

xx (178886.1)



CINVESTAV
BIBLIOTECA CENTRAL

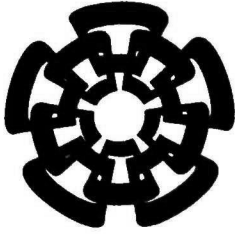


SSIT000009524

TK165 98

.237

2009



**CINVESTAV
IPN
ADQUISICION
DE LIBROS**

Centro de Investigación y de Estudios Avanzados del I.P.N.
Unidad Guadalajara

**Modelado Declarativo Auxiliado por
Conocimiento - Declarative Modeling
Based On Knowledge**

Tesis que presenta:

Jaime Alberto Zaragoza Rios

para obtener el grado de:

Doctor en Ciencias

en la especialidad de:

Ingeniería Eléctrica

Directores de Tesis

Dr. Félix Francisco Ramos Corchado

M.C. Veronique Gaildrat

Guadalajara, Jalisco, Diciembre de 2009.

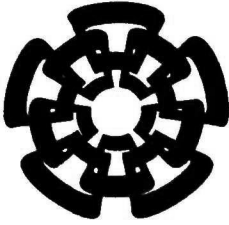


CENTRO DE INVESTIGACIÓN Y
DE ESTUDIOS AVANZADOS DEL
INSTITUTO POLITÉCNICO
NACIONAL

COORDINACIÓN GENERAL DE
SERVICIOS BIBLIOGRÁFICOS

CLASIF.: TK165. G8 .237.2009
ADQUIS.: SSI-602
FECHA: 21 Mayo - 2010
PROCED.: DON. - 2010
\$

L64600-1001



Centro de Investigación y de Estudios Avanzados

del I.P.N.

Unidad Guadalajara

**Modelado Declarativo Auxiliado por
Conocimiento - Declarative Modeling
Based On Knowledge**

A thesis presented by:

Jaime Alberto Zaragoza Rios

to obtain the degree of:

Doctor in Science

in the subject of:

Electrical Engineering

Thesis Advisors:

Dr. Félix Francisco Ramos Corchado

M. C. Véronique Gaildrat

Guadalajara, Jalisco, December 2009.

Modelado Declarativo Auxiliado por Conocimiento - Declarative Modeling Based On Knowledge

**Tesis de Doctorado en Ciencias
Ingeniería Eléctrica**

Por:

Jaime Alberto Zaragoza Rios
Maestro en Ciencias

Centro de Investigación y de Estudios Avanzados del I.P.N.,
Unidad Guadalajara 2004-2006

Becario de CONACYT, expediente no. 190965

Directores de Tesis

Dr. Félix Francisco Ramos Corchado
M.C. Veronique Gaildrat

Modelado Declarativo Auxiliado por Conocimiento - Declarative Modeling Based On Knowledge

**Doctor of Science Thesis
In Electrical Engineering**

By:

Jaime Alberto Zaragoza Rios

Master in Computer Science

**Centro de Investigación y de Estudios Avanzados del
I.P.N., Unidad Guadalajara 2004-2006**

Scholarship granted by CONACYT, No. 190965

Thesis Advisors:

Dr. Félix Francisco Ramos Corchado

Dr. Véronique Gaildrat

Doctor in Sciences Thesis in Electrical Engineering

Presented by:

M.C. Jaime Alberto Zaragoza Rios

to obtain the degree of:

Doctor in Sciences

in the specialty of:

Electrical Engineering

Dr. Félix Francisco Ramos Corchado
Thesis Advisor

M. de C. Véronique Gaildrat
Thesis Co-Advisor

Dr. José Luis Leyva Montiel
Sinodal

Dr. Luis Ernesto López Mellado
Sinodal

Dr. Juan Manuel Ramirez
Arredondo
Sinodal

Prof. Jean-Luc Koning
Sinodal

Dr. Marco Antonio Ramos Corchado
Sinodal

December 15, 2009

Acknowledgments

I would like to thank National Council on Science and Technology, CONACyT, for providing PhD Scholarship number 1910965. This research is also partially supported by CoECyT-Jal project No. 2008-05-97094.

I would also like to thank my thesis supervisors, Dr. Félix Francisco Ramos Corchando, and M. de C. Véronique Gaidrat, for their valuable guidance in the completion of this research.

Also special thanks to my co-workers from the Distributed Systems group at CINVESTAV, as well as the people from the VORTEX Lab at IRIT, in Toulouse, France.

Dedicated to my parents, Jaime Zaragoza Infante and Margarita Rios Gonzáles.

Resumen

La tecnología moderna ha permitido la creación y representación de *Mundos Virtuales* y criaturas con un alto nivel de detalle, tal que vistos en películas, a veces es difícil distinguir cuales elementos son generados por computadora y cuales no. Así mismo, los juegos de vídeo han alcanzado un nivel cercano al realismo fotográfico.

Sin embargo, tal tecnología está en manos de habilidosos diseñadores, artistas y programadores, para los cuales toma de semanas a años para obtener esos resultados.

