The skylineEngine System

Remei Ridorsa - Gustavo Patow
Geometry and Graphics Group
Universitat de Girona
rridorsa@ima.udg.edu - dagush@ima.udg.edu

Abstract

In this paper we present **skyline**Engine , an urban procedural modelling tool developed as a testbed for new algorithms and techniques in urban modelling. In spite of being a starting open project, it has many features only available on high-end commercial modelling systems, like pattern-based district styling definitions, possibility to import city maps from images or from OpenStreetMap files, parameterizable models of cities and buildings, global city control through image maps (districts, landuse, height, etc.), and a user-friendly building modelling module based on shape grammars. This system also presents some novel features that make it a unique system, like a graph-based paradigm that allows the user to create content-rich cities with distinct districts, major and minor roads, blocks, lots and buildings, but also other urban elements like streets, sidewalks, parks, bridges and landmarks. Also, during its development we have developed new ways of generating urban content which increase the realism of the resulting environments.

1 Introduction

Procedural modelling is an umbrella term for a number of techniques in computer graphics to create 3D models from rules instead of constructing them from geometric primitives such as cubes and spheres.

Hence, procedural modelling focuses on the rule-based generation of semantic 3D content, rather than the cumbersome and complex manual editing of individual 3D primitives. As a consequence, procedural modelling is several times more efficient than traditional modelling.

In film, games, and other applications, consumers expect richer, higher-quality digital content. Because budgets won't allow content producers to increase cost significantly, producers have only one choice: they must improve their tools. One of the main drivers of these trends is urban content. Cities are huge, richly detailed artifacts often required in digital productions. Modelling them with existing tools can take hundreds of man months.

Nowadays, procedural geometric city modelling is an open problem which has no standard or open source solutions. This issue is quite important in many different sectors like architectonic and urban design, cinema and theater, virtual reality and, of course, the videogames industry. Within this, there appear several interesting sub-problems that must be faced. One of these is the modelling of streets, buildings and other architectonic structures.

Features: The **skyline**Engine system offers a variety of features currently only available at high-end urban modelling systems, like the commercial system CityEngine [10]:

- **skyline**Engine offers unique street pattern-based tools to design urban layouts. Street patterns such as grid, organic or circular are available.
- Real street networks of any city can be imported directly from images or from OpenStreetMap [12]. Furthermore, street

networks or lots designed in other programs can be imported via the **.obj** file format.

- Inside **skyline**Engine there is a building definition module, **building**Engine , based on a shape grammar implementation [10] but evolved to a more user-friendly visual environment. The graph-based paradigm is specialized for architectural 3D content and offers modelling possibilities to control or vary mass, elements, proportions, rhythms or materials.

- The parameters of the many buildings and streets can be controlled globally via image maps (for example the building heights or the land-use distribution). This allows intuitive city modelling and quick changes.

Contributions: Although the **skyline**Engine system is under heavy development, we have been able to identify and solve several open issues in the state-of-the-art literature, namely:

- Contrary to the accepted standard procedural city development models that assume a linear pipeline, we have developed a non-linear pipeline as the only way to deal with a real-city heterogeneity. See Section 4.

- During development, we have identified and solved a few open problems that arise when trying to create cities, either real ones or in the style of existing ones:

  To allow the user to define arbitrary cities through adequate (and arbitrary) node connections, we have created a configurable and flexible node-based modelling system. Also, we have developed a new method to subdivide blocks into lots in the style of European cities like Paris or Barcelona.

