# Interactive Architectural Modeling with Procedural Extrusions

TOM KELLY
University of Glasgow
and
PETER WONKA
Arizona State University

We present an interactive procedural modeling system for the exterior of architectural models. Our modeling system is based on procedural extrusions of building footprints. The main novelty of our work is that we can model difficult architectural surfaces in a procedural framework, e.g. curved roofs, overhanging roofs, dormer windows, interior dormer windows, roof constructions with vertical walls, buttresses, chimneys, bay windows, columns, pilasters, and alcoves. We present a user interface to interactively specify procedural extrusions, a sweep plane algorithm to compute a two-manifold architectural surface, and applications to architectural modeling.

Categories and Subject Descriptors: I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modeling—*Modeling packages*

Additional Key Words and Phrases: procedural modeling, roof modeling, urban modeling

**ACM Reference Format:**

Kelly, T. and Wonka, P., Interactive Architectural Modeling with Procedural Extrusions. ACM Trans. Graph. VV, N, Article XXX, XX pages.
DOI = 10.1145/XXXXXXX.YYYYYYY

## 1. INTRODUCTION

The main motivation for our work is to develop an *interactive* and *procedural* modeling tool for *complex* architectural surfaces. We are interested in procedural and interactive modeling for three reasons. First, procedural descriptions allow edits to architectural surfaces at multiple levels and previous edits will adapt to subsequent ones. For example, the scene in Fig. 2 can be edited by reshaping the building footprints, and the model buildings, including the complete roof construction, will change according to the new input. Second, procedural modeling is the most efficient method to generate larger urban environments. Finally, we want to combine interactive and procedural modeling, because a frequent obstacle to using procedural tools is that it requires scripting. Eliminating scripting will enable more people to use procedural modeling tools.

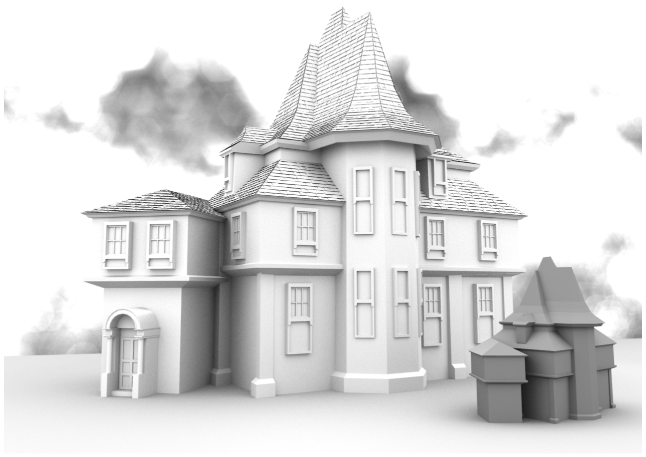


Fig. 1. Procedural extrusions allow a footprint (2d plan) to be extruded to form the walls and roof of a house (inset). Meshes and procedural details can then be attached (main).

Our goal is to model complex architectural features, including overhanging roofs, dormer windows, interior dormer windows, roof constructions with vertical walls, buttresses, chimneys, bay windows, columns, pilasters, and alcoves. See Fig. 1 for an example showing some of these features. These complex architectural surfaces have not been handled in procedural modeling before, and the main contribution of this paper is to introduce the first procedural modeling solution that includes these surfaces. Previous work in procedural modeling using shape grammars [Müller et al. 2006; Lipp et al. 2008] is able to model some architectural roof surfaces on a restricted set of footprints, but not the more complex roofs of arbitrary footprints shown in this paper.

The first part of our solution is to identify the most important edits and to design a user interface to specify procedural extrusions.

Authors' addresses: T. Kelly, Department Of Computing Science, Lilybank Gardens, Glasgow, UK; email: twakelly@gmail.com; P. Wonka, Arizona State University, Tempe, USA; email: pwonka@gmail.com. We would like to thank the reviewers and Michael Schwarz for many helpful suggestions for improving the various drafts of the paper. The research was supported by NSF, the FIT-IT program from FFG, and the Scottish Informatics and Computer Science Alliance.

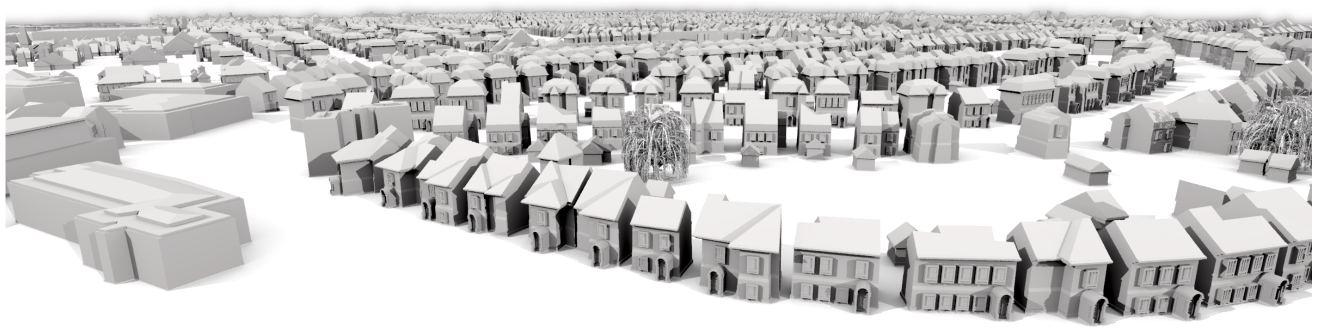


Fig. 2. We present an interactive procedural modeling system that is able to model difficult architectural surfaces, such as roof constructions. This figure shows procedural extrusions applied to 6000 floorplans from a GIS database of Atlanta.

We consider this part interesting because after analyzing examples, such as the one shown in Fig. 1, it is not clear how to model such a building, and what editing operations are even necessary to ensure that a larger class of interesting architecture can be modeled. An important aspect of our solution is to model buildings from floorplans and profile curves, see Fig. 5. In Sec. 3 we will describe our user interface in more detail including the architectural configurations that motivated the different user interface parts. The goal of our work is to have tools that are expressive enough to be able to quickly model most aspects of a building. We will evaluate our system on a catalog of 50 buildings in various styles in Sec. 6 to demonstrate the efficiency of our tools and to document geometric configurations that are difficult to reproduce.

The second part of our solution is a collection of algorithms to compute procedural extrusions from the user specification, see Sec. 4. We propose a sweep plane algorithm to grow the architectural surface upwards and to handle various events stemming from user edits or plane intersections. Our algorithms are inspired by the straight skeleton [Aichholzer et al. 1995]. We want to note that the computational geometry community emphasizes provably correct algorithms and therefore often favors rational arithmetic. In contrast, our work consists of heuristic algorithms that emphasize computation speed and are geared towards a floating point implementation. While our heuristics include various mechanisms to make the results more robust, it is possible that the computations can fail. For example, in the Atlanta data set of 6000 buildings we noted that two roof planes were not computed correctly. The approximate nature of our floating point computation also results in roof planes being moved by millimeters.

The contributions of our work are:

—the design of the system and the set of tool choices to enable procedural modeling of complex architectural surfaces.

—heuristic algorithms to generate a polygonal mesh from the user specification that is approximately consistent with the input data.

—the evaluation of the system on a collection of examples to verify its practical utility, and to identify configurations that are difficult to model with our tools.

## 2. RELATED WORK

In architecture, Stiny pioneered the idea of shape grammars [Stiny 1975]. In computer graphics grammars were used as a design tool

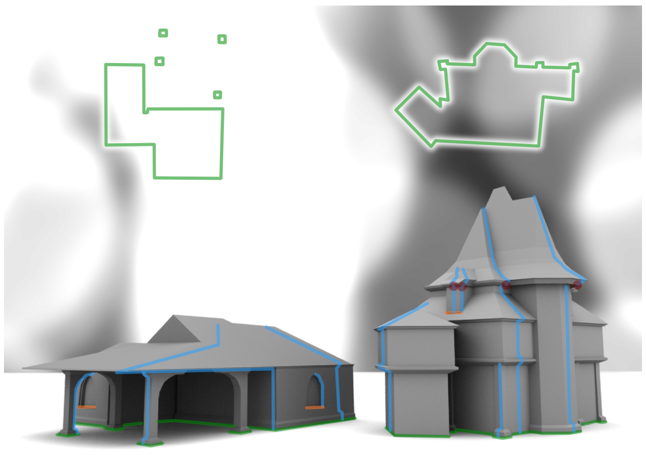


Fig. 3. These two examples show architectural surfaces overlayed with the user input. Plans (green), profiles (blue), natural steps (orange) and offset events (red) are specified in the user interface. The output of our system is an architectural shell (gray).

for architecture by Wonka et al. [2003], Müller et al. [2006], Aliaga et al. [2007], and Lipp et al. [2008]. L-systems [Prusinkiewicz and Lindenmayer 1991] were also proposed for procedural modeling of architecture [Marvie et al. 2005]. Merrel and Manocha [2008] propose a more general approach that can create new models from an example mesh. Given a man-made model as input, great results for reshaping were achieved by Cabral et al. [2009] and Gal et al. [2009]. The output of our procedural extrusions could also be further processed to distribute brick patterns [Legakis et al. 2001].

The straight skeleton was introduced by Aichholzer et al. [1995; 1996] and the authors commented how the straight skeleton computes a very plausible roof construction over a polygon. The idea of weights for the straight skeleton has been briefly mentioned by Eppstein and Erickson [1998], but the topic was only developed for a convex polygon decomposition [Aurenhammer 2008]. These papers are the inspiration for our work, and they aim to make a contribution to theory in computational geometry. In contrast, we focus on the application to modeling and extensions that are inspired by the demands of our modeling system. A starting point for

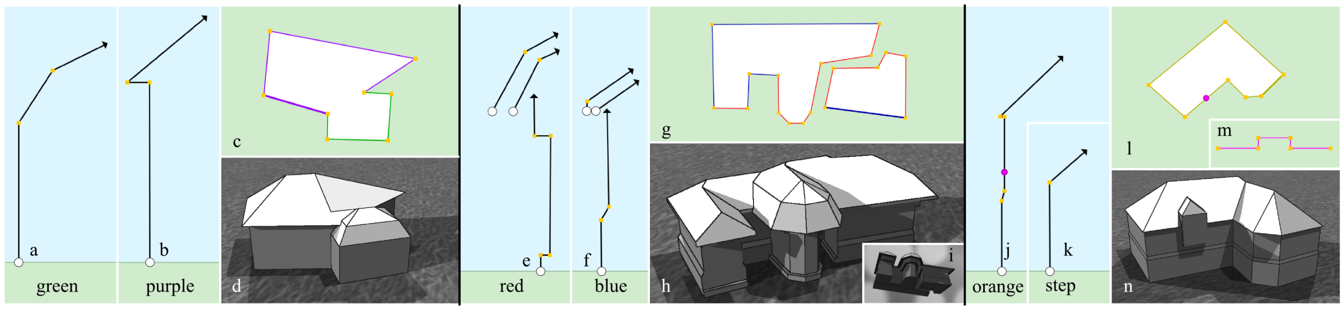


Fig. 4. Three example buildings constructed in our user interface. We demonstrate multiple profiles on a simple plan (abcd), modeling overhangs (efghi) and anchors (jklmn). Simple profiles (ab) are applied to the green and purple edges of the plan (c) to create the geometry (d). Note the horizontal profile section. Overhangs are defined using an additional pair of profile polylines associated with every edge (ef) to create typical roof geometry (hi). Anchors (magenta circles) are defined on the profile (j) and the plan (l) to position features. In this example the anchors position a rectangular natural step (m) with a profile (k) that creates a roof-window (n).

our implementation was the work by Felkel and Obdrzalek [1998] and Cacciola [2004].

