

# Generating Consistent Buildings: a Semantic Approach for Integrating Procedural Techniques

Tim Tutenel, Ruben M. Smelik, Ricardo Lopes, Klaas Jan de Kraker and Rafael Bidarra

## Abstract

Computer games often take place in extensive virtual worlds, attractive for roaming and exploring. Unfortunately, current virtual cities can strongly hinder this kind of gameplay, since the buildings they feature typically have replicated interiors, or no interiors at all. Procedural content generation is becoming more established, with many techniques for automatically creating specific building elements. However, the integration of these techniques to form complete buildings is still largely unexplored, limiting their application to open game worlds. We propose a novel approach that integrates existing procedural techniques to generate such buildings. With minimal extensions, individual techniques can be coordinated to create buildings with consistently inter-related exteriors and interiors, as in the real world. Our solution offers a framework where various procedural techniques communicate with a moderator, which is responsible for negotiating the placement of building elements, making use of a library of semantic classes and constraints. We demonstrate the applicability of our approach by presenting several examples featuring the integration of a façade shape grammar, two different floor plan layout generation techniques, and furniture placement techniques. We conclude that this approach allows one to preserve the individual qualities of existing procedural techniques, while assisting the consistency maintenance of the generated buildings.

## Index Terms

Façade shape grammars, floor plan generation techniques, procedural modeling of buildings, semantic modeling.

## I. INTRODUCTION

Games increasingly take place in highly detailed virtual worlds, often featuring complex urban environments. Notable recent examples include *Assassin's Creed*, *Elder Scrolls* and *Grand Theft Auto* series, where players explore extensive cities filled with detailed and visually appealing façades. Typically, these cities are modeled by hand, requiring an enormous amount of effort and huge production costs for game development studios. *Grand Theft Auto IV*, for example, took over 1000 people, more than three years and \$100 million to complete.

Manuscript received March 23, 2011.

Tim Tutenel, Ricardo Lopes and Rafael Bidarra are with the Computer Graphics Group, Delft University of Technology, Delft, The Netherlands, email: {t.tutenel|r.lopes|r.bidarra}@tudelft.nl.

Ruben M. Smelik and Klaas Jan de Kraker are with the Modelling, Simulation and Gaming Department of TNO, The Hague, the Netherlands, e-mail: {ruben.smelik|klaas\_jan.dekraker}@tno.nl.

This research is supported in part by the GATE project, funded by the Netherlands Organization for Scientific Research (NWO), and by the Portuguese Foundation for Science and Technology (FCT), under grant SFRH/BD/62463/2009.

In such games, the whole virtual environment is required to have a visually stunning appearance, regardless of whether buildings are important to the main storyline or not. However, the lack of time and resources constrains game developers to create the unimportant collateral buildings in a fast and minimal way, for example, featuring manually detailed façades but either no interiors or a fixed set of interiors, replicated all over the city.

This lack of integral, '*enter-anywhere*' buildings is especially noticeable in recent open world games, such as *Red Dead Redemption* or *Fable III*, which are raising the bar on sandbox-based gameplay. In fact, their game mechanics offers more and more freedom to roam around, encouraging to divert from the main objective to explore the environment, or even use it in more creative ways. The importance players are giving to spontaneous exploration is also evidenced by the success of exploration-based game *Minecraft*. Precisely in this type of games, the above mentioned finalized buildings could significantly improve the gameplay. Not only could the motto '*exploration-for-the-sake-of-it*' be then much better realized, but urban environments could be made fully *accessible* for players to use them as they please. In *Assassin's Creed*, for example, strategic, designer-placed hay stacks could be used to hide from enemy patrols. With '*enter-anywhere*' buildings, any house could potentially serve that purpose.

However, the cost of manually modeling interiors for every building is simply unbearable. Hence the urgent need and interest for methods that can automatically create such buildings. Procedural content generation techniques are expected to play an important role in solving this problem, even though they are often far from matching the expressive range of manual modeling. In particular, it seems very affordable to have procedural methods automatically generate large portions of content, regardless of whether this is ready for (pre-)production or it is only a basis for being further worked out by an artist.

More formally, we need buildings exhibiting two characteristics, each of them presenting its own challenge:

- 1) **complete** buildings, i.e. '*enter-anywhere*' buildings consisting of not only a façade, but also interiors, stairs, furniture, etc. The main challenge is the time it takes to produce all that content, which recommends the use of procedural content generation methods.
- 2) **congruent** buildings, i.e. buildings with plausible elements in harmony and without conflicting elements (in Section III we will discuss several kinds of conflicts). The main challenge here is that most current procedural techniques generate just one type of building element, without taking into account the remaining elements.

Within the context of this research, including the title of this article, buildings that are both complete and congruent are designated **consistent** buildings.

Current research in the area focuses on procedural methods for generating many aspects or *elements* of urban environments, including road networks, building lots, façades, roofs and floor plans. However, the generation of these elements poses its own challenges, as evidenced by the use of very distinct techniques to solve each of them. Unfortunately, the integration of all those procedural techniques to yield a combined output is still in its infancy [1].

There are basically two approaches to attempt such integration: (i) for each type of building, develop a *new* dedicated generator that bundles a few techniques selected, implemented and integrated in an *ad hoc* fashion for that specific purpose; a recent example of this is the dedicated approach to generate dwelling houses proposed in [2], which will be further analyzed in Section II; or (ii) develop a generic framework that is able to integrate a variety

of *existing* techniques, already mature, implemented and proven, into a versatile procedural content generator.

We argue that the second approach above is superior to the first, regarding both flexibility, expressive power and ease of use. In this article we propose such a framework, which is able to integrate any procedural techniques, and combine them to generate all sorts of *consistent* buildings, as efficiently as a dedicated approach.

With dedicated solutions, the strong coupling among building elements is enforced by *ad-hoc* mechanisms for maintaining the consistency of the resulting building. Our main contribution, in contrast, is a semantic approach that brings this consistency management among building elements to an independent central framework, without significantly harming performance. The working of this framework is inspired on our semantic modeling background [3], [4].

With this approach, procedural techniques have a common framework on which they are led to collaborate in the generation of consistent buildings. Moreover, this approach is not limited to façades and floor plans: with minor modifications, any existing procedural techniques deemed suitable for contributing to complete building generation (*e.g.* roof or lot shape generation, furniture placement) can be loosely coupled and integrated in the generation process.

This versatility in reusing and recombining existing procedural techniques brings about other advantages, when compared to dedicated approaches. For one, developers can focus on local specialization, *i.e.* concentrate on improving individual generation techniques for a building element, while the framework handles the integration of its output into the complete building. Also, replacing old or under-performing techniques for specific building elements, or trying out new ones, becomes much easier, increasing development flexibility.

This article is structured as follows. In Section II we survey different current techniques contributing to the generation of buildings, with a special focus on techniques that still lack integration: façade and floor plan generation. In Section III, we describe in detail our semantic approach for integrating procedural techniques. In Section IV, we show several results of this approach, using examples where a façade grammar, a floor plan and a furniture placement method are integrated. In Section V we briefly discuss the approach in the light of its results. Finally, in Section VI we present our conclusions and future work.

## II. RELATED WORK

Procedural generation techniques have been proposed for almost every aspect of virtual worlds, ranging from vast landscapes (see *e.g.* [5], [6]) to urban environments (see *e.g.* [7], [8], [9], [10]). In urban settings, extensive research has been done towards procedural buildings. So far, most researchers proposed independent methods to generate the exterior, *i.e.* the façade, and the interior, *i.e.* the floor plan, of buildings. There are some recent techniques that attempt to integrate these two aspects, although showing some limitations. They are essentially stand-alone methods that: (i) focus more on one aspect, neglecting the other, and (ii) do not re-use existing methods.

### A. Façade generation

In the field of automated generation of building façades, *L-systems* were among the first techniques to be proposed [7]. These rewriting systems create buildings by manipulating an initial arbitrary ground plan (a lot shape) with

transformation and extrusion modules.

To obtain more interesting building shapes, several approaches have been devised. Wonka *et al.* [11] introduced the concept of *split grammar*, a formal context-free grammar designed to produce building models. The split grammar resembles an L-system where shapes are primitive elements rather than symbols. Coelho *et al.* [12] proposed an urban modeling process that is based on L-systems as well. This process generates, from external data, a tree-like description of the overall scene structure. L-systems are used to generate detailed building models that emerge from the abstract set of data.