2 Previous Work

One of the first procedural techniques describing 3D city generation was introduced in [13], based on the idea of L-systems [15]. A similar approach for the template-based street network generation is that of [18] which also uses a rule-based rewriting system. Procedural methods have also been extended to the synthetic generation of buildings [23] [10] and to the creation of buildings and facades imitating real-world structures [11]. Nevertheless, these approaches require the explicit specification of a set of rewriting rules and/or the model is generated at once which limits editing. Chen et al. [2] proposed using tensor fields to guide the generation of street graphs. Recently, Lipp et al. [9] presented an interactive editing environment for procedural buildings, but their system presented some usability issues related to the fact that they continued using a text-oriented rule-based paradigm, while **skyline**Engine is based upon an interactive flow-oriented metaphor that is both simpler to use and to understand. For more information, we recommend the interested reader to the survey from Kelly and McCabe [8] or the more recent ones [21] [20].

3 Overview

In this section we aim at giving an overview of the whole **skyline**Engine system. As we will explain in the following section, **skyline**Engine is a user-configurable urban modelling system that, from very simple configuration files and global-control images, can produce content-rich cities with thousands of complex elements. See Figure 1. The core of **skyline**Engine is a non-fixed non-linear procedural node-based modelling system. See Figure 3. The flow is composed of *streams* of geometric elements and nodes that receive these *streams* and output new geometry after processing the input one, in a similar way as modern hardware uses shaders to process a stream of input vertices and fragments. The nodes are wired among themselves forming a network that is both flexible and powerful enough to represent almost all urban elements, from roads and streets to parks, buildings and landmarks.

In Figure 1 we can see this workflow in practice: from user provided land-use (left-top)

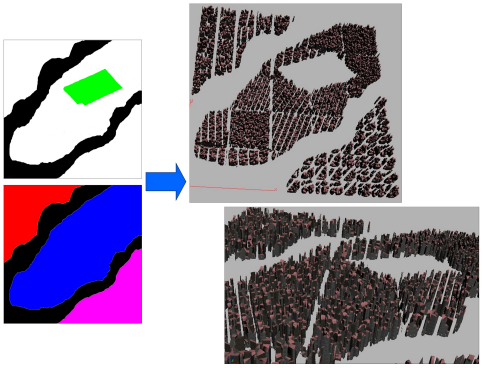

Figure 1: Disctrict-based modelling.

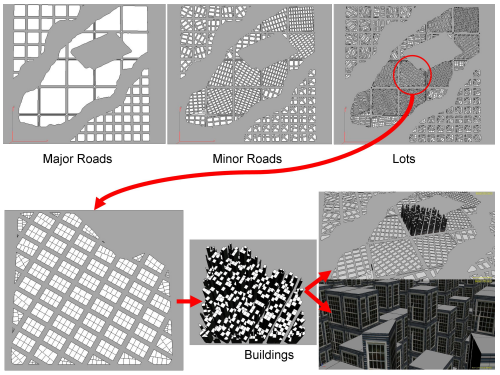

Figure 2: Data amplification: from two user-provided images (as shown in Figure 1), we obtain a major-roads distribution with 86 primitives (polygons). After minor roads are traced, the resulting geometry has 931 primitives, each representing a block. Finally, the subdivision into lots results in 7471 primitives, each one being a building lot. In the bottom row we can see the lots of a small neighborhood of the city. These lots, then, after the **building**Engine module, result in several buildings being created.

and district distribution (left-bottom) maps, plus some simple XML-based configuration files, the geometry for a Manhattan-like city is created.

4 Graph-based City Modelling

As described above, in the traditional literature there is a common acceptance of a five-stage pipeline for urban modelling, which is shown inside the blue box in Figure 3. However, practice shows that this model is insufficient when including a broader city model that includes other features like parks, landmarks, streets and others. Let's look at the example of urban parks. There are cities which have a park so large that it can be considered as a district by itself. This is the case of Central Park in Manhattan. See Figure 1. Other parks are much smaller, having sizes ranging from a few blocks to the space equivalent to a single building. Procedurally, all these parks can be treated with a single park module, but used at very different levels when considering the mentioned pipeline: large parks should be called directly from the district design phase, while smaller ones should be instantiated at the block or at the lot levels. This clearly breaks the linear pipeline model we have mentioned, generating a simple Directed Acyclic Graph (DAG), as shown by the module *Parks* and its links from the mentioned different city levels. Thus, **skyline**Engine presents a non-linear pipeline as the only way to deal with a real-city heterogeneity. See Figure 3. The idea of using a visual, graph-based paradigm is not new in Computer Graphics, starting from the pioneering work by Haeberli [7], and continuing with Maya's Hypergraph [6] or SideFX's Houdini [17].

