can be used to eliminate the possibility of isolated loops 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. 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 Constraint-Based Approach

In this section, we present our constraint-based synthesis algorithm.

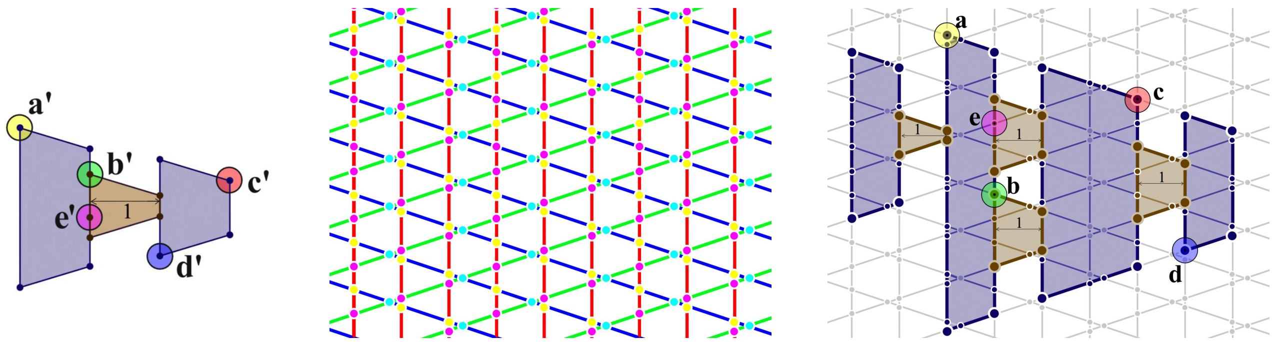
4.1 Overview

We first discuss incidence constraints that are used to specify that more than three faces are incident to a vertex to generate a non-trihedral vertex. To add these incidence constraints, we need a new way to describe the neighborhoods around non-trihedral vertices. We uses Boolean expressions as explained in Section 4.2. These representations are used to determine which neighborhoods can be adjacent to one another. The vertices of the output are constructed where several planes intersect. Vertices incident to four or more faces require that four or more planes intersect there which requires the planes be spaced more carefully as described in Section 4.4.

The Boolean expressions describing the states can also be used to apply dimensional constraints to the synthesized model. By disallowing any states that would permit the objects to stretch beyond its fixed dimensions, dimensional constraints are created as described in Section 4.5. Connectivity constraints are imposed in Section 4.6 by changing the order in which the states are assigned. Large-scale constraints are applied by changing the probabilities of the states that are assigned to each vertex, as described in Section 4.7. An algebraic constraint is described in Section 4.8.

4.2 Representing Neighborhoods with Boolean Expressions

In this section, we discuss how to describe states using Boolean expressions. The terms Boolean expression, neighborhood, and state can all be used interchangeably. The Boolean expressions are simply used to describe the neighborhoods. Our ultimate goal is to assign a neighborhood to each vertex of Figure 2(b). Each vertex



(a) Example Shape $E(\mathbf{x})$ (b) Parallel lines diving up the plane. (c) Acceptable Output Shape $M(\mathbf{x})$

Figure 2: *(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. The brown object's width is constrained to be one line spacing width. The width of the blue object is not constrained. The objects are also constrained to be fully connected.*

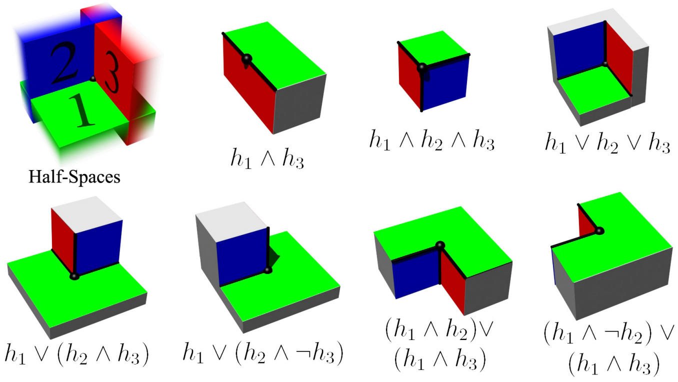

2
3
1
Half-Spaces
$h_1 \wedge h_3$
$h_1 \wedge h_2 \wedge h_3$
$h_1 \vee h_2 \vee h_3$
$h_1 \vee (h_2 \wedge h_3)$
$h_1 \vee (h_2 \wedge \neg h_3)$
$(h_1 \wedge h_2) \vee$
$(h_1 \wedge h_3)$
$(h_1 \wedge \neg h_2) \vee$
$(h_1 \wedge h_3)$

Figure 3: *One edge and six vertex states are described using Boolean expressions of three half-spaces. In our algorithm, every neighborhood is represented by a Boolean expression.*

has a set of possible neighborhoods that could be assigned to it and these are called states.

The incidence and the adjacency constraints are concerned with the neighborhoods surrounding points. To impose the adjacency constraint at non-trihedral vertices, we need a way to describe the neighborhoods there. Neighborhoods are represented using half-spaces which are related to the faces of the polyhedra. Every face has a plane that is parallel to it and that intersects the origin. This plane divides space into two half-spaces. The face's normal points into one half-space and away from the other. Let us associate each face with the half-space that its normal points away from. These half-spaces can be used to describe every neighborhood of the polyhedra using a combination of Boolean operations. A few examples of these combinations are shown in Figure 3. For every point $\mathbf{p}$ in E, there exists a Boolean expression that will produce a neighborhood identical to $\mathbf{p}$. This set of Boolean operations is found using the following method:

1. If $\mathbf{p}$ is on a face and h_i is the half-space associated with the face, then h_i alone produces a neighborhood that is identical to $\mathbf{p}$. If $\mathbf{p}$ is on a face whose normal points in the opposite direction, then h_i^C describes the neighborhood around $\mathbf{p}$.

2. If $\mathbf{p}$ is on an edge whose two adjacent faces are associated with the two half-spaces h_i and h_j, then the neighborhood around $\mathbf{p}$ 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.

