EUROGRAPHICS 2011 / M. Chen and O. Deussen
(Guest Editors)

Volume 30 (2011), Number 2

Interactive Modeling of City Layouts using Layers of Procedural Content

M. Lipp[1], D. Scherzer[1], P. Wonka[2] and M. Wimmer[1]

[1]Vienna University of Technology [2]Arizona State University

Abstract

In this paper, we present new solutions for the interactive modeling of city layouts that combine the power of procedural modeling with the flexibility of manual modeling. Procedural modeling enables us to quickly generate large city layouts, while manual modeling allows us to hand-craft every aspect of a city. We introduce transformation and merging operators for both topology preserving and topology changing transformations based on graph cuts. In combination with a layering system, this allows intuitive manipulation of urban layouts using operations such as drag and drop, translation, rotation etc. In contrast to previous work, these operations always generate valid, i.e., intersection-free layouts. Furthermore, we introduce anchored assignments to make sure that modifications are persistent even if the whole urban layout is regenerated.

Categories and Subject Descriptors (according to ACM CCS): I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—

1. Introduction

Procedural city modeling is a rapidly evolving field in computer graphics, with applications to urban planning, design reviews, game design and others. Procedural techniques are often based on grammars and parameters, which makes it difficult to achieve exactly the desired outputs. Recent work has introduced a new paradigm for the procedural modeling of building facades [LWW08]: interactive modeling with direct control, allowing direct modifications of the generated output without having to go back to the original procedural specification. However, for the urban layouting step, there are no such solutions. This is mainly because city layouts do not offer a regular structure as facades do, but can be topologically very complex. City layouts are responsible for the overall aspect of an urban model and for controlling all the other parts of a procedural city generation system, and therefore it is paramount that the user has a powerful control mechanism for this step.

Current urban layouting systems [WMWG09, Pro10] offer only limited editing possibilities once an urban layout has been produced procedurally. While it is possible to drag individual nodes in the street graph, the resulting urban layout has intersections and is therefore not *valid* anymore (see Figure 1). To make the layout valid requires expensive manual operations like filling the gaps, reconnecting the street graph when an element is deleted, making space for new elements, etc. Even worse, if the underlying procedural description changes, the whole layout has to be regenerated from scratch, losing all manual customizations since they are not persistent. More complex operations like consistently merging different layouts, possibly from different sources (procedural or manual), are practically impossible.

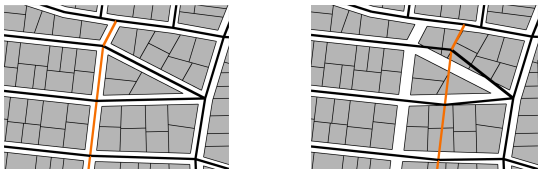

Figure 1: *This figure illustrates how, using traditional urban layouting tools, a simple translation transforms a valid layout (left) into an invalid one with intersections (right).*

In this paper we present an interactive city modeling system that is built on persistent editing operations that remain in the space of valid urban layouts. The system combines procedural edits, local manual edits, and higher level manual edits. It is designed to meet the following research challenges:

Direct control and editing of procedural layouts. Ideally,

to modify an urban layout, designers would like to use simple and intuitive editing operations, like soft selection of elements, drag and drop, insertion and deletion of elements etc. Most importantly, these operations should again produce a valid urban layout. They need to handle *changes in topology*, and should be designed to have *local influence* only. In particular, they should be *reversible*, so that the original appearance of a city part is restored when an inserted element gets dragged to another location (this is also called *circular editing*). An example of such an operation supported by our method is shown in Figure 2.

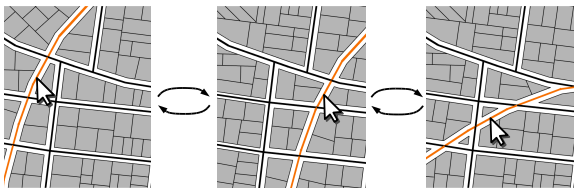

Figure 2: *The orange street is moved and rotated. The underlying parcels update accordingly. When the street is moved or rotated back, the original layout is regained again, providing circular editing capabilities.*

Combining urban layouts. One of the most frequent problems that occur in urban modeling is to consistently merge urban layouts at different levels. Examples include inserting a (manually modeled) parking lot, a park, a whole street, a whole block (like a shopping mall) or even a whole quarter, into an existing city layout. Consistent merging capabilities allow designers to reuse components or to model components separately. There is currently no solution to merge urban layouts automatically. One such merge process possible with our system is shown in Figure 3.

Figure 3: *Content from a different source, highlighted in orange, is inserted into the layout and moved, scaled and rotated. Full circular editing is supported.*

Persistence. Changes applied by the user should survive local and global editing operations. Urban layouting drives the whole city generation, like parameter assignments, distribution of landmarks etc., so these assignments need to survive modifications of the urban layout. Modifications to an already customized urban layout belong to the most expensive design operations for example in level design.

Main contributions. In order to meet the research challenges previously listed, this paper introduces a new set of editing operations that transform one valid urban layout into another valid one. Full *circular editing* capabilities like drag and drop, insertion, deletion etc., with arbitrary topological changes are provided. The operations are based on the combination of a *layering system* in the spirit of image manipulation programs, and a novel *merging algorithm* that consistently merges urban layouts based on graph cuts. We also extend the locator concept introduced in previous work to achieve *persistent anchored assignments*, linked to elements in an urban layout, allowing modifications to survive global procedural modifications. These methods are implemented in a city modeling system that combines the power and convenience of procedural street generation with the flexibility and direct artistic control of a traditional content creation system.

