CEIG - Spanish Computer Graphics Conference (2012)
Isabel Navazo and Gustavo Patow (Editors)

# A procedural modeling system for the creation of huge models

Francisco Cubero, Albert Mas, Gustavo Patow

ViRVIG-UdG
Universiy of Girona, Spain

**Abstract**

*This paper presents a new general purpose procedural geometrical modeling system. It is focused on providing flexibility, modularity and scalability. Furthermore, it is taylored to manage huge geometric models, with millions of polygons. An out-of-core memory management system assures that any scene size can be generated during the modeling evolution. This generation is performed by a set of rules and operations on geometrical objects, organized as a directed acyclic graph.*

*I.3.6 [Computer Graphics]: Computer Graphics—Languages I.3.5 [Computer Graphics]: Computer Graphics—Computational Geometry and Object Modeling I.3.5 [Computer Graphics]: Computer Graphics—Three-Dimensional Graphics and Realism*

## 1. Introduction

One of the main challenges in computer graphics and interactive applications is content creation, which is a task that requires realistic and detailed geometry. At the same time, there is an increasing need to develop interactive user-friendly editing tools allowing a broader range of public to generate new content. The current approach to 3D modeling is to manually create 3D geometry using tools like Autodesk Maya or 3ds Max. This process is time consuming, tedious and repetitive, but gives to the artist full control of the final 3D model.

On the other hand, in computer graphics, procedural modeling tools allow to define large sets of geometrical data with just a few rules. These rules define, with a small set of parameters, how the geometry has to be constructed automatically. Some examples are the works that simulate plants [PL90], urban environments [PM01, MWH*06], and even automatic image-based maze generation [XK07]. The main advantage of those methods is that one does not have to store huge geometrical models, only their definitions. Then, these complex models can be generated each time we need them, and only needing the ruleset that defines the model/scene.This is often called as *geometric explosion*, referring to the fact that a small input can produce a large output. On the other hand, because in general the generated models can be huge, we need a relevant amount of computer resources to handle them, which in some cases can be prohibitive.

In this paper we propose a new visual procedural modeling system, mainly focusing on flexibility, modularity and scalability. The language is defined as a complete set of geometrical primitives and operations, and is capable to manage huge geometrical model sets. Also, it is defined in a way such that it allows to control the modeling process much in the same way than classical computer programming, with branching, loops and encapsulation. In addition, a visual language tool is presented, which enables the user to work in terms of a purely visual language. As a result, the system we present allows a designer to disintegrate complex applications into modular components. By interactively connecting simple components (the nodes), users construct a complete procedural modeling application that matches their own needs. In our system, the flow of data is controlled by wiring nodes among themselves. As a result, we replace the usual text-based input system with a visually-driven mechanism that is independent of the geometry complexity generated.

## 2. Previous Work

Procedural modeling is a kind of methodology to construct geometrical models from a set of rules which represent commands and their combinations. Some examples can be found in the works by Deussen et al. [DHL*98] and Palu-



DOI: 10.2312/LocalChapterEvents/CEIG/CEIG12/123-132

bicki et al. [PHL*09], where they presented a language to model plants and trees. In the field of urban modeling, the works presented by Parish and Muller [PM01] and Muller et al. [MWH*06], define a language for the generation of buildings. The main drawback of these methods appears during the design of complex rule combinations to generate accurate models, as its text-based paradigm makes it difficult to edit and modify.

One of the most basic approaches for of procedural modeling is the Generative Modeling Language (GML) [Hav09], which contains the basic geometric-level commands, and provides a text-based language to combine them to construct new and complex rules. GML is a rich language with a large degree of flexibility. However, in spite of the creator's efforts, it remains a complex system for the non-expert user.

Historically, procedural modelers are related with visual languages. ConMan [Hae88] was the first visual language attempt for computer graphics modeling.It presented a simple set of linked nodes to define the process pipeline. Later, in [LD99] was presented a node tree structure. In [GK07] there was presented the Plab system, that defines a node graph structure. Since these languages allowed permit the full geometrical design, they have limitations for huge models. The main idea of these methods can be found in current modeler softwares, such as Maya or Houdini. More recently, Esri's CityEngine [Esr12], Epic Games' UDK [Epi12] and Patow [Pat12] simultaneously introduced very similar visual languages for these shape grammars, where each node is a command and the connection between two nodes represents the flux of geometry between them.

To solve the hardware limitations for huge models, we can consider out-of-core techniques. In [SjCC*02] is presented a survey of this king of methods. In addition, in [CGG*04] is proposed a method to visualize out-of-core models.

