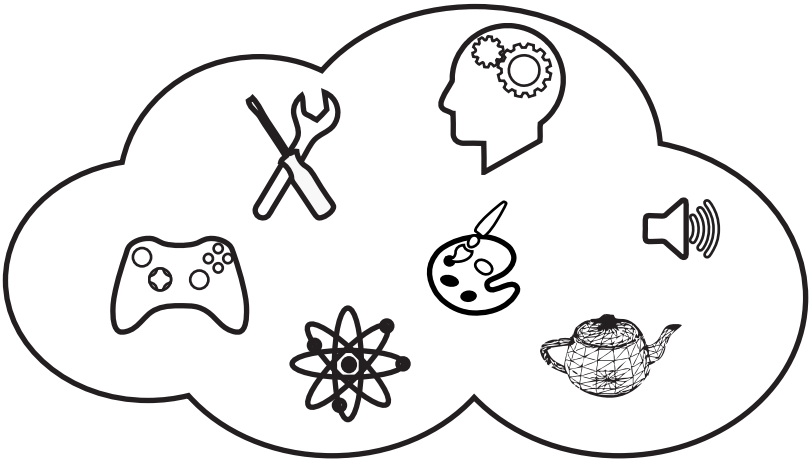
Semantic Game Worlds



Tim Tutenel

Semantic game worlds

Semantic game worlds

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universtiteit Delft,
op gezag van de Rector Magnificus prof.ir. K.C.A.M. Luyben,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen
op maandag 17 december 2012 om 12:30 uur
door

Tim TUTENEL

Master of Science informatica
Universiteit Hasselt, België
geboren te Tienen, België.

Dit proefschrift is goedgekeurd door de promotor:

Prof.dr.ir. F.W. Jansen

Copromotor:

Dr.ir. R. Bidarra

Samenstelling promotiecommissie:

Rector Magnificus,	voorzitter
Prof.dr.ir. F.W. Jansen,	Technische Universiteit Delft, promotor
Dr.ir. R. Bidarra,	Technische Universiteit Delft, copromotor
Prof.dr. C.M. Jonker,	Technische Universiteit Delft
Prof.dr. K. Coninx,	Universiteit Hasselt, België
Prof.dr. J. Whitehead,	University of California, Santa Cruz, USA
Dr. I.S. Mayer,	Technische Universiteit Delft
Dr. F.P.M. Dignum,	Universiteit Utrecht

ISBN 978-94-6203-259-0

Keywords: Semantic game worlds, declarative modeling, procedural content generation

Financial support
The work presented in this thesis was supported by the GATE project, funded by the Netherlands Organization for Scientific Research (NWO) and the Netherlands ICT Research and Innovation Authority (ICT Regie)

Preface

The research described in this thesis has been performed at the Computer Graphics and Visualization Group of Delft University of Technology. It was part of a large national research program entitled GATE (*Game Research for Training and Entertainment*, see `http://gate.gameresearch.nl`) aimed at advancing the state of the art in four different research themes: modeling the virtual world, virtual characters, interacting with the world, and learning with simulated worlds.

Within this first theme, Delft University of Technology investigated automatic creation of virtual worlds in two complementary PhD projects, started in parallel in the summer of 2007: the first project, in collaboration with TNO, focused on intuitive methods for procedural content generation, and was performed by Ruben M. Smelik [80]; the second project, focused on the role of semantics in game worlds [88], was performed by myself, under the supervision of Rafael Bidarra, and resulted in this thesis.

Put shortly, Ruben's work, of which SketchaWorld [85] is an excellent exponent, was directed at assisting designers to *better create* content; my work, instead, was centered on assisting designers to *create better*, richer content. The clear complementarity among the projects has led to a very natural and fruitful collaboration, the best example of which is the work described in Chapter 6, on a method for the generation of consistent building models, performed in cooperation with both Ruben Smelik and Ricardo Lopes. Ricardo is now continuing the research on virtual worlds in the group, focusing on automatically creating game worlds that are adapted to the player [52].

Contents

CHAPTER 1

INTRODUCTION

The visual quality of game worlds increased massively in the last three decades. Representations evolved from pixelated two dimensional drawings to stunning, beautifully lit, and highly detailed three dimensional visualizations.

Figure 1.1: 'La trahison des images' (The treachery of images) by René Magritte (1928-1929).

However, looks can sometimes be deceiving. In his painting 'The treachery of images' (Figure 1.1), René Magritte portrayed a pipe with the, at first glance, contradictory message *this is not a pipe*. The message, however, is clearly true since it is only an image of a pipe: there's no place to put the tobacco and you cannot smoke it. Game worlds suffer more and more from this problem: the visual resemblance with real life objects is great, but not when it comes to their behavior.

The closer game worlds depict reality, the more noticeable it is for gamers when objects do not behave accordingly. And the better graphics get, the bigger this gap between visual and behavioral realism becomes. As stated by Roger Chandler [14]:

> I do not expect a blocky, pixelated tree to sway in the wind or splinter realistically when I blow it to bits with a rocket launcher. But if that tree looks nearly identical to the one in my front yard, then it will be a noticeable distraction if it does not act like the real thing.

This feeling of distraction is somewhat comparable to the negative feeling that is caused by representations of people (e.g. robots or virtual humans) that look and act almost, yet not entirely, like actual humans, expressed by Masahiro Mori in the *uncanny valley* hypothesis [62].

Moreover, this lack of coherence between the visual representation of the world and the way it feels, behaves or reacts, hinders the immersion when playing a game. More specifically it breaks spatial immersion, which Bjork and Holopainen [9] explained as the experience when playing in a perceptually convincing game world, i.e. when it both *looks* and *feels* real.

When we interpret natural language, similar problems arise: there are often ambiguities in the *meaning* of words, e.g. the same word can mean different things. In the field of linguistics, the study of meaning and the relation between words or symbols and the concept they describe is called *semantics*. In other fields this concept is used as well, one of the best-known ones being the *semantic web*. This is a web of data that aids machines in mapping information on the Internet to its meaning.

In this thesis we want to set a first step towards applying this concept to game worlds. Gradually throughout this work, our idea of game world semantics will become more concrete, however as for now, the following working definition can suffice: "game world semantics is all information on a game world and its objects beyond their mere visual representation, including how they are constructed, what materials they are made of and how they behave towards other objects." We will derive a number of guidelines that are essential to any form of semantics used to describe game worlds. Furthermore we will investigate new methods and applications for semantics specifically targeted at game worlds. Ultimately, we want to get a step closer in bridging the gap between the *look* and the *feel* of game worlds.

1.1 Problem statement

To better understand why this gap is present in many games, it is important to take a look at how a game engine is typically structured. A game engine is an integrated software system of many different components. These components can be 2D or 3D rendering engines, physics engines, sound engines,

scripting interfaces, artificial intelligence (AI) components to steer agents or handle pathfinding, networking components, scene graph control and many others. Developers often assemble a game engine as a patchwork of new components, components created for previous games and middleware components providing ready-to-use solutions for particular elements of game development. Next to the many technical issues involved in integrating all these different components, it is a challenging task to create and maintain the coherence between all the different representations of an object in the game world.

These objects are represented in the first place as a set of *properties* used for both development and gameplay purposes. One can think, for example, of a unique ID, the maximum amount of ammo (in case of a gun), or the price (in case of a shop item). In addition, an object is *presented* in the game, whether that is visible by means of a geometric model, a particle system, or an icon, or audible through the use of sounds and music. Increasingly often, some *physics* information is present as well, like a mass value and a bounding box, which can be used by a physics engine to properly apply physics (e.g. gravity) and detect collisions. Finally, a game object can show some basic *behavior*, usually defined by scripts that, for example, can prescribe how to move, what animation to trigger, or how to cause damage to the avatar of the player. Although presentation, physics, and behavior make use of some properties, they are usually stand-alone, and only defined for their particular purpose. There is, therefore, a general lack of coherence between the properties used by all different game engine components.

This coherence is nowadays more difficult to maintain, as the popularity of exploration and sandbox-style games is pushing the demand for bigger and more detailed game worlds. Because it would be too expensive and time-consuming to manually create every possible detail in these game worlds, the importance of procedural modeling techniques that can automatically generate parts of game worlds, has significantly increased. However, these procedural modeling techniques often use vague parameters which are quite unintuitive. Moreover they are only able to generate limited and specific parts of a game world. In order to integrate all these different techniques to generate coherent worlds, many issues need to be overcome, as addressed by Smelik [80]. In general terms, there is a lack of an intermediate language that allows structured communication among different techniques in order to combine their output into one coherent whole.

Interactivity is perhaps the most distinguishing element of games compared to other art forms. However it makes it all the more difficult for their creators to deliver a spatially immersive, consistent, and believable product. It is one thing to design, or generate, a coherent game world, it is yet another to keep it coherent. A game world is not only changing and responding to a player's or agent's actions but also to objects and other entities in the game world. For example, when it starts raining, the earth will become wet and turn into mud. This will affect the attributes of that patch of land as well: walking in the mud will become more difficult and slow, requiring more effort. Maintaining a consistently realistic game world is a major challenge for any game developer.

We argue that what is truly missing here is a *glue* to keep everything together: glue to *combine* the different components of a game engine, and more specifically the different representations of the objects used by these components; glue to *integrate* different procedural generation techniques; and glue to *maintain* the coherence when the game world is evolving over time.

As mentioned before, semantics is often used as a way of expressing meaning, often specifically for virtual environments, as we will discuss in Chapter 2. However, such efforts present a lack of integration between semantics for the purpose of VR applications (often geometric in nature), agent control or interaction. Therefore it seems that an integrated semantic representation is necessary to be applied in the context of game worlds. Moreover, to combine with procedural generation techniques, specific requirements on a semantic specification language have to be considered, because of (i) the geometric nature of such techniques and relationships, (ii) the specific demands on meshes and structures due to the high-performance nature of games, and (iii) the procedural generation process itself.

It is important to consider why such a powerful idea of a single semantic representation for all game components is not yet used today. Most game developers only add object and world data in an ad-hoc way, specific to their current game and specific for the component that requires it. An obvious reason is the extra work it involves. It clearly involves more efforts of the designers to add this generic semantic information discussed above. It is therefore important that specifying semantics is not too time-consuming, and that such specifications are reusable between different development projects. Up until now, there is also no uniform way to represent semantics for game worlds, let alone reason over it. In this thesis we attempted to introduce a uniform model for game world semantics, based on a number of guidelines. Next to this, we worked out several applications for this semantics.

1.2 Research question

All of this leads us to the following main research question:

How can semantics improve the creation and consistency of game worlds?

To answer this question, we will answer the following key questions:

1. What is the role of semantics in the generation of coherent game worlds, both manual and procedural?
2. How can game designers be assisted in the specification of semantics?
3. How can the semantic consistency of a game world be maintained in an evolving context?
4. How can this integration of semantics influence gameplay?

1.3 Methodology

In this research project, the first step in answering these questions was identifying some major challenges in the game development process where semantics, as it exists in many different research fields, could play an important role. We particularly looked at fields where semantics is considered as a knowledge representation that allows applications to reason over it. The information gathered from these different research fields, combined with a detailed look at the current state of the art in game development, were distilled into a set of guidelines for any semantic model targeted at game worlds.

From these guidelines we derived a novel specification model aimed at *semantic game worlds*, which are game worlds that are populated with objects enriched with semantics. This model can be seen as a specific game world domain ontology. We then used this semantic model in many different fields of game development and gameplay, to investigate what impact this semantic representation had in each of those fields. We took a particular focus on procedural content generation, both when creating the specification model and when researching novel applications for it. As mentioned in the previous section, this brings forth very specific requirements to the structure of the semantics that should be used. Therefore, our model combines both geometric and non-geometric information about the game world.

All this was integrated in a game prototyping system, that we presented to game designers in order to discuss their views and evaluation of this work.

1.4 Contributions

In answering the above research questions, we made the following contributions to the field of game technology:

- A set of guidelines for any semantic representation of game worlds.
- An extension of previous semantic modeling research to represent *semantic game worlds*.
- A demonstration of how semantics can enable designers to create more coherent game worlds.
- The use of a shared semantic representation to integrate procedural generation techniques.
- The application of semantics to maintain the consistency of evolving game worlds.

1.5 Thesis outline

A visual overview of the outline can be seen in Figure 1.2. After this introduction, Chapters 2 to 4 propose how semantics for game worlds can be specified. Chapter 2 explains in more detail what semantics means in the context of game worlds, Chapter 3 sets out some guidelines for semantic game world representations and Chapter 4 proposes our semantic model for game worlds.

The next four chapters discuss how semantics can be used. Chapter 5 shows how semantics can improve procedural modeling techniques for game worlds by applying it to layout solving. We used the vocabulary from our semantic model to allow designers to create *descriptions* that specify how the resulting layouts should look like: what objects needs to be available and how they should be placed relative to one another. Such high-level semantics is translated into a set of concrete instructions that steer a general-purpose layout solver.

Chapter 6 discusses the use of semantics to integrate different procedural building modeling techniques. The semantic model serves as a vocabulary that allows multiple procedural modeling techniques to construct a single, shared representation of the scene or the world they are operating

on. The relationships defined in the semantic model are used to spot conflicts between elements constructed by the procedural modeling techniques. These conflicts are flagged and left to the individual techniques to handle.

In Chapter 7 procedural filters are introduced, which provide semantics-based customizations of game worlds. Just like imaging filters can give particular twists and effects to images, procedural filters are able to customize the appearance of game worlds, without changing the basic structure of the world. As in the previous chapters, the semantic model provides an intuitive vocabulary to specify the behavior of the filters.

Chapter 8 focuses on the use of semantics at runtime, i.e. when a game is actually being played. We introduce the concept of dynamic semantic game worlds, which are semantically-rich game worlds in which object interaction, influences of objects on their surroundings and other dynamic changes are consistently represented and automatically handled in the game world.

The next two chapters show some applications of semantics. In Chapter 9 we introduce the notion of procedural prototyping. The ideas discussed in the previous chapters are implemented into a set of tools and integrated in an environment aimed at rapid prototyping of games aided by both semantics and procedural generation. We use this environment to perform an evaluation of the concepts in this research work. Chapter 10 includes several examples of applications of our semantic model in external projects.

In Chapter 11, we evaluate our semantic model and some of the tools we built to test it. A number of developers were asked to share their opinions on both the concept of semantics in game development and some of the applications that were created. Their hands-on experience with some of our tools is shared in this chapter as well.

Finally, in Chapter 12, we end the thesis with the conclusions drawn from this research project.

1.6 Related publications

Parts of this thesis were previously published as follows:

- Chapter 2: The role semantics can play in games, simulations and object modeling is first published in [88] and later in [10].
- Chapters 4 and 8: The semantic model and in particular the concept of services is discussed in [45] and [46].
- Chapter 5: Semantic layout solving is described in [89], [91] and [90].
- Chapter 6: Consistent building generation joint research with Ruben M. Smelik and Ricardo Lopes and is published in [92].
- Chapter 7: Procedural filters are presented in [93].

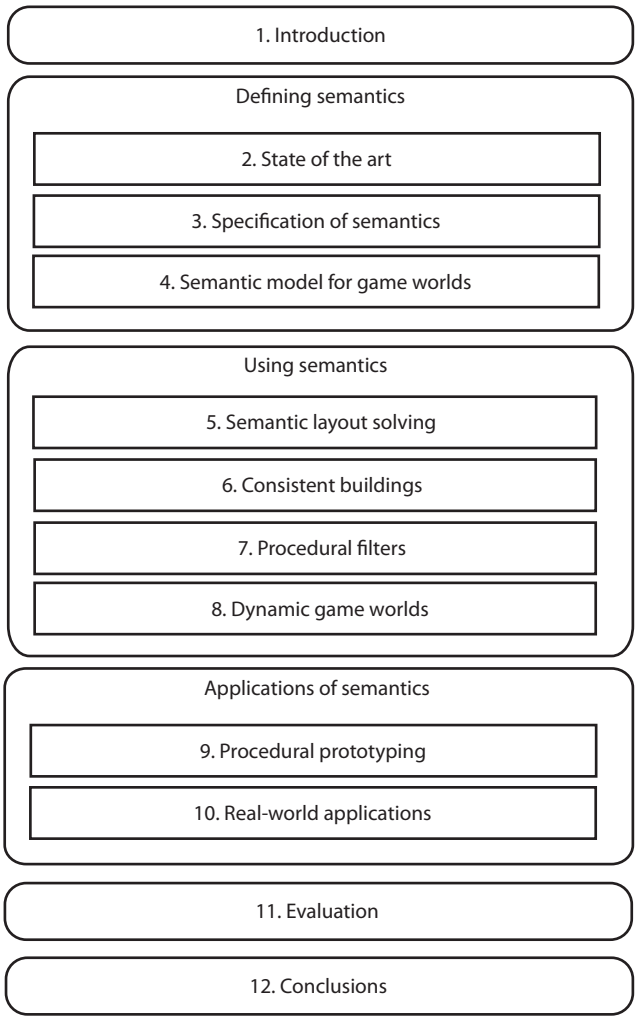


Figure 1.2: Visual outline of this thesis.

- Chapter 10: Specifying semantics for large sets of 3D models is presented in [101], semantic crowds are presented in [48], and the simulation of urban area development is presented in [19].

CHAPTER 2

SEMANTICS IN GAMES AND RESEARCH

The following chapter provides an overview of the state in the art of semantics and procedural content generated in both research and games.

We will start by discussing a number of examples of research on the concept of semantics in different fields, and especially in virtual environments. This chapter is specifically focused on research work that can be applied to game worlds, both in the development phase and the runtime phase of a game. The chapter follows by surveying procedural content generation, mainly focused on layout solving techniques, façade generation and floor plan generation, since these are the topics that will be covered in later chapters.

For both subjects, we will give an overview of how far these concepts have been used in games. We will also discuss why many of the research achievements are still underused and in what ways they could benefit from using it.

2.1 Semantics in game worlds

The first section of this chapter will discuss a number of related work in academia on topics like *smart objects* and *semantics in VR*. We will also overview what kind of information on objects and especially on their behavior is used in games and where they could still learn from the research topics.

2.1.1 Smart objects

When talking about knowledge on the behavior and interactivity of virtual objects, one of the first works that springs to mind is that on *smart objects* [42], [43], [41]. Kallmann calls an object smart when it

> "has the ability to describe in details its functionality and its possible interactions, being also able to give all the expected low-level manipulation actions. This can be seen as a mid term classification between reactive and intelligent behaviors. A smart object does have reactive behaviors, but more than that, it is also able to provide the expected behaviors from its *users*, so that this extra capability makes it to achieve the quality of smart."

This work was mainly performed in the context of physical interaction in virtual environments. Therefore objects included information on the actual gestures one needed to make to operate them, like performing a pulling motion when opening a drawer. While this is not that important in traditional gameplay, this information is very important in motion-based gameplay, e.g. for games using Nintendo's **Wiimote**, Microsoft's **Kinect** or Sony's **Move** controllers.

Peters et al. [70] take the notion of smart objects further by creating objects with information about their functionality, how NPCs (non-player characters) can interact with them, and where important features of the object are situated. The objects include information about the location where a character needs to stand to interact with the object and where he or she should look at while performing a specific action (e.g., when interacting with a computer, the character should look at the screen while typing something on the keyboard).

The concept of smart objects was applied in Maxis' **The Sims** franchise. Moreover, the objects in the **The Sims 2** were structures containing all relevant information with regards to that object, including:

> "Localized string lists, executable scripts, common definition meta data, animation lists, routing information, advertising information, interaction definitions, model information." [79]

> "A one-stop resource for any information relating to objects is very useful to have both during development and at runtime. It makes the whole game much more modular, which explains how easy it is for EA to build expansion packs" [13]

according to Alex J. Champandard, creator of the influential game AI website `aigamedev.com`.

While the idea of smart objects is a very powerful one and already contains a great deal of information that is useful in game design, it is still very much targeted at a single domain, namely object interaction. In any case, it is important that a more complete semantic model, like the one that is proposed in this work, does not fall short on the object interaction domain to this technique, but on the contrary expands on the idea, integrates it with other object information and links it with other domains.

Gutiérrez et al. [31] have presented an object representation based on the semantics and functionality of interactive digital items within a virtual environment. They consider every object not only as a 3D shape, but as a dynamic entity with multiple visual representations and functionalities, allowing for dynamically scaling and adapting the object's geometry and functions to different scenarios.

As more information is represented, the design of such objects will initially take more time and effort than the usual design of plain geometric objects. To avoid having to define each property of each object in the virtual world, Ibanez-Martinez and Delgado-Mata [36], [37] introduce a system that defines object data more efficiently. They introduced an architecture that provides a general common level on which to build specific semantic representations. The common-level data bundles information that is useful in all virtual environment applications, such as the position and dimensions of an object. Another advantage of this approach is that it takes work out of the hands of designers by automatically calculating some of these application-independent properties (e.g., the dimensions of an object based on the object model). This decreases the time a designer needs to spend manually annotating features. To remain application-independent, the common level only contains low-level information and the designer can later add more specific data, depending on the application. Basically, the user needs to enter the object type, which is used to automatically generate properties like width, height, location and orientation, based on the geometric model of the object.

This is a very important aspect from a user perspective, in this case, that of the designer creating the semantic information. Automatically generated (or derived) attributes and properties for objects and, in general, ideas to automate the definition of semantics will be a key requirement when constructing our semantic model in Chapter 4 and especially when creating tools to define semantics (see Section 9.2.1).

Objects containing interaction information, for example the smart objects by Peters et al. [70], could aid in the creation of vast and highly interactive virtual worlds. If designers could be provided with fast and efficient tools to incorporate actions in objects, game worlds could be created that are much more interactive. For this, all objects should contain information about the ways a user can interact with them: what actions can the user perform on the object (e.g., pick up, throw, drink, read, wear) and what other objects are necessary to perform this interaction (e.g., a key to open a door or a striking surface to light a matchstick). When performing an action on an object, this object will provide services to the user or to its surroundings. Wearing a coat will provide warmth only to the one who is wearing it, but a campfire will provide warmth to everyone within a certain range. A detailed classification of these properties and services is necessary to explicitly define them. It can also be integrated in tools to quickly edit the properties of an object by allowing the designer to pick

a related object class. From this related class, information can be inherited to eliminate the need to define every detail over and over again.

In related research, smart objects have already been used for planning purposes [1], by defining scripts and actions. A planning algorithm is used to make the agent move through and interact with the world. To do so, it collects relevant information about the state of the world, prepares a planning step, makes a plan, and executes it, where the agent's actions are taken from the plan. In another system [4], *Teletubbies* have been turned into intelligent agents by giving them *drives*, like hunger. Now, if an agent gets hungry, it will search for objects that have the effect of decreasing the level of hunger, make a plan to reach the object, and use it. Afterwards, its other drives will guide him to the next location/object.

2.1.2 Object relationships

An important relationship is that of inheritance. Two different objects that have the same ancestor will share some similar properties but will be in another child branch in a hierarchy tree. This kind of relationship can be expressed in a taxonomy, of which a very detailed example can be found in the feature classification model of Bitters [8]. But next to a hierarchy based on properties, more detailed relationships can be defined.

In Levison's PhD dissertation [51], we see an example of a functional hierarchy. The dissertation describes a system that enables the decomposition of a high-level task into a set of action directives. The objects have information about how they can be manipulated. They contain *sensible* knowledge, which is basic knowledge like its size, position or temperature, but also *symbolic* knowledge (e.g., about its functionality). The system uses a hierarchical class structure where every child class inherits all functionality of its parent classes. The lower a class is in the functional hierarchy, the more specialized the functionality becomes. An example of this is the class **Thing**. This class is broken up into **Object** and **Artifact**. **Artifact**, in turn, is subdivided in **Tool**, **Container**, **Cover**, and **Support**. **Artifact** is the category that defines all man-made objects. **Tool** is a tool that contains a function it can perform, for example a hammer or a screwdriver. Since the functionality of an object is important information in a virtual environment, such a functional hierarchy is another key element of any semantic model for games.

Huhns and Singh [35] use ontologies with different types of relationships between objects. Next to the basic data-modeling relationships, i.e. inheritance, aggregation, and instantiation; they also cover relationships like *owns*, *causes*, and *contains*. The system for which they proposed this idea was intended to handle communication between software agents with different knowledge domains. This could be a customer agent communicating with a travel agent about a trip. The travel agent can give details about a certain flight with a 777 type plane that might not be present in the knowledge base of the customer agent. But because of the relationships, the travel agent can inform the customer agent that this 777 has airplane as a parent object, and since the customer agent will have the general term airplane in its knowledge base, he will understand the information of the travel agent. This kind of information can be used by the AI system of a game or simulation, but it could also be a source to provide meaningful and life-like interaction to the user. For example, when the user needs

to find a power source and the virtual world contains information that a car contains an engine and that an engine is a type of a power source, this car engine could be used without the designer having to define that explicitly. In this way, virtual worlds become more interactive, and can thus lead to more emergent behavior.

In the *IConS* modeling system [29], relations between objects are described as *surface constraints* with offer and binding areas, in order to place objects automatically at correct (logical) positions. An example is a lamp, of which the bottom can be placed on the top of a table. Although useful for scene composition, they also lack semantic behavior. In a related work by Xu et al. [98], surface constraints indicate how objects can be placed on surfaces of other objects, *proximity constraints* indicate how close objects should be to others, and *support constraints* are used to indicate whether an object can support other objects, or the other way around. A difference with the IConS system is that all objects are classified, resulting in a hierarchic semantic database. Instead of defining constraints over and over again, constraints can be defined once for a class, after which it can be assigned to objects that are similar. This way, for example, all kinds of tables (coffee table, dining table, etc.) will know that they can support other objects, such as lamps. These examples show the importance of semantics in the design phase: such relationships can aid the designer when manually creating scenes or even perform object placement automatically.

2.1.3 Semantics on a world level

Next to the object level we can also define semantic information on a higher level, surpassing single objects. To identify vegetation types inside a region, parameters like soil fertility and soil nutrients play a role, but also climatic circumstances, like temperature and rainfall, should be taken into account. These parameters are not related to specific objects but to an area in the world or perhaps the entire world.

We see this approach in the work of Deussen et al. [20] that consists of an ecosystem simulation model to populate an area with vegetation. The input of this simulation model is terrain data, ecological properties of plants, and, optionally, an initial distribution of plants. Based on this data and taking into account rules for competition for space, sunlight, and soil resources, a distribution of plants inside an area is generated. When a designer has the ability to include these kinds of properties in the world, many physically-based models could integrate them to create a more realistic world.

Semantic world data is also very useful when designing adaptive virtual worlds, i.e. worlds that adapt to (i) the properties of the world (e.g., weather conditions and time), (ii) the behavior of objects inside it or (iii) the user interaction (e.g. clearing forests and constructing buildings in strategy games). These higher-level, *global*, world parameters like weather and time have a huge influence on the virtual world in its entirety as well as on the individual objects. When we look back at the vegetation example above, we see that an ecosystem model could, for example, also be applied to calculate changes in a virtual forest. Trees become older, new trees and plants appear, and others die. In a strategy or empire-building game where resources need to be gathered from forests, this could add not only realism but also strategic difficulties (cutting down trees faster than the growth of the forest will deplete wood resources). Depending on seasons, the geometric appearance of plants and trees

change but also their properties, for example the resources they offer. In summer time, fruit grows in trees and can be gathered to feed the population in a strategy game. Corn fields can provide an excellent hiding place from enemies when the crops are fully grown.

Weather and time also have an influence on materials, for example surfaces begin to reveal cracks in the paint, unprocessed wood becomes weaker, and metallic objects start to rust. These parameters can be taken into account just to add visual realism: for example, modeling paint cracks [66] or aging and weathering effects on textures [54]; but at the functional level, they can also alter the role of an object in a game. A young, strong, wooden plank will be perfect to attack an enemy, while an old, mossy branch will break more easily, and is therefore useless as a weapon.

Finally, we can also include contextual information in world semantics. Examples are parameters like the economic or living conditions, or how safe inhabitants feel. This kind of information is more important in games such as city-builders or strategy games; but also in military simulation safety levels and, in general, whenever the state of the global economy can play a role.

2.1.4 Semantics in VR applications

The use of semantics in virtual environments has also been a recurring topic in the field of VR. Bille et al. [7] explain an approach for designing VR applications at a conceptual level and in terms of concepts from the application domain. They realized this using a *Domain Ontology*, which captures the domain knowledge. The advantage of using domain knowledge is that it opens up the use of VR to a much broader community since it is easier and more intuitive for a non-VR-specialist to design his VR application. This idea was improved and extended with ways to facilitate users in creating object behavior and user interaction in the VR-DeMo project [18].

Vanacken et al. [94] introduced the use of semantic information in the modeling process of interaction for virtual environments. This work extended *NiMMiT*, a notation for multimodal interaction modeling, with semantics. They extended a driving simulator with semantic *Concepts* to better express the virtual world. Those Concepts were queried to know what objects the car could not collide with and what road type the car is on at any given time.

Latoschik et al. [50] introduced the integration of knowledge based techniques into simulative VR applications. They proposed an abstract Knowledge Representation Layer (KRL) that needed to be expressive enough to define all necessary data for several simulation tasks and which additionally provides a base formalism for the integration of AI representations. They created a knowledge base defining *entities*, *attributes*, *relations* and *concepts*, and proposed to use the semantic entities as unified object models to uniformly access the KRL at runtime. This is somewhat similar to the needs in a game engine: there needs to be a common knowledge structure that can be accessed by multiple components of the game engine.

2.1.5 Semantics in current games

Although there is a trend towards more open and explorable worlds, most games still have a strict, linear story behind them. The blend between games and movies is becoming more vague and the

story of games is becoming much more of a driving factor for gameplay. In many shooters the story is often merely a short explanation of just why killing hundreds of characters is helping to save the world.

In other genres, however, the stories are becoming much more important, e.g. Team Bondi's 2011 **L.A. Noire** or Remedy's 2010 **Alan Wake**. The themes in these stories are becoming more mature as well: Quantic Dream's 2010 **Heavy Rain**, for example, placed the player in the shoes of a father having to deal with the loss of child. Characters are becoming more complex (instead of just being good or evil) as well as putting more importance to the relationships between the player's character and others, which is the main concept of the role-playing game genre, e.g. Bioware's RPG franchises **Mass Effect** and **Dragon Age**. All in all, gaming experiences are becoming much more cinematic in nature, with probably the most notable example to date Naughty Dog's 2011 **Uncharted 3**.

Although there are also examples where the player is granted much more freedom, e.g. Bethesda Softworks' 2011 **The Elder Scrolls V: Skyrim**, most game worlds still consist of little more than geometric representations of the environment and of the objects embedded in it. Using rigid object representations, as for example pure geometric mesh models, hinders current attempts to achieve dynamic object behavior in the virtual world.

Visual immersion is getting less of a problem, since the boundaries of visual realism are continuously being pushed forward by on the one hand better graphics techniques to simulate materials and natural phenomena and on the other hand improved methods to capture real life elements; e.g. Team Bondi's 2011 game **L.A. Noire** used 16 cameras to accurately capture detailed facial animations (see Figure 2.1).

However, immersion through realistic interaction often falls short. As mentioned in the previous chapter, games quite often break spatial immersion, which is the experience when playing in a convincing game world that not only *looks* but also *feels* real [9]. Gamers have come to expect, and accept, that in some games, certain reasonable actions are not available to them, although they seem logical in the context of the game. Sometimes a knee-high fence is impossible to jump, while other, higher, walls are scaled without a problem. Or some doors in the game can be opened, while others always remain closed without any apparent reason. Again, gamers learned to live with these illogical constraints without getting distracted, but this can hardly be called an immersive experience. It is, of course, impossible to create a game where all thinkable actions a person can perform have a virtual counterpart; however, it is important to make sure there is consistency in what is possible and what is not.

A similar experience is found in the behavior of the virtual objects in game worlds. On first glance, one might be quick to dismiss this: why would a player want to use an appliance or pick up a book from a bookcase in a shooter game, where such actions seem futile? It is obvious why such actions are important when gameplay is directly influenced by them (e.g. in the Maxis' franchise **The Sims**), but why would other genres benefit from adding such behavior?

Next to the aspect of the immersion (breaking the feeling of realism by not being able to perform an action that seems logical), another more important aspect is that the absence of basic object behavior cripples the player's creative thinking. A player might want to turn on a TV to distract a guard, or use

Figure 2.1: To the left, an image of actor Aaron Staton (playing the role of detective Cole Phelps in Team Bondi's 2011 game **L.A. Noire**) in a recording room where 16 cameras captured every detail of his head and facial movements. To the right, a rendering of the captured data. (Picture source: http://3danimationcgi.com)

Figure 2.2: The destruction of a house in the game **Battlefield: Bad Company 2** from developer EA Digital Illusions CE. (Screenshot source: http://battlefield.wikia.com)

the timer on a microwave to create a delayed distraction. Or, when lost for ammo, the player might want to spray the contents of a fire extinguisher in an enemy's face to make him temporarily blind or knock him unconscious by hitting him on the head with it (or with any other heavy object). Not to mention the multitude of options to create explosive devices with little more than some household products, as we remember from high-school chemistry lessons or from the fictional TV character **MacGyver**.

One notable exception of object behavior that is present more and more in games is *destruction*. The game series that became quite famous for the level of destruction available in their game worlds is the **Battlefield** series by EA Digital Illusions CE (from **Battlefield: 1942** onwards). Vehicles and buildings can be completely destroyed, often showing quite realistic behavior upon destruction (see: Figure 2.2). Destruction physics remains not limited to shooters, however, but is also used in realtime strategy games (e.g. Ensemble Studios's 2005 **Age of Empires III**), racing games (e.g. the Codemasters' **Dirt** series), action-adventures (e.g. LucasArts' 2008 **Star Wars: The Force Unleashed**), action-stealth games (e.g. Platinum Games' still unreleased **Metal Gear Rising: Revengeance**), etc.

As mentioned in the introduction, we believe that a deeper semantic representation can be of significant use in solving all issues described in this section, especially looking at what has already been achieved on this topic in academia before in this section.

2.1.6 Discussion

Careful consideration needs to be given to creating the methods for generating and editing semantic data. For this, the use of a system as that proposed by Ibanez-Martinez and Delgado-Mata [36] can be interesting, since it tries to automatically generate numerous values of the semantic data level to reduce the design time. The types of calculated data should however be extended. If a designer did not only choose the object type but also the material of the parts, the weight and other physical properties of the object could be generated. This way, without the designer having to enter a huge amount of parameters for every single object, the game would know, for example, if an object picked up by the player's character is heavy enough to injure an opponent (and, in the first place, if it is not too heavy to be picked up).

Algorithms simulating physical phenomena, as for example, the ecosystem simulation model of Deussen et al. [20], can be significantly improved in semantically rich worlds. When detailed information about the soil of a terrain, the climatic conditions, the growth cycle of the plants, and information about other plants in the environment are included into these kinds of algorithms, a more realistic effect can be achieved.

We also noticed the importance of relationships between objects. Not only physical, geometric relationships (although these are obviously very important when considering automatic object placement for example) but more importantly inheritance relationships or functional relationships between object classes. It is clear that our semantic model should be carefully constructed with the relationships between classes and their properties, whether they are visual, material, behavioral, interactive or functional.

2.2 Procedural content generation for game worlds

Procedural generation techniques have been proposed for almost every aspect of virtual worlds, ranging from vast landscapes (see *e.g.* [65, 21]) to urban environments (see *e.g.* [67, 44, 95, 96]). For the research in this thesis, we focused mainly on the content generation for buildings and scene layouts. A full survey on the topic of procedural content generation can be found in [81].

2.2.1 Automatic layouting techniques

In this section, we review a number of systems used to automatically create layouts, specifically targeted at object placement, e.g. furniture layouts, or room placement for building floor plans.

Rau-Chaplin et al. [73] and [74] present a shape grammar to layout the different areas in a house. The authors use a large number of room units, which they call tiles, instead of automating the entire layout process. A shape grammar is similar to other grammar-based rewrite system (e.g. L-systems) however, instead of using rewrite rules to transform symbols into strings, the authors describe how to transform shapes into more detailed shapes (e.g. by splitting a shape into two parts or by inserting a particular model instead of the basic shape).

Martin [57] proposes a different method to automatically creating floor plans. First a graph is generated in which every node represents a room and every edge corresponds to a connection between rooms. Next, these nodes are given an actual location on the floor plan and the rooms are formed from there using growth rules and room weights.

Hahn et al. [32] show an important advantage of procedurally generating building interiors. This research focused on generating only the rooms that are visible from the current viewpoint. This is obviously an efficient way of handling large buildings with many different rooms (e.g. office sky scrapers). To maintain changes made in the world, all changes are tracked and stored. When a room is removed from memory at one point, and is regenerated later on, the stored changes are again applied to the regenerated room.

Layout solving based on object rules is also applied in manual scene editing systems. Xu et al. [98], describe a framework in which objects contain rules describing which type of objects their surface supports. For example, food, plates or cups can be supported by a table or a counter. Smith et al. [86] use similar links, but applied to areas. Offer and binding areas between objects are defined, e.g. the area underneath a table can be an offer area that is linked to the binding area of a chair.

Gaildrat et al. [25], [77], combine constraints and semantic knowledge in the form of implicit constraints, to help the user generate a scene. In the description phase, the designer can express how a scene should look like. These descriptions are translated into constraints that are then fed to some constraint solver.

A number of constraint solving techniques have already been investigated to create room layouts in the form of space planning problems. Charman [15] gives an overview of how existing constraint solving techniques that are not specifically focused on space planning can be applied to these problems. The author discusses the efficiency of the solving techniques and compares several space planners.

Many improvements for the discussed constraint solving techniques have been researched in the years following this study, so the results concerning the efficiency are no longer relevant. The discussed techniques, with their recent improvements, are however still applicable to layout solving. The proposed planner, steered by the conclusions from the study, works with axis-aligned 2D rectangles with variable position, orientation and dimensions. Users can express geometric constraints on these parameters, which can be combined with logical and numerical operators.

Several space planning methods were developed using constraint logic programming (CLP) [71], [34]. A more recent system that used CLP was created by Calderon et al. [12]. It is a framework that generates several different layout solutions for objects, through which the user of the framework can interactively find desirable solutions. The rules for the objects are all expressed in predicate logic statements. This gives the opportunity to provide users with more or less natural language-like rules.

Automatic layouting has also been used in interactive furniture layout systems. Merrel et al. [60] present a method that assists users by suggesting furniture arrangements based on design guidelines. It "incorporates the layout guidelines as terms in a density function and generates layout suggestions by rapidly sampling the density function using a hardware-accelerated Monte Carlo sampler". Their density function contains some functional and visual criteria to judge the layouts. The functional criteria they used were, among others, *clearance* and *circulation* (which is open space around the furniture to be usable and accessible), while visual criteria included *alignment* (of objects relative to each other and to the walls of the room) and *emphasis* (a desirable focus point in the room around which other objects are placed, e.g. a fireplace). Using this system, users can loop through some automatically generated layouts, choose a desired one and, when necessary, make some manual changes.

Yu et al. [100] present a method that creates furniture arrangements based on several examples of sensibly furnished rooms. It "extracts, in advance, hierarchical and spatial relationships for various furniture objects, encoding them into priors associated with ergonomic factors, such as visibility and accessibility, which are assembled into a cost function whose optimization yields realistic furniture arrangements." Somewhat similar to the previously discussed method, this one uses a cost function. The authors now try to minimize this cost function characterizing realistic and functional layouts. The criteria used in this cost function are, among others, accessibility, visibility (televisions and paintings have visibility requirements on their frontal surfaces) and pathways connecting doors. As opposed to the previous system, this one is intended for full automation. However, adding new rules to this system requires knowledge of the cost function and also requires users to express their rule as a term in the cost function. Therefore it is not really suited for non-technical users if extension of the available ruleset is necessary.

2.2.2 Façade generation

The following two sections focus on the generation of buildings. This section first focuses on the generation of the façade of buildings, i.e. the outer walls, the doors and windows in the walls, etc. Sometimes these techniques also include mechanisms to generate the general structure of the building.

In the field of automated generation of building façades, *L-systems* were among the first techniques

to be proposed [67]. These rewriting systems create buildings by manipulating an initial arbitrary ground plan (a lot shape) with transformation and extrusion modules.

To obtain more interesting building shapes, several approaches have been devised. Wonka *et al.* [97] introduced the concept of *split grammar*, a formal context-free grammar designed to produce building models. The split grammar resembles an L-system where shapes are primitive elements rather than symbols. Coelho *et al.* [17] proposed an urban modeling process that is based on L-systems as well. This process generates, from external data, a tree-like description of the overall scene structure. L-systems are used to generate detailed building models that emerge from the abstract set of data.

In recent years, a more specialized approach, the *CGA shape grammar*, has been applied to building façades by Müller *et al.* [63]. Shape grammars have been used and described before, especially in the architectural domain [47, 11, 49]. Architects have described shape grammars as languages of design, supported by a vocabulary of shape rules. Shape rules are specified as spatial relations, where a shape on the right side of the rule is produced and replaces the symbol on the left side (depicting when the rule can be applied).

In Müller *et al.*'s case [63] and unlike a split grammar, the shape grammar uses context-sensitive rules which allow the possibility of modeling roofs and rotated shapes. They start with a union of several volumetric shapes (the building boundary) which is divided into floors. The resulting façades are further subdivided, through shape rules, into walls, windows and doors. Yong *et al.* [99] also use an extended shape grammar, but they start at the city level, producing streets, housing blocks, roads, and, in further productions, houses with components such as gates, windows, walls, and roofs. Shape grammars have become the most accepted technique for generating building façades, as evidenced by its commercial release [72]. Epic Games also included in their commercial game engine, *Unreal Engine 3* [22], a procedural artist-driven tool for constructing buildings used in the development of city-based games [40]. The procedural system uses rulesets, similar to shape grammar rules, to split façades into scopes and automatically place meshes on them.

More recently, Müller *et al.* [64] used a very different approach for constructing building façades. Their method takes an image of a real building façade as input and is able to reconstruct a detailed 3D façade model, combining imaging and shape grammar generation techniques. Chen *et al.* [16] also proposed a method for creating building façades from images, but in this case using hand sketches as input.

On a different direction, Greuter *et al.* [30] proposed an approach where a primitive form of the integrated generation of both façades and floor plans was considered. Initially, they create a floor plan by combining several primitive 2D shapes, which are then extruded to different heights. This approach is most useful for simple office buildings. Although the concept of a generated floor plan is present, it is only used for extruding building façades and not as a room layout.

Although all of the above approaches can generate visually convincing building façades, Finkenzeller and Bender [23, 24] note that semantic information, regarding the role of each shape within the complete building, is missing. They propose to capture this semantic information in a typed graph, so that detailed building façades (doors, windows, balconies, cornices, ornaments) can be

generated, in different styles, and applied to the same building outlines. Starting with a rough building outline, building style graphs can be applied to this model, resulting in an intermediate semantic graph representation of the building. In the last step, geometry is created based on the intermediate model, and textures are applied, resulting in a complete 3D building.

2.2.3 Floor plan generation

After the techniques to generate buildings façades, the next section describes some techniques to generate the floor plan of buildings. The procedural generation of building floor plans, *i.e.* suitable inner room layouts, has been the focus of several researchers.

Rau-Chaplin *et al.* [73] show that shape grammars, often applied to building façades, can also create floor plans. In this case, shape grammars are used to create a *plan schema* containing basic room units. These individual room units are recognized and grouped to define functional zones like public, private or semi-private spaces. Individual functions are then assigned to each room, which are filled with furniture, by fitting predefined layout tiles from a library of individual room layouts.

On a different direction, Hahn *et al.* [32] present a subdivision method tailored for generating, on the fly, office buildings. The initial building structure is split up into a number of floors. On each of them, further subdivisions are applied to create a hallway zone and individual rooms. A notable feature of this method is that floors and rooms are generated or discarded based on the player's position. Re-using the same random seed in the procedure assures that discarded rooms can be properly restored.

Marson and Musse [56] also introduce a room subdivision method, but based on *squarified treemaps*. They start with the basic 2D shape of the building and a list of rooms, with desired area and functionality. Treemaps recursively subdivide an area into smaller areas, *e.g.* building shape, functional zones, and rooms. In a final step, corridors are automatically created to connect unreachable rooms.

Martin [57] proposes a graph-based method, in which nodes represent the rooms and edges correspond to connections between rooms (*e.g.*, a door). Public, private and stick-on rooms (*e.g.* closets, pantries) are gradually added to the graph by a user-defined grammar. This graph is transformed to a spatial layout, and for each node, a specific amount of "pressure" is applied to make the room expand to the desired size. Lopes *et al.* [53] also propose an expansion-based method, which grows rooms in a geometric grid representing the building lot. The initial placement of room seeds is determined by a constraint solving algorithm that takes room adjacency, connectivity and functional zones into account.

In Chapter 5, we show how we applied a semantic layout solving approach to expansion-based floor plan generation. Every type of room was mapped to a TANGIBLE OBJECT CLASS in the semantic model and for each of these TANGIBLE OBJECT CLASSES RELATIONSHIPS were defined. In this context, RELATIONSHIPS will define room-to-room adjacency. However, other constraints were defined as well, *e.g.* place the kitchen near the garden, or the garage near the street. For each room to be placed, a rectangle of minimum size is positioned at a location where all defined relation

constraints hold, and all these rooms expand until they touch each other.

Charman [15] gives an overview of constraint solving techniques that can be applied to room layout generation, if seen as a space planning problem. For example, the planner the author proposes works on the basis of axis-aligned 2D rectangles with variable position, orientation and dimension parameters, for which users can express geometric constraints, possibly combined with logical and numerical operators.

