A Semantic Scene Description Language for Procedural Layout Solving Problems

Tim Tutenel[*], Ruben M. Smelik[†], Rafael Bidarra[*] and Klaas Jan de Kraker[†]

[*] Delft University of Technology, Mekelweg 4, 2628 CD Delft, The Netherlands
[†] TNO Defence, Security and Safety, Oude Waalsdorperweg 63, 2509 JG The Hague, The Netherlands
t.tutenel@tudelft.nl, ruben.smelik@tno.nl, r.bidarra@tudelft.nl, klaas_jan.dekraker@tno.nl

Abstract

Procedural content generation is becoming more and more relevant to solve the problem of content creation for the ever growing virtual worlds of games, simulations and other applications. However, these procedures are often unintuitive or use vague parameters, making it somewhat difficult for a designer to express his or her creative intent. Even worse, most of these techniques lack an accessible and easy to use interface.

We have developed a generic layout solving approach to automatically create sensible content for virtual worlds. In that context, this paper proposes a high-level scene description language that allows designers to specify particular types of scenes. This description language allows designers to easily specify which objects need to be present in a scene, their attributes, and possible interrelationships. Application of the language, based on the rich vocabulary taken from a semantic library, is illustrated with several examples, showing its flexibility, intuitiveness and ease of use.

Introduction

Procedural techniques have proven to be a valuable solution to the problems of ever increasing virtual worlds. However, most of these techniques offer quite obscure parameterizations, making it unintuitive to understand how each parameter influences their output. Moreover, both procedurally generated and hand-crafted models are typically restricted to the mere geometric representation, failing to capture most designer's intent in a virtual world.

In our previous work (Tutenel et al. 2009) we focused on a sub domain within procedural generation: *automatic layout solving*. We developed a layout solving approach that, given a *procedure*, a set of *objects* and their *interrelationships*, incrementally generates a logical and realistic scene layout for these objects. For this, all objects are instantiated from a *semantic class library*, which classes already contain many relationships used in the layout solving process.

Copyright © 2010, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

The next question arising is then: how to create that *procedure*? Designers typically do not start by thinking of *how* to build something, but rather on describing *what* that something is. So, again: how can we provide them with an intuitive way to describe the scenes they devise?

In this paper we answer these questions introducing a visual description language aimed at defining generic scene types. This language profitably builds on the rich semantics of the semantic library mentioned above. We explain in some detail how this language is built up and how it can be used. For example, to describe a kitchen you would basically specify (possibly with some qualifiers) which objects *should* be present in it, and maybe which other objects *might* also be found in a kitchen. Naturally, more indications and requirements may be given, further guiding the generation.

We first give a brief overview of related work, including our semantic layout solving approach. Subsequently, the semantic scene description language is introduced, and the main challenges of its conversion to the procedures that feed the semantic layout solver are dealt with.

Related Work

One of the key factors for our approach is the availability of knowledge, or semantics, about the entities and objects in the virtual world. Semantic data is scarcely used in entertainment game worlds, except possibly for manual annotations. In the research arena, however, there are several notable examples of how semantics can improve the creation of virtual worlds as well as the experience of interacting with them (Tutenel et al. 2008).

Research towards the semantic web yielded many ontology languages for semantics in web documents. The structures and relationships used by these languages, such as RDF (Hayes and McBride 2004) or OWL (Smith et al. 2009), are, in principle, suitable to express semantics about virtual worlds as well.

Early work on introducing more intelligence and knowledge to virtual worlds was done by Aylett and Luck (2000), who discuss several issues on combining artificial intelligence and virtual environments. On a smaller scale,

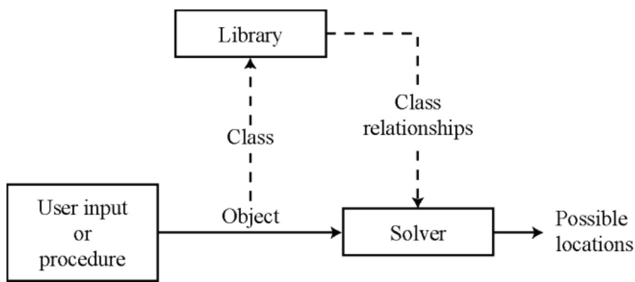

Figure 1: Layout solving approach

adding intelligence to individual objects, for example in the form of so-called smart objects, has been proposed by several authors. Kallmann and Thalmann (1998), for example, define smart objects as objects in virtual environments that contain knowledge on how a user can interact with them, e.g. the pulling motion required to open a desk drawer.

