CEIG - Spanish Computer Graphics Conference (2017)
F. J. Melero and N. Pelechano (Editors)

Procedural Semantic Cities

Otger Rogla[1,2], Nuria Pelechano[1,2] and Gustavo Patow[1,3]

[1] ViRVIG
[2] Universitat Politècnica de Catalunya
[3] Universitat de Girona

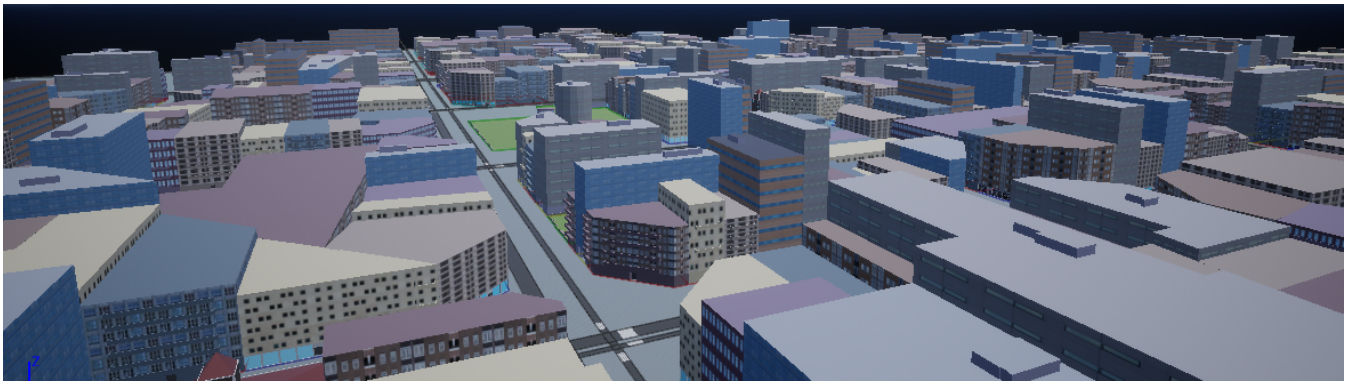

Figure 1: *Example of a city generated using our framework.*

Abstract

Procedural modeling of virtual cities has achieved high levels of realism with little effort from the user. One can rapidly obtain a large city using off-the-shelf software based on procedural techniques, such as the use of CGA. However in order to obtain realistic virtual cities it is necessary to include virtual humanoids that behave realistically adapting to such environment. The first step towards achieving this goal requires tagging the environment with semantics, which is a time consuming task usually done by hand. In this paper we propose a framework to rapidly generate virtual cities with semantics that can be used to drive the behavior of the virtual pedestrians. Ideally, the user would like to have some freedom between fully automatic generation and usage of pre-existing data. Existing data can be useful for two reasons: re-usability, and copying real cities fully or partly to develop virtual environments. In this paper we propose a framework to create such semantically augmented cities from either a fully procedural method, or using data from OpenStreetMap. Our framework has been integrated with Unreal Engine 4.

1. Introduction

Procedural modeling has witnessed an explosion in the last decade since the seminal works by Parish and Muller [PM01], Wonka et al. [WWSR03] and Muller et al. [MWH*06]. The introduction of shape grammars, and the CGA (Computer Generated Architecture) syntax have enabled the production of massive cityscapes with some simple, basic constructs. The introduction of visual representations [Esr14, Pat12] have simplified this process even further, allowing artists and designers to use standard, off-the-shelf tools such as Houdini [Sid12] for city design.

However, in spite of the many efforts in the field, the application of procedural models has remained circumscribed to movie and video-game production [STBB14], with some extensions to physical simulations [GDAB*17, VGDA*12, BMJ*11]. There has been little success in using these techniques for providing *meaning* to these environments, as they typically limit the output to only a mesh, consisting exclusively of geometry and properties such as material or texture data. Even if they do use semantics to some degree internally for the generation, those are discarded once the geometry is created and therefore cannot be used in further steps in the pipeline.

In this paper we present Semantic Procedural Cities, a system that allows intertwining the procedural creation of a city with a semantic tagging of its elements. Applications range from enhanc-

DOI: 10.2312/ceig.20171217

ing the procedural modeling process itself, for instance by allowing specific buildings and spaces to be built at recognizable urban points of interest, to the control of other urban phenomena like traffic simulation or pedestrian behavior to achieve richer environments.

Note that as opposed to traditional building or city generation software such as CityEngine [Esr14] where the final output is just a plain mesh, the output of our system provides a hierarchy of elements that may consist of procedurally generated geometry and/or reference model assets. This allows us to keep metadata such as semantics (points of interaction, tags, usage information, special navmesh areas), whereas outputting a single mesh would discard it. For example, a typical output from a CGA generated building would consist of a single mesh with the geometry for all its windows merged into it, whereas we keep semantic information that allows us to identify that a particular geometry is "a window" so that an autonomous agent could act on it (e.g., "look through it").

