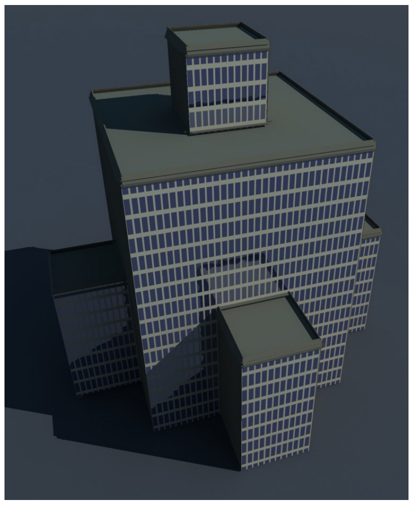
Continuous Model Synthesis

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


(a) Example Model

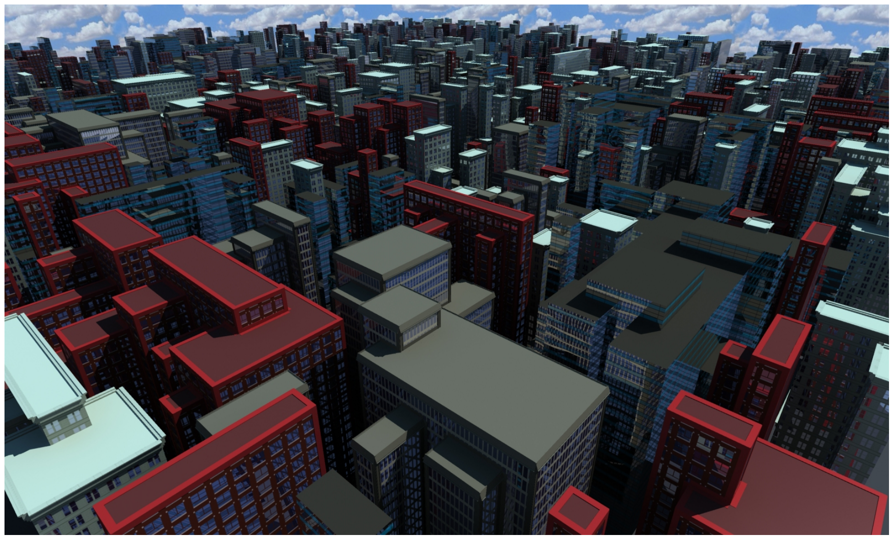

(b) Synthesized Model

Figure 1: *We demonstrate an application of our procedural modeling technique 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.*

Abstract

We present a novel method for procedurally modeling large complex shapes. Our approach is general-purpose and takes as input any 3D polyhedral model provided by a user. The algorithm exploits the connectivity between the adjacent boundary features of the input model and computes an output model that has similar connected features and resembles the input. No additional user input is needed to guide the model generation and the algorithm proceeds automatically. In practice, our algorithm is simple to implement and can generate a variety of complex shapes representing buildings, landscapes, and 3D fractal shapes in a few minutes.

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

Keywords: model synthesis, procedural modeling

1 Introduction

The problem of automatically modeling complex shapes corresponding to architectural buildings, landscapes, and fractal structures is important for computer games, virtual environments and computer-generated movies. Current 3D CAD and modeling tools are limited in terms of generating complex models (e.g. urban scenes) and can be cumbersome to use. As a result, generating 3D digital content remains a major challenge.

Many procedural modeling techniques have been developed for automated or semi-automated generation of complex shapes. These include techniques based on shape grammars, scripting languages, L-systems, fractals, or solid texturing. In practice, they are either limited to a specific class of models or require considerable user input or guidance.

In this paper, we present a novel algorithm to generate complex shapes based on model synthesis. The notion of 3D model synthesis was initially proposed by Merrell [2007] and is inspired by recent developments in the texture synthesis literature.

A model synthesis algorithm accepts a simple 3D shape as an input such as Figure 1(a) and then outputs a larger more complex model that resembles the input such as Figure 1(b). We introduce the notion of continuous model synthesis, that is broadly applicable to a variety of different input shapes and objects. Our algorithm takes a simple, closed 3D polyhedral object as an input and outputs complex shapes that are similar to it. The main idea is to generate output shapes that maintain adjacency constraints between its boundary features (e.g. faces, edges, and vertices). We ensure that around every point of the output shape there is a local neighborhood that resembles a local neighborhood of the input model.