Modelado Declarativo es un método que permite crear modelos especificando tan solo algunas propiedades para los componentes del mismo. Aplicado a la creación de Mundos Virtuales, el modelado declarativo puede ser usado para construir el mundo virtual, estableciendo la disposición de los objetos, generando el contexto necesario para incluir animación y diseño de escena, así como generar las salidas usadas por un sistema de visualización/animación.

Este documento presenta una investigación enfocada a explorar el uso del modelado declarativo para crear *Ambientes Virtuales*, usando *Explotación del Conocimiento* como apoyo para el proceso y facilitar la transición del modelo de datos a una arquitectura subyacente, que toma la tarea de animar y evolucionar la escena.

Summary

Modern technology has allowed the creation and presentation of *Virtual Worlds* and creatures with such a high level of detail, that when used in films, sometimes is difficult to tell which elements are computer-generated and which not. Also, videogames had reached a level close to photographic realism.

However, such technology is in the hands of skillful designers, artists, and programmers, for whom it takes from weeks to years to complete these results.

Declarative modeling is a method which allows to create models specifying just a few properties for the model components. Applied to Virtual World creation, declarative modeling can be used to construct the Virtual World, establishing the layout for the objects, generating the necessary context to provide animation and scene design, and generating the outputs used by a visualization/animation system.

This document presents a research devoted to explore the use of declarative modeling for creating *Virtual Environments*, using *Knowledge Exploitation* to support the process and ease the transition from the data model to an underlying architecture which takes the task of animating and evolving the scene.

Contents

1	Introduction	1
1.1	Introduction	2
1.2	The Problem	2
1.3	Description of Problem	3
1.4	Research Objectives .	5
2	State of the Art	7
2.1	Technical Introduction	8
2.2	Declarative modeling	9
2.2.1	Description	10
2.2.2	Generation	11
2.2.3	Look Up	13
2.3	Knowledge Management	13
2.3.1	Ontolingua	15
2.3.2	WebOnto	16
2.3.3	Protégé	16
2.3.4	Web Ontology Language	16
2.4	Constraint Satisfaction Problems	17
2.4.1	Backtracking	20
2.4.2	Backmarking	20
2.4.3	Backjumping	21
2.4.4	Backjumping based on graphics	21

2.4.5	Forward Checking	22
2.5	Virtual Environment	22
2.6	Related Works	24
2.6.1	WordsEye: Automatic text-to-scene conversion system	25
2.6.2	DEM ² ONS: High Level Declarative Modeler for 3D Graphic Applications	25
2.6.3	Multiformes: Declarative Modeler as 3D sketch tool	26
2.6.4	CAPS: Constraint-based Automatic Placement System	26
2.6.5	ALICE	27
3	Proposal	28
3.1	Interaction Language: A Review of VEDEL	29
3.2	Parsing Methodology	31
3.3	Modeler's Architecture	32
3.4	Creating the Model	34
3.4.1	Model Data Structure	34
3.4.2	Modeler procedure	35
3.4.3	Geometrical Validation	39
3.5	Generation of the Outputs	43
3.5.1	Model-View Controller	43
3.6	Modifying the Model	45
4	Research Outcome	46
4.1	Virtual Environment Editor Prototypes	47
4.1.1	GeDA-3D Virtual Environment Editor Prototype	49
4.1.2	DRAMA Project Module DRAMAScène	53
5	Conclusions	56
5.1	Conclusions	57
5.1.1	Future Work .	60
	Bibliography	62

List of Figures

1.1	Project Overview.	4
2.1	Fields of research.	9
2.2	Interactive process of declarative modeling.	10
2.3	Characterization for an object	12
2.4	Ontology example.	18
2.5	Different levels of detail for avatars.	23
2.6	FL-System, City Engine and Instant Architecture	24
3.1	VEDEL examples.	31
3.2	Parsed Entry Structure	32
3.3	Modeler Architecture	33
3.4	Model Structure.	35
3.5	Collision Tags	40
3.6	Characteristic points	41
3.7	Validation volume for equation 1.b	42
3.8	Special case: against	43
3.9	Special case: inside	44
4.1	Virtual Environment Editor GUI	47
4.2	Previous Prototypes: Battle of the Frogs	48
4.3	Previous Prototypes: Earlier version of GeDA-3D	48
4.4	Example 1: Top-Down view	50

4.5	Example 1: General View	50
4.6	Example 2: House environment	51
4.7	Example 2: House environment	52
4.8	Example 3: Detail view	53
4.9	Example 3: Top-Down View	53
4.10	Examples of DRAMAScène Concepts	55
5.1	GEDA-3D Architecture	60

Chapter 1

Introduction

Abstract

We present the objectives for this research, the motivation that leads us to perform research in the field of declarative modeling, the goals to be fulfilled, and the problems that must be solved in order to reach that objective. We also expose the results of our previous research.