Our main contributions include:

- A procedural tagging system that is naturally blended into the procedural creation process.
- A set of new procedural commands that extend CGA and allow a specific, fine tuning semantic tagging of urban spaces.
- A feature extraction mechanism from available GIS data, plus the possibility of enhancing this with user-customizable definitions.

2. Previous Work

After the seminal work by Parish and Muller [PM01], which introduced the concept of L-Systems for urban generation; followed by the works by Wonka et al. [WWSR03], which dealt with a mechanism for the definition of buildings; and Muller et al. [MWH*06], which presented the CGA grammar construct; the field of procedural modeling of cities flourished with many interesting works. All these efforts resulted in the origin of commercial packages, like Esri's CityEngine [Esr14], or a module of Epic's UDK [Epi09], focused on procedural urban design. We refer the interested reader to the surveys by Watson et al. [WMV*08], and Vanegas et al. [VAW*10] for a more detailed overview of the state-of-the-art in urban procedural modeling. Some works have used inverse procedural modeling to extract a procedural model from examples of the desired productions; for instance in the method by Van Gool et al. [VGMM13]building photographies are segmented to detect semantic elements (windows, balconies, etc).

With respect to the technique presented in this paper, the work by Aliaga et al. [AVB08] is relevant in the sense that it presented a method to generate urban layouts based on examples, by extracting the street network and per-parcel aerial-view images from real data to then generate new layouts by synthesizing streets and images based on data from the example layouts. Later, this technique was extended by Nishida et al. [NGDA16] to a system where the user can interactively design a urban layout. The method starts the process by letting the user select either to design roads from scratch, or to start with any existing example patch taken from OpenStreetMap. In their method, roads are generated by growing a geometric graph, where two types of growth are selectively applied: example-based growth, which uses patches extracted from the source examples; and procedural-based growth, which uses the statistical information of the source example while selectively adapting the roads to the underlying terrain and the already generated roads. In their system, the user is free to define warping, blending, and interpolation operations to produce new road network designs. Lipp et al. [LSWW11] developed a technique for interactive modeling of procedural city layouts that allows intuitive manipulation using drag and drop operations. Taal and Bidarra [TB16] developed a method to populate urban landscapes with traffic signs by analyzing the street layout at a local (i.e., a block) level. Although these works are able to reproduce realistic layouts based on input examples, their objective is purely geometrical, without taking semantic information into account.

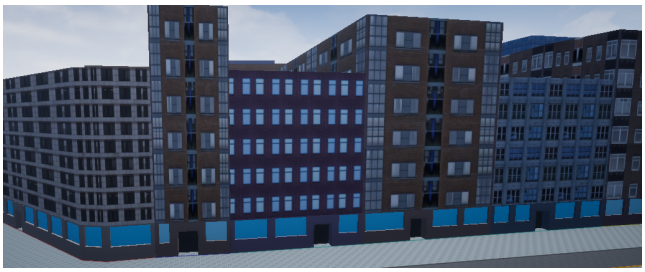

Figure 2: *Example of multiple CGA-generated buildings forced to use the same rule file. The main advantage of CGA is its adaptability to shapes and sizes, which in this case can be seen in the ground floor windows.*

On the other hand, Emilien et al. [EVC*15] presented WorldBrush, a system that, from an example input landscape, is able to learn parameters of distributions of elements (e.g., trees, grass, or rocks) constrained by the terrain's slope. Then, the system uses these parameters to consistently populate content in a larger landscape, using a copy-paste-like tool or with user-controlled interactive brushes mimicking traditional painting tools. In this work, the different types of elements are clearly identified and separated, but the objective is not to use this information any further than restricting the tools to operate on specific sets.

2.1. CGA

In the framework we propose in this paper, the generation of the building geometry is based on the popular rule-based system of Computer Generated Architecture (CGA) [MWH*06], and specifically the syntax variant used in the commercial CityEngine system [Esr14]. This brings us all the power of this procedural approach, and even though the current result examples use simple buildings with relatively low variety, skilled artists can unlock its full potential to generate highly-detailed geometric models with little work. Figure 2 shows as an example some buildings generated using the same set of rules.

The basic syntax of a CGA *rule-set* consists of a plain text file containing a list of rules. Each rule is on the form of the rule name

and an optional list of arguments, followed by an arrow, and the successor. The successor is a list of one or more rule names and operations. These items will be executed sequentially, and may also be called with arguments. An example of a rule definition is:

```
A -> B C(arg1, arg2, ...) op(arg)
```

As a *shape grammar* system, a rule is executed for a "shape" or part of it, in order to determine how it is modified. Therefore, rules are executed attached to a context, which in the CGA case this consists primarily of the *scope* and its attributes. The *scope* can be understood as an oriented bounding box (OBB), and its main attributes are the geometry (a primitive or a mesh), material, etc.