The WordsEye system (Coyne and Sproat 2001) generates specific scenes based upon a natural language description input by the user; e.g. the user can ask for a scene with a chair to the left of the table next to the door. It shows that using natural language to describe a scene is a valuable technique, although still far from the generic descriptions we need in a declarative modeling framework.

Germer and Schwarz (2009), inspired by Akazawa et al. (2005), use an agent-based approach to create furniture layouts. However, their system is restricted to furniture layouts and they only use a single parent-child relationship for every object in the scene.

There are some successful commercial systems using procedural modeling to speed up virtual world generation. CityScape, by PixelActive, and CityEngine, by Procedural Inc., are two packages that focus on urban environments. They include basic landscaping possibilities as well as several ways to procedurally add roads, districts and buildings to a virtual world. The interface and general workflow of such systems already guard you from many details behind their procedural techniques; however the scope of the generated worlds is limited to the elements shipped with these packages. Our approach is targeted to more generic layout solving problems, managing the generation of any virtual world elements that can be reduced to a layout solving problem.

Another commercial system that allows for procedural generation is Houdini, by Side Effects. However, despite its potential to procedurally create objects within a much larger scope, it also requires designers to specify most of the procedures and technical detail themselves, which is exactly what we want to avoid with our approach.

Semantic Layout Solving Approach

As explained above, our layout solving approach uses the relationships between classes defined in the semantic library. Using a procedure, instances of these classes are added one by one to the scene by the solver in a realistic and logical layout. The layout solver can also be used for manual scene editing, by which the user, instead of the procedure, will fetch each new object (see Figure 1). The solver then gets the object's relationships from the library and uses them to determine all its candidate locations. In this section, we briefly discuss the layout solving approach, particularly the semantic library and the procedures.

The *semantic library* is a hierarchic class library of which the concepts and the parent-child relationships between the classes are based on the WordNet database (Miller 1995). Every class can contain attributes, predicates, services, materials and relationships. *Attributes* describe the basic features of the class, e.g. the genre and number of pages of a book. *Predicates* are adjectives that can be associated to an object, e.g. an antique closet or a comfortable bed. *Services* represent the capacity of an object to perform a particular action. *Materials* encapsulate a number of common object properties, e.g. its density, boiling temperature or flammability. *Relationships* mainly consist of geometric relationships between classes, used by the layout solver to find sensible, valid locations for objects when generating a scene layout. All elements from the library can be used to create queries in the description language. We can, for example, declare that all flammable objects should be placed in a separate, locked area or that a kitchen needs at least one object that can heat up food.

Next to these elements, classes also contain *features*, defined as generic shapes associating semantics to an object's model. For example, most physical object classes have a front, back or a top feature defined, and a bookcase has storage features defined on every shelf. These can be used in the relationships that are contained in the semantic classes. For example, we want a coat rack to be placed with its back at 5 cm of a wall and within 1 m of a door. And in a desk setup, we want to place a computer on the top of a desk, with the front feature of the screen facing the office chair. Feature types can have embedded layout semantics, e.g. off limits features cannot overlap any other features and clearance features can only overlap other clearance features (which are used to guarantee free space, e.g. in front of a cupboard or a vending machine).

For every new object added to a scene, the layout solver will find possible locations based on the new object's relationships and features, and on those already present in the scene. The output is a list of candidate, possibly weighted, locations. For manual editing, this output can be used as a guide, e.g. by snapping to the nearest valid location or by visualizing all valid locations, showing their weights. For procedural generation, a random valid location is picked, taking the weights into account, and the object and its features are inserted into the scene. A more detailed explanation of the layout solving approach can be found in (Tutenel et al. 2009).

