Procedural filters for customization of virtual worlds

Tim Tutenel
Delft University of Technology
Mekelweg 4
2628 CD, Delft, The Netherlands
t.tutenel@tudelft.nl

Roland van der Linden
Delft University of Technology
Mekelweg 4
2628 CD, Delft, The Netherlands
roland.vanderlinden@hotmail.com

Marnix Kraus
Delft University of Technology
Mekelweg 4
2628 CD, Delft, The Netherlands
marnixkraus@hotmail.com

Bart Bollen
Delft University of Technology
Mekelweg 4
2628 CD, Delft, The Netherlands
bartbollen@bartbollen.com

Rafael Bidarra
Delft University of Technology
Mekelweg 4
2628 CD, Delft, The Netherlands
r.bidarra@tudelft.nl

ABSTRACT

Designing virtual game worlds is often a long and labor-intensive process. Moreover, when a game world needs to be slightly altered in appearance, the entire process needs to be repeated, or will at least require some repetitious tasks. Ideally, when the same game world is needed under different circumstances (e.g. in another season, before and after a war, in prosperous or poor economic conditions), the designer should be aided in this process using procedural generation techniques.

We propose an approach for the specification of procedural filters that describe how (parts of) virtual worlds should be customized to fit a particular situation based on their semantics and the conditions of the situation. This description will guide the customization process by triggering and parametrizing, among others, procedural instructions that can change the appearance of the virtual world. We will discuss how the generic nature of this approach, which favors reusability, and its integration with semantics, which increases the intuitiveness of the design process, can eliminate many of the repetitious tasks involved in performing these actions manually.

We describe an implementation of this approach that shows how some simple procedural filters can i) age an urban environment and simulate the effects of poor living conditions on the look of that environment, and ii) apply a party atmosphere to an ordinary office scene.

Categories and Subject Descriptors

I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism; I.3.6 [**Computer Graphics**]: Methodology and Techniques—*Interaction techniques*; I.6.7 [**Simulation and Modeling**]: Types of Simulation—*Gaming*

Keywords

procedural filters; procedural content generation; virtual worlds; semantics

PCGames 2011, June 28, Bordeaux, France

Figure 1: The painting *Girl with a pearl earring*, by Delft painter Johannes Vermeer; from left to right: original photo, with an emboss filter applied, and with a patchwork filter applied.

1. INTRODUCTION

With the advent of digital photography editing software, e.g. Adobe Photoshop or Google Picasa, the notion of filters has become widely popular. In that context, filters affect an input image in a large variety of predefined, often parameterized, ways, which mostly have an appealing and intuitive semantics. In this way, for instance, one can apply on a photograph, among others, artistic filters (as e.g. fresco or watercolor), stylize filters (as e.g. diffuse or emboss), or texture filters (as e.g. grain or patchwork), as shown in the examples of Figure 1.

Inspired by that concept, we introduce in this paper the notion of *procedural filters* for virtual worlds, which we define as a procedure aimed at being applied on (part of) a virtual world, that modifies the appearance and other contingent visual attributes of its objects, in order to give them a peculiar desired twist.

Regardless of whether they are created fully manually or

using procedural generation techniques, most game worlds are static, even if they contain dynamic gameplay elements, e.g. animations or scripted agents. However, the same game world might be needed in different circumstances or 'flavors'; e.g. the first act of a game might play in a city during an economic boom, while the second act might require the same city during a depression. If the world is generated procedurally, one might need to change the techniques used in order to produce the same world under these different circumstances. The situation is even worse when the game world was completely created manually, in which case designers will need to manually revamp the entire world all over again matching the new circumstances. The notion of procedural filters, introduced above, can be used to soften the load of this process. In particular, it enables designers to create and fine tune their own procedural filters, and apply them on (parts of) the game world during the design phase.

The semantics-based approach proposed in this paper provides a generic specification scheme in which a set of instructions can be tied together to describe this customization process of virtual worlds. Our approach is based on semantics, which we define, in this context of game worlds, as all information about the world and its objects, beyond their geometry. This includes object properties, high-level attributes and functional information, as well as interrelationships among different objects. The role of semantics in the design phase of game worlds was previously discussed in more detail in [18]. By allowing designers to use this object semantics in their specification, procedural filters can become both more generic and more intuitive: *generic*, because they are able to map high-level semantic attributes of objects to more low level object parameters; *intuitive*, because such attributes are more accessible than the technical, often cryptic, parameters typical of procedural generation techniques.

