Using Semantics to Improve the Design of Game Worlds

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

Ruben M. Smelik
TNO Defence, Security & Safety
Oude Waalsdorperweg 63
2509 JG The Hague
The Netherlands
ruben.smelik@tno.nl

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

Klaas Jan de Kraker
TNO Defence, Security & Safety
Oude Waalsdorperweg 63
2509 JG The Hague
The Netherlands
klaas_jan.dekraker@tno.nl

Abstract

Design of game worlds is becoming more and more labor-intensive because of the increasing demand and complexity of content. This is being partially addressed by developing semi-automated procedural techniques that help generate (parts of) game worlds (e.g terrains, cities and buildings). However, most level editors rather deficiently capture and deploy designer's intent. For example, common positional or functional relationships between objects are usually limited to pre-processing a number of anticipated cases.

In this paper we propose a novel scheme for specifying high-level semantics of objects within a game virtual world, and in particular we illustrate its application to a variety of layout solving problems raised by procedural generation methods. Our approach combines the genericity of a semantic class library with the power of a layout solver, and it shows to be both very flexible and effective. Moreover, this scheme can be useful for improving both manual, automated and mixed modeling techniques, always leading to a more efficient layouting process for game worlds.

We conclude that by allowing designers to capture more of their intent and real-life knowledge in the objects with which they populate a game world, the integration of semantics will strongly contribute to stimulate content reusability, enrich the game play, and eventually also significantly cut down design duration and cost.

Introduction

In current games, especially in role-playing games and shooters, one finds a tendency towards larger game worlds that stimulate exploration and free roaming. This means more game content is to be made, while maintaining the level of detail in game models that is expected by gamers. However, the discrepancy between the high quality, almost lifelike, appearance of game worlds and the plainness of the interaction with their objects becomes even more noticeable.

In Tutenel, et al. (2008) we explained how introducing *semantics* to game worlds can help to close this gap. We define object semantics as all information, beyond the actual 3D model, related to a particular object within the game world including e.g. physical attributes like the mass or material, functional information like how one can interact with an object. Other examples could be the subject of a book, the power of a car or the comfort level of a sofa. On the one hand, capturing more detailed information about objects and the game world will enable games with more interesting gameplay possibilities, and new opportunities for smarter AI or advanced visual effects. On the other hand, during the design of a game world, semantic information proves useful as well: relationships between objects can then be used to guide the layout of a game world level, whether designing it manually or generating it procedurally. In this paper, we focus on the deployment of semantics in this design phase.

To be able to include semantic information in the design phase, we integrated a semantic class library with rule-based layout solving, an approach that can be applied to any combination of both manual design and procedural generation of game worlds; see (Tutenel, et al. 2009) for more details. Summarizing, based on the relationships described between the classes in the library, the constraints for the layout solver are derived. Our class library provides two methods of defining relationships between objects: i) classes can contain layout rules describing relationships between other classes, and ii) they can be linked through the use of feature areas defined in the class representation. By associating a class to feature areas specified in another class, hierarchies and other relationships can be expressed. Our layout solving approach allows for evaluating the validity of a location for a particular class instance in a given layout, to accommodate user-assisted design. In combination with a layout planner, which draws objects from the semantic library as well, we can create fully automatic procedural game worlds.

In this paper we discuss the added value of semantics in the design phase of game worlds. We do this by explaining

how the integration of semantics into our layout solving approach puts more of the designer's knowledge to good use, effectively aids the designer by automating several tasks and, moreover, significantly improves game play. In the next section we briefly survey research on using semantics in the design of virtual worlds and how game play can benefit from semantics. Next, we discuss the design of our semantic class library and the use of features in the class representations. Finally, we briefly describe the layout solver which maintains the class relationships between objects, and illustrate its results with some scenes generated by our integrated prototype system.

Related work

Up until now, the use of generalized semantic information in video game worlds is almost non-existing. However much of the research regarding this topic is very well applicable in games. Many ontology languages initially developed for semantics in web documents, for example *OWL* (Smith, et al. 2004), use data structures and relationships inherent to entities in game worlds. Ontology languages are used to define classes, properties and relationships between classes. Furthermore, they provide generic rules on classes and class instances, e.g. we can define that no instance of the *Person* class can have blood group O, having parents with blood groups A and AB. In research circles, OWL is already used in combination with virtual environments, e.g. to improve data for planning techniques, as surveyed by Gil (2005), or to improve interaction in virtual environments (Vanacken, et al. 2007).