3. If $\mathbf{p}$ is on a vertex, then the procedure for computing its neighborhood's Boolean expression is more complex. Every face that intersects $\mathbf{p}$ is on a plane. Let us take all the faces that intersect $\mathbf{p}$ and use all of their planes to divide the space into cells. An example of this is shown in Figure 5. Each cell is the intersection of several half-spaces. Since the planes all intersect $\mathbf{p}$, every cell has points in the neighborhood of $\mathbf{p}$. For each cell, we determine if the points within the cell and within the neighborhood of $\mathbf{p}$ are inside or outside the polyhedron E. We take the union of all cells which have points inside the polyhedron and this is the Boolean expression that represents the neighborhood surrounding $\mathbf{p}$. Each cell is the intersection of several half-spaces and so the neighborhood at $\mathbf{p}$ is represented as a union of intersections. These expressions can often be simplified using familiar 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 states are shown in Figures 3 and 5.

This method gives us a Boolean expression describing how a polyhedra intersects a neighborhood at any point $\mathbf{p}$. However, more than one polyhedra might intersect at the same point. For example, see the points $\mathbf{b}$ and $\mathbf{e}$ of Figure 2(c). When multiple objects intersect, we can compute a Boolean expression for each object and then combine all the expressions into one. Let k_1 and k_2 be two object interiors and let b_1 and b_2 be two Boolean expressions. The notation $k_1 \cdot b_1 + k_2 \cdot b_2$ will be used to describe a neighborhood which contains the object k_1 at the points b_1 and object k_2 at b_2. For example, $1 \cdot h_1 \cap h_2^C + 3 \cdot h_2$ describes a neighborhood where an edge $h_1 \cap h_2^C$ of object 1 touches a face of h_2 of object 3.

4.3 Evaluating Boolean Expressions along Edges

In the previous section, we discuss how to describe every neighborhood or every state as a Boolean expression, but we still need to determine which states can be next to one another. The Boolean expressions can be thought about in more than one way. We have been thinking in terms of union and intersection of sets, but we could replace them with OR and AND operations of functions. Each half-space has a characteristic function $h_1(\mathbf{x})$ which evaluates to 1 if $\mathbf{x}$ is inside the half-space and to 0 if $\mathbf{x}$ is in the opposite half-space. However, a third possibility is that $\mathbf{x}$ could intersect the plane di-

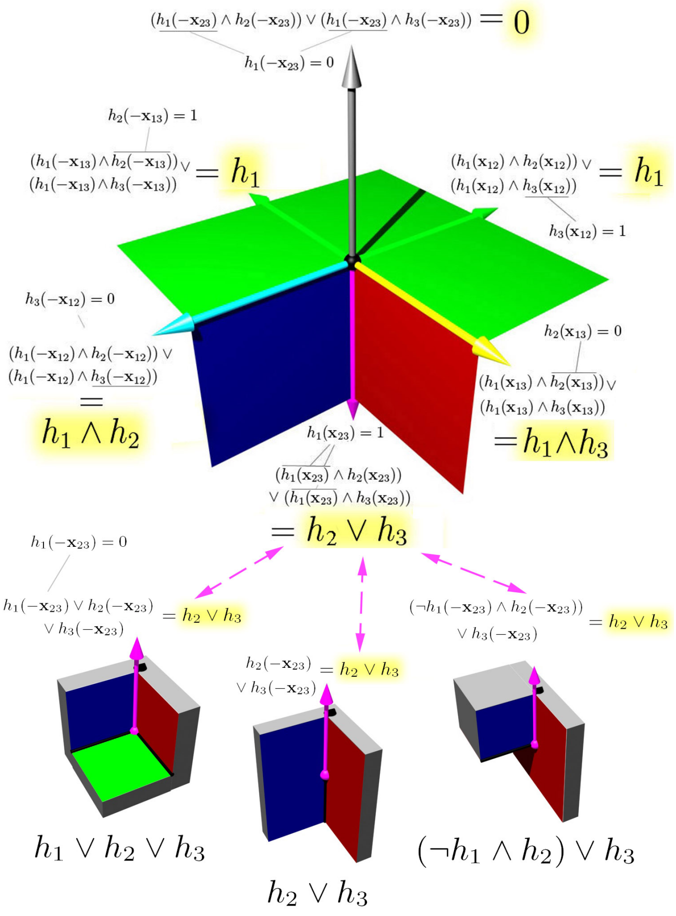

$(h_1(-\mathbf{x}_{23}) \wedge h_2(-\mathbf{x}_{23})) \vee (h_1(-\mathbf{x}_{23}) \wedge h_3(-\mathbf{x}_{23})) = 0$
$h_1(-\mathbf{x}_{23}) = 0$
$h_2(-\mathbf{x}_{13}) = 1$
$(h_1(-\mathbf{x}_{13}) \wedge h_2(-\mathbf{x}_{13})) \vee (h_1(-\mathbf{x}_{13}) \wedge h_3(-\mathbf{x}_{13})) = h_1$
$(h_1(\mathbf{x}_{12}) \wedge h_2(\mathbf{x}_{12})) \vee (h_1(\mathbf{x}_{12}) \wedge h_3(\mathbf{x}_{12})) = h_1$
$h_3(\mathbf{x}_{12}) = 1$
$h_3(-\mathbf{x}_{12}) = 0$
$(h_1(-\mathbf{x}_{12}) \wedge h_2(-\mathbf{x}_{12})) \vee (h_1(-\mathbf{x}_{12}) \wedge h_3(-\mathbf{x}_{12})) = h_1 \wedge h_2$
$h_2(\mathbf{x}_{13}) = 0$
$(h_1(\mathbf{x}_{13}) \wedge h_2(\mathbf{x}_{13})) \vee (h_1(\mathbf{x}_{13}) \wedge h_3(\mathbf{x}_{13})) = h_1 \wedge h_3$
$h_1(\mathbf{x}_{23}) = 1$
$(h_1(\mathbf{x}_{23}) \wedge h_2(\mathbf{x}_{23})) \vee (h_1(\mathbf{x}_{23}) \wedge h_3(\mathbf{x}_{23})) = h_2 \vee h_3$
$h_1(-\mathbf{x}_{23}) = 0$
$h_1(-\mathbf{x}_{23}) \vee h_2(-\mathbf{x}_{23}) \vee h_3(-\mathbf{x}_{23}) = h_2 \vee h_3$
$h_2(-\mathbf{x}_{23}) \vee h_3(-\mathbf{x}_{23}) = h_2 \vee h_3$
$(\neg h_1(-\mathbf{x}_{23}) \wedge h_2(-\mathbf{x}_{23})) \vee h_3(-\mathbf{x}_{23}) = h_2 \vee h_3$
$h_1 \vee h_2 \vee h_3$
$h_2 \vee h_3$
$(\neg h_1 \wedge h_2) \vee h_3$