Our approach enumerates multiple configurations of each vertex, edge, and face and discards any configurations that do not satisfy the constraints. The runtime performance depends on the number of distinct normal directions of the input faces. Our algorithm is

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

simple to implement and we demonstrate its performance on many models. Overall, our approach offers the following benefits:

- **Simplicity**: Our system is simple to use and the only input is a closed 3D polyhedral consisting of dozens of triangles. The algorithm proceeds automatically without needing any additional guidance from the user.
- **Generality**: We can generate a wide variety of complex shapes by simply varying the input shape, including architectural buildings, landscapes, terrains and fractal shapes. These shapes may contain holes, arches, and non-axis aligned faces.
- **Efficiency**: Our algorithm generates complex shapes in only a few minutes.

Organization: The remainder of the paper is organized as follows. We briefly survey related work in Section 2, we describe our algorithm in Section 3, we highlight its performance in Section 4, and we analyze our approach and compare it to other techniques in Section 5.

2 Related Work

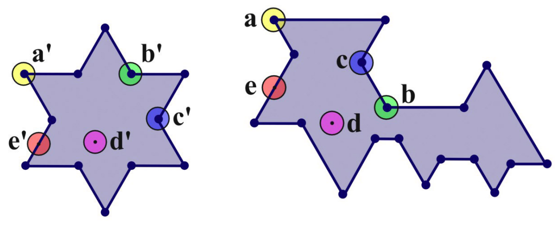
A wide variety of procedural modeling techniques have been designed to model specific types of objects or environments [Ebert et al. 2002]. L-system grammars have been used to generate plants [Měch and Prusinkiewicz 1996; Prusinkiewicz et al. 2001]. Terrain has been modeled using fractals [Musgrave et 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 or may require that the production rules for the model be specified within a grammar. In [Müller et al. 2007], images of facades are used to automatically derive shape grammar rules which are then used to model facades procedurally. Merrell [2007] introduced a model synthesis technique to automatically generate large models from a small example model provided by the user. However, this approach is restricted to inputs that fit on an axis aligned grid.

Our approach to model synthesis shares a common theme with texture synthesis. The underlying goal of both techniques is 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; Efros and Freeman 2001; Kwatra et al. 2003] and have also been applied to time-varying textures [Doretto et al. 2003; Kwatra et al. 2003] and to create 3D geometric texture on the surface of a mesh [Bhat et al. 2004]. Texture synthesis has also been used to create 3D solid textures from 2D texture exemplars of the texture expected on slices of the desired output [Kopf et al. 2007].

3 Model Synthesis

In this section, we describe our overall algorithm. We first present our approach for 2D shapes and then extend it to handle 3D models.

In Section 3.1, we present a technique that automatically identifies an adjacency constraint from the input model. This constraint guides our overall approach which is depicted in Figure 2. Starting with the input example shape shown in Figure 2(a), we create sets of lines parallel to the input edges as shown in Figure 2(c). These lines divide the plane into an arrangement of faces, edges, and vertices.



(a) Example Shape $E(\mathbf{x})$ (b) Generated Output Shape $M(\mathbf{x})$

Figure 3: *For each selected point* $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$, $\mathbf{d}$, *and* $\mathbf{e}$ *in* M, *a locally identical point* $\mathbf{a}'$, $\mathbf{b}'$, $\mathbf{c}'$, $\mathbf{d}'$, *and* $\mathbf{e}'$ *exists in the example model* E.

Each face, edge, and vertex is associated with a set of acceptable configurations or states that satisfy the constraint defined in Section 3.1. The set of states could be computed by dividing the input model along parallel lines as shown in Figure 2(b). We describe a more efficient method for computing the set of states in Section 3.2. Section 3.3 describes an incremental method that assigns states to each edge and each vertex and removes invalid states until an acceptable output shape is generated like the shape shown in Figure 2(d).

3.1 Adjacency Constraint

The user provides an example model as the input. We assume that the example model is a set of polygons that form closed polyhedral objects. Our algorithm generates a new model M that resembles the example model E. We consider the example model to be a function $E(\mathbf{x})$ of a point in space $\mathbf{x}$ where $E(\mathbf{x}) = 1$ if $\mathbf{x}$ is inside the polyhedra of E and $E(\mathbf{x}) = 0$ if $\mathbf{x}$ is not. The function $M(\mathbf{x})$ is similarly defined for the new model M.

We can find a small region of points near any given point $\mathbf{x}$ and each region constitutes one small part of the model. Every part of the new model M should resemble part of the example model, E. In order to accomplish this goal, we impose a simple and effective adjacency constraint on M. At a microscopic level, every local neighborhood found in M must also be found in E, but at a macroscopic level, M may contain interesting large-scale structures that are not found in E.

