Vectorization of Gridded Urban Land Use Data

Chris Sexton
Johns Hopkins University Applied Physics Lab
cgsexton@gmail.com

Benjamin Watson
North Carolina State University
bwatson@ncsu.edu

ABSTRACT

In the digital entertainment industry, cities are one of the largest artifacts modeled by artists. One alternative to modeling an entire city by hand is to use an urban simulation. Often, those simulations use a gridded terrain representation. Translating gridded simulation results into a more continuous, realistic representation useful in games and film can often be difficult. Our vectorization process transforms gridded urban land use data into a representation useful in entertainment pipelines and many GIS or online mapping tools. The process has three major phases. In the first phase, the raster data is analyzed and the transportation layer is abstracted and filtered. Next, the city blocks are constructed from the raster data. Third, the blocks are subdivided and land use and density are assigned to each constructed parcel. The results are much smoother than the gridded input, but maintain the land use patterns of that input. We output these results in a GIS format readable by a wide range of modeling tools.

Categories and Subject Descriptors

I.3 [**Computer Graphics**]: Miscellaneous

1. INTRODUCTION

As gaming platform capabilities increase, generating compelling 3D content is becoming one of the most challenging problems facing the digital entertainment industry. The foremost example of this is the city, which is both extremely large and highly detailed. Urban synthesis addresses this problem with automation, creating and placing buildings, tracing roads, carving out urban zones and determining land use. Our existing urban synthesis tool [7] uses a gridded simulation to produce its output. This grid is quite adequate for simulation, but far too coarse for entertainment applications. In this paper, we describe a method for automatically vectorizing gridded urban simulation data.

Our gridded synthesis tool uses agent-based simulation to create land use patterns, distributing commercial, residential, industrial, road and other land uses across the cityscape. Agents act as builders, determining property lines and assigning each resulting parcel's use according to a value model. The output matches modern development patterns well, both statistically and visually [7]. But as a modeling environment, realism cannot be our tool's only goal: it must allow artists to shape the city to meet their applied needs, which are often unrealistic by design. Thus users can input partial cities and let the tool fill in the blanks, or give direction to the simulation by manipulating local and global parameters.

The vectorization process we describe here moves through three stages: building a topologically connected transportation network, extracting and identifying city blocks, and subdivision of blocks into parcels. Vectorization requires at least three gridded layers as input: one describing land use, one describing usage density, and another describing ownership including property lines. Roads are disjoint if they are not four-connected (a grid element at location (x, y) is only four-connected to the elements at locations $(x + 1, y)$, $(x-1, y)$, $(x, y+1)$ and $(x, y-1)$). We output our vectorized results in ESRI format, a widely used GIS format that can be read by many modeling tools.

To understand why we are interested in this sort of data filtering and enhancement, it is important to have an overview of procedural modeling on the whole. We review procedural modeling, with a special focus on urban synthesis and image vectorization. We then describe our algorithms and show results, which compare well to real-life GIS data.

2. RELATED WORK

Procedural modeling is a term that refers to many different techniques in computer graphics used to automate the creation of models and textures. Today's hardware is increasingly powerful, allowing the use of more detailed models in rendering. Unfortunately, as model complexity increases, so does the amount of storage needed to store models and the amount of time needed to model complex objects. For example, for the movie King Kong, over 90,000 3D buildings were created, from over 22 million components [19]. Most of New York City was modeled procedurally, while artists focused on the landmark buildings. This allowed them to concentrate on the artistically important aspects of the city model, without having to be worried about the other 90,000 buildings that needed to be recreated for the film. In our area of research, procedural modeling is used for urban synthesis. For a good survey of urban procedural modeling, see [17] and [15]. We will only review the most directly related work.

2.1 Urban Synthesis

Perhaps the first example of urban synthesis came from Stiny [13], who published an article in Environment & Planning (B) that analyzed Palladio's system of architecture and derived a set of rules describing it. They organized these rules into a parametric shape grammar that generates ground plans, and used it to generate Palladio's Villa Malcontenta. Many architectural synthesis techniques make use of grammars derived from Stiny's.

In 2001, Parish and Müller [11] presented research using L-systems to model cities. Their system takes as input images representing elevation, population density and road constraints, and generates highways and streets, subdivides land, then creates some simple building geometry. Later, Wonka et al. [20] improved on this work with split grammars, which are capable of generating buildings facades with a wide range of styles and designs. Müller et al. [10] then derived *CGA Shape*, a grammar that permits creation of entire building exteriors in very high detail.

More recently, Chen et al. [3] use tensor fields to represent prevailing urban orientations, allowing users both local and global interactions with the field to modify street networks. Unlike Parish and Müller [11], these interactions provide extensive user control. Aliaga et al. [1] described methods for generating urban layouts from street network data and snippets of aerial-view imagery. In 2009, Vanegas et al. [14] extended prior work to infer urban layouts by using UrbanSim [16], one of the leading urban simulations developed by geographers.