Merrel *et al.* [59] recently proposed a method for generating residential building layouts. Although this approach creates complete buildings, it is highly focused on floor plan generation. The authors use a Bayesian network, trained with real-world data, to expand a set of high level requirements (*e.g.* number of rooms) into a complete architectural program (*e.g.* room adjacencies, area and aspect ratio). These architectural programs are then realized into the 2D shapes of the floor plans, through stochastic optimization over the space of possible building layouts. 3D models are generated from different style templates to fit the structure of the floor plan, including external windows, doors and roofs. Their results are different from our integration approach, since their method: (i) is specific for generating residential buildings, (ii) cannot create specific façade patterns and appearance and (iii) the façade always emerges from the floor plan, and, therefore, cannot steer the generation process. The technique is also not designed to create designs of more complex building grounds (e.g. an army camp, a motel lot, a shopping area, etc.).

2.2.4 Procedural content generation in current games

The size of game worlds has been ever increasing in the last years. The map for Rockstar's 2001 game **Grand Theft Auto III** (a little under $8km^2$) now seems tiny compared to that of Eidos' 2010 game **Just Cause 2** (over $1000km^2$). And gamers seem to expect game worlds to become bigger with every installment of a popular franchise.

One reason for the demand of bigger game worlds is the trend towards open world or *sandbox*-style games. Next to the two games just mentioned, there are many other examples, like Ubisoft's **Assassin's Creed** franchise, Rockstar's 2010 **Red Dead Redemption** or 2K Czech's 2010 **Mafia 2** (which is not truly a sandbox-style game, but did allow the player to roam the world in between missions).

Next to this, there is the constant demand for increasing detail and graphical quality in the game worlds. Games with slightly *dated* graphics will find themselves being axed by both public and critics, sometimes even despite of a high quality of gameplay.

All these trends put a huge strain on game development teams and dramatically increase the necessary budgets for titles: many current blockbuster AAA titles have budgets of tens of millions of dollars, with Rockstar's 2010 **Grand Theft Auto IV** topping this chart with a 100 million $ budget [2].

We claim that both procedural modeling into game worlds can aid game developers by decreasing time and money spent on generating game worlds without losing visual or *behavioral* details in the process. In the next two sections, we will ground these claims in some more detail.

Figure 2.3: Editing huge game worlds in the Crytek Sandbox 2 editor can become very complex.

The increasing demand for bigger and more detailed game worlds triggered a spike in the amount of time and/or people and therefore money in the creation of game worlds. Although a large part of this process is creative in nature, the more practical part involves many tasks that are very labor-intensive and, in essence, resemble high-tech variants of handicrafts. They can become very complex when editing huge worlds, as we can observe in the screenshot of the Crytek Sandbox 2 editor in Figure 2.3.

The automatic creation of game worlds is not very widespread, except in some important areas. The generation of heightmaps has been popular for many years. And automatically generating vegetation like plants and trees has become commonplace: for example, the list of commercial games using the middleware package **SpeedTree**, a package that generates virtual foliage for animations, and in real time for video games and simulations, is huge with around 100 titles [87].

However, many other features of game worlds, like houses or cities, are much less often created with the help of procedural modeling techniques, although the research results in this topic are quite impressive.

It is quite clear why sandbox games, featuring huge open worlds can significantly benefit from procedural modeling, however many other genres could benefit from it as well. Next to the impact on time and cost in the development phase, it can also provide welcome gameplay additions. Many strategy games include a *random* map option to increase the time a player can spend in the game. However this option is almost exclusive to this genre.

Shooter games could use this as an option not only to extend the lifetime of the game, but also to remove the advantage in online mode, where gamers who spend every moment of their spare time playing the game know the maps inside out. This makes it difficult to gamers who only sparsely have time to play, to keep up with them. Random maps would at least take the map knowledge advantage away from the equation. However, it is important to note, that automatic map generation for such games would involve a set of constraints and rules that are very specific to one game only and need a considerable amount of balancing.

2.2.5 Discussion

Although procedural content generation has been used quite a lot in games, e.g. random heightmap generation, procedural texture and vegetation generation, many of the techniques from academia do not seem to find their way into mainstream, commercial game development.

For that we see two main problems. First of all, there is a lack of user-control in procedural content generation techniques and their parameters are often unintuitive. More often than not, the techniques used have very little to do with the actual domain and therefore parameters that might make perfect sense to people knowing this underlying technique, have no connection to any of the vocabulary connected to the goal domain. For example, a Perlin noise map [68, 69], often used to generate heightmaps, has parameters like the initial frequency or number of octaves, but this has no meaning in the context of actual terrains or heightmaps, where one might expect terms like terrain roughness, hilliness or soil type. The concept of *declarative modeling* as proposed by Ruben Smelik et al. [85, 80], offers a solution for this problem. Declarative modeling aims at improving the efficiency of designers, by allowing them to express their design intent more directly and at a higher level of abstraction. In other words, it lets designers concentrate on *what* they want to create instead of on *how* they should model it. To allow designers with the necessary, goal domain-related vocabulary, a rich semantic model is necessary to maintain the consistency and to specify all the game world features and their relations.

The second problem is the difficulty in combining and integrating techniques that produce different elements of a game world (e.g. urban land use, road networks and buildings). Often different techniques have overlapping responsibilities: techniques to generate rivers might claim regions that overlap with regions claimed by forest creation techniques or road network generators. Again, the work of Smelik et al., specifically with their framework *SketchaWorld* [83, 84, 82] has solved a number of these issues, using a rich semantic model to map constraints and solve conflicts between different techniques.

2.3 Conclusions

Research projects have been using semantics in many different fields, as we discussed in this chapter. Physical information might spark the use of physical simulations to improve the game world or behavioral information can assist the designer in creating consistent interaction with the world. To speed up the specification process, automation should be used where possible.

In the next chapter we will use the ideas discussed in this chapter to first extract some guidelines for our semantic model for game worlds. Using these guidelines, we will distill that model and distinguish the different levels of semantics therein. Next, we will use these guidelines to propose a semantic representation model in Chapter 4.

In reviewing procedural generation techniques, we came across two important problems: the lack of user-control and the difficulty in integrating different generation techniques. We will use the semantic representation model from Chapter 4 to show the instrumental role of semantics in solving both of these problems. In Chapter 5 and 7 we give examples of how we use the vocabulary from our semantic model to ease the use of procedural content generation, blocking users from the low-level parameters of the actual techniques and presenting them with the more high-level domain-related properties defined in our semantic model. In Chapter 6, an example is given how this same semantic model serves as an intermediary representation between different building generation techniques to generate complete and consistent buildings automatically.

CHAPTER 3

DEFINITION OF SEMANTICS FOR GAME WORLDS

The previous chapter gave an overview of the current state of game worlds both in the design and in the runtime phase. We also discussed related research on semantics that shows how the quality of game worlds can be improved in both these phases.

From examining the current limitations of game worlds and studying the related work, we identified several possible improvements to the design and quality of game worlds. This chapter derives some guidelines for the specification of game world semantics for both phases. These guidelines need to make sure the proposed semantics specification will help provide those improvements.

In the previous chapter, we noticed a clear necessity for an intermediary language to integrate procedural modeling techniques, which often contain vague parameters that do not easily connect to real world concepts. These are the main gaps that our semantics specification needs to fill in the design phase.

In runtime phase, the lack of behavioral consistency can disconnect a player from the game's universe, breaking spatial immersion. We propose a number of guidelines to achieve more consistency in the game worlds, especially when it comes to dynamic game environments. Based on these guidelines we propose a three-tier specification of game world semantics: (i) the individual object level, (ii) a relationship level between different objects and (iii) an all-encompassing world level.

3.1 Guidelines for a semantic model for game worlds

When trying to apply research on semantics to video game worlds, we notice that much of the important information to store in a semantic representation of a game world is already present in some form or another. Bits and pieces of information are scattered throughout the different components of the game engine. Models are linked to scripts that describe their behavior upon interaction, the AI component stores information about agent behavior, the textures of a model often hold a clue to what materials the model is made of, etc.

Most of this information can be interpreted by computers, e.g. the scripts, however other information like filenames, comments or descriptions are only meant to help people working on these components. This means that a wealth of knowledge is not tapped to its full potential.

More often than not, there is no real cohesion between the data in these different components. A wooden table might have a 'wood.jpg' texture attached to it, but the physics engine will likely have no way of using that information to actually treat that object as made of wood. This means that designers and programmers need to define repetitious data and, moreover, this could lead to inconsistencies in the gameplay that break the player's immersion, as we mentioned in the previous chapter.

A centralized knowledge base of a semantic game world representation is therefore the basis for many important benefits. This semantic representation should be a consistent source of information that needs to be accessible by all components of the game engine and that is understandable by both man and machine.

A semantic model fit to be used for game worlds should, at least, allow designers to express all of the following aspects of game worlds:

- What a geometric model actually represents: what type of object it represents, what classes it belongs to.
- The essential characteristics of the objects in the game world, e.g. the damage that can be dealt by a sword, the amount of experience of a character or the weight of a particular object.
- The way a player (and other characters) can interact with the game world and its objects.
- How objects relate to each other: how they are placed relative to each other, what their dependencies are, ownerships etc.
- The physical characteristics of an object: e.g. what material it is made of, how it looks, how it sounds, where and how it was damaged, etc.
- How the objects behave over time, possibly influenced by other objects in the world.

This information will make it possible to have the different game components to gather information from a single, centralized and consistent knowledge base, but it will also allow people working on the game, whether they are artists or programmers, to have a better understanding of the game world

they are working on and to have a more expressive language at their disposal when communicating with the machines they work with, e.g. to more easily express their intent when creating procedural content generation algorithms.

The following two sections contain guidelines to which any semantic model should adhere to. We distilled these guidelines from (i) the current state of game design (see Chapter 2), (ii) ideas and remarks from industry professionals after a number of interviews and talks, (iii) current game development standards and practices, (iv) research in the field of semantics, and (v) basic common sense.

3.1.1 Design guidelines

We will now further mark out the requirements to which a semantic model for game worlds should adhere to.

We want to make a clear distinction between guidelines for the design phase and the runtime phase of game worlds. The first set of guidelines, targeted at design phase, focuses on:

- the design of the semantics themselves, and
- using semantics to aid designers in creating the game world.

Within these guidelines, we look at the design of game worlds from both a conceptual and a practical point of view, i.e. these guidelines are targeted at the semantic model itself and at its practical use by designers.

The first two design guidelines will target the specification of semantics, while the two other design guidelines will focus on its use during the design phase.

Guideline 1 *Inclusion of semantics should have a low impact on the design pipeline*

We mentioned in Chapter 2 that budgets for game titles have increased because it takes more people to create them and gamers expect more detail. It is quite obvious that it would be unacceptable if the inclusion of semantics would put an additional strain on development teams. This is also the most major concern developers brought up when explaining such a semantic model in interviews. Defining and using semantics should fit in smoothly in existing design pipelines and should require minimal changes.

Reusability is key in achieving this goal. It is impossible to add more information (in the form of semantics) without requiring some extra effort. However, if the semantic model highly favors reusability, these efforts could be significantly reduced. When objects share characteristics or behavior, these need to be defined only once. New objects that share characteristics with existing ones, should build on top of the existing information instead of requiring any kind of duplication.

Going from regular game worlds to semantic game worlds is quite a large step, or at least it appears to be. Therefore the impact should be reduced by integrating the definition of semantics in the existing pipeline as much as possible. Artists should be able to define necessary parts for a particular object while they are modeling it, or they should be able to assign a material to an object while texturing that model. Ideally, the tools for specifying the semantics should be integrated in the artists' preferred modeling tools.

This obviously impacts the setup of the semantic model. It should (i) include important concepts favoring *reusability*, like inheritance and aggregation - as many other semantic models do, (see Chapter 2), and (ii) allow for the use and *integration* of existing *game world assets* (e.g. models, sounds, textures and shaders).

However, because of the difference in scope and in gameplay, games will often require a different approach for the same concept. A good example of this is the damage model for the player (or enemy characters): in all games, being hit by a gun will injure a character, but some strategy games take into account where a player got hit and hinder the character's movements and speed accordingly, while many shooters use the more simple system of reducing health points when hit. This means that not all semantics is necessarily reusable between games, but that many of the basic relationships and connections between components are, e.g. getting hit by a gun leads to an injury (regardless of how an injury is designed for a particular game). Therefore the model should allow designers to override particular elements without breaking up the existing information.

We find it important that game world semantics allow for a kind of *semantic level of detail* that allows designers to specify certain behavior or information depending on the level needed for a particular game. This will allow for even more reusability between different games. Designers will be able to specify core concepts for their game into great detail, while reusing less detailed information for additional concepts, which is easier to share between games. The promising notion of semantic level of detail was not further elaborated within the scope of this research project, but is left as future work in Section 12.3.

Guideline 2 *The semantic model should provide a wide expressive range to designers*

We mentioned before that to keep up the illusion and to immerse players in the world, it is necessary that the world behaves as expected. When games play out in *realistic worlds*, gamers will expect the objects they find to behave realistically. Designers need to be able to express these behavioral rules. The concepts and objects found in the real world have already been defined in great detail. Therefore existing technologies can form a perfect base for any semantic game world model.

Providing an existing ontology based on the real world will not completely cover it for games, though. Game worlds are only limited by the creativity of their designers. Therefore a semantic model for game worlds should allow designers to create their own concepts, objects and behavior. Even games playing in realistic looking worlds often use a fictive ruleset. However, sometimes the entire game world is fictive, e.g. fantasy worlds or futuristic worlds.

There is also a huge difference in scope between games and sometimes even within the same game. A great example of this is Maxis' 2008 game **Spore**: the player led an ever-evolving creature from the cell stage, where the game focused on single-celled organisms, to the space stage, where the player could visit an entire galaxy with their creature. Designers need to be able to express concepts and behavior on any scope in a seamless and consistent way. This was another concern raised in our interviews with game designers who thought a semantic representation model might stifle their creativity.

In general the semantic model should allow designers to express their full intent without any limitations: they should never feel semantics is hindering their expressive power.

Guideline 3 *Semantics should further enable procedural generation*

We already mentioned how the sheer size of current game worlds will drive the need for new and improved procedural generation techniques. However, as we explained before, these procedural techniques often contain *unintuitive* parameters which makes them difficult to tweak without knowing the heart of the algorithm behind them. This is a result of the nature of many of the procedural generation techniques used at the moment. Often these techniques have no immediate connection with the target domain. In Chapter 2 we mentioned how Perlin noise [68, 69] is often used to generate heightmaps. However the parameters for that noise function have no immediate connection to the domain of terrain heightmaps.

A good semantic model for game worlds can help others to *understand the procedural techniques*: by hiding the unintuitive parameters and linking them to clear, understandable, goal domain related and *intuitive semantic attributes, states and characteristics*, the techniques can become understandable by any user.

Procedural techniques have been developed that can build up part of the game world (e.g. cities, streets and buildings). To allow the semantic model to be used within such generation techniques, it is important that we can express information about how the structure of the world looks like in the model. For example, we should be able to express which furniture is typically found in a room, which rooms in a house, which buildings in a city, etc. Also interrelationships between objects, i.e. how an object is usually placed relative to others, would provide these procedural generation techniques with a wealth of information.

As mentioned before, a semantic model should also allow for communication between existing techniques, to allow designers to combine and integrate multiple techniques to generate a consistent and coherent whole. Because of the geometric nature of most procedural content generation, any semantic model for game worlds should specifically allow the specification of geometric information, e.g. on the structure of objects or geometric relationships between them. Quite often, semantic models or ontologies do not have specific mechanisms or concepts to do this, making it much harder to combine them with procedural generation techniques. Because of the growing importance of procedural generation, we think that a semantic model for game worlds should allow easy specification for geometric information.

Guideline 4 *Deploying semantics should reduce, not increase, design efforts*

Based on the work we discussed in Chapter 2, we found that using information at hand, we can considerably *reduce design efforts*. One of the main examples we gave, was using physical information to *assign values automatically*: based on the material (and its density) and the volume of a model, we can automatically calculate the weight of an object. This does however require that the semantic model should be physically sound, as we will discuss in one of our gameplay guidelines.

However, from a practical point of view, it is also important to try and derive semantic characteristics from the assets used in the game. Simple techniques like using the name of a model to decide (or at least suggest) what type of object this model represents or find out the material based on the textures used in the model can significantly reduce the effort involved in designing semantically enriched game worlds.

Most of the work lies in the actual implementation of the semantic model in design tools, but for the model, this guideline has practical consequences as well. As mentioned in guideline 1, all game assets and content need to be part of the specification model. However, it becomes clear that it is crucial to have clear links between these game assets and the abstract, semantic concepts of the semantic model ontology.

3.1.2 Gameplay guidelines

This section provides guidelines to improve the gameplay quality for worlds built with the semantic model.

Guideline 5 *Designers should be able to define physically sound game worlds*

We briefly mentioned this in guideline 4: it should be possible to define physically sound systems within the semantic model. It should be possible to mathematically express dependencies between object characteristics.

Note that this does not mean that all game worlds need to adhere to real world physical laws. On the contrary: the semantic model should allow designers to build their *own system of physical laws* that are present in their own imaginative world, whether it is simplification of real-world laws or a completely fictive fantasy world.

For many years, games have been using more and more sophisticated physical simulation engines. We've seen many examples where these physical simulations led to new gameplay, a recent example being Ubisoft Montpellier's 2011 **From Dust** where the player indirectly controls the life of some island inhabitants by altering the terrain, mainly by picking up earth, water or lava and dropping it elsewhere.

Since we want to be able to build game worlds that use the same semantics for many game engine components, these physics engines are an important candidate to use the information expressed by

the semantic model. Careful consideration needs to be taken when creating this model in order to accomplish this goal.

Guideline 6 *Designers need to be able to approach game worlds and objects from different perspectives*

There are many ways one can regard an object: we can look at its shape, the texture, the composition or other visual qualities, but we can also regard it based on its function or behavior. In Chapter 2, we gave a number of examples on how different ontologies and models are used to express many of these aspects of a world.

An important way of handling this in a semantic model is allowing *multiple inheritance*, which is one of the ways to enable designers to discern objects based on multiple types of characteristics (functional, visual, material...).

Each of these characteristics needs their own set of concepts to be defined. It is important for the semantic model to include all these concepts.

Guideline 7 *The semantic model should provide a consistent way to define interaction with game worlds*

The clearest distinction between games and other types of entertainment is the clear focus on *interactivity*, making the way a player interacts with the game world one of the most important aspects of the game.

We therefore deem it very important for any semantic model for game worlds to have a detailed language to express this interaction, obviously seamlessly integrated with all other concepts of the model.

One of the most widely used semantic concepts for interaction is that of *smart objects*, discussed in great detail in Chapter 2. We think it is necessary that a semantic model for game worlds should have at least the abilities of this concept with regards to interaction, perhaps even more focused on specific gameplay-related interaction.

Guideline 8 *The world should be kept semantically consistent throughout the whole game*

It is not enough to just define a semantic game world. While playing the game, the semantic consistency of that world should be maintained. This means that the semantic model should include mechanisms that allow for this *semantic consistency maintenance* in a generic and intuitive way.

When a player interacts with the objects in a game world, the effects of that (inter)action should be applied to the game world. This can, in turn, spark more objects to react to these effects, triggering a chain reaction. A great example of a game series that heavily relied on this subject is Dynamix'

1993-1995 series **The Incredible Machine**, (a.k.a. **TIM**). The player needed to solve some puzzles by using the behavior of a number of provided objects like treadmills, balloons and different kinds of balls (each with their own weight and behavior).

We mentioned a number of times the notion of *immersion*: having a world behave as it is supposed to and in a consistent manner throughout the entire gameplay session is a very important focus point and since semantic game worlds are ideally suited to achieve that, this is an important guideline.

Guideline 9 *Semantically modeled game worlds should enable emergent gameplay*

Another guideline to keep in mind when creating a semantic model is one we discussed in great lengths in the previous chapters already: the search for *emergent gameplay*.

This guideline is closely connected to some of the other guidelines. For example, a consistent physical simulation (guideline 5), or a detailed method of interaction (guideline 7) can both lead to more emergent behavior.

3.2 Three levels of semantics

Parallel to these guidelines for a semantic model for game worlds, we distinguish three different levels of semantic information. An important distinguishing factor to organizing the information is the scope of the information.

The first level is what we call *intra-object semantics*. These are characteristics that are proper to that object, and that object only.

The second level comprises the *object interrelationships*. This includes all sorts of relationships between two or more objects. These can include part/whole relationships, placement relationships, ownership relationships, etc.

The third and final level is *world semantics*. This is information that is not specific to just some objects and rather holds for the entire world.

We will now give a more detailed description of these three levels and give some examples of the type of information that we can regard at each of them.

Level 1 *Intra-object semantics*

Under intra-object semantics, we understand all characteristics and behavior of an individual object. One of the main characteristics of an object is clearly its physical attributes, e.g. the weight of an object or its volume, but also material characteristics, e.g. whether it is flammable or can be penetrated by bullets. This is important information for guideline 5 on physical systems.

For guideline 7, on interaction possibilities, it is important to describe functional characteristics of an object, e.g. the purpose of an appliance, how it responds to user interaction or the general behavior of the object.

Obviously, more abstract, qualitative characteristics are equally important. These are characteristics that are not clearly measurable or are not to be expressed in objective values, e.g. the comfort level of a chair, the morale of an athlete or the quality of certain goods.

Level 2 *Object interrelationships*

Objects in a scene are subject to certain relationships among each other. Different objects may share properties, e.g. have the same material, but they may also be similar on a functional level, e.g. a candle and a flashlight can both offer light. Inheritance relationships are therefore clearly one of the most important ones in any kind of semantic model, as we already mentioned in guideline 1.

We also mentioned relationships based on game world structure: which objects are part of another object, e.g. a graphics card is part of a computer, a furnace is part of a kitchen or a ticket booth is part of a railway station. These objects might also have placement relationships with others, e.g. a cooker hood (or extractor hood (UK), range hood (US)) can be found at roughly one meter above a furnace or the ticket booth should be placed at the entrance of the railway station. These types of relationships are very important with regard to guideline 3.

In the previous level, we mentioned qualitative characteristics, like the quality of certain goods. While it is quite possible (and often done in games) to have qualitative characteristics that are constant throughout the game, this does not need to be the case. For this additional relationships can play an important role. The health level increase upon eating a piece of fruit will be the same for every characters, however the increase in satisfaction upon eating them, might very well depend on the appreciation of that character towards that particular type of food.

Instead of relationships between characters and objects, another often found example from games (mainly strategy games and role-playing games) is the addition of relationships between different nations or factions and between characters. These need to be expressible at the object interrelationships level of the semantic model.

Level 3 *World semantics*

Next to the object level, we can also define semantics at a global level, which we call the level of world semantics. We already mentioned in Chapter 2 that adaptive virtual worlds would greatly benefit from a generic source of information regarding the soil: e.g. nutrients in the earth or the amount of surface water. This could be used to create virtual ecosystems. A good example of this type of behavior is Rockstar's 2010 **Red Dead Redemption** where the player played a cowboy character roaming a huge open world with a living ecosystem.

Obviously time plays an important role here as well. In springtime, fully grown crop fields can be used as camouflage, and frozen rivers, in winter, can suddenly be traversed by vehicles. This could, for example, increase the strategic possibilities in strategy games.

It is clear that, although this information is not attached to any specific object in the world. But the main point to realize when designing the world semantics level in our semantic model is that attribute

values do not need to be uniform for the entire game world: temperatures and soil information will differ per region in the world. This is a clear distinction between the intra-object semantics level, where each object has a single specific value for its characteristics. This should be taken into account when creating the semantic model. Another important characteristic is that the world itself, including its characteristics, needs to be accessible by any entity in the game world. This means that when specifying characteristics for a particular type of object, concepts defined at the world semantics need to be referable.

3.3 Conclusions

This chapter contains a number of important guidelines, which we distilled from the current state of game design and research. We split these up into design guidelines on how semantics can assist designers when creating a game, and gameplay guidelines on where semantics could improve gameplay.

We also distinguished three distinct levels in game world semantics: intra-object semantics, object interrelationships and world semantics. We discussed the specific needs these levels pose on a semantic model for game worlds.

Now that we marked out the necessary requirements for a semantic model, including the levels at which the semantics needs to be specified, we will propose our semantic model for game worlds in the next chapter. In the course of this work, we will often check back in with these guidelines and assess where and how the solutions adhere to these guidelines.

CHAPTER 4

SEMANTIC MODEL FOR GAME WORLDS

In the previous chapters, we reviewed the state of the art of game world semantics in both commercial games and research and we set out some guidelines to which game world semantics should adhere to.

This chapter proposes our specification model for game world semantics. This model can be used as a ruleset to build ontologies, or alter existing ones, specifically to be used in the context of game worlds.

In this model, we define some concepts, both based on real world subdivisions of objects and based on common elements often found in games. Rules and constraints between these concepts are set and discussed.

We will also look back at the guidelines proposed in Chapter 3 and to the related work reviewed in Chapter 2. Examples from games are used to point out the importance of a concept or a rule to game worlds.

Some of the concepts we define are common in most ontologies and semantic models, however others are specifically focused on game worlds, either from a design perspective, i.e. what information can aid game developers in designing game worlds, or from the gaming perspective, i.e. what information can improve the gameplay quality. Where necessary, we will mention the importance of these concepts for game worlds and we will draw from examples from existing games to back up those claims.

The backbone of the model is based on the three levels of semantics we distinguished in the previous chapter. The main element is the entity, which can be used to specify any kind of entity one can think of in a game world: objects, characters, scenes, spaces but also more abstract, non-physical entities, e.g. a government, a company or a skill. This represents the intra-object semantics.

For these entities, we can now define characteristics or attributes, e.g. the comfort level of a

sofa, the magazine size of a gun, the happiness level of a character or the engine power of a car. Additionally we can define behavior for entities, i.e. the purpose of the entity, the ways one can interact with them and the services they provide.

For the tangible entities, we can also define the matter of which they are made, e.g. wood, metal or water. Since attributes and services can be defined for matter as well, entities can 'inherit' characteristics and behavior from the matter of which they consist.

For the next level of object interrelationships, the model allows the creation of one-to-one relationships between entities. Each relationship has a relationship type and a source and target entity. For example, we can define an *ally* relationship between two nations, an *owns* relationship between a knight and a sword or a *has knowledge of* relationship between a character and a skill.

The highest level of world semantics is represented in the model by the semantic game world itself. Since this world needs similar characteristics and behavioral information as any entity, we treat the game world as an elevated entity that is accessible by all other entities and contains all entities and relationships related to that world.

The chapter begins by elaborating on the main structure of this semantic game world. After that we further detail entities and matter by distinguishing some different categories. Attributes, services and relationships are discussed in more detail as well as the link of all concepts from the semantic model with game content like 3D models, textures and sounds.

4.1 Main structure of a semantic game world

The overarching concept that encapsulates every detail of a game world is called SEMANTIC GAME WORLD in our semantic model. It contains entities, which are instances of ENTITY CLASSES, and RELATIONSHIPS between these entities. Characteristics and behavior of ENTITY CLASSES are expressed in ATTRIBUTES and SERVICES. Since we want to define behavior for the world itself, a SEMANTIC GAME WORLD is an instance of an ENTITY CLASS as well. We formally define a SEMANTIC GAME WORLD in the following way:

SEMANTIC GAME WORLD
A SEMANTIC GAME WORLD is a virtual environment that is populated with semantically-rich entities, i.e. instances of ENTITY CLASSES, possibly connected to each other by RELATIONSHIPS. The SEMANTIC GAME WORLD is in itself an instance of an ENTITY CLASS elevated to a world level for a specific game.

The structure of the SEMANTIC GAME WORLD reflects the three levels of semantics, distinguished in Section 3.2. Each game, or at least each level of a game, contains exactly one SEMANTIC GAME WORLD, enriched by information on its behavior and characteristics, but this will be discussed later on in this chapter. The SEMANTIC GAME WORLD represents the world semantics.

The SEMANTIC GAME WORLD consists of a number of *entities*, again enriched with semantics. These entities are instances of ENTITY CLASSES which represent not only objects but also abstract

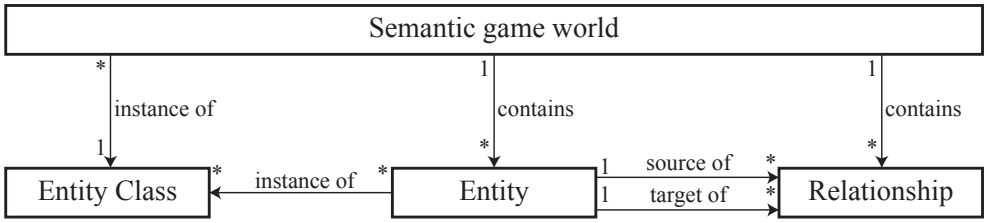


Figure 4.1: The structure of a SEMANTIC GAME WORLD containing multiple entities and RELATIONSHIPS between entities. Relationships have a source entity and target entity. Both entities and the SEMANTIC GAME WORLD are instances of ENTITY CLASSES.

entities, and represent the intra-object semantics. The characteristics and behavior of a SEMANTIC GAME WORLD are, in essence, equivalent to that of entities and therefore we define it as an entity as well, more specifically an instance of an ENTITY CLASS elevated to the world level.

Through RELATIONSHIPS, representing the object interrelationships, entities can be linked together: each RELATIONSHIP has one source and one target entity.

In Figure 4.1 the structure of a SEMANTIC GAME WORLD is presented. In the following section, we will further define and distinguish the ENTITY CLASS concept.

4.2 Entity classes

Every game world is populated with objects, from flowers, plants and trees, over streets, cars and buildings, to people and animals. More than these pure, tangible objects, we can also think of 'invisible', yet physically present entities like the inventory of backpack, a parking lot on a street or the check-in space of an airport. However, a game world can contain also more abstract, non-physical entities like a government, a skill, an idea, a story or a clue. We combine all these objects and entities underneath one concept of the ENTITY CLASS. Based on these different types of entities we notice in game worlds, we created the following division of this concept:

ENTITY CLASS

ENTITY CLASSES represent the classes for all possible game world entities. We distinguish PHYSICAL OBJECT CLASSES and ABSTRACT ENTITY CLASSES. In our semantic model we will consistently use the term *class* when talking about objects, since the model does not describe individual instances of a class, but rather classes of objects, i.e. descriptions of a collection of objects all sharing similar characteristics. There is a hierarchical structure, i.e. parent-child relationships, between classes in which children inherit the characteristics of the parent class.

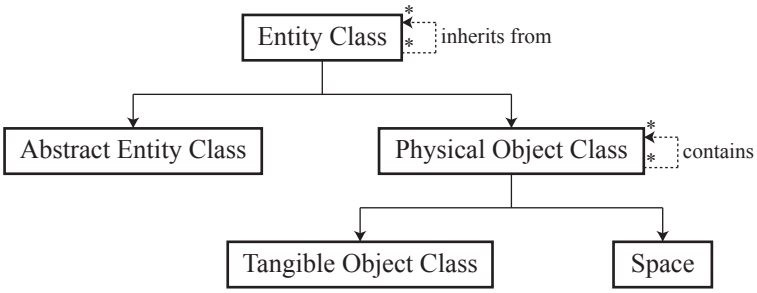


Figure 4.2: ENTITY CLASSES contain all ABSTRACT ENTITY CLASSES and PHYSICAL OBJECT CLASSES. These PHYSICAL OBJECT CLASSES are further split up in TANGIBLE OBJECT CLASSES and SPACES.

PHYSICAL OBJECT CLASS
PHYSICAL OBJECT CLASSES are all classes that represent entities that have a physical presence in the world. We split these up into TANGIBLE OBJECT CLASSES and SPACES. PHYSICAL OBJECT CLASSES can be composed of other PHYSICAL OBJECT CLASSES to create compound objects.

TANGIBLE OBJECT CLASS
TANGIBLE OBJECT CLASSES are objects made up of a single type of MATTER; e.g. a solid wooden chair or a brick wall.

SPACE
SPACES are regions bounded by a 3-dimensional shape that can contain instances of PHYSICAL OBJECT CLASSES. Examples are: inventories or parking lots. These SPACES do not have a direct physical representation other than the representation of the PHYSICAL OBJECT CLASSES inside them.

ABSTRACT ENTITY CLASS
In contrast to PHYSICAL OBJECT CLASSES, that all have a physical presence (even if not always visible in the case of SPACES), ABSTRACT ENTITY CLASSES are meant to describe entities that do not have a physical presence at all. An example of this is a **government**. The government can have physical members (presidents, ministers), but the government itself has no physical presence in the virtual world. Another example is a **company**. This company might have physical assets like factories or office buildings, but again, the company itself is an abstract entity. However we do want to describe characteristics for these ABSTRACT ENTITY CLASSES like we do with PHYSICAL OBJECT CLASSES.

The structure explained in the definitions is represented in Figure 4.2. Important to note is that ENTITY CLASSES can inherit from other ENTITY CLASSES. This includes inheriting all

characteristics and behavior.

Since next to these conceptual hierarchies (e.g. a **table** inherits from **furniture**, which in turn inherits from **physical object**), there can also be physical hierarchies to form compound objects. For example, a car is built up of some TANGIBLE OBJECT CLASSES like wheels, doors, a chassis, bodywork, but also SPACES like trunk space, multiple seating spaces and a gas tank. We therefore define that PHYSICAL OBJECT CLASSES can contain other PHYSICAL OBJECT CLASSES. This way, complex compound structures can be defined, like the example of the car. Another example is an airport. The TANGIBLE OBJECT CLASS **airport** contains, among others, a **check-in** SPACE, a **waiting** SPACE and a **security check** SPACE. The **check-in** SPACE contains a **queue** SPACE and the TANGIBLE OBJECT CLASS **check-in counter**, which in turn is composed of the TANGIBLE OBJECT CLASSES **counter desk**, **chair** and **conveyer belt**.

These TANGIBLE OBJECT CLASSES are all composed of a particular type of MATTER, which has characteristics as well. In the next section, we take a closer look at this MATTER.

4.3 Matter

In the definition of TANGIBLE OBJECT CLASS, we mentioned the term MATTER. Comparable to the real world, each tangible object is composed of a particular MATTER. Like the entities themselves, MATTER has particular characteristics. Entities composed of MATTER inherit these characteristics, e.g. a wooden table becomes flammable, since wood is flammable, and drinking a container filled with water, quenches the thirst, since that is a service defined for water. We define MATTER as follows:

MATTER

MATTER is anything that has mass and occupies space. MATTER consists of chemical elements; e.g. **Hydrogen** or **Oxygen**.

The MATTER of which an object consists plays an increasingly large role in modern games. The physical characteristics of an object, based on its MATTER, can have an important impact on the gameplay. A popular genre, the **cover-based shooter**, allows players to duck behind objects to cover from incoming enemy fire. However, depending on that object's MATTER, enemies can shoot through the cover, also depending on the weapon they use. Machine guns can rip through wooden crates, but it might take a bazooka to hit a character hiding behind concrete structures. We can state that destruction in general, which is obviously heavily dependent on MATTER, is becoming more and more important in games (also see Chapter 2).

A more widespread form of physics simulation in games is rigid-body collisions. Almost all games incorporate some form of these collisions. Again the physical characteristics of the object's MATTER, define how that object should respond to a collision, e.g. a rubber object will behave differently than a concrete one.

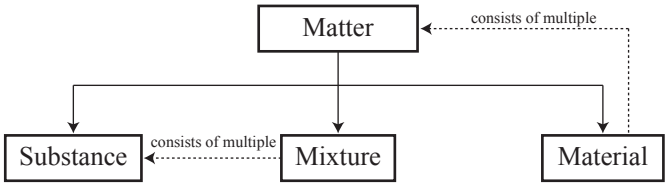


Figure 4.3: We split up MATTER in three categories: SUBSTANCES like water, MIXTURES composed of multiple substances like milk and MATERIALS composed of multiple types of MATTER like cotton.

Based on the many different types of MATTER, each with unique characteristics, we distinguish the following categories (also see Figure 4.3).

SUBSTANCE
SUBSTANCES consist of either pure chemical elements or combinations of multiple elements or other SUBSTANCES through chemical reactions; e.g. **water** or **sugar**.

MIXTURE
MIXTURES are two or more SUBSTANCES that are blended together (i.e. not chemically combined); e.g. **milk** or **sugar water**.

MATERIAL
MATERIALS are a composition of multiple types of MATTER. These can be:

- Raw materials; e.g. **cotton** or **ore**.
- Semi-finished materials; e.g. **steel**.

In the definition of SUBSTANCE, we mentioned how a SUBSTANCE can be created by chemically binding different elements or SUBSTANCES. MIXTURES can be composed of different SUBSTANCES or MIXTURES as well as MATERIALS can be composed of different MATTER, however, the main difference with SUBSTANCES is that through the chemical reaction, a SUBSTANCE is a completely new type of MATTER with unique characteristics and behavior. However, in a MIXTURE for example, the characteristics of the SUBSTANCES are maintained, e.g. the MIXTURE **milk** contains among others, the SUBSTANCE **water**. Since water can be used to extinguish flames, milk will do the same. However, two SUBSTANCES chemically bound, will not necessarily take over all characteristics. It is also much easier to break up MIXTURES into the SUBSTANCES they are composed of. In the case of a SUBSTANCE a chemical reaction will be necessary to break it up again.

We have been talking a lot about characteristics and behavior of ENTITY CLASSES and MATTER. The following section introduces the concepts of ATTRIBUTES and SERVICES which are used to specify this.

4.4 Attributes and services

Almost every object present in a game has at least some characteristics defined, or as we call them in our model ATTRIBUTES: from the **rate of fire** of a gun in a shooter to the **dexterity level** of a character in a role-playing game. Obviously also ABSTRACT ENTITY CLASSES can have ATTRIBUTES, e.g. the **book value** of a company, the **required level** of a skill (i.e. the level a character should be at before able to learn the skill) or the **popularity** of a government. Therefore, in our model, they are linked to ENTITY CLASSES to enable ATTRIBUTES to be defined for all types of entities. Next to ENTITY CLASSES, also RELATIONSHIPS and MATTER need to have ATTRIBUTES. For example, a character can have a **knows** RELATIONSHIP with a **skill**, but the proficiency of the character in a particular skill can be expressed in a RELATIONSHIP ATTRIBUTE (more examples are given in the section on RELATIONSHIPS). MATTER ATTRIBUTES can be the **fire resistance** or **density** of a material, the **nutrition value** of edible substances or the **hardness** of wood. We define them formally as follows:

ATTRIBUTE
: An ATTRIBUTE is a characteristic either of an ENTITY CLASS (e.g. the **comfort level** of chairs, **occupied** value of a parking lot, or the **net worth** of a company), a RELATIONSHIP (e.g. the **strength** of an ally relationship) or MATTER (e.g. the **density** of milk). TANGIBLE OBJECT CLASSES made up of a particular type of MATTER inherit the ATTRIBUTES from that MATTER.

An ATTRIBUTE value can be either a numerical value, a boolean value, a character string, a vector or a formula in terms of other ATTRIBUTES, e.g. the ATTRIBUTE **Gross weight** of a truck is defined as the sum of its **Tare weight** and **Net weight**.

In the real world, entities have particular functions and serve certain purposes, which should also be the case for entities in a virtual world; for example, a jacket has the SERVICE of **providing warmth** to the person wearing it or a bomb will **deal damage** to the entities in its surroundings. Some of these SERVICES are inherited from the MATTER the entity is made of: a wooden table will **fuel fire** because this is a SERVICE of the MATTER **wood**, while **water** on the other hand will have the SERVICE to **extinguish fire**.

In Chapter 8 we will give a detailed description about the composition of SERVICES, their role in game worlds and the possible requirements and effects one can define for SERVICES. We will also give an in-depth discussion of the framework we built to handle SERVICES at runtime and how this can be used to maintain semantic consistency in game worlds. For now, we define them as follows:

SERVICE
: A SERVICE is the capacity of an ENTITY CLASS or of MATTER to perform an ACTION, possibly subject to some requirements.

ACTION
: An ACTION is a process performed by an ENTITY CLASS, yielding some ATTRIBUTE value

changes or yielding new instances of ENTITY CLASSES; e.g. a vending machine providing a can of soda or an oven that **heats up** objects inside it.

That last example can be formalized in the following way:

The TANGIBLE OBJECT CLASS **oven** has a SERVICE that performs the ACTION **heat** which increases the value of the ATTRIBUTE **temperature** of objects inside it, but only if the condition "value of ATTRIBUTE **state** of the **oven** equals *on*" is satisfied.

The concept of SERVICES allows designers to describe the basic behavior and the interaction possibilities of objects in a game world in a generic way. The interaction and object behavior plays an important role in the communication between player and game world. In the previous chapters we already mentioned this importance and some examples from gameplay as well. One well-known game series where object behavior and the impact thereof on the player is a vital element of the gameplay is Maxis' **The Sims**.

Such SERVICES are often present in other games as well, but usually more sporadically and, above all, more inconsistently. Examples of these are almost all first person games, e.g. Eidos Montreal's **Deus Ex: Human Revolution**. In this game, players can pick up a vending machine, but they cannot get any products from it. There are however some objects that do allow some form of interaction, e.g. the player can turn on a water faucet or hack into a computer. Hacking is useful for gameplay, however turning on faucets is not. It is therefore quite unclear why this interaction is available, while others are not, e.g. turning off lights. We argue that a more generic and consistent way of expressing the SERVICES provided by objects, will stimulate reusability, making it easier to add them to many different games without much additional work.

4.5 Relationships

Objects and other entities in a game world are not necessarily individual entities combined in one world. Often there exist relationships between them, which we define as follows.

RELATIONSHIP

A RELATIONSHIP is a connection of a particular type associating two or more entities or ENTITY CLASSES; e.g. there might exist an **ally** RELATIONSHIP between two **nations** or an **on top of** RELATIONSHIP between a **desk** and a **pen**. As mentioned before, ATTRIBUTES can be defined for a RELATIONSHIP; e.g. the **ally** relationship might have a strength ATTRIBUTE defining the intensity of the **ally** relationship.

An example of such RELATIONSHIPS can be found in many strategy games involving diplomacy, e.g. the Firaxis series **Civilization** or many of the Paradox Interactive games like the **Europa Universalis** series. In these games nations or regions have RELATIONSHIPS defined where a score

parameter defines the strength of that RELATIONSHIP, which defines whether they are allies, neutral or enemies.

We already mentioned how conditions or requirements are defined for SERVICES. By using conditions RELATIONSHIPS can be defined implicitly. A classic example is: the **is father of** RELATIONSHIP between two **person** entities, applies when a **is parent of** RELATIONSHIP is available between these two entities and the value of the **sex** ATTRIBUTE of the source entity of the RELATIONSHIP is *male*.

Conditions for both SERVICES and RELATIONSHIPS can also be combined into the following concept:

CONTEXT
: A CONTEXT of a particular type is a collection of preconditions that describe a particular situation. In our semantic model, CONTEXTS are used (among others) to define when and how a SERVICE applies and to define implicit RELATIONSHIPS.

The concept of CONTEXT is important with regards to reusability. We will explain this using the following example. For each TANGIBLE OBJECT CLASS inheriting from **electronic device**, we want to define that it only provides SERVICES when the CONTEXT **operational** applies. Now we can define different conditions for this CONTEXT for different children of the TANGIBLE OBJECT CLASS **electronic device**. For example: for all **electronic devices** this CONTEXT would contain the condition "value of **state** ATTRIBUTE notequals *damaged*", **battery powered devices** would extend that with the condition "contains x number of instances of the TANGIBLE OBJECT CLASS **battery** for which a value for **power level** ATTRIBUTE higher than *zero*", while **electric powered devices** would extend that with "connected to instance of TANGIBLE OBJECT CLASS **power cable**, which is in turn connected to instance of TANGIBLE OBJECT CLASS **power socket**".

4.6 Game content

Now that we discussed the semantic game world model in the previous section, we need to link these entities to actual game content. Therefore, we first have some basic concepts like AUDIO, MODEL, ICON or GAME MATERIAL. Instances of these concepts can be linked to specific files, in whatever format the game requires. For example, the MODEL might link to a *.collada* file for a 3D game but also a *.bmp* file for a 2D game. We will refer to all these concepts as CONTENT. Now we will discuss some concepts that will link this CONTENT to the concepts described in the previous section.

GAME-SPECIFIC CLASS
: A GAME-SPECIFIC CLASS is an object, used in a specific game that is linked to a TANGIBLE OBJECT CLASS and a type of MATTER (or a PHYSICAL OBJECT CLASS consisting of several TANGIBLE OBJECT CLASSES). A collection of CONTENT can be defined, for which each item of CONTENT is defined for a specific CONTEXT and for a specific GAME VIEW. Values (or value ranges) can be set for the ATTRIBUTES.