Figure 4: *We can evaluate the states to figure out which states can be adjacent to them in various directions. This shows the state $(h_1 \wedge h_2) \vee (h_1 \wedge h_3)$ 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 state $(h_1 \wedge h_2) \vee (h_1 \wedge h_3)$ evaluates to $h_2 \vee h_3$ which is shown as the magneta edge. The only states that can be directly beneath the state $(h_1 \wedge h_2) \vee (h_1 \wedge h_3)$ are states which share the same $h_2 \vee h_3$ edge. We test every state to see which ones evaluate to $h_2 \vee h_3$ in the up direction which is $-\mathbf{x}_{23}$. Three examples of acceptable states are shown: $h_1 \vee h_2 \vee h_3$, $h_2 \vee h_3$, and $(\neg h_1 \wedge h_2) \vee h_3$. More than three acceptable states exist.*

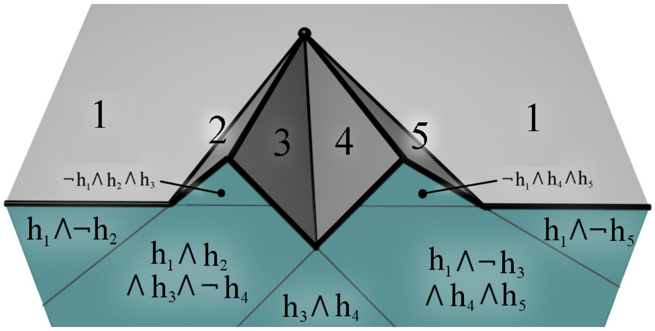

1
2
3
4
5
1
$\neg h_1 \wedge h_2 \wedge h_3$
$\neg h_1 \wedge h_4 \wedge h_5$
$h_1 \wedge \neg h_2$
$h_1 \wedge h_2 \wedge h_3 \wedge \neg h_4$
$h_3 \wedge h_4$
$h_1 \wedge \neg h_3 \wedge h_4 \wedge h_5$
$h_1 \wedge \neg h_5$

Figure 5: *Even complex vertices can be described using a Boolean expression of half-spaces. We can take all the faces that intersect the vertex and use all of the planes the faces are on to divide up the space into regions that are the intersection of several half-spaces. The regions labeled in the figure are part of the interior of the polyhedron. The vertex state can be described as the union of all the regions inside the polyhedron. The expression for this vertex when simplified is $(h_3 \wedge (h_1 \vee h_2)) \vee (h_4 \wedge (h_1 \vee h_5))$.*

viding the two half-spaces. In this case, we do not evaluate $h_1(\mathbf{x})$ as 0 or 1, but leave it as the symbol h_1. This symbolic representation provides a convenient way to determine how the states connect together.

The symbols are used when a point $\mathbf{x}$ is on a plane. To keep track of which planes an input point is on, we will use subscripts. According to this notation, the point $\mathbf{x}_{12} = \mathbf{n}_1 \times \mathbf{n}_2$ is on the planes of h_1 and h_2.

The Boolean expressions can describe many different neighborhoods or states, including vertices, edges, and faces. When we evaluate the expression at a point $\mathbf{x}$, essentially we get a new state that describes what we would encounter if we travel away from a neighborhood in the direction of $\mathbf{x}$. If the expression evaluates to 0, we travel into empty space. If it evaluates to 1, we travel into an object's interior. If it evaluates to h_1, we travel onto a face. If it evaluates to something like $h_1 \wedge h_2$ or $h_1 \vee h_2$, then we travel onto an edge. We determine which states can be next to one another by evaluating two states in opposite directions and checking if their evaluations are identical. In Figure 4, the state $(h_1 \wedge h_2) \vee (h_1 \wedge h_3)$ evaluates in the down direction to $h_2 \vee h_3$. Any state that evaluates to $h_2 \vee h_3$ in the up direction can be beneath the state $(h_1 \wedge h_2) \vee (h_1 \wedge h_3)$ and three examples of such states are shown in Figure 4.

The Boolean expressions may contain more than one object interior. In this case, we evaluate each object interior 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 the 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, one from object 1 and another from object 3, which touch.

4.4 Spacing the Planes

For each vertex of Figure 2(b), we compute a list of possible states. Each state corresponds to a neighborhood which can be described by a Boolean expression. We first find all states found in the input model. Each face, each edge, each vertex, and each object interior is a different state. However, only a small fraction of these states are acceptable at each vertex of Figure 2(b). To be acceptable, every half-space used in the state's Boolean expression must be associated with a plane that the vertex intersects. States that use four or more half-spaces pose a problem, since these states are only allowed at vertices where four or more planes intersect. States of trihedral vertices do not have this problem. They only use three half-spaces; three half-spaces require three planes to intersect; and those three planes must intersect somewhere. But four planes may not intersect anywhere. For these cases, we need to choose the plane spacing so that four planes intersect.

We first discuss how to compute the points where three planes intersect and then discuss how to get a fourth plane to intersect the same points. Let $\mathbf{n}_1, \mathbf{n}_2$, and $\mathbf{n}_3$ be the normals of three sets of planes. All planes within the each set are parallel and evenly spaced. Let s_1, s_2, and s_3 be the spacing between the planes within each set. The first and second sets of planes intersect at a line that points in the $\mathbf{n}_1 \times \mathbf{n}_2$ direction. If $\mathbf{p}$ is a point on the line, then $\mathbf{p}'$ is also on the line if $\mathbf{p}' - \mathbf{p} = (\mathbf{n}_1 \times \mathbf{n}_2)t$ for some 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,

$$\mathbf{p}' = \mathbf{p} + c_3 s_3 \frac{\mathbf{n}_1 \times \mathbf{n}_2}{\mathbf{n}_3 \cdot (\mathbf{n}_1 \times \mathbf{n}_2)} \tag{2}$$

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 $\mathbf{n}_1 \times \mathbf{n}_3$ and $\mathbf{n}_2 \times \mathbf{n}_3$ directions. Three planes intersect at the points

$$\mathbf{p}' = \mathbf{p} + \frac{c_1 s_1 \mathbf{n}_2 \times \mathbf{n}_3}{\mathbf{n}_1 \cdot (\mathbf{n}_2 \times \mathbf{n}_3)} + \frac{c_2 s_2 \mathbf{n}_1 \times \mathbf{n}_3}{\mathbf{n}_2 \cdot (\mathbf{n}_1 \times \mathbf{n}_3)} + \frac{c_3 s_3 \mathbf{n}_1 \times \mathbf{n}_2}{\mathbf{n}_3 \cdot (\mathbf{n}_1 \times \mathbf{n}_2)} \tag{3}$$