1.1 Introduction

There has always been the need to represent ideas, in order to transmit, preserve, and make them available to others. Oral language was the first method to achieve these goals, followed by painting, and then writing. Any of these methods is enough when the ideas represented are easy to express. However, as these ideas become more and more complex, methods to represent them also become more and more specialized.

Fortunately, these methods can be implemented as tools. However, the complexity of tools useful to handle the representation of complex ideas needs a learning period, going from simply understanding the way to hold the tool properly such as pencils, to different ways to achieve the desired results. Also, each tool can be composed of different material, and applied in a number of ways, on different elements.

These tools have evolved through time, reaching rich versions that are implemented through computational systems, which allow a greater flexibility in their usage, even in ways that are not possible in the physical world using manual tools. In these ways, modern tools allow representing almost any kind of idea, from entertainment to education, and from visual aid to formal training. In computer science, *Virtual Reality* (VR) is a discipline which is useful for representing an actual or syntetic world, and can be perceived by the users in a variety of forms: text, sound, visuals, or even sensations.

1.2 The Problem

However, creating *Virtual Worlds* (VW) is a complex task carried out by a complete staff of modelers, programmers and artists. Completing a project can take from a few days to several years. Those projects can go from custom presentations for small-business clients to big productions, like movies or video games. The tools needed to create these VW have different degrees of complexity, which forces the users to pass through training, taking from a few hours to several weeks. In addition, some of these tools require specialized input hardware, that in some cases is costly and difficult to use correctly.

A final user who desires to use a VR may feel intimidated by the cost of having a staff to develop a complex application or discouraged by the learning step necessary to obtain satisfactory results for simple applications. Also, developers must have a certain degree of skill or talent to create results that approach the original idea. Thus, non-expert the users can find it difficult to create the VWs they intent to use and may prefer to leave the task to experienced creators or use other options. However, final the users are the ones who really need VR technology.

To ease the taks of creating a VW we propose the creation of a tool which will use the

necessary procedures to create such VWs, using as input only a set of properties for the world defined by the user. The VW can later be extended to a *Virtual Environment* (VE), where the entities include in the VW perform actions, display emotions, and are subject to changes made by a set of rules established by the users.

To tackle the problem we divide the problem of using VR technology in several steps: Creating the VW, specifying the scene, that is, the actions to take place in the VE, and visualizing the scene. We focus just on the first two of these subproblems.

The document is organized as follows:

- Chapter 1, **Introduction**, presents a general view of the research project, and introduces the motivation, goals, and solutions proposed.
- Chapter 2, **State of the Art**, describes previous and current studies on Declarative Modeling, geometric constraint solvers, and knowledge management, as well as some literature on the subject.
- Chapter 3, **Proposed Solution**, states our approach in detail, the methods proposed to solve the problem we are trying to solve, the research conducted to validate such approach, and the selection of the methods that better suit the objectives of our research.
- Chapter 4, **Research Outcome**, presents the outcome of this research, the prototypes developed.
- Chapter 5, **Conclusions**, raises some future objectives to be solved in future researches.

1.3 Description of Problem

This paper proposes, formalizes and implements a method creating VEs, using simple user inputs in the form of descriptions, with a formalization close to natural language, and exploiting knowledge to validate the statements of the input, with the objective of generating a model of VE, and finally presents it to the users by means of a 3D viewer, an underlying architecture, or any desired platform.

For VE creation we understand contrut a simulated space, ruled by a set of laws (friction, gravity, elasticity, etc.) where several animated and unanimated entities can dwell and perform actions that may affect other entities and/or the environment.

Our approach uses *Declarative Modeling* (DM) to create a VE. This is a very powerful technique allowing the user to describe the scene to be designed in an intuitive manner, by giving only some expected properties for the scenario, and letting the modeler find one or several solutions, if any, that satisfy these properties [1]. Thus in our case, the VE is

created using a natural language as a description provided by final the users. The difference between our approach and those proposed in related researches is that we propose the use of a *Knowledge Database* (KB) needed to validate the input and generate the output during the process.

The DM process is usually formed by three phases [2]:

- *Description*. Defines the interaction language.
- *Generation*. The modeler generates one or more scenes that match what the users describe.
- *Insight*. Users are presented with the models, then they can choose a solution.

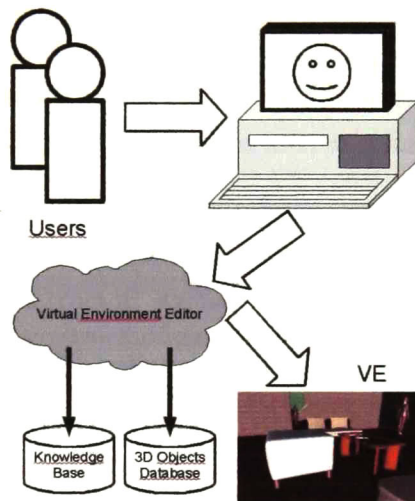


Figure 1.1: Project Overview.