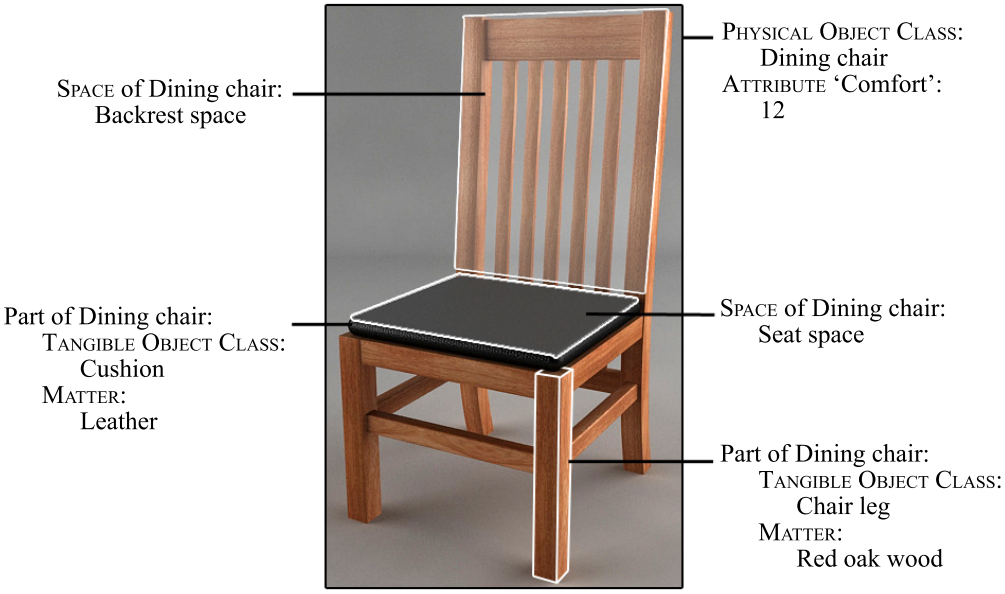


Figure 4.4: A Game-specific class for a wooden dining chair with leather seat. It inherits from Physical Object Class **Dining chair** and consists of several Tangible Object Classes and Spaces and has values specified for its Attribute.

Before a 3D model can be used in the game, its semantics needs to be specified. Entity Classes need to be assigned, Attribute values need to be filled and Spaces need to be marked. Figure 4.4 shows a Game-specific class of the Physical Object Class **Dining chair**. It consists of several parts, e.g. a Tangible Object Class **Cushion** of Matter **Leather** and a Tangible Object Class **Chair leg** of Matter **Red oak wood**. The value for Attribute **Comfort** is 12. Also some 3D shapes are defined that mark the Spaces, e.g. one for the backrest of the chair and one for the seat of the chair. 3D shapes marking Spaces can be points, lines, polygons or volumes.

Game view
: A Game view defines a particular game-specific view. Examples of these can be **3D view**, **map view**, **inventory view** or **minimap view**.

The concept of a Game view, comes from the fact that games often have different representations for the same object instance in a game. For example, a particular weapon might be represented by a dot on the minimap of the game, in the 3D view it might be represented by a complex geometric model, but when the player picks up the weapon and it disappears into the player's inventory, it might be represented in that inventory's screen with a weapon icon.

The CONTENT is not only dependent on the view, but also on the CONTEXT. For example, when an object is damaged (i.e. **state** ATTRIBUTE equals *damaged*), perhaps a different model (or icon) might be used for the same instance.

It is important to note, that the semantic representation of this object is the same throughout all CONTEXTS and GAME VIEWS.

4.7 Use of semantic information

We will now discuss how the semantic information is used in the applications that are described in chapters 5 through 8. These chapters describe procedures for layout solving, generation of buildings, procedural filters to enhance or modify objects and specifying behavior using services.

In the case of semantic layout solving, see Chapter 5, designers create descriptions with a vocabulary composed using our model. As explained previously, a GAME-SPECIFIC CLASS is created from a 3D model and can consist of SPACES. To these SPACES, 3D shapes are associated and the underlying layout solving technique will use these 3D shapes together with the RELATIONSHIPS associated to the ENTITY CLASS or its SPACES.

The layout solver uses this geometric information, so a step is needed to filter this geometric information from the semantic concepts in the descriptions. For example, a particular shape of a chair can be defined as the seating SPACE, where a pillow can be laid on by specifying a RELATIONSHIP between the seating SPACE of a chair and the TANGIBLE OBJECT CLASS **pillow**.

In Chapter 6 a similar approach is in place. In this case, several procedural content generation techniques create geometry for the game world and can register these, after associating the geometry with semantic information, to a *semantic moderator* . This process creates a semantic world representation which is shared by all procedural generation techniques. This moderator will now check all RELATIONSHIPS specified for the newly registered object with the objects that already exist in the world at the moment of registration. When the RELATIONSHIPS are associated to a geometric constraint, the geometric information is again filtered from the semantic representation and this information is used to check these constraints. If a newly registered object is not compliant to the rules specified, this is signaled to the procedural generation technique that registered it.

Procedural filters, introduced in Chapter 7, enable designers to create filters that change the look and feel of a game world. Different types of building blocks exist to perform these changes, and one of these is the semantic query block. The entire game world can be filtered for objects that match these queries, after which other building blocks can perform operations on those filtered objects.

In all these applications, a designer can use the high-level semantic information to easily and intuitively create descriptions or make queries of the game world. When solving on a geometric level is needed, the necessary information is filtered and passed along to the layout solver.

For the services in Chapter 8, the processing is a bit different. A SERVICE is a capacity of an ENTITY CLASS or MATTER to perform an ACTION. Once the object is inserted in the game world, a semantics engine continuously checks whether the conditions of its SERVICES are satisfied or not.

Once certain conditions are satisfied, the ACTIONS may be performed. So, a table will burn when exposed to fire, if the MATTER is specified as burnable above a certain temperture and there is an ACTION defined to create fire. It will not automatically create a fire knowing the property of the table. The fire of one object may have the effect that the temperature of another object increases to above the specified level, in which case the other object is also set on fire.

4.8 Discussion

Next, we will discuss how the proposed semantic model adheres to the guidelines set in Chapter 3.

We will start by discussing the design guidelines. Design guideline 1 expressed the need for the inclusion of semantics to have a low impact on the design pipeline. Our proposed semantic model favors reusability, mainly through inheritance, wherever possible. The most important one is the parent-child inheritance of ENTITY CLASSES. ATTRIBUTES will only need to be defined on the lowest common class and by using multiple inheritance, repetitive tasks become unnecessary, e.g. a **flashlight** can inherit ATTRIBUTES like **battery level** and the CONTEXT **operational** from a common class **battery powered device**, while inheriting the SERVICE **light surroundings** from another common class **light provider**.

An important downside of using multiple inheritance is the fact that it becomes impossible to prevent users of the model from creating inconsistent or ambiguous semantics. Therefore, checking the relevance and validity of the work remains the responsibility of the designer. Obviously, specification tools can help designers with this task by warning for potential inconsistencies. Semantic specifications should also allow methods to disambiguate results. In our model, we allow items, e.g. ATTRIBUTES, to have multiple names, or different items sharing names. This allows a designer to differentiate between ATTRIBUTES that were otherwise shared between two parents of the same ENTITY CLASS. Moreover, the ability of multiple inheritance is important in our views on semantic game worlds. Objects in the real world can often be regarded in different ways: a can of gasoline can be perceived as a type of fuel, but also as a potential bomb, a sofa can be used as a sitting surface, but also as a bed or as an obstacle to hide behind. It is important that designers can specify all these views on objects using semantics, even though this requires the designers to watch over the correctness and consistency of their creations.

Another point of guideline 1 was the ease of integration with existing game assets like models and audio. In the *Game content* concepts, we explained how many different types of CONTENT can be linked to underlying semantic entities and how different types of assets can be defined based on CONTEXT and GAME VIEW without changing this underlying entity. The fact that the semantic model handles all content equally, no matter what the associated file format is, makes it independent from any game engine, asset formats or target device.

Design guideline 2 demanded a wide expressive range. Our semantic model provides designers with a variety of options, e.g. one can express MATTER to the most minute detail based on the actual chemical elements of a SUBSTANCE, or instead use less defined MATERIALS or a combination of both. The limited subdivision of ENTITY CLASSES allows designers to distinguish all entities of their

game world however they like, while still providing them the necessary elements to describe their characteristics through the use of ATTRIBUTES, the definition of MATTER, the idea of SERVICES, etc.

Design guideline 3 expressed the need to further enable procedural generation techniques. Our semantic model, especially through the definition of ATTRIBUTES for MATTER and ENTITY CLASSES makes sure procedural generation techniques can be fully customized to any characteristic of an object. These ATTRIBUTES provide an effective solution to the problem of vague parameters of procedural techniques. By mapping these ATTRIBUTES to the low-level technique parameters, these techniques open up to a wide range of users that do not need to know the actual algorithms, but simply need to understand the ATTRIBUTES.

Furthermore, the addition of RELATIONSHIPS enables designers to express the structure of the world in any way that seems fit. A designer can create placement relationships to define how furniture is placed relative to each other or composition relationships can be used to define how structures need to be subdivided, e.g. to be used in *split grammars* or other procedural subdivision techniques.

Design guideline 4 asks for a reduction of the effort to design game worlds. However, the techniques to do this are mainly independent from the semantic model, we want to note that the fact that all CONTENT can be linked to semantic concepts (textures and shaders can be linked to MATTER, models to TANGIBLE OBJECT CLASSES...), makes it easier to create techniques that try to automatically assign semantics based on this CONTENT. We will further discuss this topic in the chapters about the practical implementation of this semantic model (Chapters 9 and 10).

Next, we will discuss the gameplay guidelines. Gameplay guideline 5 asked for the ability to create physically sound game worlds. For this the semantic model provides an extensive way of describing MATTER including all its characteristics through ATTRIBUTES. Linking the semantic physical and materialistic information to any physics engine will allow game worlds to be physically sound without too much effort.

Gameplay guideline 6 expressed the need for designers to approach objects from different angles. It asked for *multiple inheritance* and the ability to discern objects based on many different characteristics: functional (SERVICES), visual (CONTENT) and materialistic (MATTER). We feel that our proposed semantic model allows for a wide range of angles on any entity available in game worlds.

Gameplay guideline 7 asked for a consistent way of expressing interaction with objects. This interaction is elaborately expressible in our semantic model through the concept of SERVICES. We will discuss SERVICES in more detail in Chapter 8. This chapter will also focus largely on gameplay guideline 8 asking for a semantically consistent game world at runtime.

The final gameplay guideline 9 focused on emergent gameplay. As explained in the previous chapter, this guideline can be linked to many of the previous guidelines. Both the detailed physical representation of objects through MATTER, or a detailed method of interaction through SERVICES are excellent opportunities for designers to create emergent gameplay.

4.9 Conclusions

The semantic model discussed in this chapter is a knowledge representation model. It bears most resemblance with a frame-based knowledge representation, rather than with first-order logic representations. It is an ontology, specifically targeted towards game worlds, to represent classes and their attributes. We did not choose an existing frame-based language to implement our model, however we believe that it can serve as an ontology in many of these existing languages, some of which may bring advantages in the specification of or reasoning on game world semantics. We decided to leave that option open for future research.

In conclusion, our proposed model for SEMANTIC GAME WORLDS provides designers numerous ways of enriching any game world and any object within these game worlds with information in numerous different categories ranging from the relationships between objects to their physical representation.

The close link between these semantic model concepts and CONTENT will enable designers of any sort of game, no matter how the game world is represented (whether they are text-based adventures, 2D or 3D worlds), to semantically express that game world in design time and use the available information to improve gameplay at runtime.

In the next four chapters, we will describe some applications we developed using this semantic model. The first one is semantic layout solving. Based on descriptions created by designers using concepts from our semantic model, procedures are generated to automatically produce layouts for scenes.

CHAPTER 5

DECLARATIVE MODELING OF SCENES USING SEMANTIC LAYOUT SOLVING

In the following three chapters we will focus on the design experience of game worlds and how semantics can improve this. In this first chapter, we elaborate how semantics can improve layout solving techniques. Automatically creating layouts can help in the procedural generation of game worlds, for example, to subdivide factory lots into separate areas or to place furniture in rooms.

This chapter proposes a semantic layout solving approach that is steered by the semantics in the specification model introduced in the previous chapter. Building on top of this, we present a semantic scene description language which allows designers to easily and intuitively steer the semantic layout solver into automatically generating scenes. This language captures a generic description for a particular type of scene (e.g. bathroom, factory floor, office). This description is automatically transformed into a step-by-step procedure that feeds the semantic layout solver, allowing for the generation of an infinite number of variations on the described scene type.

Finally, by maintaining the semantic consistency after manual edits are made to scenes, we can present designers with a hybrid manual and procedural editing environment, combining the strengths of both approaches.

The proposed approach in this chapter follows the principles of *declarative modeling*. Declarative modeling aims to improve the efficiency of designers, by allowing them to express their design intent more directly and at a higher level of abstraction. In other words, it lets designers concentrate on *what* they want to create instead of on *how* they should model it [6]. Instead of modeling all scenes by hand, or use traditional procedural modeling techniques, which often use parameters and procedures that are unconnected to the object or scene one is trying to generate, the scene description language

This chapter is a summary of our previous work, published in [89], [90] and [91].

enables designers to declaratively express what scenes they want to create in terms directly connected to the target domain.

This chapter has been compiled from three of our publications, where more detailed information can be found. In particular, in [91] we describe in greater detail our whole semantic solving approach. For the implementation of this approach we used a geometric layout solving technique which is fully explained in [89]. Finally, the semantic scene description language is introduced in [90].

5.1 Semantic layout solving

The goal of our semantic layout solving approach is to automatically generate scenes, based on a designer's description thereof. In these descriptions, one can use the TANGIBLE OBJECT CLASSES and RELATIONSHIPS between classes or their SPACES, defined in the semantic model (see Chapter 4), which makes this process intuitive and accessible to a wide range of users, both technical and non-technical, since the vocabulary is directly connected to the scene that is to be generated. This, as opposed to disconnected algorithms using unrelated parameters. These descriptions are automatically transformed into procedures, which are step-by-step recipes defining which objects need to be placed in the scene. The underlying layout solving technique will place the objects based solely on geometric relationships between 3D shapes that are associated with either the entire TANGIBLE OBJECT CLASS or its SPACES. The semantic layout solver follows this procedure, one by one adding the objects to the scene based on the RELATIONSHIPS defined in the semantic library. Based on one description, as many variations as necessary can be generated all matching the intent of the designer. An overview of this approach can be seen in Figure 5.1. It is important to note that a scene itself is also an instance of a TANGIBLE OBJECT CLASS. Therefore, scenes can be placed as objects of a bigger scene, allowing the creation of hierarchic scenes.

TANGIBLE OBJECT CLASSES, and RELATIONSHIPS between them, can easily be reused in new development projects, but designers can extend on it as much as they like. It is however important to note, that to allow for new types of relationships, the translation to instructions for the underlying layout solver needs to be implemented, and if necessary the layout solver needs to be extended with new functionality. This being said, we noticed that the set of relationship types we used for all the examples in this chapter, and that are therefore readily available in our approach, was more than sufficient to express our needs.

Inspired by some of the techniques surveyed in Section 2.2.1 we developed a new rule-based layout technique [89]. This technique can calculate all possible locations for a particular object, based on its features and the already available features in the scene. We use this technique as the underlying core for the application of our proposed approach.

This section first explains the detailed workings of this technique and then describes our semantic layout solving approach and how it uses the explained technique.

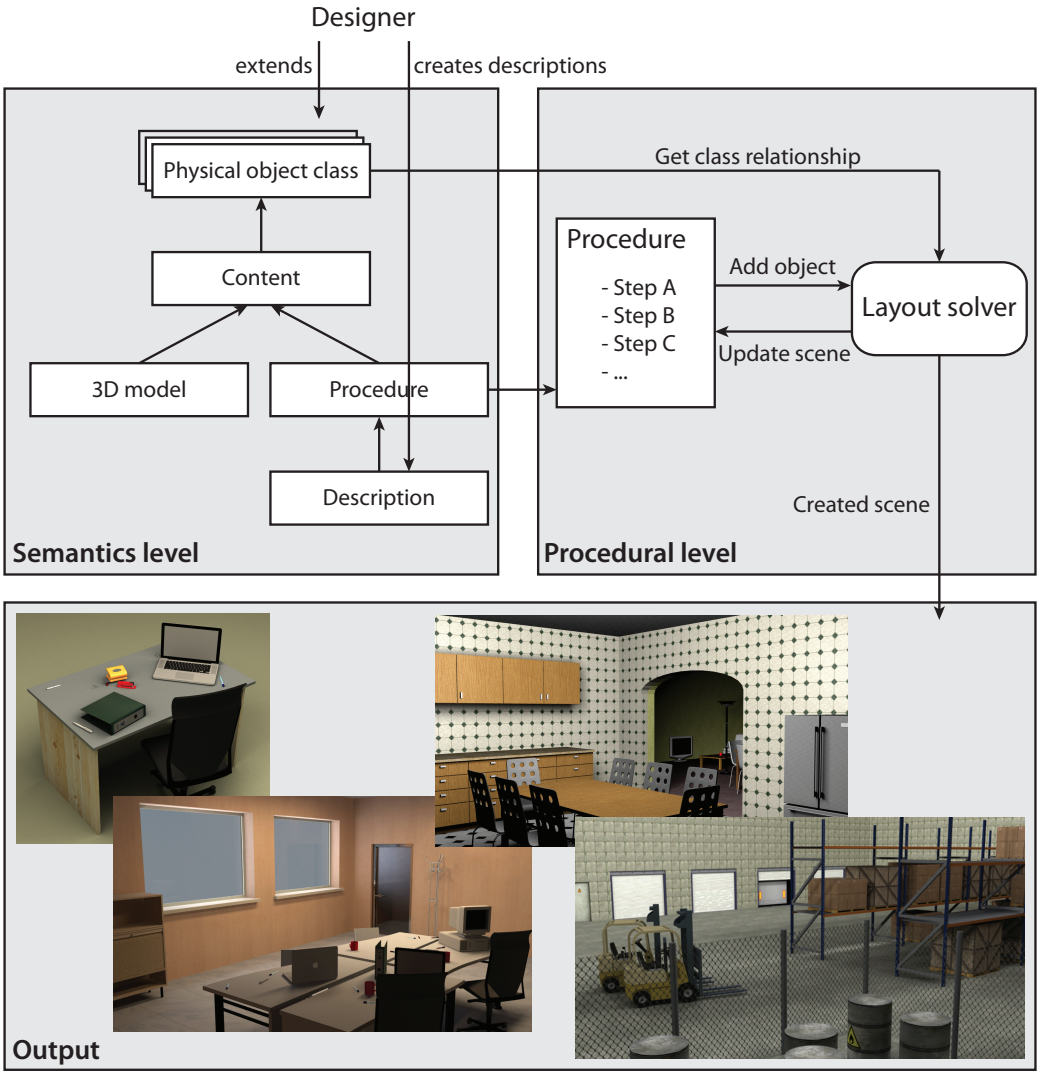


Figure 5.1: The main workflow of our semantic layout solving approach: in the *semantics level* a designer uses (or creates) TANGIBLE OBJECT CLASSES and CONTENT (e.g. 3D models) from the semantic library to create descriptions of particular types of scenes. These descriptions are transformed into procedures (see Section 5.2.3). In the *procedural level* these procedures are executed to step by step add objects to a scene based on the RELATIONSHIPS defined in the semantic library. The example output in the picture show a single desk, an office, a kitchen and a factory.

5.1.1 A rule-based layout technique

The rule-based layout technique we developed is based on two elements: *shapes* and *rules*. Shapes are basic 3-dimensional geometric representations (e.g. a box or an extruded line) and rules are geometric relationships between shapes. Shapes can also be given a special function in the solving process, either **off limits** or **clearance**. **Off limits** shapes designate regions that cannot overlap any other shape, and **clearance** shapes designate regions that need to remain empty, but that can overlap with other **clearance** shapes. The rules are geometric relationships to shapes already placed in the scene, e.g. shape X *on top of* shape Y and *next to* shape Z. When adding an object to the scene, all the shapes related to that object are placed in the scene, for example, a cupboard might have an **off limits** shape surrounding the cupboard, to make sure no other objects overlap, a **clearance** shape in front of the cupboard to designate an empty area where one can interact with the cupboard and other shapes that designate the top, left, back... of the cupboard, to be used in rules that link other objects to this cupboard.

Shapes and rules are used to find all suitable locations for a new object in a particular scene. The first step of the technique, involves finding all possible locations for a new object in the scene, based on the object's rules. For each geometric relationship type that can be used in the rules, specific methods are required to calculate all possible locations for which the relationship holds. For the **on** relationship, for example, a Minkowski subtraction is performed between the shape of the new object and the target shape on which it should be placed, resulting in a shape that encompasses all positions for which the new object is on top of the target shape. Applied to an example, a vase that needs to be placed on top of a cupboard, the shape surrounding the vase is subtracted from the *top* shape of the cupboard.

Minkowski subtraction (or Minkowski difference) is the Minkowski sum (or Minkowski addition) between two sets of points, where one of the sets of points is negative. The Minkowski sum of two sets of points is a set of the summed points from both the original sets. Formally, the Minkowski sum of two point sets A and B is defined as $A \oplus B = \{a + b : a \in A, b \in B\}$. Minkowski minus of A and B is then defined as $A \ominus B = A \oplus -B$. When these point sets are the points of two convex polygons, then the convex hull of their Minkowski sum contains each point that, used as a translation for the second polygon creates a collision between the two original shapes. In other words: $A \oplus B$ is the set of all translations for B that cause a collision between A and B. Conversely, the Minkowski subtraction $A \ominus B$ is the set of all translations for B for which polygon B is completely inside polygon A. We use both the Minkowski sum and subtraction in our layout technique.

The second step, involves removing all invalid locations from the list of possible locations calculated in the first step, based on the **off limits** and **clearance** shapes of both the new object and the scene. For each pair of object shape types and scene shape types, there are three possible cases in which overlap is not allowed and one where it is allowed, shown in Table 5.1.

For each pair of shapes for which an overlap is not allowed, the Minkowski sum between the scene shape and the object shape is calculated and subtracted from the list of all possible locations. After this step, only valid locations remain in this list.

Object shape type	Scene shape type	Overlap allowed
Off limits	Off limits	**No**
Off limits	Clearance	**No**
Clearance	Off limits	**No**
Clearance	Clearance	**Yes**

Table 5.1: Table showing which scene shape types can and cannot overlap.

This algorithm is presented in pseudo code below:

```
// Function getPossibleLocations returns all possible locations
// for newObject to be placed in the given scene
function Shape[] getPossibleLocations(newObject, scene)
{
  // ---- FIRST STEP ----
  // Creating the list of possible locations of the new object
  // based on the object's rules and the shapes already placed
  // in the given scene.

  // Before any rules are applied, the entire volume encompassing
  // the scene is deemed a valid location

  possibleLocationList = scene.boundingBoxShape

  for each objectRule in newObject.Rules
  {
    // Based on the geometric relationship type expressed in the
    // rule: the possible locations based on this rule are
    // calculated and these new locations are merged with the
    // current list of possible locations

    newLocationList = objectRule.GetPossibleLocations(scene)
    possibleLocationList = Intersect(possibleLocationList,
        newLocationList)
  }

  // ---- SECOND STEP ----
  // Now we prune this list of possible locations based on the
  // overlap rules of the shapes in the new object and the
  // already placed shapes in the current scene

  for each objectShape in newObject.Shapes
  {
    // Each shape in the current scene that cannot overlap with the
    // currently assessed shape is subtracted from the list of
    // possible locations
```

```
    for each sceneShape in scene.Shapes
    {
      if ( ! objectShape.OverlapAllowedWith(sceneShape) )
      {
        // Using the Minkowski Sum, we create an area that contains
        // all locations for which the object shape would overlap
        // with the scene shape and we subtract this area from the
        // list of possible locations

        illegalArea = MinkowksiSum(sceneShape, objectShape)
        possibleLocationList = Subtract(possibleLocationList, illegalArea
            )
      }
    }
  }
  return possibleLocationList
}
```

Now the object can be placed on one of these suitable locations: either a randomly chosen position (for automatic layout solving) or one picked by the user (in an assisted manual editing environment). This way a scene consisting of multiple objects can be filled one object at a time.

Performance is always an important issue for any solving approach. However, a generic approach will not be able to take advantage of many optimizations available for more specialized solving methods. In the algorithm above, it becomes clear that the number of shapes available in the scene could create an important bottleneck. Every time an object is added to the scene, all related shapes are added and the pruning of possible object locations based on the scene shapes will take longer and longer. In the worst-case, i.e. when every shape cannot overlap with every other shape in the scene, the complexity is near O(n*m) with n the number of shapes in the new object that need to be added to a layout and m the number of shapes available in the scene.

5.1.2 Using semantics to steer layout solving

In our semantic layout solving approach, we use the layout technique described above and the semantic model introduced in Chapter 4 to accommodate an easy and intuitive way to assist designers in creating layouts. Since the approach can generate all suitable locations for a particular object, it can be used both in a manual design environment and in a fully automatic approach:

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

2. In an automatic application, the objects need to be added to the scene by means of a procedure. Based on such a procedure, objects are step by step provided to the solver, which in turn

generates a new valid layout. The planner will be discussed in more detail in the next section.

As explained in Chapter 4, in our semantic library a TANGIBLE OBJECT CLASS contains RELATIONSHIPS with other TANGIBLE OBJECT CLASSES. Each TANGIBLE OBJECT CLASS also consists of a number of SPACES. As defined in Chapter 4, a SPACE is a region bounded by a 3D shape. The shape of a TANGIBLE OBJECT instance is its bounding box. When a placement RELATIONSHIP exists between two TANGIBLE OBJECT CLASSES, a geometric relationship is generated between the shapes of their respective TANGIBLE OBJECT instances. This way, the layout technique discussed in the previous section can be used to find all suitable locations for any TANGIBLE OBJECT instance in a particular scene.

For example, let us assume we have three TANGIBLE OBJECT CLASSES defined: **Floor**, **Wall**, and **Cupboard**. Each of these TANGIBLE OBJECT CLASSES has some SPACES defined like **Bottom**, **Left side** or **Top**. Assume also that there are two RELATIONSHIPS defined:

- a RELATIONSHIP of type **On** between **Cupboard** and **Floor**, and
- a RELATIONSHIP of type **Against** between the **Back** SPACE of **Cupboard** and **Wall**.

When an instance of **Cupboard** needs to be added to the scene, geometric relationships of type **On** are automatically added between the bounding box shape of the **Cupboard** instance and the shapes of all **Floor** instances in the scene and geometric relationships of type **Against** between the shape of the instance of SPACE **Back** of the **Cupboard** instance and the shapes of all **Wall** instances in the scene. Based on these relationships, the layout solving technique can find all suitable locations for this **Cupboard** instance. After choosing a random location among these suitable locations, the **Cupboard** instance and its related shapes are added to the scene.

5.1.3 Semantic layout solving using constraint solvers

In our implementation of the semantic layout solving approach, we used the layout solving technique explained above. However, the semantic layout solving approach can work independent of the used solver.

The important factor here is the fact that the types of placement RELATIONSHIPS in the semantic model need to be translated into low-level instructions for the solver. When using a geometric constraint solver, the **Against** relationship, e.g. “back against wall”, can be expressed as constraints on one or more points of the source SPACE (in the example the *back* of the cupboard), to be *on* the target SPACE (in the example the wall). Additionally, distance constraints on those points need to preserve the shape of the source SPACE. RELATIONSHIPS dealing with orientation, e.g. *sofa facing TV*, can be expressed using angle constraints on the lines of the source object. These types of constraints are readily available in many geometric constraint solving systems.

When someone would want to use a different constraint solver, the conversions between the types of semantic placement RELATIONSHIPS and the related constraints need to be rewritten. All

other elements, such as defined RELATIONSHIPS and TANGIBLE OBJECT CLASSES or specified descriptions for layouts can be maintained without a problem. When more functionalities of a particular constraint solver are wanted by the user, the semantic library does need to be extended with new RELATIONSHIPS types to reflect those functionalities.

5.2 Automatic scene generation

We can use the semantic layout solving approach, introduced in the previous section, to automatically generate scenes, e.g. room layouts, factory floors or office buildings. We use so-called *procedures* to guide the layout solving in adding objects to a scene, thereby creating complete scenes automatically.

To create such procedures, we developed a semantic description language that allows designers to easily describe what particular scene classes (e.g. kitchens, factories, parks) consist of. This description language is context sensitive and uses the vocabulary from the semantic model to enable designers to quickly create generic descriptions for a particular scene type in an intuitive way.

This section first discusses the concept of procedures and how they are used to automatically generate scenes. After that, the semantic description language is introduced and finally the conversion process from descriptions to procedures is described.

5.2.1 Layout planner and procedures

To use the 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 finds a valid position and orientation. We mentioned that the solver finds all suitable locations for a new object in a scene, and when used in combination with the planner, a certain position is selected.

The planner works based on a procedure: a list of statements and rules that need to be executed in order. The types of statements available in these procedures are:

Pick statements are queries to find suitable GAME-SPECIFIC CLASSES in the semantic library. Queries can contain TANGIBLE OBJECT CLASSES the GAME-SPECIFIC CLASSES need to inherit from, conditions on the ATTRIBUTE values, SERVICES they need to provide, MATTER they should consist of or PREDICATES they should contain.

Place statements trigger the placement of an instance of the last picked GAME-SPECIFIC CLASS. Additional scene-specific RELATIONSHIPS can be added to these statements.

Conditional and loop statements are similar to if-then-else statements and for- or while-loops available in programming or scripting languages.

Backtrack statements force the algorithm to backtrack a given number of statements. These can be used to redo part of the placement, if at one point no suitable locations can be found for a particular object.

Execute procedure statements can trigger sub-procedures to allow for hierarchically built up scenes. A building lot, for example, 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-procedures. The designer can make a procedure to layout rooms in a house and another procedure to layout objects in a room, for example.

Next is an example procedure for a simple kitchen scene:

1. Pick a TANGIBLE OBJECT CLASS **Refrigerator**
2. Place instance in scene
3. Repeat until sum of **Storage volume** ATTRIBUTES exceeds **1.3 cubic meters**
    a) Pick a TANGIBLE OBJECT CLASS **Kitchen cabinet**
    b) Place instance in scene
4. Pick a TANGIBLE OBJECT CLASS **Table**, made of MATTER **Wood**
5. Place instance in scene
6. If previous statement failed
    a) Backtrack 2 statements
7. Pick a TANGIBLE OBJECT CLASS **Chair**
8. Repeat **6** times
    a) Place instance in scene
9. Pick a TANGIBLE OBJECT CLASS **Appliance**, providing the SERVICE **Cook food**
10. Place instance in scene, on top of TANGIBLE OBJECT CLASS **Kitchen cabinet**

In this example, first a refrigerator is placed in the scene. After that kitchen cabinets are added until the total Storage volume exceeds 1.3 cubic meters. Now a table made of wood is added to the scene. A backtrack statement is added to ensure that this table can be placed. If not, the procedure will backtrack to the pick statement for the table, thereby picking a different table. This is repeated until a table is picked that fits the scene. Now a chair is picked and an instance thereof is placed 6 times. Finally an appliance that can cook food is placed, adhering to the scene-specific relationship that it should be placed on top of a kitchen cabinet.

5.2.2 A semantic scene description language

We propose a semantic scene description language that is a visual language aimed at allowing a designer to define the different elements of which scenes of a particular type consist.

The main goals of this language are:

1. describing which objects or components can or should be present in a given scene;
2. describing the relationships between the available objects; and
3. discerning scene 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 convert descriptions into procedures, the latter can then be used to automatically generate various instances of the described scene. That conversion process will be dealt with in the next subsection after we discussed the main features of the language.

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. Therefore each description entity consists of two components:

Object component: a detailed description of which types of objects need to added, e.g. which TANGIBLE OBJECT CLASSES they should inherit from or the MATTER they consist of. This also includes the amount of objects, which can be a fixed number or a distribution (X per square meter of scene area). One entity can group multiple types of objects that need to behave similarly in the described scene.

Placement component: a description on how the objects should be placed by defining scene-specific RELATIONSHIPS the objects described in the object component need to adhere to when being placed. These RELATIONSHIPS are either additional to the RELATIONSHIPS defined in the semantic library or they can override the ones in the library.

These two components are analogous to the *Pick* and *Place* statements from the procedures (see previous subsection).

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 line it follows. The descriptions allow selecting 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 two meters apart and place fence poles on the points and barbed wire between them, to generate a fenced area.

A collection of description entities makes up a *semantic scene description*, i.e. a generic, high-level definition for 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 spaceship interior, a forest or an industrial zone.

To discern more specific cases in a scene class, we use the notion of CONTEXT, described in Section 4.5. We described a CONTEXT as a "collection of preconditions that describe a particular situation". Designers can define context-specific behavior by changing the basic description for each particular CONTEXT. The possible preconditions for CONTEXTS can be (but are not limited to) (i) conditions on the scene or on the ATTRIBUTES of the scene (since a scene by itself is also an instance of a TANGIBLE OBJECT CLASS, as explained in the previous section), (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 **Residential house** scene, we might want to create, e.g. the CONTEXTS **Apartment**, **Small**, **Large** and **Villa**. In the CONTEXT **Large**, we will want to place more bedrooms and bathrooms, we need a larger dining room, perhaps two garages instead of one, etc. Once the conditions for which the CONTEXT holds are defined, the designer can adjust the description to fit that CONTEXT. This is done by marking changes to the default description. Depending on the CONTEXT, entities specified in the description can be altered in any way: e.g. adding RELATIONSHIPS, changing the number of instances needed or adding or removing PREDICATES in the description entity. Entities can also be completely removed or new entities can be added.

Since the scene described in a semantic description, in itself defines a new TANGIBLE OBJECT CLASS, descriptions can follow a hierarchic scene composition, e.g. we can create a description for an **Office building**, which specifies entities of the TANGIBLE OBJECT CLASS **Office room**, which in turn specifies entities 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, making the entire approach much 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; in keeping with the declarative modeling principle, designers can express what they want the scene to look like, instead of how the scene should be built or in what order the different objects are added.

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. In addition, the ability to create scenes constrained by a base shape also fits very well in their normal working methods.

5.2.3 Converting a description to a procedure

After we have created a semantic scene description, we need to convert it to a procedure, which are suitable to steer our layout solving approach. This involves three steps: (i) sort the description entities and add them to the procedure, (ii) encompass these procedure statements with loops to handle amounts defined in the entities and with conditional statements to handle context-specific operations, and (iii) perform an optimization (e.g. add some additional checks) to ease the workload of the solver when executing the procedure.

The *first step*, sorting the description entities, is the most complex one since many criteria need to be taken into account. In general terms, we need to make sure that all objects can be placed, complying with the dependencies that are present because of the RELATIONSHIPS between TANGIBLE OBJECT CLASSES. The output of this step will be an ordered list of the description entities. To accomplish this, first a dependency graph is created, in which the TANGIBLE OBJECT CLASSES of the entities are represented as the nodes, and every placement RELATIONSHIP between two TANGIBLE OBJECT CLASSES as a directed edge: outgoing edges point to TANGIBLE OBJECT CLASSES on which a node's placement will depend. Step by step the ordered list is filled. In every iteration, one entity is added to the ordered list and it is flagged as 'ordered', while entities that are not yet added to the ordered list are referred to as 'unordered'. Each iteration, the following four criteria are taken into account to choose the next entity to be added to the ordered list:

First criterion Least outgoing edges to nodes of unordered entities: when an entity depends on an ordered entity, there will be no problem in the generation phase (i.e. when a scene is being generated based on the created procedure) since the entity on which it depends will already be placed in the scene. Therefore, we only take edges to unordered entities into account in this first criterion. In case of multiple nodes with the minimum amount of edges to unordered entities, the second criterion is considered.

Second criterion Most incoming edges: we pick the object with the most incoming edges, since most other objects depend on its placement, therefore making the ordering in the next iterations easier. Again, in case of a tie, the next criterion is considered.

Third criterion Most outgoing edges to nodes of ordered entities: we pick the entity with the most outgoing edges to ordered entities, because the more dependencies an entity has, the more restricted the choice for possible locations is. In case of a tie, the fourth criterion is used.

Fourth criterion Largest object: when entities are not discernable based on the previous three criterions, we use the size of the entity's objects as the final decider. Since a description and a procedure work with object *queries*, and the actual objects are picked only in the generation phase, we do not have an exact size for each object at the conversion phase. Therefore the average size of all possible objects that match the entity's object description is used as an estimate.

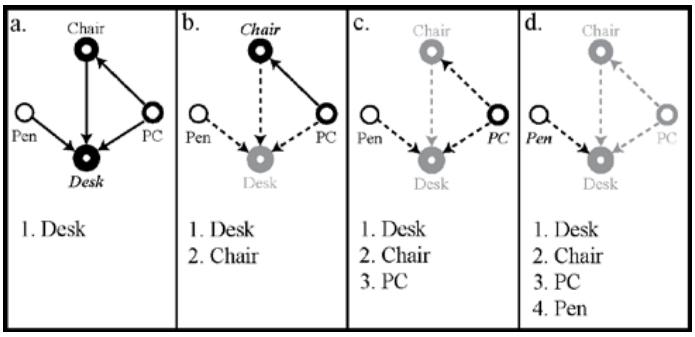


Figure 5.2: Creating an order for objects based on the dependency graph.

To clarify this process, it will be applied to the following 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. We start with an empty list and a starting graph as depicted in Figure 5.2a.

Iteration 1 The desk is the only independent entity, i.e. an entity without any outgoing edges. Therefore, based on the first criterion, the desk is added to the ordered list in the first iteration and is therefore flagged as 'ordered'.

Iteration 2 Both the chair and the pen now only have outgoing edges to ordered entities (see Figure 5.2b). Therefore, we need to decide based on the second criterion. Since the chair has one incoming edge and the pen has none, the chair is added to the list in the second iteration.

Iteration 3 In Figure 5.2c, both the PC and the pen have no outgoing edges to unordered entities and no incoming edges, so we move on to the third criterion. The PC has the most outgoing edges to ordered entities, so it is added to the list.

Iteration 4 Since there is only one unordered entity left, which is the pen, it is added to the list immediately.

In the sorting process, we could run into conflicts if 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 (i) pick one of the entities that create the circular dependency based on the above criteria, and (ii) remove the relationship connected to the outgoing edge of the picked object that creates the circular dependency. 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 (as explained in Section 5.2.1) for every object in the list.

The second step of the conversion is relatively straight forward: it adds control statements where necessary. 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.

The above two steps result in a procedure that adequately reflects the scene description. However, adding some additional rules will improve the solving process for that procedure, which is the third step of the conversion. 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 some additional procedure statements, based on the available size in the scene, are added to handle these situations appropriately: when trying to fill a small kitchen, we do not 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.

5.3 Results

As an example of our semantics-based layout solving approach, we discuss a living room procedure. This procedure is a list of fifteen statements, mainly to add a number of instances of a particular TANGIBLE OBJECT CLASS, with some constraints specified on its ATTRIBUTES, such as a table with a 50 cm maximum height for the coffee table, minimum comfort level for the seats, etc. Figure 5.3 shows an example of a living room based on this procedure and automatically laid out with this approach. On the left of Figure 5.3 is the 2D floor plan of the house (also generated using semantic layout solving, similar to the second example in this section) with the living room, showing also the **clearance** and **off limits** shapes (used by the solving technique), e.g. in front of the sofas, behind the chairs and on both sides of doorways.

Because of the integration with the semantic model, our solving approach is generally usable for many different layout problems. To apply the solver to a different style of scene, one only needs to add the necessary TANGIBLE OBJECT CLASSES to the library (should they not be available yet). As a second example, the solver was used for the automatic generation of a building floor plan (Figure 5.4). For this, some different room types were added to the semantic library, containing RELATIONSHIPS about e.g. the neighboring rooms, and some specific RELATIONSHIPS, 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. After this is generated, we apply two post-processing steps, which are not included in the semantic layout solving approach and are specifically developed for generating building floor plans. A first post-processing step is applied that grows the rooms to fit the building shape, as can be seen in Figure 5.4. A second step is used to generate a basic *connection* between doors within a room. This connection was then used to create a virtual corridor of **clearance** SPACES to make sure there remained a clear passage between these doors.

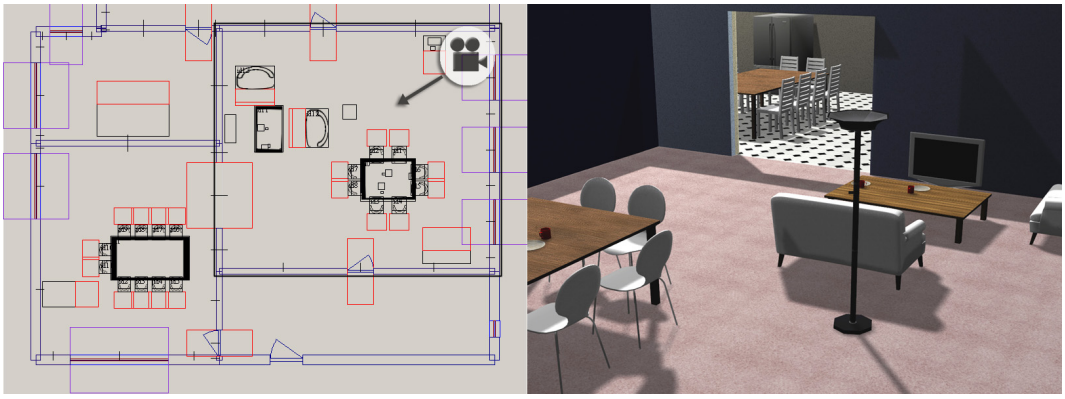


Figure 5.3: Automated generation: (left) 2D floor plan of a house created with our solving approach; (right) 3D visualization of the living room (from viewpoint indicated by camera in the left image).

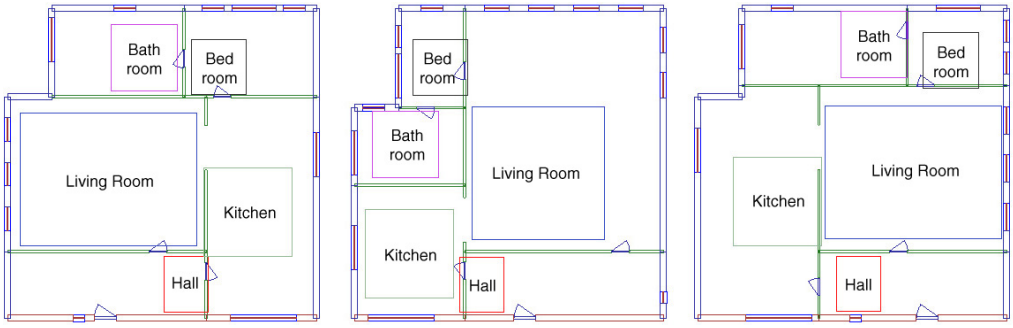


Figure 5.4: Three floor plans generated with the semantic layout solving approach. First areas for the rooms (see colored rectangles with room names) are placed within an empty building, with the minimum dimensions for each particular room and according to the RELATIONSHIPS defined between them. Afterwards these areas are grown to create the actual rooms that fit the building shape.

Figure 5.5: Some example scenes built with descriptions (top: factory floor, center: office, bottom: a road through a forest).

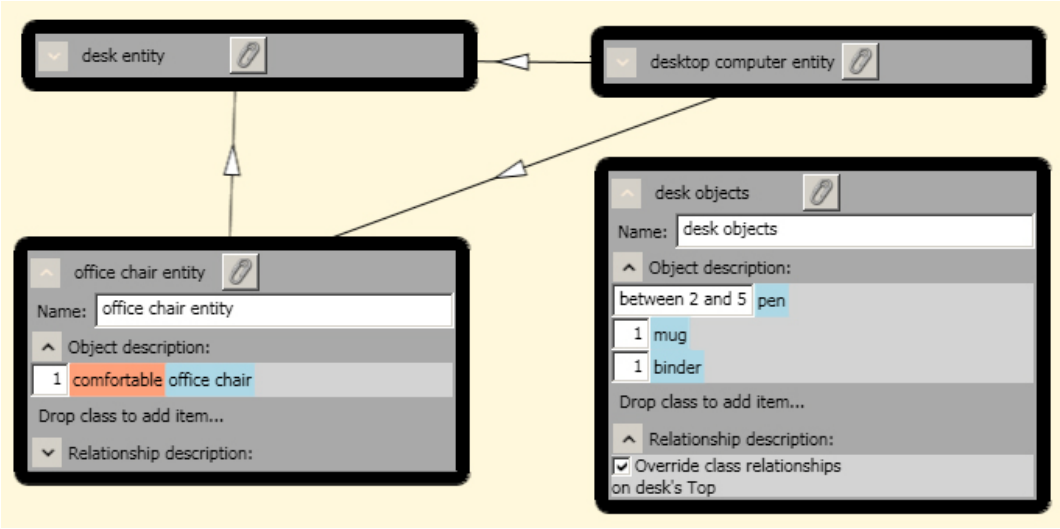


Figure 5.6: A description for a single desk setup with four entities. Arrows show the common relationships between entities present in the semantic library. Since the 'desk objects' entity overrides the class RELATIONSHIPS, no edge to the desk is shown.

The scenes in Figure 5.5 were created based on descriptions created with the semantic description language: a factory floor, an office and a road running through a forest.

The factory (Figure 5.5 top) consists 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 Section 5.2.2. For example, the area shape for the pallet racks is split up in a number of rows, and on each row an empty pallet rack is created that contains **pallet storage** SPACES. The factory description contains an entity for some pallets to be placed inside these pallet storage SPACES.

The road geometry (Figure 5.5 bottom) is created using a system, unrelated to the semantic layout solving approach. It functions based on a path and profile: the path is considered to be the center of the road, which is offsetted to generate the different elements described in the profile (e.g. 1m sidewalk, 3m lane, a second 3m lane, 1m sidewalk, 0.5m grassy bank). We extended this road geometry system to place SPACES such as **roadside** or **bicycle path** based on the profile. For example, in the bottom example of Figure 5.5 a lamppost is placed every 20 meters on the **roadside** SPACE, 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 5.5 center) from the user perspective. 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 defined. A fourth entity groups multiple objects, all with similar behavior: some pens, a mug and a binder. The main RELATIONSHIPS for these TANGIBLE OBJECT

CLASSES are overridden and an additional RELATIONSHIP defines that they should be placed on top of the desk. This description, which is shown in Figure 5.6, is now ready to be exported to a procedure.