Otto (2005) used the RDF language (Hayes and McBride 2004), for which OWL is a vocabulary extension, to create a semantic virtual environment focused on multi-user interaction. The general goal of creating more 'intelligent' virtual environments appeared already earlier, e.g. in Aylett and Luck (2000), who discuss the issues in combining artificial intelligence with virtual environments. At an object level, adding intelligence was proposed before, for example in the form of so-called *smart objects*. Kallmann and Thalmann (1998) define smart objects as objects in virtual environments that contain knowledge on how a user can interact with them. For a desk drawer, for example, you can specify the pulling motion to open it.

This approach was used by Peters, et al. (2003) to steer the behavior of non-playable characters (NPCs) in virtual worlds. In their approach the objects are central in the interactions between characters. A smart *bar* object contains user slots where an NPC can order a drink, following a number of steps (customer orders a drink, bartender hands over the drink, customer pays, etc).

Abaci, et al. (2005) use *action semantics* to improve planning by intelligent agents. The agents use the semantic information to decide what actions can or cannot be performed on a certain object. They define features on the 3D models to facilitate animations, e.g. where to position the hands to grasp the object. To perform the planning, an agent can query possible and relevant actions it can take on objects in its surroundings, in order to reach a goal. This way the agent can, for instance, find out that it can open a door that is blocking him from bringing a crate inside.

Since we focus on the design process of game worlds, relationships between objects are very important. We already discussed an example of an ontology language that allows the definition of high-level relationships between classes of objects. Huhns and Singh (1997) use these relationships to handle communication between agents with different knowledge domains. Agents with a complex knowledge base can still communicate with an agent with less knowledge by reasoning on the relationships between classes unknown by the second agent and classes the agent does know. Next to the basic data modeling relationships of inheritance, aggregation and instantiation, they use relationships like *owns*, *causes* and *contains*.

Functional relationships between objects can be useful in planning applications. Levison (1996) describes a system to decompose a high level task into a set of action directives. For this he created a functional hierarchy between classes. When the task of an agent is to make light, he can both use candles or a flashlight to perform this task. The functionality can therefore be defined on a common parent of these two classes.

On a lower level, we see many geometric relationships between objects in a virtual world. These are often used in scene editing applications to assist the user with placing objects. Xu, et al. (2002) use parent-child relationships to define which objects can be supported by others. And in Smith, et al. (2001) these relationships are created with *offer* and *binding* areas that can connect with each other.

In the WordsEye system of Coyne and Sproat (2001), relationships expressed between objects in natural language sentences are parsed and transformed to constraints between different 3D models and on the properties of these models, to create a layout that depicts the given sentence. Each object has information like its different subparts or the default size. Objects also contain functional information: cars and other road vehicles are linked to the verb *ride*. When parsing a sentence *John rides to town*, one of these objects is chosen to depict this scene.

The examples of the use of semantics discussed above highlight the importance of capturing relationships between classes in virtual worlds in general, and in the design phase in particular. However, despite many convincing research results shown, most of these techniques have not seeped through to commercial game development yet. One notable exception is the idea of smart objects used in The Sims™ (see Forbus et al. 2001). In the next section we show how we integrated semantics with our layout solving approach to improve the design of game worlds.

Semantic Layout Solving

We developed a layout solving approach to accommodate both user-assisted design and fully automated procedural generation. Given an initial layout, the solver can position

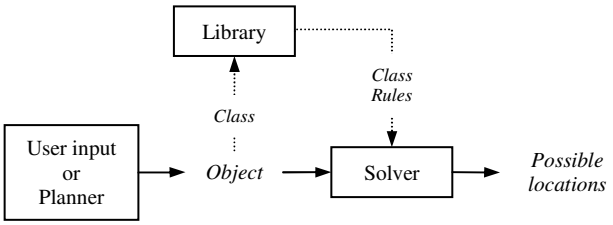

Figure 1: Scheme of the semantic layout solving approach: either the user or the planner adds an object to the existing scene; based on the associated class, the placement rules are chosen and the solver determines the possible locations for the new object.

a new object in that layout, complying with a set of rules. This initial layout can be empty or e.g. contain walls when creating a layout for the objects inside a room. By iteratively providing a set of objects to the solver, a valid layout is created. This process is represented in Figure 1.

In a manual design application, the user adds the new object. The locations deemed valid by the solver, can either be shown as guidance to the user, or the application can place the new object immediately on a valid location.