Applications to architectural modeling of extrusion operations and the straight skeleton were demonstrated by several authors, e.g. [Laycock and Day 2003; Havemann 2005; Müller et al. 2006; Kelly 2006; Autodesk Inc. ]. Our goals are similar to these approaches and we contribute new extensions to the straight skeleton and an interactive procedural modeling system.

## 3. USER INTERFACE DESCRIPTION

Our interface controls a sequence of extrusions that are particularly suitable for creating the shell of architectural models. In this section we will introduce the functionality of our user interface.

### 3.1 Modeling With Profiles

The inspiration for our work comes from simple roofs, that are defined by a 2d polygonal floorplan and an angle that defines the roof slope. We extend this simple concept to construct a wide variety of roofs by using different angles on each edge and entire building shells by changing the angle as we ascend using *profiles*, Fig. 3.

To construct our geometry we use a sweep plane that rises vertically from the base of the building. This sweep plane defines an *active plan* that combines the changing profiles, and discrete modifications to create complex architecture.

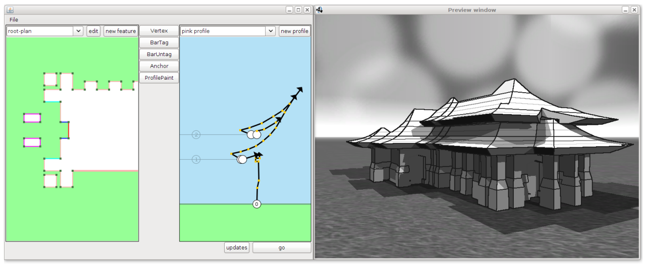


Fig. 5. The interactive interface during the design of a temple. The right window contains the output preview whilst the left window contains the plan and the profile editors.

### 3.2 Plans and Profiles

The UI consists of one pane showing the plan, one pane for profile, and a 3d preview window as shown in Fig. 5.

The plan is a set of edges (see Sec. 4.1 for the definition of a plan). For each edge in the plan, there is an associated collection of polyline segments, called a *profile*, that define the shape of a cross-section through the building at that plan edge. As the user edits either the plan or the profiles, our system shows the resulting architectural shell in the 3d preview window.

The user interface presents standard operations for inserting, deleting, and moving vertices in the profile polylines, and vertices (called corners) in the floor plan. In Fig. 4(abcd) we show an example with one polygon as the plan (c) and two profiles (a and b). The plan edges are color coded to show which of the two profiles is associated with each edge. Note that the profiles have to be monotonic in the vertical direction, but we allow horizontal polyline segments as special case. In the implementation, every change of direction in a profile will lead to an edge direction event (Sec. 4.5).

### 3.3 Overhangs

One important design choice we had to make is how to model overhangs. There are two possibilities: 1) Allow the user to draw arbitrary polylines as profiles that can go up or down in the vertical direction; 2) Force the user to explicitly model profiles as multiple polylines where each polyline must be monotonic in the vertical direction. After some experiments we decided that the second option makes it easier to synchronize overhangs across multiple profiles. We will explain the process of modeling overhangs using the second example in Fig. 4 (efhgi).

The user creates the input floor plan shown in (g). The edges in this floor plan are color coded as either red or blue. A red edge will be extruded according to the red profile (e) and the blue edges will be extruded according to the blue profile (f). The final geometric construction is shown in (h) and (i). The red profile as well as the blue profile each consist of three polylines. Each of these polylines is monotone in the vertical direction. Modeling overhangs is an explicit operation. The overhang is modeled by inserting two new polylines into both profiles at a certain height. In the user interface this is one atomic insertion operation. If the user clicks to add a new vertex overhang in one profile, then all profiles will obtain two new polylines at the same height. The user can edit the new polylines for each profile independently, only the starting height will remain synchronized. In the implementation, we will trigger a profile offset event (Sec. 4.6) at this height.

## 3.4 Anchors

Several editing operations require us to locate features on the manifold. These might include meshes, such as doors and windows, or discrete changes to the plan, such as chimneys. The features have to be placed so that they can still be located after subsequent edits. This is called the persistence problem in editing procedural models [Lipp et al. 2008], and we introduce *anchors* as a solution in our system.

The user can place anchors by selecting a location in the 3d view, or by selecting points on the input plan's edge and the corresponding profile polyline. In Fig. 4 (jklmn) the anchors are shown as magenta circles on a floor plan edge and a profile edge. We allow the user to select from two types of anchor on a plan edge — relative and absolute. A relative anchor's location is a fraction of its length on the active plan edge. If the edge is represented in the active plan at the specified height, the feature is instanced.

Absolute anchors are defined on an input plan edge, and define a plane perpendicular to this edge. The intersection of this plane and the corresponding edge in the active plan at the height specified by the profile anchor defines the instance location. Because an edge in the active plan may shrink as it ascends, absolute anchors may not be instanced if they lie outside the edge on the active plan. Because an active plan edge may grow, it is possible to position absolute anchors beyond the ends of the input plan edge.

## 3.5 Plan Edits

Discrete edits to the plan at a certain height are know as *plan edits*. These are located by anchors specified by the user, and may modify, create or delete edges in the active plan. In the example Fig. 4 (jklmn) a plan edit is introduced at the location of the anchor. The plan edit itself is a set of edges (m). These edges are extruded along the new profile (k). Again the user is offered several techniques with different advantages. *Forced steps* insert an arbitrarily set of edges into the plan, while *natural steps* offer a range of simple shapes that can be inserted. For reasons that are discussed later, forced steps are more powerful, but can lead to self intersections, while natural steps are guaranteed to create manifold geometry.

As shown in Fig. 6 we can use discrete plan edits to locate features such as roof windows, or chimneys. Additionally, by adding a rectangle exterior to the active plan, and applying the appropriate profiles, we can create buttresses, as in Fig. 23. If the input plan has several repeated elements, such as bay windows or buttresses, plan edits give a convenient tool for defining an *instance* once, whilst repeating it at a number of different anchored locations.

## 3.6 Positioning Procedural Details

Anchors can be used to mark the top-left and bottom-right elements in a grid of features, such as windows. Parameters can be set to control the width of the repetitions, and when combined with relative anchors, allow features to be distributed on resizable façades. Variations on this theme allow rows or columns of features to be located, for example a line of dormer windows on a roof. Anchors may also be used to specify the location of complex external features, such as windows and doors, described by arbitrary meshes.

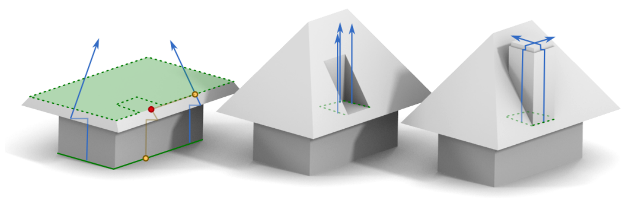


Fig. 6. Left: The plan (solid green line) and profiles (blue lines) define the shape of the structure. The anchors (orange) locate the chimney (red). A natural step is inserted into the building at the anchored location (dashed green lines). Middle: The finished 3d geometry, showing the profiles for the new edges. Right: Alternative natural step which adds an additional rectangle into the plan (dashed green lines) to specify a chimney.

Faces of the output model can be identified by adding *tags* to the profile segment. These are represented by small triangles in the user interface. Once the manifold is complete, the faces that were generated from the specified profile segment are post-processed in a particular way, for example to add tiles to the roof.

## 3.7 Modeling Larger Environments

We provide tools to model larger environments by example. We implemented several feature extraction algorithms to automatically label the edges of a building's plan. Example labels are length $\in \{short, medium, long\}$ and orientation $\in \{street, side, back\}$. The most important label is an angle computed by orienting the building to the street and mapping the normal vectors of the footprint edges to the unit disk. We assume that each footprint in the environment is labeled with a building type by another procedural algorithm. For each building type we can assign one or multiple profiles to each edge type including a probability value if more than one profile is assigned.

## 4. COMPUTING PROCEDURAL EXTRUSIONS

In this section we give an overview of the procedural extrusion algorithm. We begin by defining the terms used in the algorithm, the inputs and outputs, before outlining the many possible events that take place. Finally we give details for the computation of each event type.

## 4.1 Definitions

In this paper we compute an architectural shell in 3d Euclidean space with a $xyz$ world coordinate system. The up direction is along the $z$ axis. See Fig. 7 for an illustration of the terms.

A *(floor) plan* is a planar partition (a straight line planar embedding of a planar graph) that divides a plane into *inside* and *outside* regions. A plan has corners and edges. A plan is embedded in a plane parallel to the $xy$-plane (the ground plane), so that all corners of a plan have the same $z$ (height) value. We require that the boundaries of a plan are a non-intersecting collection of oriented polygons. The inside is on the left-hand side of each oriented polygon edge.

The polygons are typically oriented counter-clockwise, but polygons describing holes are oriented clockwise. Additional bounded regions may be recursively located inside a hole. The $j$th polygon is described by $n^j$ polygon corners $c_i^j \in R^3$ with $1 \leq i \leq n^j$. Each corner $c_i^j$ is connected to the next corner (according to the polygon orientation) by an *edge*, $e_i^j$. In everything that follows, indices should be treated cyclically, so that in a polygon with corners $c_1^j$, $c_2^j$, and $c_3^j$, the corner $c_4^j$ means $c_1^j$.

Each edge in a plan is associated with a *direction plane*, $dp_i^j$, which contains the edge. It is defined by an angle $\theta$ such that $-\pi/2 \leq \theta \leq \pi/2$. A vertical direction plane has $\theta = 0$, whilst a direction plane oriented towards the inside (outside) satisfies $\theta > 0$ ($\theta < 0$ respectively). The angle is measured between the direction plane and a vertical plane that also contains the edge.