Once the TANGIBLE OBJECT CLASS **desk setup** is defined, with its associated procedure, some RELATIONSHIPS are defined: the desk setups should be preferably placed next to other desks or opposed to other desk setups. In the office description, this TANGIBLE OBJECT 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 closets and a coat rack are included. Again, the RELATIONSHIPS for placing these last two TANGIBLE OBJECT CLASSES are already defined. This example demonstrates that using our semantic description language, in combination with our layout solving approach, one can create a description for any office space in a matter of minutes. And since one description can be used to generate countless variations, a whole office building can be filled with office spaces without copying the same space over and over again (as is often the case in many game worlds). Moreover, context-sensitive changes to the description will make it even more powerful and usable under different circumstances. In the office example, we can quickly adjust the description for the CONTEXT in which the owner of the office is the boss: e.g. changing the amount of desk setups to 1, adding the **antique** predicate to the furniture entities and adding an entity defining two seats in front of the desk (for visitors). This context-sensitive approach therefore allows many opportunities for reusability, again relieving some of the efforts involved in creating game worlds.

5.4 A hybrid approach: combining manual and automatic modeling

We already explained that the layout solving technique (described in Section 5.1.1) is specifically developed with both manual and automatic modeling in mind: the technique can compute *all* valid, suitable locations for a particular object.

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, we can use the description (and corresponding procedure) for an office desk setup, explained in Section 5.3, as a building block to be used by designers in a level editor. A designer can drag and drop an abstract desk 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, one gets unique blocks every time you drop one. This is somewhat comparable to the nowadays common function of terrain editors, with which the designers can brush a region that is automatically filled with randomly placed trees and shrubs. In other words, the designer maintains full control over the end result, but some common, boring tasks, e.g. laying pens and other items on every desk in the entire game world, is automated.

A hybrid manual and automatic modeling environment includes other features as well. The environment can present users with multiple suggestions for a particular scene. The user chooses one of these suggestions and continues from that suggestion.

The user can lock certain regions and ask to regenerate the remainder of the scene. In our layout solving approach, this can be handled by removing all objects outside the locked region from the

scene and try to add them again to the scene based on their previously used RELATIONSHIPS.

Obviously every automatically positioned object should also be manually moveable as well. Applying manual moves to automatically placed objects might however break up the semantic consistency in a scene. For example, objects placed on top of another object should be moved together with it. In general terms, if an object is automatically placed with a dependency to another object, it should be repositioned when this other object is moved.

A consequence of the above is that it is crucial to maintain the semantic consistency after manual edits. Upon a basic transformation (translation or rotation) of an object, we perform the following algorithm: (i) find all objects directly or indirectly dependent on the transformed object, (ii) perform the same relative transformation on all dependent objects, (iii) remove any objects that are no longer on valid locations, (iv) add these removed objects to the scene again using the layout solver.

This algorithm is presented in pseudo code below:

```
// The sub-function getDependentObjects returns all objects
// dependent on the given object (i.e. a relationship exists
// between this object and the given object)
function Object[] getDependentObjects(object)
{
  Object[] dependentObjects = []

  // Loop through all established placement relationships of
  // the given object
  for each relationship in object.PlacementRelationships
  {
    if relationship.Source == object
      dependentObjects.add ( relationship.Target )
    else
      dependentObjects.add ( relationship.Source )
  }

  return dependentObjects
}

// Function maintainSceneConsistency maintains the consistency
// of a given scene upon a transformation of a given object
function Object[] getPossibleLocations(scene, object,
    relativeTransformation)
{
  // -- FIRST STEP --
  // Find all objects dependent on the given object
  Object[] dependencies = getDependentObjects ( object )

  int nrOfDependencies = 0
```

```
// As long as the list of dependencies grew in the previous
// iteration, keep getting the dependent objects of the newly
// added objects
while ( nrOfDependencies < dependencies.length )
{
  Object[] newDependentObjects = []
  for ( int i = nrOfDependencies; i < dependencies.length; ++i )
  {
    // Merge the list of dependent objects of the currently
    // assessed one with the list of new dependent objects
    newDependentObjects = Union ( newDependentObjects,
        getDependentObjects ( dependencies[i] ) )
  }

  nrOfDependencies = dependencies.length

  // Merge the list of new dependent objects with entire list
  dependencies = Union ( dependencies, newDependentObjects )
}

// The list 'dependencies' now contains all objects that are
// directly or indirectly dependent on the given object

// --- SECOND STEP ---
// The relative transformation is executed on all dependent
// objects
for each ( Object o in dependencies )
{
  o.ExecuteTransformation ( relativeTransformation )
}

// --- THIRD STEP ---
// Check if all objects are still in valid positions, otherwise
// the object is removed
Object[] removed = []

for each ( Object o in dependencies )
{
  if ( ! o.relationshipsStillHold () || scene.
      overlapsOffLimitsOrClearanceShapes ( o ) )
  {
    scene.remove ( o )
    removed.add ( o )
  }
}
```

```
  // ---- FOURTH STEP ----
  // Add the removed objects to the scene again
  for each ( Object o in removed )
  {
    // First try to add maintaining existing relationships.
    // This includes using the same target objects, e.g. a computer
    // which was placed on a desk, will try to be placed on that
    // same desk again.
    if ( ! scene.addObjectAccordingToExistingRelationships ( o ) )
    {
      // If this fails, add the object in a normal way, i.e.
      // according to defined relationships in the semantic library,
      // but not regarding previous target objects. In the previous
      // example: the computer will be placed on a desk, but not
      // necessarily the original desk
      scene.placeObjectAccordingToLibraryRelationships ( o )
    }
  }

}
```

An issue we run into, when running this algorithm is that when objects are dependent on the objects that were removed, the semantic consistency might be broken again for these objects. To handle this, this entire procedure is repeated for each object in list "removed" (and recursively for the removed objects in the new iteration of the algorithm). If a circular dependency is found, i.e. if an object is added to the list "dependencies", that was already an element of any of the "removed" lists, then we immediately remove it from the scene and just add it based on the RELATIONSHIPS, without maintaining the object dependencies. This way all objects still end up on valid and suitable locations in the scene.

5.5 Conclusions

This chapter introduced the notion of semantic layout solving that effectively applies our semantic model for game worlds to automatically generate scene layouts. Our semantic layout solving approach consists of two main components:

- A *scene description* which defines *what* objects need or can be available in a particular type of scene,
- RELATIONSHIPS to define *how* objects need to be placed relative to each other.

Moreover, the generally known concepts from the semantic model, make extending and reusing both descriptions and RELATIONSHIPS quite easy. By using SPACE instances of objects in the

RELATIONSHIPS, one can specify the placement of objects relative to a particular *feature* of another object, e.g. "an office chair should be placed facing the front of the computer monitor". The concept of CONTEXTS can assist in creating variations and customizations on the layout descriptions to adapt the results to specific characteristics of the environment (e.g. the size and accommodations of a building, or the personality of the inhabitants).

An important feature of semantic layout solving is that it does not require using our rule-based layout technique. It can be integrated with any technique that can output all suitable locations, based on a number of one-to-one geometric constraints. Therefore, employing optimized layout solving techniques can increase the performance of our approach, making it possible to apply the approach for real-time generation. Currently, using our non-optimized layout solving technique, the approach requires between 2 and 5 seconds to generate a relatively complex scene of about 20 objects, each with 2 to 4 defined RELATIONSHIPS.

We can conclude that our semantic layout solving approach is suitable for generating scene layouts both automatically and semi-automatically in a hybrid approach. Its integration with the semantic model makes specifying layouts intuitive and easily extendable. Moreover, the use of CONTEXTS makes descriptions easily adaptable to specific circumstances. In tune with the concept of declarative modeling, we allow designers to focus more on what they need, i.e. describe what objects they want in the scene and how they should relate to one another, instead of having to focus on how to model or build the world.

The semantic layout solving approach is one of many approaches and techniques to automatically generate parts of game worlds. However, the integration of these different techniques can be quite hard, especially when different techniques are responsible for overlapping structures. In the next chapter, we will describe the use of a semantic moderator to integrate several procedural generation techniques to create consistent buildings.

CHAPTER 6

GENERATING CONSISTENT BUILDINGS USING SEMANTICS

In Chapter 5 we discussed an approach to procedurally generate scene layouts based on intuitive descriptions using the vocabulary from our semantic model. Next, we will discuss a method to create complex structures by integrating multiple procedural techniques that generate nested components.

Several procedural techniques exist to generate very specific elements of a virtual world. However it is sometimes difficult to blend different techniques into a consistent whole. For example, several techniques can create a specific component of a building like the façade, the floor plan or the room layout (e.g. using semantic layout solving explained in Chapter 5). Still, it is impossible to simply combine the outputs of these different techniques, since they are not tuned to each other. For example, the floor plan generation techniques might add an inner wall cutting right through the middle of a window placed by the façade generation technique, or the latter might place a window in the wall of a room that the layout generation technique has marked as a storage room.

In general terms, separate procedural content generation techniques have no knowledge about elements created by other techniques. Furthermore, there is no knowledge on the rules and constraints that exist between the different elements that make up a game world. Therefore we propose to use semantics to *moderate* between multiple procedural generation techniques and to detect conflicts based on a set of semantic constraints.

Just like a semantic model can be the glue that binds different components of a game engine, it can also serve as the glue to combine multiple procedural generation techniques that generate overlapping components of a virtual world. The semantic model provides a vocabulary in which multiple techniques can communicate in a similar way about the elements of the game world they are

This chapter is a summary of our previous work, published in [92].

generating and based on rules and constraints in the model, conflicts can easily be found and flagged to the separate techniques in order to handle them. Not only can semantics serve as a language to construct a consistent representation of the combined model, it can also aid in resolving conflicts that may arise when multiple generation techniques want to create new entities conflicting with ones already present in the model.

This chapter introduces a semantics-based integration approach between procedural generation techniques. This approach was implemented using our semantic model and applied to several techniques that generate nested elements of buildings, resulting in complete and consistent buildings, demonstrated in an example of a North American style villa.

6.1 Semantic integration approach

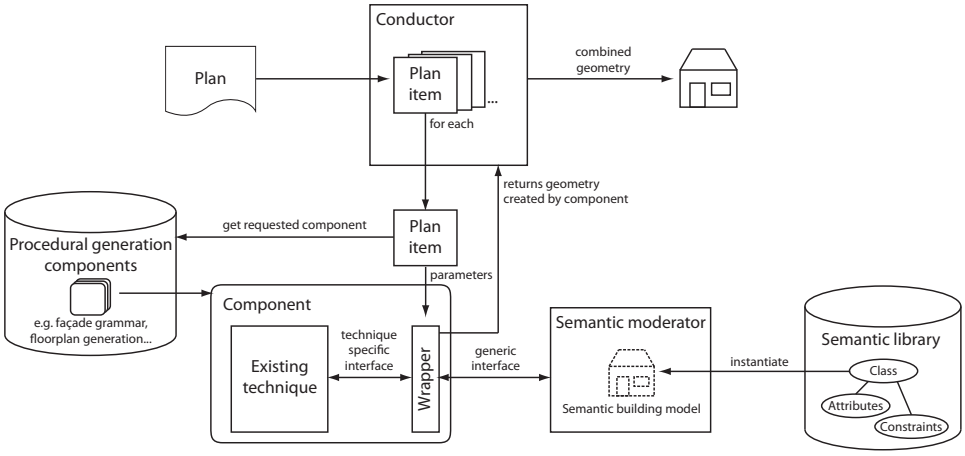


Figure 6.1: Framework for integrating procedural techniques: *moderator* (with *semantic library* and *generic interface*), *components*, *wrappers*, *conductor* and *plan*

Typically, each procedural technique is able to generate one specific architectural element of a building (e.g. façade, floor plan, furniture, lot shape), but mostly without much regard for other building elements. Therefore, the main challenge of integrating those individual components is foremost to watch over the consistency of the building, either avoiding or properly handling any conflicts arising among building elements.

The main idea behind our approach is to establish a semantic moderator, which shares relevant building information with the individual procedural components, so that they can make good and timely decisions. This information, combined by the moderator into a unified *semantic representation*

of the building, forms the basis for the advice that it provides to individual components in order to avoid conflicts, *i.e.* inconsistent results. In our approach, we distinguish three categories of building elements conflicts:

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

Figure 6.1 outlines the framework architecture to support this integration approach. The various procedural components are made available through a wrapper interface and are invoked according to a building plan. The moderator, in turn, helps prevent the conflict types mentioned above, managing the communication with the procedural components, and providing them with building advice. In the following paragraphs, we discuss this framework in detail.

6.1.1 Semantic moderator

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

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

Register a building element

A procedural component can *register* a new building element with the moderator. This can either *approve* the registration, meaning that the new building element is deemed valid for integration in

the building representation, or *reject* it, meaning that the new building element causes a conflict that cannot be handled in any other way. In the latter case, the component should retract its conflicting element. For each registered building element, a corresponding semantic element is instantiated and inserted in the semantic building representation, possibly with specific values for some of the class attributes; for example, a window instance could have a Boolean attribute value indicating whether or not the window glass is tinted.

Register a constraint

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

Inquire about a building element

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

Inquiries can also be used to find out whether a *potential* building element could be successfully registered, i.e. approved as valid by the moderator. Such an inquiry does not imply registration, or even creation, of elements, and it can be generically defined as follows: can an instance of class c, with attribute values $a_1 \dots a_n$, with shape s be placed at position p and orientation o? In order to answer such inquiries, the moderator first gathers all constraints mentioning class c and, for each of them, evaluates whether they are satisfied for shape s at position p and orientation o. For example, say we want to inquire whether we can place a non-tinted window of shape s at position p with orientation o, i.e. *inquire(window{tinted:false}, s, p, o)*. The example constraint defined above references a window class with attribute *tinted* equal to false. Therefore, the moderator checks whether shape s, with the given position and orientation, is adjacent to the shape of any private room. If so, that *non_adjacency* constraint is not satisfied, and, therefore, the building advice is negative. This same constraint evaluation mechanism is used to evaluate the previously described inquiries, e.g. to inquire which type of room is adjacent to a particular wall.

The methods to evaluate these constraints were initially built for the semantic layout solving approach (see Chapter 5). In the semantic moderator these methods are used to identify whether or not a building element at a given position in the scene is placed according to its related constraints, as is explained above. If the position for the building element conflicts with the related constraints, then

a negative building advice is given, which should be handled by the component, e.g. by retracting the element.

These constraints are represented using the RELATIONSHIP concept in our semantic model: source feature (i.e. a PHYSICAL OBJECT CLASS) - the relationship type (i.e. RELATIONSHIP) - target feature (also a PHYSICAL OBJECT CLASS), with a number of ATTRIBUTES (depending on the RELATIONSHIP). For example: PHYSICAL OBJECT CLASS *vase* - RELATIONSHIP *on* - *top* SPACE of PHYSICAL OBJECT CLASS *cupboard*, represents that a vase should be placed on the top of a cupboard. These declared constraints can be mapped in a straightforward way to the actual constraints used by the layout solving methods (as explained in Chapter 5).

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

Select valid positions for a building element

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

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

6.1.2 Wrapping components

The integration of existing procedural components within the same framework has attractive advantages, since it allows the generation of more complex structures (e.g. a complete house with interior and exterior instead of simply a façade). Even when techniques exist to completely create such complex structures, allowing the integration of more atomic techniques results in a lot more flexibility: the most suitable technique can be chosen for every individual element of the game world. The counterpart of this integration, of course, is that there is some implementation effort involved.

We now describe the implementation steps required and the impact of the integration process on each procedural component.

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

The main purpose for a component wrapper is to provide access to the functionality of the moderator using a generic interface, as shown in Figure 6.1. Such a wrapper only needs to be implemented once for each procedural component, regardless of the number of other components or the type of building being generated.

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

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

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

Finally, it should be mentioned that minor alterations will need to be made directly in the component's procedural generation method. At least, the wrapper methods need to be invoked throughout it at the correct time. An example is the registration of elements with the moderator before they are definitively placed. Still, the implementation of the wrapper interface is the most important step required for the successful integration of a new procedural component. After a component's wrapper is implemented in the correct way and the mentioned minor alterations to the procedure have been performed, that component becomes and remains integrated within our framework. All its functionality, including notification events, remains intact regardless of changes to, and replacements of, other components.

6.1.3 Plan and conductor

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

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

Currently, building plans are specified using a declarative scripting language developed for this framework. Among other things, this language provides commands for declaring the components used in the plan, and invoking them in a desired order. The invocation of a component, declared using the *execute* command with the respective parameters, is supported through a call to its wrapper. An example of the syntax of this language is shown in [92].

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

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

Plans are also responsible for another mechanism: sharing and passing building elements from one component to the next, to allow for further detailing by the latter. This is an indirect type of

communication: the moderator distributes among components the semantic elements representing the building elements, according to the needs explicitly specified in the plan. A good example of this are building elements produced by a floor plan component: after registration, floorplan elements could be passed to a shape grammar to detail its geometry or texture. The plan specifies and controls if and how registered elements are passed to which other components. For instance, a plan can specify what the shape symbol of the semantic element (originally created by the floorplan component) should be and, optionally, which semantic attributes are mapped to shape parameters.

As follows from Figure 6.1, the conductor is responsible for executing plan steps, or items, in the correct order. The conductor's function is to parse the plan and, for each item, invoke the correct component through its wrapper. The conductor automatically maps commands in plan items, such as *execute* or *resume*, to the corresponding wrapper methods.

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

6.2 Example of a typical North American style villa

The example proposed in this work is of a typical North American one-storey house with a front porch. This type of building has a non-trivial floor plan and the façade should be generated accordingly. In [92] two more examples can be found. One of them is a Greek holiday villa and the other is a motel lot including rooms, a parking lot and swimming pool.

6.2.1 Building plan

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

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

6.2.2 Generation results

Figure 6.2 presents two example results generated for the above Meadowdale building plan. In the first example, Figure 6.2 (a), we see that the front porch is placed at the front wall segment of the great room, and an additional door is placed on a side wall segment of the kitchen. Window types

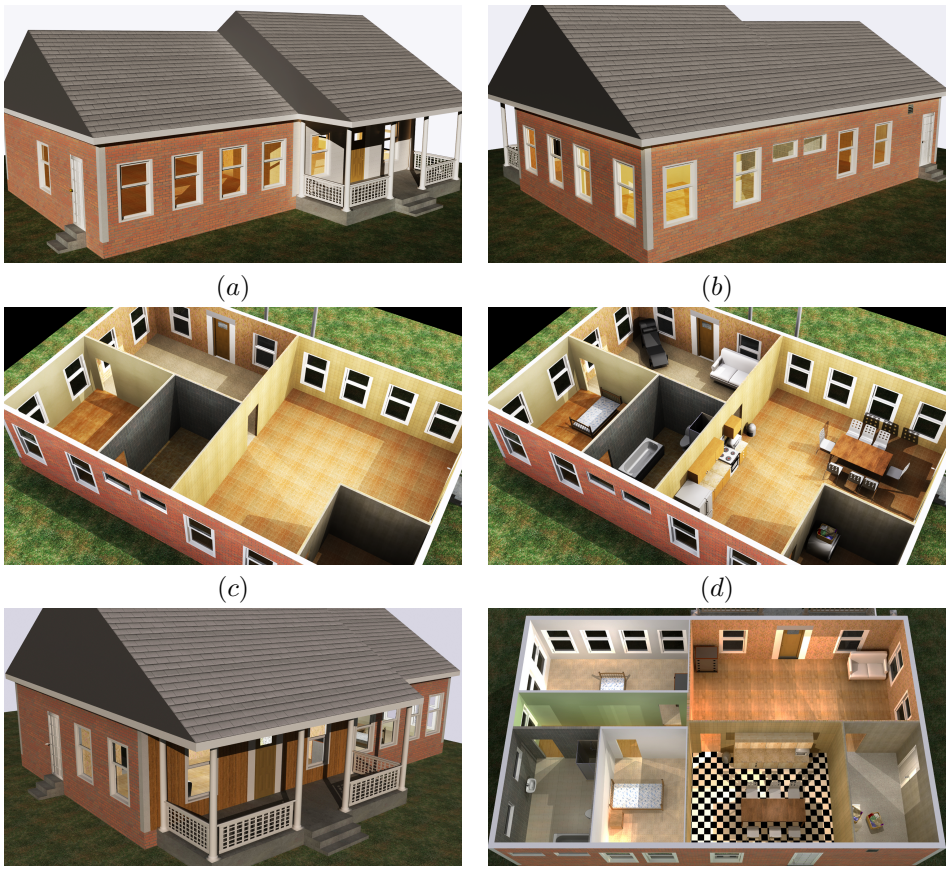


Figure 6.2: Generation of a North American villa: (a) front view with porch; (b) back view with different types of windows and side-doors depending on adjacent rooms; (c) top view on the different rooms (great room, kitchen, bathroom, bed room, laundry); (d) automatically placed furniture based on room types and object relations; (e)-(f) front view and interior view of the same plan, but now using another floor plan generation technique, that of Marson and Musse [56].

and patterns match with the function of the adjacent rooms, as can be seen in Figure 6.2 (b): small windows are placed in the bathroom wall segment; and no windows, but a door and air vent, in the laundry segments. Figure 6.2 (d) shows that the automatically placed furniture matches well with the function of the rooms.

In the second example, the floor plan is generated by the technique of Marson and Musse [56].

Figures 6.2 (e) and (f) show an exterior and interior of the same building. The most noticeable difference between the floor plans, by comparing Figure 6.2 (d) and (f) is the absence of L-shaped rooms and the presence of a corridor. Since this technique uses squarified treemaps, it is unable to produce non-rectangular rooms. To include a corridor, we modified the input parameters for the floor plan component to add two bedrooms instead of one. This resulted in the creation of a corridor to link the bathroom and the two bedrooms to the living room.

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

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

6.2.3 Plan execution

In the first step of the plan, the shape grammar component determines the building footprint inside the garden and extrudes and registers its volumetric shape. As no further rule matches are found, the component halts and the plan proceeds with the floor plan generation. The house has a great room, a kitchen and laundry room, a bathroom, and either one bedroom (in the first example, Figure 6.2 (a)-(d)) or two bedrooms (in the second example, Figure 6.2 (e)-(f)). In the second example, a corridor is automatically added by the floor plan generation technique (see [56]).

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

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

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

6.3 Conclusions

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

This approach used our semantic model, explained in Chapter 4, for integrating different components that implement existing procedural techniques, each of them generating specific building elements. A semantic moderator communicates with these procedural components, and provides them with valuable guidance in order to prevent conflicts among the generated building elements. In this way, we are able to preserve the individual qualities of the integrated components. The moderator keeps a semantic building model that represents each building element generated by the procedural components. Based on this model and on a number of constraints, it maintains the consistency of generated buildings.

The example in the previous section and the additional examples from [92] highlight the central role of the semantic representation by the moderator, coordinating and advising components towards the goal of generating consistent buildings. By correctly using the generic interface of the moderator, procedural components can obtain advice on the impact on building consistency of each of the elements they propose to include. For this, all building elements generated by different components are combined in the central semantic building model, to ascertain that their location and semantics do not conflict with each other. An example of spatial consistency is that the semantic moderator assures that the walls created by a floor plan generator do not intersect the windows created by a façade generator. Another example, but now of functional consistency, is that both these same windows and the furniture (laid out by a third procedural component) are all generated according to the function of the room.

We showed the applicability of our approach with examples from our prototype system, featuring the integration of a façade shape grammar, two different floor plan layout generation techniques, and furniture placement using our semantic layout solving approach explained in Chapter 5. This integration required small modifications and therefore we can consider these to have a relative low burden on developers. Wrapping components and writing a building plan, as done for our examples, are reasonably straightforward implementation tasks. To give an indication of the amount of effort for integrating a new component, the components used for our examples typically took a single developer less than one working day to integrate. The shape grammar component required slightly more effort, as the calls to the semantic moderator had to be made available in the CGA grammar as new shape operations, but still, it was fully integrated within two days.

This integration also had a low impact on performance. Our results show that performance is hardly affected by the moderator checks, conversions and operations. In the examples shown, this overhead lies between 1% and 2% of the total running time (less than 100 ms, in absolute time). This overhead in computation time for the semantic moderator functionality can therefore be considered as perfectly acceptable. Of course, if we were to use highly optimized procedural components, the overhead would be relatively larger, but still minor when compared to the computational cost of most

individual procedural methods.

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

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

Once a game world is either created by hand or generated using procedural generation techniques, we might want to alter its appearance to match changing circumstances. Therefore, the next chapter introduces the concept of procedural filters that can perform such changes, just like image filters change the appearance of pictures.

CHAPTER 7

PROCEDURAL FILTERS

In Chapters 5 and 6, we discussed two ideas that can improve procedural or automatic generation techniques using semantics and thereby aiding designers in creating game worlds: semantic layout solving and using semantics to integrate procedural content generation techniques.

In this chapter we introduce a new concept that allows for easy scene customization. Once game worlds are finished, they are usually static and therefore suiting in the context and the situation for which they were built. But a game might need the same world under different circumstances. In such cases, instead of forcing designers to manually rebuild every element of the game world all over again, we propose using *procedural filters*.

Procedural filters provide a layer of customization that can be applied to a finished game world and the objects therein. These filters will not structurally change the world, but add or change its finishing based on a new context, for example, we might want the same game world in springtime for one level of the game, and in wintertime for another level. Filters use building blocks to alter the visual appearance of objects in the game world. By using the semantic information available in the objects, filters can easily fine-tune their appearance to the specific circumstances of the game world and of the objects themselves.

This chapter will explain the general structure of procedural filters and some results created with example filters. In Section 2.2.4, we already explained how there is a growing trend towards bigger game worlds and how this puts a strain on designers. An even bigger problem arises when a game's story requires the game world to adapt in some way. An example of this is 2K Czech's 2010 game **Mafia 2** in which the player takes on the role of Vito Scaletta, son of Sicilian immigrants, who after returning from fighting in World War 2, gets tangled up in the Falcone crime family. The first part of the game plays out in the winter of 1945 in Empire Bay, a fictional city based on New York, Chicago,

This chapter is a summary of our previous work, published in [93].

Los Angeles and Detroit. Vito, however, gets a prison sentence for stealing ration gas stamps from the Office of Price Administration. After an interlude in prison, Vito rejoins society in April 1951 and finds himself in a new world. Not only is it springtime, Empire Bay has transformed from a depressing, dark, war time city to a bright, lively, prosperous one. This created an amazing, immersive experience for players: leaving prison to come back to a city that looks new and full of life proved to be a memorable experience in the game. The difference in eras was noticeable in the colors, the music and even in the handling of the new era cars. However, this also meant that the city modeling and visualization needed a considerable change. Obviously, a lot of new buildings and cars had to be modeled, however many parts of the world that were kept also required a makeover. Especially the difference in seasons required a lot of extra effort: for example, in the winter the roads and rooftops need to be covered with snow and in the spring, outdoor tables and seats are placed in front of cafés and restaurants.

Recreating a game world under different circumstances can be a time-consuming process, however the experience can prove to be worthwhile for the player.

In this sense, it would be great to give designers a tool to make the customization of game worlds easier. We took our inspiration for the notion of procedural filters from 2D image editing tools. 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 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 7.1.

Inspired by that concept, we developed the notion of *procedural filters* for virtual worlds, which we define as: *procedures 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', as in the example we gave of **Mafia 2**. 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 if 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 can be used to soften the load of this process. In particular, it enables designers to incrementally create and fine tune their own procedural filters, and apply them on (parts of) the game world during the design phase.

Embedding these procedural filters with semantics provides a generic specification scheme in which a set of instructions can be tied together to describe this customization process of virtual worlds. Using our proposed semantic model, 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

Figure 7.1: The painting *'Het meisje met de parel'* (Girl with a pearl earring), by Johannes Vermeer (1665-1667); from left to right: original image, with an emboss filter applied, and with a patchwork filter applied.

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, you 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 contains 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, once 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 this chapter, we discuss the procedural filter approach, and present some examples generated using a filter editor (which will be explained in Section 9.2.3).

7.1 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 instructions that form a certain procedure. A filter is represented by a graph, in which the instructions 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.

7.1.1 Filter instructions

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. procedural generation methods.

We will now describe each category in more detail:

- **Basic operations** are obviously necessary to perform even the simplest 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 keeping the philosophy of procedural filters in mind, in order 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. In addition to changing the color or texture, one can add or remove shaders, alter shader parameters to match the semantic characteristics of the object or the scene or simply modify or replace 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.

The final two categories are more elaborate than the above mentioned ones, and will be explained in the next two subsections.

7.1.2 Using semantics in procedural filters

Semantic queries are probably the most important and powerful instructions with regard to the reusability and generic nature of procedural filters. Querying the semantics of a scene involves mostly selecting from the scene objects of a particular PHYSICAL OBJECT CLASS and querying their ATTRIBUTES. When such instructions are available, the filters will be much less ad-hoc and therefore much more reusable. Moreover, the ATTRIBUTES of an object (e.g. their age or the level of destruction) can be directly linked to parameters of filter instructions or to variables of shaders used by the object's material. Such 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.

7.1.3 Procedural generation methods in filters

Instructions in this category 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 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. Based on the effect one wants to create, a range of procedural techniques would be useful in any procedural filter system.

For almost any procedural generation technique, we can find a use in procedural filters. *Procedural textures* are an extremely powerful technique for customization. We can add rust to metal objects, add moss or other kinds of natural phenomena to walls, add stains on various objects, create burn marks or crackling paint effects, etc. The use of such textures is virtually endless in game world customization. *Procedural destruction* could be used to generate effects for scenes of cities during a war, after riots, after an explosion or demolition. It could be used to smash in windows, tear down walls or other structures and break up compound objects. For some other effects, we need *procedural geometry generation*: we might want to add shrubbery and weed to the yard and planks in front of windows to create the effect of an abandoned lot. Or as we will see in our examples later on, we can use it to generate rubbish on a front porch, or to throw around empty cans and bottles to create an after-party effect. It is clear that the more such techniques are available to a designer, the more versatile the filters become.

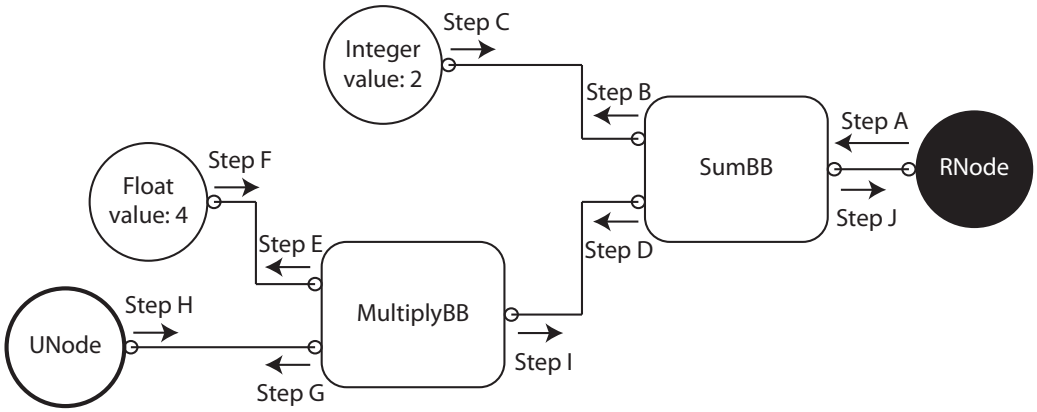


Figure 7.2: The execution flow of an example filter. See the accompanying explanation for more details.

7.1.4 Filter execution

To make sure the entire filter is executed, but only that portion which is necessary, the execution starts from the end and works its way back to find values only for the necessary input, following a *depth-first search* approach. Starting from the result, instructions ask results from preceding instructions to decide their input until no further input is required. In Figure 7.2 this flow is represented in an example.

Step A When the result of a filter needs to be obtained, we work our way back from the RNode or *result* node. In the example, the RNode is connected to the *Sum* building block.

Step B The Sum building block needs two input parameters. The top parameter is obtained first.

Step C This input parameter is connected to an integer primitive with value 2.

Step D Now the bottom input parameter for the Sum building block is obtained.

Step E This second parameter of Sum is connected to a *Multiply* building block, and again two inputs are necessary. We start by obtaining the top one.

Step F This input parameter is connected to a float primitive with value 4.

Step G Now the bottom input parameter for Multiply is obtained.

Step H This input parameter is connected to a UNode, meaning it is dependent of another filter. The UNode will execute this filter and return its result.

Figure 7.3: 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.

Step I The Multiply building block returns the multiplication between the two obtained input values.

Step J The Sum building block returns the sum of the two obtained input values to the result node. This will, in turn, pass it along to the *caller* of this example filter.

7.2 Examples

To show the application of procedural filters we used the editor, discussed in Section 9.2.3, to create a party filter and several example filters, performing aging and deterioration filters for houses. 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, applying them to those scenes.

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 in turn combinations of smaller sub-filters and blocks: e.g. using 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 7.3.

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 that can be handed as a parameter to the building block, which, in turn, is fed a random 2D point on the texture to create a variety of 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 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 threshold and the moss texture when below the threshold. In other words: the higher the threshold, and thus the higher the age of the building, the more moss will be visible.

In Figure 7.4, we see a street with a row of similar houses, generated using the same shape grammar. In Figure 7.4a, all houses are new. In Figure 7.4b, 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 7.4c, 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 7.5, 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 Chapter 5. 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 would 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).

7.3 Conclusions

This chapter proposed our *procedural filter* approach aimed at assisting designers in customizing virtual worlds or scenes. We defined procedural filters as sets of instructions to be applied on (part of) a virtual world in order to customize its appearance or give it 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 incrementally build up a relatively complex filter. We implemented this approach within a visual editing and testing environment for procedural filters (discussed in more detail in Section 9.2.3), and showed the results of a number of test filters.

Figure 7.4: 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 7.5: 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.

When semantic attributes are 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. We showed this by integrating the concepts from our semantic model (see Chapter 4 into the implemented procedural filter editor.

Based on interviews with people involved in game development, we noticed that the editor and the approach in general were perceived as a good addition to the current game development workflow. Most interviewees were convinced it could save a lot of time and would greatly increase usability. An evaluation of the procedural filter approach can be found in Section 11.2.4.

The last three chapters have been applications of the semantic model to aid the creation of game worlds. In the next chapter, we will introduce the concept of SERVICES to enable designers to specify behavior of game objects.

CHAPTER 8

SERVICES

Up to this point we have looked at how our semantic model can be used to help designers build up a game world more easily by using (i) semantic layout solving, (ii) a semantic moderator to integrate procedural generation techniques and (iii) procedural filters.

We now focus on the runtime phase of the game. In Chapter 4 we briefly discussed the concept of SERVICES that can be used to describe *functional* information of objects, i.e. the way they behave and respond to interaction of the player, characters and other objects.

This chapter will explain in detail the structure of SERVICES, their components and their relation to other entities. We will take a closer look at how the interactions and effects of SERVICES are handled at runtime, and how this helps maintaining the semantic consistency of the game world. We also show some practical applications of services to simulate and handle particles and agents in a dynamic game world.

By expressing semantics on the behavior of objects or the game world itself, we can capture how the world needs to respond to changing environments, as well as when and why these changes should happen. By allowing designers to specify this behavior and by handling the effects of the behavior at runtime, SERVICES can help to create dynamically changing game environments.

8.1 The structure of services

Entities in the game world should be able to behave as their real-world counterparts, for example, a jacket has the service of providing warmth to the person wearing it, but a campfire or electric heater provides the same heating service, only to all objects within a certain range. As such, services are a very powerful way to express semantics in game worlds.

This chapter is a summary of our previous work, published in [45] and [46].

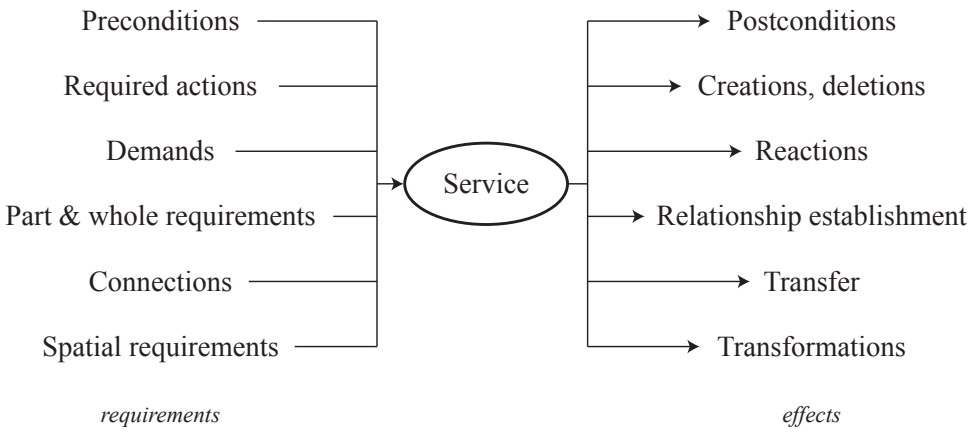


Figure 8.1: A service can have several types of requirements and effects.

In the following section we will come back to some of the definitions given in Chapter 4. We defined a SERVICE as the capacity of an ENTITY CLASS or of MATTER to perform an ACTION, possibly subject to some requirements. An *action* is a process performed by an instance of an ENTITY CLASS (and consisted of MATTER), yielding some effects like ATTRIBUTE value changes or yielding new instances of ENTITY CLASSES; e.g. a vending machine providing a can of soda or an oven that **heats up** objects inside it. An ACTION always requires an *actor* and optionally a *target* and/or some *artefact* (like a *gun* for the shoot SERVICE between two characters), all of which are either ENTITY CLASSES or MATTER.

First we will discuss what kind of *requirements* one can specify for a SERVICE and what possible *effects* the execution of a SERVICE can have on entities in the surroundings of the actor or target of the SERVICE, or on otherwise related entities. Then we discuss the importance of CONTEXTS, introduced in Chapter 4, in combination with SERVICES on reusability and ease of specification. Finally, we describe how *temporal* and *spatial* properties play a role in the structure of a SERVICE.

8.1.1 Requirements and effects

As explained in the definitions, SERVICES specify the ability to perform a certain ACTION subject to some requirements. These ACTIONS in turn yield some effects. We will give a detailed overview of the different types of requirements and effects that can be specified, each with some examples.

Figure 8.1 shows all types of requirements and effects that can be associated with a service. First, we enumerate the different types of requirements:

Preconditions defining when a service can or should be provided.

- A vending machine can dispense soda cans if some are *available in its inventory*.
- A TV only increases fun when *switched on*.
- A door can only be opened when it is *not locked*.

Required actions necessary to perform in order for the service to be provided.

- A jacket only provides warmth when *worn*.
- A book provides knowledge when someone *reads* it.
- A chair will provide comfort when one *sits* on it.

Demands of some goods in exchange for the service to be provided.

- A vending machine will only dispense soda cans if the actor deposits *a coin*.
- It is only possible to craft an object on a workbench when the *necessary raw materials* are available.
- You can only chop down a tree when using *an axe*.

Part & whole requirements for the service to be provided.

- A car needs *a functioning engine* to run.
- A printer will only work when a *cartridge or toner* is available and not empty.
- A bicycle needs *two wheels* and *pedals* to be ride.

Connections that are required to provide the service.

- Electrical devices need to be connected to the *electricity mains*.
- A hose needs a connection with a *water faucet* to spray water.
- A cart needs to be connected to a *horse or other beast of burden* to be able to move.

Spatial requirements for the service to be provided.

- The automatic doors will open when someone walks or stands *in front of the doors*.
- Water will extinguish fire *upon contact*.
- Wood will catch fire when *near a flame*.

Next we enumerate the different effects:

Postconditions after the service is executed.

- After eating an apple, the eater's *hunger level goes down*.
- When a civilization hands over a tribute to another one, the *strength of the ally relationship will increase*.

- After turning on an electric device, its *state becomes 'on'*.

Creations, deletions of instances or matter that occur as the result of the service.

- A sword factory creates *new sword instances*.
- While driving, *gasoline is removed (deleted)* from the gas tank.
- After exploding, a *bomb is removed*.

Reactions on the service.

- Upon hitting an enemy, that enemy will strike back by *hitting the assailant*.
- When a smoke detector is *triggered* by some smoke, an alarm will *sound*.
- When restructuring a company, the workers might react by *revolting*, which can involve going on a strike or sabotaging the company's supply chain.

Relationship establishment (or termination) after performing the service.

- After giving birth to a kid, a *parent relationship is established between mother and child*.
- When the player sells an object the *ownership relationship with the object is removed and another one is established between the buyer and the object*.
- When defusing a bomb, the *connection relationship is removed between e.g. the detonator and the explosives of the bomb*.

Transfer of entities as a result of the service.

- A vending machine will transfer the purchased *soda can* from its inventory to the buyer.
- Equipping a tool or weapon will transfer it *from the inventory to the player's hand*.
- A teleport will transfer a player *from one space to another*.

Transformations happening as a result of the service.

- After reaching a certain age, a *chick transforms into a chicken*.
- After casting a spell, a character might *transform into something else*.
- A weapon might transform from a *semi-automatic weapon into an automatic one* after applying an upgrade.

All these requirements and effects allow designers to specify a wide range of different services in a uniform and generic fashion. *Uniform*, because the basic structure of a SERVICE is always the same: ACTION - requirements - effects: a basic and simple structure that, however, does allow very complex behavior. And generic, because, by allowing to specify each SERVICE on the best-suitable level in the hierarchy of ENTITY CLASSES (i.e. the most generic ENTITY CLASS to provide the SERVICE), we prevent designers from having to specify the same SERVICE multiple times. For example: the

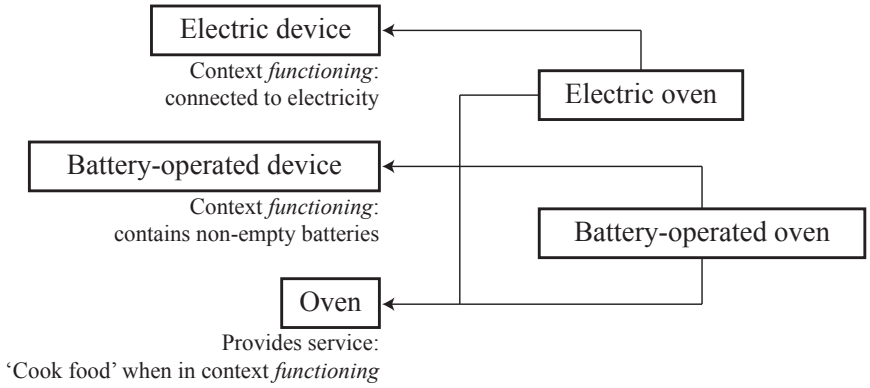


Figure 8.2: Both electric and battery-operated ovens can cook food, but the required CONTEXT named *functioning* differs and is specified higher in the hierarchy for the generic TANGIBLE OBJECT CLASSES **Electric device** and **Battery-operated device**. This way, no behavior needs to be specified for the two specific ovens, since they inherit all necessary behavioral semantics from their parents.

SERVICE of *killing* should be specified for a generic ENTITY CLASS **weapon** from which other ENTITY CLASSES like **spear**, **gun** or **sword** can inherit.

However, requirements for the SERVICE might differ for each of these inheriting ENTITY CLASSES. To handle this, we can use CONTEXTS.

8.1.2 Context-sensitive services

The SERVICE requirements can be combined in the concept of CONTEXTS (introduced in Chapter 4 and used for semantic layout solving in Chapter 5). The generic setup of the SERVICES in our semantic model, combined with the CONTEXTS, has important potential for reusability. Both requirements and effects of actions can be defined (and redefined) for an ACTION on each level of the entity hierarchy. The same holds for the conditions of a particular CONTEXT for an ENTITY CLASS. This way, both CONTEXTS and the SERVICES that use these CONTEXTS can be separately specified at different hierarchic levels. This way, two ENTITY CLASSES inheriting from the same parent class can both differently redefine a CONTEXT used in a SERVICE of their parent class. We will clarify this with an example.

In this example, we specified the TANGIBLE OBJECT CLASSES **Oven**, **Electric device**, **Battery-operated device**, **Electric oven** and **Battery-operated oven**. **Electric oven** inherits from **Electric device** and **Oven**, while **Battery-operated oven** inherits from **Battery-operated device** and **Oven**, (see Figure 8.2). We can define the CONTEXT **functioning** for an **Electric device** as a required connection to the electricity mains and for a **Battery-operated device** as requiring non-empty

batteries. The SERVICE **Cook food** can be specified for the generic TANGIBLE OBJECT CLASS **Oven**. Now, simply by means of the inheritance, both the **Electric oven** and the **Battery-operated oven** will contain all necessary behavioral semantics. No new SERVICES and no new requirements or effects need to be specified in either of these TANGIBLE OBJECT CLASSES.

8.1.3 Spatial and temporal properties

We already explained that designers can specify effects of SERVICES and their requirements, possibly through the use of CONTEXTS. However, SERVICES might also be influenced by spatial and temporal properties.

First of all, the *spatial* properties define the *range of influence* of the effects of a service. A jacket warms up the person wearing it, an oven warms up the items inside it and a heater or camp fire warms up all entities within a particular range. It is obviously necessary that these differences can be specified in the SERVICE. This range of influence can go beyond a physical range. The effects can influence all related entities of the actor or target of the service. For example, when creating a strategy or city-building game, one might want to create a SERVICE for a resource gathering building that increases the resource supply of the owner, i.e. the source of a RELATIONSHIP of type **owns** for which that building is the target.