Some operations allow creating multiple results, each of which can trigger the execution of further rules. A few of those operations require the presence of a selector block, which specifies different successors for different cases. The decision of what item to execute depends on how the specific operation interprets the selector. The two most prominent examples of this are the *comp* operation, which executes a successor for each type of elements (faces, edges or vertices) of the current geometry, or the *split* operation that cuts the geometry at different offsets and executes a different successor for each interval. The syntax for these selector blocks is:

```
op(args) { selector1: successor1 | ... }
```

For a more complete reference of the language syntax and control structures, we refer the reader to the manual of CityEngine [Esr14]. Listing 1 shows an example of the syntax of a rule set (expanded with some of our new operations). Note that redundant white spaces and line breaks are ignored (except inside strings), and expressions, operands, and the call syntax are essentially the same ones as in the C or C++ programming languages.

3. Framework overview

Figure 3 shows the steps and the flow of data of our framework. Cities are generated in a two-step process: first, the generation of the city layout; and then the generation of the semantically-augmented geometry for each lot.

For the generation of the layout, we provide two alternatives: a fully procedural generation, or importing real-world GIS data. In our case the city layout consists of the set of lots (defined by their footprint or shape, location, and the lot usage) as well as the road network (a graph where each edge is a road).

Since our goal is to provide a semantic tagging system, not a new method for generating rich city layouts, for the fully procedural case we have implemented a very simple square-like block generation algorithm, which generates blocks of buildings in a regular grid shape. Some parameters can be tweaked, such as the number of blocks, number of buildings per block, lot dimensions, road size (or 0 for none). All resulting lots in this case are thus square-shaped. More sophisticated strategies, such as methods based on growing road networks, could be easily integrated into the framework, but this lies outside the scope of this work.

The second layout generation alternative is based on loading real-world urban data from standard sources (e.g., OpenStreetMap [HW08, Ope17]). Our system reads the layout information from an input GIS data file, for which we currently support

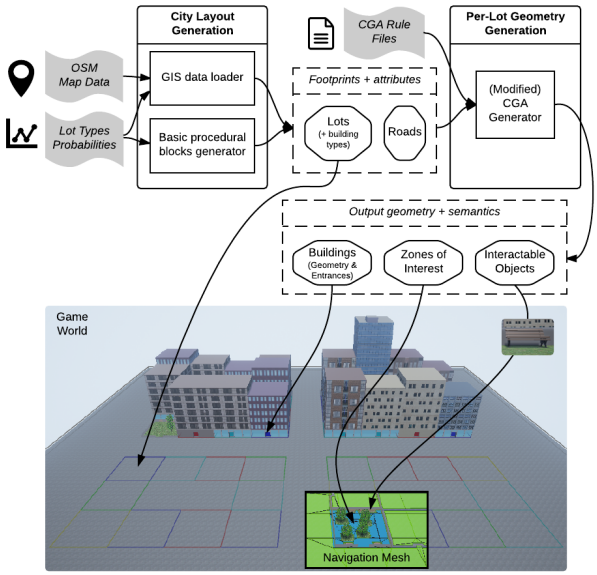

Figure 3: *Overview of the framework and the input and output data of each step.*

the OSM format (in XML syntax). OpenStreetMap was chosen because it is an open and widely supported format, for which data files are easily and freely obtainable, while it also supports semantic data. This semantic data contains metadata such as lot types, which indicates for example whether there is a restaurant, or a shop in the ground floor. The OpenStreetMap data is then processed in order to fix several geometric problems (e.g. incoherent vertices order that result in wrong normals, polygons with degenerate regions that need to be removed, successive collinear edges that are merged together, etc.), and also to extract the relevant data (such as the lot type) from the semantic information supported by the input format.

As Figure 3 shows, regardless of the city generation method being used, we have another input file which contains statistical probabilities for the different lot types. For the case of full procedural generation, the probabilities file is used to classify all the lots. For the second method, where we use GIS data, we found that in some zones there are buildings that lack zone usage information, so in these cases we complete the original data with building types distributed according to the probabilities given by this file.

For the generation of building or lot geometry, a custom implementation of CGA (Computer Generated Architecture [MWH*06]) is used, which generates building geometry by executing a set of grammar-based rules written by the user, whilst augmenting the results with semantic data. For this last purpose, and with the objective of providing more flexibility in the tagging operations, our system incorporates a few simple, but powerful new commands that, for example, enable the introduction of customizable labels at the finest level required. For the roads, we first convert the edges (lines) of the road network into 2D polygons, and then we use CGA to gen-