5 Main skylineEngine Functional Modules

In this section we are going to introduce the different modules that we have implemented for **skyline**Engine . We start by explaining the main classes (**Network** and **s**ENode) that control the behavior of all other classes, followed by the input module and the respective files used to control the city creation process, and finally, we introduce some of the most important modules in the system.

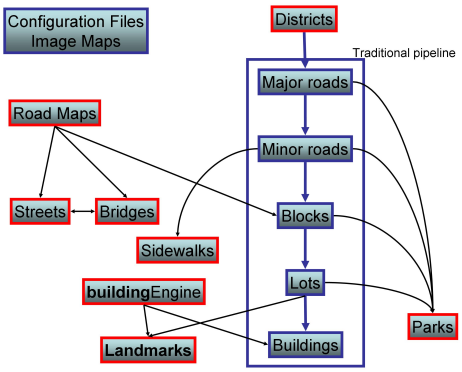

Figure 3: The non-linear **skylineEngine** workflow. In the blue box, the commonly accepted pipeline elements. In red the new, non-linear elements we have added for our system.

5.1 Network and sENode classes

The city creation mechanism in **skylineEngine** works around two key classes: the **Network** and the **sENode** classes. The first one has the role of instantiating the different modules that create the city (e.g.: Major roads, Blocks, Parks, etc.), connecting them to each other in such a way that the products by one module are readily processed by the following ones. All modules instantiated must come from classes that inherit from the abstract **sENode** class, which provides the basic functionality to interconnect nodes with a simple mechanism, allowing the creation of a *stream* of geometry from one node to its descendants. Both classes are loosely coupled, as **skylineEngine** uses a factory method based on a dictionary of pairs *(nameId, definition)*, where *nameId* is a name the author gave to one of his modules, and *definition* is a procedural description of the module instantiation mechanism. For most modules, as they are built with just one class, this simply amounts a call to a simple class constructor. When the author creates new modules, he/she only has to register them with **skylineEngine** and they will automatically be added to the dictionary and become available to any **Network** that may use them. This is specially useful for visual network creation, as the system only needs to add a visual interface to select the registered nodes, without needing further intervention from the module author.

We also created the **stdNetwork** class, which is a convenience class that builds a simple city in the most standard possible way, following the scheme given by the blue modules in Figure 3. To increase flexibility, it uses standard pre-defined names and accepts a simple list of associations with the actual node names to replace during construction. The standard names are: 'majorRoads', 'minorRoads', 'blocks', 'lots', 'buildings', plus some optional nodes to get statistical information like block/lot area or perimeter, or filter nodes to exclude geometry from evaluation (like blocks too small to be subdivided into lots). If a name does not exist, the system will automatically raise an exception. We have provided default nodes for most of them, but any implementation must, at least, provide 'majorRoads', 'minorRoads'. This is the case of the **patternNetwork** and **stdLoadMapNetwork** subclasses, which respectively instantiate the pattern-based mechanism already described and the direct loading of roads from a file. In this latter case, the map is created by a 'majorRoads" node and the 'minorRoads' is a pass-through null node given that the street division is already provided by the map. See Figure 4.

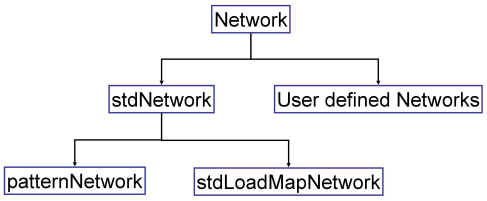

Figure 4: The standard **Network** class hierarchy.