In our previous research we focused on the *Description* phase of the DM technique. We defined a language centered in the creation of *Virtual Scenarios* (VS) and developed a tool that can analyze the descriptions written in such language. To present a visual outcome of the description, we used the rendering machine provided by the GeDA-3D [3] architecture,

which also hosts the tools for evolving the scene. In addition to the language, a KB was constructed, and used in the parser to process the terms in the description, and generated the appropriate outputs.

Our current research is focused on the two last steps of DM. This means that once all the possible properties to the environment and to the entities that will dwell in it are assigned, we need to validate the positioning of each entity in the scenario. This will be conducted over the model generated in the previous step, where a tentative positioning is conducted, although it is not validated. The model can be processed using two tools: A *Constraint Solution Problem* (CSP) Solving Algorithm or a Geometric Constraint Solver. The second option involves the construction of a specialized tool or using a commercial product to solve possible model conflicts. The first method implies defining and structuring a method to solve a CSP.

While the latter option involves using widely tested tools, the cost and the necessary changes in the process where a great drawback. The former option allows better adaption of tools to our project, because the KB will also contains constraint information for validating the spatial relationships between entities and the environment.

A lexical-syntactic parser, a model creator, an inference machine and an output generator form the *Virtual Environment Editor* (VEE), our VE constructor tool. The model creator is formed by a declarative modeler and a CSP solving algorithm, which validates the description and generates the possible models, and allows the modification of the same.

Finally for the context constructor, we need to consider the rest of the GeDA-3D architecture [3], which includes several modules on which the modeler depends, and that are also dependent on the modeler. The VW created through the scene editor contains all the information necessary for the architecture to make a correct representation of the VW.

1.4 Research Objectives

We consider several objectives to be reached during the development of this research. Some of these objectives are oriented to develop novel declarative methods for VE generation, while others are focused on solving the problems for constructing our use case, which is a VEE necessary to validate our proposal. We list the objectives for this research next:

- Defining a method for integrating knowledge exploitation into DM.
- Integrating the use of knowledge in a CSP solver algorithm.
- Establishing the necessary information to be included in a KB, in order to create a model for a VE.

- Designing the architecture for a VEE, based on DM, which can receive an input based on a language specifically defined for VE description.
- Including in the VEE architecture the necessary means to access a KB.
- Adding methods in the VEE architecture to allow the generation of different types of outputs, in such a way that those outputs can be added, modified or removed without modifying the modeler itself.
- Implementing the proposed methods for DM and CSP solving into the architecture designed for the VEE.
- Including into the design of the VEE the necessary means to allow the creation of VE in a number of ways, from standard input text, to haptic devices input.
- Integrating the VEE with the rest of the GeDA-3D architecture.

The final result of this research is to propose a friendly, easy to use tool based on DM for final, non-experienced the users (Figure 1.1). Thus, it must be possible to use the solution obtained through our method as an input of a 3D viewer, an underlying architecture, or any desired platform.

Chapter 2

State of the Art

Abstract

In this chapter we expose the methodologies used in our research. First, we present in detail the DM method. Next, we explore the different approaches for knowledge representation and exploitation. Later, different methods for solving Constraint Satisfaction Problem are presented. Finally, we take on some concepts for VR, and review current researches in the area.

2.1 Technical Introduction

Before we start detailing the methodology used and the implementation procedure employed for creating the *Virtual Modeler* (VM), we need to explain some concepts for a better understanding of our project.

A **User Interface** (UI) is the aggregation of means employed by the users to interact with a system. It provides the methods for input, allowing the manipulation of the system, the output, or the presentation of the effects resulting of the users interaction.

The **Backus-Naur Form**, or BNF, is a formal way to describe formal languages. Consists of a context free grammar to define the syntax of a programming language by using two sets of rules: i.e., lexical rules and syntactic rules. The EBNF or Extended Backus-Naur Form is a metasyntax notation used to express context-free, an extension of the basic Backus-Naur Form (BNF) metasyntax notation.

A **Token** is a block of text that can be categorized. This block of text can also be known as a lexeme. Through categorization, a lexical analyzer processes lexemes and provides meaning to them. This is known as tokenization. A token can have any kind of presentation, as long as it is a useful part of the structured text.

An **Application Programming Interface** or API, is a set of standardized requests. In essence, it provides the methods for accessing a program services. An API is formed by routines, data structures, object classes and/or protocols provided by libraries and/or operating system services.

A **Model-View Controller**, or MVC, is a paradigm where the user inputs a model of the external world, and the visual feedback to the users is explicitly separated and handled by three types of objects, each of them specialized for its task. The view manages the graphical and/or textual output, the controller interprets the inputs from the user, commanding the model and/or the view to change as appropriate. Finally, the model manages the behavior and data of the application domain, responds to requests for information about its state (usually from the view), and responds to instructions to change the state (usually from the controller).