In recent years, a more specialized approach, the *CGA shape grammar*, has been applied to building façades by Müller *et al.* [13]. Shape grammars have been used and described before, especially in the architectural domain [14], [15], [16]. Architects have described shape grammars as languages of design, supported by a vocabulary of shape rules. Shape rules are specified as spatial relations, where a shape on the right side of the rule is produced and replaces the symbol on the left side (depicting when the rule can be applied).

In Müller *et al.*'s case [13] and unlike a split grammar, the shape grammar uses context-sensitive rules which allow the possibility of modeling roofs and rotated shapes. They start with a union of several volumetric shapes (the building boundary) which is divided into floors. The resulting façades are further subdivided, through shape rules, into walls, windows and doors. Yong *et al.* [17] also use an extended shape grammar, but they start at the city level, producing streets, housing blocks, roads, and, in further productions, houses with components such as gates, windows, walls, and roofs. Shape grammars have become the most accepted technique for generating building façades, as evidenced by its commercial release [18]. Epic Games also included in their commercial game engine, *Unreal Engine 3* [19], a procedural artist-driven tool for constructing buildings used in the development of city-based games [20]. The procedural system uses rulesets, similar to shape grammar rules, to split façades into scopes and automatically place meshes on them.

More recently, Müller *et al.* [21] used a very different approach for constructing building façades. Their method takes an image of a real building façade as input and is able to reconstruct a detailed 3D façade model, combining imaging and shape grammar generation techniques. Chen *et al.* [22] also proposed a method for creating building façades from images, but in this case using hand sketches as input.

On a different direction, Greuter *et al.* [23] proposed an approach where a primitive form of the integrated generation of both façades and floor plans was considered. Initially, they create a floor plan by combining several primitive 2D shapes, which are then extruded to different heights. This approach is most useful for simple office buildings. Although the concept of a generated floor plan is present, it is only used for extruding building façades and not as a room layout.

Although all of the above approaches can generate visually convincing building façades, Finkenzeller and Bender [24], [25] note that semantic information, regarding the role of each shape within the complete building, is missing. They propose to capture this semantic information in a typed graph, so that detailed building façades (doors, windows, balconies, cornices, ornaments) can be generated, in different styles, and applied to the same building outlines. Starting with a rough building outline, building style graphs can be applied to this model, resulting in an intermediate

semantic graph representation of the building. In the last step, geometry is created based on the intermediate model, and textures are applied, resulting in a complete 3D building.

### B. Floor plan generation

To create complete buildings, interiors must be added to the façade. The procedural generation of building floor plans, *i.e.* suitable inner room layouts, has been the focus of several researchers.

Rau-Chaplin *et al.* [26] show that shape grammars, often applied to building façades, can also create floor plans. In this case, shape grammars are used to create a *plan schema* containing basic room units. These individual room units are recognized and grouped to define functional zones like public, private or semi-private spaces. Individual functions are then assigned to each room, which are filled with furniture, by fitting predefined layout tiles from a library of individual room layouts.

On a different direction, Hahn *et al.* [27] present a subdivision method tailored for generating, on the fly, office buildings. The initial building structure is split up into a number of floors. On each of them, further subdivisions are applied to create a hallway zone and individual rooms. A notable feature of this method is that floors and rooms are generated or discarded based on the player's position. Re-using the same random seed in the procedure assures that discarded rooms can be properly restored.

Marson and Musse [28] also introduce a room subdivision method, but based on *squarified treemaps*. They start with the basic 2D shape of the building and a list of rooms, with desired area and functionality. Treemaps recursively subdivide an area into smaller areas, *e.g.* building shape, functional zones, rooms. In a final step, corridors are automatically created to connect unreachable rooms.

Martin [29] proposes a graph-based method, in which nodes represent the rooms and edges correspond to connections between rooms (*e.g.*, a door). Public, private and stick-on rooms (*e.g.* closets, pantries) are gradually added to the graph by a user-defined grammar. This graph is transformed to a spatial layout, and for each node, a specific amount of "pressure" is applied to make the room expand to the desired size. Lopes *et al.* [30] also propose an expansion-based method, which grows rooms in a geometric grid representing the building lot. The initial placement of room seeds is determined by a constraint solving algorithm that takes room adjacencies, connectivities and functional zones into account.

Tutenel *et al.* [31] applied a generic semantic layout solving approach to expansion-based floor plan generation. In this approach, every type of room is mapped to a class in a semantic library and for each of these classes relationships can be defined. In this context, relationships will define room-to-room adjacency. However, other constraints can be defined as well, *e.g.* place the kitchen next to the garden, or the garage next to the street. For each room to be placed, a rectangle of minimum size is positioned at a location where all defined relation constraints hold, and all these rooms expand until they touch each other.

Charman [32] gives an overview of constraint solving techniques that can be applied to room layout generation, if seen as a space planning problem. For example, the planner the author proposes works on the basis of axis-aligned

2D rectangles with variable position, orientation and dimension parameters, for which users can express geometric constraints, possibly combined with logical and numerical operators.

Merrel *et al.* [2] recently proposed a method for generating residential building layouts. Although this approach creates complete buildings, it is highly focused on floor plan generation. The authors use a Bayesian network, trained with real-world data, to expand a set of high level requirements (*e.g.* number of rooms) into a complete architectural program (*e.g.* room adjacencies, area and aspect ratio). These architectural programs are then realized into the 2D shapes of the floor plans, through stochastic optimization over the space of possible building layouts. 3D models are generated from different style templates to fit the structure of the floor plan, including external windows, doors and roofs. Their results are different from the integration approach we propose, since their method: (i) is specific for generating residential buildings, (ii) cannot create specific façade patterns and appearance and (iii) the façade always emerges from the floor plan, and, therefore, cannot steer the generation process.

### III. SEMANTIC INTEGRATION APPROACH

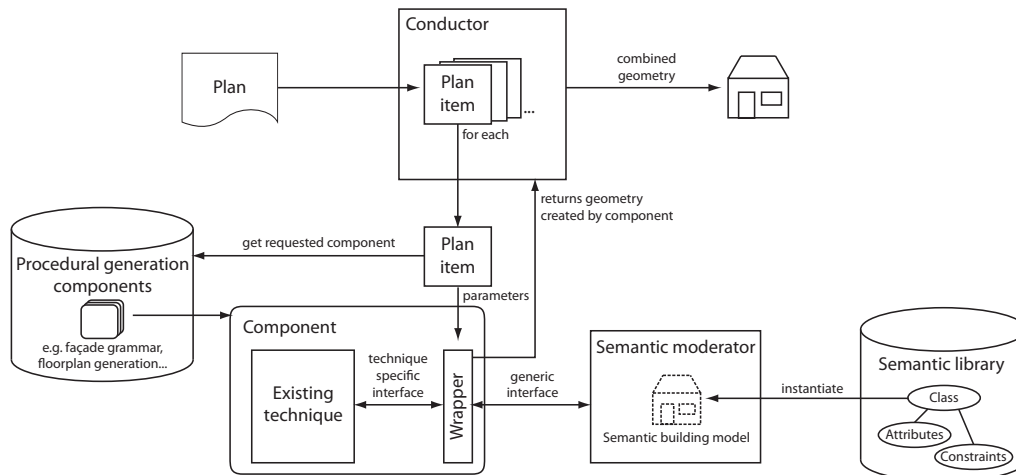


Fig. 1. Framework for integrating procedural techniques: *moderator* (with *semantic library* and *generic interface*), *components*, *wrappers*, *conductor* and *plan*

In this section, we describe in detail our semantic approach to integrate existing procedural techniques for generating consistent buildings, as defined in Section I, *i.e.* buildings consisting of different elements, without any conflict among them. Typically, each procedural technique is able to generate one specific element of a building (*e.g.* façade, floor plan, furniture, lot shape), but mostly without much regard for other building elements. Therefore, the main challenge of integrating those individual components is foremost to watch over the consistency of the building, either avoiding or properly handling any conflicts arising among building elements.

The main idea behind our approach is to establish a semantic moderator, which shares relevant building information with the individual procedural components, so that they can make good and timely decisions. This information, combined by the moderator into a unified *semantic model* of the building, forms the basis for the advice that it provides to individual components in order to avoid conflicts, *i.e.* inconsistent results. In our approach, we distinguish three categories of building elements conflicts:

- *intersection* conflicts, occurring when building elements that should not intersect each other, overlap in some way. For example, façade windows should not intersect inner walls, furniture should not obstruct inner doors, etc.
- *functional* conflicts, occurring when building elements with incompatible roles are associated. For example, bathrooms should not have the same type of window as bedrooms.
- *exclusion* conflicts, occurring when a *required* unique building element is placed such that it becomes impracticable in the resulting building, and has to be removed from it. For example, a required fireplace should only be placed on one of the possible locations where it has a feasible path to the (façade or roof) chimney. This conflict is particularly problematic with components that do not allow any backtracking, which unfortunately is often the case.