5.2 Input Module

Configuration for **skylineEngine** is provided through rules encoded in an XML-based file format. These files start with a header identi-

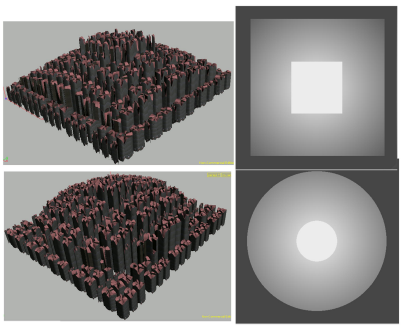

Figure 5: Height map.

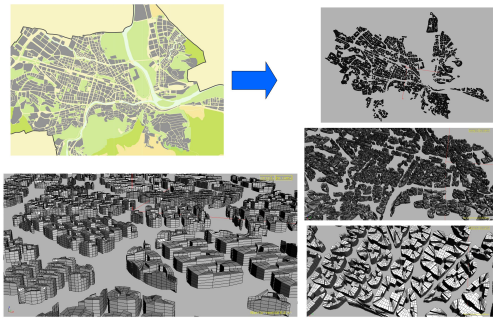

Figure 6: Direct input form a city map (Girona, Spain) to the corresponding mass models. Observe we have used our Paris-like lots subdivision mechanism.

fying them as skylineEngine files, and are followed by the overall city plan, which can be either a synthetic city or a real city. In the first case the rules in the file should inform whether is a single-pattern city (e.g. a regular grid like downtown Manhattan) or a multi-district city. See Figure 1. In the second case, a district map should be provided, and the XML file should contain a subsequent rule defining the urban characteristics for each district. Also, the user can optionally specify a land-use map telling the system the areas where there cannot be any buildings, like on water or park areas (at this level, parks are thought to be large areas like Central Park in New York). At this stage the user is also able to specify a height map for the city, controlling the skyline of the whole city with a simple map, as shown in figure 5. For implementation details, please, refer to the following sections.

On the other hand, the user can choose to directly load a city map. These can be specified as a simple image map, or as an OpenStreetMap [12] file. See Figure 6. OSM is an open standard that creates and provides free geographic data such as street maps to anyone who wants them. The project was started because most maps one thinks of as free actually have legal or technical restrictions on their use, holding back people from using them in creative, productive, or unexpected ways. In skylineEngine we have chosen to use OSM as our vector-based standard, and all modules creating new street networks for synthetic cities use OSM as input/output file format.

5.3 Major & Minor Roads Modules

As mentioned above, districts, major-road (avenues) and minor-road modules are in charge of generating the street network of any city. If the user selects to load the map for a real city, then both modules are collapsed into a single node that simply reads the file and passes the resulting network to the next modules, e.g. to generate the city blocks. But, for synthetic cities, we have developed a user-controllable hierarchical system: Districts, major roads and minor roads.

5.3.1 Districts

In skylineEngine districts play a fundamental role to specify areas with different structural properties, that range from simple street patterns to the architectural style. The user can specify (in the configuration file) a districts map, each one with its own associated pattern [18]. Each pattern can be either grid, radial, organic or a mixture of these. Implementation of grid and radial patterns is simple, as they are just geometrical patterns. See Figure 2. We create organic patterns by gener-

```
1: currMap = ∅
2: for each pattern ∈ districMap do
3:    Generate pattern
4:    Position and orient pattern
5:    bounds = Bounds(pattern, districtMap)
6:    finalP = Clip(pattern, bounds)
7:    finalP = Merge(finalP, bounds)
8:    currMap = Merge(finalP, currMap)
9: end for
10: if exists land-useMap then
11:    currMap = Clip(currMap, land-useMap)
12: end if
13: return currMap
```

Figure 7: Districts and patterns

ating an initial dot set in the plane which is then processed by a relaxation method based on Voronoi diagrams, by using Lloyd's algorithm, as described in [3] and [16]. We then extract the major roads as the contours of the weighted Voronoi diagram.