Among other approaches for procedural modeling, Peytavie et al [PGMG09] presented a tiling method for generating piles of rocks without any computationally demanding physically-based simulation. Kelly and Wonka [KW11] proposed a generalization of the classic concept of an extrusion where a set of profiles are used to extrude a floor-plan to generate whole building structures, plus an anchoring system to locate assets like doors, windows or other elements. Talton et al. [TLL*11] proposed a Markov Chain Monte Carlo system to generate the variations on a given ruleset to achieve a desired target. Their applications included all king of procedural modeling techniques, from plants and trees (L-systems) to buildings (CGA Shapes) to whole cities (random blocks). Lin et al. [LCOZ*11] presented an algorithm for interactive structure-preserving retargeting of irregular 3D architecture models, taking into account their semantics and expected geometric interrelations such as alignments and adjacency. The algorithm performs automatic replication and scaling of these elements while preserving their structures by decomposing the input model into a set of sequences, each of which is a 1D structure that is relatively straightforward to retarget. As the sequences are retargeted in turn, they progressively constrain the retargeting of the remaining sequences. Musialski et al. [MWW12] proposed a novel interactive framework for modeling building facades from images, exploiting partial symmetries across the facade at any level of detail. Their workflow mixes manual interaction with automatic splitting and grouping operations based on unsupervised cluster analysis. Ceylan et al. [CML*12] presented a framework for image-based 3D reconstruction of urban buildings based on symmetry priors: Starting from image-level edges, they generate a sparse and approximate set of consistent 3D lines, which are then used to simultaneously detect symmetric line arrangements while refining the estimated 3D model. Merrell et al. [MSK10] presented a method for automated generation of building layouts for computer graphics applications. In their approach, given a set of high-level requirements, an architectural program is synthesized using a Bayesian network trained on real-world data. The architectural program is realized in a set of floor plans, obtained through stochastic optimization. The floor plans are used to construct a complete three-dimensional building with internal structure. Krecklau and Kobbelt [KK11] presented a system for the easy generation of interconnected structures such as bridges or roller coasters where a functional interaction between rigid and deformable parts of an object is needed. Their approach mainly relies on the top-down decomposition principle of shape grammars to create an arbitrarily complex but well structured layout. In their work, Benes et al. [BSMM11] present guided procedural modeling, an approach that allows a high level of top-down control by breaking the system into smaller building blocks that communicate. In their work, the user creates a set of guides that define a region in which a specific procedural model operates. These guides are connected by a set of links that serve for message passing between the procedural models attached to each guide. In this approach, local control is introduced by the building blocks themselves, but further control is left to the the detailed level of procedural systems. However, none of these works deal with procedural techniques for visually creating huge models as we do here, and these approaches can be easily incorporated in a system like ours.

## 3. Overview

This work presents a new procedural modeling system defined with a two layers structure: *Procedural Abstraction Layer (PAL)* and *Geometrical Abstraction Layer (GAL)*. The PAL defines a set of rules for the automatic geometry generation. These rules are represented by a directed acyclic graph that represents the full scene. The graph nodes perform operations such geometry creation and transformation. The GAL layer is responsible of managing the geometrical data. Also it implements the interface to create and edit the visual language and the out-of-core management system. This out-of-

core system is capable to manage the geometry data memory requirements, using main or secondary memory in function of the current geometrical scene and the current hardware memory storage. The PAL interacts only with the GAL, that creates an interface to the classical 3D View API systems, such as OpenGL or DirectX (see Figure 1). Hence the PAL becomes platform independent, and only a new GAL interface would be required for new environments.

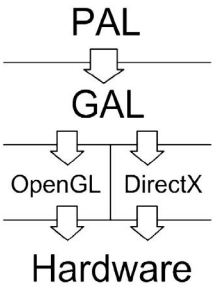


**Figure 1:** *PAL and GAL system.*

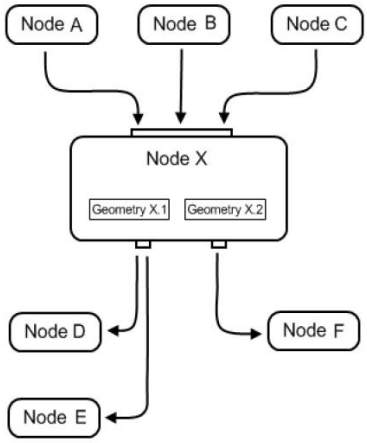


**Figure 2:** *PAL node example. It gathers three data flows and generates to geometric data sets that are sent by two different output channels.*

## 4. Procedural Abstraction Layer (PAL)

The PAL directed acyclic graph represents the scene using the graph nodes and the connections between them to define rules for the procedural modeling. The graph nodes define the rule operators that generate or modify the geometry. The graph connections define how and where the rules are applied. This kind of graph allows the representation of the full scene and the geometry data flow. The flow is controlled using geometry generation nodes or branching control nodes The system permits to query and visualize the data at any part of the graph. Note the importance of the acyclic property, so the data flow locks must be avoided.

### 4.1. Nodes and connections

In general, PAL nodes perform operations on their input geometric data flow. Each operator receives an incoming geometry flux, processes it, and provides at its outputs the resulting processed geometry. This processing that occurs inside each node is controlled by the user through the node parameter interface. Also, the user controls the geometry flow between the nodes, and has complete control of the operations applied to every bit of geometry. This way, the user is able to configure the whole production process in arbitrary ways, resulting in a procedural model, which can get arbitrarily complex. These capabilities go far beyond a mere scene description language, actually presenting all the constructive elements of a full general-purpose procedural engine.

These operations are performed on an incoming geometry flux from one or more nodes, except for the case of so called *generator ones*. Then, the result is stored and sent to other downstream nodes. The process of receiving and sending the data is controlled by the node input and output channels (see Figure 2). For every node, there is only one input channel that gathers the incoming geometry from one or more nodes, creating a unique data flow. However, an implementation with differentiated input channels is also feasible, although we discarded it in our current implementation to simplify ease of use. On the other hand, ever node can be multiple output channels to create one or more data flows. The rationale is that there exist operations that, for instance, divide the geometry flux according to a user-provided criteria, assigning each output channel to each of the resulting geometry sets. The connections between nodes are represented by unidirectional data pipes, that define the connection between one output channel of a node and one input channel of a downstream node.