Simulations developed to meet the urban synthesis needs of the entertainment and related industries differ from geographic simulations in three primary respects. First, while geographic simulations emphasize accuracy and prediction, synthesizing simulations are essentially modeling tools. As such, they are interactive, steerable, and often used to produce completely fictional (if convincing) environments. Second, as tools for understanding and anticipating urban processes, geographic simulations need not simulate cities in any great spatial detail. In contrast, synthesizing simulations are used to produce rich visuals, and therefore simulate at a much finer spatial scale. Finally, real-world accuracy and understanding in geographic simulation typically require extensive and highly detailed input. Synthesizing simulations may be modeling completely fictional environments, and are meant to amplify human input, not simply transform it. They therefore require much less from the user, if indeed they require anything at all. Our own synthesizing simulation [7] is an excellent example of these differences: it is highly steerable, simulates down to a 12 meter scale, and can produce results without any input whatsoever. Weber et al.'s [18] simulation has many similarities to our own, but works on a vector rather than a gridded substrate. It does not require vectorization, but like vector displays, slows as output complexity grows.

2.2 Vectorization

The subfield of computer graphics that may be most closely related to our research is image vectorization, which transforms image rasters into sets of continuous edges and curves. Our vectorization process differs from image vectorization by exploiting its noiseless input and knowledge of urban content. For a good summary of research on vectorization of rasterized line drawings, see [8].

The skeletonization of binary images by image thinning is a fundamental image processing technique, one we use to assist in subdivision of city blocks into parcels. Zhang and Suen [21] described a simple, efficient method that produces fairly low quality skeletons. Improving this quality requires significant additional computation. Fan et al. [6] traverse a sparsely sampled version of the input image, creating an orthogonally zigzagging set of connected bars, which are used to generate a skeleton. Dori and Liu [5] extended this idea to use medial axis points, improving quality further. Zou and Yan [22] instead used constrained Delaunay triangulation, raising quality at skeletal joints.

These vectorization techniques have found wide use in urban applications. Chiang et al. [4] matched rasterized to vectorized maps by comparing intersection "fingerprints" extracted from the maps. Carrozzino et al. [2] and Mena [9] both vectorize road imagery. They use the high resolution of their input to compensate for noise. Because our input is noise-free, we can work effectively with much lower resolution input.

3. OUR URBAN GRID VECTORIZATION

3.1 Input Blocks and Parcels

We have two basic forms of input data from the city simulation: gridded layers and a per-parcel list. The three layers describe land use, usage density and ownership. Each patch (grid cell) in the land use layer indicates one of the following types: unused; water; secondary or primary road; residential, commercial, industrial, park or reserved uses. In the density layer, patches contain a population level. Each patch in the ownership layer contains a parcel ID. Each entry in the per-parcel list describes usage density, land use type, and member patches. The land use data can be seen in Figure 1.

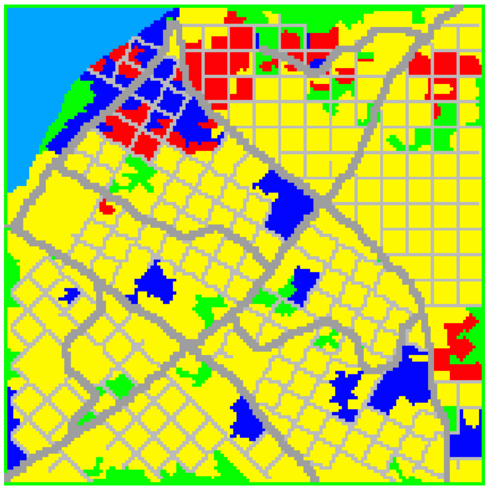

Figure 1: View of land use data from simulation

After loading the land use data into a simple 2D array, we create blocks. There are two different types of blocks: internal blocks are completely surrounded by existing roads, while external blocks are not. External blocks are typically found on the edge of the city, where development continues (or has been constrained by the end of the simulated grid).

Internal blocks are made up of patches that are not part of roads or bodies of water. We construct them using a "flood fill": beginning with a seed patch, we visit all patches adjacent to the partially completed block and add them to the block if their type is appropriate. (Note that two or more blocks may be surrounded by the same roads when water or reserved land is also surrounded by these roads). We compute external blocks similarly, but also exclude undeveloped land. We associate simulated parcels with blocks by traversing the parcel list, and for each parcel, locating the block that contains the first patch in each parcel.

3.2 Vectorizing Roads

Having identified the grid cells that make up blocks, we begin building vectorized roads and blocks. We first create an undirected graph that represents the road network. We link graph nodes if the corresponding road grid cells are four-connected, then smooth the resulting network by averaging connected graph neighbors. We next connect primary and secondary road segments and eliminate overly short graph cycles. After additional smoothing, we define road curbs and intersections, and place circles on culs-de-sac.