2. Previous Work

This paper addresses the editing of city layouts, including street networks, parcels, and parameter distributions for building generation. In a procedural production environment, our work can be complemented by other components for: 1) the generation of three-dimensional street geometry [Zim07], 2) the procedural modeling of buildings [MWH*06], 3) the editing of procedural building models [LWW08], and 4) plant generation [PL91]. A more extensive review of procedural urban modeling methods can be found in a recent survey paper by Vanegas et al. [VAW*10].

While there is substantial work on the procedural *generation* of city layouts [PM01, KM07, CEW*08, WMWG09, AVB08, VABW09], the work on *editing* procedural layouts has only started recently, so that no competing solutions to the problems explained in the introduction exist in the literature. However, there are some promising initial ideas for the editing of urban layouts. We can categorize these strategies into three categories:

Direct low-level editing. The street graph network generated by a procedural model can be edited using traditional interactive editing operations, such as moving vertices (intersections), adding edges (street segments), and deleting edges of the graph. These operations were used by most procedural systems starting from [PM01], and are commercially available in Procedural's CityEngine [Pro10]. The important unanswered problem is how to preserve local edits after a change to the procedural model is made and how to ensure that the urban layout remains valid.

Global procedural regeneration. The most common operation in procedural models is to change parameters of the procedural model and then regenerate the complete model. Examples of this type of operation are moving city centers [PM01], changing the underlying tensor field [CEW*08], changing simulation parameters during an urban simulation [WMWG09], and changing population

density values or job locations [VABW09]. While these operations may be applied only locally, the global procedural regeneration results in (potentially drastic) global changes, even in parts of the model that are distant from the local edit of the parameters. These global changes are difficult to anticipate and control for a designer due to a lack of direct control, and prevent any form of direct control mechanisms such as modifying individual streets or lots.

Local procedural regeneration. Another editing operation is to select a part of the model, delete it, and regenerate the deleted part [CEW*08, AVB08]. Similarly, Kelly and McCabe [KM07] proposed a mixture of interactive and procedural techniques: Major roads are created manually, while the minor roads enclosed by main roads are created procedurally. The capability of local regeneration is an attribute of the procedural model, and also other recent papers can be set up to constrain modifications to an area [WMWG09, VABW09]. Local procedural regeneration shares some drawbacks of direct low-level editing and global procedural regeneration: global changes destroy local edits, and even local procedural regeneration can be difficult to control.

Our work is also inspired by Lipp et al. [LWW08], who introduced several methods connected to editing procedural models in the context of facade modeling. Further, we were inspired by editing operations on graphs (e.g. [ABVA08, ZHW*06]) or man-made objects (e.g. [CLDD09, GSMCO09]) through optimization. We share the goal of enabling editing on graphs from these papers, but the challenges posed by procedural models require a different methodology. In particular, optimization strategies often lead to global changes that are difficult to control.

Layered procedural modeling. Outside the context of city editing, a layer-based procedural editing method that targets single objects and supports triangle meshes was introduced by Schmidt and Singh [SS08]. In contrast to this, we target city-wide modeling and support three semantically different categories (streets, parcels and assignments) for each layer.

Graphcut. In image processing, graphcut [FF62] can be used to merge images [KSE*03]. Zhou et. al [ZHW*06] showed how to use graph cuts to merge overlapping mesh regions in the context of geometric texture synthesis. In contrast to our work, they use multiple local graph cuts in overlapping regions, while we use a global graph cut on the whole city layout.

3. Transformations of Urban Layouts

The main problem to solve in a city modeling system is how to transform a *valid* urban layout into another one, as illustrated in Figure 1. For this we introduce three basic transformation operators and show how they can express most direct urban layout editing operations.

3.1. Definition of Urban Layouts

An *urban layout* U consists of a *street network* and *parcels*, as shown in Figure 4. The street network is given as an undirected planar graph $G = (V, E)$ with nodes V and edges E. We also refer to the nodes as crossings and the edges as streets (note that for simplicity we do not distinguish between street segments and streets). A parcel $p \in P$ is a possibly concave, simple polygon with at least 3 vertices. When referring to either a street, a crossing or a parcel we use the term *element*.

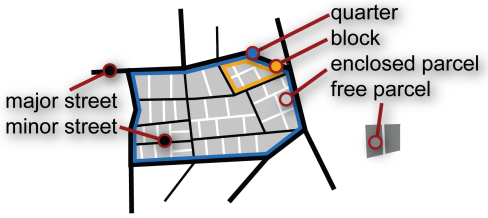

Figure 4: *The basic building blocks for urban layouts.*

A face of an embedded planar graph is a cycle that surrounds a region that contains no edge. We call the faces of the planar graph G the *blocks* B of an urban layout. The faces obtained when ignoring all minor streets are called *quarters*.

Street network and parcels are connected through a binary ownership relation $O \subseteq P \times B$ between parcels and blocks: Every parcel p that is completely inside a block b is *owned* by this block, and $(p,b) \in O$. A parcel is owned by at most one block. Parcels without an owner are called *free* parcels, while the others are called *enclosed* parcels. Given an urban layout, O can be calculated by first finding all blocks and then performing containment tests of all parcels with those blocks.

Crossings, streets and parcels can have an arbitrary amount of key-value pairs attached, we refer to them as *tags*. The tags of every street must at least contain the key `streetType` with the value of either `minor` or `major` to discern between minor and major streets (other street types like highway would also be possible).

The definition of urban layouts so far does not ensure that such layouts make sense. It allows streets intersecting without crossings, parcels intersecting streets etc. In order to restrict editing operations to "useful" urban layouts, we define an urban layout U as being *valid* if (1) there are no intersections between streets and (2) parcels do not intersect streets or other parcels. We denote such a layout with $\overline{U}$.