The connections management system controls the requirement that the graph must be acyclic. This is verified each time a new node is inserted or any connection is modified. Moreover, system controls consistency, which implies controlling when and where the geometry data has to be regenerated. When a node parameter is modified an invalidation signal is created. This signal is spread to the output connected nodes, becoming invalidated nodes. The signal is spread in cascade way, and it stops at the final nodes (see Figure 3). Then, when a node recieves a geometry request, by example to be displayed, this geometry has to be regenerated, spreading an update petition to the input nodes. The spread stops in those nodes that are valid (see Figure 4).

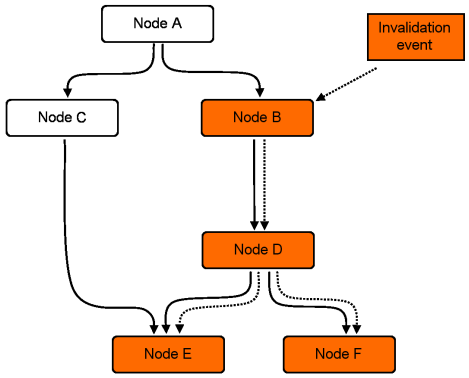


**Figure 3:** *Invalidation propagation example. A change in node B produces a signal invalidation spread on nodes D, E and F.*

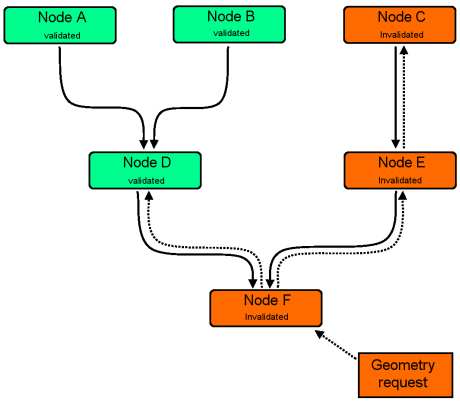


**Figure 4:** *Geometry request propagation example. Nodes A, B and C are valid and do not need to recompute anything. Node F receives a geometry request, and as it is an invalid node, the request is propagated to its input invalidated nodes (orange).*

### 4.2. Geometry generation and transformation

There are two main node types for generating and transforming the geometry. The generator nodes are used to generate the basic geometry, that is the beginning of the data flow. They have only one output channel and no input channel. There are generator nodes for basic shapes, such as spheres or prisms, and generator nodes that import external geometrical data for more complex shapes. The transformation nodes perform basic geometrical transformations on the input geometry, such as translation, rotation, scale and shear. The output channel returns the transformed geometry. In addition there are other transformation nodes that perform more complex operations, such as nodes that change the mesh subdivision level or that performs torsions.

### 4.3. Data flow control

There are three kind of nodes to control the data flow: grouping, branching and looping nodes. The grouping nodes encapsulate a directed acyclic subgraph into only one node (see Figure 5). The subgraph data flow input and outputs match with the grouping node input and output channels.

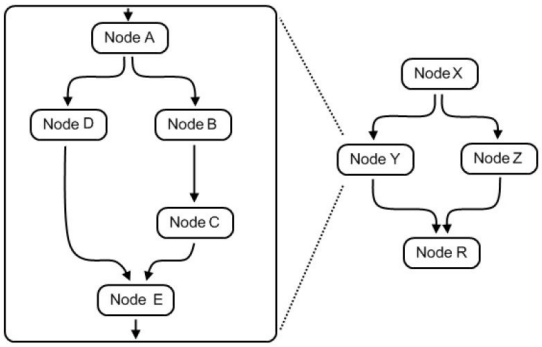


**Figure 5:** *Grouping node example.*

The branching nodes are useful to control the data flow with selections or filtering. The selector nodes select one of the input channel sources and export the geometry by only one output channel. The filtering nodes gather the input data flow and export those geometry that validate a condition. In Figure 6 there is a simple example of branching, where the geometry is translated in two ways from $X$ axis. In addition, there are the opposite branching nodes, the sumatory nodes, that join two or more geometries in only one data flow.

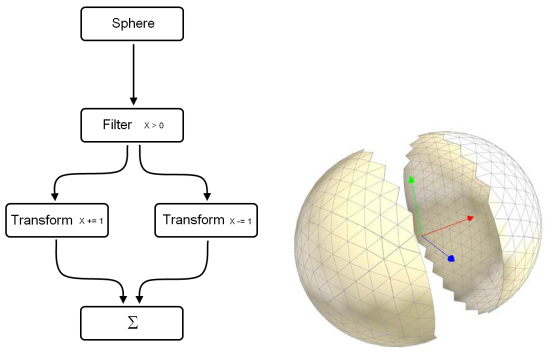


**Figure 6:** *Branching example.*

Finally, the iterator nodes group a data flow segment into a loop controlled by a fixed number of iterations or by a condition. These nodes modify the geometry, or accumulate the

changes using a sumatory node, for each loop. In Figure 7 there is an example of an iterator node that accumulates the results for each loop, creating a mesh of objects.

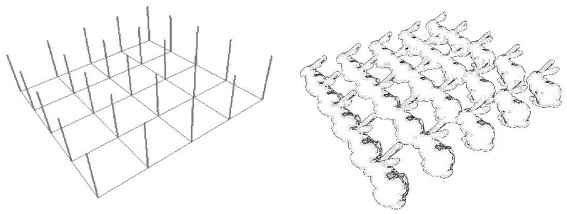

**Figure 7:** *Iterator node example. For each loop a new bunny is translated to the next grid vertex placement.*

### 4.4. Bypassing data

The PAL node process the input data, stores the results and send them to the output channels. The data storing for each node implies a poor performance in computing time and memory storage, specially on sumatory and transformation nodes. A clear example of this is the subgraph located into an iterator node. In these cases, we need to bypass the data between nodes. There are two nodes that bypass the data, named *Complex Storer* and *Transform Storer*.