3.2.1 *Defining Road and External Block Topology*

The simulated input may have directional ambiguity at primary-secondary road intersections. We eliminate redundant connections between secondary and primary roads to fix these ambiguities. From each secondary road patch adjacent to a primary road, we visit all four-connected secondary patches, halting at any primary-disconnected patches. In this case a patch is primary-disconnected if it is neither four- nor eight-connected to a primary road patch (a grid element (x, y) is eight-connected to all of its four-connected neighbors, as well as the elements at locations $(x+1, y+1)$, $(x+1, y-1)$, $(x-1, y+1)$ and $(x-1, y-1)$). If we do not reach any primary-disconnected patches, then all visited patches are directly adjacent to primary road and may be deleted because they go only where the primary road already goes. If we do reach primary-disconnected patches, then for each disconnect we save only one path back to the primary road. The patches in this path are in the shortest path between the disconnected patch and the closest patch that is four-connected to primary road. We remove all visited patches not in one of these shortest paths and give them a type that indicates they were once road but are no longer. This can be seen in the green squares in the bottom of Figure 2.

We next build an undirected graph from the road patches found in the input data. We begin by locating the graph nodes. We iterate over all patches in the data set, treating primary and secondary road patches differently, because primary and secondary roads have different widths (secondary roads are one patch wide, primary width are two patches wide). We locate the graph node for secondary road patches at the patch center. Primary road patches receive a graph node only if their east, southeast, and south neighbor patches are also primary roads, effectively creating primary road graph nodes only when the corresponding patch is at the center of the wider primary road. Primary graph nodes are located at the patch's southeast corner.

We now revisit the patches in the simulation data to connect the graph nodes. A road patch's node is linked to the node corresponding to any four-connected road patch. Note that many primary road patches will not have a corresponding graph node, and will therefore not become part of the road graph. At this stage in the process, because graph nodes are located on a grid and are four-connected, links are oriented in one of the four cardinal directions. The upper left of Figure 2 shows an example.

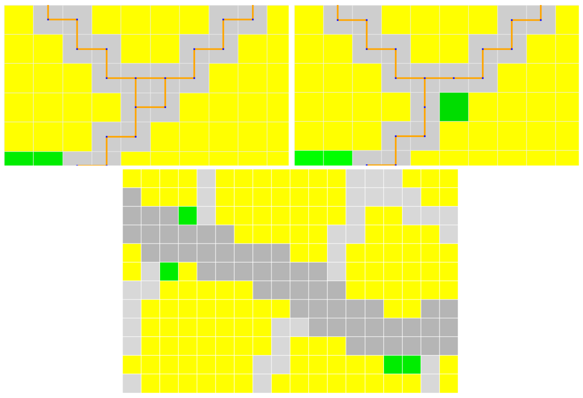
Figure 2: Elimination of unclear road patches

To eliminate very short graph cycles in the secondary roads and reclaim single patches of land, we iterate through all secondary graph nodes, attempting to follow links in the south, east, north and west directions. If we are able to follow all of these links, we know there is a short cycle in the graph, representing four patches organized into a 2x2 square. To eliminate it, we find the patch four-connected to two patches that aren't road, remove it from the graph, and mark the patch as removed road. This stage can be seen in the upper right of Figure 2. Cycles in primary roads and other larger clusters of connections are handled in a later stage by condensing clusters through determining input and output nodes and centroiding the loops. These larger clusters of loops do not lend themselves well to returning land for development.

We now examine all graph nodes to identify intersections, which are defined as nodes having more than two links, or linked only to other intersections. Having found these points, we then replace clusters of adjacent intersections with one simplified intersection, unless the intersection node would only be connected to two other nodes (See Figure 3). We find the location for simplified intersections by averaging the location of the unsimplified adjacent intersections.

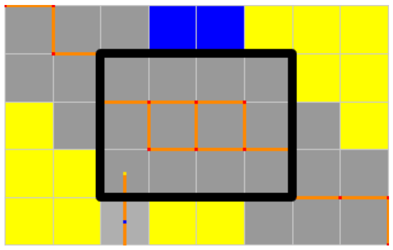
Figure 3: Cluster of loops that don't represent an intersection

We now enter a logistical and organizational phase. Until this point, the graph was represented using a (partially filled) four-connected lattice corresponding to the simulation grid, enabling us to use techniques dependent on cardinal direction or grid adjacency. We now discard this representation in favor of the more traditional, unconstrained and efficient representation that contains only those links a node actually possesses. We also look for linked graph nodes that are more than two patch widths away from each other, eliminating such long links by inserting nodes until all resulting links are shorter than two patch widths. These distant nodes can occur when intersections are condensed and other cycles are removed from the network. This ensures that all nodes have roughly the same weight in the filtering process described below. Finally, we place every node's links into a clockwise ordering from due north. This step supports inferences about turning direction.

While roads form the perimeters for most blocks, they do not for external blocks. We now construct external perimeters for any external blocks found earlier. Such perimeters will have undeveloped land on one side, and developed land on the other. We therefore gather a connected list of all patch vertices that bound an undeveloped patch. Two vertices in this list are connected only if the edge between them is undeveloped on one side and undeveloped on the other. To build perimeters, we first search for a vertex in our list that only has one point connected to it (such points are located where the perimeter meets the road). Starting from this point, we walk the connectivity in our list, inserting visited vertices into our perimeter list and removing them from our bounding list. When connectivity ceases and the other side of the perimeter is reached, we check the bounding list, and if it is not empty, begin working on the next perimeter. We convert the resulting perimeter lines to a series of nodes and links connected to our road network. Finally, we are left with a network ready for manipulation.