Fig. 1 outlines the framework architecture to support this integration approach. The various procedural components are made available through a wrapper interface and are invoked according to a building plan. The moderator, in turn, helps prevent the conflict types mentioned above, managing the communication with the procedural components, and providing them with building advice. In the following paragraphs, we explain this framework in detail.

#### A. *Semantic moderator*

The semantic moderator is responsible for watching over the consistency of the integrated building, by examining and approving the requests of each procedural component. For this, it maintains a semantic building model, which represents all building elements, including their attributes and constraints, by means of *semantic elements*. Each of these semantic elements is an instance of a class described in the so-called *semantic library* [33], and carries therefore all its semantics.

The semantic library provides a hierarchical class database, partly based on the WordNet ontology [34], where each class specifies and represents object semantics, *i.e.* all information, beyond its 3D geometric model, that helps convey the meaning and the role of an object in the virtual world. This includes its attributes, properties, services, and also constraints and relationships, possibly with objects of other classes [35]. Each class, and its instances, in this database inherits this information from its parents, comparable to the object-oriented programming paradigm. This entire knowledge base is represented and stored in a purpose-built relational database. To increase performance, the necessary classes are prefetched into memory, *e.g.* in this case, the class *building* and all related classes of *building*. Naturally, all instances of these classes are ultimately associated with a specific geometric model (*e.g.* a large brown, leather sofa). Among other uses, the semantic library has been successfully deployed for handling object

interactions in games using *services* [35], for driving a *semantic layout solving* approach [33], and for supporting the generation of *procedural filters* [36].

The semantic building model integrates therefore a flexible and rich representation of building elements (*e.g.* floors, rooms, windows, walls, chairs), including their attributes (*e.g.* the area of a room), constraints (*e.g.* an outer door should lead to a public room), roles (*e.g.* public and private rooms) or relationships (*e.g.* adjacency between rooms). This semantics, as we will see, is instrumental in the consistency maintenance performed by the moderator, particularly for conflict detection and identification. Semantic elements are also associated with some minimal geometric data, including a position, an orientation and a primitive shape, which is an abstracted representation of the building element's actual 3D geometry (*e.g.* a line, polygon, extruded line, extruded polygon).

Each procedural component, in its generation procedure, can resort to the moderator in a number of ways, which we now describe in detail.

1) *Register a building element*: a procedural component can *register* a new building element with the moderator. This can either *approve* the registration, meaning that the new building element is deemed valid for integration in the building model, or *reject* it, meaning that the new building element causes a conflict that cannot be handled in any other way. In the latter case, the component should retract its conflicting element. For each registered building element, a corresponding semantic element is instantiated and inserted in the semantic building model, possibly with specific values for some of the class attributes; for example, a window instance could have a boolean attribute value indicating whether or not the window glass is tinted.

2) *Register a constraint*: besides new building elements, components can also register new *constraints*, to be satisfied between two building elements. A variety of different constraint types can be devised, enforcing *e.g.* connectivity, proximity, adjacency or non-adjacency between elements. Constraints as these have two operands, indicating the two semantic elements they act upon; or, more precisely, those operands consist of the respective semantic class descriptions, possibly containing some attribute values to narrow down the constraint definition. For example, we can declare that non-tinted windows cannot be adjacent to private rooms with the constraint *non\_adjacency(window{tinted:false}, private room)*. These constraints, together with other constraints available in the semantic library, are used in building inquiries, as discussed next.

3) *Inquire about a building element*: first of all, components can *inquire* the moderator about *registered* building elements. Such inquiries provide components with advice based on up-to-date information on the integrated building model, which they can incorporate in their decision process for creating new building elements. For example, components can inquire about which room is adjacent to this exterior wall, which rooms share this interior wall, what is the function of this room, etc.

Inquiries can also be used to find out whether a *potential* building element could be successfully registered, *i.e.* approved as valid by the moderator. Such an inquiry does not imply registration, or even creation, of elements, and it can be generically defined as follows: can an instance of class *c*, with attribute values  $a_1 \dots a_n$ , with shape *s* be placed at position *p* and orientation *o*? In order to answer such inquiries, the moderator first gathers all constraints mentioning class *c* and, for each of them, evaluates whether they are satisfied for shape *s* at position *p* and orientation

*o*. For example, say we want to inquire whether we can place a non-tinted window of shape  $s$  at position  $p$  with orientation  $o$ , i.e.  $inquire(window\{tinted:false\}, s, p, o)$ . The example constraint defined above references a window class with attribute *tinted* equal to false. Therefore, the moderator checks whether shape  $s$ , with the given position and orientation, is adjacent to the shape of any private room. If so, that *non\_adjacency* constraint is not satisfied, and, therefore, the building advice is negative. This same constraint evaluation mechanism is used to evaluate the previously described inquiries, e.g. to inquire which type of room is adjacent to a particular wall.

The methods to evaluate these constraints were initially built for the semantic layout solving approach [31]. For this purpose, we built methods that, given a scene, an object shape and geometric constraints between the object shape and other shapes inside the scene, can generate all valid positions and rotations for that new object. For a more detailed explanation of how the solver works, how these methods were implemented and why these methods were built instead of using existing geometric constraint solving techniques, we refer the reader to [31]. In the semantic moderator these methods are used to identify whether or not a building element at a given position in the scene is placed according to its related constraints, as is explained above. If the position for the building element conflicts with the related constraints, then a negative building advice is given, which should be handled by the component, e.g. by retracting the element.

These constraints are represented in the following way: source feature (or object) - relationship type - target feature (or object), with a number of parameters (depending on the relationship type). For example: *vase* class - *on* relationship - *top* feature of *cupboard* class, represents that a vase should be placed on the top of a cupboard. The declared constraints in the semantic library can be mapped in a straightforward way to the actual constraints used by the layout solving methods.

Since semantic elements use primitive shapes to represent the shape of building elements in the moderator, the required geometric tests (adjacency, overlap) are relatively simple and have therefore very little impact on the overall performance at the expense of a marginal amount of accuracy. Typically, it is safe to assume that building elements, such as windows, can be reasoned with using a primitive shape instead of a highly detailed mesh including e.g. the ornamentation of a window frame.

4) *Select valid positions for a building element*: finally, a procedural component can approach the moderator with a list of candidate positions for a given building element, requesting it to *select* a given number of valid positions for that building element. This is typically used for specific types of building elements that need to be placed once (or any fixed number of times) in the entire building, such as an external ventilation unit, satellite dish or chimney. Explicitly selecting a valid location to later place the element is a useful advice for procedural components that do not allow backtracking. This function is particularly suited to handle *exclusion* conflicts, explained at the beginning of this section. Validation of each candidate position is handled in the same way as described above: for each of the candidate locations, the moderator will check whether the existing constraints are satisfied, in which case the location is deemed valid. From the valid candidate locations, it selects the requested number of positions at random. These selected positions are marked within the semantic building model.

Using the above moderator functionality, procedural components are indirectly made aware of the results of each other's actions, through communication with the moderator. By registering, inquiring and selecting, components are provided valuable building advice, to which they can timely react and thus prevent the occurrence of *intersection*, *functional* and *exclusion* conflicts.

### B. Wrapping components

As highlighted in Section I, the integration of existing procedural components within the same framework has attractive advantages. The counterpart is, of course, that there is some implementation effort involved. We now describe the implementation steps required and the impact of the integration process on each procedural component.

The main two implementation steps that need to be taken are (i) implementing a wrapper interface for the component, and (ii) modifying its generation procedure to include the proper semantic moderator queries (*i.e.* registering elements and constraints, inquiring about building elements and requesting and inquiring about marked positions).

The main purpose for a component wrapper is to provide access to the functionality of the moderator using a generic interface, as shown in Fig. 1. Such a wrapper only needs to be implemented once for each procedural component, regardless of the number of other components or the type of building being generated.

The secondary purpose of the wrapper is to allow components to be notified, through the moderator, of the results of actions performed by another component. For this, the moderator has a notification mechanism that informs all components of changes in the semantic building model. Through its wrapper, in turn, a component can handle specific notification events, triggering their own actions when another component performs a specific action. For example, a texture generator can create an appropriate wallpaper when an inner wall is registered by a floor plan generation component.