P*rocedures* serve to create particular classes of scenes (e.g. a street or a living room) or parts thereof (e.g. an office

workspace, consisting of a desk, a chair, a computer and some other stuff on the desk). In this way, scenes can be described and created in a hierarchic way.

The two main operations of a procedure are *picking* an object, by performing a query on the semantic library, and *handing* it *on* to the layout solver to be placed in the scene. Queries are made using the vocabulary of the semantic class library, for example requesting ‘an antique armoire’, ‘some object that provides (the service) heating’, ‘a desk made of wood’ or ‘a tall building with at least 10 floors’.

Object placing is typically performed without further parameters, according to the relationships defined in its class, but can also be extended with scene specific relationships. Procedures can also contain conditional elements and loops to yield context-specific behavior or to place multiple instances of the same object. For example, we can specify to place office workspace elements for every employee in a given office, or to use antique, expensive furniture when generating the boss’ office. This can also be used to add some variation, e.g. add between 2 and 4 plants in a room.

Given a procedure, which is an ordered set of the discussed operations, and a random seed a complete and valid scene layout can be generated. Based on the procedure operations and the relationships defined in the semantic library, the solver makes sure every object in the scene is placed in a valid and logical way.

Procedure-based layout solving used as described above generates satisfactory results, but is still restrictive and rigid, as evidenced for example by the ordered structure of procedures. We want to provide designers with all its power, while allowing them more descriptive freedom. Semantic scene descriptions, introduced in the next section, offer designers a more natural and less formal way of expressing their intent when describing a scene class.

Semantic Scene Description Language

Our proposed semantic scene description language is a visual language that allows a designer to define the different elements of which scenes of a particular type consist.

The main goals of our language are:

1. describing which objects or components can or should be present in a given scene class;
2. describing the relationships between the available objects; and
3. discerning variations depending on time and context.

In other words, it allows designers to specify which and how objects should occur in every scene instance of a given type. And, provided we manage to automatically convert descriptions into procedures, the latter can then automatically generate various instances of the described class scene. That conversion will be dealt with in the next section, here we discuss the main features of the language.

We will also explain how context-dependent changes can be made to the descriptions to adjust to varying circumstances and conditions between different instances (i.e. specific scenes) of a class of scenes.

The main building block to achieve the goals above consists of *description entities*, defining which objects need to be present and how they should be placed, possibly using some scene specific placement relationships.

An entity can contain one or more objects to be placed under the same conditions. The number of instances to be placed can be precisely defined (e.g. add '3 pens' or 'between 5 and 10 paperclips' on a desk), or it can be more loosely defined (e.g. 'enough shelves to fit at least 200 books', or 'approximately one tree per ten square meters').

A description entity consists of two components: a description of the objects that need to be placed (*what*), and some additional relationships that need to be satisfied when placing the objects (*how*). These two components are analogous to the 'pick' and 'place' operations of the procedures.

In the semantic description editor, to describe which objects need to be placed designers can simply drag and drop elements from the semantic library on the description entities. Additionally, the editor also provides a textual input method, with which an entity can be 'picked' using a simple natural-language like description grammar.

The second component of a description entity deals with the placement of its objects, and consists of the scene specific relationships between them. Most relationships are inherent to the objects' classes in the semantic library, but relationships specific to their role in the scene need to be described here. For example, when describing a storage facility where many desks are stored, we want to discard the common geometric relationships between desks: the desks can be placed back to back or maybe even stacked on top of each other to save space in the store. Relationships in the scene description language are defined with the same vocabulary used in the semantic library.

Every scene can have a main shape constraint: this can be an area, a path or just a point that defines the general shape of the scene. For example, a kitchen or forest is constrained by the outline of the room or forest area and a street is constrained by the path it runs on. The descriptions allow selecting feature shapes from the scene as shape constraints of child objects and some basic transformations on these shapes. This allows us, for example, to select the border of a scene, split it up in points and lines two meters apart and place fence poles on the points and barbed wire constrained by the lines, to generate a fenced area.