In the following, we define three operations on valid urban layouts $\overline{U}$, i.e., the result will again be a valid urban layout: The non-topological transform T, a flexible merge operation M_f and a hard merge operator M_h. In Section 4 we will then describe how most direct editing operations can be expressed using T, M_f, M_h and a layering system.

3.2. Non-Topological Transform

Many editing operations consist of small changes to an existing layout. For example, the user drags one node, denoted v, in the street graph by a small amount, represented by the affine transformation A. As a *small* transformation we define one that does not change the topology of the street graph (i.e., the dragged street does not intersect a new street). For such small transformations, the user expects the layout to remain valid (i.e., the parcels in adjacent blocks move with the changed street), and that these changes have as few side effects as possible.

Therefore we introduce the non-topological transform $T(\overline{U},A,W)$, where A is an affine transformation and W contains a weight $w_v \in [0,1]$ for every node in $v \in \overline{U}$. In the previously mentioned example of a dragged node v, A would be a translation and the weights would be 1 for v and 0 for the other nodes. $T(\overline{U},A,W)$ ensures that after applying A the layout is still valid, and works as follows:

(1) Every node $v \in V$ is transformed using $v_1 = (Av)w_v + v(1.0 - w_v)$. This creates a potentially invalid layout U_1. (2) We test U_1 for street to street intersections, to ensure that there are no topological changes. If there are intersections, $T(\overline{U},A,W)$ simply returns the original layout $\overline{U}$, essentially ignoring the transformation A. (3) If there are no street to street intersections, we modify the parcels in U_1 to ensure there are no street to parcel intersections. This creates a valid layout $\overline{U}_2$, which is then returned. Note that concurrently to our work, the newest release of the commercial Cityengine [Pro10] features some kind of automatic parcel updating, but no details on the implementation have been published. Our method of updating the parcels has several steps:

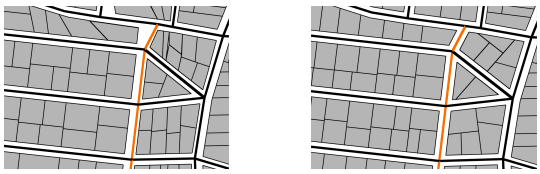

Figure 5: *The orange street was moved. Left: Smooth transformation of parcels, exhibiting some distortions. Right: Local parcel regeneration.*

Finding affected parcels. All parcels contained in blocks adjacent to transformed streets need to be updated. The algorithm goes through all blocks and checks whether any of the vertices v contained in the cycle defining the block has a non-zero weight w_v. In turn, all affected parcels contained in those blocks are found using the parcel ownership relation O. We denote blocks before applying the transformation A as b and after applying A as b_1.

Parcel update. It is desirable to update parcels in a "smooth" way. However, larger deformations require adding or removing parcels. We therefore provide two mechanisms for parcel updates: a *smooth transform*, which geometrically distorts the parcels to fit the new block, and *local regeneration*, which procedurally recreates the parcels in the block. The differences are shown in Figure 5. We decide on a per-block basis which mechanism to employ: When either the area of the corresponding block or the angles enclosed between connected block edges change more than user-defined thresholds, the parcels of this block are regenerated, otherwise they are transformed smoothly.

Smooth transform. We first calculate the mean value coordinates [HF06] of every vertex of every affected parcel with respect to its untransformed owner block b. Using those coordinates, we recalculate all vertex positions with respect to the transformed block b_1. This method retains the original parcel layout, but may lead to distortions.

Local regeneration. First, all parcels originally belonging to b are deleted. The new parcel boundary is obtained by *shrinking* b_1 in order to accommodate for the distance of the parcels to the road, using the skeleton-based algorithm found in CGAL [Cac09]. Finally, parcels are generated procedurally in the new parcel boundary using the method introduced by Weber et al. [WMWG09].

3.3. Flexible Merging using Graphcut

Editing operations that change the topology of the street network are much harder to realize than topology-preserving ones. An important contribution of our paper is that we express these edits using sequences of operations that involve *merging* two different urban layouts, as will be discussed in Section 4. As a simple example, an arbitrary translation of a street can be achieved by moving the street to a separate layout, translate the street there, and merging the temporary layout back. More involved operations require the merging of whole city parts with user-defined priority maps.

As the heart of these operations, we introduce a flexible binary merge operator $M_f(\overline{U}_a,\overline{U}_b)$ that is designed to merge two urban layouts $\overline{U}_a$ and $\overline{U}_b$, producing a new valid urban layout. Similar to alpha mattes in image processing, we allow the artist to flexibly assign priorities to elements in the layouts.

Unlike image mattes, the priority cannot be incorporated through a simple compositing operation of regular images. Instead, in this section, we show a compositing algorithm on urban layouts that is computed using graph cuts.

Let us first review graph cuts [FF62]: Consider a graph $G = (V,E)$, a source $s \in V$, a sink $t \in V$ and a *capacity* c_e associated with every edge $e \in E$. Then an *s-t* graph cut partitions the vertices into two subsets S and T with $s \in S$ and $t \in T$. The cut-set includes all edges whose vertices are in different partitions. The cut is minimal if the sum of all edge capacities in the cut set is minimal.