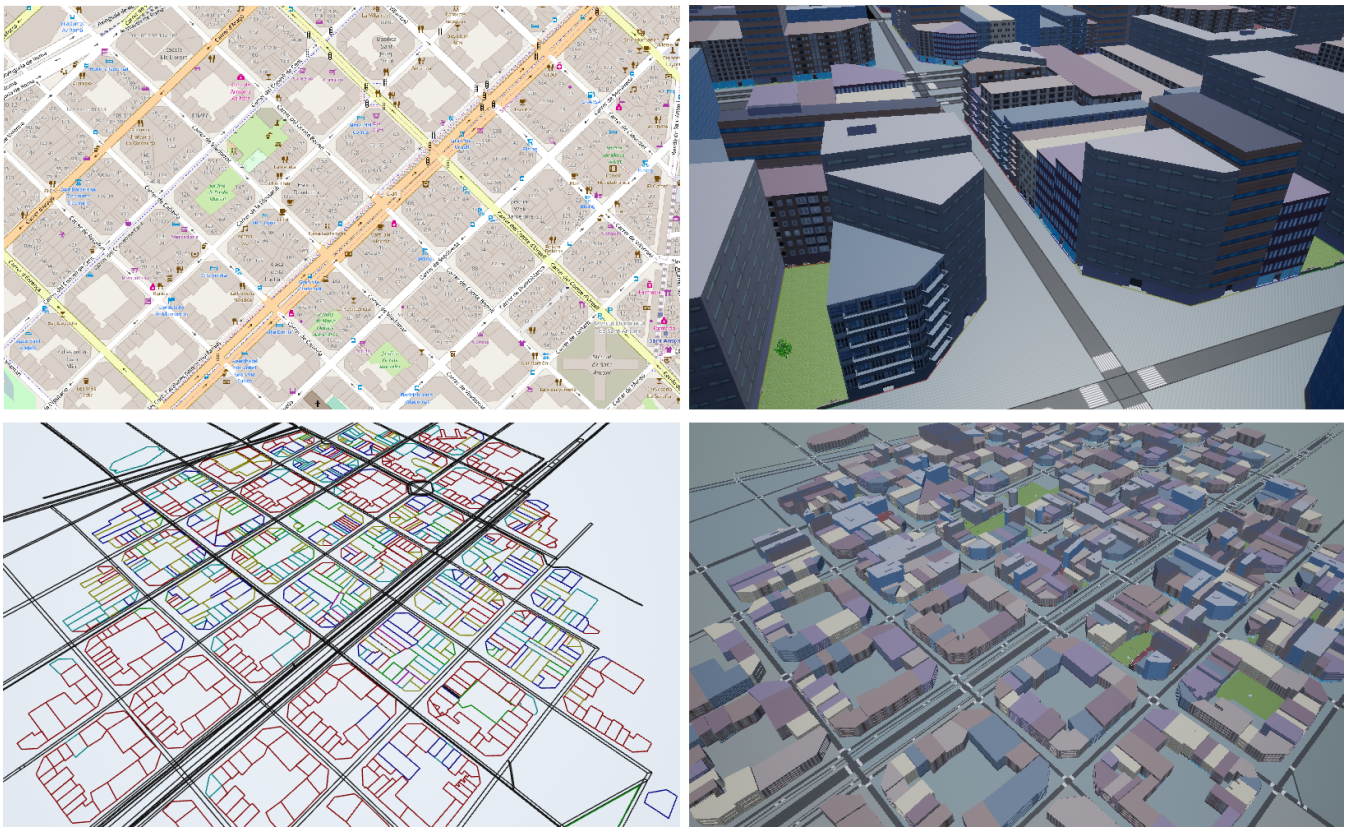

Figure 4: *Generation of a section of the city of Barcelona. Top Left: input OSM data as seen in the official web page. Bottom Left: the generated layout (lots and road network), where each color represents a different lot type (usage). Bottom Right: aerial view of result after the CGA generation. Top Right: a close-up of the result.*

erate its geometry. This allows us to create, for example, crosswalks near the intersections.

Figure 4 shows the data at the different steps of the framework, using the GIS data loader for a region of the city of Barcelona. From the input OSM data, lots and roads are extracted, and then geometry and semantics are generated for each. Notice also how there are some zones with a same lot type (represented by the color), with red corresponding to residential buildings.

4. Semantically-augmented per-lot generation

The generated cities need to consist of both the geometry and the *semantic* information that could be used later on, for example to drive the citizens' behavior. As opposed to previous works where cities and population are two independent elements with no connection other than the calculated navigation mesh (*navmesh*) and additional manually annotated information, our framework presents a direct connection between these two entities.

We extend the CGA specification for the generation of the geometry, to define different kinds of semantic elements within the process. The primary element is that, in our implementation, we can define within the lots one or multiple entrances that could be used as entry/exit locations to plan individual's trajectories. This allows, for example, to define a store on the ground level and an entrance to the residences in the upper floors.

Similarly, we can specify zones of interest during the generation of each lot, like park areas. This provides the flexibility to generate results such as a small building with a little interior park within the same lot, as opposed to having the entire lot occupied exclusively by a single large building or by a park.

Finally, we take advantage of the recursive refinement nature of CGA, which intuitively could be visualized as generating a tree of oriented bounding boxes that are not restricted to their parent box and have properties attached (e.g., material), and where the leaves correspond to the actual generated geometry. We allow subsets of the generated geometry (i.e. subtrees) to be tagged with user-defined tags, which are then created as separated objects. This can be used for example to mark some of the generated geometry as a bench or a lamppost.