The final purpose of the wrapper is to handle the conversion between a component's specific shape representation (*i.e.* data structure, coordinate system, etc.) and the common shape format used by the moderator. Whenever a new building element is registered, a notification event is provided to all other components. However, not all components will necessarily have to do something with it; *e.g.* a facade grammar component typically does not need to know the positions of all the furniture placed by a layout component. Only the components that require information on that element need to convert it to their internal format. As a result, introducing more components will not necessarily have an exponential impact on the computational efficiency of the building generation.

Of course, a specific wrapper can include more functionality relevant to its procedural component. After communicating with the moderator, a component might need to perform additional actions. Typical examples include: (i) what to do when an element cannot be registered, or (ii) immediately selecting a position and creating a building element after getting a number of marked locations for this element. These additional actions can be implemented within the wrapper methods or directly in the existing procedural technique, if that is preferable.

Finally, it should be mentioned that minor alterations will need to be made directly in the component's procedural generation method. At least, the wrapper methods need to be invoked throughout it at the correct time. An example

is the registration of elements with the moderator before they are definitively placed. Still, the implementation of the wrapper interface is the most important step required for the successful integration of a new procedural component. After a component's wrapper is implemented in the correct way and the mentioned minor alterations to the procedure have been performed, that component becomes and remains integrated within our framework. All its functionality, including notification events, remains intact regardless of changes to, and replacements of, other components.

### C. Plan and Conductor

Our semantic approach described so far enables components to collaborate, through their wrappers, in the generation of consistent buildings. However, the invocation of the various components still needs to be orchestrated in such a way that they constructively work together, *i.e.* following the correct steps in the appropriate order. The order of invocation of components often has an influence on the end result, and designers therefore need to have sufficient control over this.

To support this degree of control, we created the concept of a *conductor* and its *building plan*. Plans are simple documents where one can declare which components should be used, when and how to use them. Designers can create separate plans for different building types using the same integrated components. Primarily, designers use plans to control the sequence in which components are invoked, and also to provide values for the input parameters that each component requires. Varying these is what allows one to define different building types. For example, using different values for the style and lot shape parameters of a façade grammar allows one to create different building façades. Bear in mind that multiple executions of the same plan but with different random seeds, typically result in variations of the same building type, since most procedural techniques are stochastic in nature.

Currently, building plans are specified using a declarative scripting language developed for this framework. Among other things, this language provides commands for declaring the components used in the plan, and invoking them in a desired order. The invocation of a component, declared using the *execute* command with the respective parameters, is supported through a call to its wrapper. An example of the syntax of this language is shown in the excerpt of one of the examples (Villa Neos, discussed in Section IV-D):

---

```
// First, we list assets, which are text files
// in which some parameters
// for components are defined
asset "data/neos_floor1.txt" : floor1Params;
asset "data/neos_floor2.txt" : floor2Params;
asset "data/neos_facade.txt" : facadeParams;

// Now we import the libraries
// for the different components
import "CGAShapeGrammar.dll" : shape;
import "LopesFloorplanGenerator.dll" : interior;
import "TutenelLayoutSolving.dll" : solver;

// We first invoke the CGA shape grammar component
```

```

facade = execute(shape::Component, lot, facadeParams);
// the parameter 'lot' is a predefined variable for
// the building lot on which a plan is executed:

// We query the moderator for the building floors
// in the villa (created in the previous step)
flr1 = moderatorQuery("class: building floor{floor number:1}");
flr2 = moderatorQuery("class: building floor{floor number:2}");

// We invoke the floor plan generator to generate
// a room layout for both floors
floorplan = execute(interior::Component, flr1, floor1Params);
floorplan = execute(interior::Component, flr2, floor2Params);

// Resume the shape grammar for remaining details
// e.g. texture interior walls, etc.
resume(shape::Component, facade);

// Now we invoke layout solving procedures
// to fill the rooms with furniture
layout = execute(solver::Component, "../data/kitchen.proc", moderatorQuery("class: kitchen"));
layout = execute(solver::Component, "../data/bathroom.proc", moderatorQuery("class: bathroom"));
...

```

---

In particular cases, a straightforward one-step sequential invocation of a set of components can be sufficient for generating a consistent building. This is especially the case for situations where the constraints and dependencies between the building elements produced by the different components are fairly loose. An example is generating the façade of a one-floor building after the complete creation of a floor plan. If the only constraint is to avoid intersection conflicts between windows and interior walls, and the invocation of both components follows the standard procedure of registration and inquiries, then their sequential invocation can create a multitude of consistent building variants.

However, such cases are rare. For the vast majority of buildings, stronger dependencies are present and step-based execution of components is needed for consistent results. For example, a façade generator creating a multiple floor building might need to wait for the generation of one floor plan to complete, before resuming with the next floor's façade. Plans can include step-based execution of components if the wrappers are implemented to support it. Note that, although some components can execute in a step-wise fashion, that is unfortunately not enough to support backtracking, i.e. undo or redo a step of a specific component that turned out to yield an unsuitable configuration. The main reason for this is that to support backtracking in our approach, every component should support backtracking as well, and this would be an unreasonable demand since it would exclude many interesting procedural techniques.

Plans are also responsible for another mechanism: sharing and passing building elements from one component to the next, to allow for further detailing by the latter. This is an indirect type of communication: the moderator distributes among components the semantic elements representing the building elements, according to the needs explicitly specified in the plan. A good example of this are building elements produced by a floor plan component:

after registration, floorplan elements could be passed to a shape grammar to detail its geometry or texture. The plan specifies and controls if and how registered elements are passed to which other components. For instance, a plan can specify what the shape symbol of the semantic element (originally created by the floorplan component) should be and, optionally, which semantic attributes are mapped to shape parameters.

As follows from Fig. 1, the conductor is responsible for executing plan steps, or items, in the correct order. The conductor's function is to parse the plan and, for each item, invoke the correct component through its wrapper. The conductor automatically maps commands in plan items, such as *execute* or *resume*, to the corresponding wrapper methods.

Finally, the conductor is also responsible for assembling the resulting 3D geometry generated by each component. For this, the conductor maintains a building model graph, where each node contains the geometry of a building element generated by a component. Currently, components are responsible for supplying this geometry defined in the common coordinate system and scale. After all geometry has been generated, this graph is optimized for interactive rendering.

#### IV. RESULTS

This section aims at illustrating the potential of our integration framework, as well as demonstrating its feasibility by means of three examples of automatically generated consistent buildings. For this, we have selected, implemented and integrated four independent procedural components in our framework. These four components generate façades, floor plans (two different components) and interior furniture layouts. Moreover, the examples of consistent buildings discussed in this section also help make clear that component integration requires only minor modifications to each of the techniques.

To procedurally generate the exterior of our buildings, we selected the CGA shape grammar proposed by Müller *et al.* [13]. This was a natural choice since the CGA shape grammar is a well known and accepted method, and has become somewhat of a standard for procedural building façades. We implemented a shape rewriting system and a subset of the CGA's shape operations, with which we can define production rules to generate both the volumetric shape of buildings and their façade details.

To better evidence the ease of integration of components in our framework, we experimented with two alternative techniques for generating floor plans. The first method we integrated is our own grid-based procedural floor plan generation method [30], which is not based on rewriting nor shape subdivision. This choice helps demonstrate that the integration works for two very distinct components, *i.e.* a (façade shape) grammar and an algorithmic method. The second method we integrated, is a floor plan generator based on squarified treemaps [28]. We chose this second method to demonstrate that the integration of new methods in the framework is relatively easy. It also highlights that, for a specific building element, we can switch to a different generation component simply by changing a couple of lines in the building plan.