In a procedural filter, a designer describes how a scene should be changed to match a particular situation. For example, we can create a filter to turn a landscape into a winter landscape, to provide an ordinary office with a party atmosphere, or to turn a street into a gang-led war zone. Each filter describes a procedure that is to be followed to alter the scene. This procedure can be built up using a wide range of possible instructions, which in turn can be given parameters based on the semantics of the world and its objects.

Finally, as we will see, developed filters can be used as components for building more complex filters: for example, a filter to make a building look neglected can be a combination of simpler filters that add cracks to windows, put graffiti on the walls, and spread some random trash in the front yard.

In the next section we discuss existing related work, giving some examples of other research results towards procedural ornamentation or, in general, customization of game worlds. We also give an overview of other tools and applications that are comparable to the proof of concept editor. We then give a detailed overview of the concept of procedural filters, followed by a description of our implementation of both this concept and a visual editor to create and modify procedural filters. Finally we show some examples of filters that we implemented, and draw some conclusions.

2. RELATED WORK

Most procedural generation techniques are targeted at generating new geometry to fill the virtual world. There are procedural techniques to generate terrain heightmaps (e.g. [12]), plants and trees (e.g. [3]), urban environments with roads (e.g. [6]) or buildings (e.g. [14, 11, 5]). Usually the output of these techniques can be altered by tweaking some parameters or by using different assets (different textures or models). It is, however, much more difficult, and in some cases, impossible, to use such procedural techniques to slightly modify parts of an already existing world, nor should that ever be the intention of procedural content generation techniques.

In [21], Whitehead surveyed the research sub domain of procedural decorative ornamentation. Parts of that article come close to the idea of procedural filters: applying different decorative styles to existing scenes. The author gives an overview of techniques and research examples that could be used to perform these stylistic changes.

To perform these changes, we can use techniques from many procedural fields, as instructions in procedural filters. Often driven by fractal techniques, many textures can be procedurally generated (e.g. [4, 15, 16]). Textures can be generated to match particular materials like wood, bump mapped textures for stucco materials etc. Noise textures can play an important role in applying changes to materials, for example to add moss to wooden objects or rust to metal objects. Other, more complex techniques, have been developed to model paint cracks [13] or to apply weathering and aging effects on textures [8].

Commercial products are available that use procedures to combine different texturemaps and 2D imaging filters to generate new textures. Some examples of such software are Genetica [17] and MaPZone [2]. These can either be generated in the software package itself and exported to a static file or generated on the fly. In the second case, it is possible to parameterize the textures based on the circumstances of the game world. Such a process can be an important and powerful component of procedural filters.

The procedural filter approach, proposed in this paper, expects a virtual world as input. However, we expect this world to be semantically annotated in some way: e.g. we want to know what kind of object a particular model represents. It is obvious that this takes a little extra effort at design time. Fortunately a lot of research has been performed and is still being performed to automate this process, e.g. by using automatic shape recognition algorithms. An example of such a method to automatically specify semantics for large sets of 3D models can be found in [22]. Moreover, once semantics has been specified for a game world, it can be used in many other ways besides procedural filtering. As said before, in [18], we gave an indepth overview of the role semantics can play in both the design phase and the runtime phase of games and simulations.

3. PROCEDURAL FILTER APPROACH

As stated above, the notion of procedural filters was conceived in analogy with filters used in 2D imaging software, which in turn have their roots in the optical filters for photo cameras. Attaching a new filter to your camera lens will not change the content of a photograph, but it will create a different atmosphere or a different effect for it. Procedural filters should work in a similar way: the actual content of the scene should not be changed in a significant way, it will just be "rendered" in different conditions.

A procedural filter consists of a set of connected instruc-

tions that form a certain procedure. A filter is represented by a graph, in which the instructions of the filter are represented as the graph nodes and the dataflow from the output of one instruction, to the input of the next, is represented by directed edges. We distinguish the following categories in these instructions:

1. basic operations (e.g. mathematical operations, conditional statements, creation of primitives...);
2. object transformations;
3. material alterations;
4. changing or loading assets;
5. semantic queries; and
6. automatic content generation.