A *profile* is a polyline that is used to control the direction plane of an edge. A profile is modeled in a local 2d $wz$-coordinate system and consists of a list of $m$ points $t_i$. The location of point $i$ is $(t_i.w, t_i.z)$ and we require that $t_i.z \leq t_{i+1}.z$. The profile defines $m-1$ angles, $\theta_1..\theta_{m-1}$. The angle $\theta_i$ is calculated as the clockwise angle between a vertical line and the line $t_i$ to $t_{i+1}$. The angle lies in the range $-\pi/2 \leq \theta_i \leq \pi/2$ and the final angle is constrained such that $\theta_{m-1} > 0$.

## 4.2 Overview

We describe the input, the output, and give an outline of the algorithm.

**Input:** The input of the algorithm is a (floor) plan, called the *input plan*, profiles associated with the edges of the input plan, profile offset events, and anchor events. Anchor events specify the location of plan edits or a mesh instance.

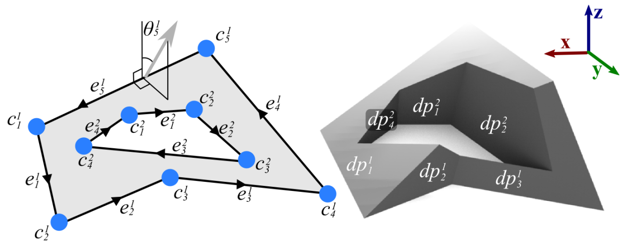


Fig. 7. Our algorithm constructs the architectural shell, shown on the right, for an input plan, shown on the left. In this simple example, each profile only has a single segment; Adding additional segments to the profile eventually allows us to model an entire building, including the walls. The input is defined by the corner positions $c_i^j$, the angles $\theta_i^j$, and the corner connectivity. The output is a shell consisting of faces on the respective direction planes, $dp_i^j$.

**Output:** The main output of the algorithm is an *architectural shell* (3d mesh) in the $xyz$ world coordinate system. In the non-degenerate case the shell is watertight and two-manifold. An architectural shell is a polygonal mesh stored in a half-edge data structure. For the sake of clarity we refer to these output edges as *arcs* (after Aichholzer et al. [1995]). The half-edge data structure stores a set of vertices in $R^3$, a set of arcs between the vertices, and a set of planar faces which may contain holes. Faces are defined by a counter-clockwise ordering of arcs.

The architectural shell can then be post-processed to apply textures, add procedural geometry (such as roof tiles), and attach meshes at anchor points.

```
main begin
    $Q$ = new priority queue;
    foreach corner $c_i^j \in inputPlan$ do
        foreach plan edge $e_i^j \in planDataStructure$ do
            $p1 = e_i^j.directionPlane$;
            $p2 = c_i^j.previousEdge.directionPlane$;
            $p3 = c_i^j.nextEdge.directionPlane$;
            $IE$ = intersect ($p1, p2, p3$);
            /* Queue ordered by z-height                    */
            $Q$.insert ($IE, IE_i.z$);
    /* Insert edge direction events, profile offset
       events and plan events into the queue            */
    foreach event $ue \in userEdits$ do
        $Q$.insert($ue, ue.z$);
    $sweepZ = 0$;
    while ! $Q$.empty() do
        $event = Q$.nextEvent();
        if $event.position.z \geq sweepZ$ then
            $sweepZ = event.position.z$;
            /* handleEvents may insert additional
               events into the queue                     */
            handleEvent($event$);
end
```

Fig. 8. Pseudo-code for the main loop.

**Outline:** The algorithm extrudes the input plan using a sweep plane algorithm. At each height of the sweep plane a 2d cross-section through the building is another 2d plan. We call the plan associated with the current sweep plane the *active plan*. To extrude a plan, each plan edge moves to be colinear with the intersection of the direction and sweep planes. This movement and the implicitly defined geometry is straightforward until an *event* occurs. During events, we process modifications to the active plan. After inserting edges into the active plan, we must recalculate the intersection events between the direction planes. The core algorithm, Fig. 8, is a loop that handles events according to their height.

**Data structures:** The *plan data structure* describes the implicit active plan on the sweep plane [Felkel and Obdrzalek 1998]. This structure is a doubly linked list of corners. Each corner has a pointer to the next corner and the previous corner (assuming counter-clockwise order) and a pointer to its previous and next edges, Fig. 10. At the beginning of the algorithm the plan data structure encodes the input plan. During the sweep the data structure is updated to encode any changes to the active plan. To give a concise description we define the algorithm by discussing changes to the implicit active plan.

The second important data structure is a priority queue that sorts events by ascending height. Intersection events are automatically created, while others (edge direction events, profile offset events and anchor events) are defined by the user. We fill the priority queue with a large number of potential intersection events. An intersection event occurs wherever three or more direction planes intersect.

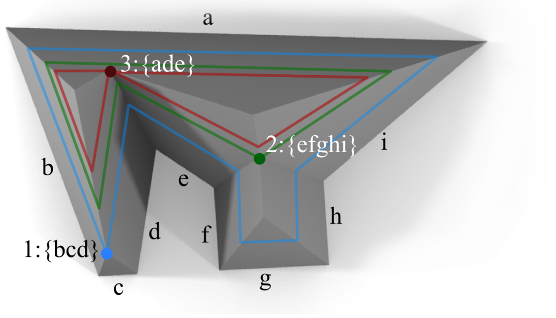


Fig. 9. An example construction demonstrating basic intersection events, and the active plan (blue, green and red lines) on the sweep plane after each event is processed. In (1) three adjacent direction planes collide at an *edge event*. In (2) we see a vertex event where more than three direction planes collide at one point. Finally, in (3) we show a *split event* that splits the area bounded by the plan.

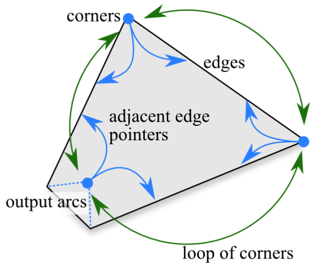


Fig. 10. The plan data structure, shown part way through the sweep.

Given the active plan at all event heights, the generation of the half-edge data structure describing the architectural shell and subsequent triangulation of shell-faces is fairly straightforward.

## 4.3 Description of Events

In this section we describe the events encountered as the sweep plane ascends.

**Generalized Intersection Event:** There are three event types, given by previous authors [Felkel and Obdrzalek 1998; Eppstein and Erickson 1998], which automatically occur to the edges in the active plan as the sweep plane rises. *Edge events* occur as the length of an edge shrinks to zero. When an edge shrinks to zero the direction planes defined by three consecutive (linked by corners) edges collide (Fig. 9, 1). *Split events* take place when two adjacent direction planes, and one non adjacent direction plane collide (Fig. 9, 3). These split the region bounded by the active plan into two parts. Finally *vertex events* occur in the degenerate case when more than three direction planes collide at one point (Fig. 9, 2).

Unfortunately, we did not find this categorization of events helpful to designing an algorithm. In practice architectural models give rise to a large number of degenerate events and the implementation is dominated by special event handling. Since edge and split events are special cases of a vertex event, we introduce one *general intersection event* that consists of an arbitrary number of direction planes, bounding one region, intersecting at one point. See

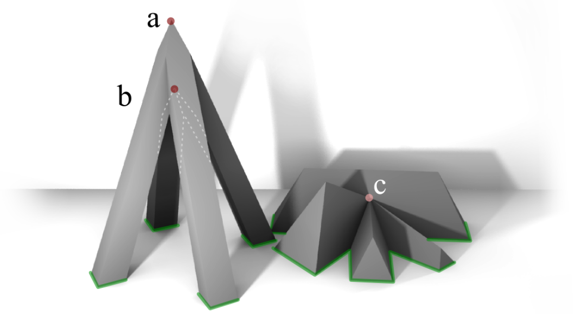


Fig. 11. Procedural extrusions that give rise to three degenerate cases. At a and b, four faces collide at one point. Point c shows seven faces colliding from a variety of angles, including horizontally.

Fig. 11 and 12 for four examples. We introduce a new algorithm to resolve this generalized intersection event that uses *chains* of edges involved in the intersection.

**Edge Direction Events:** An edge direction event occurs when a profile curve changes direction. The event updates the angle and direction plane associated with a set of edges in the active plan.

**Profile Offset Events:** Profile offset events occur at heights specified by user edits. Intuitively, a profile offset event results in additional inside regions being added to the active plan at the specified height.

**Anchor Events:** Anchor events specify locations on the architectural shell, and are defined by the user. There are two types of anchor events. *Plan Edit Anchors:* These modify the active plan to insert new features such as chimneys, or dormer windows. *Mesh Anchors:* These store the location of the anchors as an attachment point for geometry.

## 4.4 Generalized Intersection Event

Generalized intersection events perform topological changes on the active plan to ensure that it never self-intersects as the sweep plane ascends. These events are automatic, not user driven.

There are many possible topologies that can give rise to a generalized intersection event. Previous authors have described how to adjust the active plan to deal with split and edge events. These are the most frequent events when the input is a random polygon. We describe a generalization of these techniques to deal with the most likely class of topologies when dealing with architecture, a *locally connected region*. When our interface is used to model architecture, these account for the vast majority of events. A locally connected region is a region, that immediately before the event is locally equivalent to a topological disc, Fig. 11 (abc). In a single event the locally connected region may be either an "inside" or "outside".

In addition to locally connected regions, there are several unlikely classes of increasingly degenerate events in which the intersecting edges define a nested boundary, Fig 15. When this situation occurs we give a warning in the user interface that the output may be undesirable.

**Event Detection:** We use expanded bounds for intersection location clustering. This addresses two stability problems.

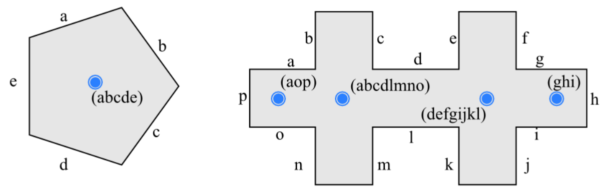


Fig. 12. Left: Five faces creating an intersection event. Right: Events can interfere with each other if they have the same height, in this case the four points share a roof ridge.

First, in symmetrical inputs, like architectural plans, it is very common for more than three direction planes to meet at a point. To avoid degenerate output in a floating point situation it is necessary to identify intersections whose locations are close together, and treat these as a single event. See Fig. 12 (left) for an example.

Second, direction plane intersections that are far apart from each other can interfere if they are close to one other in height, Fig. 12 (right). It is necessary to detect and handle these together to ensure the region bounded on the sweep plane does not self-intersect and to resolve the ambiguities that can occur (described in Sec. 4.8).