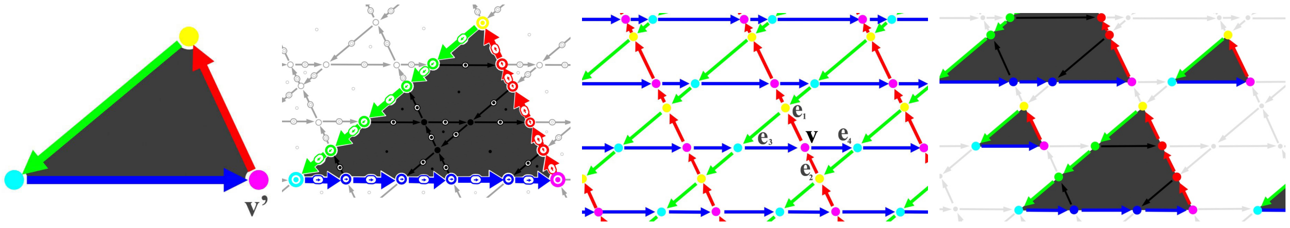
The adjacency constraint states that for every point $\mathbf{x}$ in M, there must exist a point $\mathbf{x}'$ in E that is identical to it locally. In Figure 3, the points $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$, $\mathbf{d}$, and $\mathbf{e}$ are locally identical to the points $\mathbf{a}'$, $\mathbf{b}'$, $\mathbf{c}'$, $\mathbf{d}'$, and $\mathbf{e}'$. More formally, the point $\mathbf{x}$ is *locally identical* to $\mathbf{x}'$ if for some small $\epsilon > 0$ and for all small vectors δ where $||\delta|| < \epsilon$,

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

We refer to this as a *continuous* contraint since it is defined on a continuum of points, not on a discrete grid.

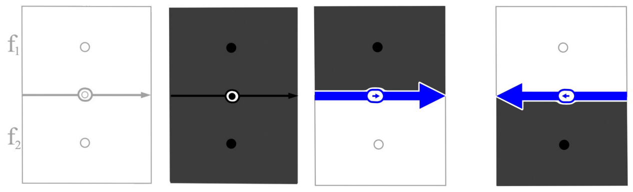
3.2 Finding Valid States (2D Case)

To satisfy the constraint, each edge in the new shape M must have the same slope as an edge in E. For each distinct slope, we create a set of parallel lines as shown in Figure 2(c). The lines divide the plane into small faces. These faces are the basic components of the new shape M. Each face f is assigned one of two possible states: 0 or 1. In state 0, all points within the face $\mathbf{x} \in f$ are part of the exterior, $M(\mathbf{x}) = 0$. In state 1, all points $\mathbf{x} \in f$ are part of the interior. If the faces are smaller, M is composed of more components and is more detailed. The size of the faces is



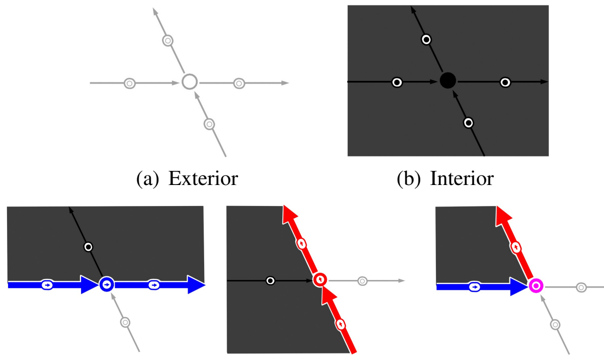
(a) Example Shape $E(\mathbf{x})$ (b) Parallel lines dividing $E(\mathbf{x})$ (c) Parallel lines diving up the plane. (d) Output Shape $M(\mathbf{x})$

Figure 2: *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 states in the output (d) can be found by dividing the input along parallel lines (b).*



(a) Exterior (b) Interior (c) Edge (d) Edge

Figure 4: *Possible states of an edge. Only (a-c) are found in the example shape (Figure 2(b)).*



(a) Exterior (b) Interior (c) Edge (d) Edge (e) Vertex

Figure 5: *The five possible states of vertex v in Figure 2(c). Only these states are found in the example shape (Figure 2(b)).*

determined by how far apart the lines are spaced. Generally, we choose to space all the lines identically. The lines could be spaced more closely in part of the model or for certain sets of parallel lines to create varying amounts of detail, but in general we choose not to do this in order to maintain a homogeneous appearance throughout the output model.