We also reuse the "instance" (i.e., "insert") operation of CGA, but we modify its implementation in order to keep track of which model to instantiate, and do it later, as opposed to immediately

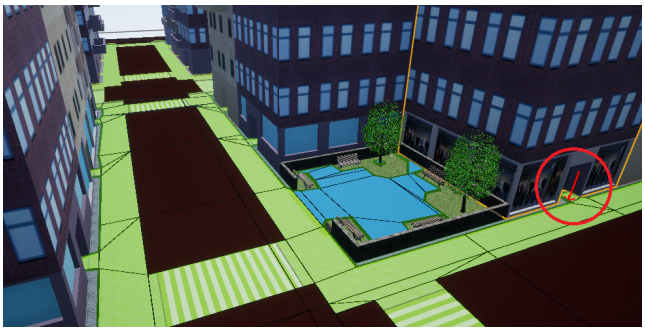

Figure 5: *Visualization of the supported semantics: definition of navmesh areas (park in blue, impassable in dark red, green for default), creation of objects during the CGA generation (benches and trees), and building entry points (the red arrow inside a circle on the right-hand side).*

```
@StartRule
Lot   --> split(x) { 2: House | ~1: Park }

House --> extrude(8)
          comp(f) { front: Front | all: Wall }
Front --> split(z) {~1: Wall | 1: Door | ~1: Wall}
Door  --> entrance("house")
          extrude(-1)
          comp(f) { back: NIL | all: Wall }

Park  --> zone("park")
          split(x) {~1: NIL | 0.2: Bench}

@Object("bench")
Bench --> s('1, '1, '0.5)
          extrude(0.2)
```

Listing 1: *Example of a semantically augmented CGA rule file.*

copying its mesh into the scope geometry. This allows us to prevent duplicating such typically complex geometry, and take advantage of the instanced rendering feature provided by game engines. Furthermore, this enables us to keep metadata such as points of interaction (e.g., sitting positions) that can be defined on a per-model basis by the user, instead of at each occurrence.

Figure 5 shows a representation of the different supported semantic elements in the final city as generated by our framework. In Listing 1 we show a simple but complete example of a CGA rule file extended with our new semantic elements. It splits the lot in two along the X axis, generating a building with an entrance on one side, and a zone marked as a park which also contains a bench on the opposite side.

The most important aspect of our city generator is that, as the city is being automatically created, it tags along different elements and zones with semantics that could be evaluated by a simulation module to drive the city dynamics. Also, the city semantics are combined with the final navigation mesh, allowing the user to obtain a navigation mesh that has been automatically enhanced with relevant information to drive pathfinding and behaviors (i.e., destination positions, find a closest park to rest, etc).

Once the semantically augmented city is created, it is processed by the target simulation engine (or game engine), which will store the generated semantic components (e.g., buildings, zones of interest or interactable objects) to be used during the simulation, and to generate the navigation mesh taking into consideration such information.

To further enhance the flexibility of our framework, there is an additional layer of interaction at the end of the process. This layer allows users to inspect the generated city and the defined areas, allowing them to tune the parameters and the rules if needed. At this point it is also possible to further refine the city information or content by manually editing the results (e.g., to add specific unique landmarks such as monuments). This provides a fine level of control over the final result.

4.1. Extensions to CGA rule syntax

Formally, the aforementioned extensions to CGA consist of the introduction of two new operations into the CGA language that add semantic information, which can be written in the body of the rules, and affect the current *CGA scope* (i.e., the current "bounding box"). These operations are:

- `entrance("btype")`, *btype* $\in$ *{school, house, shop, workplace, leisure, ...}*: defines an entrance point to a building of the user-specified type, at the origin position of the current CGA scope. The building type is a string, whose possible values are defined by the user and depend on the intended applications.
- `zone("ztype")`, *ztype* $\in$ *{park, road, bike lane, ...}*: defines a zone of interest of a user-specified type spanning the current scope. If the current scope is a 2D scope (i.e. polygon), it will be extruded into a prism of infinite size along its normal direction. Like in the previous operation, the possible zone types are defined by the user.

In addition, a new CGA annotation (i.e. a rule metadata tag that can be written just before a rule) is supported:

- `@Object("otype")`, *otype* $\in$ *{bench, light, fountain, ...}*: tags the production of the rule and all its sub-productions as a separate entity to the lot geometry, and marks it with the specified user-defined tag. The tag may be any string, and its meaning depends on the user.

As indicated earlier, the output of this extended CGA generation is not just a single final mesh, but a tree-like hierarchy of generated geometry, which also contains the information generated by the previous operations. In section 5 we provide details regarding how this tree is used to generate a city scene as part of the integration with Unreal Engine 4.

5. Integration with Unreal Engine 4

Our prototype of the framework has been integrated into Unreal Engine 4 [Epi12], which is a powerful game engine that offers state-of-the-art features and tools, as well as an editor program. We considered other alternatives such as Unity 2017, also available free

of cost for non-commercial projects, but ultimately we choose Unreal Engine 4 because the possibility of accessing its source code as well as its novel features, and support of programming in C++.

For the generation of the buildings geometry, we used a custom implementation of a CGA interpreter. To implement it we used Flex [E*87] and Bison [C*88] to generate the code to parse the rule files language. The interpreter does not depend on Unreal Engine 4, and can be used in a standalone manner. Nonetheless, we also implemented a module to help managing CGA rule files inside the Unreal Engine editor. Among its features, it accepts rule files as a new "asset"" type, showing a preview of the generation result on the Asset Explorer view. Moreover, it can generate the CGA shapes as a tree of nested components in an object, which allows the user to inspect the nodes of the CGA tree and render for each one its corresponding CGA scopes. This feature is very helpful as it allows the user to debug while writing the rule files.