The objects can also be added to the solver by the planner. Based on a user-created plan, objects are step by step added to the solver, which in turn generates a new valid layout. This plan is a high-level procedure for the automatic generation of a game world layout. The planner will be discussed in more detail later in this section.

Semantic Class Library

The semantic class library, conceived for integration with our solving approach, has a hierarchic setup: classes inherit the properties, the feature-based representation and the placement rules of their parent classes. The class properties are available in each step of the solving process. Cupboards, for example, have a *storage volume* property, which is usable to evaluate the amount of available storage.

After creating a new 3D model, the designer associates it to an existing class in the library, or creates a new class for it with the desired properties, feature elements (explained in more detail in the next subsection) and rules. The class property values, specific to the 3D model, are defined by the designer. Based on the mesh part names and materials in the 3D model, some basic properties are calculated automatically, as e.g. the volume and mass of a model or the quantity of certain sub parts. When the class properties are defined, the model is ready to be used in the solver. An instance of the 3D model can be added directly to the solver or by referencing to the class, in which case a suitable model will be picked from the library.

Feature-based Class Representation

For each class in the library, we define a generic representation, valid for all instances of that class, that consists of a number of *object features*, which are 3D shapes containing a tag. In our solving approach, these features are used to derive valid and invalid positions for each instance of that class. Each feature type, which is designated by the tag, has overlap rules to other feature types. For example, an *OffLimit* feature cannot overlap with any other feature type and is therefore used to indicate solid areas in an object. The *Clearance* feature is used to indicate areas that should remain free to interact with the object or to reach it, e.g. the area in front of a cupboard. The *Clearance* feature cannot overlap with any other features except for other *Clearance* features, since this open area can be shared between multiple objects. An additional step of our solving mechanism consists of checking whether every *Clearance* feature is reachable from e.g. the entrance of the room. When one or more of the *Clearance* features in the proposed layout cannot be reached, the entire layout is deemed invalid.

The *Area* feature is meant to be used as a sort of placeholder for other objects and is particularly suited for the planner. An *Area* feature cannot overlap with any other feature when it is placed, but once placed, every feature type can overlap with it. This guarantees a free area, which can be filled with appropriate objects later. Figure 2 depicts this in a factory layout: first some *Area* features are added to designate a storage area, an area for some lockers and an area for forklifts (top); once these areas are positioned, the desired objects can be placed inside these areas (bottom).

The size and position of a feature shape are defined relative to the volume of the class instance, e.g. on top of, to the left of or at the center of the model, etc. This facilitates reusing the class representation for all instances regardless of the size or shape of the used 3D model. A possible extension to this approach could be unbound features, i.e. features that are not yet fixed to a shape. When the designer associates the 3D model to a class, a position and a size for each unbound feature would need to be designated either by hand, or possibly through the use of tags in the names of sub meshes of the 3D model. This way the features could be attached automatically to these sub meshes, and the model can be directly linked to the class library, without an extra processing step.

In any case, it will always take some extra time to define these feature areas and to create the class representations. However, as explained in the Related Work section, they are useful not only in the design phase of a game world. For example, *Clearance* features can be used in path finding algorithms. Every object of the *Seat* class contains one or more features designating where a character can sit down. In the layout solving phase, these features are used to, e.g. place a drink as close as possible to the *Seating* feature as one would do in a real situation. But these features can just as well be used in behavior planning or in the animation of NPCs. Since all these features can be applied in other phases of the game development cycle as well, that extra time investment is more than justified.

Object Relationships

We mentioned before that features are used in placement rules as well. Feature tags do not need to link to specific feature types and can get any tag a designer wants to use.