**Event Clustering:** After initialization we iteratively process (potential) events stored in the priority queue. To address the two previously mentioned event detection problems, we cluster the events in two directions. We poll the priority queue to collect all intersection events whose height, $z$, is within some threshold, $\delta_1$, of the initial event. Second, we cluster all the events according to their location after projection onto the $xy$ sweep plane. The clustered volume is therefore a cylinder of radius $\delta_2$ and height $\delta_1$. See Fig. 13 for an illustration of the clustering step. For our building floorplans with a size in meters we use double precision floating point representations and values $\delta_1 = 10^{-4}, \delta_2 = 10^{-6}$, found through trial and error on our large procedural floorplan set. There are certain pathological inputs which cause this clustering stage to fail. An example would be a row of events, each within $\delta_1$ of another, which could contain an arbitrary number of events. In such a case we alert the user with a warning message, but none of the users reported such a situation.

**Input:** The input of a generalized intersection event is a point $l \in R^3$, and a set of three or more active plan edges, $f$, whose associated direction planes intersect at $l$. The point $l$ is calculated as the center of the clustered volume.

**Output:** The output of a generalized intersection event is an updated active plan. This represents the bounded region on the sweep plane after the event.

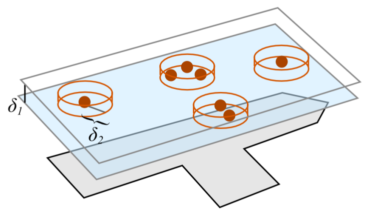


Fig. 13. When an event is processed we simultaneously extract all intersection events within a height of $\delta_1$. Then we cluster all events that are within a cylinder of radius $\delta_2$ and height $\delta_1$.

**Chain construction:** We process the edges involved in the clustered intersection events into a set of *chains*. A chain defines a connected portion of the active plan boundary involved in the event, Fig 14 (a). A chain, $h^i$, is a list of consecutive active plan edges, $\epsilon^i_1 ... \epsilon^i_{hmax_i}$. A cyclic *chain list*, $b$, contains all such chains, $h^1 ... h^{bmax}$ (we assume a cyclic index). The list is ordered by the edge's orientation around $l$.

The list of chains, $b$, is now processed to update the active plan in two stages. First within each chain (intra-chain), and then between the chains themselves (inter-chain).

**Intra-chain handling:** In a chain of 2 or more edges, the interior edges shrink to length 0 as we approach the intersection event, Fig 14 ($\epsilon^2_2$). Therefore in the intra-chain stage we remove all interior edges from a chain $h^i$, leaving only the start, $\epsilon^i_1$, and the end, $\epsilon^i_{hmax_i}$, of the chain as shown in Fig. 14 (cd). That is, if $hmax_i \geq 3$, then edges $\epsilon^i_2 .. \epsilon^i_{hmax_i - 1}$ are removed from the active plan, being replaced by a new corner at $l$, connecting the end of $\epsilon^i_1$ to the start of $\epsilon^i_{hmax_i}$.

**Inter-chain handling:** In a typical intersection event, the closest edges in adjacent chains move into each other. To allow this without self-intersections the inter-chain stage takes place between each adjacent pair of chains, $h_x$ and $h_{x+1}$ in the cyclic chain list $b$. Firstly, if any chains contain only one edge, we split that edge by inserting a corner at $l$, Fig. 14 (de). Secondly, for each pair of adjacent chains we create a new corner at $l$ and connect the start of the last edge in the proceeding chain, $\epsilon^x_{hmax_x}$, and the end of the first edge in the following chain, $\epsilon^{x+1}_1$, Fig. 14 (e). Finally the inter-chain stage finishes by removing any unreferenced corners from the active plan.

In addition to this basic technique, there a several implementation issues that we address — the filtering of invalid events, checking for chain intersections and local non connected events.

**Filtering invalid events:** Before the clustering stage we remove any invalid edges from the edge set, $f$. Because the intersections are detected using unbounded direction planes, there may be edges in $f$ that do not approach $l$ on the active plan. Such edges are removed from $f$. The line defined by the intersection of the direction plane and the sweep plane may pass close to $l$, however the line-segment defined by the associated active plan edge may not. A small epsilon range, $\delta_3$, expands the length of the edge and ensures that collisions occur reliably. On our inputs we find $\delta_3 = 10^{-5}$ a sufficient margin.

Second, an edge may have been removed from the active plan by a previous event. These edges are also removed from $f$. After filtering, if the number of edges in $f$ is less than 3, the event is ignored.

**Post inter-chain intersections:** There are rare situations where the chains after the inter-chain stage no longer form a valid plan on the sweep plane. To test if the chains form a valid plan we predict the chain locations on a plane higher than the active plan. If any of the chains intersect each other we use an application of the winding rule to calculate valid region boundaries for the current active plan. This may re-orient some edges, as well as insert new edges or corners into the active plan.

**Local non connected events:** The above handling of locally connected events is sufficient to create large cityscapes, Fig. 2. There are however, a class of degenerate topologies that can occur on the active plan, which are not connected, such as in Fig. 15. While we do not present a solution for each of these classes, the following observations are made.

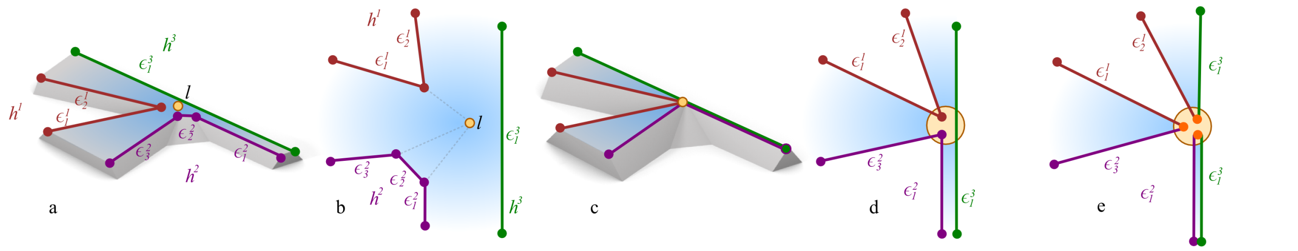


Fig. 14. Active plan modification during a generalized intersection event. a) The active plan just before the intersection of chains $h^1$ (red), $h^2$ (purple), and $h^3$ (green) at the point $l$. The chains consist of edges, $\epsilon$. b) The topology just before the event. c) The active plan geometry at the event, note the disappearing region bounded by coincident edges $\epsilon_1^2$ and $\epsilon_1^3$. d) The topology of the chains after the intra-chain stage. An edge, $\epsilon_2^2$, has been removed. e) The topology of the active plan after the inter-chain stage. The edge in a chain of length one, $\epsilon_1^3$, has been divided at $l$. After the edges are linked at $l$, the active plan has been split into three regions. The three new corners (e, orange) are at the same location, $l$, but they are expected to move in different directions over the course of the extrusion process.

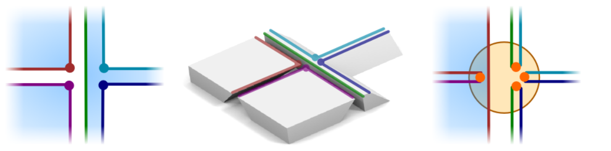


Fig. 15. A complex, non-connected region, causing an intersection event. Left: The active plan just prior to the intersection event between all the 9 edges. Middle: The output shell at this height, showing the edge angles and the colliding edges. Right: One of several non-symmetrical solutions that removes all but one edge.

If the edges in a chain form a closed loop, the chain may be simply removed. If there is more than one chain of length one, the associated chain edges must be parallel and the geometry between such adjacent chains may also be removed. Finally, if a chain is nested inside another chain there are situations where inter-chain updates no longer work. Here we just note that reversing a section of the enclosing chain is enough to keep the area enclosed on the active plane well formed. We note that we could not find an example where such degenerate events were part of a meaningful architectural construction.

## 4.5 Edge Direction Events

A set of edge direction events are created for each profile. An edge direction event updates the angle and direction planes of a set of edges. There are two types of edge direction event, *standard* and *near horizontal*. Standard edge direction events are constructed from a single angle in the plan, while a near horizontal edge direction event is constructed from two consecutive angles and a distance. These values are calculated from the profile polyline.

4.5.1 *Standard Edge Direction Events.* **Input:** A set of edges, $f$, in the active plan, each associated with the same profile and a single new angle for all the edges, $\gamma$. **Output:** A new active plan which replaces the original.

For each of these edges $e_i^j \in f$, we update the associated direction plane by setting its angle to $\gamma$. The edge, $e_i^j$, continues to propagate over the sweep plane as defined by the new angle.

4.5.2 *Near Horizontal Edge Direction Events.* We need a separate approach as the angle associated with an edge, $\theta$, approaches $\pm\pi/2$, as two parallel (horizontal) direction planes do not intersect to form a line. Additionally, as the angle approaches these limits we are colliding near coplanar planes, causing numerical instability. As Fig. 16 illustrates, we first increase the angles for numerical robustness, recursively apply procedural extrusions, and then project onto the sweep plane. This produces the required horizontal surface.

**Input:** A set of edges in the active plan, $f$, associated with the profile, a distance, $d$, a direction angle, $\gamma$, and a following angle, $\zeta$. The angle $\gamma \approx \pi/2$ ($\gamma \approx -\pi/2$) specifies the direction of the horizontal as towards the inside (respectively outside) of the active plan. $\zeta$ specifies the angle of the following non-horizontal edge event. **Output:** A new active plan which replaces the original.

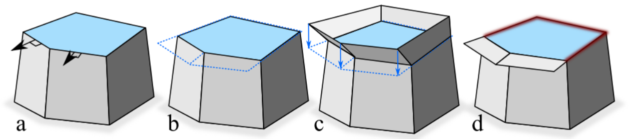


Fig. 16. The horizontal section desired (b) can be created by an additional application of procedural extrusions to calculate the offset in the given direction. After flattening (c) unchanged edges (red, d) are ignored.

First we create a temporary plan as a copy of the active plan. For each edge in the original plan, $e_i^j$, and associated angle $\theta_i^j$, the temporary plan has an edge $E_i^j$, and associated angle $\Theta_i^j$. Secondly we update the angles in the temporary plan according to the following mapping:

$$\Theta_i^j = \begin{cases} \tan^{-1}(d) & \text{if } e_i^j \in f \text{ and } \gamma > 0 \\ -\tan^{-1}(d) & \text{if } e_i^j \in f \text{ and } \gamma < 0 \\ 0 & \text{otherwise} \end{cases}$$

A recursive application of procedural extrusions extrudes the temporary plan for a height of one unit. The temporary active plan is projected onto, and replaces, the active plan in the original procedural extrusion instance. That is, $e_i^j$ is replaced by $E_i^j$ if it exists in the updated plan. If it does not exist in the updated plan $e_i^j$ is removed from the active plan. The location of $E_i^j$ is projected onto the original active plan. Finally the values of $\theta$ in the original skeleton are updated using the mapping:

$$\theta_i^j = \begin{cases} \zeta & \text{if } e_i^j \in f \\ \theta_i^j & \text{otherwise} \end{cases}$$

Occasionally multiple edge direction events occur at the same height. In this situation the direction events are sequenced by the order of user creation. The user can manually override this priority.