To build the city scene in the engine, the hierarchy resulting from the modified CGA generation is traversed, and engine "entities" are created accordingly. Unreal Engine is based on a entity-component system (ECS), and we create an entity for each building. This entity is assigned a mesh component containing the building main geometry, as well as additional custom components such as a marker for entrance positions (which could be then queried to e.g. drive agent behaviors). And for models loaded using the instantiate ("i") operation, a "static mesh component" is created in order to take advantage of the instanced rendering capabilities of the engine. In the case of zones, a "nav modifier volume" is created. Finally, for nodes in the hierarchy marked with an @Object tag, an entity separated from the building geometry is created.

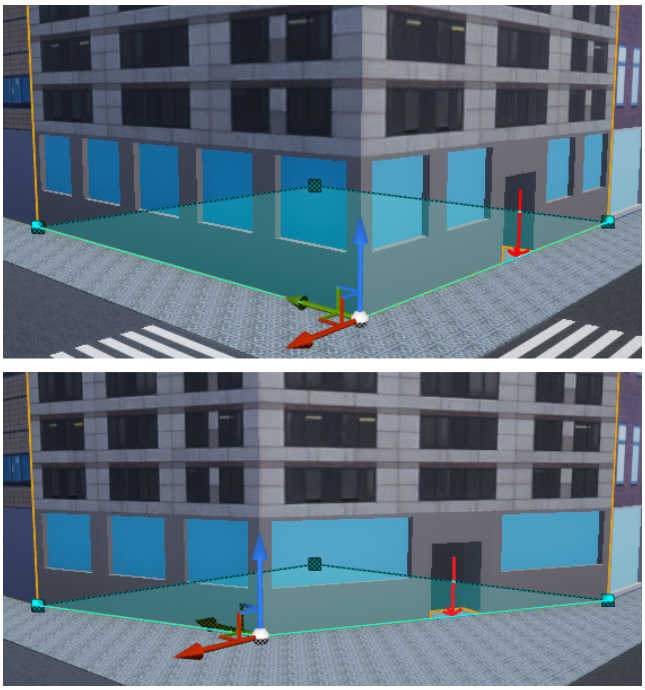

Figure 6: *Example of editing the city layout directly in the Unreal Engine 4 Editor, by modifying the footprint of a building. Notice how thanks to CGA generation, the generated geometry and semantics, and in particular the street-level windows and the entrance, adapt automatically accordingly.*

5.1. Manual edition

Currently there are two different possibilities regarding how the user can manually modify the resulting cities in our prototype.

The first and most obvious choice is by editing the input data, i.e. the OSM file used as input. For this purpose, there are several existing tools to edit the 2D OpenStreetMap data, such as JOSM [JOS17]. Using this tool the user may edit existing data, but also create a new map from scratch.

Using editors such as JOSM the user has the flexibility of not only adding new elements to the map (such as buildings and roads or punctual objects), but also to attach metadata to the existing or new elements in the form of tags (string key-value pairs). In this way they may specify semantic data such as the buildings usage (residential, bar, shop, etc.) or even opening hours, but also information on its shape such as the number of floors or architectonic style. In the case of the roads the user can add tags with information about the type of road, for example secondary, main, or road with bike lane.

The second choice for editing consists of modifying directly the results in the Unreal Editor. The generation of the city is performed in two sequential steps: (1) generation of layout in the form of lots and streets, and (2) generation of the geometry for each lot. Hence, the user may alter the layout directly before the geometry generation, for example by changing a lot type, shape or location, or CGA rule file used to generate its geometry; or even add unique buildings such as monuments. Another possibility consists of altering the final result by adding new semantically tagged items such as benches, lampposts, etc. Figure 6 shows on the right, an example of the user editing the shape of a lot. The user has moved a vertex of a quadrangular footprint, and as result the building geometry regenerated with the windows at the street level adapting to the new walls.

6. Results

We have shown results of our framework throughout the paper. Figures 1 and 2 displayed mostly the visual output of our system. However in this section we want to emphasize the semantic information included, as this is the main novelty of our work.