We will now describe each category in more detail:

- **Basic operations** are obviously necessary to perform even the most simple procedure. Some mathematical operations or conditional statements, loops and the likes are vital. These operations also include the creation of primitives, which are constant values like booleans, real numbers or strings, that can be used as input for other instructions.
- **Object transformations** are translations, rotations or scalations of objects already present in the world. These transformations can be used to *mess up* a scene by, for example, adding some slight random transformations to objects on a previously neat, tidy office. It is of course important to use these transformations with the initial goals of procedural filters in mind: not to change the world in a significant, structural way.
- **Material alterations** are changes to the actual materials (colors, texture maps and shaders) used to render an object. Next to changing the color or texture, we could add or remove shaders, alter shader parameters to match the semantic characteristics of the object or the scene or simply change the entire material.
- **Changing or loading assets** could be loading new textures to be used in the previously described instructions, or switching model assets. This could, for example, be used to change a model with a broken up version of the same object, or switching fully-leaved trees with empty trees, to simulate an autumn setting.
- **Semantic queries** are probably the most important and powerful instructions with regards to the reusability and generic nature of procedural filters. Querying the semantics of a scene involves mostly selecting objects of a particular class from the scene and querying their attributes. When such instructions are available, the filters will be much less ad-hoc and therefore much more reusable. Adding semantics to an object, means in the first place, linking it to a generic class, often taken from an ontology. Instead of selecting models based on e.g. filenames, the linked class is a more generic way of performing selections. Moreover, the attributes of an object (e.g. their age or the level of destruction) can be directly linked to parameters of other instructions or to variables of shaders used by the object's material. These more high-level semantic attributes are also more intuitive and understandable for a broader range of users. When a filter is created that maps these semantic attributes to the more vague, lower-level parameters of a procedural instruction, this filter can be applied by designers, without requiring them to delve into the actual workings of the procedural technique.
- **Automatic content generation** instructions are techniques that handle a wide range of procedural functionality like the generation of noise or other procedural textures, performing procedural destruction on existing models or using automatic layout solving techniques to displace objects or to place new objects in the world. The availability of such techniques obviously has a significant impact on the power of the implementation of procedural filters: the more, and the more varied, the instructions in this category, the more expressive and therefore powerful procedural filters become.

An essential feature of the concept of procedural filters is their *reusability*: once a filter has been specified, including its input and output, it can be subsequently used as an individual instruction in other filters. This way we can, for example, create a filter that adds a *crack* to the material of an object. This filter, applied with different parameters, could then be used to add radial cracks to glass or to add a more straight crack to a wall.

A filters, in and of itself, does not add restrictions or constraints. However, upon creation of a filter, the user can add restrictions through the use of the semantic queries: e.g. a filter can specify to add a rust-shader to all metal objects, but it is equally possible to only add rust to instances of one specific class. It is therefore up to the designer to create a filter that matches his or hers intent. Similarly this also affects the reusability of a filter. In the application examples section, we show the use of a party filter. In it, balloons and empty cans are spread around an office scene. We could create this filter by defining that the cans need to be spread around on the desks, however this would make the filter only usable in an office scene. Applying the filter to a living room would not change the scene at all. On the other hand, if we were to specify in the filter that the cans need to be spread around on any kind of surface, this filter would already be reusable in many different kinds of scenes. Again, it is up to the designer how generic and reusable a filter is.

In the next section we will discuss an application of this proposed approach.

4. A PROCEDURAL FILTER EDITOR

Based on the approach introduced above, we developed a prototype procedural filter editor. This system provides a variety of nodes or *building blocks*, each representing an instruction, with its inputs and outputs. By interactively connecting these to (outputs and inputs of) other building blocks, one can easily create a directed graph representing the procedure intended for the filter. Typically, a filter has itself one or more input nodes, including possible user-provided values for settings, intensities, etc. Moreover, most filters have also a so-called *scene object* node as input. Scene object nodes can represent both individual objects and whole scenes