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

$$\begin{aligned} s_4 &= s_1 \frac{c_1 \mathbf{n}_4 \cdot (\mathbf{n}_2 \times \mathbf{n}_3)}{c_4 \mathbf{n}_1 \cdot (\mathbf{n}_2 \times \mathbf{n}_3)} + s_2 \frac{c_2 \mathbf{n}_4 \cdot (\mathbf{n}_1 \times \mathbf{n}_3)}{c_4 \mathbf{n}_2 \cdot (\mathbf{n}_1 \times \mathbf{n}_3)} \\ &+ s_3 \frac{c_3 \mathbf{n}_4 \cdot (\mathbf{n}_1 \times \mathbf{n}_2)}{c_4 \mathbf{n}_3 \cdot (\mathbf{n}_1 \times \mathbf{n}_2)} \end{aligned} \tag{4}$$

for some $c_4 \in \mathbb{Z}$. This equation needs to be solved to get four planes to intersect at multiple points. This algebraic relation represents multiple equations that need to be solved since each combination of c_1, c_2, and c_3 produces another equation. If we solve this equation for the combinations $(c_1, c_2, c_3, c_4) = (1, 0, 0, 1)$ 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. Essentially, we have three equations and four unknowns s_1, s_2, s_3, and s_4. By solving for these linear equations, we produce a 3D lattice of points where a non-trihedral vertex state may appear. However, this only takes care of a single non-trihedral vertex state. There may be more of these states in the input and they would each require more equations to be solved. There are even more difficult vertex states to handle like the vertex shown in Figure 5 which involve 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 pyrmaid. 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 that the shapes better fit on a lattice, but not so much that the normals significantly change the results. A third option is to leave a few of the equations unsatisfied. When this happens, non-trihedral vertices will be generated at fewer locations, but this might be adequate to produce a good final result.

4.5 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.

Since the objects are created on sets of evenly spaced planes, the lengths of each object must be an integer multiple of the plane spacing. Objects with non-integer dimensions like for example 1.5-plane spaces can pose a problem. To deal with these objects the planes could be spaced more closely. If they are spaced twice as close, an object that was 1.5-plane spaces wide would become three planes wide which is a round number. 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 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 force an object to be 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 6 shows a simple 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. To stop their growth, we disallow all vertex states in which the object passes through the plane. The object is on both sides of the plane if $h_2 \wedge b \neq 0$ and $\neg h_2 \wedge b \neq 0$ where h_2 is the half-space parallel to the plane and b is the Boolean expression describing the vertex state. By removing all states where $h_2 \wedge b \neq 0$ and $\neg h_2 \wedge b \neq 0$, we guarantee that the objects do not grow more than one plane-space wide.

4.6 Connectivity Constraints

In many applications, controlling the connections between objects is important. For example, this is important when creating urban models with roads. In most cities, one could choose any two points on a road map and find a path that connects them. However, model synthesis algorithms could generate isolated loops or cycles of road networks that are not connect to each other. This problem can be addressed by changing the order in which the states are assigned. We begin by choosing a starting location at random and creating an object (e.g. road) there. Then the roads are all grown out from this initial seed. This means that we only assign road states to vertices

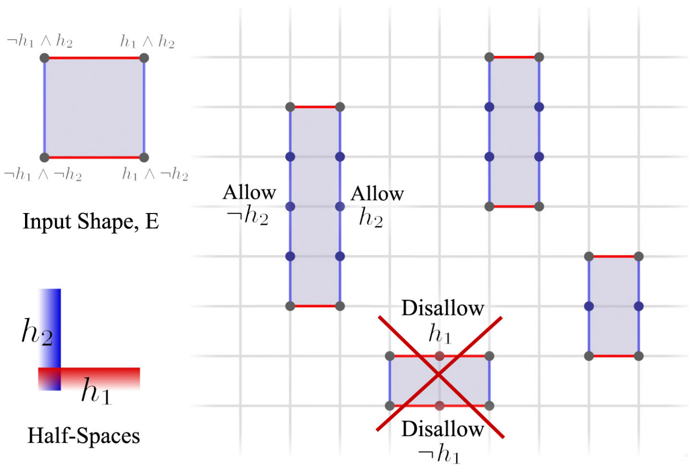


Figure 6: *Dimensional Constraint. To create objects that are only one plane space wide horizontally, we disallow any states which pass through the vertical h_2 planes such as h_1 or $\neg h_1$. In this way, our algorithm can satisfy dimensional constraints.*

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 is to not use seeds at all and to assign the states in any order. This is useful when the user wants to create many 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, everything could be grown out not from a single seed, but from multiple seeds.

4.7 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 choose to give each state an equal probability of being chosen, but we could easily modify the probabilities so that they are higher for any particular objects the user wants to appear within some areas. The user could even set some probabilities to be zero in some places. If a state's probability drops to zero, we can remove it entirely and then propagate the removal as usually done when assigning states (see Section 3.2). 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.

4.8 Algebraic Constraints and Bounding Volumes

The model synthesis algorithm creates a set of parallel planes for every distinct normal of the input. As a result, handling curved input models with many distinct normals are computationally expensive because of the large number of planes that would have to be created. However, the number of distinct normals can be greatly reduced by using bounding boxes and other bounding volumes in place of complex objects. The algorithm could be run using the bounding volumes in place of the input model and complex objects can be substituted back into the output model M after it is generated.

	Input Size (polygons)	Output Size (polygons)	Time (minutes)
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 1: *Complexity of the input and output models and computation time for various results computed on a 2.8 GHz single-core PC.*

There are several alternative ways the user can constrain the dimensions. The object's dimensions could scale freely in a direction or be fixed (see Section 4.5). A third option is to let an object scale, but to require that it must scale uniformly in two or three directions. For example, the cylinder in Figure 7 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. 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 7. 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 be uniform in all directions $s_x = s_y = s_z$. For example, the dome in Figure 7 remains spherical only in this case. This can be accomplished by placing a bounding box around the sphere and cutting off a tetrahedron as shown in Figure 7. Since model synthesis scales tetrahedra uniformly in all directions, the output will create a uniformly scaled copy of the bounding box.