We now begin a number of steps designed to reposition roads to improve appearance. We start by straightening intersections, which due to the input simulation grid have connecting roads that meander heavily. At each intersection, for each pair of incoming roads, we evaluate their fit to a line. If that fit is reasonably good, we snap road nodes near the intersection to the fitted line. We use a linear regression on nodes in the pair that are at most five patch widths from the intersection. We define "reasonably good" to be a residual norm of 1.2 or less (an experimentally determined value that is slightly over one patch width). For an example of intersection straightening, see Figure 4.

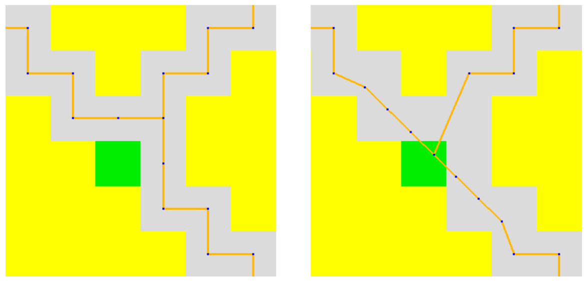

Figure 4: Intersection straightening (Left: before, Right: after)

3.2.2 *Merging and Smoothing Road Graphs*

To smooth roads, we average the locations of connected road nodes. We do not initially smooth nodes at dead ends, intersections or primary/secondary road connections. Averaging only includes nodes of the same road type, and only occurs when there are at least two linked nodes. Having repositioned roads outside of intersections, we straighten intersections again. We calculate the dot product between all pairs of incoming roads, and when the angle between the pair approaches 180 degrees, we simply place the intersection on the line between the two closest nodes on the incoming road pair. An example of how smoothing operates on the network can be seen in Figure 5.

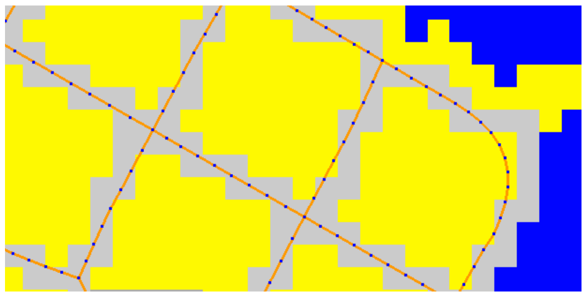

Figure 5: Road smoothing and intersection straightening

Until this point, primary and secondary road graphs have been completely disjoint. This is a natural consequence of their separate land use types in simulation output. Having connected, cleaned and smoothed both graphs separately, we merge them into one graph. Nodes that must be merged correspond to secondary road patches that are four-connected to a primary patch. We merge these nodes by linking them with the closest node in the primary network. After all such merges, we simplify any resulting intersection clusters, as described above. The end result of this pass can be seen in Figure 6.

We use a similar process to connect external block perimeters to the road graph, producing a complete partitioning of the map into blocks. In this case, perimeter ends are connected to any road node not already connected to a perimeter.

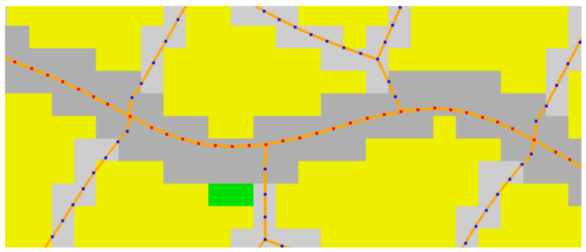

Figure 6: After connecting secondary and primary

Simulation, cleaning and smoothing sometimes produce short road segments between intersections that enclose undeveloped land that is awkward to develop. We eliminate these by locating cycles in the road graph less than 15 nodes or less, and removing one segment.

We now simplify overly complex intersections to avoid problems when we eventually add road geometry such as curbs. We remove any dead ends connected directly to the intersection and angled within 90 degrees of another road. We also remove any nodes that are extremely close to the intersection. Primary nodes are removed if they are within one patch width of one another, secondary nodes if they are within 1/2 patch width.

We now smooth the road graph further (eight smoothing passes seems to work well for most simulation output). To avoid over-smoothing of high frequency features, we add nodes where curvature is high. If the angle defined by three consecutive nodes outside an intersection is less than 90 degrees, we add three nodes on each side of the central node. If the angle is less than 135 degrees, we add two nodes per side, and if it is less than 160 degrees, we add one per side.

3.2.3 *Defining Curbs*

We now add curbs, pavement and culs-de-sac to the road graph. (We leave external block perimeters unchanged). Primary curbs are one patch width away from the graph nodes, while secondary curbs are half a patch width away. Curbs are oriented parallel to the graph links in their segments, and pavement is a triangle strip using curb vertices as its vertices. At dead ends on secondary road segments, we add octagonal culs-de-sac.