The algorithm for districts in shown in figure 7, where we iterate over the patterns defined in the district map and the configuration file. For each pattern, we firstly generate and position it. Then, operation *Bounds* computes the boundary of the pattern as drawn in the district map. Next, the operation *Clip* cuts the streets to the extent of the pattern, and the results are merged along with the boundaries of the pattern itself. If there is a land-use map defined, the resulting city map is clipped against it to guarantee no building is constructed where it should not be, e.g. on water.

5.3.2 Minor roads

For the Minor Roads module, as they usually consist of small regular grids bound by avenues (the major roads), we simply compute the intersection of each area with a regular grid of cubes, resulting in small regular patterns of grid type. This can be seen in Figure 2. In this case, the user is free to specify either a fixed angle for the major areas in each district, or leave the system to choose a random angle for each grid.

5.4 Blocks Module

For OSM and path-based Major and Minor modules, it is easy to realize that the resulting network of street segments and intersections forms a graph. We use this graph to find all blocks in a city: blocks are closed polygons, that can be found as the minimal closed circuits. See Figure 8. To detect a closed circuit we start from any node in the graph. Then we calculate all the geometric angles between incoming segment and the outgoing segments of all paths connected to this node (with the exception of the first node where we calculate the outgoing segments angles with respect to the x-axis). From them we choose the segment with the lowest positive angle and then we continue through that street segment to the next node. When a neighbour node has been used, we eliminate it from the list of node neighbours and we also eliminate its respective path from the list of paths connecting neighbours. We repeat this process until we get again to the initial node. Then, we store this circuit as a list of the street segments we visited, and we start a new search from the first node having remaining neighbours.

Using this method, we get all city blocks plus an extra circuit containing all the other ones: this external circuit is exactly the base outline of the city. The rest of circuits found are block areas which represent a partition (in the mathematical sense) of the city area: their union is the whole city and their pairwise intersections are empty.

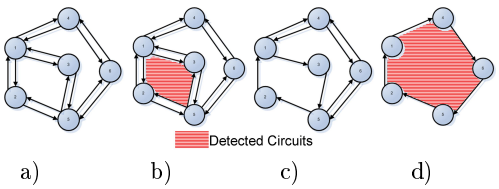

Figure 8: Circuits detection algorithm: a) each undirected edge is duplicated to create two directed ones. b) a circuit is detected and c) it is removed. d) Finally, the outer bounding circuit is isolated and eliminated.

Once we have circuits identified, either by

the previously described algorithm or directly imported from an image or from a **.obj** file, we reduce their size to leave some space for sidewalks, and the inner space is sent to the next stages in the pipeline (e.g. **lots**).

5.5 Lots Module

After the creation of highways and streets the populated area of the city is subdivided into many small areas which we call blocks. Those have to be divided into lots for the placement of buildings. Currently, in the state-of-the-art literature, there are two ways of partitioning a block into the lots (parcels) for buildings: In [13], a block is divided into smaller units using a simple, recursive algorithm that divides the longest edges that are approximately parallel until the the subdivided lots are under a threshold area specified by the user. The other commonly accepted method is [1], where the main axis of the oriented bounding box of the block is calculated and stochastically sampled with points. Then, copies of the points are created on each side of the axis, and their Voronoi diagram is calculated. Finally, the intersections of the Voronoi diagram with the contour of the block are used to help define the parcels. The first method is well suited for certain US cities, while the second is clearly though for Manhattan-like cities.