5 Results

Figures 1, 7 - 13 show a variety of models that were generated using our algorithm. The generated models are large and detailed and it would be quite difficult to model them manually using a CAD or authoring system. The models each satisfy multiple constraints which depend on the application. Dimensional constraints are used in Figure 1 to give the platforms and beams a fixed thickness. They are also used to constrain the width of the road in Figure 10, the width of the spacecrafts in Figure 9, the size of the pipes in Figure 11, and the width of the roller coaster track in Figure 12. Incidence constraints are repeatedly used in Figure 8 to create architecture with four faces touching at a single vertex. Connectivity constraints are used in Figures 1, 8, and 9 to grow the objects out from a few seeds and this controls the distribution of the objects so they are not all crowded together. The roads in Figure 10 and the pipes in Figure 11 are fully connected to a single seed. In Figure 9, the parts of the spaceships are connected by beams and have gaps in between. Figure 9 demonstrates that model synthesis can generate shapes which have a high genus. Bounding volumes were used in Figures 1 and 9 to generate curved objects. Algebraic constraints were used in Figure 7. A large-scale constraint is used in Figure 13 to generate several different types of objects in the form of the characters "GPM".

Each of the models was generated without requiring much effort from the user. The input models E are composed of only a few dozen polygons. Each of the constraints can be specified by only changing a few parameters or in the case of Figure 13 an image of the letters "GPM" Table 1 shows the computation time for modeling each shape and the size of the input and output models. The size is given in terms of the polygon count of the bounding volumes. All of the displayed images include artistic decorations to the vertices and edges and some include complex objects that were generated from bounding volumes. The polygon count does not include any of these decorations. The computation time depends both on the output size and on which input model is used. Some models can be computed much more quickly than others. The road model (Figure 10) can be computed quickly because it is flat and does not really use all three spatial dimensions. The 'GPM' model takes the longest time to compute because it uses several different input models including a spaceship model and several building models.

6 Analysis and Comparison

Other methods are also able to efficiently produce large detailed models of buildings [Müller et al. 2006; Wonka et al. 2003]. They are targeted specifically for modeling buildings and are less useful for modeling other shapes. They also may require more guidance from the user. These algorithms use shape grammars to construct the shapes. In order to produce different shapes, the user must specify and adjust many production rules of the grammar. This requires more effort from the user.

Previous model synthesis techniques [Merrell and Manocha 2008] only use the adjacency constraint. Using prior algorithms without the dimensional and algebraic constraints, most of the results would appear distorted and unnatural. Without the connectivity constraint, the resulting models would contain mostly small crowded objects. Without the incidence constraints, none of the buildings in Figure 8 would be generated. The large-scale constraint is required to generate the pattern in Figure 13.

Limitations: The amount of time and memory that model synthesis needs depends on the number of vertices. Vertices are generated wherever three or more planes intersect. The number of planes depends on the number of distinct face normals. If there are n distinct normals and m parallel planes for each normal, there could be up to $O(n^3m^3)$ vertices. The number of distinct normals can be reduced by using bounding volumes, but only to a certain extent since bounding volumes can oversimplify some objects. This makes generating curved objects using model synthesis especially difficult. A related problem is that it is difficult to generate both large and small objects simultaneously. Small objects require closely spaced planes while large objects require large volumes which together means that many planes must be created.

Like most procedural modeling techniques, model synthesis is designed to work on objects that are self-similar. Model synthesis works best on objects with parts that are not only similar to each other, but that identically match. Objects without parts that exactly match can be used with model synthesis, but they often produce results that match the input too closely. Since model synthesis is intended for applications in digital entertainment or gaming, we assume that objects in the input are free from significant errors in the vertex positions. Model synthesis works efficiently on man-made structures that can be represented with a few planar faces, but it is inefficient when used on organic and curved shapes because of the large number of distinct normals n that these shapes have.

Another limitation is that the objects often need to have a grid structure. The grid is a necessary part of some of the constraints. The dimensional constraint assumes the dimensions fit on a grid. The incidence constraint assumes that the vertices can be fit onto a grid. The structure of the grid depends on the plane spacing which can be altered to accommodate some shapes, but not all shapes as explained in Section 4.4. Some shapes may produce an overconstrained set of equations when using Equation 4. Several strategies for dealing with this problem were discussed in Section 4.4, but each of them has downsides.

7 Conclusion and Future Work

We have presented several major improvements to model synthesis that allow the user to more effectively control the output. We enable the user to fix dimensions of objects, to specify a large-scale structure of the output, to produce connected results, to add bounding volumes, to have multiple object interiors, and to generate shapes with complex vertex states. Further work is needed to improve the efficiency of model synthesis, especially when generating large and small objects together. More work is needed for handling curved objects beyond using bounding volumes. One important constraint that is still missing is one to create symmetrical objects.

Acknowledgements

We would like to thank the reviewers for their comments. This work was supported in part by W911NF-04-1-0088, NSF award 0636208, DARPA/RDECOM Contracts N61339-04-C-0043 and WR91CRB-08-C-0137, Intel, and Microsoft.

References

ALIAGA, D. G., VANEGAS, C. A., AND BENEŠ, B. 2008. Interactive example-based urban layout synthesis. *ACM Trans. Graph. 27*, 5, 1–10.

AULT, H. 1999. Using geometric constraints to capture design intent. *Journal for Geometry and Graphics 3*, 1, 39–47.

BOUMA, W., FUDOS, I., HOFFMANN, C., CAI, J., AND PAIGE, R. 1995. A geometric constraint solver. *Computer-Aided Design 27*, 6, 487–501.

CABRAL, M., LEFEBVRE, S., DACHSBACHER, C., AND DRETTAKIS, G. 2009. Structure preserving reshape for textured architectural scenes. *Computer Graphics Forum (Proceedings of the Eurographics conference).*

CUTLER, B., DORSEY, J., MCMILLAN, L., MÜLLER, M., AND JAGNOW, R. 2002. A procedural approach to authoring solid models. *ACM Trans. Graph. 21*, 3, 302–311.

EBERT, D. S., MUSGRAVE, F. K., PEACHEY, D., PERLIN, K., AND WORLEY, S. 2002. *Texturing and Modeling*. 3rd ed. Academic Press.