The last procedural component we selected is a technique for furniture placement, supporting object layout solving in arbitrary spaces. For this purpose, our own semantics-based layout solver [31] was found very suitable. For details

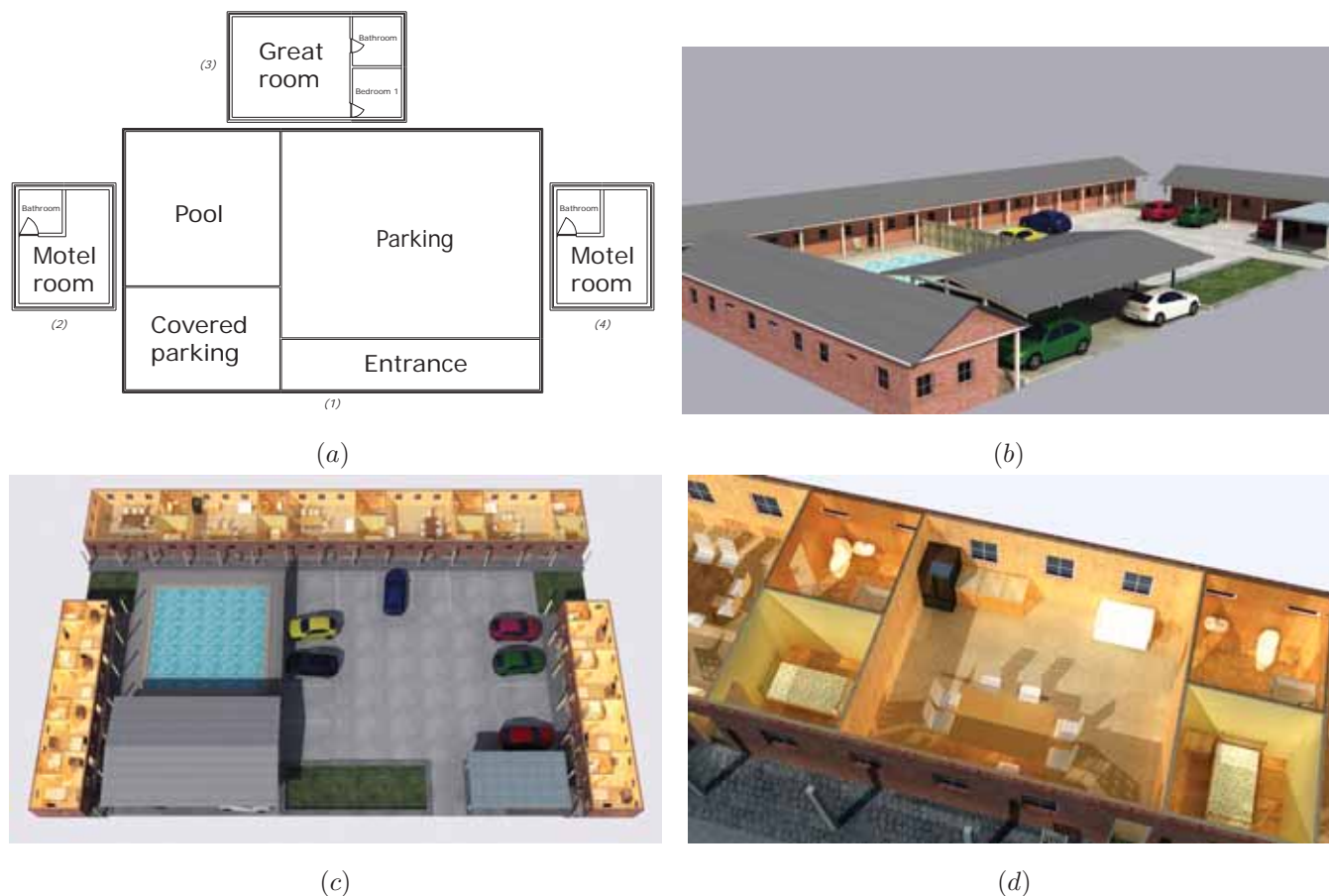


Fig. 2. Generation of a motel complex: (a) layouts generated at three different steps of the plan, by the floor plan component, (1) lot, (2) basic room for left building aisle, (3) suite room for main building aisle, (4) basic room for right building aisle; (b) front view of the motel lot, with three buildings aisles, two parking lots and a swimming pool; (c) top view of different room layouts (basic and suite) and matching building façade regular pattern (d) focus on suite room layout example.

on the functionality and inner working of each of the selected techniques, we refer to the respective publications.

Combining these four components, we have composed building plans for three different building types: a motel complex, an American house, and a Greek holiday villa. We believe these examples are very illustrative of the versatility of our integration approach, and clearly highlight the different aspects of building consistency.

#### A. Wrappers implementation

For each component, we created a specific wrapper to communicate with the semantic moderator. This wrapper provides the necessary calls and notifications for *registration* of building elements and *inquiries* for building advices. Furthermore, it provides conversion of generated results from the component specific model (e.g. 3D geometry, a 2D grid of tiles, etc.) to the common model used in the moderator, ensuring the registered elements scale, orientation and location are coherent. As explained in Section III, wrappers are not required to convert all building elements



Fig. 3. Generation of a North American villa: (a) front view with porch; (b) back view with different types of windows and side-doors depending on adjacent rooms; (c) top view on the different rooms (great room, kitchen, bathroom, bed room, laundry); (d) automatically placed furniture based on room types and object relations; (e)-(f) front view and interior view of the same plan, but now using another floor plan generation technique, that of Marson and Musse [28].

received through the wrapper, but can be more selective and filter for relevant information from other components. All of the above functionality was implemented for the façade, floor plan and interior layout components.

Of course, in each component, the usage of its wrapper had to be implemented as well. For instance, for the floor

plan and interior layout components, registration calls or inquiries were added at specific points in the algorithm. For the shape grammar component, we provided each call as a shape operation (registration) or function (inquiry). They were written within a grammar definition file as part of the normal shape derivation rules. This made the interaction with the moderator easy and more intuitive, *e.g.* within a conditional rewriting rule we can inquire whether deriving the current shape to a window is allowed here, and if not, rewrite it as a plain wall segment instead.

As mentioned in Section III, the building plan can determine not only when but also to what extent each component is executed, allowing for interleaved, step-by-step execution of components. For this, *break* and *continue* calls were added for the wrapper for each component. A break call can have component-specific parameters. For instance, for a shape grammar component, a break point can be placed at a specific shape symbol, halting executing when that symbol is about to be derived.

In the following subsections the three examples will be outlined. For each of them we explain their unique attributes, outline the created building plan and show the generated results.

### B. Example 1: Motel Aloha

As a first example of a building created with several integrated components, we consider a typical motel. The motel lot consists of a number of sections: an entrance and reception building, two parking lots (one of which is covered), a swimming pool and three motel room aisles. The motel rooms come in two variants, a basic room setup with bed and bathroom, and a more spacious suite featuring a living room and kitchen combination.

1) *Building plan*: The plan for this motel uses three procedural components (grid-based floor plan generator [30], CGA shape grammar [13], and the semantic layout solver [31]), and is divided in six steps, executed by the component indicated in brackets:

- 1) Layout motel lot in several sections (floor plan);
- 2) Create exterior geometry of each section (shape grammar);
- 3) Generate and iterate the basic room layout for both the left and right building aisle (floor plan);
- 4) Generate and iterate the suite layout (floor plan);
- 5) Detail building façades (shape grammar);
- 6) Add furniture to each motel room (furniture).

2) *Generation results*: See Fig. 2 for an example of the motel complex, resulting from this plan. Fig. 2 (a) illustrates the intermediate layouts generated by the floor plan component: (1) shows the layout of the motel lot, produced in the first step of the plan. The individual basic and suite room layouts generated for the left, main and right building are shown, respectively, in Fig. 2 (a) (2), (3) and (4). Fig. 2 (b) gives an overview of the motel complex model. In Fig. 2 (c), and in the close-up Fig. 2 (d), we see the layout of the two types of rooms, including the matching furniture models.

In our implementation, which is not optimized for performance, the motel took on average 12 seconds to generate on a consumer PC. Only about 5% of this time was spent on the semantic moderation of procedural components.

The majority of the computation time was thus spent with the procedural generation components, such as the shape grammar component.

3) *Plan execution:* As follows from the building plan above, the floor plan is responsible for generating both the layouts of the sections in the lot and the interior layout for the two types of rooms. These layouts are sequentially generated by the floor plan component and are registered with the semantic moderator. They do not include windows, external doors or other façade elements, since these are created by the shape grammar component, further on in the plan (in step 5).

Layout of the motel lot layout is stochastic, so that the layout and the shape of each of the sections (parking, motel buildings, etc.) slightly differs each time this plan is executed. To generate the lot layout, the plan considers each section to be a "room" with a certain weight and uses the floor plan component to generate a suitable layout, adhering to defined adjacency constraints (*e.g.* the entrance is adjacent to the parking lot). These sections, after registration with the moderator, are introduced to the shape grammar component as polygonal shapes, with the name of their semantic class mapped to a shape grammar symbol. As a first pass, the shape grammar rewrites each section, except for the motel buildings, to their final detailed geometric models. For the motel buildings, only the volumetric shapes are derived and registered, after which the shape grammar component halts its execution (step 2).

The volumetric shapes of the motel buildings are passed to the floor plan component. Unlike typical houses, the floor plan for the motel is generated in a repetitive mode. In this mode, one layout is generated and repeated over all the separate motel rooms. Both the basic and the suite variants are generated in this way. We included this repetitive layout here to show that is possible to generate an uniform structure using the plan, which could be desirable for a motel or office space. Of course, this uniform layout is not applicable to all scenarios; it is always possible to layout each room individually to obtain more variation.