## 4.6 Profile Offset Events

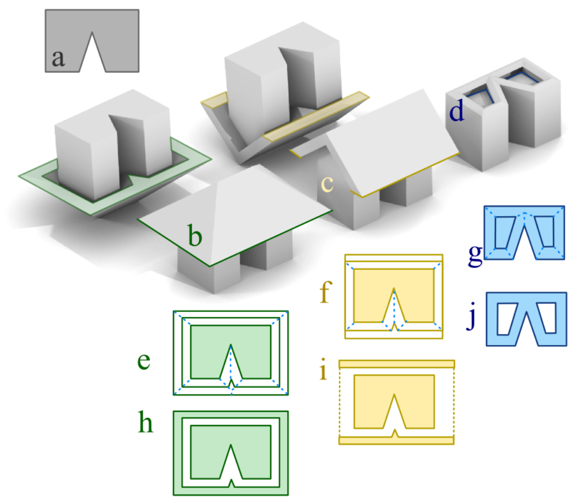


Fig. 17. Some meshes that can be computed from an input plan (a) using profile offset events. Buildings b and c are shown in two orientations. By creating two offset boundaries (e) that define an offset region (h), an overhanging roof (b) can be generated from an arbitrary plan (a). If two edges are disabled in the profile offset event, open-ended roofs can be created (c,f,i). Finally, by offsetting inside the active plan, walled roofs can be created (d,g,j).

Profile offset events specify the start of overhangs. The difficulty of specifying and handling profile offset events comes from the procedural definition. While it is easy to specify overhangs for a given region, the geometry must adjust itself according to subsequent user edits. Our technique must procedurally perform changes to the active plan without creating awkward self-intersections.

At a profile offset event an additional inside region, called an *offset region*, is inserted into the active plan (see Fig. 17). Two offset boundaries are grown from the active plan to enclose the new offset region. We introduce new edges and corners into the active plan to represent this newly enclosed region on the sweep plane. The new edges are classified as inside, outside, or side, depending if the edge stems from the first boundary, the second boundary, or an intersection operation between the two boundaries described later in this subsection.

**Input:** A map for each edge in the active plan, $e_i^j$ to a tuple, $t_i^j = \{disabled_i^j, dist\_inside_i^j, dist\_outside_i^j, profile\_inside_i^j, profile\_outside_i^j\}$ and a single $profile\_side$. The variable $disabled_i^j$ is a boolean value that specifies if the offset region associated with this edge is present in the output; $dist\_inside_i^j$ and $dist\_outside_i^j$ are real values that define distance and direction from the active plan of the inside and outside offset boundaries; $profile\_inside_i^j$, $profile\_outside_i^j$ and $profile\_side$ are profiles. We require that all values of $dist\_inside_i^j$ and $dist\_outside_i^j$ have the same sign; a positive (negative) sign indicates an offset (respectively inset) of the active plan. To ensure proper topology on the active plan, the distance, $dist\_inside$, is constrained to be non-zero. **Output:** The output of an offset event is an updated active plan, typically with the additional region defined either inside or outside of the input active plan.

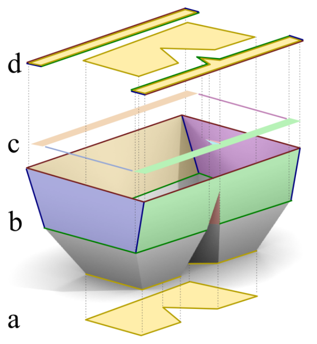


Fig. 18. The recursive application of procedural extrusions (b) to a plan (a) from Fig. 17 (c). The faces between $z = 1$ and $z = 2$ are projected onto the primary active plan (c), before being merged (d). Zero area faces (blue and purple) are removed, and profiles assigned based on the origin of the edge. In (d) green edges are assigned $profile\_inside$, red $profile\_outside$ and blue $profile\_side$.

We create a temporary plan as a copy of the primary (input) active plan. For each edge in the primary plan, $e_i^j$, the temporary plan has an edge $E_i^j$, and an associated profile, $profile\_recursive_i^j$. Edge $E_i^j$ is constructed by projecting $e_i^j$ onto the plane $z = 0$. The profile $profile\_recursive_i^j$ defines the angles $\Theta_i^j = tan^{-1}(dist\_inside_i^j)$ at $z = 0$, and $\Theta_i^j = tan^{-1}(dist\_outside_i^j)$ at $z = 1$. We execute a recursive application of procedural extrusions using the temporary plan as input. It is executed from height 0 to 2, to create a temporary output shell. Faces of the shell between the planes $z = 1$ and $z = 2$ are projected onto the primary active plan, forming the offset region, Fig. 18.

The projection associates each tuple, $t_i^j$, with an offset region in the primary active plan. The entire offset region is bounded by the projected edges, $r$. Additionally the projection defines a 1:1 mapping between the new edges, $e_l^k \in r$, and a subset of the temporary shell's arcs $A_l^k$. We remove from the primary active plan any edges in $r$ that enclose an offset region of area 0 or that are associated with a tuple containing a value of $disabled_i^j = true$. We update the profile, $profile_i^j$, associated with each edge, $e_i^j$, in the primary active plan according to the function:

$$profile_i^j = \begin{cases} profile_i^j & \text{if } e_i^j \notin r \\ profile\_inside_i^j & \text{if } A_i^j \text{ lies in the plane } z = 1 \\ profile\_outside_i^j & \text{if } A_i^j \text{ lies in the plane } z = 2 \\ profile\_side & \text{otherwise} \end{cases}$$

Finally we merge adjacent parts of the offset region to avoid self-intersections. We remove the corresponding edges and corners from the active plan.

## 4.7 Insertions into the polygon

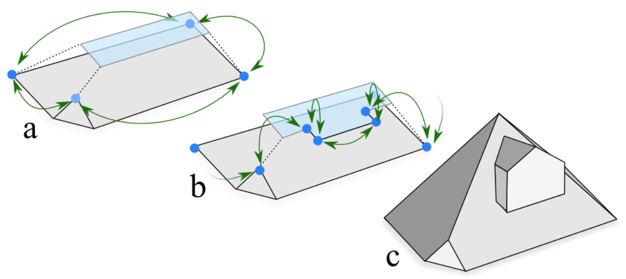


Fig. 19. Inserting a plan edit into the active plan during execution. a) The plan data structure (blue dots, green arrows) implicitly defines the active plan (cyan). b) To insert new edges into the active plan, corresponding edges are linked into the plan data structure. c) The resulting architectural shell.

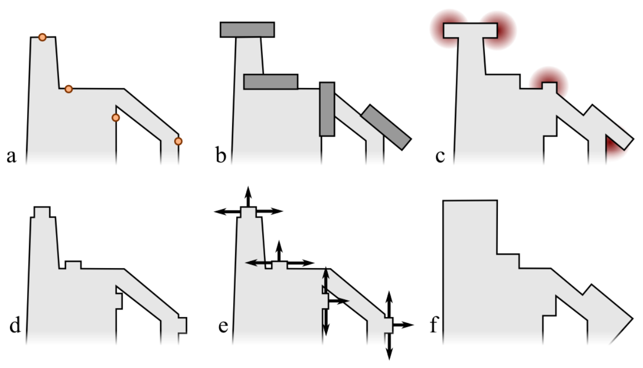


Fig. 20. Given an intricate plan, calculating a robust perturbation is challenging. Forced steps are positioned at the location of the anchors (a, orange). These are combined with the boundary using a geometrical union operation. However many geometry artifacts are undesirable (c, red) in an architectural situation. Given natural steps at certain positions (a, orange), small changes to the boundary are made (d), which are then grown (e) using a recursive application of procedural extrusions, to create more natural geometry (f).

Plan edits introduce discrete changes to the active plan at specified heights. We describe how plan edits operate efficiently and detail two methods to define them.

When performing a plan edit, some edges are deleted, some edges are moved, and some edges are inserted, Fig. 19. These new edges are at the height of the current sweep plane.

Our user interface offers two types of plan edits. Inserting an arbitrary shape gives the largest variety of geometric designs. However these *forced steps* offer no guarantees that the resulting active plan will not self intersect and create an invalid topology. The challenge comes again from the procedural nature of our approach and the fact that the edit has to work for all input plans. *Natural steps* offer a solution to this problem by using a recursive application of procedural extrusions to insert edges into the active plan.

Natural steps are calculated on the active plan at a given height by amending a small (typically $10^{-3}$ by $10^{-3}$) protrusion. This is offset by a recursive application of procedural extrusions such that it does self intersect, Fig. 20. This is similar to the edge direction events of Sec. 4.5. This application of procedural extrusions is constructed by assigning $\theta = 0$ to all edges not part of the feature, and a user defined $\theta$ to those edges in the protrusion. The resulting temporary active plan is calculated at a specific height, and this is incorporated into the original active plan. The new edges in the active plan have the relevant profiles assigned to them.

## 4.8 Ambiguities in Procedural Extrusions

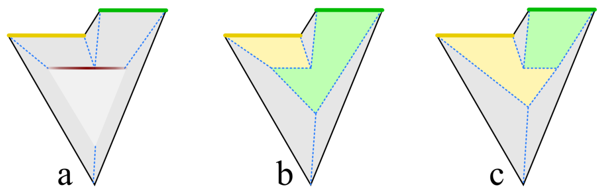


Fig. 21. An ambiguous situation that arises in the case of a concave input plan, a. The intersection of the yellow and green edge's direction planes gives two possible output arc directions (a, red). This may be resolved into two ways, b,c.

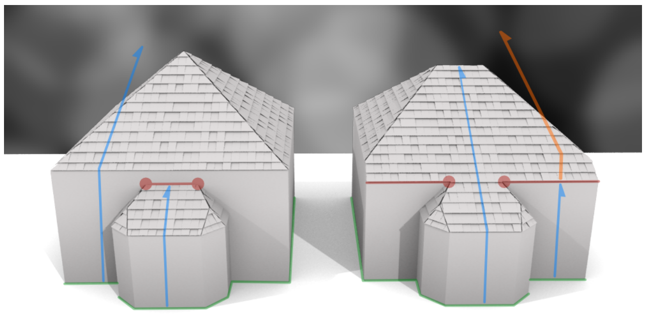


Fig. 22. Two identical bay windows that lead to the same two events (red circles) involved in an ambiguous situation (red line). To resolve the ambiguous situation, a single edge must be chosen to replace the others. The building on the left (right) resolves the ambiguity using the volume maximizing (respectively minimizing) priority technique. The resulting unused section of the original profile is shown in orange. Note that in each case, two ambiguous events occur at the same height, and must create globally consistent output.