Road segments that terminate at intersections require more complex processing that ensures that their pavement does not overlap. Processing proceeds by considering curbs attached to each adjacent pair of road segments, when an intersection's road segments are ordered using a consistent clockwise winding. The result of processing each pair is a curb corner. Each road segment in the current pair has two parallel curbs. Each such curb intersects both of the parallel curbs from the other road segment, forming four candidate corners. We find the correct corner by inserting all candidates into the intersection's winding order, and choosing the corner between the current road segment pair. Note that when the segments in a pair are parallel or nearly so, no corner is actually required, and in fact the floating-point calculation of any curb intersection is unreliable. In such cases we simply add an edge between the ending vertices of two of the road segment curbs. We identify the two curbs to connect in this way by inserting all four curbs for the pair into the intersection's winding order. The two curbs to connect will both be located between the current road segment pair.

With these corners we can trim our road segment triangle strips, and construct a convex intersection polygon. Road segment pavement terminates with an edge between the corners of each of its curbs. Intersection pavement is represented by a convex polygon constructed from the corner vertices, in clockwise intersection order. Figure 7 shows road geometry.

3.3 Vectorizing Blocks

We find block boundaries by tracking surrounding curb loops or external boundaries. At some locations, boundaries will include parcel or water edges. We then identify oblong or "pierced" blocks that would benefit from internal spines used during subdivision into parcels. Oblong blocks have oriented bounding boxes with aspect ratios less than 1:2. "Pierced" blocks have a cul-de-sac in their boundary, and should have parcel boundaries radiating from the cul-de-sac circle into a spine.

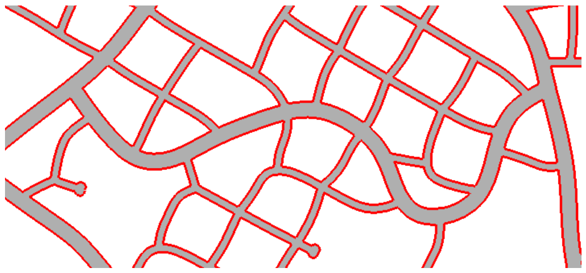
Figure 7: Road geometry with road edges highlighted

3.3.1 *Outlining Blocks*

Vectorized blocks match the gridded blocks we extracted from the input simulation results. To find all of our internal blocks, we follow road edges or links in a consistent direction (e.g. turning right for a clockwise order). Each time we reach our starting point, we enclose a new block. One resulting "block" is in fact the entire outer boundary of our road network. We save this perimeter for later use, but discard it from our current list of vectorized blocks.

Blocks containing water should not include that water. We therefore split these blocks using water boundaries. We identify simulation blocks containing water, identify the corresponding vectorized block, and insert a smoothed version of the water boundary in the simulated block into the vectorized block.

We vectorize external blocks using the external block boundaries generated earlier, and the external road perimeter that we saved during the creation of internal city blocks. We begin by reconnecting the two endpoints of each external block boundary back to the external road perimeter. We then transform each such boundary into a completely closed block boundary by exploiting the perimeter's clockwise ordering (recall that like internal blocks, the perimeter lists road nodes in clockwise order). Vectorized external block boundaries begin at the lower-numbered (in the clockwise perimeter order) end of the saved portion of the external block boundary, connect back to the road perimeter, and are then closed by following the perimeter in counter-clockwise order. If an external block has two or more saved external boundary lengths (meaning there is undeveloped land bounded by the road perimeter and the external block), then it is followed rather than the road perimeter. The external block is completely enclosed when we reach the starting point of our traversal.

Road smoothing at particularly sharp turns can leave some external blocks inside the smoothed road perimeter. We identify such blocks by finding those with boundary lines with both end points inside the road perimeter and with more than 50% of nodes inside perimeter, and effectively delete the corresponding blocks by removing these boundary lines before the corresponding external blocks are vectorized.

To associate all newly generated vectorized blocks with the simulation blocks, we calculate the center of each simulation block. If the center is inside the simulation block, we look for the vectorized block that contains the center and associate it with the simuGlation block. If the center isn't actually in the block (e.g. the block is highly concave), we instead use the parcel center that is closest to the block center, ensuring that that the center is actually inside the block. If there are any simulation blocks that aren't associated with generated vectorized block, we assign them to the vectorized block that has the center closest to the center of the simulation block. We do not often make use of this failsafe. Note that blocks containing water boundaries will already have simulation-vector pairings, making the matching operations here unnecessary.

3.3.2 Finding Block Spines

When city blocks are more rectangular than square, parcel boundaries often form a "spine" aligned with the long axis of the block. These spines ensure efficient and profitable allocation of property. We define such "oblong" blocks as those having an aspect ratio less than 1:2. We compute the aspect ratio using a bounding box oriented to fit the block tightly using principal component analysis. In addition, any block containing a cul-de-sac receives a spine: such dead ends are only inserted into blocks when interior parcels need access to the road network, and a spine ensures that these interior parcels will be created. We do not create spines on any exterior block, because they would make the parcels on the exterior side of the parcel inaccessible.