The floor plan component registers the rooms it generated with the semantic moderator, including attributes like the room function (bathroom, bedroom, etc.). The interior walls are registered per continuous segment and introduced to the shape tree of the halted shape grammar component.

The shape grammar component now resumes, deriving shapes for the interior walls and creating the roofs and the façade details, such as windows and doors. The façade is constructed using a window pattern that is applied to the entire span of a building outer wall. This repetitive pattern is specified in the shape grammar to achieve the uniform façade patterns typically found in motels. However, inquiries are used to determine whether a window is allowed at a certain position and whether it should be a normal or small bathroom window.

In the last step, the furniture component is called to populate each individual room, according to function. Since room registration (by the floor plan component) included their function, this can be queried by any other component, at any time. In the furniture component, a semantic description states what kind of objects should be present in these rooms, and the layout solver places these objects based on their defined relationships and constraints. The resulting furniture layouts are similar for equal room types although still unique for each.



Fig. 4. Example of a Greek holiday two-storey luxurious villa: (a) front view with veranda and pool; (b) back view with different types of windows depending on adjacent rooms; (c) second floor with balconies, terrace and staircase; (d) first floor with several rooms

### C. Example 2: Meadowdale house

The following example is a typical North American one-storey house with a front porch. This type of building has a more complex floor plan than a motel suite and, accordingly, the façade should be generated differently.

1) *Building plan*: The building plan of this example is quite straightforward, consisting of these four consecutive steps (again executed by the components in brackets):

- 1) Create coarse volumetric building shape (shape grammar);
- 2) Layout the house's rooms (floor plan);
- 3) Detail the complete building (shape grammar);
- 4) Add furniture to each room (furniture).

2) *Generation results*: Fig. 3 presents two example results generated for the above Meadowdale building plan. In the first example, Fig. 3 (a), we see that the front porch is placed at the front wall segment of the great room, and an additional door is placed on a side wall segment of the kitchen. Window types and patterns match with the function of the adjacent rooms, as can be seen in Fig. 3 (b): small windows are placed in the bathroom wall

segment; and no windows, but a door and air vent, in the laundry segments. Fig. 3 (d) shows that the automatically placed furniture matches well with the function of the rooms.

In the second example, the floor plan is generated by the technique of Marson and Musse [28]. Fig. 3 (f) show an exterior and interior of the same building. The most noticeable difference between the floor plans, by comparing Fig. 3 (d) and (f) is the absence of L-shaped rooms and the presence of a corridor. Since this technique uses squarified treemaps, it is unable to produce non-rectangular rooms. To include a corridor, we modified the input parameters for the floor plan component to add two bedrooms instead of one. This resulted in the creation of a corridor to link the bathroom and the two bedrooms to the living room.

This second example shows some of the possible variation in outputs of a single plan, including variation in the façade component (*e.g.* textures, front porch at a different location) and in furniture placement. Of course, the same rules for windows types, and the position of doors and the air vent, apply in the second example as well.

Meadowdale took on average 7 seconds to generate. Most of the computation time was spent in the shape grammar and layout solving components, each about 40 % of the total computation time. The grid-based floor plan component took 9% of the total time to generate the fairly straightforward floor plan of Meadowdale. Less than 1% was spent on the semantic moderation of procedural components.

3) *Plan execution:* In the first step of the plan, the shape grammar component determines the building footprint inside the garden and extrudes and registers its volumetric shape. As no further rule matches are found, the component halts and the plan proceeds with the floor plan generation. The house has a great room, a kitchen and laundry room, a bathroom, and either one bedroom (in the first example, Fig. 3 (a)-(d)) or two bedrooms (in the second example, Fig. 3 (e)-(f)). In the second example, a corridor is automatically added by the floor plan generation technique (see [28]).

Several adjacency constraints are defined (*e.g.* between the bedroom and the bathroom and the kitchen and the great room). A special type of adjacency constraint requires the great room to be at the front of the building. Typically, In this type of houses, the front porch is directly connected to the great room: on the left side of the building in the first example, and on the right side of the building in the second example.

For both the interior and exterior walls of this building, continuous wall segments are registered and passed to the shape grammar as separate shapes. Unlike the motel example, wall segments are used instead of complete side walls. For buildings as motels and offices, creating an uniform façade pattern is more important. For residential buildings, the façade is more reflective of the interior layout and room function. Using wall segments ensures that each wall shape belongs to only one room, making it easier to generate façade segments that match with the rooms.

Again, within each room, appropriate furniture is automatically placed. For instance, in the kitchen, bottom and top cabinets, a stove and refrigerator are placed against the wall, while a dining table surrounded by chairs are placed in the centre of the room.

#### D. Example 3: Villa Neos

Our last example features a modern and luxurious Greek holiday villa. This villa has two floors, the second smaller than the first one because of a large open balcony. Inside, an interior staircase connects both floors.

1) *Building plan and components:* The corresponding plan is similar to the previous example of the North-American villa, with the exception that this building features two storeys.

- 1) Create coarse volumetric building shape (shape grammar);
- 2) Layout the villa's first floor (floor plan);
- 3) Layout the villa's second floor (floor plan);
- 4) Detail the complete building (shape grammar);
- 5) Place furniture in each room (furniture).

2) *Generation results:* One of the results generated by this plan is shown in Fig. 4. Fig. 4 (a) and (b) shows the veranda and second floor balconies from different angles. Note the staircase connecting both floors in Fig. 4 (c) and (d).

Villa Neos took on average 9 seconds to generate, of which less than 1% was spent on the semantic moderation between components.

3) *Plan execution:* More than just showcasing the possibilities of our approach to create different building types, the interesting aspect of this example is the way the staircase is integrated across two different components. The staircase shaft is determined by the shape grammar during the creation of the coarse building shape (step 1). It is registered to the moderator as a semantic object of class *staircase*. In steps 2 and 3 of the plan, using an inquiry, the staircase is obtained and passed to first and second floor plans as a room that is treated as *fixed* during the layout process (see [30] for details). In this way, we ensure that the staircase placement is congruent between two floors.

## V. DISCUSSION

The examples presented in the previous section show the potential of integrating existing procedural techniques as a method for generating consistent buildings. These examples highlight the central role of the semantic moderator within our framework, coordinating and advising components towards the goal of generating consistent buildings only. By correctly using the generic interface of the moderator, procedural components can obtain advice on the impact on building consistency of each of the elements they propose to include. For this, all building elements generated by different components are combined in the central semantic building model, to ascertain that their location and semantics do not conflict with each other. An example of spatial consistency, taken from the results in Section IV, is that the semantic moderator assures that the walls created by a floor plan generator do not intersect the windows created by a façade generator. Another example, but now of functional consistency, is that both these same windows and the furniture (laid out by a third procedural component) are all generated according to the function of the room.

The examples in the previous section also show that, when properly integrated, the individual procedural components do not significantly divert from their standard behavior. Components still execute their individual procedures, while

communicating results with the moderator helps them to prevent the building model from reaching an irreversible invalid state, where required building elements are misplaced or excluded. For instance, a façade generator can use the *selection and marking* mechanism to prevent the exclusion of an (initially misplaced) front door.

In the previous sections, we outlined the implementation extensions and alterations required for each component to integrate with our framework. We consider these to have a relative low burden on developers. Wrapping components and writing a building plan, as done for our examples, are reasonably straightforward implementation tasks. To give an indication of the amount of effort for integrating a new component, the components featured in Section IV typically took a single developer less than one working day to integrate. The shape grammar component required slightly more effort, as the calls to the semantic moderator had to be made available in the CGA grammar as new shape operations, but still, it was fully integrated within two days.

Building plans can be written in countless different ways, *e.g.* by giving priority to façade patterns (Aloha motel example) or room layout (Meadowdale example). Allowing such flexibility in the plan creation process enables designers to benefit from the conflict-solving advantages of our framework, while giving them the freedom to configure their plan in the most adequate order for each building type. For example, an office building plan could require rooms to be generated after the building facade is finished, thereby ensuring a regular exterior pattern, whereas for a residential villa one could rather create a façade after the floor plan is completely determined. Our semantic integration approach conveniently supports both ways.