The *Complex Storer* node stores internal references to the stored geometry of the nodes connected to the input channels. If the node becomes invalidated and receives a geometry request signal, it queries to the referenced nodes about the geometry. This is very usefull on accumulation operations, such are sumatory or iteration nodes.

The *Transform Storer* node is a *Complex Storer* node that stores a transformation matrix. It is used to replace the basic transformation nodes. When it receives a geometry request signal, the stored geometry in the referenced node is updated with the transformation matrix, bypassing the data to the output channel. In addition, the *Transform Storer* node joins the transformation matrix into only one when there are multiple sequential basic transformations.

## 5. Geometry Abstraction Layer (GAL)

The GAL system manages the geometry creation, modification and storage without considering the overall data flow. GAL system defines an interface between the procedural data and the geometric data, making them independent and scalable.

The geometry data structure (see Figure 8) is based on a list of triangles. To manage other data, such as normal vectors, colors, numerical values, transformation matrices or any kind of data, both triangles and vertices have associated lists of attributes. Each attribute is defined by an identifier, that is composed by a label, the vertex or triangle identifier,

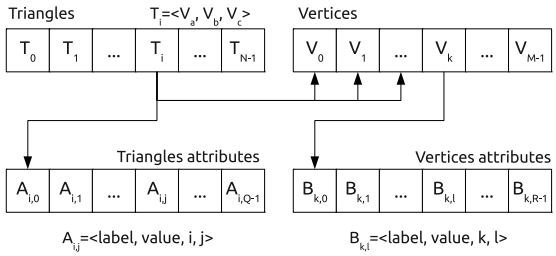


**Figure 8:** *Geometry data structure.*

and the attribute value. There are no limitation on size lists or on attributes data types.

GAL only offers geometry data creation and query operations. Because of GAL is independent of PAL, we do not have to worry about any complex operation with the geometry.

### 5.1. Memory management

The GAL system defines an interface to control the memory management. Basically, we try to store objects in main memory as much as possible. Thus, when a new object is created, the interface choose where it has to be stored, selecting the main or the secondary memory, depending on the current free main memory and the estimated object size. The selection method calculates

$$\frac{M_f - e_m(S)}{M_t} = M_e$$

where $M_f$ is the current free main memory, $M_t$ is the total main memory, $e_m$ is the estimated memory size of object $S$, and $M_e$ is the estimated percentage of free main memory available after storing the object $S$. If $M_e < t_m$, where $t_m$ is a user threshold that means the main memory percentage that must remains free, the object is stored in main memory. Otherwise, the object is stored in secondary memory.

This out-of-core method uses from $2^{64}$ upto $2^{128}$ bits of memory addressment per vertices and triangles, that means we can manage really huge geometrical models, about the order from $10^{19}$ to $10^{77}$ triangles respectively. Due system limitations about maximum filesizes, the storage uses a sequence of files, creating an index system to get access to any vertex or triangle with independence of what internal file is stored. The vertices list is stored in a sequence of fixed size files (see Figure 9). This size is used by the index system to access the vertices. The triangles list is stored in the same way than vertices.

The triangles attribute values lists are stored together in one file sequence without indexing (see Figure 10), because the attribute values are variable in size. To get access to the attribute values, another indexed file sequence is stored. It

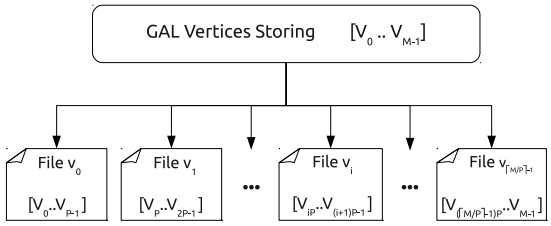


**Figure 9:** *Vertices list storing structure.*

contains for each attribute the identifier, the file identifier where the attribute value is stored and the value location and size in bytes. The same considerations have been done for the vertices attribute values lists.