A collection of description entities makes up a *scene class semantic description*, i.e. a generic, high-level definition of a specific class of scenes. In other words, it describes not *one* particular scene but *all scenes* of a certain class, e.g. a dining area, an office, a street, a dungeon, a space ship interior, a forest or an industrial zone.

To discern more specific cases in a scene class, we introduce the notion of *context*, which is a set of conditions that together define a specific scene variant. Designers can define context-specific behavior by changing the basic

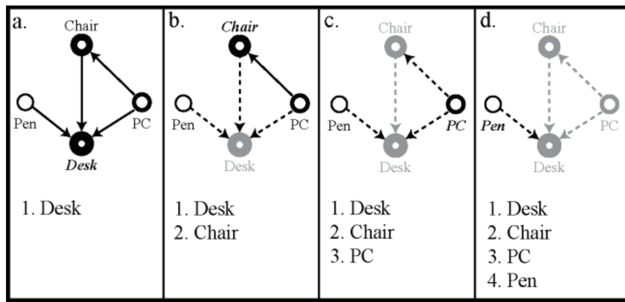

Figure 2: Creating an order for objects based on the dependency graph.

description for each particular context. A context can be defined by means of (i) conditions on the scene or on the class attributes of the scene, (ii) the presence or lack of a particular predicate, or (iii) global semantic data (e.g. safety conditions of the neighborhood). For example, when creating the description of a *home* class, we might want to create a context called *large house* based on the total available area for the house. In this context, we will want to place more bedrooms and bathrooms, we need a larger dining room, perhaps two garages instead of one, etc. Once defined the conditions for which the context holds, the designer can adjust the description to fit that context. Depending on the context, entities can be added, removed or altered in any way: e.g. adding relationships, changing the number of objects needed or adding or removing predicates in the description entity.

Since every description in itself defines a new class in the semantic library, descriptions can follow a hierarchic scene composition, e.g. we can create a description for an office building, which includes instances of an office room class, which in turn includes multiple instances of a desk setup. This allows designers to focus on the core elements of every scene, while incrementally specifying the structuring of each child element in a separate description. This makes the entire approach more scalable and reusable.

The semantic description language is aimed at providing designers with an intuitive way to specify particular types of scenes that can be regarded as a layout solving problem, e.g. placing furniture in a room or trees and plants in a forest, laying out objects on a desk or creating the layout of an industrial zone. Designers merely have to focus on the different elements that make up a particular type of scene: the ordering of the different objects is handled by the conversion to procedures and the common object relationships are found in the semantic class library.

In informal interviews with designers regularly using procedural modeling, it became clear that especially our hierarchic approach to the generic definition of scenes is well suited to their way of thinking. They tend to break up every element to its key components and then, in turn, design these different components as separate elements. Also the ability to create scenes constrained by a base shape fits very well in their normal working methods.

Converting a Description to a Procedure

We need to convert the semantic scene descriptions to procedures to be usable in our layout solving approach. For this, we first need to order the description entities to add them in an ordered procedure. Then we need to encompass procedure operations with loops to handle amounts defined in the entities and with conditional statements to handle context-specific operations. A final step is an optimization phase where we add some operations (e.g. additional checks) to ease the workload of the solver when executing the procedure.

We first need to derive a global ordering for all objects in the description. For this, we use the partial ordering in a dependency graph, in which the objects are represented as the nodes, and every relationship between two objects as a directed edge: outgoing edges point to objects on which a node's placement will depend.

Next, we sort the list of objects according to the following criteria:

First: least outgoing edges to objects not yet picked;
Second: most incoming edges;
Third: most outgoing edges to already picked objects;
Fourth: largest object.

Notice that in the first criterion, we only consider the edges to object nodes that are not picked yet. When an object depends on an already picked object, there will be no problem in the generation phase (i.e. when a scene is being generated based on the created procedure) since the object on which it depends will already be placed in the scene. In the second criterion we pick the object with the most incoming edges, since most other objects depend on its placement. In the third criterion, we pick the object with the most outgoing edges to already picked objects, because the more dependencies an object has, the more restricted the choice for possible locations is. For example, if a PC should be placed on a desk, in front of the chair (2 dependencies) and a phone should just be placed on a desk (1 dependency), you do not want the phone to be placed first since it might end up in front of the chair, possibly hindering the placement of the PC. The last criterion, based on the size, prevents small objects from hindering the placement of bigger object. When the size difference is significant, we give this criterion a higher importance. Since a description and a procedure work with object queries, and the actual objects are picked only at procedure execution-time, we do not have an exact size for each object at the conversion step. Therefore the average size of all possible objects that match the query is used as an estimate.