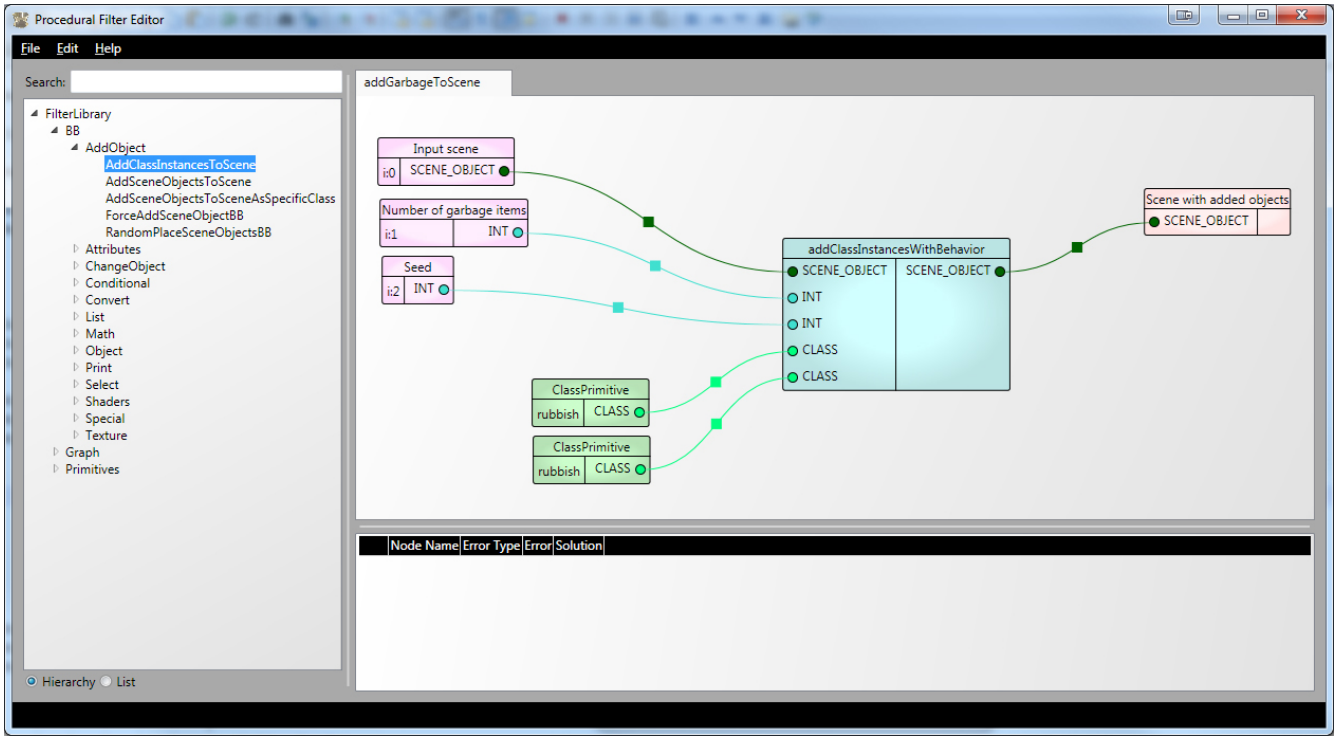

Figure 2: A screenshot of the procedural filter editor showing a filter to fill the scene with rubbish.

(i.e. compositions of multiple scene objects). In addition, after applying a number of instructions, the filter typically returns the modified scene object as its output. Scene object nodes, thus, have at their disposal both the geometric and the semantic information of an object.

Each object used in our game worlds belongs to some class of the so-called *semantic library* [19], and carries therefore all its semantics. The semantic library provides a hierarchical class database, partly based on the WordNet ontology [10], where each class specifies a set of attributes, including functional information, and also geometric relationships with objects of other classes. Among other uses, the semantic library has been deployed for handling object interactions in games using *services* [7] and in a *semantic layout solving* approach [19]. In the present context, we make use of the descriptions in the semantic library to handle semantic query instructions (see Section 3).

While many classes in the semantic library may prescribe some geometric model(s) for its instances, many other classes rely on procedural generation methods to create the geometry for their instances, as e.g. the consistent buildings generated with our integrated approach [20]. In both cases, their instances share in the corresponding attributes and semantics, and are therefore suitable for performing semantic queries. Consequently, our procedural filters are applicable to the entirety of game worlds: both manually created and procedurally generated objects.