The *temporal* properties define how the effects of a SERVICE vary in time. The most important temporal property is the choice between *discrete* and *continuous* effects. Discrete effects can be a vending machine supplying a single can, an ATM machine supplying a certain number of bills or a grenade decreasing the health of people within a certain range one single time. Continuous effects can be the continuing increase of the hunger level of a person, a television gradually increasing the *fun* of people in the surroundings or a poisonous gas cloud continuously decreasing the health of all people within the cloud.

A second important temporal property is one dealing with chances. It is often not desirable to have an effect happen every time, or always at the same moment. For example, we might want to define that there is a one in twenty chance that a grenade does not detonate. We do not want to specify that a living creature dies when reaching a certain age, however the chances of that happening do increase at a higher age. A character performing a lockpick ACTION on a lock, will sometimes fail. To account for this, we include *chance* properties within the definition of a SERVICE.

In that last example, we might want to say that the chances of a lockpick ACTION failing are dependent on the abilities and experience of a character. Also, all other values specified in the SERVICES might need to be dependent on other ATTRIBUTES. For example, the decrease of the level of hunger ATTRIBUTE of a character might be dependent of the nutrition value ATTRIBUTE of the eaten piece of food. It is therefore important that an attribute can be expressed depending on ATTRIBUTE values of the actor, target or artefacts of the SERVICE or otherwise related entities.

8.2 Consistency maintenance using a Semantics Engine

To handle SERVICES of entities, we created a proof-of-concept implementation which we call the *Semantics Engine*. Analogously to what a physics engine does with in-game physics, the Semantics Engine maintains the in-game semantic consistency of the world. After creating GAME-SPECIFIC CLASSES, instances of them can be placed in a semantic game world. By making use of the Semantics Engine, game programmers do not have to implement the execution of the semantic behavior of these instances, as the engine is charged with this handling.

The engine has several main features. First, it maintains all the instances that have been created, and checks whether they have any active SERVICES. This procedure to update the semantics engine is presented first in pseudo code:

```
// The semantics engine keeps a list of all currently active services
Service[] currentlyActiveServices;

// ---- UPDATING THE SEMANTICS ENGINE ----
// Because we want the resources needed by the semantics engine to be as
// limited as possible, we allow the user to specify a particular area
// that needs to be updated. This can, of course, be the entire world if
// necessary.
function void updateEngine(area, deltaTime)
{
  // We loop through all instances in the given area to
  // look for changes in services
  for each entityInstance in area
  {
    CheckServiceActivity(entityInstance);
  }

  // All services that are currently active are updated
  for each service in currentlyActiveServices
  {
    UpdateService(service);
  }
}


function void CheckServiceActivity(entityInstance)
{
  // For all defined services for the instance, the
  // requirements are checked
  for each service in entityInstance.Services
  {
    allRequirementsSatisfied = true;

    // First we check whether or not the action that triggers the service
```

```
    // is executed in the previous time step.
    if (service.ActionExecuted())
    {
      for each requirement in service.Requirements
      {
        // Based on the type of requirement the necessary checks are made
        //  e.g. preconditions of attribute values or the position of the
        // instance for spatial requirements, etc.
        if (!requirement.IsSatisfiedFor(entityInstance))
        {
          allRequirementsSatisfied = false;
          break;
        }
      }
    }
    else
      allRequirementsSatisfied = false;

    // If necessary the service is actived or deactivated
    if (service.IsActive && !allRequirementsSatisfied)
      DeactivateService(service);
    if (!service.IsActive && allRequirementsSatisfied)
      ActivateService(service);
  }
}

function void ActivateService(service)
{
  service.IsActive = true;
  currentlyActiveServices += service;
  for each effect in service.Effects
  {
    // All discrete effects are handled now (the continuous effects are
    // handled in the UpdateService function, since they need to happen
    // as long as the service is active instead of just once. Since
    // chances are attached to effects, these are also taken into account
    if (effect.IsDiscrete && Random.GetChance(effect.Chance))
      HandleEffect(effect);
  }
}

function void DeactivateService(service)
{
  service.IsActive = false;
  currentlyActiveServices -= service;
}
```

```
function void UpdateService(service)
{
  for each effect in service.Effects
  {
    // All continuous effects are handled in this update function.
    // Like for discrete effects, chances are taken into account.
    if (effect.IsContinuous && Random.GetChance(effect.Chance))
      HandleEffect(effect);
  }
}

function void HandleEffect(effect)
{
  //—— Based on the type of effect, it is now handled:
  //—— this can be the creation or deletion of an instance,
  //—— the establishment or termination of a relationship, etc.
}
```

For all instances in the area that needs to be updated, the engine checks if there is any change in the activity of the SERVICES. For every SERVICE it is checked whether or not the ACTION that triggers this SERVICE is executed. If so, then all requirements are evaluated. For a precondition, for example, it is checked whether the corresponding ATTRIBUTE has a particular value. The ACTION does not need to be triggered by another entity, but might also be a scheduled ACTION, e.g. that needs to be executed every second, or every minute. Based on this, the SERVICE can be (i) activated when inactive and all requirements are met, (ii) deactivated when active and not all requirements are met anymore, or (iii) else it is left unchanged. When being activated, the SERVICE is added to the list of active services and all discrete effects are handled. When being deactivated, the SERVICE is removed again from the list of active services. After this first step, all SERVICES in the list of active services are updated. This means that all of the continuous effects are handled. Both for the discrete and the continuous effects, chances are taken into account, i.e. when a random value between 0 and 1 is bigger than the occurance chance of the effect, the effect is not applied. The handling of effects is different based on the type of effect. It might involve the creation (or deletion) of an entity instance or a relationship, the transformation of an instance into another entity, a change in an attribute value, etc.

At runtime, the engine updates the semantic representation of the game world, by performing the instances' ACTIONS for the right amount of time on the relevant targets. This might have an effect on ATTRIBUTE values of instances, or the inventory of an instance. This, in turn, can result in an active SERVICE becoming inactive, or in the activation of an inactive SERVICE, of which the ACTIONS will then be executed. For example, if someone uses the **turn on** ACTION on an oven, it will get in the *on* state, making it heat up its inventory items as long as it stays on.

The semantics engine also offers game programmers useful methods to improve in-game object interaction. An example is the enforcement of the **take** ACTION, which should not always be executed

automatically as a SERVICE, but only when the player actually chooses to pick up an in-game item. Another example is requesting what ACTIONS are useful to perform on a certain instance, which the programmer can use for the graphical user interface of a game, or whatever way he chooses to let the player decide what ACTION he wants to perform. Another feature of the engine, regarding the interaction between artificial agents and game objects, will be described in the following section.

In addition to the engine, we have implemented an interface between the semantic layer and an actual game. It makes sure that the Semantics Engine is updated and that instances that are created or removed by the engine, are actually created in and removed from the game world. For example, when the engine makes someone execute the **eat** ACTION on a pie, it notifies that the pie should be removed from the game, which is then done through that interface. It also allows instances to convey more than semantic only, as they might be subject to physics as well, or linked to particles, as discussed in the following section.

Because of the Semantics Engine, (ATTRIBUTES of) instances are constantly updated, resulting in a lot of dynamics, assuming SERVICES have been defined for them. This leads to adaptive game worlds that change over time, forcing the player to adapt as well and think about the results of an ACTION, but also allowing him to think creatively to accomplish something. This is a great improvement with respect to games that are currently available. However, it also leads to a downside, if the addition of semantics to games results in too much extra memory overhead, or if processing the handling of SERVICES requires too much time. This is undesirable, because current games already require a lot of computational power and there are still limitations on the hardware. The Semantics Engine has therefore been optimized in some important ways. For example, instances are only updated when they have active SERVICES, and checking for new active SERVICES is only done when specific properties are modified.

In order to measure the performance of the engine, we have created three simple test cases. To provide reliable and representative results, we have decided not to use a world with different GAME-SPECIFIC CLASSES, but a world with instances that are all based on the same GAME-SPECIFIC CLASS. Testing the execution time of single ACTIONS is immeasurably small. Upscaling everything to the extreme, on the other hand, is a good way to check whether the engine is capable of handling a huge amount of semantic instances. For each test case, we have therefore generated 10, 100, and 1000 instances, respectively.

1. In test case 1, humans are becoming hungry, by having a SERVICE that has an effect on their 'hunger' ATTRIBUTE each second.

2. In test case 2, campfires are placed throughout the world in such a way that they have two other fires in their radius, which they heat up by raising their 'temperature' ATTRIBUTE. As more fires are added, more targets will be affected, resulting in extra overhead, and making it a useful test.

3. In test case 3, we keep providing factories just enough steel to produce one sword at a time, in order to test a SERVICE of the demand/supply type. For simplicity, we assume that the amount of steel is unlimited, and there are no SERVICES defined for the swords.

Table 8.1: The update time of the Semantics Engine for Test Case 1

	10 humans	100 humans	1000 humans
Minimum (ms)	0	0	5
Maximum (ms)	1	1	8
Average (ms)	0	0.0036	5.6

Table 8.2: The update time of the Semantics Engine for Test Case 2

	10 fires	100 fires	1000 fires
Minimum (ms)	0	0	8
Maximum (ms)	1	3	11
Average (ms)	0.002	0.066	8.96

Table 8.3: The update time of the Semantics Engine for Test Case 3

	10 armories	100 armories	1000 armories
Minimum (ms)	0	0	426
Maximum (ms)	2	40	936
Average (ms)	1.11	29	512

The update times of the engine have been measured, and are shown in Tables 8.1, 8.2, and 8.3. For all test cases, we have used an Intel Core 2 Quad Q9000 2.00 GHz PC with 6 GB RAM. Having 100 instances and SERVICES requests more of the engine than 10 instances, but performance is still quite well, and will probably not slow down a game that much. The extreme cases show that the engine can still be improved, although the average is still pretty decent for an engine that, being a research prototype, has not yet been optimized as much as possible. Test case 3 shows a bottleneck, which can be explained by the ongoing creation of swords and exhaustion of steel. It is clear that the engine is not yet capable of handling thousands of instances at a time, although this extreme scenario is unlikely to appear in real game worlds. In strategy games, for example, instances will only be created gradually over time.

8.3 Applications of semantic game worlds

Here we describe some specific proof-of-concept implementations that show the use of services and the semantic model in action. More importantly, these examples show that the generic nature of the concept of SERVICES in our semantic model make them easy to integrate with existing game engine components and systems.

8.3.1 Particle systems

Particle systems were introduced as a method for the modeling of fuzzy objects [75]. Although designed for films in the 80's, particles are nowadays heavily used in games as well. Their primary use has been unchanged, though, as they still serve as eye candy. Some examples are the visualization of fire, clouds, and water.

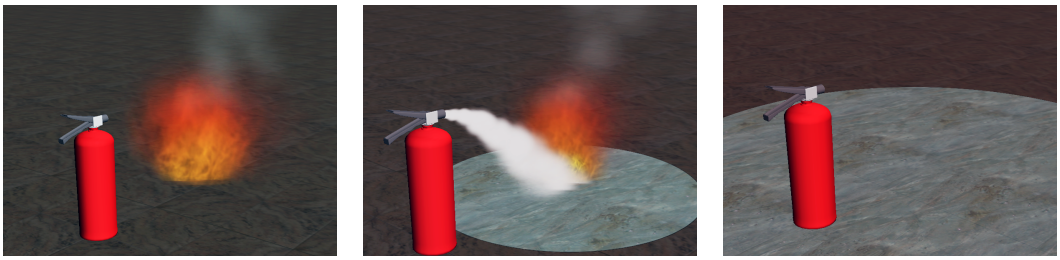
Particle systems consist of emitters that spawn particles at a certain rate, and emit them with a certain speed in a fixed or random direction. Particles themselves have basic properties (e.g. radius, transparency, and color), possibly subject to changes over lifetime. In games, they are usually visualized by textures that are oriented towards the viewer, a technique called *billboarding*. Particles may be affected by a physics engine, making them subject to forces like wind and gravity.

In some games, particles are combined with the handling of physical processes. An example is *BioShock* [39], where a fireball can melt ice, and bodies of water can be electrified. Although those effects work as expected, we think that defining the interaction between such processes can be simplified by combining semantics with particle systems. Similarly, simulations will be able to become more than meets the eye, as they offer more generic behaviors, therefore leading to a more widespread use. For example, once the effect of electricity "particles" on water has been defined, water throughout the entire game world can be electrified, instead of just the intentionally placed bodies of water, as is the case in *BioShock*.

For this, we consider SUBSTANCES as ENTITY CLASSES that can provide one or more SERVICES, like PHYSICAL OBJECT CLASSES. Instead of assigning them a model for visualization, we link them to a particle emitter or particle properties. A fire model, for example, consists of two emitters: one for the flames, and one for smoke. A flame is then considered an ENTITY CLASS as well, having certain properties for the behavior of its particles, and a SERVICE stating that it should heat up its surroundings. As long as the fire is burning (which can be defined as an ATTRIBUTE), it will spawn new flame and smoke particles. The fire itself does not have a SERVICE, as it is just a holder for flames and smoke. A flame, on the other hand, has a SERVICE, and can be reused for other entities that should emit flames, like a furnace.

Particle emitters can also be linked to PHYSICAL OBJECT CLASSES. In most of these cases, it may be required to define the source of the particles as well. Consider a fire extinguisher with fire fighting foam, where the foam, having a SERVICE to cool down entities it collides with, is represented by particle properties. Only as long as there is foam in the extinguisher's inventory, it is able to spawn particles in the form of foam. Held in front of a fire, the flames will cool down, eventually extinguishing the fire.

The approach with particles works in theory, but one should take care in practice, when handling SERVICES for entities that are represented by particles. Particle systems can consist of hundreds of particles at the same time, and updating all of their SERVICES each time might require too much processing power, especially if collision checks have to be made. Although research has been done towards the optimization of collision detection [26], we overcome the problem in our framework by not considering particle semantics unless the game requires to. This way, the game programmer



(a) (b) (c)

Figure 8.3: Extinguishing a fire with fire fighting foam.

can decide the amount of collision checks, what kind of collision checks to make (particle-particle or particle-physical object), and report collisions to the semantics engine, which takes care of the SERVICES of the representing entities. For example, in case of games that already require a lot of processing power, the programmer might reduce the amount of checks, while in case of a less demanding game or a small number of particles, checks can be made during each update of the game. Figure 8.3 presents an example of a fire (a). By using the fire extinguisher, the fire will gradually become smaller (b), until it is extinguished. The ground becomes wet, though (c).

Letting SUBSTANCES have their own SERVICES also opens up the possibility to mix fluids. When two SUBSTANCES are mixed together in the right amount (stated as a certain ratio), they both disappear (semantically speaking), making place for a new MIXTURE, possibly having its own SERVICES. In role-playing games, for example, this might be useful for the creation of potions, where different ingredients (each having services of their own) mixed together can result in a potion with its own specific effect. This already happens in the game *The Elder Scrolls IV: Oblivion* [5] and many other RPG's.

8.3.2 Agents

In many games, the player is not the only character in the environment. Non-playable characters inhabit many worlds as well, providing assistance to the player, trying to chase him instead, or just wandering around. These characters used to be pretty dumb, doing no more than what them was prescribed. Nowadays, all characters are expected to show more complex behavior: reacting to the player's actions, adapting to environmental changes, working together, and so on. The driving force behind these so-called *agents* is artificial intelligence (AI). Over the years, research towards agent systems has resulted in agents that behave more autonomously. However, with the increase in complexity and dynamic behavior of current virtual worlds, new techniques are still required.

A recent technique that can be used to let agents adapt to dynamic changes within an environment, is *reinforcement learning*. Mehta et al. [58] let agents adapt their behavior sets in complex real-time domains during runtime, and let them monitor and reason about their behavior execution to carry out revisions on behaviors. Hanna et al. [33] describe an approach to deal with the high dimensionality

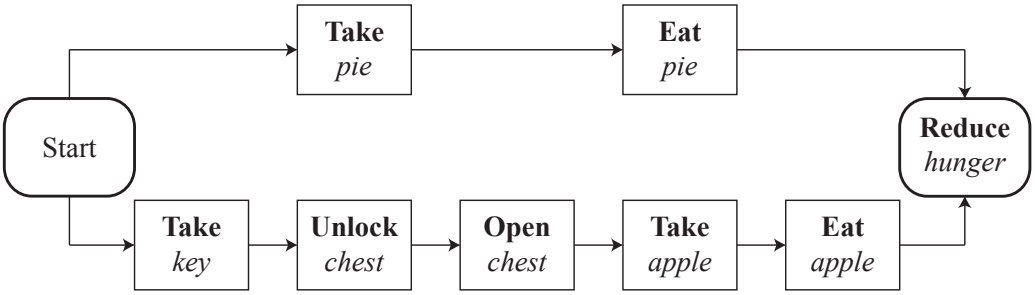


Figure 8.4: A graph with all possible paths to reduce the hunger attribute.

of agent sensor state spaces, by dividing them into smaller sub tasks.

With the addition of a semantic layer to game worlds, agent behavior can be easily improved, as agents should be able to make use of semantic objects as well. Our framework supports the use of agents, by providing several methods to search through the instances in the game world. In case an agent is looking for a specific game object, the Semantics Engine will provide information about all instances of that game object, including their positions, so the agent can find its way towards them. This is not sufficient, though, as some instances might be inaccessible, for example in inventories of other instances, and can only be supplied when certain service requirements have been met. An example is a can of soda that is only supplied by a vending machine in exchange for a coin.

In related research, smart objects (see Chapter 2) have already been used for planning purposes, as by using scripts, actions are defined [1]. A planning algorithm is used to make the agent move through and interact with the world. To do so, he collects relevant information about the state of the world, prepares a planning step, makes a plan, and executes it, where the agent's actions are taken from the plan. In another system, *Teletubbies* have been turned into intelligent agents by giving them *drives*, like hunger [4]. Now, if an agent gets hungry, he will search for objects that have the effect of decreasing the level of hunger, make a plan to reach the object, and use it. Afterwards, his other drives will guide him to the next location/object.

The Semantics Engine makes use of pathfinding techniques to find instances, while considering all possible requirements and actions to retrieve them. For this, a directed graph is created that contains all the paths to reach a certain goal. For example, this goal can be to find a specific instance, or to let an attribute reach a specific value. Although there is a wide variety of graphs in existing techniques, our version consists of two or more nodes, including at least a start node and an end node, the latter representing the goal. Between the start and end node, one or more paths can be present, each providing an alternative route to get to the goal. Each path contains one or more nodes, each linked to an ACTION that should be executed by an agent. By using *backward chaining*, so starting from the end node and moving towards the start node, the entire graph can be created. More information about pathfinding techniques and backward chaining can be found in [61].

Take for example an agent that is hungry. It might request the Semantics Engine to look for instances that can reduce the value of its 'hunger' attribute. Suppose that somewhere in the world, there are a pie and an apple, the latter one being locked inside a chest. Also suppose that the key is lying at a place where it can be freely taken. The resulting move graph might then look like the one in Figure 8.4. Starting from the goal, the engine looks for suitable instances, and checks whether they can just be taken. If that is the case, like in the upper path of the figure, there will be a simple path that just consists of a 'take' move node, and a node containing the ACTION to reduce the hunger: 'eat'. In case a suitable instance is owned by another entity, the engine checks whether that entity has a service to supply it. If so, the requirements are checked, possibly inserting a node that demands the agent to perform a certain ACTION on the entity. This process is recursively repeated, possibly resulting in more paths, which are appended to the path that is observed at that time. In the lower path of the figure, this is shown by the apple in the chest, for which the key should be found first. Note that due to backward chaining, the engine creates this graph from the right to the left, one node at a time, while the agent should perform the actions from the start to the end, choosing only one concrete path.

The structure of services share similarities with the *Affordance Theory* as introduced by Gibson [28]. He defined affordances as all possible actions in the environment in relation to the actor's capabilities. This idea has been used in many AI projects often used to simulate human behavior for agents, e.g. by Silverman et al. [78]. Our model allows actions and consequences to be specified in a similar manner, namely, between an actor and a target (the actor's capabilities can be taken into account when specifying the effects of an action). And, as mentioned before, the semantic game world can be queried to return the possible actions (or affordances) in a particular area, e.g. in the surroundings of a particular character, so it can be used by other systems that simulate intelligent human behavior. The difference with our model is that we are not focused specifically on human (or agent) behavior. We want to specify behavior of any object in the world, e.g. of plants or animals aging or growing, a heater warming entities in its surroundings without them interacting with the heater or a smoke alarm that sounds when smoke is present in a particular area. This allows the semantics engine to simulate effects of actions by both agents and other objects in the world. Furthermore, a service does not need to be triggered by an action. Some services might be triggered by attribute changes (and therefore potentially implicitly by effects of other actions) or on regular intervals.

In a state-of-the-art report about intelligent virtual environments, some techniques are discussed where AI is used as a component of virtual environments [3]. One of the problems that the authors point out is action selection. If there are multiple tasks that an agent can do at a given moment, there should be a way to select the action that is the 'best' for that moment. This could be achieved by using reasoning mechanisms or rule-based production systems. The Semantics Engine provides a graph evaluator, with which all possible paths can be evaluated, according to certain criteria. The least number of moves is a possible criterion, although another agent might want to cross the shortest distance. The evaluator tells an agent what move to perform next. Whenever a (sub)path has been chosen, and a move has become invalid (for example, an instance might have been taken by someone else between search and execution), an agent can go back in the graph to try another path, or search again.

Our framework allows agents to be aware of everything that is happening in the game world, and to be aware of all available objects. Although this may be desired for some games, having all-aware agents may be unrealistic, and even unfair towards the player. Instead of giving agents knowledge about the entire world, a game developer might decide to let an agent have its own world state, and apply learning patterns to update its knowledge about the whereabouts of other entities. This can be used in combination with the search methods, making an agent use only the objects that it has come across. Although the current framework does not support individual world states, future work might include it.

In case a game object should be intelligent (any living entity controlled by an agent), *desires* can be specified for it. A desire can be pursuing a specific instance, executing a service, or reaching a specific attribute value. An example is the desire to keep the hunger level at a certain value. In case the level becomes higher than that value, the agent can go look for an instance in the world that satisfies its hunger. As another example, one might also design an agent with the service of supplying first-aid kits to the player, and specify that service as a desired service, so that the agent will autonomously go look for first-aid kits in the world, and bring them to the player.

In Section 10.4, the idea of *semantic crowds* is introduced applying some of the ideas explained above. Virtual crowds are steered based on the SERVICES available in the virtual world: they query the Semantics Engine to know how to fulfill certain goals and desires using the objects in the world. Besides Section 10.4, even more information on semantic crowds can be found in [48].

8.4 Discussion

This chapter introduced the notion of SERVICES in our semantic model. SERVICES are built around a simple basic structure of *requirements* and *effects*. In our model, we decided on a number of different types of such requirements and effects, based on an extensive list of examples of current games from many different genres. This is not an exhaustive list, and for some less usual behavior other types might be necessary. Space and time play an obviously important role, therefore the SERVICES have properties to take them into account.

Our main goal was to show that a generic behavior specification (i) allows and encourages creative solutions by the player, (ii) is important for reusability, (iii) is essential in obtaining more spatial immersion, and (iv) is instrumental for coherent and believable behavior of AI characters.

Firstly, a more *generic and reusable* behavior specification is important to *spark creativity* from the player. Instead of coming up with fixed solutions to problems or puzzles in the game world, designers can define the problem and, using the behavioral knowledge of the objects in the game world, players can come up with their own solutions to it. For example, instead of defining ad-hoc that the player can snap a rope holding an obstacle blocking the player's way, by using a torch, a designer could specify that ropes can be cut with a sharp object or burnt using fire, which suddenly opens up an entire array of possible solutions. We claim that such a generic specification for object behavior is an important step towards more *emergent gameplay*, sparking players' creativity and catering a more personal playing experience.

Secondly, a generic specification will aid designers in creating object behavior more *quickly and easily*. Not only are SERVICES reusable among different objects in a single game, it is also quite easy to reuse the information between multiple games. This opens up opportunities to bring object behavior into games where this behavior is not necessarily an important gameplay feature. If it takes a lot of time to specify object behavior, designers will not be eager to do that for, for example, a first person shooter. However, as we will discuss in the next paragraph, this is vital to immerse the player even more into the game world. And since we showed, using our SERVICES, that any SEMANTIC GAME WORLD can reuse behavioral information created for other games, it becomes very easy to bring more detailed behavior into game worlds that would otherwise remain static and non-interactive.

We already mentioned how spatial *immersion* (the experience of playing in a perceptually convincing game world, i.e. when it both *looks* and *feels* real [9]) could be increased and offered more easily by allowing designers to define object interaction and behavior information in a generic and reusable way. We explained before how visual immersion is becoming less and less of a problem because of the huge increase in visual quality of game worlds. However, when the behavior of objects stays behind, the immersion might still be broken: an object that looks exactly as its real-life counterpart, but does not behave like it, feels very awkward, even though many players have learnt to live with such inconsistencies. Using SERVICES to increase spatial immersion will be an extra step in creating believable, immersive playing experiences.

And, lastly, when object behavior becomes more realistic, the behavior of computer controlled *characters* cannot stay behind. When interaction with objects is much more detailed for the player, but the characters around the player do not make use of them in a correct and logical way, the immersion will still be broken. Therefore, we showed in Section 8.3.2 how agent behavior can be extended with SERVICES to find solutions for their desires in the information embedded in a SEMANTIC GAME WORLD. Using objects in a consistent way compared to the way a player uses them is vital to create a realistic effect. Although SERVICES are not the solution to all issues around agent behavior, it can be an important tool to create a closer connection to agent behavior and the game world.

More diverse and detailed behavior of objects and characters, and more options for the player, obviously also have an important downside. A lot of time and money is spent in testing a game to improve robustness. Many open world games often suffer from numerous bugs and more emergent gameplay opportunities will not help developers in their never ending quest to make the game robust and bug-free. Moreover, more complex behavior of the objects might make it more difficult to spot exploits and conflicts. A thorough system to spot conflicts in the object behavior is necessary in order for our idea of SERVICES to be applied in current game development.

Another issue is the question how far one should go in behavioral details. For example, we can simply define that that ACTION **shoot** with the effect **decrease health** can be performed by the corresponding SERVICE between a person (actor) and a living creature (target) using a gun (artefact). However we could make this much more complex by expressing that a person can perform the **pull** ACTION on the trigger of a gun, with the effect to launch a bullet at a certain speed, and that being hit by a bullet decreases the health of a living creature. It is clear that having too detailed behavior is only cumbersome for the designers and does not bring with it more gameplay opportunities. This will come down to common sense of designers and the specific needs for a particular game to make

decisions on how far one wants to go in detailing object behavior.

Furthermore, what is important for one game might not be important for another and vice versa. This could therefore hinder the reusability: e.g. in an action game it might be enough to define that a punch decreases the health of the target, however in a boxing game the effects need to be much more complex: a punch in the stomach will hinder the target's movement while a punch to the head will influence the target's stability and balance. On first sight, this makes this behavior unsuitable to reuse between these two games. We claim that this calls for a *semantic level of detail*. Just like the visual detail of an object will need to increase when closer to the camera, the behavior of an object needs to follow similar rules: the behavior of objects in the player's surroundings will have to behave as realistically as possible, while objects far away do not. And this could prove to be a solution to the problem mentioned above: one could define a maximum semantic level of detail for a particular SERVICE for a particular game. This idea however requires significant attention and is now left as future work.

In Section 8.2 we described the workings of our Semantics Engine to simulate the behavior defined in SERVICES. Since it needs to cater for all possible requirements, effects, spatial properties and temporal properties, and since our proof-of-concept implementation was never optimized, the performance is still below optimal. Since there are a lot of checks to be made in the simulation of the SERVICES, an important improvement could be an event-based engine: only check or change the activity or inactivity of a SERVICE when the corresponding parameters have changed.

However, this will never change the fact that an ad-hoc, optimized solution for a particular behavior will likely be faster than a generic one. This is not necessarily a problem, though: in Chapter 9 we explain the idea of a procedural prototyping tool for games and we will show how, using SERVICES for behavior and procedural modeling for the environments, a quick prototype game can be created in which gameplay can be tested before designers are finished with creating the actual game worlds. In this prototype game, it will immediately become clear if there are performance problems with the generic handling of object behavior, and where exactly these problems arise. At this point it is still an option to create more optimized, ad-hoc solutions for these parts of the object behavior that cause performance issues.

In general terms, it is not always desirable to leave the behavior to a generic simulation system. One such example is the specific behavior and handling of a car. The physics behind this behavior are too complex to be easily specified in our model for SERVICES. It is therefore desired to simulate this in an ad-hoc component of the game code.

However, in spite of the issue above, we made clear that SERVICES, or any other generic specification for object behavior, is unmissable in any game world that needs to be truly immersive and realistic. The reusable setup will make it more easily available in many more games, even when object behavior does not play an instrumental role in the gameplay. Moreover, the wider range of opportunities to players will spark their creativity while playing a game. Equally so for the AI controlled characters or agents that can use the behavioral information to increase the consistency and realism of their behavior.

CHAPTER 9

Pro^2 PROCEDURAL PROTOTYPING

In the previous four chapters, we explained the use of our semantic model for both procedural modeling (Chapters 5-7) and to facility the specification of more complex object behavior in an easy way (Chapter 8). A game development phase that can benefit from both these ideas, is the prototyping phase. When prototyping a game, typically the large majority of the content is either not available or in an unfinished state. This makes it hard to already fully comprehend how certain gameplay elements will feel in the finished product, making the evaluation of the gameplay quite hard as well.

First of all, procedural modeling can be a useful way to create highly detailed and large game worlds to start testing gameplay right from the start of the development process. These procedurally generated worlds might even serve as the starting point for designers to build the eventual final version of the game world on.

Secondly, the simplified, generic and reusable method of specifying complex object behavior by means of SERVICES, will help developers in testing and prototyping the behavior of objects and to test the impact of that behavior on gameplay.

To test this idea, we created the Pro^2 (Procedural Prototyping) environment. This environment contains a number of editors we implemented to specify semantics and to use the semantics in the procedural modeling ideas explained in the previous chapters:

- The Entika editor allows designers to specify all semantics from our semantic model, generic and reusable for all games. In a *game-specific* edition of the editor, designers can create references to game content specific to a particular game in the generic semantic entities.

- In the semantic scene description editor, one can create descriptions for a particular scene that can be used to automatically create layouts.

- The filter editor was designed to create and edit procedural filters (see Chapter 7).
- Next to combining these editors (which can be used separately as well), the prototyping environment also allows designers to create plan scripts (as explained in Section 6.1.3) to use the integration technique for procedural generation methods using a semantic moderator.
- A level editor enables designers to generate a particular level, using the components created in all the above editors as parts.
- A plugin to open terrains created with the *SketchaWorld* environment allows SketchaWorld terrains to be used as the basis for a level. SketchaWorld is a declarative modeling framework developed by Smelik et al. [85, 80].
- Finally, a testing environment allows designers to playtest a level and to store data from the playthrough.

In this chapter, we will first give an overview of the environment and the importance of procedural modeling and complex object behavior using SERVICES in this idea. We will also give a detailed description of some of the more important editors and components on the previous list. Finally, we will show an example of the use of the prototyping environment: prototyping a 3D city-building game.

9.1 Semantic procedural prototyping environment

The goal of the prototyping environment is to show how the combination of semantics and procedural generation can improve the prototyping process of a game. The environment allows the users to test one or more levels of a game, generate data and feedback on the playthrough for the developers to analyze and alter the level or the behavior of the game accordingly.

This level can be built using procedural generation components discussed in the previous chapters: automatically layouted scenes based on descriptions, plans to generate structures using multiple techniques by using the semantic moderator and procedural filters to change the appearance of virtual worlds. Also SketchaWorld maps can be loaded and used to produce such levels. Next, we will give more details on this setup.

The Pro^2 Procedural Prototyping environment uses *solutions*, similar to e.g. Visual Studio. This solution contains all related files to a single game prototype: the content for that game, the specified semantics, the files that describe the levels for the game and logs of playthroughs of these levels. This is a list of all the file types ordered into three categories:

1. Content files:

   SketchaWorld project files The basis for the game world one is trying to prototype can be a terrain created by SketchaWorld. The project file contains all information about that terrain. More on this in Section 9.2.4.

Procedural filter files Procedural filter files are meant to alter the look of a particular part of the game world. In Section 7.1, we give an overview of such filters.

CGA grammar files Generating procedural objects can happen using shape grammars. We chose to use CGA grammars, created by Müller et al. [63], since we already had an implementation to test our consistent buildings idea from Chapter 6.

Material files Material files are a custom format for materials (including textures, colors and shaders) that can be linked to semantic MATERIALS.

Procedural plan files To integrate procedural techniques using our semantic moderator, a plan needs to be created that guides this process. How such a plan looks like can be found in Section 6.1.3.

Scene description files The game world, or parts of it, can be created using semantic layout solving. Scene descriptions contain information about how a scene should be created: the entities present in the scene and scene-specific relationships. In Section 5.2.2 we described in detail how the language for such descriptions looks like.

2. Semantics specification:

Generic semantic database The first, and most important, file of the semantics specification is the generic database. This is a SQLite database containing all generic semantic information, reusable over many game projects.

Game-specific semantic database For each game, a specific database, separate from the generic one but also in the SQLite database format is stored. In here semantics specific for that game is stored as well as references to the content (e.g. 3D models, sprites or any of the content files from the above list) in the ENTITY CLASSES.

3. Game files:

Level file We created our own level format to store how each level of the game world is structured. The difference with ordinary game level formats is that it does not contain direct references to content (e.g. a 3D file). It contains instantiations of ENTITY CLASSES, positioned and rotated into that level. Through the references specified in the game-specific semantic database, content can be found to actually populate the game world when running the game.

Game log files Once all (temporary) content is created, semantics is specified and some levels are created, designers can start playing the game. Information about each playthrough is stored in game log files. In there, they can track whether or not everything is working as they hoped and, of course, try to find missing or broken gameplay elements that need to be solved in the finished product.

These three file types are related to the three main steps involved in prototyping a game using the Pro^2 environment: (i) creating new content for the game, or placeholder files for unfinished content, (ii) specifying the semantics of the new game, and (iii) testing the game.

The first step is the creation of the procedural *content files*. When designers are not finished with some of the necessary content, which is highly likely in the starting phases of a game's development, *placeholder art* needs to be created. Procedural generation is an ideal solution for this. A quickly generated game world with terrain and cities using SketchaWorld projects, complete structures and automatically layouted scenes using plans and scene description files, or procedurally generated objects using shape grammar files are perfect to populate a prototype game world in a fast and easy way.

Obviously plain boxes could be used for this, however this will hinder the level designers in getting a representable idea of the game world. For example, the visibility of the terrain or other objects might be significantly different when using boxes versus more detailed (procedurally generated) geometry.

The second step is creating the *semantics* for the game that is being prototyped. The fundamentals are taken from a generic semantics database. This includes all reusable semantics created in previous projects. Potentially, this generic database needs to be extended based on the needs of the new game.

The semantics specific to this game only, are stored in the game-specific database. This database will be the main work area during the prototyping. The Entika editor created for this specification is explained later in Section 9.2.1. GAME-SPECIFIC CLASSES include references to content in order to specify what the actual geometry of such entities is. Obviously this can be already made 3D models or sprites, but the files created in the first step can just as easily be referenced as content in a GAME-SPECIFIC CLASS.

The next step is creating a *level* using the ENTITY CLASSES created in the second step. By creating an instance of these ENTITY CLASSES, and giving a location and rotation to the instances, a level can be built up. Since references to the content exist in the GAME-SPECIFIC CLASSES (also created in the second step), everything is now in place to create and visualize the game world.

The level is now ready for testing. As explained above, numerous data about the playthrough is stored and can be checked out while playing the game, or afterwards.

After the created level is tested, designers can make a good evaluation based on their experience and the logged data. Now a next iteration in the prototyping process can begin: new content files can be added, content files can be edited or refined, behavior of objects can be adjusted in SERVICES, new ENTITY CLASSES can be created or old ones can be modified and levels can be modified. After that, the game is ready to be tested again. Iteration after iteration, the gameplay can be improved and, long before finished content is available, plenty of gameplay testing can already be performed and evaluated.

A screenshot of the editor can be found in Figure 9.1. Navigating to all the files of the current solution happens by either selecting them in the menu bar on the left (see Figure 9.2), or by clicking on the tab bar listing all the currently opened solution files (see Figure 9.3). Notice that in the menu bar, the actual semantic database files are not shown. Instead shortcuts to user-chosen semantic concepts are shown to allow quick navigation. Once a file is selected, it is shown to the right in the proper editor. For example, in the screenshot a kitchen description is shown in the scene description

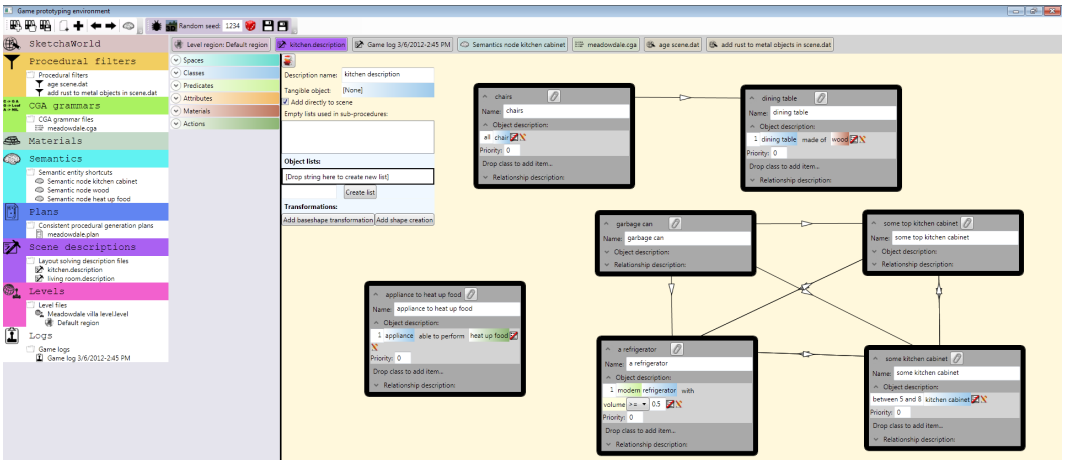


Figure 9.1: A screenshot of the Pro^2 environment. To the left is a menu with all the solution files (see Figure 9.2), to the right the selected file shown in the appropriate editor, in this case a scene description (see Figure 9.5). On the top is a toolbar and a *tab bar* with the currently open files to quickly switch between (see Figure 9.3).

editor (see also in Figure 9.5). In a separate window, a 3D engine is running in which previews of the content can be shown (e.g. a preview of a CGA grammar one is editing) or the entire game. Ideally, during the testing of the game, a second screen is used for this window as it allows users to play the game world while keeping an eye on the logged data.

We will give a concrete example scenario of the use of our prototyping environment later in Section 9.3.

9.2 Combining design phase applications of semantics

For some of the above mentioned files, contained in a prototyping solution, we implemented separate editors. The next section gives an overview of the implementation details and the features of these editors. We will also give an overview of the SketchaWorld framework and how it was integrated into the prototyping idea.

9.2.1 Entika editor

The Entika editor makes it possible to edit semantic databases. The idea behind this editor is to specify new components (including ENTITY CLASSES, SUBSTANCES, ATTRIBUTES, and ACTIONS), and modify or remove existing ones, where each type of component is stored in its own library. To do so, the editor provides the user with an overview of the available libraries, and the components that populate them. Each component can be specified in great detail, meaning that its semantic information

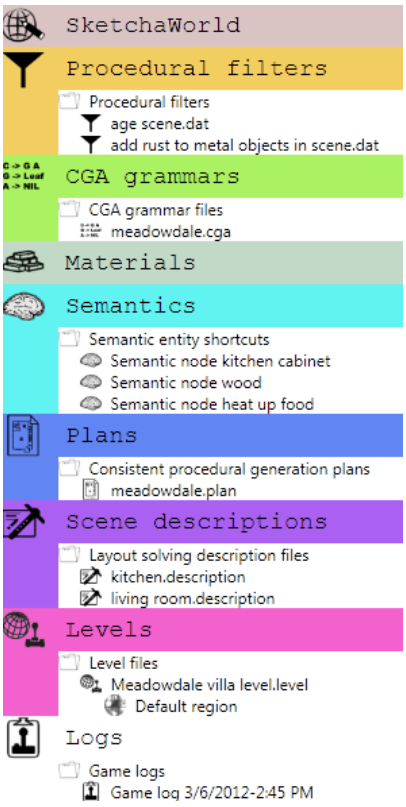


Figure 9.2: The menu of the Pro^2 environment giving an overview of all files in the currently opened solution.

Figure 9.3: The tab bar of the Pro^2 environment. It allows for quick navigation between open solution files.

can be fine-tuned at will, ranging from the name and description, to RELATIONSHIPS between that component and other components. PHYSICAL OBJECT CLASSES, for example, can be equipped with ATTRIBUTES. Furthermore, it is possible to specify the values and quantities that are required for some of the relations. Above all, SERVICES can be defined, with their requirements, ACTIONS, temporal properties, and spatial properties. Derived classes will inherit the semantic information (including ATTRIBUTES and SERVICES) from their parents, although specific values can be overruled if necessary. When, for example, the PHYSICAL OBJECT CLASS superclass is assigned the *mass*

attribute, each underlying class will inherit this attribute, but the specific mass value can be modified for each of them.

In addition to generic and reusable components, GAME-SPECIFIC CLASSES can be created. Besides customization of inherited semantics, GAME-SPECIFIC CLASSES can be further customized with content, e.g. with geometric models, textures, and audio. For each of these extra properties, preconditions can be specified to indicate when they have to be used. For example, only when a radio is in the 'on' state, it should play music. Inventories can be specified too (think of chests, bottles, and jackets with pockets that can contain items), just like parts and the offset with respect to their whole (think of a silencer that can be attached to a pistol). This can be annotated on the 3D models, allowing parts, possibly having SERVICES of their own, to be detached from their whole during the game, and attached again on their correct position.

The end-users of the Entika Editor are game designers, and to make the design of semantic objects for them as easy as possible, usability was one of the aspects that were aimed for. In the editor, this has been achieved by providing a clear and distinctive overview of all information (e.g. different colors for different components), the possibility to hide unwanted information, and providing user-friendly ways to quickly establish and remove relations (e.g. drag and drop), and edit other semantic information. Furthermore, due to inheritance, many relations only have to be defined once. We did a user evaluation of the editor, which can be found in Section 11.2.2.

Figure 9.4 shows a screenshot of the Entika editor. On the left, all libraries are displayed; on the right, some of the semantics of the selected TANGIBLE OBJECT CLASS *human being* are shown. The user is able to modify its names and description, and observe a list with its parents and children. Because of inheritance, the human being has a *health* and *mass* ATTRIBUTE, defined at one of its parents. In addition, the human has an ATTRIBUTE of its own, *hunger*, having a default value of 10, and ranging from 0 to 100.

9.2.2 Semantic scene description editor

We created an editor using the semantic scene description language explained in Section 5.2.2. The editor is a simple canvas where description entities are displayed. To the left of this canvas, a list is present with all semantic components (ordered by type, e.g. TANGIBLE OBJECT CLASSES, ATTRIBUTES...). A screenshot of the editor can be seen in Figure 9.5.

As explained in Section 5.2.2, the basic element of this language is the description entity that contains two elements.

- Which entities need to be available or could be available in the scene.
- How these entities should be placed in this particular scene.

In the editor, these elements can be expressed in the description entity blocks (see Figure 9.6). Creating such an entity block happens by dragging a TANGIBLE OBJECT CLASS to the canvas. Editing the blocks happens by either dragging other semantic entities to the description entity block,

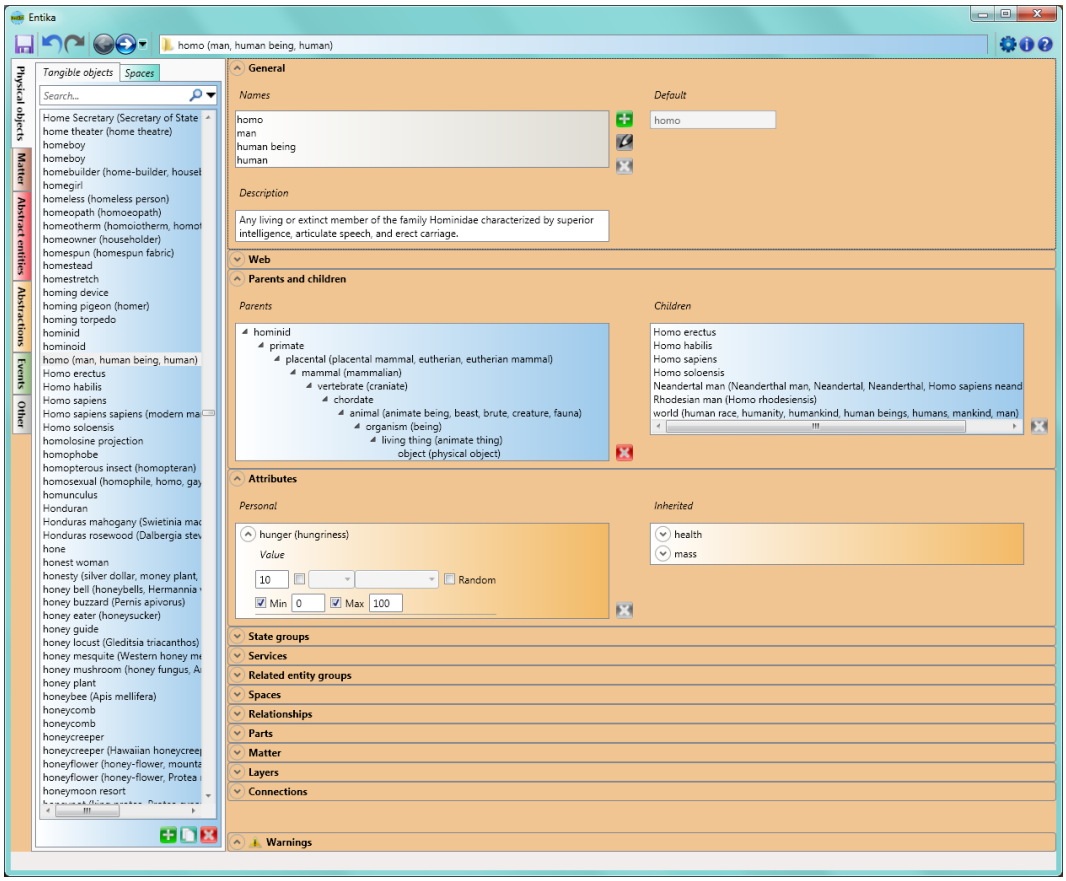


Figure 9.4: A screenshot of the Entika editor.

or by editing textboxes on this block. For example, in Figure 9.6, we see an entity block that signifies that we want to place one refrigerator. This was extended by dragging a PREDICATE **modern** and the **volume** ATTRIBUTE to the block. For this ATTRIBUTE, we also changed the condition to say greater than or equal to 0.5. This way, we described that in this particular scene, we want the layout solver (which automatically generates scenes based on the descriptions created using this editor, see Chapter 5) to place exactly one modern refrigerator with a volume of more than half a cubic meter.