For **skyline**Engine and for Manhattan-like cities, we have implemented a simplified version of the second method that simply longitudinally splits the block in two and then creates a certain user-controlled number of lots at the sides of the subdivisions. See Figure 9(right). However, for cities like Paris or Barcelona, the original method presented in [13] does not provide a block partition that resembles the one in these cities. So we decided to use a different partitioning system, which basically starts by splitting the contour line of the block into equally sized (possibly curved) segments. Then for each segment end, we compute its inner normal as the vector perpendicular to the tangent line passing through that point. Finally, we displace each segment inwards by a distance randomly chosen so that its average meets a user-provided value. As segments could potentially be complex curved lines, we actually decided to further split each segment in three rectilinear sub-segments, all of which are applied the previously described displacement method. As can be seen in Figure 9(left), this method results in quite plausible lot distributions, but we have observed it can cause weird situations for too concave or too small polygons, although this is similar to the problems reported in [5] for the creation of their security border.

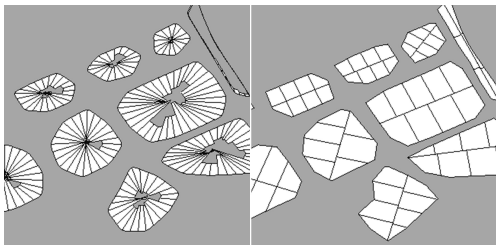

Figure 9: Different ways of creating lots: Paris-like and Manhattan-like.

5.6 buildingEngine

Our **building**Engine module is based on an interactive visual editing scheme for shape grammars, allowing the creation of rulebases from scratch without text file editing and in a user-friendly way. See Figure 10. This is based both on a key realization, that the rulebase represents a DAG, and on a shift in the development paradigm from a product-based representation to a rule-based one. This also opens the door to new possibilities, from simple model verifications to full model editing through graph rewriting operations. For more details, please refer to [14].

6 Other Modules

There is a number of new modules recently developed, or currently under development for **skyline**Engine . In particular, we can mention modules for:

- **Streets, Sidewalks and Parks**: we are currently developing modules to fill the

Figure 10: The user can design buildings with a visual graph-based paradigm. From the Raccolet House (colored in the front), we have created all the other buildings in this Figure.

empty spaces in our cities: streets with road markings, street lights, sidewalks, trees, benches, etc.

- **City Evolution**: we have developed a module reproducing the geometric urban temporal expansion part described in [22], where the street network grows over time following a set of predefined heuristics. Also, we are currently complementing this module with an urban traffic simulator that will enable us to assess the usage of each street, to guide our city growth mechanism, as described in that same paper.

- **Example-based street modelling**: We also have developed a module implementing the *geometric* part of [1], where corners of an existing city are copied to a new empty space and re-connected with new streets following statistical patterns of the original city.

- **Bridges**: One module that is currently under heavy development is a bridge module to create procedural, sound bridge structures. In particular, we are concentrating in suspension bridges like the *Brooklyn Bridge* or the *Manhattan Bridge*. See figure 13.

- **East Ancient Buildings**: another interesting module that is under development is the creation of pagodas and other east ancient buildings, following the procedures described in [19].

7 Implementation

As we were interested in the graph-based paradigm for city generation, we have developed a fully functional prototype on top of SideFX Houdini 3D modelling and animation system [17]. For that, we have used a mixture of Houdini's own nodes, embedded Python scripts and external Python methods. See Figure 11. The code for **skyline**Engine is available from [4].

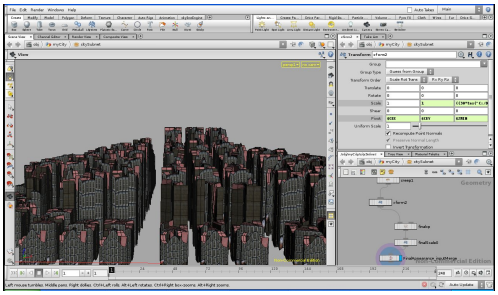

Figure 11: The user is allowed to visually control the city creation flow by inspecting (and controlling) the respective nodes.