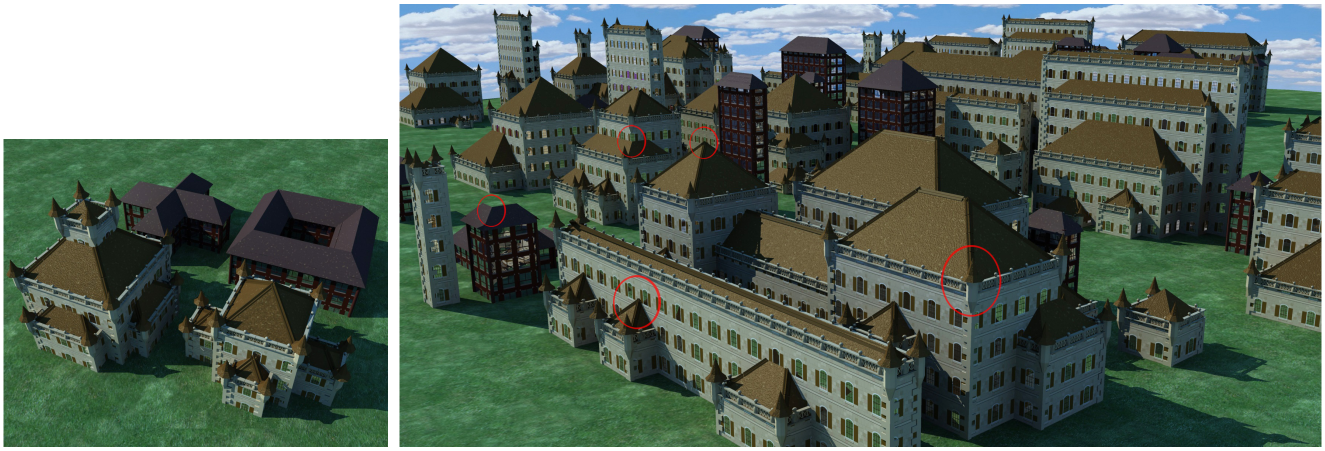
EFROS, A. A., AND LEUNG, T. K. 1999. Texture synthesis by non-parametric sampling. In *IEEE International Conference on Computer Vision*, 1033–1038.

HAN, C., RISSER, E., RAMAMOORTHI, R., AND GRINSPUN, E. 2008. Multiscale texture synthesis. *Proceedings of ACM SIGGRAPH '08 27*, 3, 51.

HOFFMANN, C. M., AND ROSSIGNAC, J. R. 1996. A road map to solid modeling. *IEEE Transactions on Visualization and Computer Graphics 2*, 1, 3–10.

HOFFMANN, C. M., LOMONOSOV, A., AND SITHARAM, M. 1998. Geometric constraint decomposition. In *Geometric Constraint Solving*, B. Bruderlin and D. Roller, Eds. Springer-Verlag, 170–195.

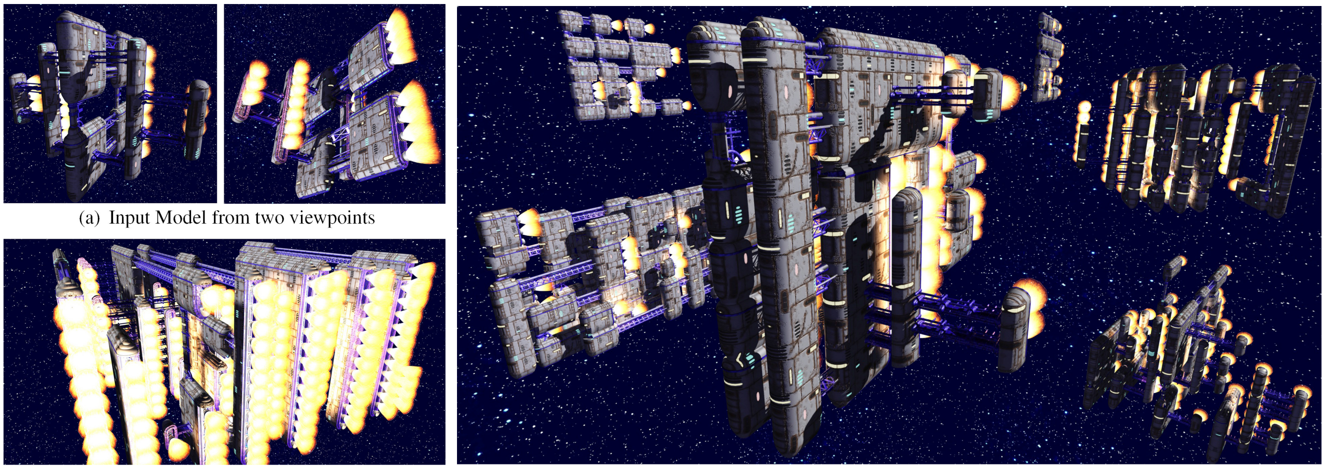
IGARASHI, T., MATSUOKA, S., AND TANAKA, H. 1999. Teddy: a sketching interface for 3d freeform design. In *Proc. of ACM SIG-*



(a) Input Model

(b) Output Model

Figure 8: *Many complex buildings (b) are generated from four simple ones (a). The output contains many vertices that have been constrained to intersect four faces and a few of these vertices are circled. The result also uses the connectivity constraint to space the buildings apart which gives the buildings more room to develop into more interesting shapes.*