The faces of Figure 2(c) touch along the edges. One should not confuse the edges of Figure 2(c) with the edges of M. The edges of M are boundaries between its interior and exterior, but Figure 2(c)'s edges are simply sets of points which may or may not be on the boundaries. Figure 4 shows two faces that touch along a horizontal edge. Let f_1 be the upper face and f_2 be the lower face and let s_1 and s_2 be the states assigned to them. An *assignment* A is defined as a pairing between a state $s \in \{0, 1\}$ and a face f, $A = (f, s)$. The *state* of the edge s_e in between f_1 and f_2 is defined as the set of assignments made to f_1 and f_2, that is $s_e = \{(f_1, s_1), (f_2, s_2)\}$. The edge state $s_e = \{(f_1, 0), (f_2, 0)\}$ means that both faces are exterior (Figure 4(a)). In the cases where $s_1 = s_2$ (Figure 4(a) & 4(b)), the adjacency constraint is satisfied along the edge since the points on the edge are locally identical to the interior or exterior of E. Figure 4(d) shows an interior region below an exterior region along a horizontal edge. Such a configuration never appears in the example shape shown in Figure 2(b) and it violates the adjacency constraint. The state shown in Figure 4(c) is found in the example model and is allowed into M. In this example, the horizontal edge has three possible states. Edges in general may have three or four associated states.

We also must ensure that the adjacency constraint is satisfied at the vertices of Figure 2(c). Let us pick a magenta vertex and call it v. The vertex v lies at the intersection of two different edges, the horizontal blue edges and the upward red edges. Let e_1 and e_2 be its two adjacent upward edges and e_3 and e_4 be its two horizontal edges. Let s_1, s_2, s_3, and s_4 be the states assigned to e_1, e_2, e_3, and e_4 respectively. The state of the vertex s_v is defined as the set of assignments made to its four edges $s_v = \{(e_1, s_1), (e_2, s_2), (e_3, s_3), (e_4, s_4)\}$.

We need to find a list of states that satisfy the adjacency constraint at vertex v. If all neighboring edges have exterior states (Figure 5(a)) or if all have interior states (Figure 5(b)), then the adjacency constraint is satisfied at v. To identify other valid states, we find all edges of the example shape E that have the same slope as e_1 or e_3. In this case, we find two additional edge states shown in Figures 5(c) and 5(d). To identify more valid states, we search through the vertices of E. We only find one vertex v' in E whose two adjacent edges have the same slope as e_1 and e_3, which results in the state shown in Figure 5(e). The states of other vertices in Figure 2(c) can also be found by comparing the slopes of each vertex's adjacent edges with the edges and vertices of the input shape.

3.3 Assigning Consistent States (2D Case)

Algorithm 1 gives an overview of continuous model synthesis. We have constructed a list of several possible states to assign to each edge and each vertex (Step 2). Let us call this list of possible assignments $C(m)$. We gradually narrow down the list of assignments until each edge and vertex is assigned only one valid state. Let m be the list of current assignments. Eventually m will contain assignments to every edge and vertex and will determine the output model M.

Initially, no states are assigned ($m = \emptyset$) and $C(m)$ includes all possible states identified in Section 3.2. Every state that agrees with the adjacency constraint is equally valid. At each edge or vertex, we randomly select and assign a single state from among the choices found in $C(m)$.

Figure 6(a) depicts the initial configuration of $C(m)$. Initially, every face, edge, and vertex has all possible states present and then

Algorithm 1 Overview of Continuous Model Synthesis

```
1: Create planes parallel to $E$ to form solid regions, faces, edges,
   and vertices.
2: Find list of all acceptable states at each edge and vertex and
   store in $C(m)$.
3: while there exists an unassigned edge or vertex $v$ do
4:     Randomly select a state $s$ from $C(m)$.
5:     Assign $s$ to $v$. $m = m \cup \{(v, s)\}$.
6:     Remove from $C(m)$ all state that are incompatible with the
       assignment $(v, s)$.
7: end while
```

we assign states to several edges and vertices along the highlighted row. These assignments limit which adjacent states are possible. We remove edge and vertex assignments that disagree with their neighboring assignments. An edge assignment (e, s_e) agrees with an adjacent vertex state s_v only when $(e, s_e) \in s_v$, since vertex states are defined as sets of adjacent edge assignments. The result after the first removal is shown in Figure 6(b). This removal limits which states are possible in other edges and vertices, which compels us to remove more states. We continue to remove more and more states until there are none left to remove as shown in Figure 6(c). We continue to select from the remaining assignments in $C(m)$ and then update $C(m)$. This continues until every edge and vertex has been assigned a single state. The shape produced in Figure 6(c) achieves the desired result of a triangle that is similar to the input triangle in Figure 2(a) and that satisfies the adjacency constraint everywhere.