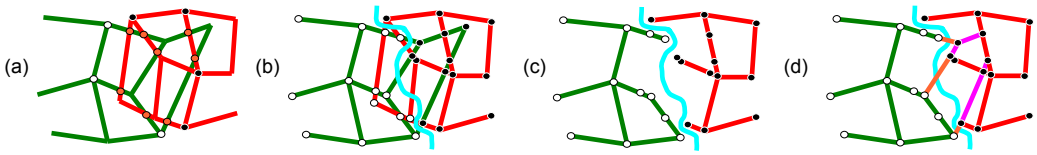

Figure 6: *Application of graph cut to city layouts. (a) Creation of shared graph. Green: $\overline{U}_a$, red: $\overline{U}_b$, orange dots: intersections, white/black dots represent nodes with constraint arcs to source/sink. (b) The blue line represents a possible cut. White and black dots now represent the graph coloring. (c) Deletion of streets with nodes in a wrong partition. (d) Mending of holes by including certain streets of $\overline{U}_a$.*

In image processing, a graph cut is often employed to merge different images [KSE*03] when blending is not desirable. The question now is how to cast the merging of two urban layouts $\overline{U}_a$ and $\overline{U}_b$ into a graph cut problem. The general idea is to work on the street networks of the layouts, and interpret the user priorities as capacities for the graph cut.

However, there is an additional challenge: In order to calculate a graph cut, a single shared graph needs to be constructed from the two source layouts. In image processing, this is straightforward, as the different images share a common pixel grid which defines a shared graph. There is no obvious shared graph for the city layouts $\overline{U}_a$ and $\overline{U}_b$. Further challenges are the creation of source and sink nodes, and the reconnection of the two partitions. The whole procedure works as follows, and is illustrated in Figure 6:

(1) Assign priorities. (2) Create a shared graph. (3) Automatically create a source and a sink, and create constraint arcs to them. (4) Search for a minimal s-t cut. (5) Delete streets that are in the wrong partition, and reconnect partitions. (6) Update the corresponding parcels.

Assign priorities. Numerical priority assignments to nodes are done by the artist using anchored assignments as shown in Section 5, giving two separate priority distributions for $\overline{U}_a$ and $\overline{U}_b$. Each street samples the corresponding priority distribution at its midpoint to obtain its capacity c_e to be used for the graph cut.

Creation of a Shared Graph. We need to bring $\overline{U}_a$ and $\overline{U}_b$ into a common shared graph G in order to apply the graph cut algorithm: First, every street from $\overline{U}_a$ is copied to the shared graph G. Then, an intersection test of every street in $\overline{U}_b$ with the streets in G is performed (coincident streets are deleted). A new crossing is created at every intersection, and the streets involved in the intersection are rerouted along these new crossings. We call these new crossings *intersection nodes* N_i. During insertions, we add a tag to every street indicating its original layer. An example shared graph is shown in Figure 6(a). To provide better numerical stability, we remove all dead end streets.

Source and Sink Connections. We create a source node corresponding to layer $\overline{U}_a$, and a sink node for layer $\overline{U}_b$. While source and sink do not have a position in space, it is crucial which nodes are connected to source and sink: The minimal cut should be located in the region where $\overline{U}_a$ and $\overline{U}_b$ overlap. If it were outside, large holes would occur after step 5 in the algorithm (see grey lines in Figure 7 for such cuts). Therefore, analogously to image merging [KSE*03], the borders of the overlapping region need to be constrained to belong to one of the partitions, forcing the cut to be in the overlapping region. This is done by connecting each border node to source or sink using a constraint arc, e.g., an edge with a high capacity, which essentially forces the node to be in the source or sink partition after the cut.

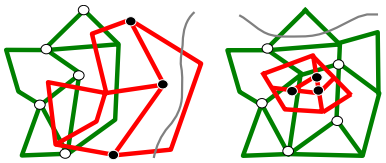

Figure 7: *Visualization of constraint arcs: White and black dots represent necessary constraint arcs to the source and sink respectively. Left: Overlapping region. Right: Red graph is completely contained in the green graph. Grey lines are examples of cuts that we want to prevent.*

This is simple for regular images, but for irregular street networks we need to employ a more involved algorithm to find border nodes: (1) Find potential border edges, which connect one node a from $\overline{U}_a$ or b from $\overline{U}_b$ with an intersection node from N_i. All nodes a and b are potential border nodes. (2) Remove all potential border nodes a that are inside a block of $\overline{U}_b$, and all nodes b inside a block of $\overline{U}_a$. This removes all nodes that can not be a border because they are located inside a block of the other graph. (3) All remaining nodes are border nodes. An example set of border nodes is shown in Figure 7 (left).

There is one special case, shown in Figure 7 (right), that this algorithm does not handle: When the graph $\overline{U}_a$ is completely inside the graph $\overline{U}_b$, no border nodes will be found for $\overline{U}_a$. To still constrain some nodes of $\overline{U}_a$, we use the following heuristic: The nodes of the edges with the n highest priorities are constrained. We found $n = 5$ to be sufficient in our test cases.

Computing minimal cut. The algorithm of Edmonds and Karp [EK72] is used to solve for the maximum flow and respectively the minimum cut. The result is a coloring of the graph where white nodes correspond to the source partition and black nodes to the sink partition, as shown in Figure 6(b).

Deletion of streets and reconnection. Now we delete all streets whose nodes are not in the correct partitions. $\overline{U}_a$ nodes should be in the white partition, $\overline{U}_b$ nodes in the black one. This separates the graph as seen in Figure 6(c). In order to mend the created holes, we first add back all streets of $\overline{U}_a$ that have at least one correct node, shown as orange lines in Figure 6(d). This may still leave some holes. Therefore we start a depth first search at each border street of $\overline{U}_a$ and add back all encountered $\overline{U}_a$ streets until a street would enter a block of $\overline{U}_b$. All streets added back this way are shown in magenta in Figure 6(d).