It should be remarked that the implementation effort to integrate multiple components is naturally dependent on the coherence of the global choice of components. Classifying this effort as straightforward reasonably assumes that the individual procedural techniques were chosen to minimize conflicting situations. The framework by itself cannot totally assure that implementation work will always be kept to a minimum. In other words, if two procedural techniques do not naturally fit well together, you can hardly make them fit any better regardless of the amount of integration work put in it. Consider the example of a floor plan generation technique which creates rooms individually and assembles them to form a new building shape. If this unknown building shape needs to fit inside the building lot shape, which could have been generated by another component, many modifications might be necessary to assure that the results of those two components fit. Another complicating factor might stem from differences in capabilities of components, for instance when integrating a furniture generator that only supports placement in rectangular rooms with a floor plan technique that produces arbitrary room shapes.

Regarding performance, our results show (i) that it strongly depends on each component, and (ii) that it is hardly affected by the moderator checks, conversions and operations. In the examples shown, this overhead lies between 1% and 2% of the total running time (less than 100 ms, in absolute time). This overhead in computation time for the semantic moderator functionality can therefore be considered as perfectly acceptable. Of course, if we were to use very optimized procedural components, the overhead would be relatively larger, but still minor when compared to the computational cost of most individual procedural methods.

In some specific instances, a plan or component input specification could lead to a building that can not be completed in a valid way, according to the semantic moderator. This entails that an executing component can not

fulfil its current task and report this issue back to the conductor, which presents the user an error message with a description where the building plan failed. This situation can only be resolved by fixing either the building plan or the problematic component input specification, e.g. its shape grammar, room layout constraints or the building lot shape and dimensions.

Currently, the major limitation in this approach relates to our somewhat naive implementation of the mechanism for combining elements' geometry. As stated in Section III, individual components are responsible for assuring that their generated geometry is converted to the common coordinate system and unit of scale. Although its impact on performance is minimal, this does require an additional implementation effort for each component that is to be integrated. A better alternative would be to automate these steps within the framework itself. In this line, a geometry moderator able to automatically and consistently transform building elements would be a valuable contribution to further smoothen the integration process.

## VI. CONCLUSIONS AND FUTURE WORK

In this article, we proposed a novel approach for automatically generating consistent virtual buildings, *i.e.* buildings consisting of a variety of plausible architectonic elements, all in harmony with each other. Among other uses, such '*enter-anywhere*' buildings are especially suitable for open game worlds and exploration-based gameplay.

This approach provides a semantic framework for integrating different components that implement existing procedural techniques, each of them generating specific building elements. Examples of these components are procedural generators for façades, floor plans, lot shapes, furniture or textures. In our approach, a semantic moderator communicates with these procedural components, and provides them with valuable guidance in order to prevent conflicts among the generated building elements. In this way, we are able to preserve the individual qualities of the integrated components. The moderator keeps a semantic building model that represents each building element generated by the procedural components. Based on this model and on a number of constraints, it maintains the consistency of generated buildings.

We showed the applicability of our approach with examples from our prototype system, featuring the integration of a façade shape grammar, two different floor plan layout generation techniques, and furniture placement techniques. This integration required small modifications, which were straightforward to implement, and did not affect the performance of the procedural components.

This integration approach has valuable advantages over dedicated approaches. These include the ease of integrating new components and to put them into existing plans. This makes it possible to use the best technique for each building element, for each specific building type. Examples of building elements for which we could integrate such dedicated techniques are underground structures and layouts of gardens. Also, we argue that this approach brings both power and flexibility to the building generation process. Plans for generating different types of buildings can easily be elaborated, once the required procedural components have been integrated in the framework. Subsequently, the framework is able to execute them, invoking the available components in any desired combination. Furthermore, this approach allows one to focus on improving individual components, without being concerned with how these

internal changes affect the consistency of the final outcome.

For future improvements, we envision some options for extending the generic interface of our semantic moderator. An example of such an extension is the management of architectural styles between components, *i.e.* between interiors and exteriors (*e.g.* matching colors, patterns). Currently, these styles are part of each procedural component. The designer who is overseeing the integration process is responsible for selecting components with compatible styles. A clear improvement over this would be to introduce a style moderation mechanism that is aware of different architectural styles, including how they can be applied to the different procedural components. Separating style and structure generation would necessarily require more communication and new constraints in our framework. A new mechanism, either a new moderator or an extension to the current one, would need functionality to coordinate: (i) building structure-only generation, (ii) creation of style appropriate to fit all the structure (*i.e.* all individual components), and (iii) application of style to all the structure, in a consistent fashion. Such a clear separation of style and structure would definitely be a valuable contribution to procedural generation of buildings.

It would also be interesting to investigate whether this approach can be applied to other areas of procedural generation of virtual worlds. A first example could be the generation of an urban environment. In this setting, the semantic moderator could be used to avoid typical conflicts occurring between a new building and the urban environment. For instance, using an extended version of the semantic moderator, one could avoid generating windows that look out directly on a wall of a neighboring building. Another example is the placement of light posts on the pavement, where one would want to avoid blocking building doorways and ground floor windows, as far as possible. The semantic moderator would need new functionality, similar to the current one, to share new types of information. The semantic library [33] used by the moderator already conveys attributes for these concepts and properties, therefore this type of extensions are within reach. However, for our framework to become a more generic integration platform, other challenges would need to be addressed, *e.g.* supporting automatic geometry combination, consistency checking for additional aspects like playability and more detailed planning methods.

We are especially interested in using the proposed integration approach in other contexts of virtual worlds. We aim to include our approach in SketchaWorld [37], a virtual world modelling framework that uses a declarative approach to procedural generation of virtual worlds. We also plan to populate buildings with objects enriched with semantic services [35]. These services allow for a more complete and non-scripted way for players to interact with virtual objects. Applied to these new contexts, not only buildings, but also procedurally generated virtual cities would encourage players to explore rich open worlds. In these new worlds, players could interact with their environment with less limitations, in a more natural and meaningful way.

In short, our semantic approach allows one to integrate existing procedural techniques, while preserving their individual qualities, thus allowing for the automatic generation of very detailed and consistent buildings.

#### ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments. We thank Fernando Marson for kindly providing the source code of his floor plan procedural generator and for assisting us in its integration process. Finally, we

thank Matthijs Schaap for his assistance in rendering most examples.