8 Results and Conclusions

In this section we present some sample results of the **skyline**Engine system. In figure 2 we can see the intermediate steps the algorithm follows to create a Manhattan-like city, and the final result can be seen in Figure 12. As can be seen, although the resulting city is a synthetic one, it has the look and feel of an existing one. However, **skyline**Engine is also able to handle real-world cities, as Figure 6 shows.

Also, the **skyline**Engine system is capable of a very large data amplification: starting from two user-provided images (as shown in Figure 1), we obtain a major-roads distribution with 86 primitives (polygons). After minor roads are traced, the resulting geometry has 931 primitives, each representing a block. Finally, the subdivision into lots results in 7471 primitives, each one being a building lot. These lots

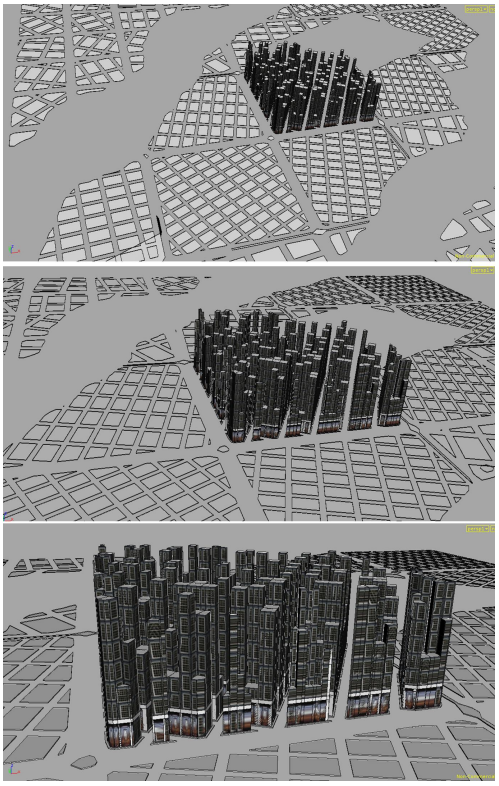

Figure 12: Manhattan-like city

are feeded onto the **building**Engine module, resulting in several buildings being created. See Figure 2.

9 Future Work

One promising avenue is the exact localization and manipulation of urban landmarks: a realistic city has certain specifically recognizable buildings or even whole streets that must be precisely characterized, located and positioned. Having a uniform, accurate and controllable way of doing so is an open problem for the current state-of-the-art in urban modelling, and remains as a challenge for **skyline**Engine .

skylineEngine together with the **building**Engine module are really flexible in permitting the user to build any kind of buildings and 3D objects from low-polygonal to high-resolution geometry. Additionally, it is possible to manage the preferred level-of-detail. So the user can decide to get the maximum performance for his working pipeline by, for example, generating low-detail models for large camera distances. With only few adjustments he can generate from the scene a more detailed version for a close-up.

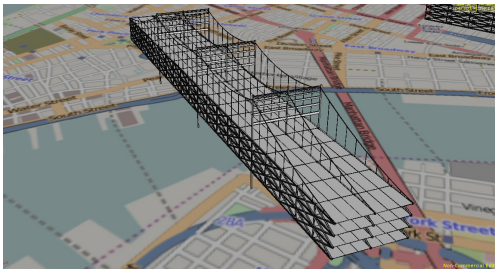

Figure 13: Close view of a bridge.

Batch Export of 3D Models becomes a specially important pending task. As shown in Section 8, data amplification can produce extremely large models, so a system for saving and recovering the whole geometry to and from disk is needed, as well as a specific out of core rendering methodology.

Acknowledgements

Thanks to Joel Coll for the bridge image. Raccolet House model by DAZ 3D (www.daz3d.com). This work was funded with grant TIN2007-67120 from the Ministerio de Educación y Ciencia, Spain.