Note that it would also be possible to alter this algorithm by adding back streets of $\overline{U}_b$ instead $\overline{U}_a$. The choice of adding streets of $\overline{U}_a$ or $\overline{U}_b$ to mend the holes essentially determines if $\overline{U}_a$ or $\overline{U}_b$ should be preferred near the cut. We allow the artist to override the default behavior of using $\overline{U}_a$ with a global setting.

Incorporation of parcels. To correctly handle parcels, the following preparation steps are performed before the shared graph is created: First, all free parcels are enclosed with streets. This is necessary because the merging only considers the streets. Then, all blocks B_a, B_b and ownership relationships O_a, O_b of the layouts $\overline{U}_a$ and $\overline{U}_b$ are calculated.

Using those relations, adding the parcels back after the graph cut is done as follows: First we find all blocks of the graph cut result. Then, for every block that contains only streets of one layer, the parcels that were previously defined for this block are found using the ownership relation and are added. The parcels for all the other blocks are procedurally regenerated.

3.4. Hard Topological Merge

One important special case of the flexible merge $M_f(\overline{U}_a, \overline{U}_b)$ occurs when the priorities of $\overline{U}_b$ are much higher than the priorities of $\overline{U}_a$: All the elements of $\overline{U}_b$ will be present in the result. This has two advantages: First, having all the elements of $\overline{U}_b$ retained may be the desired result when an artist merges a small but important element into a city. Second, we can significantly speed up this special case, as we will show in this section.

Let us therefore introduce the *hard* topological merge operator $M_h(\overline{U}_a, \overline{U}_b)$, with the property that all elements of $\overline{U}_b$ are present in the result. The main idea to speed this up is that we already know where the minimal graph cut should be: All streets of $\overline{U}_a$ that intersect the concave hull of any connected component of $\overline{U}_b$ must be in the cut set. This ensures that the cut is just outside of $\overline{U}_b$.

As we know where the cut should be, we do not need to perform a graph cut. But there is another chance for improving performance: In the first step of the flexible merging, a shared graph is created by intersecting every street of $\overline{U}_b$ with the streets of $\overline{U}_a$. However, as we know that no street of $\overline{U}_a$ should protrude into a concave hull of $\overline{U}_b$, we can simply intersect those concave hulls with $\overline{U}_a$, reducing the amount of intersection tests. This simplifies the algorithm to the following:

(1) Insert all elements of $\overline{U}_b$ into the result. (2) Find the concave hulls of connected components. (3) Clip all elements of $\overline{U}_a$ against the concave hulls, and insert the clipped result. (4) Incorporate the parcels.

To make the clipping numerically stable, we insert streets that do not enclose any block (shown in Figure 8) separately using line intersections.

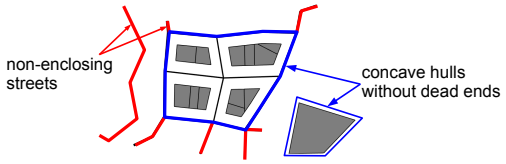

Figure 8: *Streets are classified into hull and non-enclosing streets for numerically stable clipping.*

Finding the hulls and the non-enclosing streets can be done using the block list B_b and the ownership relation O_b. All streets that are not a border of any block are non-enclosing streets. All streets that are a border of exactly one non-empty block are part of a concave hull without dead-ends. All concave hulls can now be found by using the block-finding algorithm while ignoring all streets that are not part of the concave hull.

4. Editing Operations Using Layers and Layout Transformations

In this section we introduce our layering concept and explain how most editing operations can be mapped to a combination of layers and the operators introduced above. All operations result in valid urban layouts.

Layering is well known in image processing tools, and merging of layers is trivial in this domain. For urban layouts this is more involved, therefore we use the operator M_f or M_h described above for the merging of layers.

Layer definition. A layer L consists of one urban layout, i.e., a street network and parcels. A scene can have a finite number of layers $(L_1, L_2, \ldots, L_n)$. What is finally displayed, exported, etc., in an interactive editing system is a merged

layer L_m which is iteratively defined using $L_m = L_n$ and continues with $L_m = M_f(L_{n-i}, L_m), i = 1 \ldots n-1$, until all layers are included. This definition ensures that elements in layers with a higher number precede elements on lower layers. If an artist wants to reduce the amount of layers, two layers can be collapsed to one combined layer using M_f or M_h.

4.1. Basic Editing Operations

Simple and soft selections. The basic editing workflow involves *selecting* elements in an urban layout and *transforming* them. Selections of elements are represented as a weight $\in [0,1]$ for every element. A user can select *single* elements or a *region* using mouse clicks. This sets the weights for all selected elements to 1 and the other ones to 0. We also provide a soft-selection tool: When the user clicks on a city part, the weights of every city element are set according to the distance to the mouse position.

Geometric transformations. Affine transformations, denoted as A, like translation, rotation and scaling, can be applied to a selection. When the artist does not want topological changes to occur, this can be represented as a non-topological transformation $T(\overline{U}, A, W)$ of an urban layout as discussed in Section 3.2, where W represents the selection weights.

Topological changes. If the artist wants to have topological changes while still retaining cyclic editing capabilities, we allow this using the following flexible and generic way: The artist can delete the selected elements in L_a and insert them into a new layer L_b, and apply the transformations to L_b. Now an arbitrary amount of cyclic editing operations can be performed in L_b, as shown in Figure 2 and 3, without modifying L_a (except for the initial deletion of the selected elements). Through the layering system, they will be merged at the new position with the current layout, always giving a valid urban layout as result.