## REFERENCES

- [1] R. M. Smelik, K. J. de Kraker, T. Tutenel, R. Bidarra, and S. A. Groenewegen, "A Survey of Procedural Methods for Terrain Modelling," in *Proceedings of the CASA 2009 Workshop on 3D Advanced Media in Gaming and Simulation (3AMIGAS)*, Amsterdam, The Netherlands, June 2009.
- [2] P. Merrell, E. Schkufza, and V. Koltun, "Computer-Generated Residential Building Layouts," *ACM Transactions on Graphics*, vol. 29, no. 5, 2010.
- [3] R. Bidarra and W. Bronsvort, "Semantic Feature Modelling," *Computer-Aided Design*, vol. 32, no. 3, pp. 201–225, 2000.
- [4] T. Tutenel, R. Bidarra, R. M. Smelik, and K. J. de Kraker, "The Role of Semantics in Games and Simulations," *ACM Computers in Entertainment*, vol. 6, pp. 1–35, 2008.
- [5] F. K. Musgrave, C. E. Kolb, and R. S. Mace, "The Synthesis and Rendering of Eroded Fractal Terrains," in *SIGGRAPH '89: Proceedings of the 16<sup>th</sup> Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA: ACM, 1989, pp. 41–50.
- [6] D. S. Ebert, S. Worley, F. K. Musgrave, D. Peachey, and K. Perlin, *Texturing & Modeling, a Procedural Approach*, 3rd ed. Elsevier, 2003.
- [7] Y. I. H. Parish and P. Müller, "Procedural Modeling of Cities," in *SIGGRAPH '01: Proceedings of the 28<sup>th</sup> Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA: ACM, 2001, pp. 301–308.
- [8] G. Kelly and H. McCabe, "Citygen: An Interactive System for Procedural City Generation," in *Proceedings of GDTW 2007: The Fifth Annual International Conference in Computer Game Design and Technology*, Liverpool, UK, November 2007, pp. 8–16.
- [9] B. Watson, P. Müller, O. Veryovka, A. Fuller, P. Wonka, and C. Sexton, "Procedural Urban Modeling in Practice," *IEEE Computer Graphics and Applications*, vol. 28, no. 3, pp. 18–26, 2008.
- [10] B. Weber, P. Müller, P. Wonka, and M. Gross, "Interactive Geometric Simulation of 4D Cities," *Computer Graphics Forum: Proceedings of Eurographics 2009*, vol. 28, pp. 481–492, April 2009.
- [11] P. Wonka, M. Wimmer, F. Sillion, and W. Ribarsky, "Instant Architecture," in *SIGGRAPH '03: Proceedings of the 30<sup>th</sup> Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA: ACM, 2003, pp. 669–677.
- [12] A. F. Coelho, A. A. de Sousa, and F. N. Ferreira, "Modelling Urban Scenes for LBMS," in *Web3D '05: Proceedings of the 10<sup>th</sup> International Conference on 3D Web Technology*. New York, NY, USA: ACM, 2005, pp. 37–46.
- [13] P. Müller, P. Wonka, S. Haegler, A. Ulmer, and L. V. Gool, "Procedural Modeling of Buildings," in *SIGGRAPH '06: Proceedings of the 33<sup>rd</sup> Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA: ACM, 2006, pp. 614–623.
- [14] H. Koning and J. Eizenberg, "The Language of the Prairie: Frank Lloyd Wright's Prairie Houses," *Environment and Planning B: Planning and Design*, vol. 8, no. 3, pp. 295–323, 1981.
- [15] G. Cagdas, "A Shape Grammar Model for Designing Row-houses," *Design Studies*, vol. 17, no. 1, pp. 35 – 51, 1996.
- [16] D. Y. Kwon, "ArchiDNA: A Generative System for Shape Configuraton," Master's thesis, University of Washington, 2003.
- [17] L. Yong, X. Congfu, P. Zhigeng, and P. Yunhe, "Semantic Modeling Project: Building Vernacular House of Southeast China," in *VRCAI '04: Proceedings of the 2004 ACM SIGGRAPH International Conference on Virtual Reality Continuum and its Applications in Industry*. New York, NY, USA: ACM, 2004, pp. 412–418.
- [18] Procedural, inc., "CityEngine," Available from <http://www.procedural.com>.
- [19] Epic Games, "Unreal Engine 3," Available from <http://www.unrealtechnology.com>.
- [20] James Golding - Epic Games, "Building Blocks Artist Driven Procedural Buildings - Game Developers Conference 2010," Available from <http://gdcvault.com/play/1012655/Building-Blocks-Artist-Driven-Procedural>, 2010.
- [21] P. Müller, G. Zeng, P. Wonka, and L. V. Gool, "Image-based Procedural Modeling of Facades," in *SIGGRAPH '07: Proceedings of the 34<sup>th</sup> Annual Conference on Computer Graphics and Interactive Techniques*, vol. 26, no. 3. New York, NY, USA: ACM, 2007.
- [22] X. Chen, S. B. Kang, Y.-Q. Xu, J. Dorsey, and H.-Y. Shum, "Sketching Reality: Realistic Interpretation of Architectural Designs," *ACM Transactions on Graphics*, vol. 27, pp. 11:1–11:15, May 2008.
- [23] S. Greuter, J. Parker, N. Stewart, and G. Leach, "Real-time Procedural Generation of 'Pseudo Infinite' Cities," in *GRAPHITE '03: Proceedings of the 1<sup>st</sup> International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia*. New York, NY, USA: ACM, 2003, pp. 87–94.
- [24] D. Finkenzyler, "Detailed Building Façades," *IEEE Computer Graphics and Applications*, vol. 28, no. 3, pp. 58–66, 2008.

- [25] D. Finkensteller and J. Bender, "Semantic Representation of Complex Building Structures," in *Computer Graphics and Visualization (CGV 2008) - IADIS Multi Conference on Computer Science and Information Systems*, Amsterdam, The Netherlands, July 2008.
- [26] A. Rau-Chaplin, B. Mackay-Lyons, and P. Spierenburg, "The LaHave House Project: Towards an Automated Architectural Design Service," in *Proceedings of the International Conference on Computer-Aided Design (CADEX)*, Hagenberg, Austria, September 1996.
- [27] E. Hahn, P. Bose, and A. Whitehead, "Persistent Realtime Building Interior Generation," in *Sandbox 2006: Proceedings of the ACM SIGGRAPH Symposium on Videogames*. New York, NY, USA: ACM, 2006, pp. 179–186.
- [28] F. Marson and S. R. Musse, "Automatic Generation of Floor Plans Based on Squarified Treemaps Algorithm," *IJCGT International Journal on Computers Games Technology*, vol. 2010, pp. 1–10, January 2010.
- [29] J. Martin, "Procedural House Generation: a Method for Dynamically Generating Floor Plans," Research Poster presented at I3D '06: SIGGRAPH Symposium on Interactive 3D Graphics and Games, 2006.
- [30] R. Lopes, T. Tutenel, R. M. Smelik, K. J. de Kraker, and R. Bidarra, "A Constrained Growth Method for Procedural Floor Plan Generation," in *Proceedings of GAME-ON 2010, the 11th International Conference on Intelligent Games and Simulation*. EUROSIS, 2010.
- [31] T. Tutenel, R. Bidarra, R. M. Smelik, and K. J. de Kraker, "Rule-based Layout Solving and its Application to Procedural Interior Generation," in *Proceedings of the CASA 2009 Workshop on 3D Advanced Media in Gaming and Simulation (3AMIGAS)*, Amsterdam, The Netherlands, June 2009, pp. 15–24.
- [32] P. Charman, "Solving Space Planning Problems Using Constraint Technology," in *NATO ASI Constraint Programming: Students' Presentations, TR CS 57/93, Institute of Cybernetics, Estonian Academy of Sciences, Tallinn, Estonia*, 1993, pp. 80–96.
- [33] T. Tutenel, R. Bidarra, R. M. Smelik, and K. J. de Kraker, "Using Semantics to Improve the Design of Game Worlds," in *Proceedings of AIIDE 2009 - 5th Conference on Artificial Intelligence and Interactive Digital Entertainment*, Stanford, CA, USA, October 2009.
- [34] G. A. Miller, "WordNet: A Lexical Database for English," *Communications of the ACM*, vol. 38, pp. 39–41, 1995.
- [35] J. Kessing, T. Tutenel, and R. Bidarra, "Services in Game Worlds: a Semantic Approach to Improve Object Interaction," in *Proceedings of the International Conference on Entertainment Computing*, 2009, pp. 276–281.
- [36] T. Tutenel, B. Bollen, R. van der Linden, M. Kraus, and R. Bidarra, "Procedural Filters for Customization of Virtual Worlds," in *PCGames '11: Proceedings of the 2011 Workshop on Procedural Content Generation in Games*. New York, NY, USA: ACM, 2011.
- [37] R. M. Smelik, T. Tutenel, K. J. de Kraker, and R. Bidarra, "A Declarative Approach to Procedural Modeling of Virtual Worlds," *Computers & Graphics*, vol. 35, no. 2, pp. 352–363, April 2011.



**Tim Tutenel** graduated in computer science at Hasselt University, Hasselt, Belgium in 2006. He is a Ph.D. student at Delft University of Technology, Delft, The Netherlands on the subject of Semantics in games.

He is currently working in a research project on automatic creation of virtual worlds. His research focus is on layout solving, object semantics, and object interactions.



**Ruben Smelik** graduated in computer science at the University of Twente, the Netherlands in 2006. He is a scientist, and a PhD student at the TNO research institute.

He is currently working in a research project on automatic creation of virtual worlds. His research focus is on methods and techniques for creating geo-typical virtual worlds for serious games and simulations.



**Ricardo Lopes** received the B.Sc. and M.Sc. degrees in information systems and computer engineering from the Technical University of Lisbon, Lisbon, Portugal, in 2007 and 2009, respectively. His Ph.D. research subject is the “Generation of adaptive game worlds.”

His current research interests include adaptivity in games, player modelling, interpretation mechanisms for in-game data, and (online) procedural generation techniques.



**Klaas Jan de Kraker** graduated (1993) at and received his Ph.D. (1998) in computer science from Delft University of Technology, Delft, The Netherlands.

He is a member of the scientific staff at the TNO research institute, where he is leading various simulation projects in the areas of simulation based performance assessment, collective mission simulation, multifunctional simulation and serious gaming.



**Rafael Bidarra** graduated in 1987 in electronics engineering at the University of Coimbra, Portugal, and received his Ph.D. in computer science from Delft University of Technology, Delft, The Netherlands, in 1999.

He is currently an Associate Professor of Game Technology at the Faculty of Electrical Engineering, Mathematics and Computer Science of Delft University of Technology. He leads the research line on game technology at the Computer Graphics Group. His current research interests include: procedural and semantic modeling techniques for the specification and generation of both virtual worlds and game play; semantics of navigation; serious gaming; semantics of navigation; game adaptivity and interpretation mechanisms for in-game data. He has published many papers in international journals,

books and conference proceedings. He integrates the editorial board of several journals, and has served in many conference program committees.