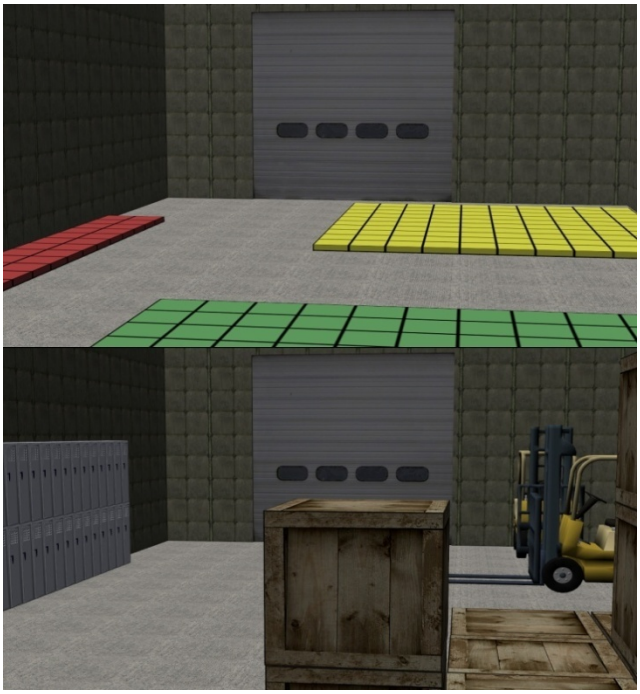

Figure 2: A factory floor layout created with the use of Area features: at the top we see the first steps of the layout plan where empty areas are reserved. At the bottom we see the areas populated with the appropriate objects.

Therefore, features not linked to a specific type do not have overlap rules, but they can still be used in placement rules. For example, when defining rules for a dining table setup, plates and glasses can be placed on top of the table. In the representation of the *Table* class, a *TableTop* feature on top of the table model is present. By adding a rule that the plates and glasses should be restricted to *TableTop* features, we are guaranteed they will be positioned on top of the table. Since walls in a room are stamped with features too, they can be used to position objects with their back against a wall, i.e. against a *Wall* feature.

These types of feature rules indirectly specify relationships between classes, but we can define rules based on direct class relationships as well. Some of these relationships hold in every case and some are specific to the situation. The latter ones are expressed in rules in a planner step, as we explain in the next subsection. The general relationships are formulated in rules directly linked to the classes. For example, we can define as a rule of the Sofa class, that when there is an instance of the TV class present in the same room, a sofa should be facing that TV.

Layout Planner

So far we have explained the basics of the semantic library integrated in the solving approach. To use this approach for procedural generation of (parts of) game worlds, a layout planner was created that iteratively provides the solver with new objects, for which the solver creates a valid layout. We mentioned that the solver finds all valid locations for a new object in a layout, and when used in combination with the planner, a certain location (scored on the basis of weight factors, as discussed in the next section) is selected. The planner works based on a procedure or a *plan*: a list of statements and rules that need to be executed in order. Examples of such rules might be: "place X instances of class Y", or "place as many objects of class Z as possible". We also added rule control elements as if-then-else statements or loops. In the plan's steps, we can refer to the property values of the already placed objects in a scene. This way we can create rules like "keep adding cupboards until the sum of all *Storage Volume* properties exceeds 1.3 cubic meters". In a rule like "place one instance of the *Seat* class", the planner picks an object from the library associated to the *Seat* class or one of its child classes. We can use the property values to guide this choice. For example, when creating a plan for a modern house, we could put a maximum constraint on the *Age* property. When the planner is unable to execute a step (e.g. when there is not enough space), the planner can use backtracking to retry parts of the procedure without necessarily rejecting the entire layout up to that point.

A virtual world is built up hierarchically. A building lot consists of a house and possibly a garden; the house consists of multiple rooms, etc. For a designer, most of these hierarchies are obvious, so he or she can employ this knowledge by creating sub-plans. The designer can make a plan to layout rooms in a house and another plan to layout objects in a room, for example. This makes the planner useful in manual scene editing as well. Procedural content generation cannot, and should not, take over the job of a game world designer, but it can alleviate it by automating some tasks. For example, when we create a sub plan for an office setup, the designer can use this as a building block for the game level. One could drag and drop an abstract *office setup* block into the game world and have the planner place a desk, a chair, a computer, etc. Instead of using fixed blocks of objects, you would then get unique blocks every time you drop one. This is somewhat comparable to the nowadays common function of terrain editors, with which the designer can draw a region that is automatically filled with randomly placed trees and shrubs.

Layout Solver Setup

In this section we briefly describe how the solving approach works. A more detailed overview of the layout solver, with all its advantages and limitations can be found in (Tutenel et al. 2009). When adding an object to the current layout, first the representation of the associated class is instantiated for this particular object. When a position is found for the new object, the features from the instantiated representation are added to the layout. Based on the new object's features and those already present in the layout, all possible locations for the new object are found. For this, the feature type rules are used: for the new object only those positions are kept that do not generate conflicts. On these possible placement areas, the class rules