Inference is a particular property from KB. Is the action of extrapolating new information from current knowledge, and is a useful characteristic for validating concepts and properties, since the the users can begin stating simple characteristics, which can be combined to infer complex knowledge, extending the capabilities of the system which makes use of the KB.

The modeler was coded in the Java language, given its multi-platform capabilities and the need for using the Protégé OWL API, written in the same language. The Standard Development Kit selected was the last one available, Java SE 6, and the coding was conducted

under the Integrated Development Environment (IDE) Eclipse Ganymede.

The ontology was defined on the Protégé Framework. The version chosen was 3.2, since later releases have compatibility issues with ontologies created by previous versions.

2.2 Declarative modeling

Declarative modeling focuses on what users want, instead of the method used to obtain the result, or how to reach that solution. DM can be applied to a variety of problems, and has been used in several fields such as work flow systems [4] or biosystems [5]. It can be characterized as a multidisciplinary method, which involves several research fields, such as virtual reality, knowledge management or artificial intelligence (figure 2.1).

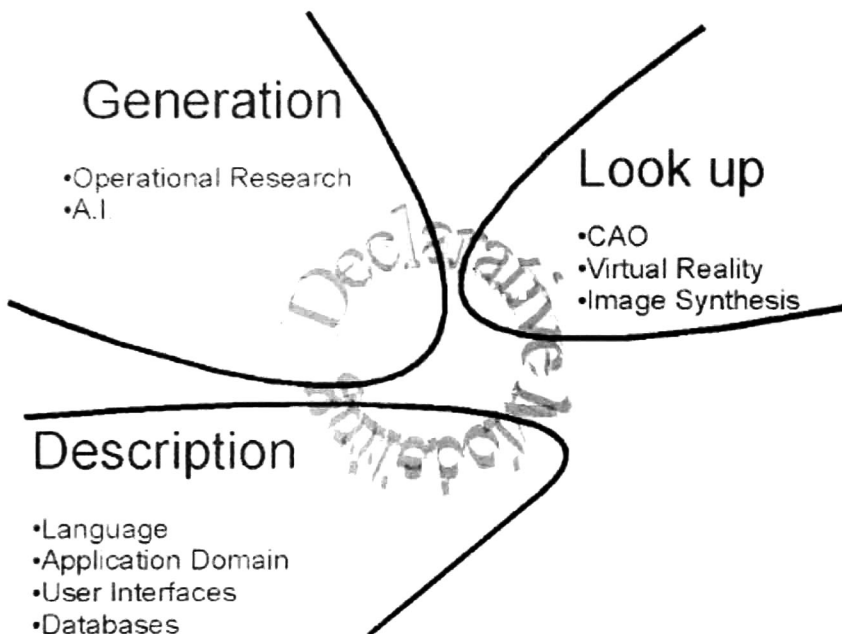


Figure 2.1: Fields of research.

Since our field of interest is the generation of VS using this method, we use the DM definition from Demetri Plemenos et al[1]:

Definition 2.1 (Declarative Modeling). A very powerful technique, allowing to describe the scenario to be designed in an intuitive manner, by only giving some expected properties of the scene, and letting the modeler find solutions, if any, verifying these properties.

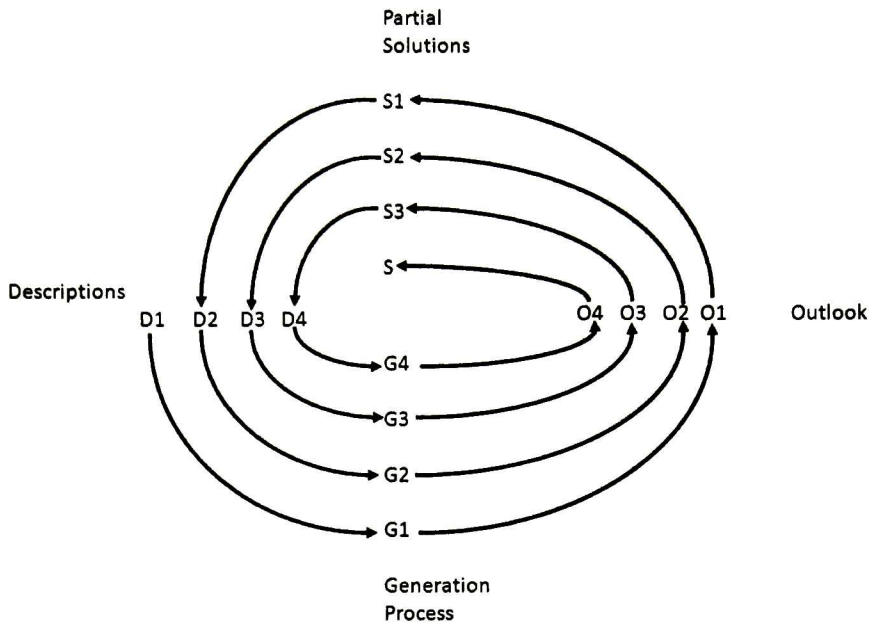


Figure 2.2: Interactive process of declarative modeling.