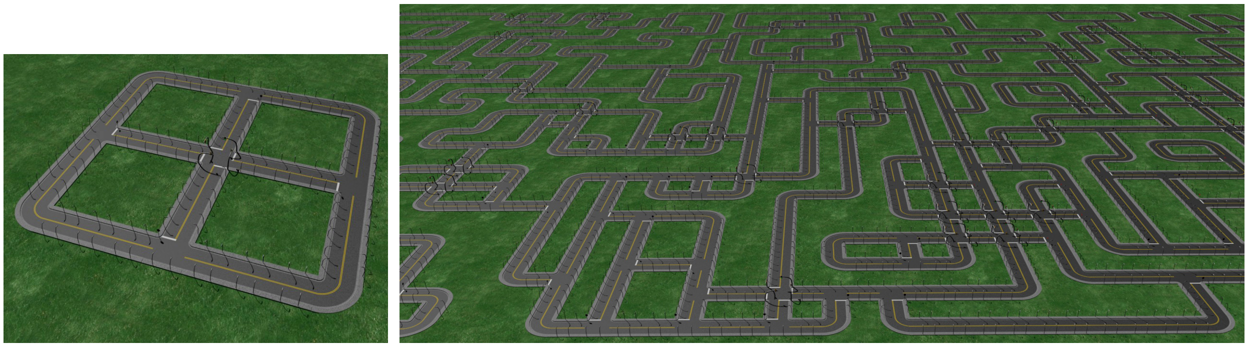


(a) Input Model from two viewpoints

(b) Output without the Connectivity Constraint

(c) Output with the Connectivity Constraint

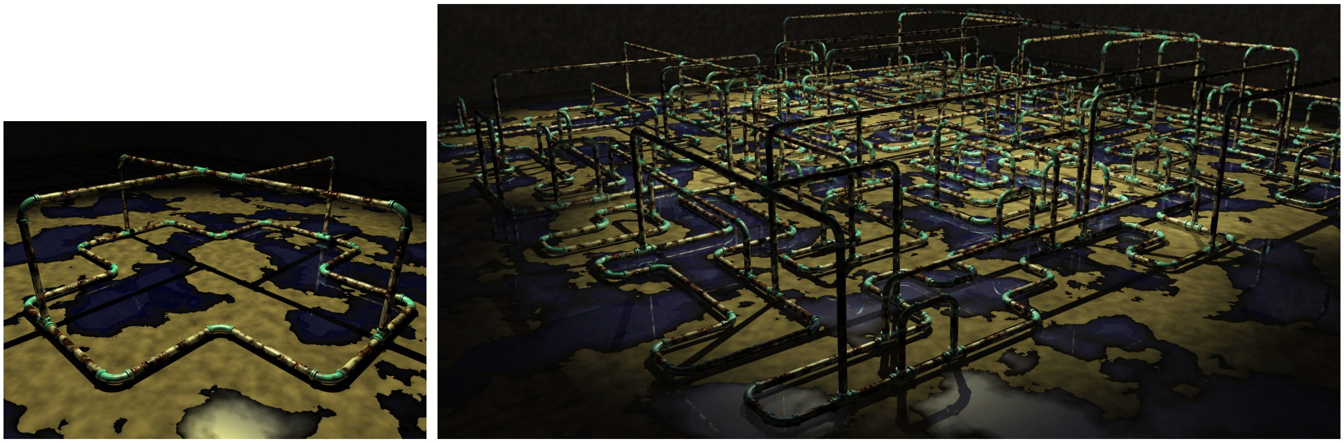
Figure 9: *A fleet of spaceships (b,c) is automatically generated from a simple spaceship model (a). Without the connectivity constraint several dozen small unconnected spaceships are generated (b), but they are all packed together. With the connectivity constraint, six large spaceships are generated (c). Dimensional constraints are extensively used to ensure 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

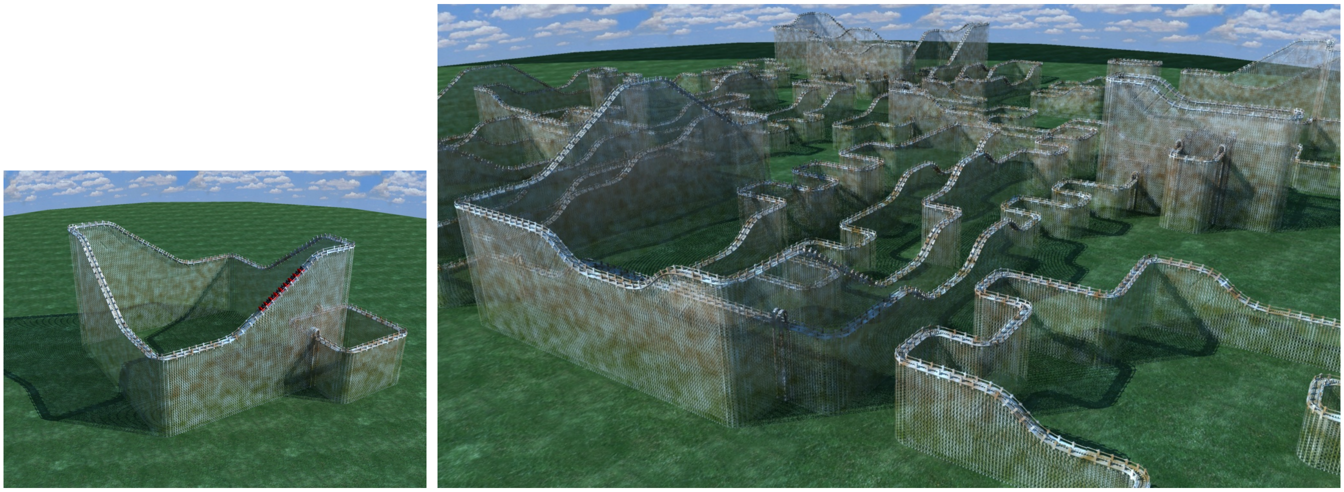
Figure 10: *A large fully connected road network is generated (b) from a few streets using the connectivity constraint. The dimensions of the roads are also constrained.*



(a) Input Model

(b) Output Model

Figure 11: *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 12: *Several long roller coasters (b) are generated from one simple ones (a). Dimensional constraints are used to keep the track a certain width.*