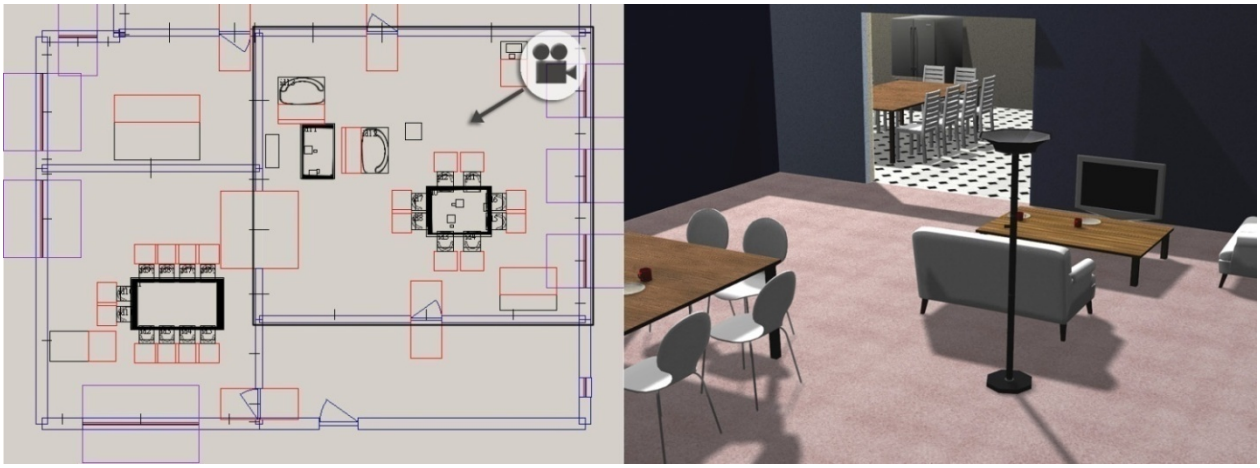

Figure 3: Automated generation: (left) 2D floor plan of a house created with our solving approach; (right) 3D visualization of the living room (from viewpoint indicated by camera in the left image).

are evaluated. In addition, when using the planner, the extra rules defined in the planner step are applied.

These feature and class rules would define either valid or invalid locations, but such a black-or-white approach is not always desirable. We want to be able to define that an object of a particular class attracts or detracts objects of another class. For this we use attractors and detractors that assign weights for the possible placement areas. These weights will deem some locations as unlikely, yet not completely invalid, for a specific object.

The output is a list of areas where the new object can be placed, possibly linked to a weight. When a designer is manually creating a world, this output can be used as a guide, e.g. by snapping to the nearest valid location or by showing the valid locations visualizing the weights. As mentioned earlier, for procedural generation, a random valid location is picked, taking the weights into account, and the object and its features are inserted into the layout.

Not every layout deemed valid is equally good. We included the ability in the solver to compare two layouts on multiple components. One score component consists of the scores designated to the different objects: the more objects, the better the layout. Because of the *Clearance* features, it is guaranteed that there is at least a minimum of free space to use the objects in the layout. In some cases, however, one might wish to increase the score of a layout when it has more open spaces (since e.g. this would imply more comfort, maneuverability, etc.). These two score components counteract each other, so we have a weight factor to balance these components. A designer can add score components based on the class properties, for example a higher score when there is more storage space.

Results

As an example of our semantics-based layout solving approach, we discuss a living room plan. Every 3D model used in the examples is associated with a class that contains, besides several properties, a set of placement rules. Usually you only need between one and three simple rules per class to obtain convincing results. The plan is a list of fifteen steps, mainly to add a number of instances of a particular class, with some constraints specified on this class' properties, such as a table with a 50 cm maximum height for the coffee table, minimum comfort level for the seats, etc. Figure 3 shows an example of a living room based on this plan and automatically laid out with this approach. On the left of Figure 3 is the 2D floor plan of the house with the living room, showing also some of the features, e.g. there are *Clearance* features in front of the sofas, behind the chairs and on both sides of doorways.

Because of the integration with a semantic class library, our solving approach is generally usable for many different layout problems. To apply the solver to a new scenario, one only needs to add the necessary classes to the library. As a second example, the solver was used for the automatic generation of a building floor plan; see the house layout in Figure 3 (left). For this, some different room types were added to the semantic class library, containing rules about e.g. the neighboring rooms, and some specific rules, e.g. requiring the hallway to be connected to a wall facing the street. In the plan, areas for each room are created with the minimum dimensions and a suitable layout for these areas is generated. In the end, a post-processing step is applied that grows the rooms to fit the building shape.

Conclusions and Future Work

In this paper we showed the usefulness of the integration of a semantic class library with our layout solving approach. We also successfully exemplified a variety of cases with the use of the planner. To further validate the approach, its integration with an interactive design environment is underway.