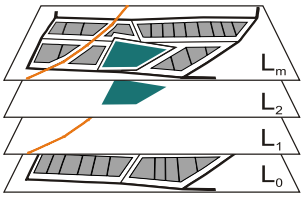

Figure 9: *Layering: An existing urban layout in layer L_0 is merged with a street on layer L_1 and with a park on layer L_2, resulting in the merged layer L_m.*

Insertion and deletion. Inserting a new element (street, parcel, ...) works by placing it on a new layer L_o. Through the definition of the layering system, L_o will be automatically merged with the existing layers to give a merged result L_m (see Figure 9). Upon *deletion*, the selected elements are simply removed from the urban layout. Since no new intersections are created, the resulting layout is valid.

4.2. Further Examples of Direct Artistic Control Using Layers and Merging

Persistent local changes. In procedural editing systems, changing the input parameters to the procedural algorithm usually causes a regeneration of parts of or the whole urban layout. In order to protect local modifications to the urban layout, the user can mark important elements with the key-value pair $protected = true$. Now, before the regeneration occurs, the system automatically copies all marked elements into a new higher level layer. Then the urban layout in the original layer is replaced with the newly generated one, and the merging process ensures that the protected elements are retained. Figure 10 shows an example of this process.

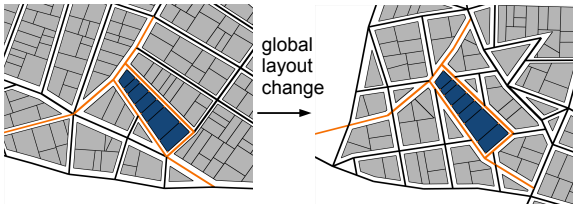

Figure 10: *Left: The user marked the orange elements as protected. Right: After a global layout change, the elements are still preserved.*

Merging assets. It does not matter if the urban layout in a layer originates from a procedural creation algorithm or was hand crafted by a designer. The layering system allows merging content from different sources in a unified way. As an additional benefit, a procedural algorithm does not need to know anything about the layering system, and thus every street generating algorithm can be employed.

Combining styles using flexible merging. When street networks with different styles are defined in different layers, an assignment of priorities for the flexible merging can be used to specify where a specific style should be used. An example of this is shown in the accompanying video.

Tweaking in an advanced development stage. In the context of computer games, moving gameplay-relevant urban layout parts into a distinct layer allows moving them freely around the city. This enables fine tuning during the whole production process.

4.3. Artist Interaction

We will now explain how the introduced methods are actually presented to the artist in a graphical user interface, as shown in Figure 11.

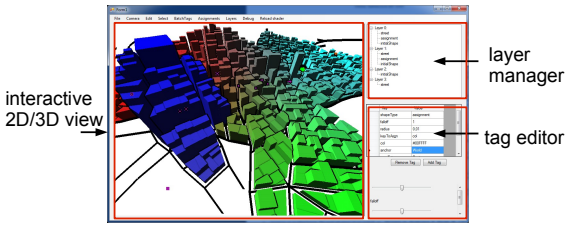

Figure 11: *The user interface of our implementation.*

A layer manager allows creation, arrangement and deletion of layers. Using buttons, selected elements can be copied or moved between layers. Importing of external assets into layers as well as exporting of a merged result is also supported.

In an interactive 2D/3D view, a real-time rendering of either a single layer or the merged result is shown. A tool bar provides standard selection and transformation operations like rotation, scale and translate. As a default, transformations on one layer are always *non-topological*. To ensure this, and to maintain validity, transformations modifying the topology of one layer are detected using intersection tests and automatically undone. In order to perform a *topological change*, on a press of a button the current selection is moved to a new layer. Transformations are then applied on the new layer, and the merging ensures valid layouts. For every layer, the user can specify if a flexible or hard merge should be performed.

Finally, a tag editor allows direct key-value pair assignments to selected objects. It is also used for the anchored assignments introduced in the next section.

Advanced operations can also be performed with this user interface: Merging of assets is achieved using the layer manager, combining of styles using the layer manager and a priority assignment. To simplify persistent local changes, we provide a shortcut that first marks all selected elements as protected before a regenerated urban layout is imported.

5. Persistent Anchored Assignments

In a city modeling system, urban layouts can be refined by assigning tags (key/value pairs) like building height to elements in the layout. In previous work, this is either done using global image maps [PM01] or by directly modifying tags of individual elements. However, when using our flexible editing operators, those methods would not be appropriate: A global image map would not follow geometric transformation, while direct assignments to elements would be lost when the elements are deleted (for example caused by a global regeneration), causing a persistence issue.

Therefore we introduce *anchored assignments*, which follow geometric transformations and are persistent after element deletions. An example is shown in Figure 13. In the following sections we will first define them and then show how they are used.

5.1. Definition of Anchored Assignments

An anchored assignment consists of *tags*, a *target*, a *world space position* denoted as *Pos*, and an *anchor*.

Tags describe *what* key/value pairs to assign, and can represent properties like building height, type or style. They can also point to specific 3d assets to be used, enabling the placement of landmark buildings. Also priority assignments for our flexible merge operator can be represented in tags.

The world space position is a simple vector representing the global placement of the assignment in the city.

A target specifies to *which elements* the tags should be assigned. It contains a list of element types that should be affected, including minor/major streets, nodes, or parcels. Further, it specifies if only the nearest element of given type to *Pos* or a complete region centered at *Pos* should be affected. For regions, it is also possible to specify a numerical distribution, for example a radial falloff with respect to the distance to *Pos*. This is especially useful to define priority distributions for flexible merging.

An anchor describes how *Pos* should react to transformations of the layout. It contains a pointer to one specific element of the layout. When this element is transformed, the same transformation is applied to *Pos*, essentially moving the assignment relative to the element.