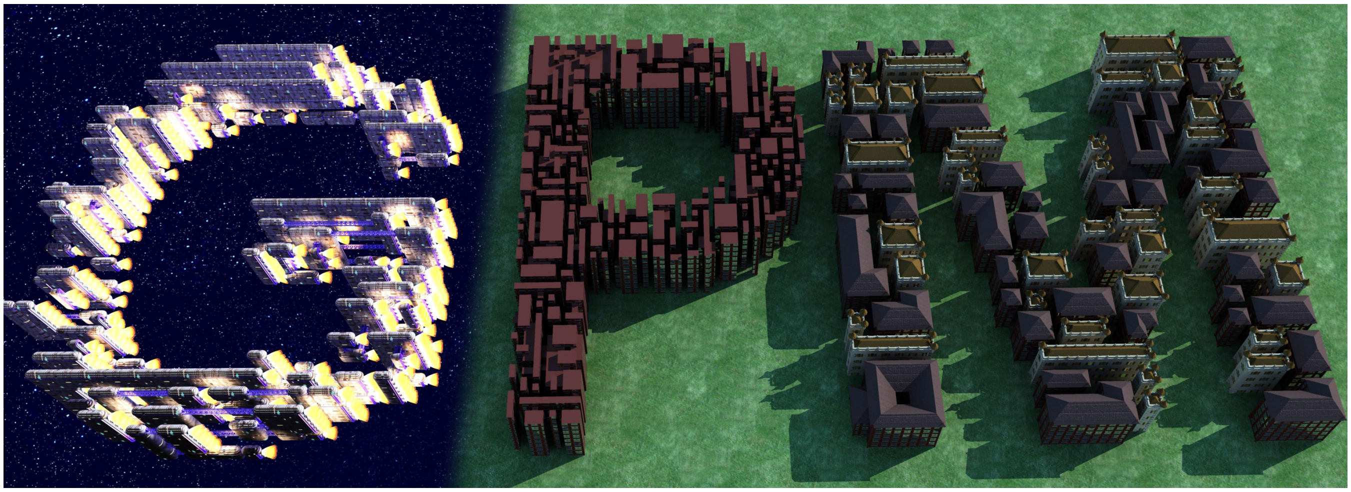
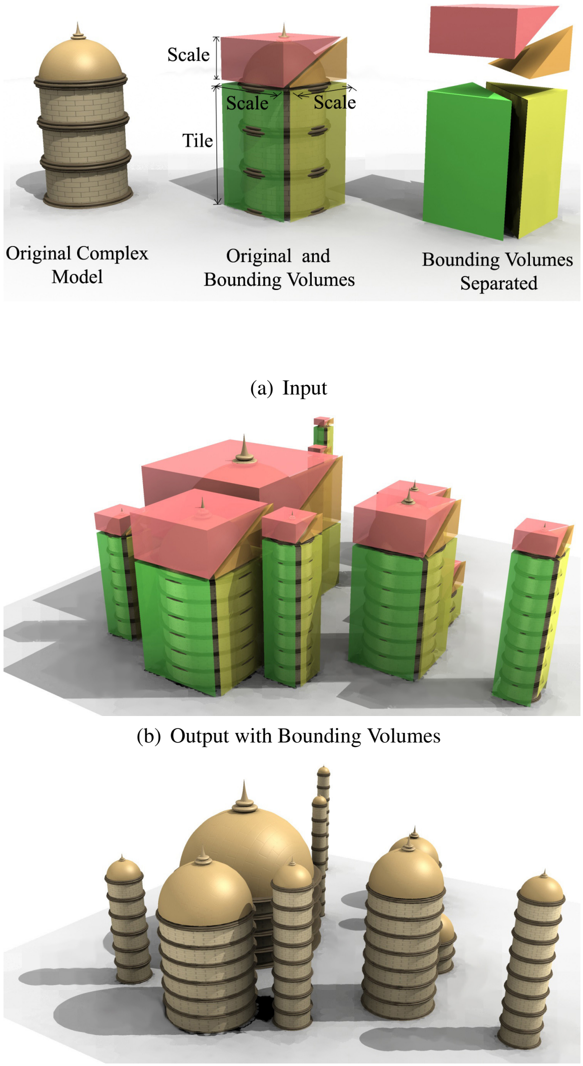


Figure 13: *Large-scale constraints are used to build spaceships in the shape of the letter ‘G’, rectangular buildings in the shape of the letter ‘P’, and buildings from Figure 8 in the shape of the letter ‘M’.*



(a) Input

(b) Output with Bounding Volumes

(c) Output without Bounding Volumes

Figure 7: *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.*

GRAPH '99, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 409–416.

KOPF, J., FU, C.-W., COHEN-OR, D., DEUSSEN, O., LISCHINSKI, D., AND WONG, T.-T. 2007. Solid texture synthesis from 2d exemplars. *ACM Trans. Graph. 26*, 3, 2.

KRAMER, G. A., AND QH, B. B. 1992. Solving geometric constraint systems. MIT Press, 708–714.

KWATRA, V., SCHDL, A., ESSA, I., TURK, G., AND BOBICK, A. 2003. Graphcut textures: Image and video synthesis using graph cuts. *Proc. Of ACM SIGGRAPH '03*, 277–286.

LEGAKIS, J., DORSEY, J., AND GORTLER, S. 2001. Feature-based cellular texturing for architectural models. In *Proc. Of ACM SIGGRAPH '01*, 309–316.

MERRELL, P., AND MANOCHA, D. 2008. Continuous model synthesis. *Proc. of ACM SIGGRAPH ASIA '08*.

MERRELL, P. 2007. Example-based model synthesis. In *I3D '07: Symposium on Interactive 3D graphics and games*, ACM Press, 105–112.

MITRA, N. J., GUIBAS, L., AND PAULY, M. 2006. Partial and approximate symmetry detection for 3d geometry. In *ACM Transactions on Graphics*, vol. 25, 560–568.

MĚCH, R., AND PRUSINKIEWICZ, P. 1996. Visual models of plants interacting with their environment. In *Proc. Of ACM SIGGRAPH '96*, 397–410.

MÜLLER, P., WONKA, P., HAEGLER, S., ULMER, A., AND GOOL, L. V. 2006. Procedural modeling of buildings. *ACM Trans. Graph. 25*, 3, 614–623.

MÜLLER, P., ZENG, G., WONKA, P., AND GOOL, L. V. 2007. Image-based procedural modeling of facades. *ACM Trans. Graph. 26*, 3, 85.

MUSGRAVE, F. K., KOLB, C. E., AND MACE, R. S. 1989. The synthesis and rendering of eroded fractal terrains. In *Proc. Of ACM SIGGRAPH '89*, 41–50.

NEALEN, A., IGARASHI, T., SORKINE, O., AND ALEXA, M. 2007. Fibermesh: designing freeform surfaces with 3d curves. *Proc. of ACM SIGGRAPH '07 26*, 3, 41.

POTTMANN, H., LIU, Y., WALLNER, J., BOBENKO, A., AND WANG, W. 2007. Geometry of multi-layer freeform structures for architecture. *Proc. Of ACM SIGGRAPH '07*.

PRUSINKIEWICZ, P., MÜNDERMANN, L., KARWOWSKI, R., AND LANE, B. 2001. The use of positional information in the modeling of plants. In *Proc. Of ACM SIGGRAPH '01*, 289–300.

SMITH, J., HODGINS, J., OPPENHEIM, I., AND WITKIN, A. 2002. Creating models of truss structures with optimization. *ACM Trans. Graph. 21*, 3, 295–301.

WATSON, B., MÜLLER, P., VERYOVKA, O., FULLER, A., WONKA, P., AND SEXTON, C. 2008. Procedural urban modeling in practice. *IEEE Computer Graphics and Applications 28*, 3, 18–26.

WEI, L.-Y., AND LEVOY, M. 2000. Fast texture synthesis using tree-structured vector quantization. In *Proc. Of ACM SIGGRAPH '00*, 479–488.

WONKA, P., WIMMER, M., SILLION, F., AND RIBARSKY, W. 2003. Instant architecture. In *Proc. Of ACM SIGGRAPH '03*, 669–677.