We show that procedural extrusions (as well as the weighted straight skeleton [Eppstein and Erickson 1998]) are ambiguous in the concave case. Different modeling choices lead to different ambiguous-case resolution strategies, Fig. 22.

The ambiguous case may arise when two (or more) neighboring edges in the active plan become colinear on the same side of a region. This happens, for example, when edges previously separating the colinear edges are eliminated. We may also arrive in this situation if the input, or any of the plan edits, introduce colinear edges.

If the two neighboring and colinear edges bound different sides of a region the output is an arc representing a ridge and the computation proceeds as normal, assured that at the opposite end of the ridge the two edges will collide again [Felkel and Obdrzalek 1998]. This is not an ambiguous event and we can distinguish the regular roof ridge case from an ambiguous event by making use of the fact that we use oriented edges to describe a plan. When the edges have the same orientation (they bound the same side of a region) we are not able to determine the direction of the output arc, Fig. 21. This produces an ambiguity.

The individual ambiguous events need to be solved consistently from a global perspective, Fig. 21; This is one reason for the vertical clustering outlined earlier. All colinear consecutive edges involved in an intersection on the active plan are grouped together. We resolve the situation by merging all consecutive edges into one and applying the profile of the edge that has the highest priority. Then we remove the other edges involved in the ambiguous events. To select the one edge we assign a priority ordering over the edges and choose the edge with the highest priority.

We introduce three possible priority schemes. It is interesting to note that most architectural roof structures (such as bay or dormer windows) enclose the maximum volume in the ambiguous case. This leads to our default scheme in which the highest priority edge, $e_i^j$, has the lowest (closest to $-\pi/2$) associated angle, $\theta$. Alternately the minimum case (largest associated $\theta$) may be useful when estimating conservative offsets. The third option is to manually define the priority function in the user interface. Section 6.2 describes situations where it is desirable for the user to manually define the priority function.

## 5. RESULTS

### 5.1 Modeling Results

In this section we show several interesting applications of modeling with procedural extrusions.

Fig. 23 shows many typical architectural shells that are not possible using just the straight skeleton or other existing procedural modeling tools. We can also create buildings with horizontal roof overhangs, such as Fig. 24. The alcoves and columns show how disconnected regions can merge together and interact. This is only possible because we allow negative angles for the roof planes.

Procedural extrusions may be used on a large scale to describe cityscapes. We created a procedural model using about 6000 footprints from Atlanta (see Fig. 2). The current model has three million polygons, 5 different building styles, took 20 minutes modeling time, 10 minutes to compute the procedural extrusions, and 15 minutes to render. Our current limitation is that we were not able to find a rendering infrastructure to render a few hundred million polygons of a detailed model. We therefore had to omit ornaments and some details of the roof constructions from the designs.

We implemented the proposed system in Java and measured the running time of our system on 64bit 2.6GHz Xeon.

We created a procedural model for town homes adjacent to a curved street, Fig. 26. The street can be reshaped interactively, while the building models adapt to the new footprints. Finally, procedural extrusions can be used to model other architectural features such as windows or moldings, Fig. 27.

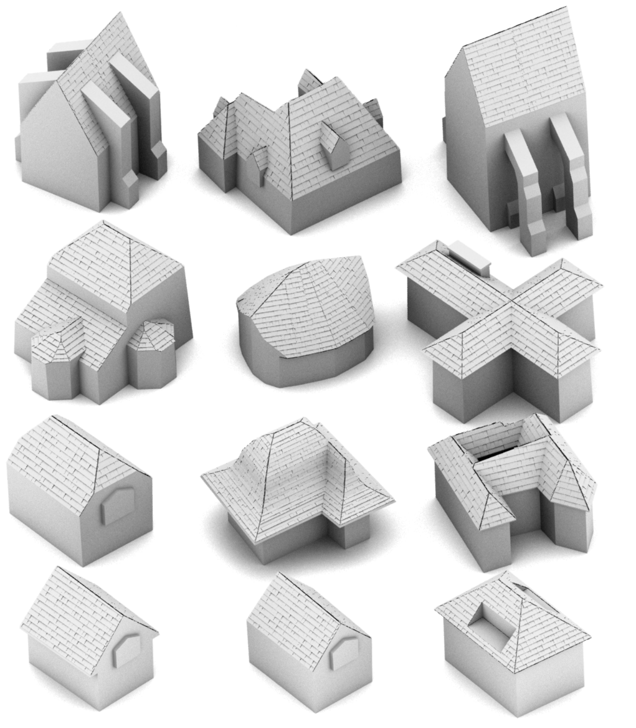


Fig. 23. From top, left: buttress, dormer windows, flying buttress, bay windows, curved plan, eight faces meeting on a symmetrical footprint with a chimney, hipped roof, curved roof, a horizontal overhang, an overhanging gable, standard gable and interior dormer windows

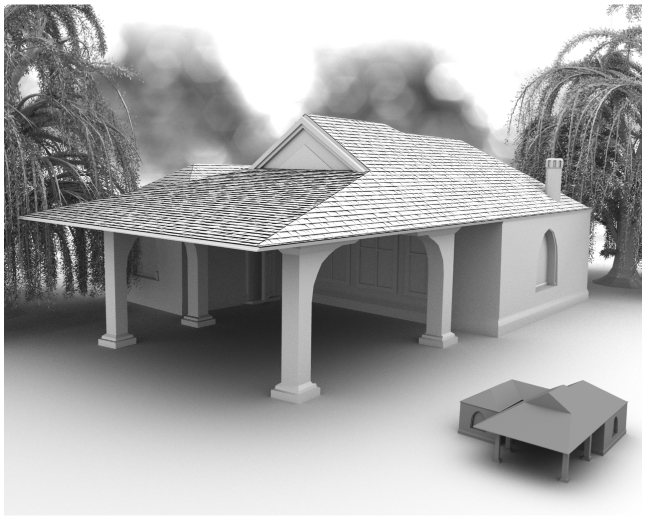
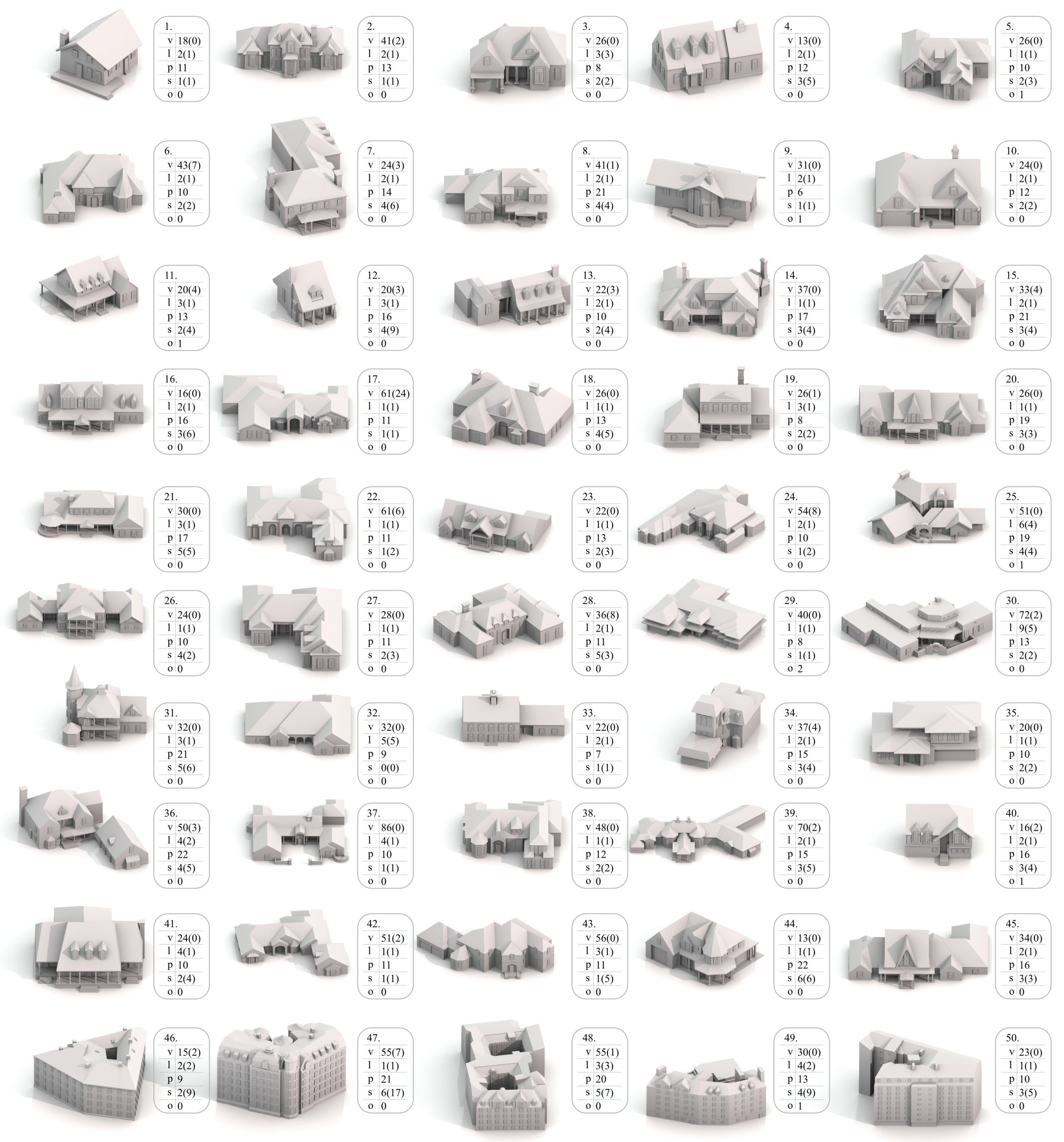