For the sixth category of instructions (automatic content generation), we have implemented a number of powerful procedural techniques that have a wide variety of possible uses. For example, we developed building blocks to generate noise maps and crack textures, and also to handle texture composition. In addition, we developed nodes that make available the previously mentioned semantic layout solving approach [19]. They make use of the relationships specified in classes of the semantic library in order to find possible and suitable locations for instances of those classes. This functionality can be used in building blocks to add objects of a particular class to a scene, e.g. add rubbish to the front yard of a deteriorated building, spread around some garbage or empty liquor bottles in an office to create a party atmosphere.

To apply a filter, a depth-first search is performed, starting in the output node(s) of the filter. Any building block input nodes are calculated only when necessary, as decided by the building block itself. For example, the conditional block will only execute one of the two input branches based on the result of the condition.

The procedural filter editor provides a very intuitive visual editing environment. The user interface was inspired by other node based editing environments such as Shader FX [9] or Filter Forge [1]. The user can drag and drop building blocks onto the filter canvas, including a special building block that calls a sub-filter. This way, filters can be used as individual instructions of other filters, as explained in the previous section. The input and output are represented as dots on the building block and can be easily connected with each other, or with filter input and output nodes as well as primitive nodes. For primitive nodes, a constant value can be assigned in the editor.

To guide the user in the data flow, colors mark possible connection types for each node. They become gray when multiple types are still possible and are immediately updated after every added or removed connection. A screenshot of the editor, showing a filter to add rubbish to a scene, can be seen in Figure 2. The filter shown in the screenshot, has

Figure 3: Four versions of the same house: (from left to right) the first one is not filtered, the second has some cracked windows, the third one has graffiti on the walls and the fourth one has rubbish in the front porch.

three input nodes: the input scene, the number of garbage items and a random seed; and one output node: the resulting scene. The input nodes are passed along to an instruction that places instances of a class (in this case the *rubbish* class) in the scene.

In complex filters, it might become difficult to spot mistakes or missing links in the data flow. This is solved by a control in the bottom of the screen that shows warnings whenever a filter is incomplete, e.g. when the data flow is interupted, or when something else is missing in order to have a functioning filter (e.g. a missing input or output node). When no warnings are left, the designer is ensured that the filter will execute correctly.

Once it is clear the filter can execute, the designer wants to know if the output matches the expectations. To check this, the filter editor provides a preview window, where one can test and visualize the effects of applying a filter on a given scene. In that preview window, scenes can be loaded and saved and the designer can select elements in the scene. The designer can apply the created filter on the entire scene, but also on a selection of the scene, which makes it possible to test sub filters that are used as building blocks in larger filters.

For the representation of virtual worlds and their objects, we are using our own geometry representation, which is a node-based structure with nodes being either group nodes (combining multiple sub-nodes) or meshes. Meshes contain a vertex buffer, an index buffer and a material with basic characteristics like diffuse, ambient and other color values, a number of texture maps and possibly also shaders. However, it is possible to unlink the building blocks from the actual code that applies the instructions, which would make the filters and the filter editor reusable by simply reimplementing the building block code for use with other geometry representations. It is also trivial to code a new building block and add it to the current code base, by registering a new building block (including the different input and output combinations) to the filter library.

5. APPLICATION EXAMPLES

To show the application of procedural filters we used the editor, discussed in the previous section, to create a number of example filters. We created aging and deterioration filters for houses and a party filter. This included the creation of scenes, in our case using procedural content generation techniques, collecting or creating assets (e.g. models, textures, shaders...) and actually creating the filters in the editor.

The first set of filters was created for an urban environment (a street or a neighborhood). One filter applies aging effects on a house (e.g. moss growing on the walls or cracks appearing in them) and another one handles effects having to do with deterioration of buildings because of neglect or vacancy, e.g. rubbish gathering on the porch, windows getting smashed in and graffiti being sprayed on the walls. Both these filters are combinations of smaller sub-filters and blocks: e.g. using the crack texture generation block to create smashed windows, using texture composition to add graffiti to walls or using the semantic layout solving block to add cans and other rubbish to the front porch. Images of these three example filters applied to a procedurally generated Dutch-style middle-class house can be seen in Figure 3.