Following this definition, we aim to provide a system allowing users to create a VE data model, which can then be used by an underlying architecture to execute a simulation of a scene. We can divide the DM process into three steps:

2.2.1 Description

During this step, the properties and relationships between entities are provided. There are several interaction modalities, such as scripting, gestural or language interaction, as well as multimodal options. We take the language interaction approach, since it is a direct method, less intimidating than gestural methods (either mouse or haptic inputs), and, if handled properly, can be close to natural language, allowing an easy interaction between the modeler and the user. An important part of the description step is the semantic knowledge management, with the objective of avoid stating all the concepts explicitly by the user. Any ambiguity must be solved, using the specificities given in the description to fill the gaps in the model (such as placing books on a bookshelf, or orienting the audience in a theater toward the scenario) and obtain the context for the VE. This step defines the interaction language and UI, so the approach must be carefully selected and specified. We present our interaction language and UI in chapter 3.

2.2.2 Generation

Generation is the search of a consistent solution, through the analysis and evaluation of the properties stated. The properties are interpreted and solved according to a set of constraints and are used for obtaining all possible values for the variables in the problem, and exploring the solution space in order to find solutions matching with the user's requests. These solutions can be found by defining and solving a CSP. Several methods have been used to solve CSP, such as search trees [6] or specific procedural approach. We focus on constraint satisfaction, where methods such Space-CSP [7], Numeric-CSP [8] or Metaheuristics ([9], [10]) have been used. The efficiency of these methods depends on adequacy of the solving method, the representation of constraints, the complexity of the search space, and the application domain. To choose a method we consider four criteria: memory space, accuracy of the representation, efficiency, and simplicity of implementation. We also consider that complete methods are best suited to scenarios with few objects, while metaheuristic methods perform best with numerous elements. Metaheuristics are not able to certify the optimality of the solutions they find, while complete procedures have often proved incapable of finding solutions whose quality is close to that obtained by the leading meta-heuristics particularly for real world problems, which often attain notably high levels of complexity [9].

Objects must be characterized in order to be represented by the constraint set (figure 2.3). In this research, the representation of the objects must include its position, orientation and size, as well as relative and spatial relationship with other objects. The search space model can be represented in several ways:

- Explicit, storing all the possible values. However, the increasing amount of necessary memory is evident.
- Semi-explicit, associating an explicit representation with every variable, but using a projection method.
- Implicit, where the description contains the representation. Memory space is constant, however, it requires a test of consistency.

The constraints can be determined by properties or contextual data, implying that the properties must correspond to constraints. Constraints can also be represented in three levels, depending on the complexity, number of variables, and generation approach: Implicit (constraint is too complex), projective (using projection methods), and explicit (if constraint is simple enough). The model is created using iterative methods, refining the solution and satisfying the constraints at each interaction. First, the values for properties of the entities are solved, then, these values are validated against the rest of the model. If any constraint is violated or not satisfied, the model must be modified. Thus, the process repeats until finding a solution or a stop condition is reached.

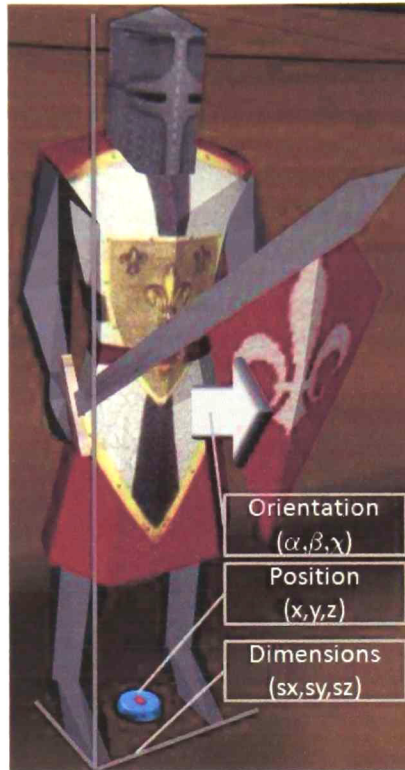


Figure 2.3: Characterization for an object

Properties are used for three main tasks: defining the layout of the objects, partitioning the space and build complex objects. We can also classify them in five sets:

- **Basic**, when they are used to set the exact object characteristics,
- **Fuzzy**, which allow partial, imprecise and negative descriptions,
- **General**, mainly concerning morphological features, positions, numbering, and appearance,
- **Specific**, assigned to predefined shape models, and
- **Spatial**, used to define the relative or absolute position of objects in the scenario.

2.2.3 Look Up

This allows the users to view all or part of the produced results. It can work using two methods: presenting to the users the solutions, or just presenting the most balanced solution. In both cases the users can decide to modify the solution and adapt it, so it comes closer to the mental image for the VE. There are several methods for the look up, such as freeze or comment [6], classification of solutions [2], presentation tools [11], navigation tools [12] or incremental refinement of the description [13].