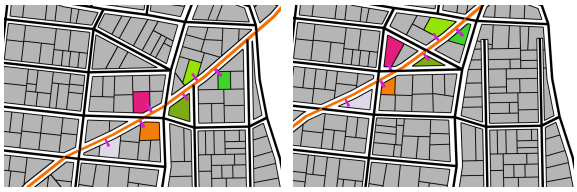

Figure 13: *Purple lines represent a color assignment anchored to the adjacent street, with a parcel as target. When the orange street is moved, the anchored assignments stay relative to the street, and apply to the nearest parcel.*

5.2. Usage of anchored assignments

We will now show how to create anchored assignments, how they maintain persistence, and how they are actually applied to the layers.

Creation. For every layer, a user can add an arbitrary amount of assignments. To add one assignment, we provide a graphical assistant. Here, the user can input the key/value pairs and specify the target properties. Then the user clicks somewhere in the city to specify *Pos*. Finally, he clicks on a specific element in the layer to set the anchor.

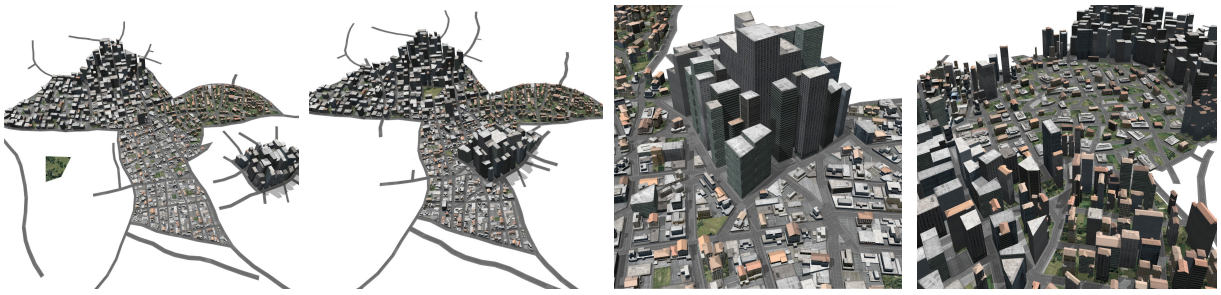

Figure 12: *Different editing stages for a city. From left to right: (1) The long street on the left, the park, the main city and the city center on the right are on separate layers. (2) This is the result of moving the layers on top of each other. (3) Close-up of the merged part. (4) The flexible merging operator was then used to incorporate a part from the street network layout of Rome.*

Persistence. Every layer has a list of assignments, which are stored separately from the urban layout. This ensures that deletions of elements in the layout do not delete the assignments, this way they are persistent. However, when an element is deleted, we have to update every anchor that references to this element. This is done by modifying the anchor to point to the nearest element of the same type instead.

Application. The actual application of the assignments to the elements is done during the layer merging process. Recall that two layers are iteratively merged using $L_m = M_f(L_{n-i}, L_m), i = 1 \ldots n-1$. Now, every time before M_f is called, we apply the assignments stored for layer L_{n-i} in the following way:

For every assignment, (1) using the target specification, we search for the elements $\in L_{n-i}$ where the tags should be applied, and add the tags to those elements. In case the target specifies a region with a numerical distribution, the tags are multiplied with those values before they are added (of course multiplication is only performed when the values are of numerical type). (2) In case the same key was already defined in an element, our system keeps the previously specified value. This way assignments on higher levels precede assignments on lower levels.

6. Results and Discussion

We have implemented our methods in a stand-alone C# application. Urban layouts can be imported from and exported to the CityEngine [Pro10]. To generate final city geometry, the grammar system in the CityEngine is used, based on the tags assigned to objects. In our case, the colors were used to choose different building types. Figure 12 shows an example urban layout with geometry created in the CityEngine.

Artist feedback. During the design of our interaction methods, we consulted artists and programmers from a computer game company on their ideas and needs for an urban modeling tool. They noted that the lack of merging of handcrafted assets as well as the missing direct artistic control was a major disadvantage in previous work. We conducted an informal guided user session. The initial feedback is positive, especially the seamless integration of assets from multiple sources using a layering system as well as flexible merging is intuitive from an artist's point of view, and has promising potential to facilitate artist collaboration. They also appreciated the predictability of M_h when merging small city parts.

Validity. Determining the validity of an urban layout is a complex issue, because the definition of validity depends on the application. For example, an urban planner may consider a factory in a residential area as invalid, while a movie director could employ seemingly unexpected cities to increase the dramatic effect. In urban planning, also higher level semantics like symmetries, repetitions or context may be of interest.

Our approach for validity definition is therefore a compromise trying to find the lowest common denominator for different applications. We define validity based on intersections of elements in urban layouts. One advantage of this low-level syntactical approach is that a validity test is simple to implement. Further, an intersection-free layout should be the basic requirement for most applications, making it intuitive for most users. The main disadvantage is that it is not possible to automatically capture the previously mentioned higher level semantics. Adding them would be an interesting area for future work.

3D Terrains. In our current implementation, all operators work on a two-dimensional plane. Cities with terrains represented as height fields have to be projected onto this plane first. After the operators, the result can be projected back onto the terrain. This has the disadvantage that steep roads cannot be detected and removed using our operators. Therefore it would be interesting future work to incorporate the gradient of the terrain into our operators.

Performance. We achieve interactive frame rates of around 15fps (on an Intel Core2 Quad 6600) for moderately sized

cities of about 3,000 parcels and 560 streets on four layers, when moving one layer and performing merging with M_h. Using M_f, the frame rates are around 5fps. In our current implementation, no spatial acceleration structures are employed for intersection calculations, therefore we think that there is a high optimization potential.