The creation of the cracks is handled by a building block which can create *cracked* lines to a texture. A simple crack can be created with a one dimensional midpoint displacement algorithm. To generate the typical pattern of a smashed in window, the building block allows designers to create multiple cracked lines from one center point, which can be handed as a parameter to the building block (and which is fed a random 2D point on the texture to create window cracks).

For the texture composition, we use a simple shader that combines a number of textures. By using the appropriate building blocks to add such a shader and the correct textures to a material, it becomes possible to create for example a wall with graffiti. To create the moss on the roof and walls, the same building blocks are used, however with a slightly different shader that combines textures based on a *combination* texture. In our case we used a building block that creates a Perlin noise texture, which we used as the combination texture. The *age* attribute of the building is

Figure 4: The same street but in various conditions: a) all houses are new and intact (no filter applied), b) the same houses have different ages, achieved using filters that add moss on the walls and roof, cracks in the walls and rust on the drain pipes, c) the same street, but with a *high vandalism* filter that uses additional sub-filters to produce smashed-in windows, graffiti and garbage.

Figure 5: On the left, two views from an office are shown. To the right, we see the same scenes, but with a party filter applied to them: some balloons, empty bottles and cans are spread around the room, a few cakes are placed on the desks, and the desks and computers have been slightly rotated to give a more messy effect.

used as the threshold value: in the pixel shader, we use the roof texture when the corresponding grayscale value in the noise texture is above the treshold and the moss texture when below the treshold. In other words: the higher the treshold, and thus the higher the age of the building, the more moss will be visible.

In Figure 4, we see a street filled with similar houses, generated using the same shape grammar. In Figure 4a, all houses are new. In Figure 4b, the houses are given a random age, which defines their level of deterioration. This attribute is used in filters that apply cracks to the walls, rust on the drain pipes, and, as explained above, moss on the walls and roof. Finally, in Figure 4c, a *high vandalism* filter is applied to the street, which uses a number of the previously explained filters as instructions to add graffiti to the walls, cracks in some windows and garbage on the front porches. Finally in Figure 5, we see an automatically laid out office room, with a number of desks with office appliances, all placed using the semantic layout solver discussed in the previous sections. On that scene, we applied a *party filter* which performs a number of operations, mainly spreading around objects like cans, empty liquor bottles and some balloons. It also adds a small random translation and rotation to some of the objects to give a more messy appearance to the scene.

It is important to note that the visual quality of the output of the filters also depends on the quality of the used assets. For example, if we were to use a more complex shader or a higher quality textures for our moss, the visual quality could be improved, without the structure of the filter having to change (only the values of a number of parameters like the shader or texture path).

6. CONCLUSIONS AND FUTURE WORK

In this paper we proposed a *procedural filter* approach aimed at assisting designers in customizing their virtual worlds or scenes. Procedural filters were defined as sets of instructions (i.e. procedures) to be applied on a virtual world or its objects in order to customize their appearance and give them a peculiar twist. Procedural filters are, thus, the 3D virtual world equivalent of 2D digital imaging filters. They provide step-by-step instructions of how a virtual scene should be customized and how its appearance should change based on attributes and circumstances. They allow designers to intuitively express the visual changes proper to a particular situation. We identified and discussed the categories of instructions necessary or desirable for this purpose. The recursive nature of this approach encourages reusability and allows designers to build up a relatively complex filter piece by piece. We implemented this approach within a visual editing and testing environment for procedural filters, and showed the results of a number of test filters.

When using semantic attributes, if available, filters can be created that are more intuitively parameterizable. By mapping high-level semantic attributes (e.g. 'level of destruction') to the more low-level parameters of procedural techniques (e.g. parameters of a noise function or some threshold value for texture composition), the use of such filters will become easier and more *readable*, even without knowing the exact workings of the procedural techniques involved.

Currently we are working on coupling procedural filters to

semantic *predicates*. This way we can create new objects by adding predicates to classes, and applying the coupled filters to instances of these classes. We are also looking into how filters can be deployed in a dynamic virtual environment, in order to efficiently reproduce its changing attributes.

7. ACKNOWLEDGMENTS

This research has been supported by the GATE project, funded by the Netherlands Organization for Scientific Research (NWO).