It is important to consider two aspects in the design of a DM: first, the interpretation of the properties, where we should consider the translation of the properties into the constraints set. If this is not well defined, the result will be different from the users' requests. A correct interpretation must consider the *normality* of the context, this is, the normal usage for the object, its intrinsic and deictic orientation, and the notion of a *pivot* element, which functions as an orientation beacon to all the objects in the scenario.

The second aspect is the look up step, where the selection of the "good" solutions must be carefully directed. The modeler can generate all the solutions, but this would imply the exploration of the whole solution space. It can select a set of representative solutions. Or it can present only one, as long as all constraints are valid in the solution. Also, the modeler should allow the users to modify interactively the solutions, adding new properties as the objects are manipulated.

2.3 Knowledge Management

Knowledge Databases can be used in a variety of areas, from medicine [14] to genetics [15]], and for diverse tasks such as determine non-redundant information system architectures, support information management, enable shared understanding and communication between different entities, or facilitate the inter-operability of diverse systems [16] [17].

As we stated in the previous section, an important part during the description step in DM is the semantic management. In other words, how the translation between the properties stated in the description of the modeler data structure is handled. There are many ways to define an entity or a concept, from its physical appearance to the list of its attributes, elements or parts. For example, a virtual human being can be semantically represented as a complete unit, it can be described as the conjunction of several elements (head, torso, arms, legs), or can be defined as a set of philosophical statements (self-aware, conscious, intelligent).

Many methods have been proposed to represent knowledge, such as logic, semantic networks or rules [18]. Those methods are used in formal representation such as ontology, description logic or logic schemes, and in diverse applications such as expert systems or workflow applications [19]. To agree with the concept of knowledge management, we take

the description from Alavi et al [19]:

Definition 2.2 (Knowledge). Knowledge is information contained in the mind of individuals (which may or may not be new, unique, useful, or accurate) related to facts, procedures, concepts, interpretations, ideas, observations, and judgments.

From these concepts, we define the knowledge needed to define an entity as a collection of data, organized in a way that can be related with each other, and that has been formatted in a way that can be modified, corrected and eliminated as new information appears. This is in an analogous way to a Data Base, where the information is stored with a specific method, formatted within a layout, and presented according to the users' request. Since knowledge can be casted by many representations, we represent knowledge as a KB, parting from the analogy between knowledge for an entity and a Data Base.

A specific type of KB is called *Ontology*. From a philosophical point of view, ontology is the explicit specification of a conceptualization: a simple and abstract view of the world intended to be represented, with the objective to achieve some goal. For systems based on knowledge, what “*exists*” is exactly what can be represented.

Definition 2.3 (Speech Universe). When the knowledge of a domain is represented in a declarative formalism, the set of objects that can be represented is called Speech Universe. The set of objects, and the relationships that can be described between them, is reflected in the representation vocabulary used by a knowledge-based program to represent that knowledge [20].

A basic ontology of the real world is the relation existing amid all existing things, classified according to the way in which they exist. That is, the way in which something has reached the reality and the way it actually exists [21].

The systems and services based on knowledge are expensive to build, test and maintain. A software methodology based on formal specification of shared resources, re-usable components and complementary services is required. The specifications of shared vocabularies have an important place in the role of such methodology, because different applications require different reasoning services, as well as special purpose language to support them.

Therefore, there are several challenges to overcome for the development of a knowledge-based, shared and reusable software. As conventional applications, knowledge-based systems are based on heterogeneous hardware platforms, programming languages and network protocols. However, knowledge based systems have certain special requirements for interoperability. Such systems operate and communicate using sentences in a formal language. They make requests and send answers, taking “*previous knowledge*” as input. As agents in an AI distributed system, they negotiate and interchange knowledge. Communication at the knowledge level needs conventions at the three levels: representation language format, agents communication protocol, and content specification of the shared knowledge [20].

Since our project deals with the creation of a scenario using only a description written in a natural-like language, it is necessary to design the way to achieve a correct analysis of the sentences, as well as to keep coherence in the scenario. An example of this could be the sentence “The whale is in the living room” If we are not talking about a toy, logically, a whale cannot live a living room; it is too big and is a sea mammal. This can be derived from the ontology and the sentence can be marked as invalid. If the sentence is changed to “The toy whale is in the living room” the analysis must result correct, since a toy shaped like a whale can be in a living room. As before, this conclusion can be derived with the help of the ontology.

The Ontology will help us to conduct the semantic analysis of the language, since this analysis can be reasoned using the knowledge extracted from the ontology. In the previous example, for the word “whale” the ontology will return information that states that a whale is a sea animal, a mammal, and that it is several meters long and is heavy. From there, it can be reasoned that it is not correct for a whale to be placed in a living room. In the second example, the properties for a toy allow it to exist in a living room, and the in the shape of a toy whale. Thus, we will use the ontology to exploit knowledge and assure the semantic coherence of the sentences in the description used for generating the VE.