Unfortunately, for some input models, it is possible the algorithm may make an incorrect assignment that eventually 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 and so it must backtrack. If this occurs we use the strategy described in [Merrell 2007] which is to not try to fill up the whole modeling space at once, but to modify small parts of the space as shown in Figure 7. It is much less likely the algorithm will make an incorrect assignment if it is only modifying a small part of the space. Empirically, we have found that our algorithm almost always succeeds when modifying a volume of 10 x 10 x 10 or smaller (in units of plane spacings). Many input models such as those shown in Figures 1, 10 - 12 do not have this problem. With these inputs, the algorithm always succeeds even when modifying huge volumes. But even in the worst case, there is never any danger of an over-constrained input causing an immediate failure since solutions exist 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 parts to find it.

3.4 Generating Three-Dimensional Models

The three-dimensional problem of generating 3D models is quite similar to the 2D problem. To demonstrate the algorithm, we will use as our example model $E(\mathbf{x})$ the tetrahedron shown in Figure 8(a). For every distinct face normal, we create a set of parallel planes having the same normal. For the tetrahedron, we create four sets of parallel planes shown in Figure 8(b). These planes partition the space into solid regions. Each solid region has two possible states. It could have all interior points or all exterior points. Solid regions touch along faces, faces touch along edges, and edges touch at vertices.

Vertices occur wherever three planes intersect. Let us pick a vertex and call it v and try to determine what states are possible at v. In order to perform this step, we search the example model for all vertices whose three faces are parallel to the three planes at v. Suppose that we intentionally picked v so its three planes match the faces that touch the tetrahedron's top vertex. The vertex v has nine possible states as shown in Figure 9. It could be a vertex as shown in Figure 9(e). We could remove one of the faces, yielding the three edge states shown in Figure 9(d). We could remove two of the faces, yielding the three face states shown in Figure 9(c). We could remove all three faces, yielding a pure interior (Figure 9(b)) or a pure exterior state (Figure 9(a)). The method for finding states in 3D is described in more detail in Algorithm 2. A similar procedure can be to determine which states are allowed at the edges. Once we find the possible states of the vertices and edges, we can use the method described in Section 3.3 to assign a single state to each.

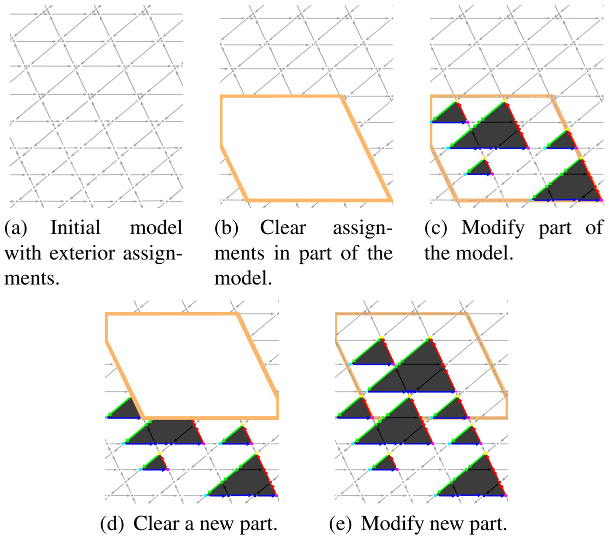

(a) Initial model with exterior assignments. (b) Clear assignments in part of the model. (c) Modify part of the model. (d) Clear a new part. (e) Modify new part.

Figure 7: *This 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.*

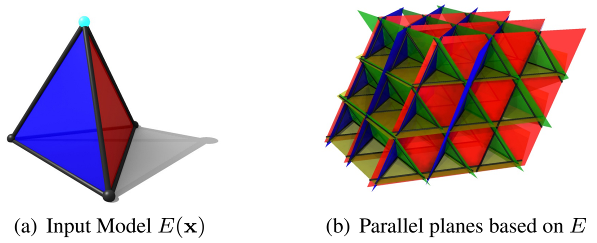
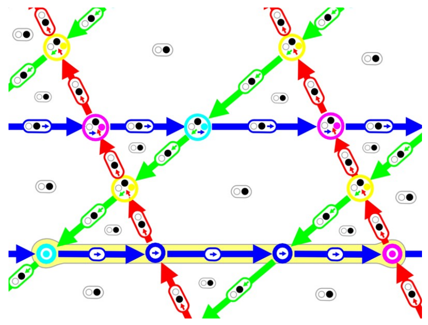
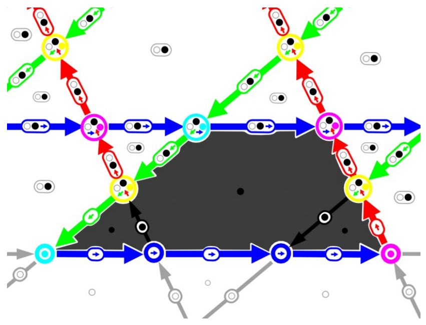