The context-sensitive nature of descriptions was handled in the editor as follows. CONTEXTS can be added to the description. When none of the added CONTEXTS are selected, all changes made in the description canvas will be made regardless of the CONTEXT. Once the user selects one of the added CONTEXTS, changes in the canvas are only applied for that particular CONTEXT. Switching to another CONTEXT, will make these last changes disappear from the view. For example, we can

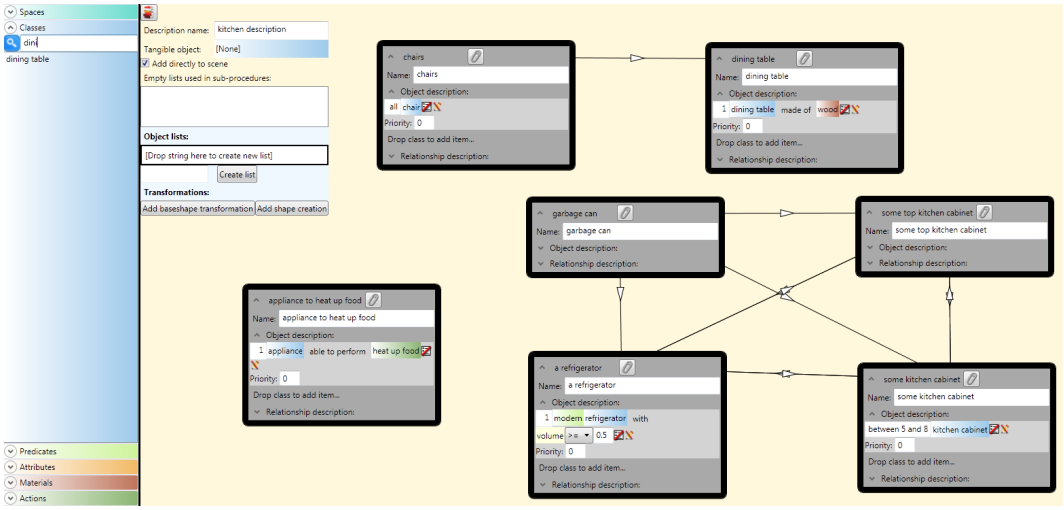


Figure 9.5: A screenshot of the scene description editor showing a description to create a kitchen. To the left, the lists of semantic concepts (TANGIBLE OBJECT CLASSES, ATTRIBUTES...) and to the right the description entities in the gray boxes with black border. A bigger version of the refrigerator entity in the screenshot can be found in Figure 9.6

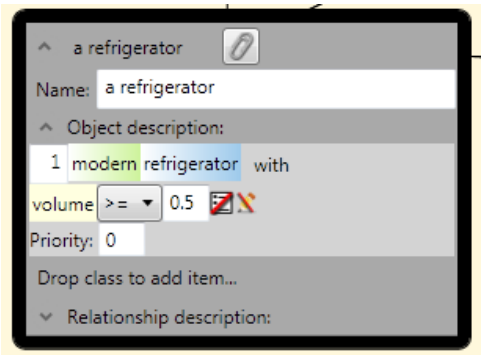


Figure 9.6: A close up of one of the scene description entities from the kitchen description shown in Figure 9.5. The refrigerator entity is built up of *one* instance of the TANGIBLE OBJECT CLASS **refrigerator**, having the PREDICATE **modern** and having a value for the **volume** ATTRIBUTE greater than or equal to *0.5* (cubic meters).

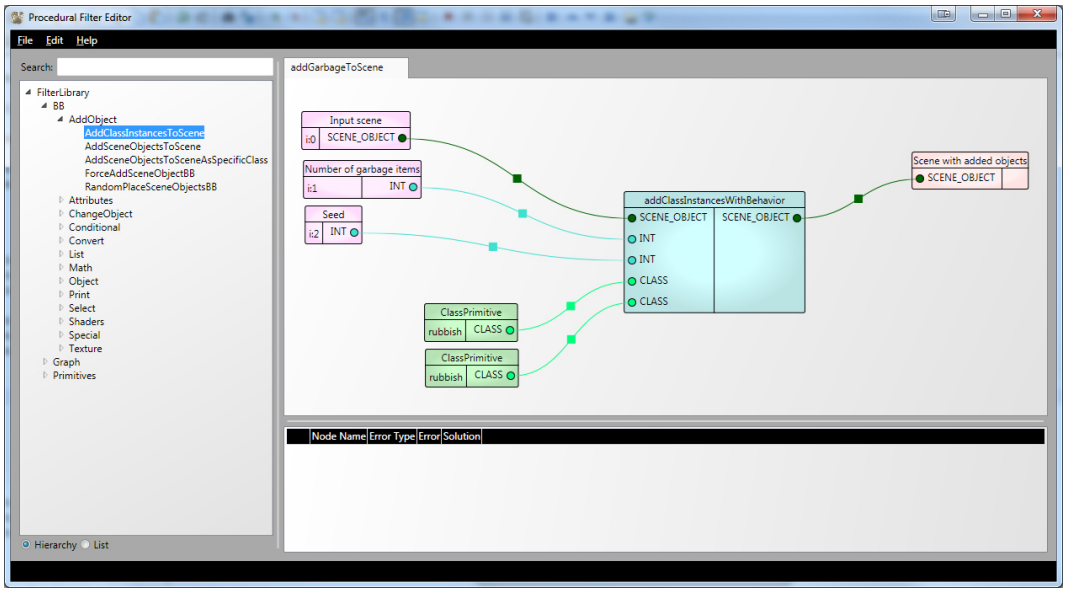


Figure 9.7: A screenshot of the procedural filter editor showing a filter to fill the scene with rubbish.

add the CONTEXT **large** (which e.g. contains a minimum condition on the total area of a room) to a kitchen description. With this CONTEXT selected, we might apply the following changes to the original description: increase the number of kitchen cabinets or increase the value of the condition on the volume ATTRIBUTE of the refrigerator.

9.2.3 Procedural filter editor

Based on the procedural filter approach introduced in Chapter 7, we developed a 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 (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 a TANGIBLE OBJECT CLASS from our semantic model. We use our model to handle semantic query instructions explained in Section 7.1.2.

While many TANGIBLE OBJECT CLASSES may prescribe some geometric model(s) for its in-

stances, many other TANGIBLE OBJECT CLASSES rely on procedural generation methods to create the geometry for their instances, as e.g. the consistent buildings generated with our integrated approach, which is explained in Chapter 6. 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 (see Section 7.1.1 for an overview of all instruction categories with an explanation), 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 use the semantic layout solving approach explained in Chapter 5. It can be used in building blocks to add objects of a particular TANGIBLE OBJECT 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.

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 [55] or Filter Forge [38]. 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 9.7. The filter shown in the screenshot has 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 interrupted, 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. A screenshot of this viewer window is shown in 9.8.

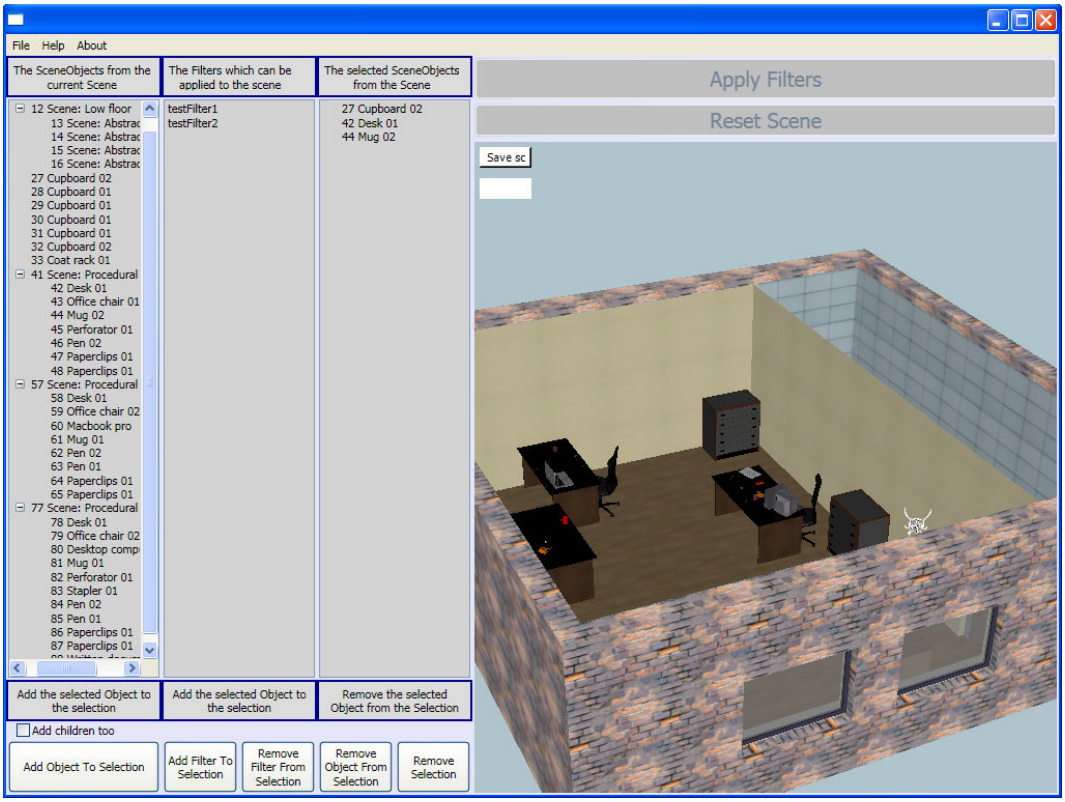


Figure 9.8: The viewer window where designers can select elements from the scene and apply filters to them or to the entire scene at once.

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.

9.2.4 Integration with SketchaWorld

Smelik et al. [85, 80] created the SketchaWorld framework. This declarative modeling framework allows users, whether they are experienced virtual world modelers or not, to create highly-detailed game worlds, including: (i) terrain, (ii) vegetation, (iii) rivers and road networks, (iv) cities, and (v) buildings, all based on a simple sketch. In a matter of minutes, such a sketch can be made, and SketchaWorld immediately generates a complete and detailed virtual world.

This is done by combining and integrating multiple complex procedural content generation techniques, including the automatic resolution of conflicts between different types of elements, e.g. adding a bridge where a river and a road cross each other.

After this world is created in the SketchaWorld editor, the entire project can be saved. These project files can be loaded into our Pro^2 environment and can form the basis of a game level.

An important aspect of SketchaWorld terrains is that they are too embedded with detailed semantics. Since SketchaWorld uses the concepts developed in our semantic model, the terrains fit in nicely in our semantics-driven prototyping environment. By assigning an ENTITY CLASS to the geometry and the elements placed by the SketchaWorld framework, these terrains can immediately be extended with semantics like complex behavior through SERVICES.

9.3 Example: prototyping a 3D city building game

To show the workings of the Pro^2 environment, as well as the advantages of using our unique mix of semantics and procedural content generation in the prototyping process, we will go through a hypothetical user scenario. In this scenario, the user wants to create a simple 3D city-building game in the line of the Sierra Entertainment *City Building Series*, e.g. **Caesar I-IV**, **Pharaoh** or **Immortal Cities: Children of the Nile**. The main activities of the player in these games are placing buildings that can either gather resources for the player, or that transform these gathered resources into new resources.

The first step in creating this prototype game is adding the first building, which is the woodcutter's hut. Typically this building gathers wood for the player. We have a CGA grammar for exactly such a building, so we add it to the Pro^2 solution. A view of the generated building can be seen in Figure 9.9.

To add the functionality of the building, we create a TANGIBLE OBJECT CLASS with the name **woodcutter's hut**, which inherits from the TANGIBLE OBJECT CLASS **building**. In the first version of the prototype game, we would like to keep it as simple as possible. Therefore we want to define that the building increases the wood resources of the player. This means we also need a player entity. We choose to specify an ABSTRACT ENTITY CLASS named **player** for this purpose, since we have no intention of creating a physical representation of the player in the game. An ATTRIBUTE named **wood supply** is created and added to this ABSTRACT ENTITY CLASS **player**. To create a connection between the buildings and the player, we create an **owns** RELATIONSHIP with the **player** as *Actor* and **building** as *Target*.

Figure 9.9: A woodcutter's hut created from a CGA grammar.

Now we specify the SERVICE to gather wood.

- In a new ACTION, which we name **gather wood**, we define an automatic event with the **woodcutter's hut** as the *Actor*.
- The effect of the ACTION is increasing the **wood supply** ATTRIBUTE of the *Source* of the **owns** RELATIONSHIP where the event's actor (in this case, the woodcutter's hut) is *Target*.
- The temporal properties of the event are discrete and infinite with an interval of 1 second.

To test this, we add a test level with one woodcutter's hut and an instance of the ABSTRACT ENTITY CLASS **player** which we name *jefke*. In the log, we can now see the continuous increase of jefke's wood supply, shown in Figure 9.10.

Now we can start experimenting with this wood supply service. Having a constant and infinite supply of wood by a woodcutter's hut is not really interesting from a gameplay perspective. Therefore we can make it more complex. For example, we might try to use tree instances in the game world. Therefore we need to perform the following steps in the prototyping environment.

- Add TANGIBLE OBJECT CLASS **tree** to the semantic database (if not yet present) and add some tree instances in the test level around the woodcutter's hut.

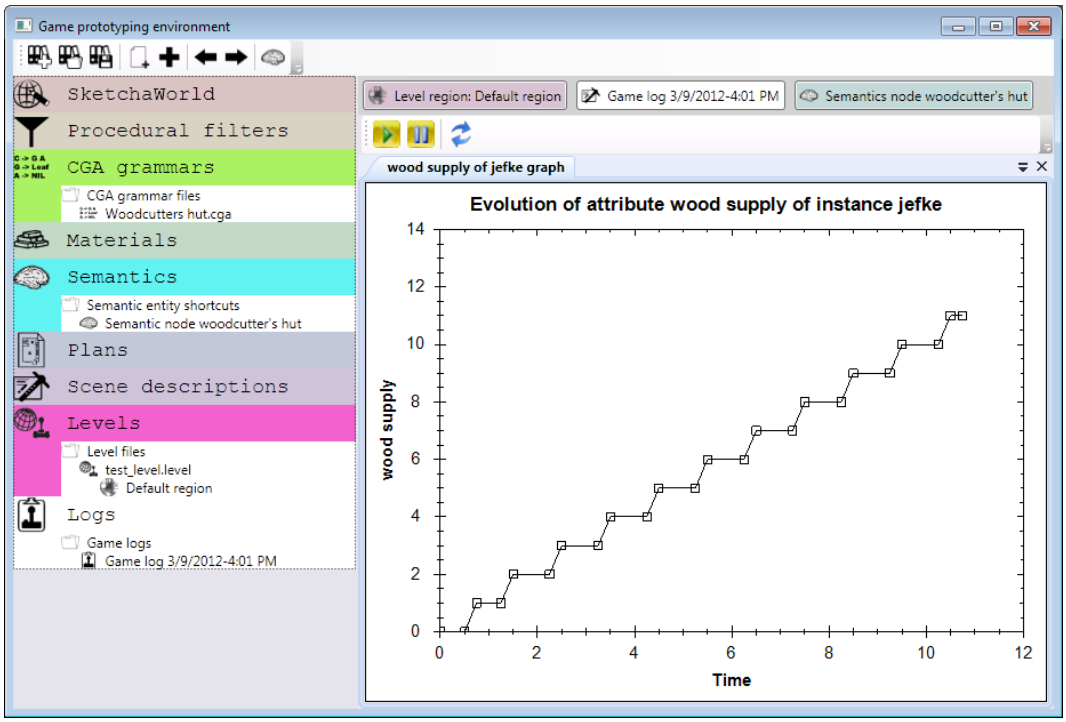


Figure 9.10: The woodcutter's hut in the level increases the wood supply of player 'jefke'. The figure shows the graph in the game log representing this.

- Create a precondition in the gather wood event that a tree needs to be available within a certain range.
- Add another effect for this event to remove a tree instance.
- Increase the amount by which the wood supply is increased.
- Increase the interval between the executions of the event.

Another factor to experiment with could be increasing the wood supply based on an ATTRIBUTE **size** of the tree. All of this can be quickly specified using the Entika editor and equally quickly tested. No code or scripts need to be created.

Because of the wide range of procedural solutions, we can quickly create a new CGA grammar for a farm building by reusing the woodcutter's hut's grammar, change some textures and add some rows of small plants, see Figure 9.11. This is obviously an easy way of creating a representable model for

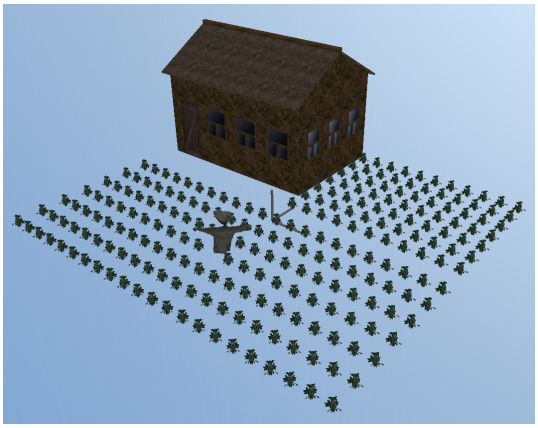


Figure 9.11: A farm house, created using the woodcutter's hut's grammar by quickly changing some textures and adding some plants.

a farm and other buildings like it, without having to perform any modeling or waiting for designers to create the content.

Using a scene description and the semantic layout solving approach, we can easily create ranches which are moderately complex layouts that include some pastures, a stable, a house, some beehives and some background items like barrels, hay bales or cribs. Or it could be used to generate interior layouts in order to experiment with using the houses' interior as a manner of visualizing the resources it has in stock, as was done in 2K games' **CivCity: Rome** (see Figure 9.12).

To test the gameplay principles under different circumstances, we could use SketchaWorld maps. In SketchaWorld, users can quickly change their terrains by changing some parameters or adding or removing some sketch elements. This way we could quickly test our 3D city building game in terrains with dense forests as well as in barren terrains with sparse trees. Or we could test it in lush grasslands, in arid deserts or in the mountains. In a matter of minutes, completely different terrains can be generated to really test the gameplay of our prototype game under the most diverse circumstances, allowing taking gameplay into account when coming up with the locations in which the game will really come into its own.

Similarly to our workflow for the woodcutter's hut, the user can now experiment with services for all other buildings as well. Long before final models (or final grammars for procedural generation) are finished, the behavior of the buildings in our city building game can be tested and searched for gameplay flaws, while already getting a representational 3D view of the result.

When production continues on the game and the actual content for the game is created, it is very simple to replace the placeholder content by their finished version. One only needs to change the referenced content of the GAME-SPECIFIC CLASSES and the next time the prototype game runs, the new content will be present.

Figure 9.12: In the game **CivCity: Rome**, the player could look into the interiors of residential and commercial buildings to notice which resources the building has in stock. For example, in the bottom right building, three jugs of water on the floor, two jars of olive oil on a wall rack and three pieces of meat hanging from that rack. (Screenshot source: http://oriongames4u.blogspot.com)

9.4 Conclusions

In this chapter we discussed how the combination of semantics and procedural content generation could mean an important boost to the prototyping phase of game worlds. To show this, we created the Pro^2 (Procedural Prototyping) environment. We showed how it worked and which editors we integrated into it and we gave a hypothetical example scenario for a 3D city-building game.

The major advantages of using both semantics and procedural content generation to quickly prototype the look and the behavior of a game world are the following:

- Without writing any code or scripts, SERVICES can be specified to test complex object behavior.
- Before any content is finished, designers are already capable of testing important gameplay

mechanisms.

- The testers will be able to give feedback on the general structure of a level or game world before creation of the actual level or game world is started, by using a procedurally generated game world with placeholder content.
- Once designers are finishing the actual content for the game, they can easily replace the placeholder content in the prototype.
- Even from a very early stage, testers can get an idea of the game in a more or less populated and semi-finished world instead of an abstract, empty test world.

By combining the easy specification of behavioral semantics with procedural content generation, prototyping game worlds could become significantly easier and faster. Using procedurally generated game worlds to test gameplay created using SERVICES and all other elements from the semantic model, flaws in the gameplay are spotted early in production. More importantly, feedback on a level's structure can be given to the designers of that level as early as possible. Step by step new content will replace the placeholders and the testing experience will become more and more representational of the final result.

CHAPTER 10

APPLICATION OF SEMANTICS IN EXTERNAL PROJECTS

Chapters 5 to 9 introduced applications of our semantic model created in our research project. Next to these, several external collaborations were undertaken to show other uses of semantics for game worlds. This chapter will introduce these.

The first application was a collaboration with the Dutch company *re-lion*. A knowledge transfer project allowed us to combine our semantic model with re-lion's virtual world editor *Builder* to create a semantic search system to easily find the correct 3D model.

A second application is *procedural infrastructures*, which is an approach to generate large-scale infrastructures, such as airports, automatically using RELATIONSHIPS between different areas and sub-areas of the wanted infrastructure.

To facilitate the specification of semantics, an approach was created to quickly specify semantics for large sets of 3D models. The user can choose training models for particular TANGIBLE OBJECT CLASSES, and using shape recognition and segmentation, other 3D models will be assigned the correct class automatically.

Two MSc students have used the semantic model in their Master thesis projects. Nick Kraayenbrink's thesis on *Semantic crowds* introduces a semantic approach to steer the behavior of crowds. Rick de Ridder's thesis on *simulating urban area development for semantic game worlds* created an approach to generate urban areas by simulating the history behind that area.

10.1 re-lion Builder: Semantics used in a serious games development tool

re-lion BV is a Dutch company from Enschede that, among other activities, focuses on *terrain database generation tools*. One of their main tools is *Builder*, as is shown in Figure 10.1. Builder is a tool that allows a user to easily model a particular terrain. This includes both geospecific environments (constructing a real area based on aerial photographs and height and vector data), and geotypical environments. For these last environments, the resulting terrain only needs to look similar to a certain region, e.g. having clear characteristics of a Middle-Eastern region.

A concept unique to Builder is the fact that the terrain is not edited at the level of separate meshes, but using so-called *building blocks*. These are elements like houses, bridges or roads that can be manipulated as a whole.

The amount of building blocks is quite large, and therefore a powerful search mechanism was added using our semantic model. When the user performs a search, a semantic library, filled with a huge amount of concepts and relationships, is queried. Search terms are found in the semantic library and using hierarchical and other relationships all related concepts to the search terms are found. These related concepts are all used to find additional building blocks, others than just the ones containing the exact original search terms. This results in a more complete list of building blocks and therefore the user will almost never miss out on a suitable building block because of too strict search terms or because he does not know the exact term used when adding the building block.

When new building blocks are added, the same approach is used to give suggestions to the user for additional tags or names for the building block. This, again, will make finding suitable building blocks easier.

In the near future, Builder will use the available relationships to find suitable building blocks to fill an area for which only GIS or other vector data is present. When a lot or an area of the terrain needs to be filled with a particular building block, the semantic library is used to find building blocks that fit the annotations from the GIS data. An example might be: a lot is marked *building*, while the encompassing region is marked *agricultural*. Since the term *agricultural* is related to *farm* and *farm* is marked in the library as a type of *building*, Builder could now decide to place a *farm* building block, instead of just choosing any of the building blocks marked with *building*.

10.2 Procedural infrastructures

In Chapter 5, on *semantic layout solving*, was explained how RELATIONSHIPS between PHYSICAL OBJECT CLASSES were used to automatically create layouts, e.g. furniture placement in a room. The same idea can be applied to position certain areas within large infrastructures such as shopping malls, airports or train stations. We collaborated with our colleagues Fernando Marson and Soraia R. Musse from the *Pontificia Universidade Catolica do Rio Grande do Sul* in Porto Alegre, Brasil to create the idea of *procedural infrastructures*.

To create the specification for a particular infrastructure, the vocabulary of our semantic model is used. A designer first creates a set of constraints for each of the areas (represented as PHYS-

Figure 10.1: A screenshot of re-lion Builder (top) and of a terrain created using this tool (bottom).

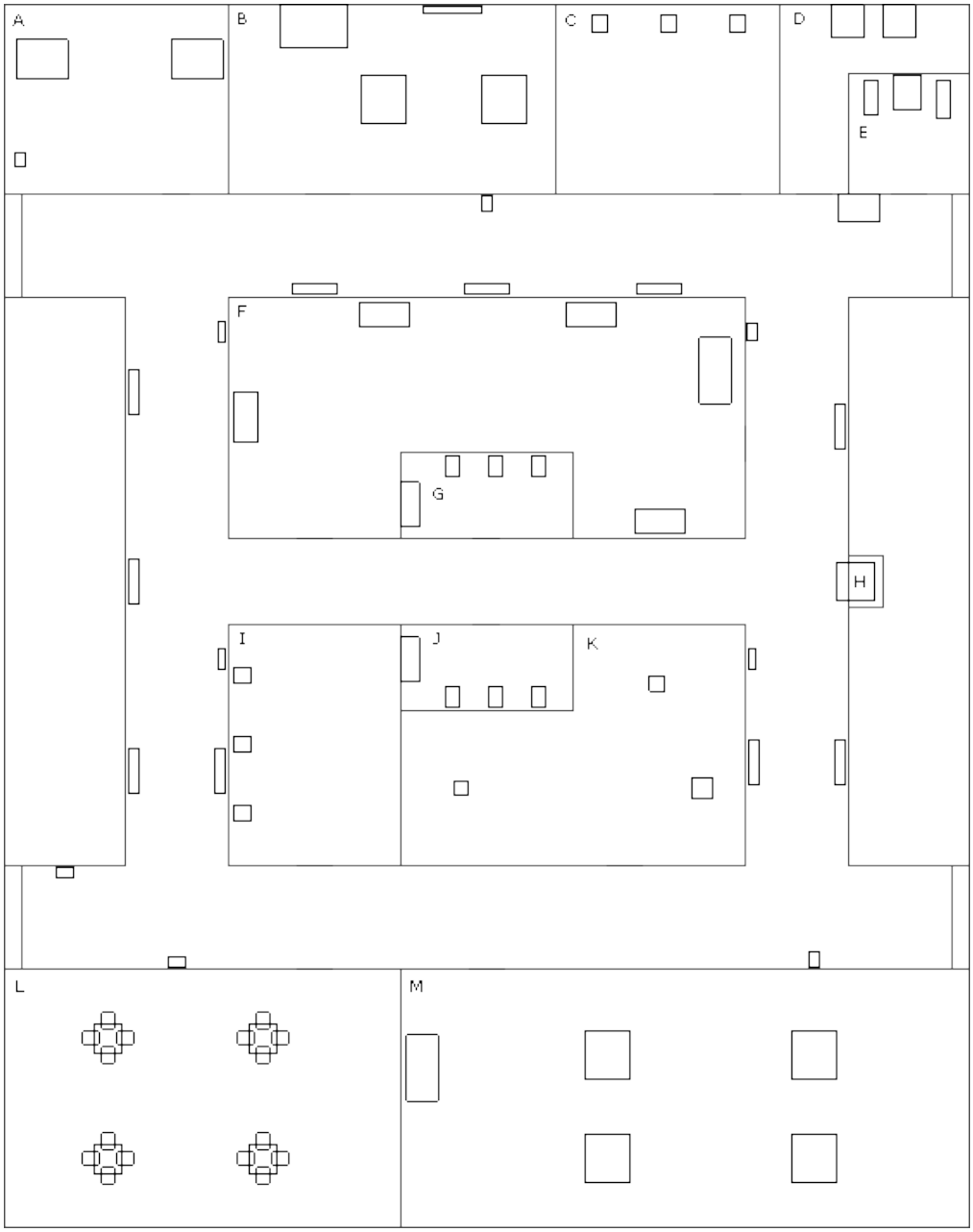


Figure 10.2: The area of an airport before customs with hallways, shops and a bank.

ICAL OBJECT CLASSES) that need to be placed inside the infrastructure. These rules can be on the ATTRIBUTES of PHYSICAL OBJECT CLASS, e.g. on the dimensions of the area, or can be RELATIONSHIPS with other areas, e.g. restrooms should be placed inside a waiting area. Additional ATTRIBUTES can define how many of such areas need to be available, e.g. at least 1 restroom for every 500 square meters of waiting area.

Using this specification, the algorithm starts constructing the infrastructure. The algorithm is a subdivision algorithm: given a shape as the boundaries of the infrastructure, it will produce subdivisions in which the different areas are positioned. First rough locations are found for each of the necessary areas, based on the constraints defined in the specification, i.e. central positions for the areas without defining area sizes or a definitive shape yet. Now the straight skeleton algorithm is used to create the path for the hallways in between the different areas. Now this path is offset with a width depending on the circumstances, e.g. in an airport, hallways towards the airplane can be much narrower than the hallways in between shops. The remaining area can now be subdivided further based on the previously found rough locations. Once the areas are subdivided, our semantic layout solving approach can be used to further fill the areas with objects, if necessary.

In Figure 10.2, an example infrastructure is shown of an airport before customs. On the top is a hallway with benches on one side, and several establishments on the other side, including a bank (A), hotel (B), cafe (C), a concealed room with plum trees (D), and an arcade hall (E). An ATM is also present in the hallway, in the façade of the hotel. The central block contains a toy store (F), toilets (G for males, J for females), juice shop (I) and an art gallery (K). A fountain can also be found close by (H). The bottom hallway contains a burger restaurant (L) and another hotel (M).

10.3 Specifying semantics for large sets of 3D models

As we mentioned before, specifying semantics should not become a cumbersome task for designers on top of many other tasks they already need to perform. Therefore we need to limit the workload of this specification to a minimum. In collaboration with our colleagues Xin Zhang and Rong Mo from the Northwestern Polytechnical University, in China we proposed a framework to specify semantics of large sets of 3D models that will help to minimize human involvement in this process. The framework consists of three modules: classification, segmentation and annotation. We associate a few models with tags representing their TANGIBLE OBJECT CLASSES and classify the other models automatically. Once all models have been classified in different groups, we take a certain number of models as template models in each group, and segment these template models interactively. We then use the segmentation method (and parameters) of the template models to segment the rest of the models of the same group automatically. We annotate the interactively segmented parts and use a graph to represent them. Automatic annotation of the rest of the models is then performed by subgraph matching. This workflow is presented in Figure 10.3.

In the first step, the classification, the user needs to choose appropriate training models. Training models have a large impact on the classification result. Less training models would result in a bad classification result. However, too many training models need lots of tedious work and are not suitable for large sets of 3D models. The best training models should spread over all the ranges of 3D models.

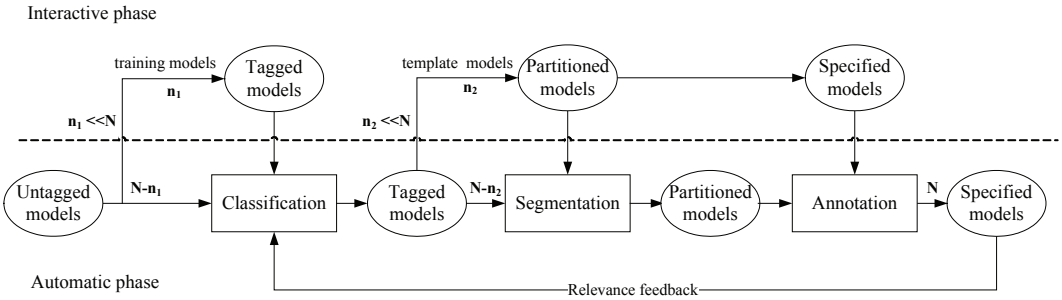


Figure 10.3: The framework of our method for specifying semantics of 3D models.

Therefore, we rely on the k-means method to cluster all the 3D models and choose several models for each TANGIBLE OBJECT CLASS present in the set of 3D models. We use these tagged models (tagged meaning: assigned to a TANGIBLE OBJECT CLASS) as training models. Then, we classify all the remaining untagged models based on the training models and associate these untagged models with the TANGIBLE OBJECT CLASS of the tagged models in the same class. In the end, the untagged models are specified. The specified models are further used to improve the classification.

In the second step, the user can choose a segmentation technique and parameters for each of the TANGIBLE OBJECT CLASSES. Using one technique to perform segmentation for all models is unrealistic, since no techniques exist that perform well for all types of models. For example, segmenting a man-made object or a freeform model like a model of a human being or an animal is vastly different. Once the segmentation technique and parameters are chosen for each TANGIBLE OBJECT CLASS, they are applied to template models chosen by the user (these can be the same as the training models for the first step, but this is not necessary). Choosing the template models wisely is obviously an important factor towards good specification results. When there are a number of different model shapes in the same class, the template models should not all be models of the same shape, but instead reflect the variability of the shapes. A simple example might be the table class. A set of models might contain a number of round tables and a number of rectangular tables. For the best result, both round and rectangular tables need to be chosen as template models to guarantee optimal results.

Once a template model has been segmented, the user manually annotates some model parts, again using the TANGIBLE OBJECT CLASSES (for each TANGIBLE OBJECT CLASS parts can be specified in our semantic model). A graph is defined to represent the annotated model parts, where each node denotes a model part and an edge keeps the geometric relationships between two model parts. The automatically partitioned models are also represented by graphs. Now, the automatic annotation of all other models can be accomplished by subgraph matching, finding a mapping between the graph of the template model and the graph of the automatically partitioned model.

A more detailed explanation of this approach can be found in [101]. The method can achieve a good

rate of automatic annotation of 3D models with limited user interaction, especially for the 3D models that can be separated into consistent parts. This is an important step towards bringing semantics into the game development process, since no developer will want to spend a lot of time in specifying semantics for their entire collection of 3D models. This approach is a significant improvement in the time it takes to perform this specification.

10.4 Semantic crowds

In the semantic crowds project, which was the MSc project of Nick Kraayenbrink, the goal was to extend current techniques to steer crowds by using semantics available in our semantic game worlds (see Section 8.3.2). The proposed semantic crowds approach is subdivided into two major parts: the crowd and the agent. The crowd model contains the global composition of a crowd in different demographics and the ratio in which they are present. For each of these demographics, a semantic agent model describes their specific behavior.

The semantic crowd model consists of two independent components: the crowd profile and the crowd socket. The crowd profile defines all environment-independent aspects of the crowd. It contains:

Demographics These are groups of agents that have something in common, e.g. children, parents or staff members.

Demographic slices Descriptions of non-overlapping demographics that cover at most the entire crowd. One can think of this as the distribution of agents in demographics when a cross-section of the crowd is made.

Crowds A crowd is a collection of demographic slices, together with a *default demographic*. Because the slices do not need to cover the entire crowd, it may happen that an agent is part of none of the demographics from the slices. If that is the case, the agent will be part of that default demographic.

The crowd socket configures all aspects required to insert the crowd into a concrete environment. It contains:

Agent object type First and foremost, to insert a crowd into an environment it must be known which type of entity will represent agents (e.g. a generic human, a police officer or a car).

Spawn spaces These spaces specify where in the environment the agents will come (or spawn) from.

Spawn rate & conditions For each type of spawn space, the desired spawn rate must be specified, which can be either a fixed value or a random distribution. Conditions may also be imposed on the spawn space to prevent agents from spawning. For example, agents might only spawn when the door is open, or while there are less than 500 agents in the environment.

The second part of the semantic crowds approach is the agents, which are the ones that ultimately make use of the semantics present in the environment. Agents have a set of goals that they wish to fulfill, each one containing one or more semantic desires, as introduced in Section 8.3.2. For each goal, they also have an urgency valuation function, or rather a way of determining how urgent each of the goals is.

The agent's thought process goes as follows. First the agent re-evaluates all its current desires, and determines which are the most urgent, e.g. as the *hunger* ATTRIBUTE of the agent grows, the desire to eat some food will become more urgent. Now, ways are searched to fulfill a particular desire. The agent will query its world view for each TANGIBLE OBJECT instance able to perform an ACTION that affects the desires. If performing that ACTION involves currently unsatisfied requirements, ACTIONS are searched that will satisfy these. This will generate a list of ACTIONS that need to be performed by the agent. For example, to satisfy its hunger, a list might be acquired that orders the agent to eat an apple that can be acquired by purchasing it in a store. To do this, the agent first needs to get some money from the ATM machine.

A colored heat map in Figure 10.4 shows a scenario in which a crowd containing adults (red), elderly (blue) and children (green) walk around in the airport area created using the procedural infrastructures and shown in Figure 10.2. We see that almost all of the agents visit the ATM on the top hallway in front of the hotel or the bank, since most other activities require money. The elderly sometimes stop near benches to rest. And the children mostly visit the arcade hall and the toy store.

Semantic crowds is a novel approach that allows one to easily define crowd templates in an easy and portable way, and re-use them without modification for virtually any semantic environment, in which the objects available are spontaneously used in a meaningful manner. This is achieved by having each agent query the environment to find whatever objects are deemed suitable to fulfill its desires. A more detailed explanation can be found in [48].

10.5 Simulating urban area development for semantic game worlds

Everything is placed somewhere for a reason. In a city, many choices are based on how the layout of the city used to look like. A road might be bent because there used to be a giant building nearby, a certain neighborhood might be poor and dangerous, but in the past it might have been a rich neighborhood, so there might still be buildings you would expect in a richer neighborhood but in pretty poor conditions because they have been abandoned for example. A city that already existed in the middle ages might have been surrounded by city walls. Remnants of the city gates might still be there, and the main road patterns will still be visible (a town square in the centre with concentric circles around it). This inspired us to create a project to simulate urban area development to create urban environments for semantic game worlds.

Many techniques for procedurally generating urban areas have been proposed. These are aimed at recreating patterns seen in real world cities. Either by directly implementing templates for road networks or by using rules for growing road networks. Some also include determining land uses of lots created within the cells of the road network either via the user or via simulation. Many methods

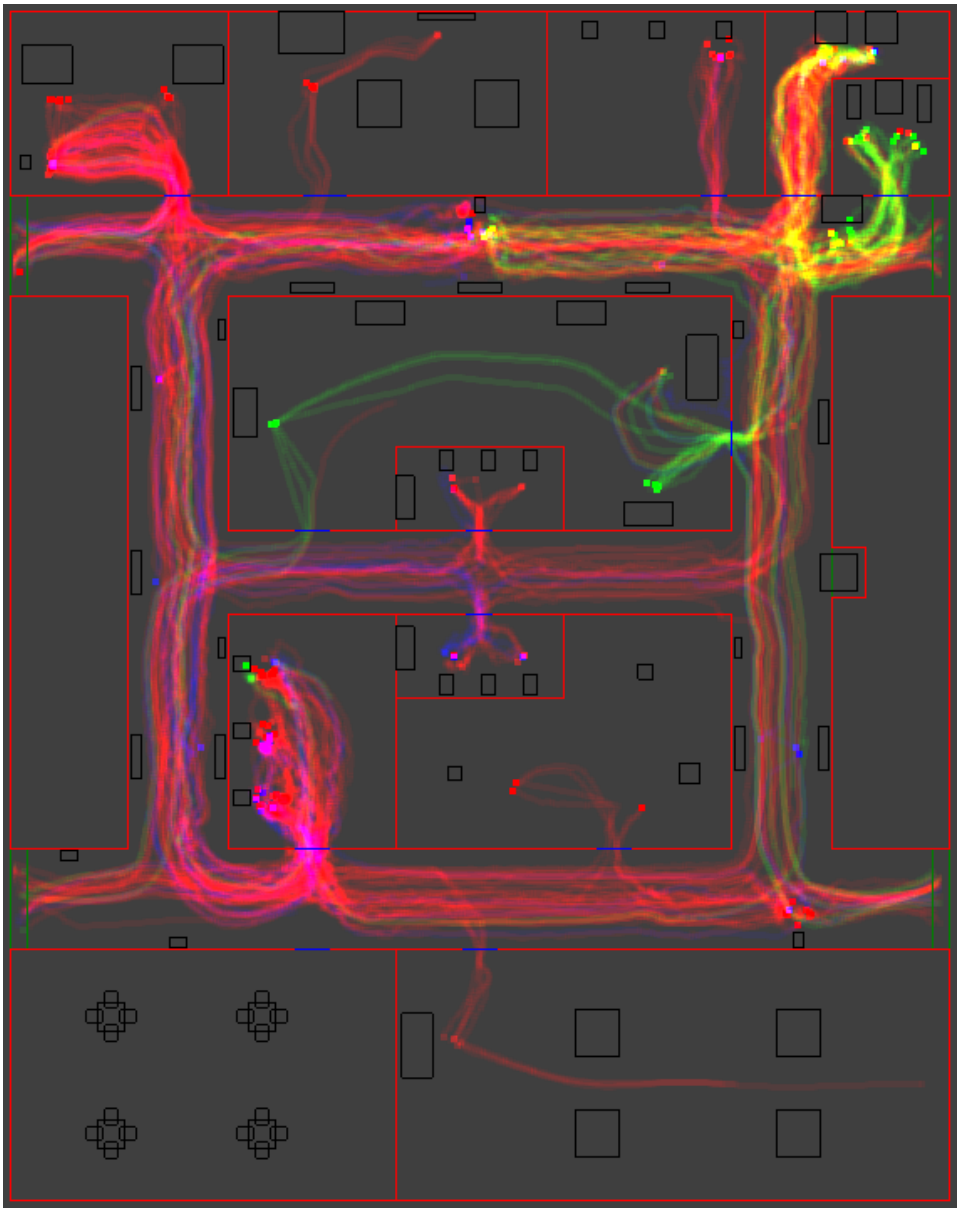


Figure 10.4: This figures shows the heat maps of crowds in the airport environment created using the procedural infrastructures (see Figure 10.2). Red shows adults, blue shows elderly and green shows children.

proved to be able to generate realistic results. Though the resulting urban areas lack meaning, both in history and semantic relations.

The complete history of a city is important for its growth, especially the first settlement. A city can start as a small village around a bridge because it is a place for trade and transport. This will cause the city to develop a harbor for transport over water and gates to be able to levy toll on carts and keep unwanted people outside the city. Also the resources play an important role in the early development of villages. If the urban area is near mountains, stone is cheap and will therefore be used more often in buildings than villages far from rocky areas where buildings are more likely to be built from clay bricks or wood. These semantics influence the success and growth of a city and therefore also the later development.

UrbSim, the name of the project of Rick de Ridder's MSc thesis [19], takes into account the meaning behind the steps made in the creation of an urban area, although these steps are not necessarily realistic. They just need to obey the laws set for the game the urban area is generated for. This will ensure that the created content will fit into the semantics of the game and the created content could even be developed further during the game to make the game more dynamic. The terrain on which it is build, the available resources, and events that occur (such as disasters) influence the growth of the settlement and also its type. Other factors like technological advancement and neighboring settlements may also contribute to the final shape of the urban area.

UrbSim simulates the land use over time by placing, updating and removing lots from the urban area. It uses resources to define which lot types repel and attract each other. Also the shapes of lots are affected by local terrain features. This gives more meaning to the placement and shapes of lots. The resulting urban area is a collection of lots shaped by the terrain and the resources either produced by the terrain or the lots around it. Each lot has specific semantic data attached to it which could be used in the game to interact with the lot.

Unique to this method is the lot generation that is not based on subdivision but rather on growing and expanding from one location to find a suitable shape and size for a lot. Simulation of land use for procedural urban area generation is not uncommon, though the used approach is not grid-based which is not done often. By focusing more on the semantics and game worlds this solution to generating digital urban areas differs even more from other proposed methods. It extends current techniques by adding more history and meaning to the procedurally generated urban areas. An example of an urban are generated using this method can be seen in Figure 10.5.