We build spines by making a binary image approximation of the block, performing constrained Zhang/Suen image thinning [21], and then converting the image back into a vectorized representation. (Other thinning methods would generate slightly improved spines, but they are much more complex and less efficient). To convert the block into a binary image, we rasterize the block's axis-aligned bounding box using eight pixels per patch. If a pixel is inside our block's polygon, it is set to black, otherwise it is set to white. After thinning, we re-vectorize the result by adding edges between all four-connected spine pixels. In any remaining points that have one or fewer edges, we add edges between any eight-connected neighbors. We then smooth the resulting vectorized spine by averaging vertices with their neighbors, prune any short branches off the primary spine that are less than two edges in length, and trim any spine ends that are within one patch width of a block edge. Finally, we make the spine look more like a manmade, piecewise-linear polyline using simplification. This process traverses the spine, summing edge-to-edge turning angles at each vertex. When this sum exceeds 20 degrees (an empirically determined threshold), we use an edge to replace all the vertices between the vertex at which summing began, and the vertex at which the threshold was exceeded. To make parcel simplification easier (see below), we ensure that spine vertices containing more than two edges are not directly connected by a single edge.

3.4 Describing Parcels

To create parcels of ownership and assign use and density, we first match each vectorized block to a simulated block. We then subdivide each block to produce the same number of parcels as the simulated block, using a constrained and randomized binary space partitioning. We assign parcel use and density with a similarly randomized and constrained process that references the matching simulation block. We generate 50 such parcelings, and retain only the parceling that matches the simulation most closely from the simulation. We then export the data to an ESRI shapefile that can be read by various GIS and modeling tools.

3.4.1 Generating Parcels

Urban blocks are divided into separate parcels. Accordingly, we must divide the vectorized blocks into parcels. If a vectorized block has a spine, we start by comparing the number of spine ends to the number of simulated parcels. If there are more parcels than ends, spine-based parceling will create too many parcels, and we discard the spine. Otherwise, we add parcel boundaries between each end and the closest point on the vectorized block. Figure 11 shows the results. If the number of vectorized parcels is still less than the number of parcels in the corresponding simulated block, we subdivide further. To match the vectorized subdivision as closely as possible to the simulation's subdivision, we generate many (currently 50) randomized subdivisions, and pick the subdi- vision that matches the simulation subdivision most closely. The simulation grid's resolution makes its parcel boundaries extremely coarse, and only loosely defines the target subdivision.

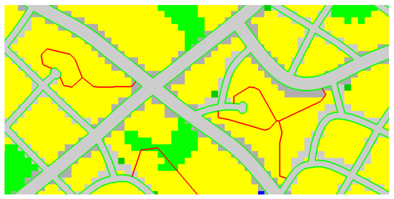

Figure 8: Example of generated spines for blocks

Further subdivision begins by making a copy of the block. We repeatedly split the largest remaining parcel in a constrained, random fashion, continuing until we produce the number of parcels in the simulated block. Each split creates two new parcels with a new parcel edge that splits and is perpendicular to a road-facing edge. We choose the road-facing edge randomly and place the new parcel edge at a random location biased toward the center of the road-facing edge using a normal distribution. To avoid inappropriate splits caused by local perturbations in the block boundary (which contains edges with length of roughly one patch width or less), we simplify the boundary of the block's copy, and perform subdivision on that simplified block. We then transfer each split to the original vectorized block, where it may not be perpendicular to the road-facing edge it intersects. We evaluate this split against the following constraints:

1. Minimum parcel size: each new parcel must have at least 75% of the area of the minimum parcel size in the simulated block.

2. Road access: each parcel must have a road facing edge that is at least 0.5 patch widths in length.

3. Limited rectangularity: if the original parcel had an aspect ratio greater than 0.25 (not too rectangular), the two generated parcels must have an aspect ratio that is at least as large (not more rectangular). If the original parcel has an aspect ratio less than 0.25 (too rectangular), both of the two generated parcels must increase the aspect ratio (be less rectangular).

4. Limited variation in residential parcel size: If residential land use fills 70% of the block, parcels must be roughly equal in size. Neither parcel may have less than 30% of the original parcel's area.

If the new parcels meet these constraints, we allow the split and attempt the next. If we cannot generate an acceptable split of the largest parcel after multiple attempts, we attempt to split a random parcel.

Because the simulation does not explicitly output undeveloped block parcels, we will have to subdivide some vectorized blocks into more parcels than the simulation explicitly indicates, in order to create these undeveloped "negative" parcels. We define these undeveloped parcels as being parcels that have no land use and are available for development. If the proportion of undeveloped land in a block is more than 25%, we mark it for additional subdivision and parceling. We could determine the number of additional parcels to create by counting the number of disjoint regions of undeveloped land in the simulated block, however many of these regions are too small to bother recreating. Our current heuristic is to double the number of undeveloped parcels already created in the vectorized block, after the initial round of land use assignments. The added parcels provide enough additional granularity for our vectorizer to maintain the overall proportion of developed to undeveloped land (see land use assignment below).