(a) Input Model $E(\mathbf{x})$ (b) Parallel planes based on E

Figure 8: *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 Results

Figures 1, 10 - 13 show a wide variety of different models generated using our algorithm. The generated models are large and detailed and it would be time-consuming to model them manually using a CAD or authoring system. The output model for each result was generated in less than two minutes as shown in Table 1. The total time spent by the human user is also short. The user only supplied


(a) We begin by allowing all possible states to be at every face, edge, and vertex. Then we assign a few states to the highlighted locations.


(b) Using Figures 4 and 5, we remove adjacent states that are incompatible with the previously assigned states.

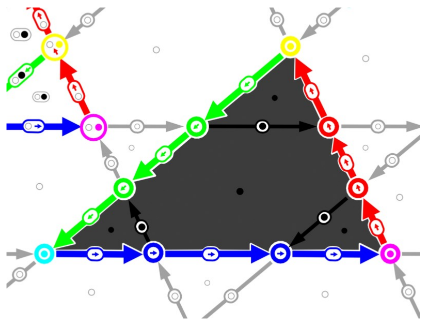
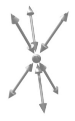
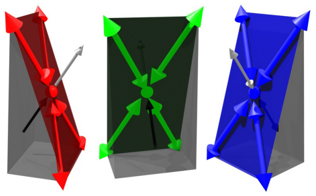
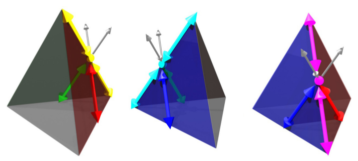

(c) After states are removed, their removal causes other states to be removed. The result after all incompatible states have been removed is a triangle.

Figure 6: *The evolution of the list of possible states $C(m)$ over time.*


(a) Exterior

(b) Interior


(c) Face States


(d) Edge States

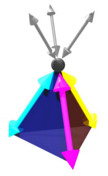

(e) Vertex

Figure 9: *The possible states of a 3D vertex found in the input model.*

	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

Table 1: *Complexity of the input and output models and computation time for various results.*

the size of the output and the example models which are quite simple. In order to demonstrate how simple the input models can be, we created them with only a few dozen polygons as shown in Table 1, but they could be much larger. They were manually created in a few minutes using 3D Studio Max. The exact same algorithm generated all the models shown in Figures 1, 10 - 13 without adjusting anything but the input model and the output size. The output models are not merely copies of the input model, but contain interesting new features not found in the input.

Algorithm 2 Method for finding all acceptable states of a 3D vertex which lies at the intersection of three planes with the normals $N = \{n_1, n_2, n_3\}$. The vertex states are defined as assignments to the vertex's six adjacent edges.

```
1: Find a point $\mathbf{p}$ on each face, edge, and vertex of the input whose faces are parallel to $N$.
2: **for all** points $\mathbf{p}$ **do**
3:     **for all** directions $\mathbf{d} \in \{n_1 \times n_2, -n_1 \times n_2, n_1 \times n_3, -n_1 \times n_3, n_2 \times n_3, -n_2 \times n_3\}$ **do**
4:         Determine the geometry at $\mathbf{p} + \epsilon\mathbf{d}$. There are (14) possibilities. If $\mathbf{p} + \epsilon\mathbf{d}$ is ...
5:             ...on a face, determine its normal direction (4).
6:             ...on a edge, determine the normal directions of its two faces and if the angle between them is a reflex angle (8).
7:             ...in the interior or exterior (2).
8:     **end for**
9: **end for**
```

5 Analysis and Comparison

Other methods are also able to efficiently produce large detailed models of buildings [Müller et al. 2006; Wonka et al. 2003], but 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 specific and adjust many production rules of the grammar. This addition user input has the advantage of giving the user greater control over some aspects of the result, but it requires more effort from the user. Müller et al. [2006] adapt architectural concepts to derive a set of specific shape rules for buildings. Our approach is rather complementary, where the user only needs to specify an input model.