For example, suppose we want to create a scene with a desk, a chair in front of the desk, a PC on the desk and facing the chair, and a pen on the desk. In Figure 2a, the starting graph is shown. The desk is the only independent object (i.e. without any outgoing edges), so this is picked first. Both the chair and the pen now only have outgoing edges to already picked objects (see Figure 2b). But since the chair has one incoming edge and the pen has none, we pick the chair first. In Figure 2c, both the PC and the pen

Figure 3: Some example scenes built with descriptions (top: factory floor, middle: office, bottom: a road through a forest)

have no outgoing edges to objects that are not yet picked and no incoming edges, but since the PC has the most outgoing edges to already picked objects, and is also the biggest, it is picked first. Finally we pick the pen.

In the ordering process we could run into conflicts when circular dependencies are found in the graph, e.g. a scene with a cupboard next to a standing clock, which should be placed next to a couch, which in turn should be placed next to the first cupboard, all along the same wall. In this circular reference, none of the objects can be placed. To solve this, we will pick one of the objects in the circular reference based on the criteria mentioned before. We now remove the relationship connected to the outgoing edge of the picked object that creates the circle reference. The object will be placed without maintaining that relationship, and since the circle is now broken in the graph, all other objects can be ordered without any problems.

Once object sorting is finished, a procedure is created with pick and place operations (see previous section) for every object in the list.

The amount defined in each description entity (this can be an exact amount, a range or a distribution) is handled by a repeat loop in the procedure. The context-dependent elements of a description are encompassed by a conditional statement in the procedure, based on the conditions defined for the corresponding context.

These two steps result in a procedure that adequately reflects the scene description. However, we also want additional rules to improve the solving process for that procedure. An object query can return objects of varying sizes. It would be useless to pick the largest of objects from the query, when trying to fill a small scene. Therefore we would want to add some additional procedure steps to handle these situations appropriately: when trying to fill a small kitchen, we don't want the solver to pick a giant refrigerator, since it will be impossible to fill the entire scene. The same idea could be applied to entities with a ranged amount, e.g. between three and six kitchen cabinets. In small scenes, the lower end of the range could be used, and in bigger scenes the higher end. That way, we are also less likely to end up with a giant kitchen with only a few cabinets and a small refrigerator.

Results

The scenes in Figure 3 are created based on example descriptions for a factory floor, an office and a road running through a forest.

The factory (Figure 3 top) is built up of an area for the pallet racks, a vehicle area and a fenced area for dangerous goods. The fence and pallet racks are created using shape-constraints as explained in the section on the description language. For example, the area shape for the pallet racks is subdivided in a number of rows, using a feature transformation, and on each row an empty pallet rack is created that contains *pallet storage* features. The factory description contains an entity that ensures the placement of some pallets within these *pallet storage* features.

The road geometry is created using a separate system that works based on a path and profile. The profile definitions can contain features such as *roadside* or *bicycle path* that can be used to place roadside objects in the descriptions. For example, in the bottom example of Figure 3 a lamppost is placed every 20 meters on the *roadside* feature, facing the *center* of the road. Trees in this example are placed using a description entity with a distribution.