In Figure 4 we showed the data at the different steps of the framework, using a region of the city of Barcelona as input. In the figure we can also see the final result and a close-up, in which we can observe the different kinds of buildings being generated. We can see the correspondence of lots such as parks from the input to the end result, as well as the relative lack of shops and services (as denoted by the presence of fewer icons in the input map) in primarily residential areas. Note that the colors used in the layout representation (bottom left), indicate the semantic information regarding the lot usages, so for example green is used to tag parks and red is

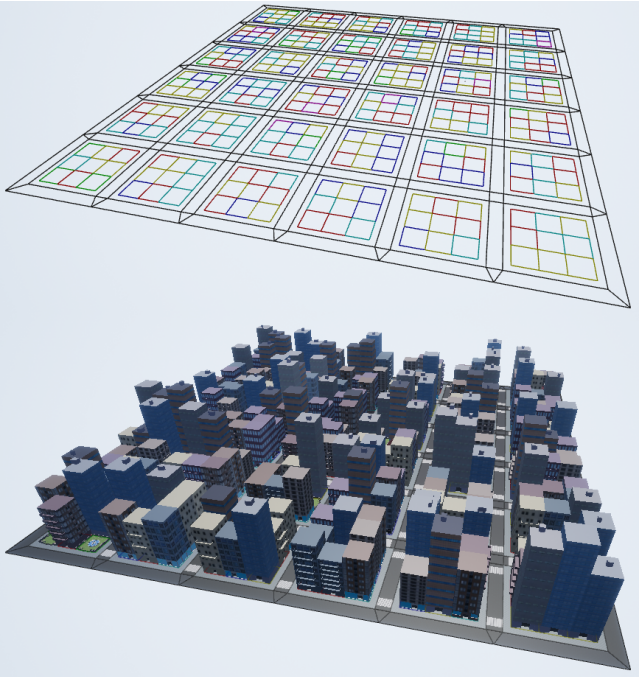

Figure 7: *Aerial view of a fully procedurally generated city, using the procedural square-block layout generator. Colors indicate the semantic information associated with each building or lot.*

used to tag residential buildings. Other colors represent semantic information of those buildings which have a business or service at the ground floor (bar, school, shop, etc). This information is based on the tags existing in the input map (Figure 4 top left), or automatically generated under an user-defined lot type distribution if missing in the input data.

Figure 7 shows an example of a generated city using our fully procedural approach with the simple square-blocks algorithm. As in the previous example, colors are used to visualize semantic information which in this case has been fully procedurally created and attached to the lots. Figure 8 shows a close-up from a street from this same city, showcasing some semantic objects such as the benches in the parks, as well as different kinds of buildings, and shops. In our 3D visualization we use different geometry and textures on the windows of the street level, to indicate the semantic information associated with the ground floor of each building (e.g. offices have only a door, bars have half-wall windows, and shops have full-wall windows width textures representing the shop type). Figure 5 showed a visual representation of the semantic elements of another part of the same city. It prominently features the semantic information associated with the navigation mesh, which can be relevant to drive the heuristic of the path finding algorithms for simulating virtual humanoids (for example, assigning higher weight to crossing the street outside the sidewalks or pedestrian crossings).

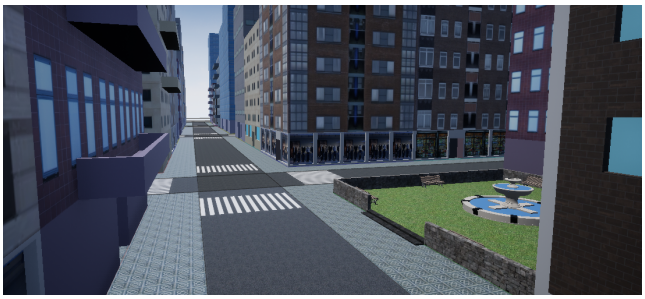

Figure 8: *Close-up view of a street of the city in Figure 7.*

7. Conclusions and Future Work

In this paper we presented a framework that expands the procedural generation of virtual cities, by augmenting the process to add semantics to its elements. This system does not output a single mesh as result, but instead, it provides a hierarchy of entities that include semantic metadata, which prevents the loss of data that may be of interest for other applications. For example, such semantic can be of high value to run simulations of virtual inhabitants with purposes. Our main contribution is the augmentation of the CGA procedural modeling process to include such semantic information.

Our framework focuses on the generation of a semantically augmented city, aiming at enriching the simulation of agents. Therefore a future line of work is studying methods to perform such simulation for a multi-agent system, with the aim of driving more realistic and complex behaviors. This is a topic of interest in the fields of city planning and for the development of solutions for smart cities. Another relevant fields of application includes videogames and film industries where more believable crowds are needed.

Currently we use a very simplistic approach for the generation of the synthetic cities layout, so alternative techniques could be implemented from the wide literature on this topic, such as procedurally growing road networks [NGDA16].

On the semantic side, an interesting extension would be allowing to "copy and paste" geometric or semantic characteristics of a certain zone into another zone. This would solve one of the current issues with the input from OSM data, which is the lack of information in certain zones, or even the representation of the buildings in a blocks as a single building. Examples of data which could be copied are lot types (density, distance from lots of the same type, distance from other types), building heights or styles, or individual semantic objects such as ATM machines. This last kind could be based on probabilistic distributions such as the ones used in WorldBrush [EVC*15].

Acknowledgments

This work was partially funded by the TIN2014-52211-C2-1-R and TIN2014-52211-C2-2-R projects from Ministerio de Economía y Competitividad, Spain. Otger Rogla has a FPU grant, funded by the Ministerio de Educación, Cultura y Deporte, Spain.