We also plan to extend the types of semantic information. At the moment we deploy information about the objects and their relationships. It is important, however,

to include more global semantic information like time and contextual information or cultural preferences. A room will look differently at dinner time or when the inhabitants are having a party, and near Christmas there might be some specific decorations. This solving approach is equally useful for generating outdoor environments as well. For this, it might use information about particular areas in the game world like the soil type, the depth where bedrock is reached, neighborhoods, etc.

In short, both manual and procedural game world design can be greatly improved by capturing designer's intent in the semantics of game objects. As a result, content becomes reusable and parts of the design process can be automated, which in the end will cut down design costs. Furthermore, once available, this semantics will likely stimulate game play, and present new opportunities in fields as diverse as AI, interaction and animation. Our preliminary work on the integration of semantics to improve object interaction and gameplay has been recently proposed in (Kessing et al. 2009).

Acknowledgments

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

References

Abaci, T., Cíger, J. and Thalmann, D. 2005. Planning with smart objects. In Proceedings of the International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG), 25-28. Plzen - Bory, Czech Republic.

Aylett, R. and Luck, M. 2000. Applying Artificial Intelligence to Virtual Reality: Intelligent Virtual Environments. *Applied Artificial Intelligence* 14(1): 3-32.

Coyne, B. and Sproat, R. 2001. WordsEye: an Automatic Text-to-Scene Conversion System. In Proceedings of International Conference on Computer Graphics and Interactive Technologies (SIGGRAPH 2001), 487-496. Los Angeles, California, USA.

Forbus, K. and Wright, W. 2001. Some Notes on Programming Objects in The Sims™. Northwestern University.

Gil, Y. 2005. Description Logics and Planning. *AI Magazine* 26(2): 73-84.

Hayes, P. and McBride, B. 2004. RDF Semantics. http://www.w3.org/TR/rdf-mt/. Visited April 1, 2009.

Huhns, M. N. and Singh, M. P. 1997. Ontologies for Agents. *IEEE Internet Computing* 1(6): 81-83.

Kallmann, M. and Thalmann, D. 1998. Modeling Objects for Interaction Tasks. In Proceedings of the 9th Eurographics Workshop on Animation and Simulation (EGCAS), 73-86. Lisbon, Portugal.

Kessing, J., Tutenel, T. and Bidarra, R. 2009. Services in Game Worlds: A Semantic Approach to Improve Object Interaction. In Proceedings of the 8th International Conference on Entertainment Computing (ICEC). Paris, France.

Levison, L. 1996. Connecting Planning and Acting Via Object-Specific Reasoning. Ph.D. diss., University of Pennsylvania.

Otto, K. 2005. The Semantics of Multi-User Virtual Environments. In Proceedings of the 2005 Workshop Towards Semantic Virtual Environments, 35-39. Villars, Switzerland.

Peters, C., Dobbyn, S. and Mac Namee, B. 2003. Smart Objects for Attentive Agents. In Short Paper Proceedings of the 11th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision, Plzen - Bory, Czech Republic.

Smith, G., Salzman, T. and Stuerzlinger, W. 2001. 3D Scene Manipulation with 2D Devices and Constraints. In Graphics Interface Proceedings 2001, 135-142. Ottawa, Ontario, Canada.

Smith, M. K., Welty, C. and McGuinness, D. L. 2004. OWL Web Ontology Language Guide. http://www.w3.org/TR/owl-guide/. Visited April 1, 2009.

Tutenel, T., Bidarra, R., Smelik, R. M. and de Kraker, K. J. 2008. The Role of Semantics in Games and Simulations. *Computers in Entertainment* 6(4): 1-35.

Tutenel, T., Bidarra, R., Smelik, R. M. and de Kraker, K. J. 2009. Rule-based Layout Solving and its Application to Procedural Interior Generation. In Proceedings of the CASA'09 Workshop on 3D Advanced Media in Gaming and Simulation (3AMIGAS), 15-24. Amsterdam, The Netherlands.

Vanacken, L., Raymaekers, C. and Coninx, K. 2007. Introducing Semantic Information during Conceptual Modelling of Interaction for Virtual Environments. In Proceedings of the 2007 Workshop on Multimodal Interfaces in Semantic Interaction, 17-24. Nagoya, Japan.

Xu, K., Stewart, J. and Fiume, E. 2002. Constraint-Based Automatic Placement for Scene Composition. In Graphics Interface Proceedings 2002, 25-34. University of Calgary.