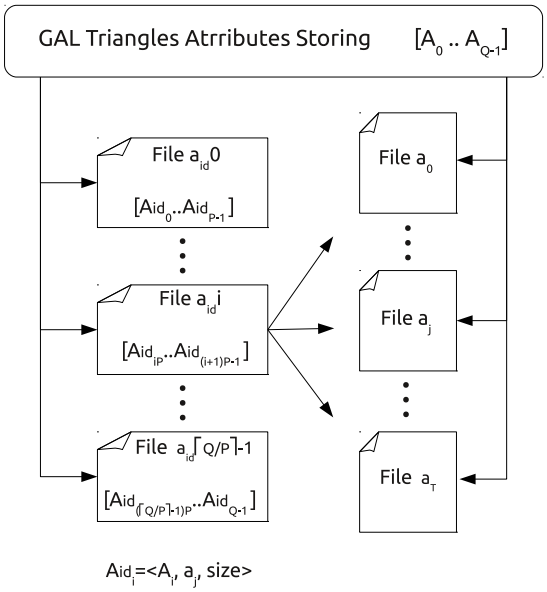


**Figure 10:** *Triangles attributes lists storing structure.*

### 5.2. Visual language

Over the years researches have realized that text-based approaches are not the most practical way to cope with the inherent complexity of procedural models. This resulted in several proposals for its replacement by visual languages since the pioneering work by Haeberli [Hae88], followed by the works by Deussen et al. [DHL*98] and Palubicki et al. [PHL*09], and more recently the works by Gengster and Klein [GK07] Patow [Pat12], but the latter being specific for bulding design. We have developed a visual tool to provide an edition system for the designer, where the user can edit the operations graph, create or destroy nodes, connect nodes among themselves, and adjust their local parameters. This can be seen at Figure 11, where the user can edit the node graph (right) and interactively preview the results of the changes (left). The user design the model in a graph view, where the designer can edit the graph, create new nodes and define connections. The changes are shown interactively in a 3D view, that uses any kind of 3D View API, such as OpenGL or DirectX. The 3D view updates are controlled by a visibility flag that is stored in each node. When a node has the visibility flag activated, all the internal stored geometry becomes visible. This means when 3D view needs to be refreshed, by an edition of the graph by example, the visible nodes send their respective update signals to the invalidated nodes, regenerating the geometry data only on those nodes that is needed, avoiding the regeneration of the full scene. Since the graph view is enough to design the model, the 3D view do not has interactive edition tools. Finally, some interfaces are provided to edit GAL information, such as vertices values, colors and other data types.

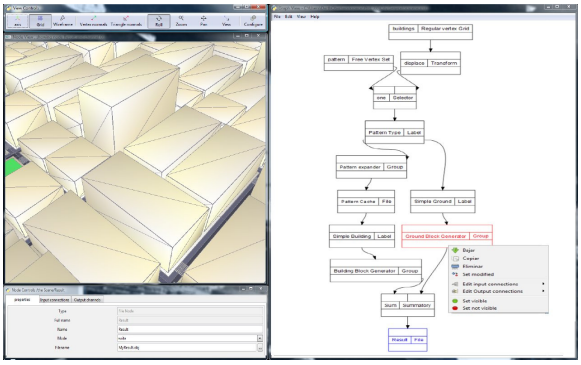

**Figure 11:** *Visual language editor. At right, the graph edition view. At top left, the 3D view. At bottom, the interface to modify the geometry data.*

## 6. Application case: Urban Modeling

To demonstrate the capabilities of the method we have adapted it to perform a city building procedural modeler. To achieve it we have created new PAL nodes to control the building specific generation. These nodes control the building base size, the openings and their windows, the number of levels, the facades and the roof. We use a database with a set of predefined objects for complex geometries, such as the windows library (see Figure 12).

We have designed a simple method to generate automatically the city. This method generates each building in a random way starting from a simple prism-like primitive, but other starting shapes can be easily used. The parameters such are levels, window types, roof types and building shapes are chosen stochastically. The levels are chosen randomly giving more priority to those buildings with a medium level size

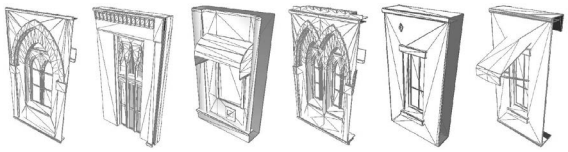

**Figure 12:** *Pre-generated windows geometry library.*

to get more realistic skylines. The building shapes are chosen from a set of predefined blocks with different building shapes and placements (see Figure 13).

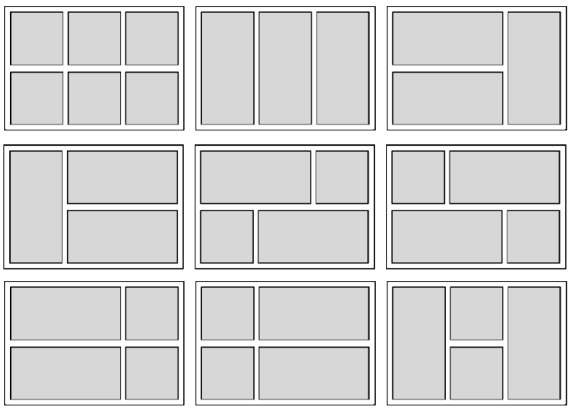

**Figure 13:** *Pattern set for the building block configurations.*

## 7. Results

The results presented here are compared using main memory or the out-of-core method. The tests have been processed using an Intel Core2 Quad at 2.4GHz, 8MB of cache memory and 4GB of DDR3 main memory. The out-of-core results have been tested with a 1.5TB SATA2 hard disk, with 64MB of cache memory and 7500 rpm. The system has been implemented using Python, so it is multi-platform and the new nodes can be implemented in an easy way.

To test the building generation, we have designed a simple building shape. Then, the computing time has been evaluated using main memory and out-of-core management, comparing the same building with different levels and windows. One can see that the computing time increases linearly with the geometry size. Also, the main memory is faster than the out-of-core management. Although, in some cases the main memory is not enough to process huge geometries. In Figure 14 there is a computing time graph in function of number of building levels and window type. Note the building with fourth window type cannot be processed in main memory for more than 3 levels due his size.

In Figure 15 you can observe some examples of this testing buildings.

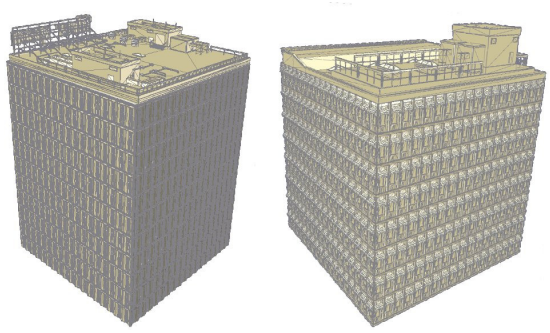

**Figure 15:** *Examples of test buildings, designed to test the system on a plausible technical basis (no to get realistic buildings). At left, a building with 11 levels using the second type windows (see Figure 12), requiring 618000 triangles. At right, a building with 8 levels, third type windows, requiring 105100 triangles.*

In Figure 16 there are some examples of the buildings combinations as seen in the Figure 13.