Fig. 24. Inset: the output of our procedural extrusions using a complex footprint, horizontal sections and plan edits. We are able to create pillars, covered parking and alcoves respectively. Main: A procedural condo with roof texture surrounded by procedural trees



| | v | l | p | s | o |
|---|---|---|---|---|---|
| 1. | 18(0) | 2(1) | 11 | 1(1) | 0 |
| 2. | 41(2) | 2(1) | 13 | 1(1) | 0 |
| 3. | 26(0) | 3(3) | 8 | 2(2) | 0 |
| 4. | 13(0) | 2(1) | 12 | 3(5) | 0 |
| 5. | 26(0) | 1(1) | 10 | 2(3) | 1 |
| 6. | 43(7) | 2(1) | 10 | 2(2) | 0 |
| 7. | 24(3) | 2(1) | 14 | 4(6) | 0 |
| 8. | 41(1) | 2(1) | 21 | 4(4) | 0 |
| 9. | 31(0) | 2(1) | 6 | 1(1) | 1 |
| 10. | 24(0) | 2(1) | 12 | 2(2) | 0 |
| 11. | 20(4) | 3(1) | 13 | 2(4) | 1 |
| 12. | 20(3) | 3(1) | 16 | 4(9) | 0 |
| 13. | 22(3) | 2(1) | 10 | 2(4) | 0 |
| 14. | 37(0) | 1(1) | 17 | 3(4) | 0 |
| 15. | 33(4) | 2(1) | 21 | 3(4) | 0 |
| 16. | 16(0) | 2(1) | 16 | 3(6) | 0 |
| 17. | 61(24) | 1(1) | 11 | 1(1) | 0 |
| 18. | 26(0) | 1(1) | 13 | 4(5) | 0 |
| 19. | 26(1) | 3(1) | 8 | 2(2) | 0 |
| 20. | 26(0) | 1(1) | 19 | 3(3) | 0 |
| 21. | 30(0) | 3(1) | 17 | 5(5) | 0 |
| 22. | 61(6) | 1(1) | 11 | 1(2) | 0 |
| 23. | 22(0) | 1(1) | 13 | 2(3) | 0 |
| 24. | 54(8) | 2(1) | 10 | 1(2) | 0 |
| 25. | 51(0) | 6(4) | 19 | 4(4) | 1 |
| 26. | 24(0) | 1(1) | 10 | 4(2) | 0 |
| 27. | 28(0) | 1(1) | 11 | 2(3) | 0 |
| 28. | 36(8) | 2(1) | 11 | 5(3) | 0 |
| 29. | 40(0) | 1(1) | 8 | 1(1) | 2 |
| 30. | 72(2) | 9(5) | 13 | 2(2) | 0 |
| 31. | 32(0) | 3(1) | 21 | 5(6) | 0 |
| 32. | 32(0) | 5(5) | 9 | 0(0) | 0 |
| 33. | 22(0) | 2(1) | 7 | 1(1) | 0 |
| 34. | 37(4) | 2(1) | 15 | 3(4) | 0 |
| 35. | 20(0) | 1(1) | 10 | 2(2) | 0 |
| 36. | 50(3) | 4(2) | 22 | 4(5) | 0 |
| 37. | 86(0) | 4(1) | 10 | 1(1) | 0 |
| 38. | 48(0) | 1(1) | 12 | 2(2) | 0 |
| 39. | 70(2) | 2(1) | 15 | 3(5) | 0 |
| 40. | 16(2) | 2(1) | 16 | 3(4) | 1 |
| 41. | 24(0) | 4(1) | 10 | 2(4) | 0 |
| 42. | 51(2) | 1(1) | 11 | 1(1) | 0 |
| 43. | 56(0) | 3(1) | 11 | 1(5) | 0 |
| 44. | 13(0) | 1(1) | 22 | 6(6) | 0 |
| 45. | 34(0) | 2(1) | 16 | 3(3) | 0 |
| 46. | 15(2) | 2(2) | 9 | 2(9) | 0 |
| 47. | 55(7) | 1(1) | 21 | 6(17) | 0 |
| 48. | 55(1) | 3(3) | 20 | 5(7) | 0 |
| 49. | 30(0) | 4(2) | 13 | 4(9) | 1 |
| 50. | 23(0) | 1(1) | 10 | 3(5) | 0 |

Fig. 25. The example cases and modeling statistics. *v* Vertices in modeled plan (additional vertices); *l* Polygons in modeled plan (polygons in library plan); *p* Number of profile sections in model; *s* Number of natural steps designed (number of natural step applications); *o* Number of offset events.

## 6. EVALUATION

To evaluate the skeleton as a modeling primitive we constructed 50 buildings. Here we detail the process we undertook to perform the modeling.

### 6.1 Evaluation Setup

We modeled each building from a plan and a perspective image. A set of four simple meshes (Fig. 28) were used to add detail to

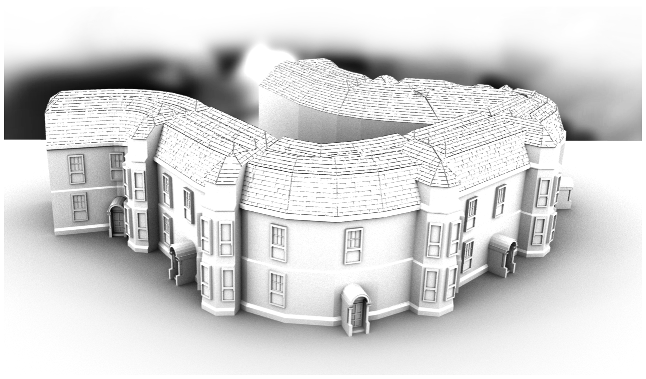


Fig. 26. A procedural model that renders a street from a spline. In this case the street was generated by four points defining the street's curve. Seed points were grown using another application of the skeleton to create the building footprints.

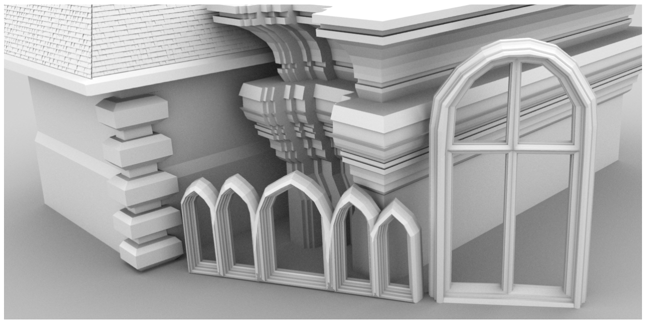


Fig. 27. Using a creative set of profiles, a wide range of architectural features can be created. By setting the input in a different plane, various windows may be extruded.

the structures. The events used for modeling were edge direction events, profile offset events and natural steps.

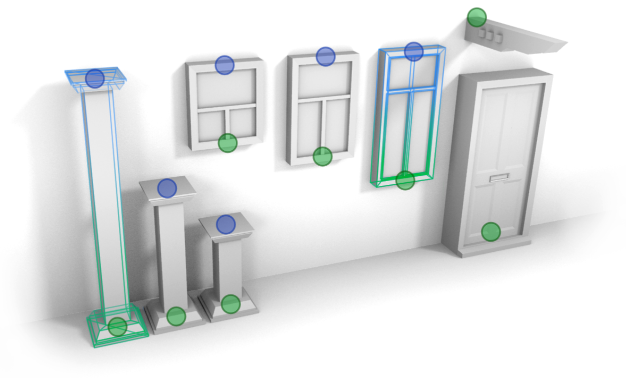


Fig. 28. The four example meshes used in the evaluation. The meshes are parameterized via control points (blue and green circles) and can be instanced to different sizes.

We undertook the evaluation with the goal that all major roof features from the elevation drawings should be present, although smaller details (such as cornices, plumbing and decorative windows) may be excluded. We traced the plans from those specified or aerial views of the property. The construction of profiles and positioning of features was performed by eye by the first author of this paper.

The first 45 buildings were taken from a library of ready designed architectural styles for family homes [Hanley Wood, LLC. 2010]. We modeled the first example in each of the categories the library provided. The library contained styles as diverse as *ranch* or *Dutch* (Fig. 25, examples 13 and 32 respectively), however much of the stylistic content was dependent on architectural details that were replaced with our simple meshes. Because the plans were predesigned, they had predominantly $90^\circ$ and $45^\circ$ degree angles between floorplan edges. That is, the design was not constrained by environmental features. To provide more challenging examples, we chose an additional five buildings from European cities that had irregular plans (Fig. 25, examples 46-50). These buildings were modeled from satellite and aerial views, Fig. 29.

The modeling times ranged from 20 to 120 minutes with a mean time of 63 minutes. Features on the input plan smaller than 30cm were not modeled. We also recorded a number of additional metrics for each building: the number of vertices in the input plan and in the model; the number of corner-loops in the input and in the model; the number of profiles in the model, the number of offset events, the number of natural step templates and the number of instances of those steps.

## 6.2 Evaluation Results

It was possible to model all the buildings using our interface. Some roof-lines were easier than others, and in this section we describe some of the problems encountered.

The most common issue when modeling was the construction of roof areas that contained edges not specified in the input plan (Fig. 30 (a). In these circumstances it was necessary to add extra edges to model these features. These would either be added in the plan, leading to the difference between the vertices in the input plans and the model in several of the examples, or by natural steps at certain heights.

We share a limitation with the straight skeleton that certain smaller edits to the footprint can result in bigger changes to the roof surface [Eppstein and Erickson 1998]. For example when two adjacent edges with different angles are nearly parallel, the behavior of the resulting roof can be erratic as the angle between the edges is set to greater than, or less than zero. In practice these edges do not appear often in architecture, and we often end up adding a perpendicular edge (Fig. 30, a).

In several circumstances one face relies upon another, spatially separated, face to halt its propagation at the correct time, that is an edge is fated to meet another (Fig. 30, b). When another feature blocks, or changes the course of one of these faces, the other may not terminate, or collide in an unexpected location. These fated edges lead to potentially undesirable intermediate outputs while editing.

Modeling circular arches was difficult because any adjustment in the width of the arch, would have to be accompanied by a re-scaling of the profiles. Modeling techniques such as shape grammars are able to retain such semantic information to automate such a process, and it is possible to imagine a similar system for the procedural extrusions.

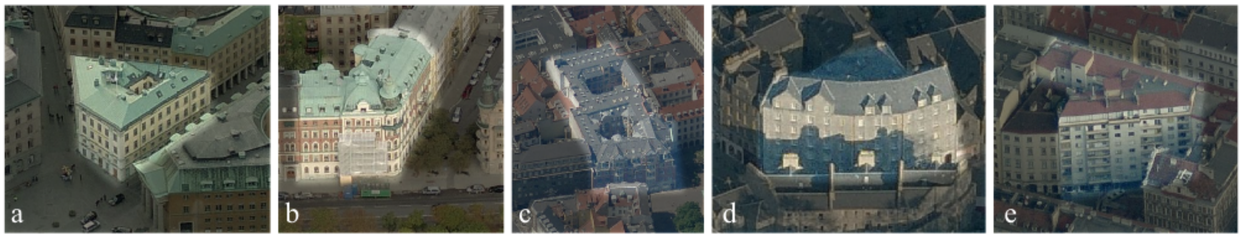


Fig. 29. Sample aerial photographs of buildings used for modeling examples 46 to 50 in Fig. 28. a,b) Stockholm, c) Copenhagen, d) Edinburgh, e) Vienna. ©2011 Microsoft Bing Maps [Microsoft Corp. ].