8. REFERENCES

[1] Filter Forge. `http://www.filterforge.com/`, 2011.

[2] Allegoritmic SAS. MaPZone. `http://www.mapzoneeditor.com/`, 2011.

[3] O. Deussen, P. Hanrahan, B. Lintermann, R. Měch, M. Pharr, and P. Prusinkiewicz. Realistic Modeling and Rendering of Plant Ecosystems. In *SIGGRAPH '98: Proceedings of the* 25^{th} *Annual Conference on Computer Graphics and Interactive Techniques*, pages 275–286, New York, NY, USA, 1998. ACM.

[4] D. S. Ebert, S. Worley, F. K. Musgrave, D. Peachey, and K. Perlin. *Texturing & Modeling, a Procedural Approach*. Elsevier, 3^{rd} edition, 2003.

[5] D. Finkenzeller. Detailed Building Facades. *IEEE Computer Graphics and Applications*, 28(3):58–66, 2008.

[6] E. Galin, A. Peytavie, N. Marchal, and E. Gurin. Procedural Generation of Roads. In *Computer Graphics Forum: Proceedings of Eurographics 2010*, volume 29, pages 429–438, Norrköping, Sweden, May 2010. Eurographics Association.

[7] 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*, pages 276–281, 2009.

[8] J. Lu, A. S. Georghiades, A. Glaser, H. Wu, L.-Y. Wei, B. Guo, J. Dorsey, and H. Rushmeier. Context-aware textures. *ACM Trans. Graph.*, 26, January 2007.

[9] Lumonix. Shader FX. `http://www.lumonix.net/shaderfx.html`, 2011.

[10] G. A. Miller. WordNet: A Lexical Database for English. *Communications of the ACM*, 38:39–41, 1995.

[11] P. Müller, P. Wonka, S. Haegler, A. Ulmer, and L. V. Gool. Procedural Modeling of Buildings. In *SIGGRAPH '06: Proceedings of the* 33^{rd} *Annual Conference on Computer Graphics and Interactive Techniques*, pages 614–623, New York, NY, USA, 2006. ACM.

[12] F. K. Musgrave. *Methods for Realistic Landscape Imaging*. PhD thesis, Yale University, New Haven, CT, USA, 1993.

[13] E. Paquette, P. Poulin, and G. Drettakis. The simulation of paint cracking and peeling. In W. Stuerzlinger and M. McCool, editors, *Proceedings of Graphics Interface*, pages 59–68, 2002.

[14] Y. I. H. Parish and P. Müller. Procedural Modeling of Cities. In *SIGGRAPH '01: Proceedings of the* 28^{th} *Annual Conference on Computer Graphics and Interactive Techniques*, pages 301–308, New York, NY, USA, 2001. ACM.

[15] D. Peachey. Solid texturing of complex surfaces. In *Computer Graphics (SIGGRAPH '85 Proceedings)*, volume 19, pages 279–286, July 1985.

[16] K. Perlin. An Image Synthesizer. In *SIGGRAPH '85: Proceedings of the* 12^{th} *Annual Conference on Computer Graphics and Interactive Techniques*, volume 19, pages 287–296, New York, NY, USA, 1985. ACM.

[17] Spiral Graphics. Genetica 3.5. `http://www.spiralgraphics.biz/genetica.htm`, 2011.

[18] T. Tutenel, R. Bidarra, R. M. Smelik, and K. J. de Kraker. The Role of Sematics in Games and Simulations. *ACM Computers in Entertainment*, 6:1–35, 2008.

[19] 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 -* 5^{th} *Conference on Artificial Intelligence and Interactive Digital Entertainment*, Stanford, CA, USA, October 2009.

[20] T. Tutenel, R. M. Smelik, R. Lopes, K. J. de Kraker, and R. Bidarra. Generating Consistent Buildings: a Semantic Approach for Integrating Procedural Techniques. **Submitted for publication**.

[21] J. Whitehead. Toward proccedural decorative ornamentation in games. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, PCGames '10, pages 9:1–9:4, New York, NY, USA, 2010. ACM.

[22] X. Zhang, T. Tutenel, R. Mo, R. Bidarra, and W. Bronsvoort. Specifying semantics of large sets of 3D models. **Submitted for publication**, 2011.