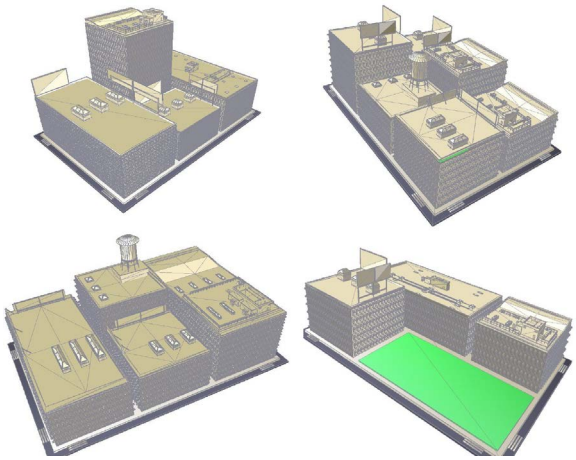

**Figure 16:** *Examples of random building blocks generation. The models sizes are, from top left to right bottom, 789, 596, 583 and 322 thousand triangles size.*

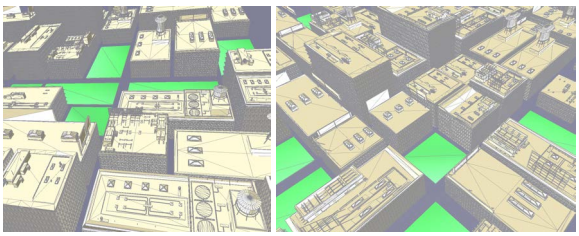

**Figure 17:** *Two neighborhoods of our final city. Left: 12 blocks, Right: 17 blocks.*

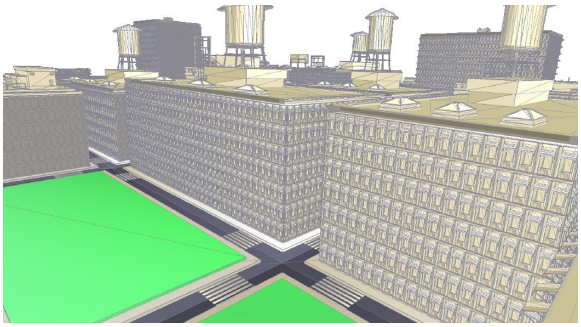

**Figure 18:** *Closer view of automatically generated city with about 61 million triangles.*

| Reference | Block | Build | Vertices | Triangles | Mem | Time |
|---|---|---|---|---|---|---|
| Fig.16-left-top | 1 | 4 | 538K | 583K | 52.1 | 0d 1:11 |
| Fig.16-right-top | 1 | 4 | 697K | 789K | 63.2 | 0d 1:26 |
| Fig.16-left-bottom | 1 | 6 | 560K | 596K | 58.2 | 0d 1:19 |
| Fig.16-right-bottom | 1 | 3 | 283K | 323K | 21.9 | 0d 0:30 |
| Fig. 17-left | 12 | 38 | 4468K | 4135K | 418 | 0d 6:22 |
| Fig. 17-right | 17 | 68 | 8039K | 7594K | 760 | 0d 8:43 |
| Fig 19 | 130 | 467 | 57066K | 61637K | 5104 | 4d 20:22 |

**Table 1:** *Figures for some examples. From left to right, model name/reference, number of blocks, buildings, vertices, triangles, storage (in MB) and time (days hours:minutes).*

Finally, we have combined a set of building blocks to construct a regular city. In Figure 18 and Figure 19 there are two views of the final generated city. It has 467 buildings organized into 130 blocks. The total model size is 61 million of triangles and about 5GB memory requirements. Obviously, this model cannot be managed only using the main memory, and the out-of-core system allows us to achieve this kind of huge geometrical models.

In Figure 17 we can see views of two neighborhoods of 12 and 17 blocks respectively.

## 8. Discussion

As shown in the previous section, the proposed procedural modeling editor is capable to manage huge scenes. The out-of-core management system allows to work with geometric models that would be impossible to manage only using the main memory. The relation between the computational times using main or secondary memory is lineal and has a reasonable difference. Moreover, the visual editor and the presented results demonstrate that is relatively easy to model complex and huge scenes with just a few nodes.

The main drawback is the overall computational time. The main reason is the chosen development language, Python, because it is an interpreted language. On the other hand, Python provides platform independence and scalability, so it is easy to develop new nodes. A suitable improvement would be to parallelize the system, with concurrence or with a computers cluster.