Limitations. Our current implementation has no mechanism to combine streets or crossings that are relatively near to each other. This can result in very small parcels being generated, which can cause overlaps when actual street geometry is generated. In future work this can probably be solved by introducing a routine to combine nearby objects after merging. Further, the minimal graph cut can sometimes exhibit unintuitive and erratic changes. Especially when the priorities of the two layers are very similarly small, layer movements can cause the minimal cut to jump around a few blocks. This can be observed in the accompanying video, when the layer is moved over an important region in the lower layer. In such a case, the hard merge can be more suitable.

7. Conclusion

This paper presents a city modeling system based on the concept of valid urban layouts. We show that three basic operations, in combination with a layering system, can express all important editing operations on urban layouts. The main advantage of our method is that editing operations like dragging, deletion and insertion, and merging of different layouts from arbitrary sources (procedural or hand-crafted) always produce a valid urban layout. This greatly reduces the cost of editing procedural cities. In the future, we want to add an error function to the flexible merging, with the goal of improving the quality of the graph cut result. Also, we would like to extend the system with convenient tools like merging or snapping to nearby objects.

Acknowledgements. This research was supported by the Austrian FIT-IT Visual Computing initiative, project GAMEWORLD (no. 813387), and by the NSF, contract nos. IIS 0915990, CCF 0643822, and IIS 0757623.

References

[ABVA08] ALIAGA D. G., BENEŠ B., VANEGAS C. A., ANDRYSCO N.: Interactive reconfiguration of urban layouts. *IEEE Comput. Graph. Appl. 28*, 3 (2008), 38–47. 3

[AVB08] ALIAGA D. G., VANEGAS C. A., BENEŠ B.: Interactive example-based urban layout synthesis. *ACM Trans. Graph. 27* (2008), 160:1–160:10. 2, 3

[Cac09] CACCIOLA F.: 2D straight skeleton and polygon offsetting. In *CGAL User and Ref. Manual*, 3.5 ed. CGAL Editorial Board, 2009. 4

[CEW*08] CHEN G., ESCH G., WONKA P., MÜLLER P., ZHANG E.: Interactive procedural street modeling. *ACM Trans. Graph. 27* (2008), 103:1–103:10. 2, 3

[CLDD09] CABRAL M., LEFEBVRE S., DACHSBACHER C., DRETTAKIS G.: Structure-preserving reshape for textured architectural scenes. *Computer Graphics Forum 28*, 2 (2009), 469–480. 3

[EK72] EDMONDS J., KARP R.: Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM 19*, 2 (1972), 248–264. 6

[FF62] FORD L., FULKERSON D.: *Flows in Networks*. Princeton University Press, 1962. 3, 4

[GSMCO09] GAL R., SORKINE O., MITRA N. J., COHEN-OR D.: iwires: an analyze-and-edit approach to shape manipulation. *ACM Trans. Graph. 28* (2009), 33:1–33:10. 3

[HF06] HORMANN K., FLOATER M. S.: Mean value coordinates for arbitrary planar polygons. *ACM Trans. Graph. 25*, 4 (2006), 1424–1441. 4

[KM07] KELLY G., MCCABE H.: Citygen: An interactive system for procedural city generation. In *Fifth International Conference on Game Design and Technology* (2007), pp. 8–16. 2, 3

[KSE*03] KWATRA V., SCHÖDL A., ESSA I. A., TURK G., BOBICK A. F.: Graphcut textures: image and video synthesis using graph cuts. *ACM Trans. Graph. 22*, 3 (2003), 277–286. 3, 5

[LWW08] LIPP M., WONKA P., WIMMER M.: Interactive visual editing of grammars for procedural architecture. *ACM Trans. Graph 27*, 3 (2008), 102:1–10. 1, 2, 3

[MWH*06] MÜLLER P., WONKA P., HAEGLER S., ULMER A., GOOL L. V.: Procedural modeling of buildings. *ACM Trans. Graph. 25* (2006), 614–623. 2

[PL91] PRUSINKIEWICZ P., LINDENMAYER A.: *The Algorithmic Beauty of Plants*. Springer Verlag, 1991. 2

[PM01] PARISH Y. I. H., MÜLLER P.: Procedural modeling of cities. In *Proc. of ACM SIGGRAPH 2001* (2001), Fiume E., (Ed.), ACM Press, pp. 301–308. 2, 8

[Pro10] PROCEDURAL INC.: Cityengine, www.procedural.com, 2010. 1, 2, 4, 9

[SS08] SCHMIDT R., SINGH K.: Sketch-based procedural surface modeling and compositing using Surface Trees. *Computer Graphics Forum 27*, 2 (2008), 321–330. Proceedings of EG 2008. 3

[VABW09] VANEGAS C. A., ALIAGA D. G., BENEŠ B., WADDELL P. A.: Interactive design of urban spaces using geometrical and behavioral modeling. *ACM Trans. Graph. 28* (2009), 111:1–111:10. 2, 3

[VAW*10] VANEGAS C., ALIAGA D., WONKA P., MÜLLER P., WADDELL P., WATSON B.: Modeling the appearance and behavior of urban spaces. *Computer Graphics Forum 29*, 1 (2010), 25–42. 2

[WMWG09] WEBER B., MÜLLER P., WONKA P., GROSS M. H.: Interactive geometric simulation of 4d cities. *Computer Graphics Forum 28*, 2 (2009), 481–492. 1, 2, 3, 4

[ZHW*06] ZHOU K., HUANG X., WANG X., TONG Y., DESBRUN M., GUO B., SHUM H.: Mesh quilting for geometric texture synthesis. *ACM Trans. Graph. 25* (2006), 690–697. 3

[Zim07] ZIMMERMANN M.: *Procedural Construction of Streets*. Tech. rep., ETH, 2007. 2