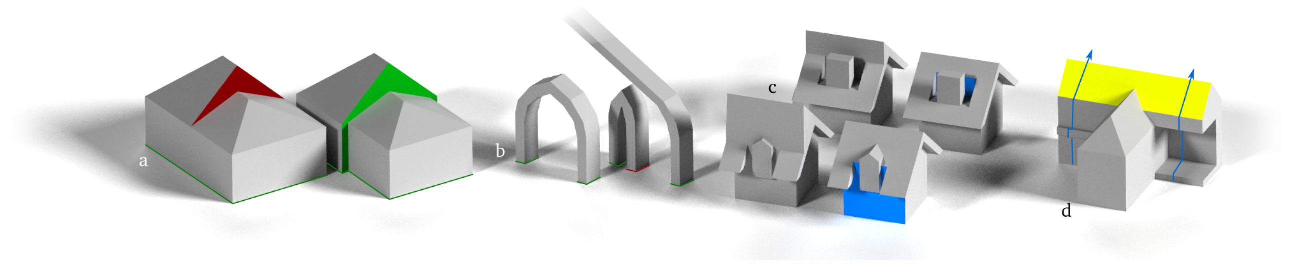


Fig. 30. a) The red roof face is not described in the input polygon(left). By creating a small change to the input polygon we can create the desired face (green). b) left: edges can be expected to collide at a certain height (green polygons), right: however when these edges are involved in other events (such as those from the red polygon), there may be undesired consequences, here a non-terminating polygon. c) Some structures (such as dormer windows and chimneys) do not obey the volume-maximizing resolution to the ambiguous case, in this situation we have to lower the ambiguous case priority of some edges (blue) to get the desired result. d) A face (yellow) may be shared between two profiles (blue lines), defining co-planar profile sections requires patience on behalf of the user.

It is not convenient to model a roof that is held only by a large number of pillars, because it is not easy to model the transition from pillars to the roof. For example, pergolas (Fig. 25, example 31) contain no walls to allow the plan to generate a roof. These were not a large part of our data set, and were approximated by walled structures of similar volume.

It was occasionally necessary to override our default of a volume maximizing priority in the ambiguous case. For example, in the case of a chimney stack or a dormer window (Fig. 30, c). To do this we used tags to specify high priority and low priority profile segments. This proved simple compared to the alternative of specifying a priority for every pair of segments.

While allowing one profile to split into two is the simple case of inserting an edge with a step, allowing two profiles to merge to one is more difficult (Fig. 30, d). We see this architectural feature as two different profiles to merge at the top of a shorter roof (Fig. 25, examples 3, 20). To design a profile with a face co-planar to another is difficult, especially if the second edge starts from an edge parallel, but not colinear to the first.

Natural steps proved very versatile for inserting edges into the polygons. For example, Fig. 25 (example 34) required a new edge internal to the plan for the back-facing wall of the tower. By positioning a wide square natural step on the end of the building, it was possible to split the polygon into two. One partition became the tower, and the other the remainder of the roof structure.

From a development perspective the algorithms are difficult to implement. It is hard to give a formal guarantee that the implementation will work correctly on all inputs. This may be observed when using our user interface, as occasionally a face will not contain enough arcs to close the area. In this case the face will not be visible to the user. This may occur once in every 5 minutes of interactive editing with multiple edits per second; It is certainly possible to construct pathological input cases. In the procedural case, we visually identified missing faces in two of the meshes, from the 6000 floorplans in the GIS database, Fig. 31.

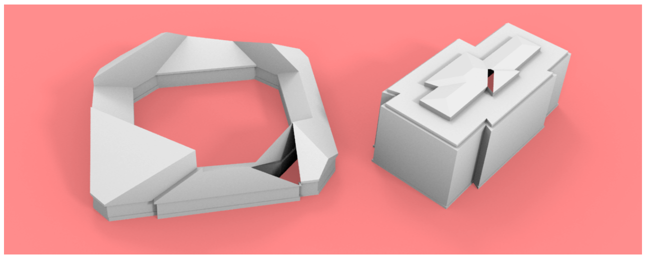


Fig. 31. The two observed examples of missing geometry. Note the missing roof sections in both buildings.

However, our modeling system is more specialized than most commercial polygonal modeling packages. The virtual model of Atlanta is unique and we argue that no existing approach can model a city of comparable (roof) complexity in reasonable time.

## 7. DISCUSSION

A major design decision for our system was to choose between a rational arithmetic or a floating point implementation. Our floating point implementation is better suited to interactive modeling applications because it prioritizes interactive update speeds over high precision. A rational arithmetic approach may be important to give theoretical guarantees and such an alternative implementation would be very valuable.

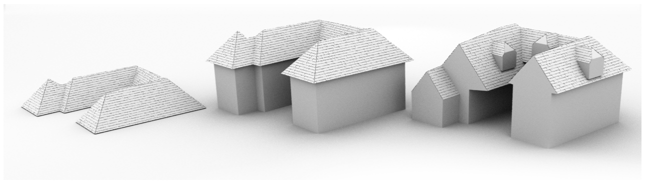


Fig. 32. Left: Straight skeleton; Middle: Straight Skeleton with angle changes; Right: Procedural extrusions

We are the first to introduce an algorithm for extrusions using edges with independent per-edge angles (weights). This results in a 3d instead of a 2d algorithm. We are also the first to recognise the difficulty of independent per-edge angles. The possibility of a 2d weighted skeleton is discussed in previous work [Eppstein and Erickson 1998; Barequet et al. 2008], but no algorithm is given and the ambiguous cases were not discovered. Even though our work is based on previous work in the unweighted case, e.g. [Cacciola 2004; Felkel and Obdrzalek 1998], our modifications result in substantial improvements in the range of forms that can be produced, Fig. 32.

In previous work [Havemann 2005] the direction of the extrudes is monotonic in the upwards direction, that is they are limited to angles above the sweep plane. By using profile offset events, we can allow non-monotonic profiles.

## 8. CONCLUSIONS

We believe that the combination of interactive and procedural modeling is a significant boost to artists productivity and a great complement to existing modeling tools. In some sense our work is complementary to previous work by Lipp et al. [2008]. Our approach to encoding procedural models is very different from the previous shape grammar approach [Müller et al. 2006; Lipp et al. 2008]. We believe that we are the first to provide a solution for the procedural modeling of roofs, procedural modeling from arbitrary building footprints, and other complex architectural surfaces. However, previous work is better suited for placing elements on facade planes and we see some potential in combining both approaches in future work.

The main contribution of this paper is the design of the system and the set of tool choices to enable procedural modeling of complex architectural surfaces. Procedural extrusions can model many complex architectural surfaces that could not be easily modeled with previous procedural modeling tools. Examples are curved roofs, overhanging roofs, dormer windows, interior dormer windows, roof constructions with vertical walls, buttresses, chimneys, bay windows, columns, pilasters, and alcoves.

## REFERENCES

AICHHOLZER, O. AND AURENHAMMER, F. 1996. Straight skeletons for general polygonal figures in the plane. In *Computing and Combinatorics*. Springer-Verlag, 117–126.

AICHHOLZER, O., AURENHAMMER, F., ALBERTS, D., AND GAERTNER, B. 1995. A novel type of skeleton for polygons. *Journal of Universal Computer Science 12,* 12, 752–761.

ALIAGA, D. G., ROSEN, P. A., AND BEKINS, D. R. 2007. Style grammars for interactive visualization of architecture. *IEEE Transactions on Visualization and Computer Graphics 13,* 4, 786–797.

AURENHAMMER, F. 2008. Weighted skeletons and fixed-share decomposition. *Comput. Geom. Theory Appl. 40,* 2, 93–101.

AUTODESK INC. Revit$^{TM}$. http://www.revit.com.

BAREQUET, G., EPPSTEIN, D., GOODRICH, M. T., AND VAXMAN, A. 2008. Straight skeletons of three-dimensional polyhedra. In *ESA '08: Proceedings of the 16th annual European symposium on Algorithms*. Springer-Verlag, Berlin, Heidelberg, 148–160.

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).*

CACCIOLA, F. 2004. A CGAL implementation of the straight skeleton of a simple 2d polygon with holes. In *2nd CGAL User Workshop*.

EPPSTEIN, D. AND ERICKSON, J. 1998. Raising roofs, crashing cycles, and playing pool: applications of a data structure for finding pairwise interactions. In *SCG '98: Proceedings of the fourteenth annual symposium on Computational geometry*. ACM, New York, NY, USA, 58–67.

FELKEL, P. AND OBDRZALEK, S. 1998. Straight skeleton implementation. In *Proceedings of Spring Conference on Computer Graphics*. 210–218.

GAL, R., SORKINE, O., MITRA, N. J., AND COHEN-OR, D. 2009. iWires: an analyze-and-edit approach to shape manipulation. *ACM Trans. Graph. 28*, 33:1–33:10.

HANLEY WOOD, LLC. 2010. eplans.com. http://www.eplans.com.

HAVEMANN, S. 2005. Generative mesh modeling. Ph.D. thesis, TU Braunschweig.

KELLY, T. W. A. 2006. City architecture generation. M.S. thesis, Bristol.

LAYCOCK, R. G. AND DAY, A. M. 2003. Automatically generating large urban environments based on the footprint data of buildings. In *SM '03: Procedings of the ACM symposium on Solid modeling and applications*. ACM Press, NY, USA, 346–351.

LEGAKIS, J., DORSEY, J., AND GORTLER, S. 2001. Feature-based cellular texturing for architectural models. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. SIGGRAPH '01. ACM, New York, NY, USA, 309–316.

LIPP, M., WONKA, P., AND WIMMER, M. 2008. Interactive visual editing of grammars for procedural architecture. *ACM Trans. Graph. 27*, 102:1–102:10.

MARVIE, J.-E., PERRET, J., AND BOUATOUCH, K. 2005. The FL-system: a functional L-system for procedural geometric modeling. *The Visual Computer 21,* 5, 329–339.

MERRELL, P. AND MANOCHA, D. 2008. Continuous model synthesis. *ACM Trans. Graph. 27*, 158:1–158:7.

MICROSOFT CORP. Bing maps$^{TM}$. http://www.bing.com.

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

PRUSINKIEWICZ, P. AND LINDENMAYER, A. 1991. *The Algorithmic Beauty of Plants*. Springer Verlag.

STINY, G. 1975. *Pictorial and Formal Aspects of Shape and Shape Grammars*. Birkhauser Verlag, Basel.

WONKA, P., WIMMER, M., SILLION, F., AND RIBARSKY, W. 2003. Instant architecture. *ACM Trans. Graph. 22*, 669–677.