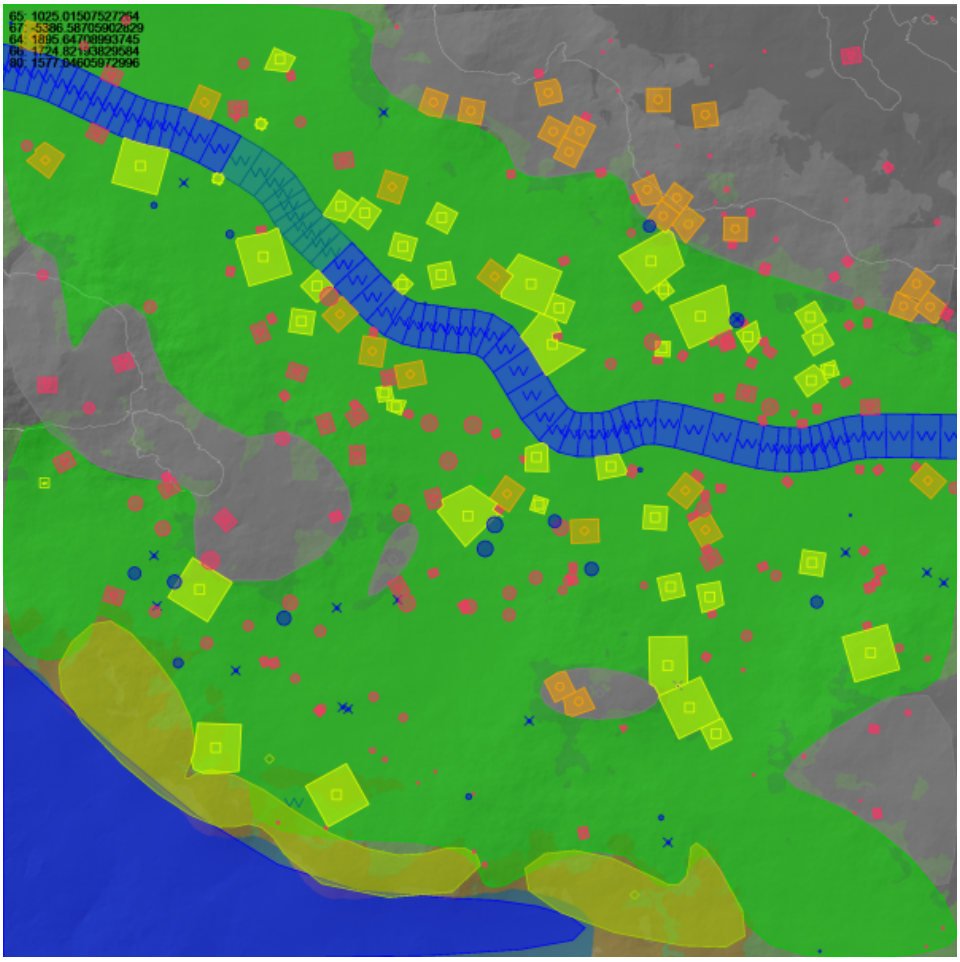


Figure 10.5: A view of an urban area generated by simulating its history. The lots shape around obstacles and terrain features like mountains and rivers. Most lots are clustered around the river, and the areas away from the river contain many wells (blue circular lots) to compensate for the lack of water. In the gray mountainous areas of the terrain, more orange lots (resource gathering) are present, that gather stone.

CHAPTER 11

EVALUATION OF THE SEMANTIC MODEL

In the previous chapters, we explained a semantic model for game worlds and a number of applications for semantics in both the design and runtime phase of games.

In Section 9.2.1 we showed the Entika editor to specify and edit semantics using our semantic model, detailed in Chapter 4.

Chapter 5 explained how to apply semantics, specified with Entika, in the context of layout solving. The use of semantics in semantic scene descriptions combined with RELATIONSHIPS showed to be a suitable approach to automatically generate scene layouts like room furniture layouts or factory floors.

We also applied semantics to the concept of *procedural filters*, to quickly add custom visual effects on virtual worlds, as shown in Chapter 7.

In the course of our research project we invited diverse groups of people to experiment with our proof-of-concept tools and to introduce them to our research results. These people, ranging from professional game designers and programmers to international researchers and students, provided a wealth of feedback on the soundness and usefulness of our general approach as well as on the usability and quality of the results of our applications of the semantic model. To make the feedback further concrete, we performed some interviews with professional programmers and artists at a Dutch serious games company. We introduced them to our ideas and the semantic model, provided them hands-on experience with our prototype Entika tool and allowed them to study and review the methods and results from some of our applications, namely semantic layout solving and procedural filters. In this chapter, we give the results of this evaluation, including a detailed setup of the interview procedure, the findings of those interviews and the positive and negative feedback from the interviewees. The chapter ends with a discussion of the interviews and some future guidelines extracted from this discussion.

11.1 Interview setup

The evaluation interviews were part of a *Knowledge Transfer Project* agreement with the Dutch serious games company *re-lion BV* from Enschede. In Chapter 10 we mentioned a collaboration project as one of our applications of semantics. Within this collaboration, some of their employees were interviewed about our research. These employees were either programmers or artists, and in one case both (a programmer with plenty of modeling experience), but they were not part of the aforementioned collaboration and had not been introduced to our research project before the evaluation.

Although the interviews were performed separately, the idea was similar to *focus group* evaluations. Since some of our prototypes were not yet at a level to be thoroughly tested, we instead proposed our ideas to them and sparked a discussion about our research results. We also prevented the discussion from turning into software user tests with a focus on GUI problems; instead, we centered on more meaningful feedback about the research and its main concepts. After the individual interviews, we had an informal talk with all of the interviewees, together with other company members to discuss our findings together. Finally, several students were also asked to perform our test, mainly as a control for our tests and questions. However, their responses regarding their experiences as a player, were also taken into account when creating this report.

Because of the limited size of the group and the prototype levels of our tools, we could not perform a quantitative analysis. Instead, we set out to get independent industry professionals' first impressions of the idea of semantic game worlds and our applications. Most importantly, we wanted to know which problems they would see in implementing some of our research into their development process. At the end of this chapter, we also distilled all this feedback into three important future recommendations.

11.2 Interview responses

This section will give a detailed overview of how the test was performed, what kinds of questions were given to the interviewees and what their responses were.

In the first part of the interview, we introduced our problem statement and a general idea of semantics for game worlds and of our model. The second part was a hands-on demo of the Entika editor. The third and fourth part were about two applications: semantic layout solving and procedural filters. For each of these parts, we have a section which elaborates on how they were performed and that discusses some of the more interesting remarks. At the end of each section, we give a short summary of the important feedback. Afterwards we discuss the closing comments of the interviewees.

11.2.1 Introduction

Problem statement

To start the interview, we asked some general questions like do you play games, how often do you play games, or what is your current job. The interviewees were first asked how they felt games evolved in the last decade, and more specifically the evolution they felt in the behavior of objects.

Most interviewees agreed that realistic behavior is lacking. Sometimes many object characteristics are present, but are not consistently used across the game, e.g. roleplaying games where objects have a weight to check the carrying limit of the player that is not reflected in the physics of that object. Another interviewee brought up the fact that there is very little evolution in the way we interact with objects in games: often they are still giant, obvious *switches* (sic) a player needs to stand over and press space to use. One interviewee remarked that much attention is spent on the *useful* items in the game, very little to background objects.

We conclude that the interviewees didn't see a huge increase in realism (or higher detail) in object behavior and interaction. However, many of them did mention a huge increase in destructibility: many games now allow all (or many) objects in the game world to be destroyed upon explosions or gunshots. It was also remarkable that destruction was one of the first things that sprung to mind when we asked about object behavior.

Asked about the topics that were most important to be immersed in a game (next to graphical realism and style), the top answer was beyond doubt a gripping story, which was the first thing the interviewees responded. Other important elements were no stuttering (frame drops immediately pull gamers from the experience), and gameplay. Sound, music and environment were other answers. Cinematic experiences, where the player races through a high paced game like in a blockbuster action movie was another answer. There were two interesting answers, from the perspective of our problem statement: *internal consistency* and what one of the interviewees called the *domino effect*.

With internal consistency, the interviewee wanted to make clear that it is not always necessary to have a world that is consistent with the real world; however internal consistency is necessary. It is unacceptable to have two similar objects behave in a different way. However, this is still often the case: sometimes a particular object is usable in one level, because it is expressly associated to the story, while similar objects in other levels are no longer usable or behave differently. This would suggest that a centralized, consistent semantic representation of the game world and all its objects could definitely increase immersion by helping developers to maintain this *internal consistency*.

The second answer we would like to focus on was the *domino effect*. With this is referred to being allowed to set up constructions of multiple objects that, when combined, can set off a huge chain of events where the effect of one object triggers an action in the next. As an example the interviewee mentioned **Little Big Planet** by Sony Computer Entertainment. A similar experience in the series **The Incredible Machine** by Dynamix was already mentioned previously in this work. The interviewee thought it would be very nice if a player could set up similar constructions in, e.g. a first person shooter to defeat an enemy in an ingenious, creative and original way. However, the same interviewee also mentioned that sometimes it is more fun to keep it less complicated so the player

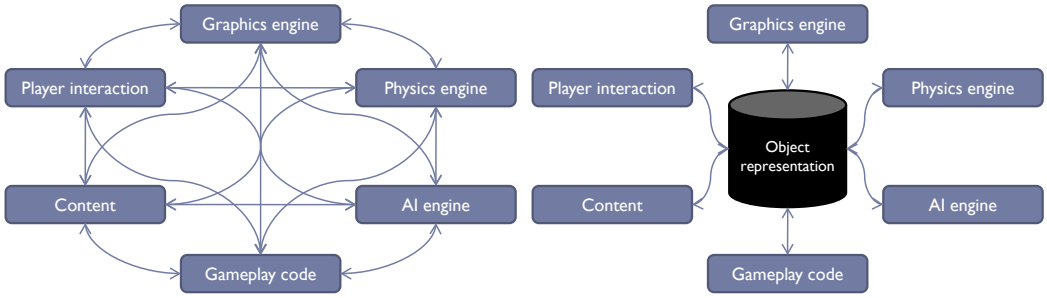


Figure 11.1: A diagram representing the idea of a black box, centralized object representation communicating with all game engine components (right) as opposed to the current system, where all components have one piece of the data-puzzle and therefore have to communicate pair-wise with all other components, if necessary.

can quickly see what objects can be interacted with, and which can't. Nonetheless, it does seem that flexible object behavior would spark players' creative thinking. Although this does not cater to all players, it does suit a certain playing style and certain game genres.

Semantic representation

The interview continued with the introduction of a centralized object representation concept, at this moment represented as a black box, containing all information about an object with which all components of a game could communicate instead of component-wise communication. This story was supported with the diagrams represented in Figure 11.1. This idea was introduced in the following way (literally translated from Dutch):

> Instead of having all components of a game communicate with each other about changes to objects (see left of Figure 11.1) we propose to introduce a centralized description. You can, for now, think of it as a black box that contains all information about an object (see right of Figure 11.1): what kind of object it is, to what categories it belongs to, which its characteristics are, from what material it is made of (and what the characteristics are of that material), which actions a player can execute on the object, what happens when it interacts with other objects, or how it behaves over time.

All of the interviewees were unanimous that such an idea could definitely be useful. One mentioned the fact that this would make it easier to extend games. He also immediately saw many opportunities for games with multiple storylines (branching story arches), making it therefore very suitable for sandbox-style games. Another interviewee mentioned that this could be a solution to provide emergent gameplay. Yet another interviewee mostly saw a gain for developers, claiming it could make the development process easier.

When asked about technical or practical limitations to a semantic object representation, a first issue that was mentioned was performance. Some interviewees were wary of it since they thought it would slow down the game (assuming overhead in an implementation of such an idea). The second issue was scalability of a potential implementation. A final issue that was mentioned by multiple interviewees was the fact that this would also make the game more *unpredictable* which makes debugging and testing the game a lot harder.

Following on this discussion, we introduced the term semantics to the interviewees as (literally translated from Dutch):

> all information about an object beyond its geometric representation.

With geometric representation we specifically mean the polygons and textures that the object consists of. The semantics itself can contain information about the stucture of an object or about geometric relationships with other objects. This was a simplified working definition of semantics, and it served as an introduction to people who were not familiar with the term in the context of game worlds.

The following concepts explained in Chapter 4, were briefly introduced: TANGIBLE OBJECT CLASS, MATTER, ATTRIBUTE, ACTION and RELATIONSHIP.

We did mention that our representation involved many more concepts but we did not go into more detail on them. Some interviewees immediately mentioned they saw specifying all this information would take up a lot of time. They did however, see some useful applications if the information is available.

One of these applications is the fact that searching through a library of models in a level editor could be significantly improved. Instead of searching on names only, materials and other information could be used to query necessary objects and models, e.g. find all metal objects in the library.

Without mentioning anything about semantic layout solving, one interviewee saw possibilities in automating game world generation as well. He immediately saw potential in the relationships between objects. The example he gave was: when placing a shop, using the relationships, an editor could assume that there would also be a need for saleable goods, an agent that serves as a vendor, etc. Perhaps not necessarily automatically placed, but at least as helpful suggestions to place manually.

One interviewee saw some in-game advantage in the fact that it would allow easier creation of simple simulations. As an example he mentioned the EA game series **The Sims**.

Summary

To summarize what this part of the evaluation learnt us in broad terms:

- Most of the interviewees agreed that the quality of object behavior is still somewhat lacking.

- Some of them envisioned important advantages in a centralized, black box, semantic representation of game world objects, including emergent gameplay through more creative solutions, a better organization of objects while designing levels and an easier way of extending games.

- However, they raised some issues concerning the extra time it would take to specify this information and the influence on the performance of games.

11.2.2 Entika hands-on demo

The interviewees had the opportunity to work with the Entika editor, the proof-of-concept editor (see Section 9.2.1) we created that allows users to specify semantics according to our proposed model for semantic game worlds.

The interviewees followed a list of instructions to create a small part of a farm simulation including animals that could grow older and eventually die. One of the animals they needed to create was a chick that transformed into a chicken when it reached a particular age. The chicken could then provide eggs to the farmer who owned the chicken.

Basically there were two points of view: two more design-focused interviewees would like to see the editor to take a step to a more visual setup, while the programmers liked more the idea of textual, sentence-like input, e.g. using autocompletion, less drag-and-drop.

Another common response was the huge amount of information shown on the screen. First of all, all lists with semantic concepts are represented in a double tab system. Second, when selecting a node (e.g. a TANGIBLE OBJECT CLASS) all possible information and lists for that node is shown (even when they are currently not used or empty). Using expanders most of this is hidden, however it does make the view a little cluttered and generally hard to find things.

However, despite these interface imperfections the interviewees were mostly positive, and considered the ideas behind the editor as convenient.

When comparing to scripting or coding, interviewees thought the editor was quicker, easier to extend and a lot easier to reuse (even among different games). There is also no way of having typo errors (although most programming IDE's or scripting editors have autocomplete and syntax highlighting to counter those), but, more importantly, concepts are unambiguous: there can, for example, exist multiple TANGIBLE OBJECT CLASSES with the same name, but they are separate nodes, and one is able to separate them based on e.g. descriptions or parents/children. One interviewee did mention that one needs to get used to it before it would be faster or easier in use. Another advantage an interviewee saw was the fact that, out of the box, the editor has much more available (especially when the editor is already filled from other projects or games). But even in an empty library, there are concepts, like parent/children hierarchies, RELATIONSHIPS, SERVICES, UNITS and more that are already available and logically related to one another.

On the negative side, some felt that using it was giving up control: they felt they did not have the simulation in their own hands anymore. It was also a bit fuzzy to one interviewee where the borders of the system lie: what can be done or specified in the editor and what cannot. Also one

interviewee mentioned that he thought an overview of the world hierarchy would be lost much sooner as opposed to using code, since the editor, in its current form only gives an overview of one node: either one SERVICE, one TANGIBLE OBJECT CLASS, etc. However, there are many ways of solving this problem in future versions of the editor.

Summary

In conclusion, it is clear that the Entika editor is not yet the definitive solution for the specification of our semantic model for game worlds. A more *visual approach*, using nodes to represent concepts and edges to show links between concepts would be the way to go. However, despite the above remarks about the editor interface, there was a unanimous *positive idea on the concepts and the used semantic model* itself. The interviewees saw a big step in the good direction for most of the problems we set out to solve when first started work on our model, noticeable from the advantages they saw in it. A more streamlined, *less cluttered way of presenting* the information is *necessary*, though, to be actually useable.

11.2.3 Semantic layout solving

Sketching rooms

Before explaining anything about the concept of semantic layout solving, we gave users a page with a top-down view of an empty room of 5 by 4 meters, 2 windows and 2 doors, 1 leading to a hallway and 1 leading to the kitchen. They were asked to draw a living room layout with both a dining and a sitting area, while verbally mentioning everything they took into account when placing the furniture.

The reason was to check whether or not our semantic layout solving approach using scene descriptions was intuitive compared to the informal way people fill a room while making the sketch. As the base, we gave them an empty room of 5m length and 4m width, 2 windows in the northern wall, a door to the hallway in the western wall in the corner with the northern wall and a door to the kitchen in the eastern wall in the corner with the southern wall. What became immediately clear was that everyone started with defining in broad terms where the dining and sitting area should be placed and the walklines between the two doors. Everybody, not surprisingly, chose to put the dining area near the kitchen door. However, some chose to break up both areas lengthwise, some crosswise. And the walkline between the hallway door and the kitchen door broke up the dining and sitting area in all sketches (as can be seen in two of the examples in Figure 11.2). In our semantic layout solving approach we use **clearance** SPACES to mark walkines, i.e. clear passages between doors in a room (see Section 5.3).

What became clear was that the interviewees sometimes came very close to the reasoning we used in our semantic layout solving approach using scene descriptions. What happened was, they first started mentioning *what* they wanted to be present in the room, e.g. (literally translated from Dutch):

> I want a TV, two couches, a table and a couple of chairs.

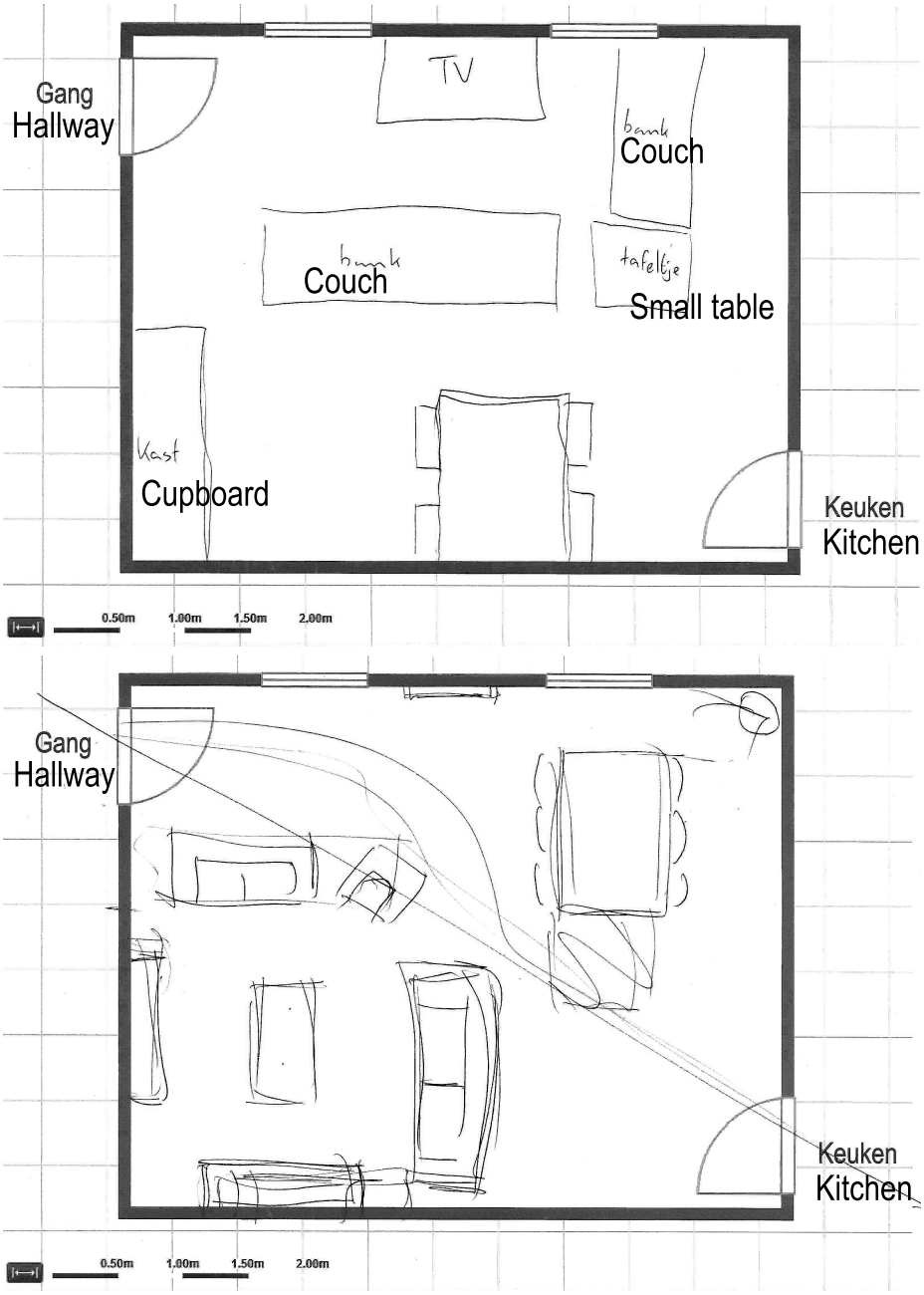


Figure 11.2: Examples of the living room layouts two of the interviewees drew.

And when they were actually drawing objects on the location they thought was suitable, they mentioned *how* the objects should be placed.

This closely resembles the setup of scene descriptions. In the descriptions, one can define what objects and in what quantities should (and could) be available in a particular type of scene. While in the specified semantic TANGIBLE OBJECT CLASSES, one can define RELATIONSHIPS (as well as additional, scene-specific relationships in the description entities, see Section 5.2.2) between objects. Others didn't explicitly split up these two actions, adding objects one by one and explaining the considered constraints with each placed object.

When it comes down to the actual criteria people used for placing objects, the interviewees used mainly expected relationships:

- Place the dining table *near* the kitchen door
- Place a plant *in* a corner
- Place dining table with short edge *against* the wall
- Place the television *between* two windows

These follow neatly the one-to-one RELATIONSHIPS specifiable in the semantic model. The fact that interviewees didn't explicitly used measurements (instead using vague terms like *near*), was expected in this experiment with a sketch. Interviewees completely accepted the fact that specifying these relationships also involves defining these details explicitly.

It was interesting to notice that a minority of the constraints used by interviewees are not directly mappable to the RELATIONSHIPS from our semantic model. One of them was:

> Place the couches in such a way that people sitting in the couches can look at each other.

This relationship is interesting because it adds the *user* of the object in the relationship. However, there is a simple solution to be able to express such a relationship in our model. A SPACE that reflects how one can *use* the object is possible to add and usable for many reasons. One of which, in this example of a couch, could be the position where an agent or player character should be placed when performing the ACTION **sit** on that couch. And when specifying this relationship, we can include this SPACE **use space**. Explicitly, this means specifying the following RELATIONSHIP:

> Place the TANGIBLE OBJECT CLASS **couch** with its **use space** SPACES **facing** a **use space** SPACE of an instance of the TANGIBLE OBJECT CLASS **couch**.

There was, however, a *constraint* one interviewee used that is more difficult to map to our structure, which was:

> I'm going to add a single-seat couch to create a closed sitting area (see Figure 11.2, bottom).

Technically, there are obvious solutions to *solve* this problem however it is unclear how to present this option to users and how it should be specified using concepts from our semantic model.

The approach

The interviewees were now introduced to semantic layout solving and how one can use it. We explained both the general approach to create a description, the three main elements of any description:

- which objects to place,
- how to place them,
- and how the layout should vary based on different contexts.

The interviewees were shown how to create RELATIONSHIPS. They already experienced creating non-placement related relationships in the Entika hands-on demo (**Owns** RELATIONSHIP between **Farmer** and **Animal**). We showed how the same one-to-one RELATIONSHIPS (with the potential addition of some parameters) were suitable to create placement relationships for semantic layout solving.

The overall response was very positive. Interviewees immediately spotted opportunities, e.g. filling entire cities with populated, furnished buildings with limited user interaction. The way the descriptions need to be specified sounded very intuitive to the interviewees.

Following that thought, our use of the concept of CONTEXTS to vary and customize scene descriptions seemed very appealing to the designers, since they find it very important that character personalities or living conditions are reflected in their house interior. Examples they gave included that a messy person would have some dirty laundry spread across rooms where that would not be typically found, or when lacking a dedicated room the ironing board could perhaps be found in the kitchen or the living room. It is therefore clear that the use of CONTEXTS is important in our semantic layout solving approach and deserves perhaps even more attention and refinement.

The hierarchic nature of the approach also spoke to the interviewees. The fact that one can build up scenes from the ground up, ever increasing the complexity without resulting in one single description that contains the entire scene.

One of the interviewees mentioned that he thought the approach felt quite suitable for house interiors, but was not convinced that it would be generally applicable to other scenes, e.g. an industrial area or park. However, although showing the examples, we did not have the time to explain in detail how the forest road with roadside objects and the factory floor were specified (see Figure 5.5).

Summary

Some of the things we noticed in this part of the evaluation:

- From observing and listening to people when creating sketches, as well as hearing the feedback from interviewees after explaining our approach, it seems safe to say that the semantic layout solving approach combined with scene descriptions is quite intuitive and, especially for room interiors, perfectly usable.

- The current concept of RELATIONSHIPS is quite capable of handling all required relationships, with only a handful of exceptions, like the *closing* an area example given a few paragraphs back.

- The use of CONTEXTS to customize and vary scenes is a powerful aid and perhaps deserves more focus and attention and should be more closely connected to the idea of *back stories* of characters.

11.2.4 Procedural filters

Finally we introduced the interviewees to procedural filters. After explaining the problem we wanted to solve with the help of such filters, we gave them an overview of an example filter. This filter was the *age world* filter, which first searches all instances of TANGIBLE OBJECT CLASS **Building** in the world and applies the *age building* filter, which in turn searches all instances made of MATTER **Wood** to apply an *add moss* filter, all objects made of MATTER **Metal** to apply an *add rust* filter and so on. We showed one example of such sub filters, the *add moss* filter and also explained that the value of the ATTRIBUTE **Deterioration** of the **Building** instance was used as the parameter for these sub filters. Based on this example filter, we showed how the filter editor works, how a filter can be created and what types of visual feedback is available in the editor.

Again, the main response was very positive. An example given to us by a designer from personal experience was the adding of snow. They added this snow shader to all polygons facing up. However this also put snow on objects underneath a cover. They are thinking of technical solutions, resembling shadow casting techniques, to solve this, but he saw the semantic approach of our procedural filter idea as a potential alternative solution to this problem.

Especially to quickly change the main feel and appearance of an entire city this could be a great time saver. One interviewee mentioned that, even when it would take multiple hours to apply a number of filters on a huge city or game world, the profit would still be significant. One interviewee also mentioned that a system based on this idea could be immediately applicable in their current workflow.

There were also some slightly negative responses as well. One interviewee found the use of the semantic vocabulary in such a filter system to be potentially dangerous. He compared this to the *search and replace* function in text editors or IDE's. For example, adding rust to *all* metal objects depending on their age, might affect objects the user was not thinking of and were not intended to change. Potentially this could be solved by giving more feedback to the user when testing the filters on an example world, e.g. outputting to the user which instances were affected by a particular filter.

Another interviewee would like a realtime, or at least semi-realtime example of the filter to immediately review mistakes made in a filter.

The editor proved quite appealing to interviewees. The node-based approach was interesting to most of them and some wanted this to be an example for where the semantic specification editor to go towards. However, this editor shows as well that such a node-based editor quickly becomes cluttered in complex examples.

A slight negative note to the idea was that people soon noticed the applicability to apply ageing effects and seasonal or weather effects, but nobody could imagine other examples where the idea could be applicable.

Summary

About procedural filters, we can conclude the following:

- Both the idea and the implemented editor based on it were very positively accepted by interviewees.
- People saw themselves using the idea and believed it could mean a great time improvement.
- The biggest negative point, however, is that interviewees only imagined a limited amount of possibilities and applications.

11.2.5 Closing remarks

We ended the interview and the test sessions with some closing remarks. The interviewees were asked about their final impressions about semantics as a black-box representation for game objects and if they thought of it as an added value.

When asked what the most important advantages of introducing semantics for game worlds are, we got the following responses:

- It becomes easier to reuse object data and behavior across different games, which increases the speed of development.
- Interaction with objects can be specified more quickly and with more detail.
- The ability to create *enter-anywhere* game worlds without encountering the famous *invisible walls*.
- In general: new possibilities.
- The player will become less limited in movements and options, therefore extra freedom, and hopefully more fun.
- This caters to gamers who like to explore every bit of the world.
- The automation ideas (semantic layout solving and procedural filters) can free up time when creating the game and gives designers more time to spend on smaller and important details.

The disadvantages according to the interviewees were:

- The idea requires some getting used to.
- It is quite a different way of working.
- More freedom to the player means more things that can go wrong and that need to be tested.

The reactions to the general idea of using the semantic model as the glue between different components of games were:

- This idea would require a significant mind-change.
- Especially with some sort of standardization, this could be the future of world creation. There could perhaps become possibilities of reusing and selling specified databases.
- One technical hurdle is keeping up performance.
- The second hurdle is the fact that specifying the information will take a lot of time.

11.3 Discussion

With regards to the problem statement, we can conclude that both as game developers and gamers the interviewees felt there were no big leaps in the evolution of object behavior in recent years. However, most agreed that more realistic object behavior could lead to more fun games, especially in some genres, mainly the exploration sandbox-style games.

When told about a generic idea of a centralized, black-box object representation, interviewees saw some advantages like possibilities for emergent behavior and more creative solutions from players, a better organization of objects in level editors and easier ways to extend games. The downside to the idea was mainly the time they thought it would take to specify the information and the possible drop in performance such a system would cause.

After learning about our idea of semantics and the concepts of our semantic model, interviewees responded well to the idea. They thought the idea was interesting and applicable and that the concepts were sufficient and clear. After working with our editor to specify the semantics, the general consensus was that even with this version of the editor time could indeed be saved, however they would much rather have a more attractive and less cluttered presentation of the information.

The semantic layout solving using scene descriptions was deemed a good idea. Based on the feedback and on the observations of users when creating a room layout sketch, we can say that the idea is also intuitive and corresponds to the way people think when layouting a room. The RELATIONSHIP concept from the semantic model is capable of handling almost all relationships and constraints thought up by the users while sketching, except one important one which constrained a particular area in a room to be *closed*. The idea of CONTEXTS is a particularly powerful one, especially when

trying to customize scene descriptions based on the personality of the character that owns, works or lives in that scene.

Both the idea of procedural filters as the implementation in the filter editor proved particularly popular with interviewees. They saw it as a great way of saving time in the development phase. People wanted to use the idea and thought it could fit nicely in their current workflow. On the downside, they could not find many more interesting applications besides ageing and destruction and the simulation of seasonal and weather effects.

In general, our semantic model and the idea of using semantics in game worlds, proved to spark an interest with the interviewees. They were especially interested in the ways it can speed up development ranging from easier specification of object behavior and interaction and procedural generation using filters and semantic layout solving. They saw it as a particularly interesting idea to create big, exploration-heavy, sandbox-style, enter-anywhere game worlds. The fact that it could save a lot of time was interesting to designers, since they think it would give them more time to spend on smaller details and more important parts of the game worlds. The negative comments had first of all to do with the fact that interviewees think it requires a significant mind-shift and that specifying the semantics takes up a lot of time. They also raised technical concerns for the performance of games using semantic game worlds.

11.4 Future guidelines

At the end of the evaluations, we can look back strongly positive on the idea of using semantics in game worlds according to our semantic model. Many of the problems we set out at the beginning of this project (i.e. lacking behavioral realism breaking immersion in games, inconsistent data between different components because of separate data, and procedural content generation not breaking into mainstream use because of unintuitive and vague parameters) could be positively affected with the inclusion of semantics, according to our interviewees based on both their experience in the game development industry and as gamers. Especially the procedural generation ideas (semantic layout solving and procedural filters) were unanimously welcomed as positive. The interviewees saw the entire process as a time saver and as an important mechanism of maintaining consistency between different components of games. However, from some of the remarks, we noticed that also with regards to player interaction and object behavior, our semantic model brings a lot of important new options to the table. We feel that it would take some more time and some striking example games (or game-like testbeds) to convince more people of the advantages in this field.

The issues concerning performance are quite relevant. Since we did not particularly focused on performance in our proof-of-concept implementations, we cannot guarantee that impact on performance of the use of semantics is neglectable or not (in our non-optimized versions, the impact is still significant both on framerate and memory use). We do, however, believe that a strong technical focus on this field could greatly decrease the loss of performance, however we cannot guarantee that it could be solved to the point where the loss becomes insignificant.

The issues regarding the presentation of semantics and its specification, are in our opinion and

based on the feedback of interviewees perfectly solvable. A more visual presentation that is perhaps context-sensitive, e.g. when creating a service only show information that is relevant to that particular service on the screen, is perfectly doable, but would require a lot of fine-tuning and testing. Templates would be another welcome addition to specify services, since the amount of options can be overwhelming, and templates and/or wizards to guide users through frequent service types. We feel that the development of such tools falls out of the reach of a scientific research project, but to become commercially feasible such an improved tool is definitely necessary.

To conclude, the three future guidelines, filtered based on this evaluation, that we find the most important and urgent are:

- Increase the performance of the semantics engine that handles behavior based on services and player interaction.
- Increase usability of tools that specify semantics and perhaps help users by automating part of those tasks.
- Focus on the use of CONTEXTS in the semantic layout solving approach regarding character personalities.

CHAPTER 12

CONCLUSIONS

In this final chapter, we draw our conclusions on the instrumental role semantics play in the design and development of virtual worlds for games and other applications. By means of our semantic model, we dealt with various fields where semantics effectively supports and enhances the process of building virtual worlds. First, the contributions of the work presented in this thesis are revisited and then some recommendations are proposed to further study the use of semantics in games in the future.

12.1 Research contributions

From studying games and their evolution in recent years, we noticed a considerable lack of *meaning* in game worlds. The behavioral realism often does not match the visual realism. We wanted to address this using semantics. Our initial definition from Chapter 1 is now formalized into the following definition:

> game world semantics is all information on a game world and its objects, including structural, geometric, physical, functional and behavioral information, all integrated in a generic and consistent model, in a way that does not constrain a designer's creativity.

Based on this definition we will now answer our research questions.

How can semantics improve the creation and consistency of game worlds?

To answer this question, we set forth in Chapter 2 by: (i) dissecting some of the current games as well as general trends in game world design and, (ii) analyzing where semantics research can provide opportunities for improvement. Based on this analysis, a set of guidelines was put forward in Chapter 3 that define the necessary characteristics any semantic model should possess to be able to seize the improvement opportunities. Chapter 4 introduced our semantic model for game worlds, created

following these proposed guidelines. This model was applied throughout this thesis to different areas of the game development process where semantics prove to be valuable. These different fields will now be revisited based on the key questions introduced in Chapter 1.

1. What is the role of semantics in the generation of coherent game worlds, both manual and procedural?

One of the main problems we observed was in the way game engines are structured: as a set of highly separate components all influencing the same game world and the objects inside it in their own unique way, but without the necessary coordination, let alone integration, between them. This makes it tough and labor-intensive for developers to produce a coherent game world. By carefully selecting the guidelines in Chapter 3, we ensured that any semantic model following these guidelines is fully equipped to be the glue between the different components of a game engine. Such models can provide developers with a vocabulary to create a single, coherent *black box* representation of the game world, with which all components can communicate and interact with in a uniform manner.

When using procedural content generation techniques, we notice a somewhat similar problem. Techniques might influence elements of the game world that overlap or conflict with elements from other techniques. Again, coordinating all used techniques with every other technique requires a lot of work and is not very robust, since introducing a new procedural generation technique to a game will mean changing many other techniques to maintain perfect coordination. By using a semantic representation as the main hub for all communication, each technique only needs to be connected to this one representation, which propagates changes to all other techniques and also identifies conflicts between techniques based on defined constraints and relationships. In Chapter 6 we described this approach and applied it to the generation of consistent buildings using multiple procedural generation techniques.

This thesis also described a second way in which semantics can play a vital role in improving the generation of game worlds: in the automatic creation of layouts. Whether it is the layout of areas on a lot, of rooms inside a building, or of furniture in a room, they can be automatically generated by means of the semantic layout solving approach introduced in Chapter 5. Its main advantage is that the final control over these layouts still remains in the creative hands of designers, by allowing them to steer the automatic layout process using an intuitive description language. This example of *declarative modeling* enables designers to focus on what they want to create instead of how they will model it. More common, repetitive tasks like one by one placing pieces of furniture in a room are alleviated, while the global structuring of a scene is left to the designer. Moreover, because of the semantics present in the generated scenes, consistency can be maintained after manual edits, by revising the defined relationships between objects in the scene.

Next, we focus on the customization of existing scenes. Procedural filters, introduced in Chapter 7, allow designers to quickly transform scenes and worlds to match different conditions, circumstances, time periods or styles. By combining the power of procedural generation and the intuitive and expressive nature of semantics, designers can easily create filters that can fine-tune the appearance of scenes without changing them structurally.

Finally, the behavior of objects in the game world was addressed. Services, introduced in Chapter 4 and analyzed in more detail in Chapter 8, provide designers with a consistent and intuitive way of expressing how objects in the game world behave, how the player or other objects can interact with them, how they react to certain events and how they evolve over time. An important aspect of services is their reusable nature: since the behavior is not tightly linked to 3D models, but instead to generic semantic entities, the defined behavior is seamlessly reusable in other games.

In these chapters, we demonstrated the importance of the use of semantics for each approach individually. In general terms, a major advantage of the use of semantics when creating game worlds manually or procedurally, is that it allows designers to reason about the world in the vocabulary of the target domain, instead of having to think in mathematical systems and parameters that often have little or no connection to the structures of the world they are actually creating.

2. How can game designers be assisted in the specification of semantics?

Guideline 1 from Chapter 3 states: "Inclusion of semantics should have a low impact on the design pipeline". Therefore, we wanted to analyze how designers can be assisted in the specification of semantics. The most important tool in this specification for our semantic model is the Entika editor introduced in Chapter 9. In building this editor, we found a number of elements vital in assisting the semantic specification. Since a semantic model for game worlds can become quite complex, it is important to properly clarify to users what type of entity or concept they are working on and how these tie together with all other concepts. In the Entika editor, we chose to use color-coding to address this, however there are many other ways to achieve this. Feedback is essential to the understanding of the model behind the tool. The Entika editor provides many forms of feedback, both visual and auditive, but as was found in the user tests in Chapter 11, some more steps are necessary in this field.

Obviously, automation can play an important role in speeding up the process of specification. In Section 10.3, we introduced a method to quickly specify semantics for large sets of 3D models. Using shape recognition and segmentation techniques, the classification and specification of 3D models with semantics can be achieved automatically. The only required interaction by the user is to provide a number of training models to the system.

Although these tools and methods provide a solid starting point to aid semantic specification, it is not yet far enough to fit in seamlessly with the current development process. In the next section, regarding future work, we will describe how this important element of game world semantics should be further addressed.

3. How can the semantic consistency of a game world be maintained in an evolving context?

We already mentioned consistency maintenance in the design phase: after manual edits, the semantic layout solving approach can fix conflicts or update certain layouts to fit with these edits, all based on defined relationships. This third key question takes that consistency maintenance to the runtime level, i.e. while playing the game. Since services play a crucial role in our semantic model for the specification of object interaction and behavior, services are also the main ingredient to maintain semantic consistency while evolving. In Chapter 9, we clarified the workings of the semantics engine. This engine ensures that the effects of interactions or events are properly handled

and that the evolution and the behavior of objects is executed.

4. How can this integration of semantics influence gameplay?

Several advantages mentioned for the previous key questions, could likely lead to innovation in gameplay. Services, especially because of their reusable nature, could lead to many more interaction possibilities: even in games where object interaction is not an integral part of gameplay, populating their game world with interactable objects makes for a more immersive experience. Moreover, more diverse ways of interacting with the world, will make problem solving in games more versatile.

The fact that the generation of game worlds can be improved and facilitated in a number of ways using semantics (as mentioned for the first key question), will make it easier for designers to create larger game worlds. And since these semantically generated game worlds are also embedded with behavioral information, these game worlds would lead to more exploration-based gameplay, something that is ever growing in popularity.

These two elements combined, might also enable emergent gameplay. Large, explorable worlds with more interaction possibilities will push players to find new and creative ways of solving problems, perhaps ways that were not originally thought of by the designers. Such emergent gameplay will furthermore personalize the experience for players instead of serving them with a single, linear one, therefore increasing their replay value.

In short, after addressing these four key questions, we can summarize the following answer to our research question:

1. Semantics can improve the generation of game worlds by allowing new ways of automatic or procedural generation, as well as by assisting designers when manually creating or editing them.
2. Semantic models, adhering to our guidelines, provide a great vocabulary to form a single and coherent representation that can serve as the glue between different game engine components. Moreover, a behavioral specification, e.g. by means of services, is instrumental in maintaining the semantic consistency while the game is running.

12.2 Application of semantics

In this section, we take a broader look on the application of semantics. First of all, we categorize the types of applications we used, i.e. the ways we reasoned on the semantics from our model. Next, we discuss how these applications fit in with research in other fields and where we want to go next.

12.2.1 Application types of semantics

In this work we have discussed a number of applications of semantics using our semantic model. These applications reason over semantics from the model in different ways. We can categorize these in three application types:

1. Using semantics to *translate* among different techniques, among different game engine components or between users reasoning in a high-level target domain vocabulary and low-level techniques.
2. Using semantics for the *moderation* of conflicts between entities based on defined relationships and components.
3. Using semantics to *evaluate* and *execute* behavior and effects of interaction between entities based on defined services.

For translation purposes, a mapping needs to be made between, e.g. a particular technique or a game engine component and semantic concepts. These concepts can then be shared between each *user* in order to communicate with other users. This application type was used in Chapter 5, where we used the semantic model to *translate* the high-level descriptions defined by designers into more low-level instructions that can be interpreted by the layout solver. In Chapter 6, we use semantics as a *translator* to allow communication between different procedural generation techniques. All techniques register the entities they created to a shared semantic representation of the building. Having a single, shared representation of the game world, using a common, target-domain vocabulary allows seamless communication between all users of the game world, whether these are designers, engine components or procedural generation techniques.

In the case of procedural filters (Chapter 7), we do not simply have a one-to-one mapping between target-domain vocabulary and e.g. a procedural technique or geometric relationship. Instead, filters allow designers to create as many layers of abstraction to their filters as they want. Because of the hierarchic design of procedural filters, low-level procedural content generation techniques can be shielded by filters that are more intuitive, which in turn can be reused in other filters. For example, a filter to apply noise can be used in a procedural filter that adds rust, which can be used in another filter to age a building.

A second application of semantics is the moderation between entities in the world. Using the relationships and components defined for entity classes, we can assess the validity of the position or orientation of entities. We define rules about allowing or disallowing overlap between certain SPACES of entities, or define RELATIONSHIPS on how entities should be placed relative to other entities. Again, this mechanism is used in our semantic layout solving approach in Chapter 5, but the same mechanisms are used to indicate conflicts between entities placed by different procedural generation techniques that are combined to generate complete and consistent buildings, as described in Chapter 6.

The third application type is the evaluation and execution of entity behavior. Using the concept of SERVICES, we can define actions and their effects to describe how entities behave over time or how they react to actions from other entities. Semantic game worlds can be queried to find possible ways of achieving certain goals, i.e. when agents need to achieve a certain state, the semantic game world can compose a set of actions that will enable the agent to achieve this state. Furthermore, we discussed the workings of a *semantics engine* (see Chapter 8) that can handle and simulate the interaction, behavior and effects of entities based on the defined semantics and the current state of the game world.

12.2.2 Broader perspective on using semantics

Designing and modeling a virtual world is a highly creative task which does, however, involve numerous technical and much less creative chores that require specific knowledge about the digital representation of a virtual world. Declarative modeling tries to reduce the technical burdens and strives to find solutions that allow designers to focus their efforts more on the creative part of their job. It tries to bridge the gap between the high-level vocabulary of the world that is being created and the more low-level terminology involved in the modeling process. In this work, we used semantics to provide that bridge: it allows translating back and forth between these two levels.

Besides having one-to-one mappings of a semantic concept directly onto, for example, a geometric relationship or a procedural generation technique, an interesting example is the concept of procedural filters. Since it allows as many abstraction layers as necessary between a high-level concept and the low-level procedural technique, designers can create a hierarchy of filters to continuously hide more and more lower level operations. We believe that such a continuous level of abstraction is the best approach to achieve declarative modeling.

Next to specifying the world and performing translations, we used semantics to specify behavior of the world and entities inside it. Through the concept of services, we allow designers to easily and intuitively specify behavior and the effects of certain behavior on the rest of the world.

Looking back on our approach and the semantic model we created, there is some overlap and similarities in different fields, one of which is the Function-Behavior-Structure (FBS) framework [76, 27]. This framework to represent design knowledge consists of three variable classes that each describe different aspects of a design object: Function variables describe what an object is for, Behavior variables describe what it does and Structure variables describe what it is. Our design of services links to the Function and Behavior variables of this FBS framework. Services describe what the functionality of an object is (which can be compared to Function) and what their effects are (which is similar to the Behavior aspect). Our model contains many more concepts that all somewhat relate to the Structure aspect of FBS, since we wanted to describe more specific information about what an object is, how it looks like, what it is made of and how it relates to other objects. Originally FBS was devised as a way of defining a functional hierarchy between objects. In design, FBS is more a physically realistic simulation while in our approach we want virtual worlds to *behave* as a realistic world without explicitly having to specify all the physics in detail.