In many ways, continuous model synthesis can be regarded as a major extension of discrete model synthesis techniques proposed by Merrell [2007]. The key difference is that in [Merrell 2007] the example models were manually partioned into small axis-aligned parts that fit on a 3D Cartesian grid. This manual partioning is time-consuming. Moreover, the previous approach does not work well on models that do not naturally fit on a grid such as Figures 10, 12, and 13. In these cases, the output of model synthesis is so tightly constrained that the previous method simply reproduces the input.

Limitations: Continuous model synthesis is more general and flexible because it can accept models that do not fit into tiles, but its generality and flexibility is still limited in several ways. While the

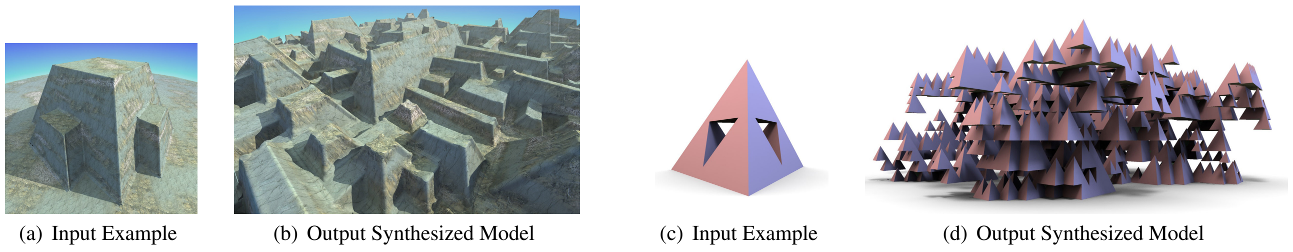

(a) Input Example (b) Output Synthesized Model (c) Input Example (d) Output Synthesized Model

Figure 10: *From the example model (a), rocky terrain is generated (b). The Sierpinski Tetrahedron (c) is used as an input to generate fractal structures (d).*

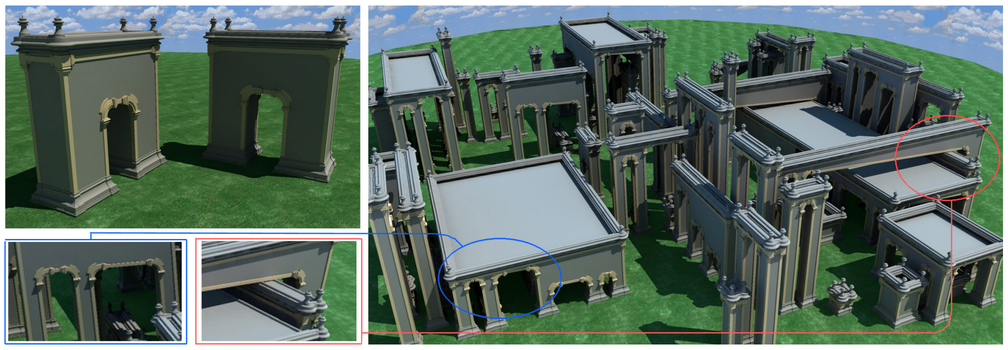


Figure 11: *From the input example model (left) many arches are synthesized (right). The output contains interesting new variations not found in the input such as structures with multiple arches and arches passing over arches (insets).*

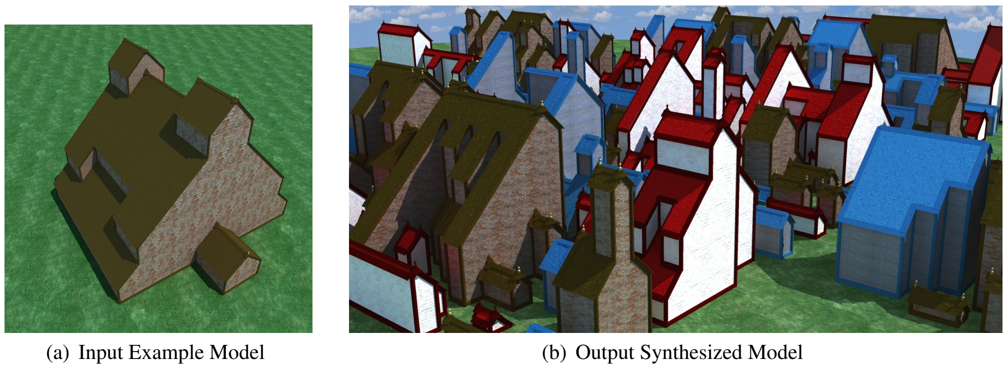

(a) Input Example Model (b) Output Synthesized Model

Figure 12: *From the input model (a), houses are automatically generated (b). Each house has a complex and unique roof structure.*