References

[AVB08] ALIAGA D. G., VANEGAS C. A., BENEŠ B.: Interactive example-based urban layout synthesis. In *SIGGRAPH Asia '08: ACM SIGGRAPH Asia 2008 papers* (New York, NY, USA, 2008), ACM, pp. 1–10. doi:http://doi.acm.org/10.1145/1457515.1409113. 2

[BMJ*11] BENEŠ B., MASSIH M. A., JARVIS P., ALIAGA D. G., VANEGAS C. A.: Urban ecosystem design. In *Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2011), I3D '11, ACM, pp. 167–174. URL: http://doi.acm.org/10.1145/1944745.1944773, doi:10.1145/1944745.1944773. 1

[C*88] CORBETT R., ET AL.: GNU Bison. http://savannah.gnu.org/projects/bison/, 1988. 6

[E*87] ESTES W., ET AL.: Flex. https://github.com/westes/flex, 1987. 6

[Epi09] EPIC GAMES: Unreal Development Kit (UDK). http://udk.com, 2009. 2

[Epi12] EPIC GAMES: Unreal Engine 4. https://www.unrealengine.com, 2012. 5

[Esr14] ESRI: CityEngine. http://www.esri.com/software/cityengine, 2014. 1, 2, 3

[EVC*15] EMILIEN A., VIMONT U., CANI M.-P., POULIN P., BENES B.: Worldbrush: Interactive example-based synthesis of procedural virtual worlds. *ACM Trans. Graph. 34*, 4 (July 2015), 106:1–106:11. URL: http://doi.acm.org/10.1145/2766975, doi:10.1145/2766975. 2, 7

[GDAB*17] GARCIA-DORADO I., ALIAGA D., BHALACHANDRAN S., SCHMID P., NIYOGI D.: Fast weather simulation for inverse procedural design of 3d urban models. *ACM Trans. Graph.* (2017). 1

[HW08] HAKLAY M. M., WEBER P.: Openstreetmap: User-generated street maps. *IEEE Pervasive Computing 7*, 4 (Oct. 2008), 12–18. 3

[JOS17] JOSM: JOSM extensible editor. https://josm.openstreetmap.de/, 2017. 6

[LSWW11] LIPP M., SCHERZER D., WONKA P., WIMMER M.: Interactive modeling of city layouts using layers of procedural content. *Computer Graphics Forum (Proceedings EG 2011) 30*, 2 (Apr. 2011), 345–354. 2

[MWH*06] MÜLLER P., WONKA P., HAEGLER S., ULMER A., VAN GOOL L.: Procedural modeling of buildings. *ACM Trans. Graph. 25*, 3 (2006), 614–623. 1, 2, 3

[NGDA16] NISHIDA G., GARCIA-DORADO I., ALIAGA D. G.: Example-driven procedural urban roads. *Comput. Graph. Forum 35*, 6 (Sept. 2016), 5–17. 2, 7

[Ope17] OPENSTREETMAP CONTRIBUTORS: Planet dump retrieved from http://planet.osm.org . http://www.openstreetmap.org, 2017. 3

[Pat12] PATOW G.: User-friendly graph editing for procedural modeling of buildings. *IEEE Computer Graphics and Applications 32* (2012), 66–75. 1

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

[Sid12] SIDEFX: Houdini 12, 2012. http://www.sidefx.com. 1

[STBB14] SMELIK R. M., TUTENEL T., BIDARRA R., BENES B.: A survey on procedural modelling for virtual worlds. *Computer Graphics Forum 33*, 6 (2014), 31–50. URL: http://dx.doi.org/10.1111/cgf.12276, doi:10.1111/cgf.12276. 1

[TB16] TAAL F., BIDARRA R.: Procedural generation of traffic signs. In *Eurographics Workshop on Urban Data Modelling and Visualisation* (dec 2016), Eurographics, Eurographics. URL: http://graphics.tudelft.nl/Publications-new/2016/TB16. 2

[VAW*10] VANEGAS C. A., ALIAGA D. G., WONKA P., MÜLLER P., WADDELL P., WATSON B.: Modelling the appearance and behaviour of urban spaces. *Comput. Graph. Forum 29*, 1 (2010), 25–42. 2

[VGDA*12] VANEGAS C. A., GARCIA-DORADO I., ALIAGA D. G., BENES B., WADDELL P.: Inverse design of urban procedural models. *ACM Trans. Graph. 31*, 6 (Nov. 2012), 168:1–168:11. 1

[VGMM13] VAN GOOL L., MARTINOVIC A., MATHIAS M.: Towards semantic city models. *Proceedings of the 54th Photogrammetric Week, Stuttgart, Germany* (2013), 11–15. 2

[WMV*08] WATSON B., MÜLLER P., VERYOVKA O., FULLER A., WONKA P., SEXTON C.: Procedural urban modeling in practice. *IEEE Computer Graphics and Applications 28* (2008), 18–26. 2

[WWSR03] WONKA P., WIMMER M., SILLION F., RIBARSKY W.: Instant architecture. *ACM Transaction on Graphics 22*, 3 (July 2003), 669–677. Proceedings ACM SIGGRAPH 2003. 1, 2