The next paragraphs describe the building of an office scene (Figure 3 center) from the user perspective. Every model used in the scenes is first added to the semantic library and linked to a class. The basic building block for the office scene is a description for a single desk setup. This setup includes entities for a desk, a desktop computer and a comfortable office chair, for which common relationships are available in the semantic library. A fourth entity adds some objects on the desk: some pens, a mug and a binder. The main relationships for these objects' classes are overridden and an additional relationship defines that they should be placed on the top feature of the

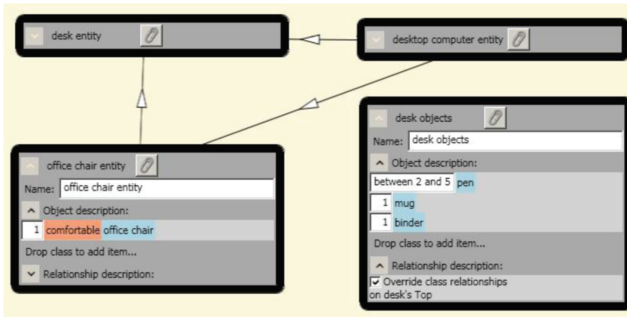

Figure 4: A description for a single desk setup with four entities. Arrows show the common relationships between entities.

desk. This description, which is shown in Figure 4, is now exported to a procedure.

Now a *desk setup* class is added to the semantic library, to which the procedure is linked. In this class some relationships are defined: the desk setups should be placed next to other desks or opposed to other setups. In the office description, this class is used in an entity containing either a fixed amount of *desk setup* instances or e.g. a distribution based on the room size. Finally entities for some cupboards and a coat rack are included. Again, the relationships for these last two classes are already contained in the semantic library. Using our semantic description language, linked with our layout solving approach, we can create a description usable for any office space we need in our virtual world in mere minutes.

Conclusions

Due to their highly technical nature, fine-tuning the output of procedural generation techniques comes down to tweaking some vague parameters, often resulting in a trial-and-error approach. In this paper we present a high-level semantic description language that makes the specification of scene layouts easier and more intuitive. This method adds an extra layer of usability to a traditional layout solving approach. With our visual language, a designer can easily build a scene description by defining the basic entities of which that scene consists. Using both the common relationships defined in a semantic class library and other scene-specific relationships added to the description entities, a suitable layout can be generated for each instance of the described scene.

Furthermore, the ability to create context-specific behavior allows a description to adjust to specific conditions or circumstances, making it usable for a range of different instances of the same scene class.

By creating an automatic conversion of these high-level descriptions to our procedures, we combined the power of our semantic layout solving approach with the ease of use of a clear and simple scene description.

Currently we are working on ways to enhance both the visual quality and realism of the generated scenes by applying a finishing pass of effects that change the look of the scene, e.g. adding dust and dirt to age objects or snow to the rooftops during winter. Another important aspect of our ongoing work is maintaining semantic consistency while editing: maintaining relationships and designer's intent of the descriptions after manual changes to a scene.

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

Akazawa, Y., Okada, Y. and Niijima, K. 2005. Automatic 3D Scene Generation Based on Contact Constraints. In Proceedings of the 8th International Conference on Computer Graphics and Artificial Intelligence (3IA'2005), 51-62. Limoges, France.

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.

Germer, T. and Schwarz, M. 2009. Procedural Arrangement of Furniture for Real-Time Walkthroughs. *Computer Graphics Forum* 28(8): 2068-2078.

Hayes, P. and McBride, B. 2004. RDF Semantics. http://www.w3.org/TR/rdf-mt/. Last visited May 7, 2010.

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.

Miller, G. 1995. WordNet: A Lexical Database for English. *Communications of the ACM* 38(11): 39-41.

PixelActive CityScape. http://pixelactive3d.com/ Last visited May 7, 2010.

Procedural inc. CityEngine. http://www.procedural.com/ Last visited May 7, 2010.

Side effects Houdini. http://www.sidefx.com/ Last visited May 7, 2010.

Smith, M. K., Welty, C. and McGuinness, D. L. 2004. OWL Web Ontology Language Guide. http://www.w3.org/TR/owl-guide/. Last visited May 7, 2009.

Tutenel T., Smelik R.M., Bidarra R. and de Kraker K.J. 2008. The Role of Semantics in Games and Simulations. *Computers in Entertainment* 6(4): a57.

Tutenel T., Smelik R.M., Bidarra R. and de Kraker K.J. 2009. Rule-based Layout Solving and its Application to Procedural Interior Generation. In Proceedings of the CASA workshop on 3D Advanced Media In Gaming And Simulation (3AMIGAS), 15-24. Amsterdam, The Netherlands.