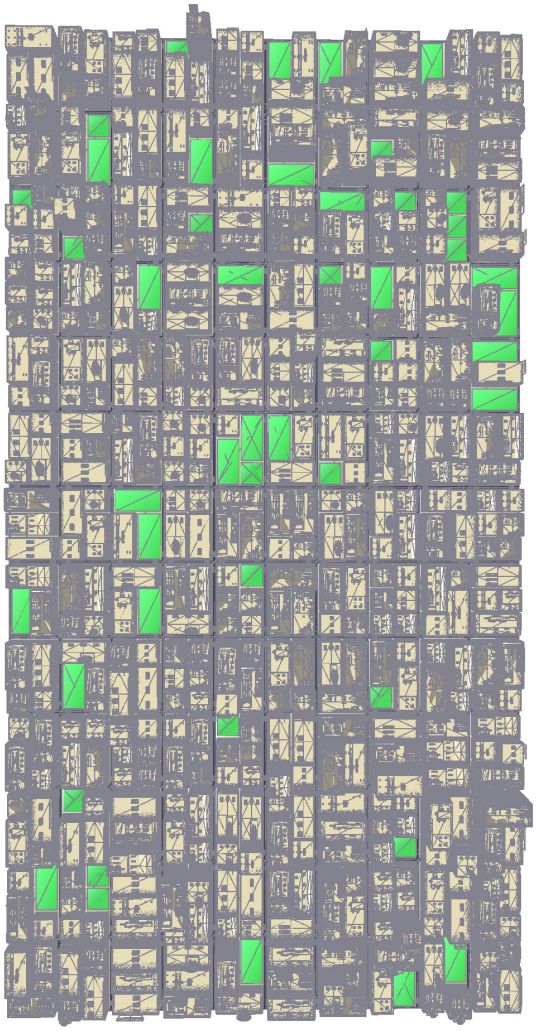

**Figure 19:** *Aerial view of the same city seen in Figure 18.*

## 9. Conclusions and Future Work

We have presented a new general purpose procedural modeler. The system is flexible, modular and scalable, and it is capable to manage huge scenes composed by millions of polygons. The procedural modeler is defined with a set of rules and geometric objects, organized in a directed acyclic graph. This graph controls the geometric data flow, where the nodes create and transform the geometry, and perform branching and looping to automatically produce complex models. The graph edition, that is the scene edition, is performed with a visual language tool.

We consider, as future work, the creation of new node types and new interfaces for other 3D view APIs. Also, different strategies to decide between whether to store an ob-

ject in main or secondary memory (e.g. LRU or other techniques), should be analyzed. Of course, this implies a whole loading/unloading mechanism to be thought of and implemented. In addition, we are interested into parallelize the system in two ways. First, applying concurrence, trying to manage the processes that transform the geometry data in efficient threads. Second, creating a communication system to manage a computers cluster, partitioning the tasks into different computers and avoiding the processing bottlenecks. Another interesting optimization is to incorporate techniques as instancing to avoid having duplicated geometry, but this would require a slightly different management system with elements that can be geometric objects and elements that can be just references (plus a transformation matrix) to other elements.

## 10. Acknowledgments

We want to thank the anonymous reviewers for their useful comments. This work was partially funded by grant TIN2010-20590-C02-02 from Ministerio de Ciencia e Innovación, Spain.

## References

[BSMM11] BENES B., STAVA O., MECH R., MILLER G.: Guided procedural modeling. *Comput. Graph. Forum 30*, 2 (2011), 325–334. 2

[CGG*04] CIGNONI P., GANOVELLI F., GOBBETTI E., MARTON F., PONCHIO F., SCOPIGNO R.: Adaptive tetrapuzzles: efficient out-of-core construction and visualization of gigantic multiresolution polygonal models. *ACM Trans. Graph. 23* (August 2004), 796–803. 2

[CML*12] CEYLAN D., MITRA N. J., LI H., WEISE T., PAULY M.: Factored facade acquisition using symmetric line arrangements. *Computer Graphics Forum (Proc. EG'12) 31*, 1 (May 2012). 2

[DHL*98] DEUSSEN O., HANRAHAN P., LINTERMANN B., MĚCH R., PHARR M., PRUSINKIEWICZ P.: Realistic modeling and rendering of plant ecosystems. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1998), SIGGRAPH '98, ACM, pp. 275–286. 2, 6

[Epi12] EPICGAMES: Unreal development kit (udk), 2012. http://udk.com. 2

[Esr12] ESRI: Cityengine, 2012. http://www.esril.com/software/cityengine/index.html. 2

[GK07] GANSTER B., KLEIN R.: An integrated framework for procedural modeling. In *Spring Conference on Computer Graphics 2007 (SCCG 2007)* (Apr. 2007), Sbert M., (Ed.), Comenius University, Bratislava, pp. 150–157. 2, 6

[Hae88] HAEBERLI P. E.: Conman: a visual programming language for interactive graphics. *SIGGRAPH Comput. Graph. 22* (June 1988), 103–111. 2, 6

[Hav09] HAVEMANN S.: *Generative Mesh Modeling*. PhD thesis, TU Braunschweig, 2009. URL: http://deposit.ddb.de/cgi-bin/dokserv?idn=977813207. 2

[KK11] KRECKLAU L., KOBBELT L.: Procedural modeling of interconnected structures. *Comput. Graph. Forum 30*, 2 (2011), 335–344. 2

[KW11] KELLY T., WONKA P.: Interactive architectural modeling with procedural extrusions. *ACM Trans. Graph. 30*, 2 (Apr. 2011), 14:1–14:15. 2

[LCOZ*11] LIN J., COHEN-OR D., ZHANG H., LIANG C., SHARF A., DEUSSEN O., CHEN B.: Structure-preserving retargeting of irregular 3d architecture. *ACM Trans. Graph. 30*, 6 (Dec. 2011), 183:1–183:10. 2

[LD99] LINTERMANN B., DEUSSEN O.: Interactive modeling of plants. *IEEE Comput. Graph. Appl. 19*, 1 (1999), 56–65. doi:http://dx.doi.org/10.1109/38.736469. 2

[MSK10] MERRELL P., SCHKUFZA E., KOLTUN V.: Computer-generated residential building layouts. *ACM Trans. Graph. 29*, 6 (Dec. 2010), 181:1–181:12. 2

[MWH*06] MÜLLER P., WONKA P., HAEGLER S., ULMER A., VAN GOOL L.: Procedural modeling of buildings. *ACM Trans. Graph. 25* (July 2006), 614–623. 1, 2