Looking at the ontologies and other solutions that are available in many diverse research fields, we can say that the model we introduced shares a good deal of similarities with some of them. We believe that many of the existing ontologies could fit in quite well with the application approaches described in Chapters 5 to 10. We did, however, devise the model specifically with procedural content generation in mind. A procedural process and the specification of geometric structures or relationships was essential for many of our applications and therefore our model had to cater to that as much as possible, while still allowing for behavioral information to be specified. In that sense, our approach spans across both the more physical or geometric research on semantics as it is often used in the CAD/CAM world, and the research fields where meaning and language are more dominant, as e.g. in semantic web studies. Our conclusion in this regard is that the unique blend that computer games

require between the state-of-the-art computer graphics and modeling techniques on the one hand, and the simulations and behavior on the other hand, create the need for such integrated, cross-disciplinary solutions. Much research towards semantics in all different research fields delivers meaningful pieces of the complex puzzle that game development has become, but careful consideration has to be given to combining them into an integrated, all-encompassing solution. This work is a first step in that direction.

Time and again, we notice that semantics is essential in the communication between humans and machines. Each field might require their own specific models or processes to represent semantics, but translation between human-readable concepts and machine-readable concepts or instructions remains a necessary one. Either from human to machine, e.g. in the context of semantic web, or vice versa, e.g. when presenting designers with low-level design elements like procedural content generation techniques.

We distinguished three application types: using semantics (i) to translate, (ii) for the moderation of conflicts and (iii) to evaluate and execute behavior. Beyond these three we focused on maintaining the consistency of the world in an evolving context, by applying the effects of objects' behavior. To increase the usability of a semantic game world solution, this idea of maintaining semantic consistency needs to be taken a step further. On the one hand, applying the logics-focused research of semantics, would be of significant benefit to designers. Rules that, for example, enforce that a 'parent' relationship between X and Y also implies a relationship 'child' between Y and X, would not only speed up the specification process, but also limit potential errors. On the other hand, the declarative modeling approach would greatly benefit from a richer vocabulary for the specification of conflict resolution. Many situations require rather specific solutions, e.g. when diverting a road with a bridge crossing a river, the bridge needs to move with the road and needs to remain over the river, or when the river becomes wider, the bridge needs to become longer. And when the bridge would become too long for the current construction type, a new type would be necessary. This kind of behavior needs not only different solving solutions, it also demands for a more expressive vocabulary focused on the resolution of conflicts. Another application type, which would be of particular interest in the context of games, is reasoning about the world in the context of its history. History can play an important role in making the player feel like the game world is alive, thereby increasing the depth of the experience. It will allow the player to find out about the stories that happened to a town while away on a quest instead of having a single fixed state. Moreover, it will provide AI agents with information on past experiences and events, which they can use to make informed decisions. This information might also be of importance when trying to adapt gameplay to the player's experience level, skill level or playing style. Much more than storing the events in a giant database, the challenge is how designers will be able to reason about and query past events in a meaningful and intuitive manner. We believe this to be the most significant and challenging application method of semantics that is yet to be researched. In the next section, we will give a more general overview of future recommendations.

12.3 Recommendations for future work

In this work, we showed how semantics, and more specifically semantic models adhering to our set of guidelines, can improve the generation of game worlds in a variety of ways. The game developers in our user studies agreed that semantics can play an important role in game development. However, they felt that the tools to specify these semantics still need improvement. We now follow with a number of future recommendations (i) to improve the specification of semantics, and (ii) to apply semantics in new domains of game development.

12.3.1 Specification of semantics

Embedding in modeling tools To further overlap the specification of semantics with the current development pipeline, it would be desirable to have this specification happen while modeling the 3D models that are to be put in the game. Especially physical and material characteristics of the object and the hierarchic structure of its parts can perfectly take place in the modeling phase. Such a semantic modeling tool can be an important step towards integration of semantics in game development.

Further automation We believe that there are many opportunities to further automate the specification process. We already showed the importance of model shapes to deduce information about classification and segmentation of objects, but also information on the used materials or physical attributes might be deduced from these models. Moreover, the process of creating object behavior could become faster by using templates or smart automatic completion of values. Many of such small improvements will eventually lead to an important speed improvement in the specification phase.

Visual representation The designers we talked to in the user studies all pushed to a more visual representation. They felt that the current, mostly text-based, approach became too cluttered and too confusing when dealing with more complex behavior. However a good way of visually representing every element from our semantic model will prove to be a significant research challenge.

Crowd sourcing Many hands make light work: cooperation in the specification phase might alleviate a lot of the work. One could specify the semantics vital to one's own game, while drawing from the efforts of others and in turn, contributing to their future work. It is obvious that to make this work, among many other challenges, the specification tools need to become more accessible and easy to use.

Semantic level of detail When cooperating, it becomes an issue that not every game needs the same attention to detail as the other. For example, a first person shooter might require a simulated eco-system, while a space shooter will not require such an eco-system to simulate the planets the player is passing at rapid speeds. Therefore, a fully customizable semantic level of detail system needs to be researched, that allows designers to choose exactly how far they want to take the detail of their semantic representation.

12.3.2 Future applications of semantics

Semantic physics engine We believe that one of the main game engine components that would benefit a lot from embedding with semantic information is the physics engine. All material characteristics available in our semantic model would be the ideal input for physics simulations, and therefore it would be desirable to have our semantic model be integrated with a physics engine.

Semantic history An important extension of semantics would be the inclusion of semantic history. Not only the current state or representation of the game world is interesting, also the events that happened during the game provide opportunities, especially linked to our research towards semantic crowds and agents. Instead of having characters behave and talk in a similar, scripted manner, they could adapt their behavior to current events. If a gunfight just went on in a street a couple of days ago, characters would likely hurry to get out of it when passing, or if the player just rescued the town from a horrifying attack, this event could literally become the talk of the town. As mentioned in the previous section, this would be a very interesting application of semantics both to increase immersion of players in the game world and allow for new opportunities for adaptive gameplay.

Performance of semantics engine As mentioned previously, the semantics engine that we developed is far from optimized. A great deal of research could still go into further increasing its performance.

Summary

Semantic game worlds

The visual quality of game worlds increased massively in the last three decades. However, the closer game worlds depict reality, the more noticeable it is for gamers when objects do not behave accordingly. One of the main reasons why the realism of the objects' behavior now often falls short to the visual realism, is that game worlds lack meaning. And since game engines tend to be developed using many unrelated components, the information that *is* available is scattered and hard to keep consistent. We argue that what is truly missing here is a glue to keep the different object representations, used by these many different engine components, consistent.

This thesis proposes the use of semantics to act as this glue: a single, consistent object representation with which all components can communicate. We put forward a number of guidelines to which such a semantic representation should adhere to. Following these guidelines we created our semantic model that can be used to produce *semantic game worlds*: game worlds that are populated with objects enriched with semantics.

To increase the realism of the object behavior, our model for semantic game worlds contains the concept of *services*. Services provide designers with a generic, yet versatile and intuitive tool to describe this behavior. Virtually all object behavior can be described using the straightforward pattern of services. The generic setup allows for easy reuse among different games, significantly decreasing the amount of time one has to spend defining object behavior. And since the vocabulary of the semantics is the same as the actual target domain of the game world, it is intuitive and easy to understand, even for people with little or no knowledge of the low-level computational mechanics behind it.

This intuitive vocabulary will make a number of the tasks involved in designing game worlds easier as well. Procedural content generation techniques often use mathematical simulations with parameters that have no meaning what so ever to the actual structure they are creating. By using semantics to conceal these parameters behind more readable semantic concepts, these procedural techniques become available to a wider audience. This idea, which is an example of so-called *declarative modeling*, allows designers to reason more about what they want to create, instead of how they will actually model it.

Semantic game worlds are faster to create, because of the reusability of services and the new possibilities for procedural content generation. Moreover, the generic structure of services gives players the opportunity to be more free and creative while trying to solve problems inside the game world, leading to a more personalized and emergent gameplay experience. Therefore, it is our strong belief that semantics will play an important role in improving both the design process of game worlds and the quality of gameplay.

Samenvatting

Semantische game werelden

De visuele kwaliteit van game werelden is sterk verbeterd over de laatste drie decennia. Echter, hoe meer game werelden de realiteit visueel benaderen, des te storender wordt het wanneer objecten zich niet realistisch gedragen. Één van de belangrijkste redenen waarom het gedrag vaak niet kan tippen aan het visuele realisme, is het gebrek aan betekenis in game werelden. Aangezien game engines vaak ontwikkeld worden met veel ongerelateerde componenten, zit de informatie die wel aanwezig is ook helemaal verspreid wat het moeilijk maakt om deze consistent te houden. Wat hier mist is een lijm om de verschillende representaties van objecten, die gebruikt worden door deze verschillende componenten, consistent te houden.

Deze thesis stelt het gebruik van semantiek voor om als deze lijm te functioneren: een eenduidige, consistente object representatie waarmee alle componenten kunnen communiceren. We hebben een aantal richtlijnen opgesteld waaraan zulk een semantische representatie moet voldoen. We creëerden ons semantisch model, gebaseerd op deze richtlijnen, dat kan gebruikt worden om *semantische game werelden* te produceren: game werelden die gevuld zijn met objecten verrijkt met semantiek.

Om het realisme van het gedrag van de objecten te verbeteren, bevat ons model voor semantische game werelden het concept *services*. Services bieden designers een generiek, maar toch veelzijdig en intuïtief middel om dit gedrag te beschrijven. Zowat elk object gedrag kan beschreven worden met behulp van het eenvoudige patroon van services. De generieke opzet laat eenvoudig hergebruik tussen verschillende games toe, wat de tijd om object gedrag te beschrijven drastisch inkort. En aangezien de vocabulaire van de semantiek dezelfde is als het doeldomein van de game wereld, is het intuïtief en makkelijk te begrijpen, zelfs door mensen die weinig kennis hebben van de onderliggende lage-niveau, computationele technieken.

Deze intuïtieve vocabulaire zal ook een aantal taken van designers bij het maken van game werelden makkelijker maken. Procedurele generatie technieken gebruiken vaak mathematische simulaties met parameters die weinig of geen betekenis hebben in de eigenlijke structuur die ze creëren. Door semantiek te gebruiken om deze parameters te verpakken in makkelijker begrijpbare semantische concepten, komen deze procedurele technieken beschikbaar voor een breder publiek. Dit idee, een voorbeeld van zogenoemd *declaratief modelleren*, laat designers toe te redeneren over wat ze willen creëren, in plaats van hoe ze het kunnen modelleren.

Semantische game werelden zijn sneller te maken, vanwege de herbruikbaarheid van services en de nieuwe mogelijkheden voor procedurele generatie. Bovendien, geeft de generieke structuur van services de spelers de gelegenheid om vrijer en creatiever te zijn in het oplossen van problemen in de game wereld, wat tot een meer gepersonaliseerde en onverwachte gameplay ervaring leidt. Daarom, is het onze sterke overtuiging dat semantiek een belangrijke rol zal spelen in het verbeteren van zowel het design proces van game werelden als ook de kwaliteit van de gameplay.

Acknowledgements

When graduating from UHasselt, I started working as a researcher at the Expertise Centre for Digital Media. At that time, starting a PhD was far from my mind. Until my good friend and colleague, and later on roommate and now employer at Macomi BV, Michele Fumarola, knowing my interest in both game development and procedural generation, mentioned to me this PhD project in Delft. Without him, I would have never known about, let alone started, this PhD project. Next to introducing me to the project, he was also a great help and support throughout my PhD, and most impressive of all, both he and his girlfriend Manu succeeded in sharing a roof with me for almost four years, a remarkable display of patience on their part.

Moving to Delft meant I had to leave the EDM after one year already, leaving many great colleagues and interesting projects behind. Regardless of the short time I worked there, I learned a lot by collaborating with Lode, Tom, Erwin, Joan, Chris and Karin on the VR-DeMo project and the CoGenIVE tool.

During my PhD in Delft, my daily supervisor Rafael Bidarra helped me significantly in my research and with our publications. He strives for perfection, but always remains positive in his criticism. And next to developing his students' academic skills, he also puts in a lot of effort to increase our personal skills, like networking and proper writing style, which proved to be quite a challenge in my case. A big thanks also goes out to my supervisor Erik Jansen, whose critical eye on the work was very much appreciated since it greatly improved the work. Also, in the final year of writing, they both provided me the necessary kick forward whenever my mind was drifting away from finishing. I would also like to thank Klaas Jan de Kraker for his valuable input in many of our publications. Special thanks as well to all my committee members, whose helpful comments improved the quality of this work substantially.

A big thanks goes out to my two partners in crime and paranymphs Ruben Smelik and Ricardo Lopes. Ruben Smelik started his PhD a couple of months before mine, and for almost the entire period we shared an office. This meant we could vent our woes to each other about buggy code, buggy papers and the worst of all, buggy reviewers. We also perfected ourselves in the art of Friday-afternoon procrastination. One of the highlights of this can still be found in our room in the shape of a 'the Witcher' papercraft figure. These afternoons also gave ample opportunity for some cultural exchange: trading "krijg ik ne cola?"[1] for "had U al koffie gehad, mevrouw?"[2]. Two years ago, Ricardo Lopes, started his PhD at our gaming alley, which meant opportunities to have lengthy discussions about games doubled. However, at least as many of these discussions were about sports, which means you can now ask me anything about the NBA draft. Unfortunately for him, he now has the task to use some of my code in his research, although that shouldn't be too much of a problem since I scattered at least tens of lines of comments throughout the tens of thousands of lines of code.

A very special mention in this work should go out to Jassin Kessing. He joined our ranks in many incarnations in between his 6 month-long Asia trips, Kenyan Safaris and new year celebrations in

[1] Quote by several unnamed rhinoceros from the Kabouter Wesley comics and cartoons.

[2] Quote by Mr. Frederik Jacobse of the duo Jacobse and Van Es, two 'vrije jongens' from The Hague.

Russia. First as an MSc student, then as a technical programmer, and finally as some other term we came up with, he completely turned around the comment to code ratio. More importantly, however, the semantic model discussed in this thesis wouldn't have been the same without our numerous brainstorm sessions. Many times at which we thought to have found the ideal structure, after filling a great many of whiteboards, one of us would always come up with some other idea that wasn't possible with it yet.

We also had fruitful cooperation with outside researchers and partners. Fernando Marson worked on procedural infrastructures to be populated with semantic crowds. And Xin Zhang joined our group to research the automatic specification of semantics for large sets of 3D models. Both these projects are discussed in this work and I would like to thank both of them for their valuable contributions. In this work we mentioned our knowledge transfer project with re-lion BV. I would like to thank the company and everyone we work with for the cooperation and in particular Paul de Groot who was our main contact at the company. Also the valuable feedback and evaluation by their employees was very interesting to us.

A great deal of MSc and BSc students cooperated with us in these years. We had our triumvirate of procedural filter guys: Roland van der Linden, Marnix Kraus en Bart Bollen, our duo of building guys: Johannes Bertens and Joachim Boers, and the solo guys Nick Kraayenbrink on semantic crowds and Rick de Ridder on evolutionary cities. Their valuable additions to this thesis and our gaming alley's framework are very much appreciated. I would also like to thank the students of Ruben, Ricardo and Rafael for their cooperation in our group: Robert Schaap, Saskia Groenewegen, Korijn van Golen, Zhi Kang Shao, Mattijs Driel, Quintijn Hendrickx and Shiyang Hu. A special mention goes to Asmar Arsala, who generated even more opportunities to have those lengthy gaming discussions, not to mention many about TV series and movies (come to think of it, how did we ever find the time to do our research in between all these discussions). Joel van Neerbos also made a significant improvement to Ruben Smelik's masterpiece SketchaWorld and I look forward to seeing how his interesting ideas on procedural generation come to fruition.

At our Computer Graphics and Visualization group, many nice colleagues helped create an ideal working atmosphere: Sergio Barrientos, Jorik Blaas, Charl Botha, Bert Buchholz, Stef Busking, Eric Griffith, Gerwin de Haan, Christian Kehl, Peter Kok, Thomas Kroes, Francois Malan, Frits Post, Noeska Smit, Rick van der Meiden, and Peter van Nieuwenhuizen. Special thanks to the current head of our group Elmar Eisemann, under whose supervision I am now performing a postdoc project on the visualization of floods, a position I also have Gerwin to thank for. Any faculty would be nowhere without proper technical support, which at our group was always handled quickly and with great enthusiasm by Ruud de Jong and Bart Vastenhouw. I would also like to thank our secretaries Onno de Wit, Stefanie van Gentevoort and Toos Brussee-Donders for all their help with the bureaucracy. Thanks as well to Matthijs Sypkens Smit, with whom we shared a room for a couple of months, and always remained a beacon of rest amidst our antics. And of course I can't forget Wim Bronsvoort, who was the only one who also appreciated the greatest sport in the world: cycling. Many of our lunch breaks were filled with talks about this sport that generates an endless stream of interesting facts and stories. Also his supervision in the project with Xin Zhang is greatly appreciated.

You can't finish a PhD on games, without doing plenty of hands-on research in the subject matter.

Thousands and thousands of hours have I spent throughout my lifetime preparing for this, playing many hundreds of games. I couldn't finish these acknowledgements without thanking all the great developers who put their efforts into all these games (efforts that are rewarded by lots of criticism on their development process throughout this work). A special mention to Pyro Studios for making my all-time favorite "Commandos 2: Men Of Courage".

And lastly, besides my friends and family, I would especially like to thank my parents. Without hesitation, they provided me a place to retreat every single weekend of this project: Belgium and specifically my hometown Linter, proved an ideal place to overcome all tribulations of a PhD.

List of Publications

T. Tutenel, R.M. Smelik, R. Bidarra and K.J. de Kraker. The role of semantics in games and simulations. *ACM Computers in Entertainment* 6(4):a57, 2008.

T. Tutenel, R.M. Smelik, R. Bidarra and K.J. de Kraker. Rule-based layout solving and its application to procedural interior generation. *CASA workshop on 3D advanced media in gaming and simulation (3AMIGAS)*, Amsterdam, The Netherlands, June 16, 2009.

R.M. Smelik, K.J. de Kraker, T. Tutenel, R. Bidarra and S.A. Groenewegen. A survey of procedural methods for terrain modelling. *CASA workshop on 3D advanced media in gaming and simulation (3AMIGAS)*, Amsterdam, The Netherlands, June 16, 2009.

J. Kessing, T. Tutenel and R. Bidarra. Services in Game Worlds: A Semantic Approach to Improve Object Interaction. *ICEC 2009 - international conference on entertainment computing*, Paris, France, September 3-5, 2009.

T. Tutenel, R.M. Smelik, R. Bidarra and K.J. de Kraker. Using semantics to improve the design of game worlds. *AIIDE 2009 - 5th conference on artificial intelligence and interactive digital entertainment*, Stanford, CA, USA, October 14-16, 2009.

R. Bidarra, K.J. de Kraker, R.M. Smelik and T. Tutenel. Integrating semantics and procedural generation: key enabling factors for declarative modeling of virtual worlds. *FOCUS K3D Conference on Semantic 3D Media and Content*, Sophia Antipolis, France, February 11-12, 2010.

W.F. Bronsvoort, R. Bidarra, H. van der Meiden and T. Tutenel. The Increasing Role of Semantics in Object Modeling. *Computer-Aided Design and Applications* 7(3), 2010.

T. Tutenel, R.M. Smelik, R. Bidarra and K.J. de Kraker. A Semantic Scene Description Language for Procedural Layout Solving Problems. *AIIDE 2010 - 6th conference on artificial intelligence and interactive digital entertainment*, Stanford, CA, USA, October 11-13, 2010.

R. Lopes, T. Tutenel, R.M. Smelik, K.J. de Kraker and R. Bidarra. A constrained growth method for procedural floor plan generation. *GAME-ON*, Leicester, United Kingdom, November 17-19, 2010.

T. Tutenel, R. van der Linden, M. Kraus, B. Bollen and R. Bidarra. Procedural filters for customization of virtual worlds. *PCGames 2011 - 2nd workshop on procedural content generation in games*, Bordeaux, France, June 28, 2011.

T. Tutenel, R.M. Smelik, R. Lopes, K.J. de Kraker and R. Bidarra. Generating consistent buildings:

a semantic approach for integrating procedural techniques. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3), 2011.

X. Zhang, T. Tutenel, R. Mo, R. Bidarra, and W.F. Bronsvoort. A Method for Specifying Semantics of Large Sets of 3D Models. *GRAPP 2012 - 7th international conference on computer graphics theory and applications*, Rome, Italy, February 24-26, 2012.

J. Kessing, T. Tutenel, and R. Bidarra. Designing Semantic Game Worlds. *PCGames 2012 - 3rd workshop on procedural content generation in games*, Raleigh, NC, USA, May 29, 2012.

N. Kraayenbrink, J. Kessing, T. Tutenel, G. de Haan, F. Marson, S.R. Musse, and R. Bidarra. Semantic Crowds: reusable population for virtual worlds. *Proceedings of the 4th International Conference on Games and Virtual Worlds for Serious Applications (VS-GAMES'12)*, Genoa, Italy, October 29-31, 2012.

Curriculum Vitae

Tim Tutenel was born on May 25th, 1982 in Tienen, Belgium. In 2000, he finished his secundary school at the *St. Tarcisius* college in Zoutleeuw. He received his Master's degree in computer science in 2006 from Hasselt University in Diepenbeek. His thesis, called *Hybrid indoor/outdoor location determination technologies* and supervised by Prof. Dr. Lamotte, researched location determination technologies and more specifically focused on the use of Wireless LAN access points to perform localization on a PDA.

After graduating, he worked as a researcher in the HCI group of Prof. Dr. Coninx at the Expertise Centre for Digital Media (EDM) in Diepenbeek. During that time, he aided in the development of CoGenIVE (Code Generation for Interactive Virtual Environments) for the VR-DeMo project.

In 2007, Tim started his PhD in the Computer Graphics group of the Electrical Engineering, Mathematics and Computer Science Faculty of Delft University of Technology. In his research, that resulted in this thesis, he investigated the impact and the role of semantics on computer game worlds from both the designer's and player's perspective. In this context, he also supervised several Master's and Bachelor's student projects.

He is currently working part-time at Macomi BV and part-time as a postdoc at the Delft University of Technology on the visualization of floods.

Bibliography

[1] Tolga Abaci and Daniel Thalmann. Planning with smart objects. In *Proceedings of the International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG '05)*, pages 25–28, Plzen - Bory, Czech Republic, January-February 2005.

[2] Unknown authors. Most expensive video games. Video game sales wiki, `http://vgsales.wikia.com/wiki/Most_expensive_video_games`, November 2011.

[3] Ruth Aylett and Marc Cavazza. Intelligent virtual environments: a state-of-the-art report. In *Eurographics 2001, STAR Reports volume*, pages 87–109, Manchester, UK, September 2001.

[4] Ruth Aylett, Anthony Horrobin, John O'Hare, Ashraf Osman, and Mikail Polshaw. Virtual teletubbies: reapplying a robot architecture to virtual agents. In *Proceedings of the Third Annual Conference on Autonomous Agents*, pages 338–339, Seattle, WA, USA, May 1999.

[5] Bethesda Game Studios. The Elder Scrolls IV: Oblivion, 2006. Bethesda Softworks.

[6] Rafael Bidarra, Klaas Jan de Kraker, Ruben M. Smelik, and Tim Tutenel. Integrating semantics and procedural generation: Key enabling factors for declarative modeling of virtual worlds. In *Proceedings of the FOCUS K3D Conference on Semantic 3D Media and Content*, Sophia Antipolis - Méditerranée, France, February 2010.

[7] Wesley Bille, Bram Pellens, Frederic Kleinermann, and Olga De Troyer. Intelligent modelling of virtual worlds using domain ontologies. In *Proceedings of MICAI 2004 Workshop of Intelligent Computing*, pages 272–279, Mexico City, Mexico, September 2004.

[8] Barry Bitters. Feature classification system and associated 3-dimensional model libraries for synthetic environments of the future. In *Proceedings of I/ITSEC Conference (NTSA)*, Orlando, FL, USA, December 2002.

[9] Staffan Björk and Jussi Holopainen. *Patterns in game design (Game development series)*. Charles River Media, December 2004.

[10] Willem F. Bronsvoort, Hilderick A. van der Meiden, Rafael Bidarra, and Tim Tutenel. The increasing role of semantics in object modeling. *Computer-Aided Design & Applications*, 7(3):431–440, 2010.

[11] Gulen Cagdas. A shape grammar model for designing row-houses. *Design Studies*, 17(1):35–51, 1996.

[12] Carlos Calderon, Marc Cavazza, and Daniel Diaz. A new approach to the interactive resolution of configuration problems in virtual environments. In *Proceedings of the Third International Symposium on Smart Graphics*, pages 112–122, Heidelberg, Germany, July 2003.

[13] Alex J. Champandard. Living with The Sims' AI: 21 tricks to adopt for your game. `http://aigamedev.com/open/highlights/the-sims-ai/`, October 2007.

[14] Roger Chandler. Looks aren't everything: Making games act real. Gamasutra article, `http://www.gamasutra.com/view/feature/3721/sponsored_feature_looks_arent_.php`, July 2008.

[15] Philippe Charman. Solving space planning problems using constraint technology. In *NATO ASI Constraint Programming: Students' Presentations, TR CS 57/93*, pages 80–96, Tallinn, Estonia, August 1993.

[16] Xuejin Chen, Sing Bing Kang, Ying-Qing Xu, Julie Dorsey, and Heung-Yeung Shum. Sketching reality: Realistic interpretation of architectural designs. *ACM Transactions on Graphics*, 27:11:1–11:15, May 2008.

[17] António Fernando Coelho, António Augusto de Sousa, and Fernando Nunes Ferreira. Modelling urban scenes for lbms. In *Web3D '05: Proceedings of the* 10^{th} *International Conference on 3D Web Technology*, pages 37–46, Bangor, UK, March-April 2005. ACM.

[18] Karen Coninx, Olga De Troyer, Chris Raymaekers, and Frederic Kleinermann. Vr-demo: a tool-supported approach facilitating flexible development of virtual environments using conceptual modelling. In *Proceedings of Virtual Concept*, pages 30–42, Capri, Italy, September 2006.

[19] G. Rick de Ridder. Simulating urban area development for semantic game worlds. Master's thesis, Delft University of Technology, July 2012.

[20] Oliver Deussen, Pat Hanrahan, Bernd Lintermann, Radomír Měch, Matt Pharr, and Przemyslaw 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, July 1998. ACM.

[21] David S. Ebert, Steven Worley, Forest K. Musgrave, Darwyn Peachey, and Ken Perlin. *Texturing & Modeling, a Procedural Approach*. Elsevier, 3^{rd} edition, 2003.

[22] Epic Games. Unreal Engine 3. Available from `http://www.unrealtechnology.com`.

[23] Dieter Finkenzeller. Detailed building façades. *IEEE Computer Graphics and Applications*, 28(3):58–66, 2008.

[24] Dieter Finkenzeller and Jan Bender. Semantic representation of complex building structures. In *Proceedings of CGV '08: Computer Graphics and Visualization*, pages 259–264, Amsterdam, The Netherlands, July 2008.

[25] Veronique Gaildrat. Declarative modelling of virtual environments, overview of issues and applications. In *Proceedings of the* 10^{th} *International Conference on Computer Graphics and Artificial Intelligence (3IA 2007)*, pages 5–15, Athens, Greece, May 2007.

[26] Marina Gavrilova and Jon Rokne. Collision detection optimization in a multi-particle system. In *Computational Science - ICCS 2002*, volume 2331 of *Lecture Notes in Computer Science*, pages 105–114. Springer Berlin / Heidelberg, 2002.

[27] John S. Gero and Udo Kannengiesser. A function–behavior–structure ontology of processes. *AI EDAM: Artificial Intelligence for Engineering Design, Analysis, and Manufacturing*, 21(4):379–391, November 2007.

[28] James J. Gibson. *The Ecological Approach To Visual Perception*. Psychology Press, new edition edition, September 1986.

[29] Michael Gosele and Wolfgang Stürzlinger. Semantic constraints for scene manipulation. In *Proceedings of* 15^{th} *Spring Conference on Computer Graphics*, pages 140–146, Budmerice, Slovakia, April-May 1999.

[30] Stefan Greuter, Jeremy Parker, Nigel Stewart, and Geoff Leach. Real-time procedural generation of 'pseudo infinite' cities. In *GRAPHITE '03: Proceedings of the First International Conference on Computer Graphics and Interactive Techniques in Australasia and Southeast Asia*, pages 87–94, Melbourne, Australia, February 2003. ACM.

[31] Mario Gutierrez, Frederic Vexo, and Daniel Thalmann. Semantics-based representation of virtual environments. *International Journal of Computer Applications in Technology*, 23(2/3/4):229–238, 2005.

[32] Evan Hahn, Prosenjit Bose, and Anthony Whitehead. Persistent realtime building interior generation. In *Sandbox 2006: Proceedings of the ACM SIGGRAPH Symposium on Videogames*, pages 179–186, Boston, MA, USA, July 2006. ACM.

[33] Christopher Hanna, Raymond Hickey, Darryl Charles, and Michaela Black. Modular reinforcement learning architectures for artificially intelligent agents in complex game environments. In *Proceedings of the IEEE Conference on Computational Intelligence and AI in Games (CIG 2010)*, pages 380–387, Copenhagen, Denmark, August 2010.

[34] Kazuyoshi Honda and Funio Mizoguchi. Constraint-based approach for automatic spatial layout planning. In *Proceedings of the* 11^{th} *Conference on Artificial Intelligence for Applications*, pages 38–45, Washington, DC, USA, February 1995. IEEE Computer Society.

[35] Michael N. Huhns and Munindar P. Singh. Agents on the web: ontologies for agents. *IEEE Internet Computing*, 1(6):81–83, 1997.

[36] Jesús Ibánez-Martínez and Carlos Delgado-Mata. A basic semantic common level for virtual environments. *International Journal of Virtual Reality*, 5(3):25–32, September 2006.

[37] Jesús Ibánez-Martínez and Carlos Delgado-Mata. Virtual environments and semantics. *UPGRADE*, 7(2):17–23, April 2006.

[38] Filter Forge Inc. Filter Forge. Available from `http://www.filterforge.com/`, 2012.

[39] Irrational Games. Bioshock, 2007. 2K Games.

[40] James Golding - Epic Games. Building blocks artist driven procedural buildings - game developers conference 2010. Available from `http://gdcvault.com/play/1012655/Building-Blocks-Artist-Driven-Procedural`, 2010.

[41] Marcelo Kallmann. *Object interaction in real-time virtual environments*. PhD thesis, Infoscience — Ecole Polytechnique Federale de Lausanne [http://infoscience.epfl.ch/oai2d.py] (Switzerland), 2001.

[42] Marcelo Kallmann and Daniel Thalmann. Modeling objects for interaction tasks. In *Proceedings of the Eurographics Workshop on Animation and Simulation*, pages 73–86, Lisbon, Portugal, August-September 1998.

[43] Marcelo Kallmann and Daniel Thalmann. Direct 3D interaction with smart objects. In *Proceedings of the ACM symposium on Virtual reality software and technology*, VRST '99, pages 124–130, London, UK, December 1999. ACM.

[44] George Kelly and Hugh McCabe. Citygen: An interactive system for procedural city generation. In *Proceedings of GDTW 2007: the Fifth Annual International Conference in Computer Game Design and Technology*, pages 8–16, Liverpool, UK, November 2007.

[45] Jassin Kessing, Tim Tutenel, and Rafael Bidarra. Services in game worlds: a semantic approach to improve object interaction. In *Proceedings of the International Conference on Entertainment Computing*, pages 276–281, Paris, France, September 2009.

[46] Jassin Kessing, Tim Tutenel, and Rafael Bidarra. Designing semantic game worlds. In *Proceedings of the third workshop on Procedural Content Generation in Games (PCG 2012)*, Raleigh, NC, USA, May 2012.

[47] Hank Koning and Julie Eizenberg. The language of the prairie: Frank lloyd wright's prairie houses. *Environment and Planning B: Planning and Design*, 8(3):295–323, 1981.

[48] Nick Kraayenbrink, Jassin Kessing, Tim Tutenel, Gerwin de Haan, Fernando Marson, Soraia R. Musse, and Rafael Bidarra. Semantic crowds: reusable population for virtual worlds. In *Proceedings of the Fourth International Conference on Games and Virtual Worlds for Serious Applications (VS-GAMES'12)*, 2012.

[49] Doo Young Kwon. Archidna: A generative system for shape configuratons. Master's thesis, University of Washington, 2003.

[50] Marc Erich Latoschik, Peter Biermann, and Ipke Wachsmuth. Knowledge in the loop: Semantics representation for multimodal simulative environments. In *Proceedings of the 5th International Symposium on Smart Graphics 2005*, pages 25–39, Munich, Germany, August 2005.

[51] Libby Levison. *Connecting Planning And Acting Via Object-Specific Reasoning*. PhD thesis, University of Pennsylvania, 1996.

[52] Ricardo Lopes and Rafael Bidarra. A Semantic Generation Framework for Enabling Adaptive Game Worlds. In *ACE '11: 8^{th} International Conference on Advances in Computer Entertainment Technology*, New York, 2011. ACM.

[53] Ricardo Lopes, Tim Tutenel, Ruben M. Smelik, Klaas Jan de Kraker, and Rafael Bidarra. A constrained growth method for procedural floor plan generation. In *Proceedings of GAME-ON 2010, the 11^{th} International Conference on Intelligent Games and Simulation*, Leicester, UK, November 2010. EUROSIS.

[54] Jianye Lu, Athinodoros S. Georghiades, Andreas Glaser, Hongzhi Wu, Li-Yi Wei, Baining Guo, Julie Dorsey, and Holly Rushmeier. Context-aware textures. *ACM Transactions on Graphics*, 26, January 2007.

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

[56] Fernando Marson and Soraia R. Musse. Automatic generation of floor plans based on squarified treemaps algorithm. *IJCGT International Journal on Computers Games Technology*, 2010:1–10, January 2010.

[57] Jess Martin. Procedural house generation: a method for dynamically generating floor plans. I3D '06: Poster Proceedings of the 2006 SIGGRAPH Symposium on Interactive 3D Graphics and Games, March 2006.

[58] Manish Mehta and Ashwin Ram. Runtime behavior adaptation for real-time interactive games. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(3):187–199, 2009.

[59] Paul Merrell, Eric Schkufza, and Vladlen Koltun. Computer-generated residential building layouts. *ACM Transactions on Graphics*, 29(5):181:1–181:12, 2010.

[60] Paul Merrell, Eric Schkufza, Zeyang Li, Maneesh Agrawala, and Vladlen Koltun. Interactive furniture layout using interior design guidelines. In *SIGGRAPH '11: Proceedings of the 38^{th} Annual Conference on Computer Graphics and Interactive Techniques*, pages 87:1–87:10, Vancouver, Canada, August 2011. ACM.

[61] Ian Millington and John Funge. *Artificial intelligence for games*. Morgan Kaufmann Publishers, 2^{nd} edition, 2009.

[62] Masahiro Mori. Bukimi no tani The Uncanny valley (originally in Japanese). *Energy*, 7(4):33–35, 1970.

[63] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural modeling of buildings. In *SIGGRAPH '06: Proceedings of the 33^{rd} Annual Conference on Computer Graphics and Interactive Techniques*, pages 614–623, Boston, MA, USA, July-August 2006. ACM.

[64] Pascal Müller, Gang Zeng, Peter Wonka, and Luc Van Gool. Image-based procedural modeling of façades. In *SIGGRAPH '07: Proceedings of the* 34^{th} *Annual Conference on Computer Graphics and Interactive Techniques*, volume 26, pages 85:1–85:10, San Diego, CA, USA, August 2007. ACM.

[65] F. Kenton Musgrave, Craig E. Kolb, and Robert S. Mace. The synthesis and rendering of eroded fractal terrains. In *SIGGRAPH '89: Proceedings of the* 16^{th} *Annual Conference on Computer Graphics and Interactive Techniques*, pages 41–50, Boston, MA, USA, July-August 1989. ACM.

[66] Eric Paquette, Pierre Poulin, and George Drettakis. The simulation of paint cracking and peeling. In W. Stuerzlinger and M. McCool, editors, *Proceedings of Graphics Interface*, pages 59–68, 2002.

[67] Yoav I. H. Parish and Pascal Müller. Procedural modeling of cities. In *SIGGRAPH '01: Proceedings of the* 28^{th} *Annual Conference on Computer Graphics and Interactive Techniques*, pages 301–308, Los Angeles, CA, USA, August 2001. ACM.

[68] Ken 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, San Francisco, CA, USA, July 1985. ACM.

[69] Ken Perlin. Improving noise. In *SIGGRAPH '02: Proceedings of the* 29^{th} *Annual Conference on Computer Graphics and Interactive Techniques*, pages 681–682, San Antonio, TX, USA, July 2002. ACM.

[70] Christopher Peters, Simon Dobbyn, Brian MacNamee, and Carol O'Sullivan. Smart objects for attentive agents. In *Proceedings of the International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG '03)*, pages 1–8, Plzen - Bory, Czech Republic, February 2003.

[71] Charles E. Pfefferkorn. A heuristic problem solving design system for equipment or furniture layouts. *Communications of the ACM*, 18:286–297, May 1975.

[72] Procedural. CityEngine. Available from `http://www.procedural.com`, 2011.

[73] Andrew Rau-Chaplin, Brian Mackay-Lyons, and Peter F. Spierenburg. The LaHave house project: Towards an automated architectural design service. In *CADEX '96: Proceedings of the International Conference on Computer-Aided Design*, pages 25–31, Hagenberg, Austria, September 1996.

[74] Andrew Rau-Chaplin and Trevor J. Smedley. A graphical language for generating architectural forms. In *VL '97: Proceedings of the 1997 IEEE Symposium on Visual Languages*, VL '97, pages 260–267, Capri, Italy, September 1997. IEEE Computer Society.

[75] William T. Reeves. Particle systems - a technique for modeling a class of fuzzy objects. *ACM Transactions on Graphics*, 2(2):91–108, 1983.

[76] Michael A. Rosenman and John S. Gero. Modelling multiple views of design objects in a collaborative cad environment. *Computer-Aided Design*, 28:193–205, 1996.

[77] Olivier Le Roux, Véronique Gaildrat, and René Caubet. *Geometric Modeling: Techniques, Applications, Systems and Tools*, chapter Using Constraint Satisfaction Techniques in Declarative Modelling, page 20. Kluwer Academic Publishers, Dordrecht Hardbound, 2004.

[78] Barry G. Silverman, Michael Johns, Jason Cornwell, and Kevin O'Brien. Human behavior models for agents in simulators and games: Part i: Enabling science with pmfserv. *Presence: Teleoperators and Virtual Environments*, 15(2):139–162, April 2006.

[79] Jake Simpson. Scripting and Sims 2: Coding the psychology of little people. In *Game Developers Conference*, 2005.

[80] Ruben M. Smelik. *A Declarative Approach to Procedural Generation of Virtual Worlds*. PhD thesis, Delft University of Technology, November 2011.

[81] Ruben M. Smelik, Klaas Jan de Kraker, Tim Tutenel, Rafael Bidarra, and Saskia A. Groenewegen. A survey of procedural methods for terrain modelling. In *3AMIGAS: Proceedings of the CASA 2009 Workshop on 3D Advanced Media in Gaming and Simulation*, pages 25–34, Amsterdam, The Netherlands, June 2009.

[82] Ruben M. Smelik, Tim Tutenel, Klaas Jan de Kraker, and Rafael Bidarra. Declarative terrain modeling for military training games. *International Journal of Computer Game Technology (IJCGT)*, 2010:1–11, 2010.

[83] Ruben M. Smelik, Tim Tutenel, Klaas Jan de Kraker, and Rafael Bidarra. Integrating procedural generation and manual editing of virtual worlds. In *PCGames '10: Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, pages 1–8, Monterey, CA, USA, June 2010. ACM.

[84] Ruben M. Smelik, Tim Tutenel, Klaas Jan de Kraker, and Rafael Bidarra. Interactive creation of virtual worlds using procedural sketching. In *Proceedings of Eurographics 2010: Short Papers*, Norrköping, Sweden, May 2010. Eurographics Association.

[85] Ruben M. Smelik, Tim Tutenel, Klaas Jan de Kraker, and Rafael Bidarra. A Declarative Approach to Procedural Modeling of Virtual Worlds. *Computers & Graphics*, 35(2):352–363, April 2011.

[86] Graham Smith, Tim Salzman, and Wolfgang Stuerzlinger. 3D scene manipulation with 2D devices and constraints. In *No description on Graphics interface 2001*, GRIN'01, pages 135–142, Toronto, Ont., Canada, Canada, 2001. Canadian Information Processing Society.

[87] SpeedTree. Speedtree: Image gallery from games. `http://www.speedtree.com/gallery/`, June 2011.

[88] Tim Tutenel, Rafael Bidarra, Ruben M. Smelik, and Klaas Jan de Kraker. The role of semantics in games and simulations. *ACM Computers in Entertainment*, 6:1–35, 2008.

[89] Tim Tutenel, Rafael Bidarra, Ruben M. Smelik, and Klaas Jan de Kraker. Rule-based layout solving and its application to procedural interior generation. In *3AMIGAS: Proceedings of the CASA 2009 Workshop on 3D Advanced Media in Gaming and Simulation*, pages 15–24, Amsterdam, The Netherlands, June 2009.

[90] Tim Tutenel, Rafael Bidarra, Ruben M. Smelik, and Klaas Jan de Kraker. A semantic scene description language for procedural layout solving problems. In *AIIDE '10: Proceedings of the* 6^{th} *Conference on Artificial Intelligence and Interactive Digital Entertainment*, Stanford, CA, USA, October 2010.

[91] Tim Tutenel, Ruben M. Smelik, Klaas Jan de Kraker, and Rafael Bidarra. Using semantics to improve the design of game worlds. In *AIIDE '09: Proceedings of the* 5^{th} *Conference on Artificial Intelligence and Interactive Digital Entertainment*, Stanford, CA, USA, October 2009.

[92] Tim Tutenel, Ruben M. Smelik, Ricardo Lopes, Klaas Jan de Kraker, and Rafael Bidarra. Generating consistent buildings: a semantic approach for integrating procedural techniques. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):274–288, 2011.

[93] Tim Tutenel, Roland van der Linden, Marnix Kraus, Bart Bollen, and Rafael Bidarra. Procedural filters for customization of virtual worlds. In *PCG '11: Proceedings of the 2011 Workshop on Procedural Content Generation in Games*, Bordeaux, France, June-July 2011. ACM.

[94] Lode Vanacken, Chris Raymaekers, and Karin Coninx. Introducing semantic information during conceptual modelling of interaction for virtual environments. In *Proceedings of the 2007 workshop on Multimodal interfaces in semantic interaction*, WMISI '07, pages 17–24, Aveiro, Portugal, April 2007. ACM.

[95] Benjamin Watson, Pascal Müller, Oleg Veryovka, Andy Fuller, Peter Wonka, and Chris Sexton. Procedural urban modeling in practice. *IEEE Computer Graphics and Applications*, 28:18–26, 2008.

[96] Basil Weber, Pascal Müller, Peter Wonka, and Markus Gross. Interactive geometric simulation of 4d cities. *Computer Graphics Forum: Proceedings of Eurographics 2009*, 28:481–492, April 2009.

[97] Peter Wonka, Michael Wimmer, François Sillion, and William Ribarsky. Instant architecture. In *SIGGRAPH '03: Proceedings of the* 30^{th} *Annual Conference on Computer Graphics and Interactive Techniques*, pages 669–677, San Diego, CA, USA, July-August 2003. ACM.

[98] Ken Xu, James Stewart, and Eugene Fiume. Constraint-based automatic placement for scene composition. In *Graphics Interface*, pages 25–34, 2002.

[99] Liu Yong, Xu Congfu, Pan Zhigeng, and Pan Yunhe. Semantic modeling project: Building vernacular house of southeast china. In *VRCAI '04: Proceedings of the 2004 ACM SIGGRAPH International Conference on Virtual Reality Continuum and its Applications in Industry*, pages 412–418, Nanyang, Singapore, June 2004. ACM.

[100] Lap-Fai Yu, Sai Kit Yeung, Chi-Keung Tang, Demetri Terzopoulos, Tony F. Chan, and Stanley Osher. Make it home: Automatic optimization of furniture arrangement. In *SIGGRAPH '11: Proceedings of the* 38^{th} *Annual Conference on Computer Graphics and Interactive Techniques*, pages 86:1–86:12, Vancouver, Canada, August 2011. ACM.

[101] Xin Zhang, Tim Tutenel, Rong Mo, Rafael Bidarra, and Wim Bronsvoort. Specifying semantics of large sets of 3D models. In *Proceedings of the International Conference on Computer Graphics Theory and Applications (GRAPP 2012)*, Rome, Italy, February 2011.

The visual quality of game worlds increased massively in the last three decades. However, the closer game worlds depict reality, the more noticeable it is for gamers when objects do not behave accordingly.

An important problem is that the data of a game world is often scattered across different components of the game engine. What lacks is a common semantic representation that can act as the *glue* between these components (see cover).

In this thesis we define *semantic game worlds* as game worlds that are populated with objects enriched with semantics. They offer game programmers a consistent representation that can be shared by game engine components. Moreover, this representation offers designers a way to moderate conflicts between procedural content generation techniques and it enables them to specify object behavior in a generic manner.

ISBN: 978-94-6203-259-0