References

[1] Daniel G. Aliaga, Carlos A. Vanegas, and Bedřich Beneš. 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] Guoning Chen, Gregory Esch, Peter Wonka, Pascal Müller, and Eugene Zhang. Interactive procedural street modeling. *ACM Trans. Graph.*, 27(3), 2008.

[3] Oliver Deussen, Stefan Hiller, Cornelius van Overveld, and Thomas Strothotte. Floating points: A method for computing stipple drawings. *Computer Graphics Forum*, 19:40–51, 2000.

[4] GGG-UdG. **skyline**Engine, 2010. URL http://ima.udg.edu/∼dagush/ Projects/ skylineEngine/.

[5] Francisco González and Gustavo Patow. Continuity mapping for multi-chart textures. In *SIGGRAPH Asia '09: ACM SIGGRAPH Asia 2009 papers*, pages 1–8, New York, NY, USA, 2009. ACM.

[6] Marc-André Guindon. *Learning Maya: Foundation*. Alias LearningTools, 2005.

[7] Paul E. Haeberli. Conman: a visual programming language for interactive graphics. *SIGGRAPH Comput. Graph.*, 22(4):103–111, 1988.

[8] G. Kelly and H. McCabe. A survey of procedural techniques for city generation. *ITB Journal*, 14:87–130, 2006.

[9] Markus Lipp, Peter Wonka, and Michael Wimmer. Interactive visual editing of grammars for procedural architecture. *ACM Transactions on Graphics*, 27(3):102:1–10, August 2008.

[10] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural modeling of buildings. *ACM Trans. Graph.*, 25(3):614–623, 2006.

[11] Pascal Müller, Gang Zeng, Peter Wonka, and Luc Van Gool. Image-based procedural modeling of facades. 26(3), 2007.

[12] OpenStreetMap. OpenStreetMap, 2009. http://www.osm.org.

[13] Yoav I. H. Parish and Pascal Müller. Procedural modeling of cities. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 301–308, 2001.

[14] Gustavo Patow. Interactive modelling of procedural buildings. In *XX CONGRESO ESPAÑOL DE INFORMÁTICA GRÁFICA, CEIG2010*, page To appear, 2010.

[15] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The algorithmic beauty of plants*. Springer-Verlag New York, Inc., New York, NY, USA, 1996.

[16] Adrian Secord. Weighted voronoi stippling. In *NPAR '02: Proceedings of the 2nd international symposium on Non-photorealistic animation and rendering*, pages 37–43, New York, NY, USA, 2002. ACM.

[17] SideFX. Houdini 10, 2009. http://www.sidefx.com.

[18] Jing Sun, Xiaobo Yu, George Baciu, and Mark Green. Template-based generation of road networks for virtual city modeling. In *VRST '02: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 33–40, New York, NY, USA, 2002. ACM.

[19] Soon Tee Teoh. Generalized descriptions for the procedural modeling of ancient east asian buildings. In *Proc. of the International Symposium on Computational Aesthetics in Graphics, Visualization, and Imaging (CAE'09)*, 2009.

[20] C. Vanegas, D. Aliaga, P. Mueller, P. Waddell, B. Watson, and P. Wonka. Modeling the appearance and behavior of urban spaces. *Proceedings of EUROGRAPHICS, State of the Art Reports (also Computer Graphics Forum, to appear)*, pages 1–18, 2009.

[21] Benjamin Watson, Pascal Mueller, Oleg Veryovka, Andy Fuller, Peter Wonka, and Chris Sexton. Procedural urban modeling in practice. *IEEE Computer Graphics and Applications*, 28:18–26, 2008.

[22] Basil Weber, Pascal Müller, Peter Wonka, and Markus H. Gross. Interactive geometric simulation of 4d cities. *Comput. Graph. Forum*, 28(2):481–492, 2009.

[23] Peter Wonka, Michael Wimmer, François Sillion, and William Ribarsky. Instant architecture. *ACM Transactions on Graphics*, 22(4):669–677, july 2003. Proceeding.