3.4.2 Filling Parcels

We assign uses to newly subdivided blocks by attempting to reproduce the distribution of uses in the corresponding simulated block. In a common trivial case, all parcels in the simulated block have the same land use. If such blocks are also at least 75% developed, we simply assign all parcels in the vectorized block the same use. Otherwise, we assign land use stochastically, using a distribution based on differences between the vectorized block and the corresponding simulation block. In particular, the distribution is shaped by distance between parcel centers, proportions of use in each block and differences between parcel sizes. More detail can be found in [12]. We show examples of land use assignment in Figures 9, 10, and 11.

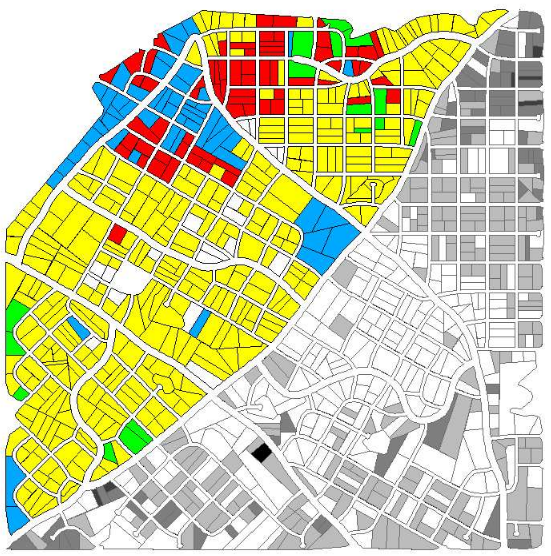

Figure 9: Example of land use and density assignment (yellow = residential, blue = industrial, red = commercial, green = park, white = undeveloped, dark is higher density). Output corresponds to Figure 1. Upper half is land use, lower half is density.

4. DISCUSSION AND CONCLUSION

The computer graphics industry is facing a daunting content challenge, especially in urban content. This paper described a method for transforming gridded urban simulation output into a vectorized format. We believe this translation is successful, as we hope Figures 1, 9, 10, and 11 demonstrate. Compare especially Figures 1 and 9, which show input and result. The vectorization smooths while preserving the essence of the urban distribution.

We demonstrated our vectorizer with a specific simulation, but designed it to be easily generalized to other simulations, assuming only that there are several types of land use, that road connectivity is indicated by grid adjacency, and that grid cells are grouped into parcels. To date, there are few gridded simulations beside our own with the fine spatial detail needed by urban synthesis applications and our vectorizer. One such may be UrbanSim [16], which others have used for urban synthesis [14]. As computation becomes cheaper and more powerful, we expect many other gridded urban geographic simulations to gain the resolution required by urban synthesis.

Weber et al.'s [18] simulation is most similar to our own, and because it combines simulation and vectorizer into one application, it runs more quickly than our own separate applications. We believe that interactivity in the two systems is comparable, though fundamentally different, with our system offering interaction both on the simulation grid itself, and on the vectorized result; while Weber et al. have no grid. As both a computational and interactive substrate, the grid may offer advantages over vectors in the long run, with the potential of GPU acceleration and an intuitive painting interaction. Our decoupling of vectorizer and simulation also allows the use of different simulators.

Future work should explore these possibilities, and address the limitations of our vectorizer. Each of its stages may be reparameterized and repeated as needed; but interactivity could be improved with undo, and with stage reordering (when algorithmic constraints allow). Especially in older urban neighborhoods, real-world parcel size can vary widely. It may be necessary to adjust the parceling algorithm in our vectorizer to reflect this.

5. ACKNOWLEDGMENTS

This research was supported by NSF grant 0326542.

6. REFERENCES

[1] D. G. Aliaga, C. A. Vanegas, and B. Benes. Interactive example-based urban layout synthesis. In *SIGGRAPH Asia '08: ACM SIGGRAPH Asia 2008 papers*, pages 1–10, New York, NY, USA, 2008. ACM.

[2] M. Carrozzino, F. Tecchia, and M. Bergamasco. Urban procedural modeling for real-time rendering. In *3D-ARCH 2009: "3D Virtual Reconstruction and Visualization of Complex Architectures"*. ISPRS, 2009.