[MWW12] MUSIALSKI P., WIMMER M., WONKA P.: Interactive Coherence-Based Façade Modeling. *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2012) 31*, 2 (May 2012), 661–670. URL: http://www.cg.tuwien.ac.at/~pm/Facade2012/index.html. 2

[Pat12] PATOW G.: User-friendly graph editing for procedural modeling of buildings. *IEEE Computer Graphics and Applications 32* (2012), 66–75. 2, 6

[PGMG09] PEYTAVIE A., GALIN E., MERILLOU S., GROSJEAN J.: Procedural Generation of Rock Piles Using Aperiodic Tiling. *Computer Graphics Forum (Proceedings of Pacific Graphics) 28*, 7 (2009), 1801–1810. 2

[PHL*09] PALUBICKI W., HOREL K., LONGAY S., RUNIONS A., LANE B., MĚCH R., PRUSINKIEWICZ P.: Self-organizing tree models for image synthesis. *ACM Trans. Graph. 28* (July 2009), 58:1–58:10. 2, 6

[PL90] PRUSINKIEWICZ P., LINDENMAYER A.: *The algorithmic beauty of plants*. Springer-Verlag New York, Inc., New York, NY, USA, 1990. 1

[PM01] PARISH Y. I. H., MÜLLER P.: Procedural modeling of cities. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2001), SIGGRAPH '01, ACM, pp. 301–308. 1, 2

[SjCC*02] SILVA C., JEN CHIANG Y., CORRÃŁA W., EL-SANA J., LINDSTROM P.: Out-of-core algorithms for scientific visualization and computer graphics. In *In VisualizationâĂŹ02 Course Notes* (2002). 2

[TLL*11] TALTON J. O., LOU Y., LESSER S., DUKE J., MĚCH R., KOLTUN V.: Metropolis procedural modeling. *ACM Trans. Graph. 30*, 2 (Apr. 2011), 11:1–11:14. 2

[XK07] XU J., KAPLAN C. S.: Image-guided maze construction. *ACM Trans. Graph. 26* (July 2007). URL: http://doi.acm.org/10.1145/1276377.1276414, doi:http://doi.acm.org/10.1145/1276377.1276414. 1

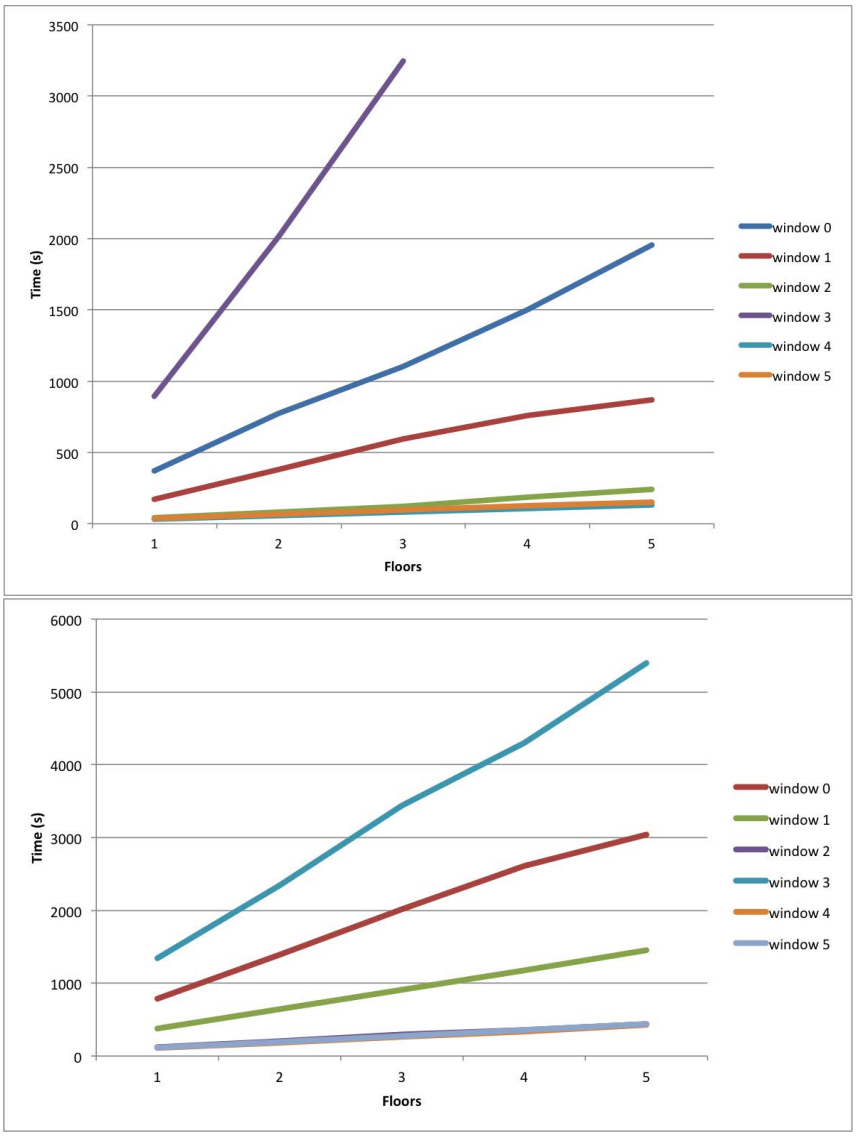


**Figure 14:** *Comparison between computing times (in seconds) using main (top) and secondary (bottom) memory for buildings with different levels and window types (see Figure 12).*