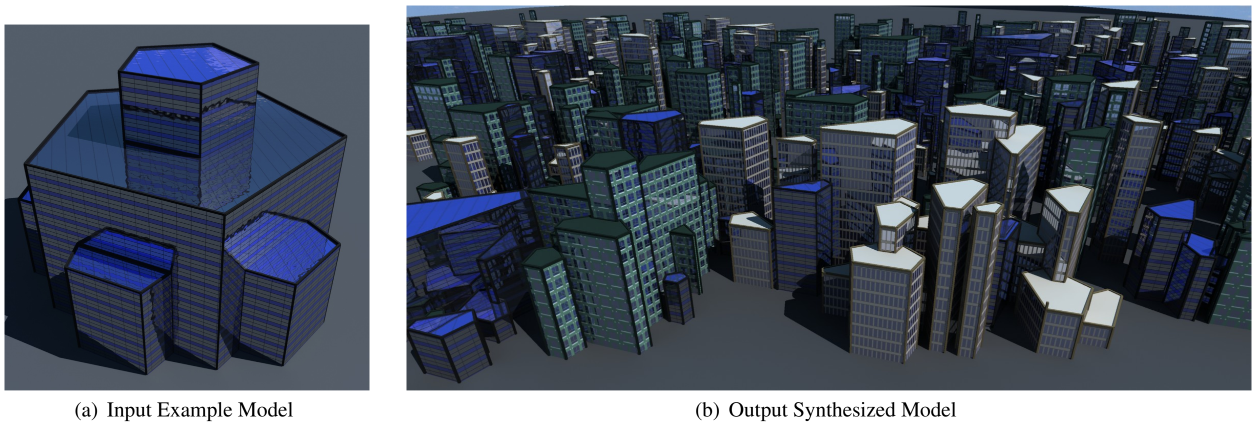

(a) Input Example Model (b) Output Synthesized Model

Figure 13: *From the input example model (a) pentagonal buildings are synthesized (b). Most of the faces are not aligned with the axes. Most buildings are complex combinations of many pentagonal shapes and have a unique shape.*

method works wells on input models that have a small number of distinct face normals, the time and memory requirements increase with the number of distinct normals. If m parallel planes are generated for each of n distinct normals, then there could be up to $O(n^3m^3)$ vertices. So it is impractical for processing curved or highly tessellated models.

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.

The algorithm described above is automatic and the user is unable to control many aspects of the output. Since the adjacency constraint only operates on a small scale, the large-scale structure of the output can only be indirectly influenced by the user and so the result may not meet the user's expectations and may look unnatural in some cases. The results could be improved by imposing additional constraints to control the size and distribution of the objects. One the the most important missing constraints is a way for the user to specify that an object must have a particular width or height since many objects used in architecture and other applications have fixed dimensions.

6 Conclusions and Future Work

We have presented a method for automatically modeling large complex shapes that resemble simple models provided by the user. The user can input a wide variety of different shapes and the algorithm outputs many valid and interesting procedurally generated shapes. The input model need not be axis aligned or fit on a grid and this gives the algorithm greater generality and flexibility. However, further work is needed to improve the quality of the generated shapes. First, many shapes such as pyramids have more than three faces intersecting at a vertex and these shapes are not handled properly. Second, we would like to allow the user to impose additional constraints. The adjacency constraint should be modified to constrain some objects to be a fixed discrete size. Such a method would combine the strengths of both a continuous and a discrete model synthesis technique and would allow the user to more effectively control the output.

Acknowledgements

We would like to thank the reviewers for their comments. This work was supported in part by ARO Contracts DAAD19-02-1-0390 and W911NF-04-1-0088, NSF awards 0400134, 0429583 and 0404088, DARPA/RDECOM Contract N61339-04-C-0043, and Intel.

References

BHAT, P., INGRAM, S., AND TURK, G. 2004. Geometric texture synthesis by example. In *SGP '04: Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, ACM Press, New York, NY, USA, 41–44.

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.

DORETTO, G., CHIUSO, A., SOATTO, S., AND WU, Y. 2003. Dynamic textures. *International Journal of Computer Vision 51*, 2 (February), 91–109.

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 FREEMAN, W. T. 2001. Image quilting for texture synthesis and transfer. *SIGGRAPH '01*, 341–346.

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

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.

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

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.

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.

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.