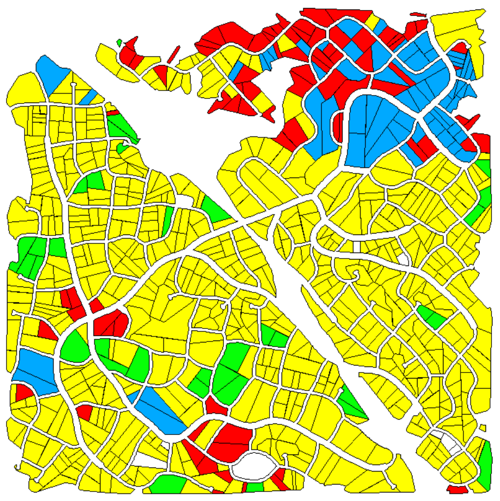

Figure 10: A second example of vectorized land use

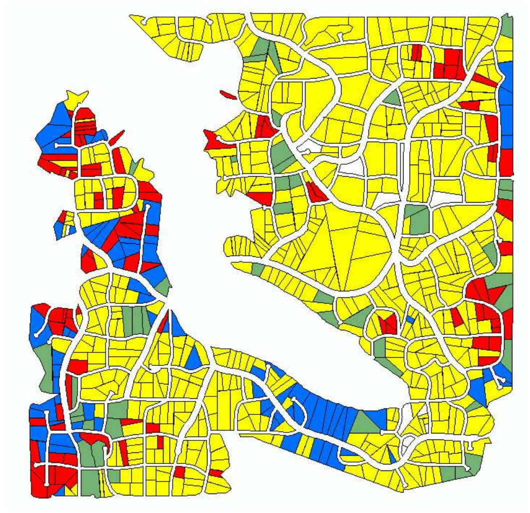

Figure 11: A third example of vectorized land use

[3] G. Chen, G. Esch, P. Wonka, P. Müller, and E. Zhang. Interactive procedural street modeling. In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, pages 1–10, New York, NY, USA, 2008. ACM.

[4] Y.-Y. Chiang, C. A. Knoblock, and C.-C. Chen. Automatic extraction of road intersections from raster maps. In *GIS '05: Proceedings of the 13th annual ACM international workshop on Geographic information systems*, pages 267–276, New York, NY, USA, 2005. ACM Press.

[5] D. Dori and W. Liu. Sparse pixel vectorization: An algorithm and its performance evaluation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 21(3):202–215, 1999.

[6] K.-C. Fan, D.-F. Chen, and M.-G. Wen. A new vectorization-based approach to the skeletonization of binary images. *ICDAR*, 02:627, 1995.

[7] T. Lechner, B. Watson, U. Wilensky, S. Tisue, M. Felsen, A. Moddrell, P. Ren, and C. Brozefsky. Procedural modeling of urban land use. Technical Report TR-2007-33, North Carolina State University Department of Computer Science, Raleigh, NC, October 2007.

[8] W. Lieu and D. Dori. From rasters to vectors: Extracting visual information from line drawings. *PAA*, 2(1):10–21, 1999.

[9] J. B. Mena. Automatic vectorization of segmented road networks by geometrical and topological analysis of high resolution binary images. *Know.-Based Syst.*, 19(8):704–718, 2006.

[10] P. Müller, P. Wonka, S. Haegler, A. Ulmer, and L. V. Gool. Procedural modeling of buildings. In *Proceedings of ACM SIGGRAPH 2006 / ACM Transactions on Graphics*, volume 25, pages 614–623, New York, NY, USA, 2006. ACM Press.

[11] Y. I. H. Parish and P. Müller. Procedural modeling of cities. In E. Fiume, editor, *Proceedings of ACM SIGGRAPH 2001*, pages 301–308, New York, NY, USA, 2001. ACM Press.

[12] C. G. Sexton. Vectorization of Gridded Urban Land Use Data. Master's thesis, North Carolina State University, November 2007.

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

[14] C. A. Vanegas, D. G. Aliaga, B. Benes, and P. Waddell. Visualization of simulated urban spaces: Inferring parameterized generation of streets, parcels, and aerial imagery. *IEEE Transactions on Vis and Computer Graphics*, 15(3):424–435, 2009.

[15] C. A. Vanegas, D. G. Aliaga, P. Wonka, P. Müller, P. Waddell, and B. Watson. Modeling the appearance and behavior of urban spaces. In *State of the Art Reports, EUROGRAPHICS 2009*, pages 1–16. EUROGRAPHICS, 2009.

[16] P. Waddell. Urbansim: Modeling urban development for land use, transportation, and environment planning. *Journal of the American Planning Association*, 68(3):297–314, 2002.

[17] B. Watson, P. Müller, O. Veryovka, A. Fuller, P. Wonka, and C. Sexton. Procedural urban modeling in practice. *IEEE Computer Graphics and Applications*, 28(3):18–26, 2008.

[18] B. Weber, P. Müller, P. Wonka, and M. Gross. Interactive geometric simulation of 4d cities. *Computer Graphics Forum*, April 2009.

[19] C. White. King kong: the building of 1933 nyc. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Sketches*, page 96, New York, NY, USA, 2006. ACM Press.

[20] P. Wonka, M. Wimmer, F. Sillion, and W. Ribarsky. Instant architecture. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pages 669–677, New York, NY, USA, 2003. ACM Press.

[21] T. Y. Zhang and C. Y. Suen. A fast parallel algorithm for thinning digital patterns. *Commun. ACM*, 27(3):236–239, 1984.

[22] J. J. Zou and H. Yan. Cartoon image vectorization based on shape subdivision. In *CGI '01: Proceedings of the International Conference on Computer Graphics*, page 225, Washington, DC, USA, 2001. IEEE Computer Society.