Many applications and standards have been developed to create, modify and access knowledge in knowledge based-form. We present some of these works, completely oriented to ontology building, next:

2.3.1 Ontolingua

The system developed at the KSL of the Stanford University [22] consists of a server and a representation language. The server provides an ontology repository, allowing the creation of ontology and its modification. The ontologies in the repository can be joined or included in a new ontology. To interact with the server the user can use any standard web browser. The server was designed for allowing the cooperation in ontology creation, easy generation of new ontologies by including (parts of) existing ontologies from a repository, and the possibility of including primitives from an ontology frame. The ontologies stored in the server can be converted to different formats to be used in other applications. This allows the use of the Ontolingua server for creating a new ontology, export it, and then use it in CLIPS-based application. It is also possible to import definitions from an ontology created on different languages to the Ontolingua language. The Ontolingua server can be accessed by other programs if they can use the ontologies stored in the Ontolingua representation language [20].

2.3.2 WebOnto

WebOnto [23] is a platform completely accessible from the internet. It was developed by the Knowledge Media Institute at the Open University and designed to support creation, navigation and collaborative edition of ontologies. In particular, WebOnto was designed to provide an interface allowing direct manipulation and that presents ontological expression using a powerful medium. WebOnto was designed to complement the ontology discussion tool Tadzebao. Thus, it is mainly a graphic tool oriented to construct ontologies. The language used to model the ontologies in WebOnto is OCLM (Operational Conceptual Modeling Language), originally developed in the context of the VITAL project to provide modeling operational capabilities to the work system VITAL [24]. This tool proposes a number of useful characteristics, like saving structure diagrams, relationships view, classes, rules, and so on, all of them individually. Other characteristics include cooperatively working in ontologies by drawing, and using broadcast and function reception.

2.3.3 Protégé

Protégé is a multi-platform package [25], designed to build domain model ontologies, and has been developed by the Informatics Medic Section of Stanford. It is oriented towards assisting software developers in the creation and support of explicit domain models, and in incorporating those models directly in software code. The Protégé methodology allows the system designer to develop software from modular components, including reusable work frames helping to build domain models and independent problem solving methods that implement procedural strategies to solve tasks [26]. The Protégé framework includes three main sections: a. the *Ontology Editor*, used to develop the domain ontology by expanding a hierarchical structure and including classes, and concrete or abstract slots; b. Based on the constructed ontology, Protégé is capable of generating a *Knowledge Acquisition tool* to input ontology instances. The KA tool can be adapted to the users' needs by using the "Layout" editor; c. The last part of the program is the *Layout interpreter*, which reads the output of the layout editor and shows the user an input screen with a few buttons. These buttons can be used to make the instances for the classes and sub-classes. The whole tool is graphical, which is friendly for non-experienced user.

2.3.4 Web Ontology Language

The Web Ontology Language is a semantic markup language designed for publishing and sharing ontologies over the World Wide Web. It is a standard [27] that forms part of the W3C Recommendations for the Semantic Web, and was developed by Deborah L. McGuinness and Frank van Harmelen, as a vocabulary extension of RDF (the Resource Description

Framework) [28]. It is a language oriented to process information context for applications that need more than just representing information for the users. Provides greater interpretability for web content than similar standards (XML, RDF or RDF-S), due to additional vocabulary and formal semantics. It can be divided in three sub-languages, each more expressive than the previous: OWL Lite, OWL DL, and OWL Full.

- OWL Lite supports primary needs for classification hierarchy and simple constraints.
- OWL DL provides maximum expressiveness, while retaining computational completeness.
- OWL Full presents the syntactic freedom of RDF and the maximum expressiveness, but does not provide computational guarantees.

OWL Lite uses only some of the OWL language features has more limitations than the others sub-languages. It allows restrictions on the use of properties by instances of a class, a limited form of cardinality restrictions, a limited intersection constructor, and the RDF mechanism for data values.

OWL DL and OWL Full use the same vocabulary, but OWL DL presents some restrictions. OWL DL requires type separation, i.e., a class can not be an individual or a property and can not be applied to the language elements of OWL itself.

An ontology written with OWL consists of three sets: classes, properties, and individuals. Each element in those classes can hold a superclass-subclass relationship. Classes define the archetype for the concepts in the ontology, and contain a set of restrictions, which are the constrains for the individuals, which are the instantiations of the classes.

The properties are related to the classes, and can be classified in two types: datatype or object, and both need a domain, the classes containing the property, and a range, that is, the subset of values allowed for that property. Datatype properties are basic level values, such as strings, integer or boolean values. Object properties indicate the relationship between classes, and contain the list of related individuals.

Finally, individuals are instantiations of the classes, and contain the actual values for the specific element in the ontological domain.

In figure 2.4 we present an example of an ontology created with OWL, in a graphical form.

2.4 Constraint Satisfaction Problems

During the generation phase, verifying the location of the different objects in the scenario is an important part of the model creation. The properties provided by the user for the

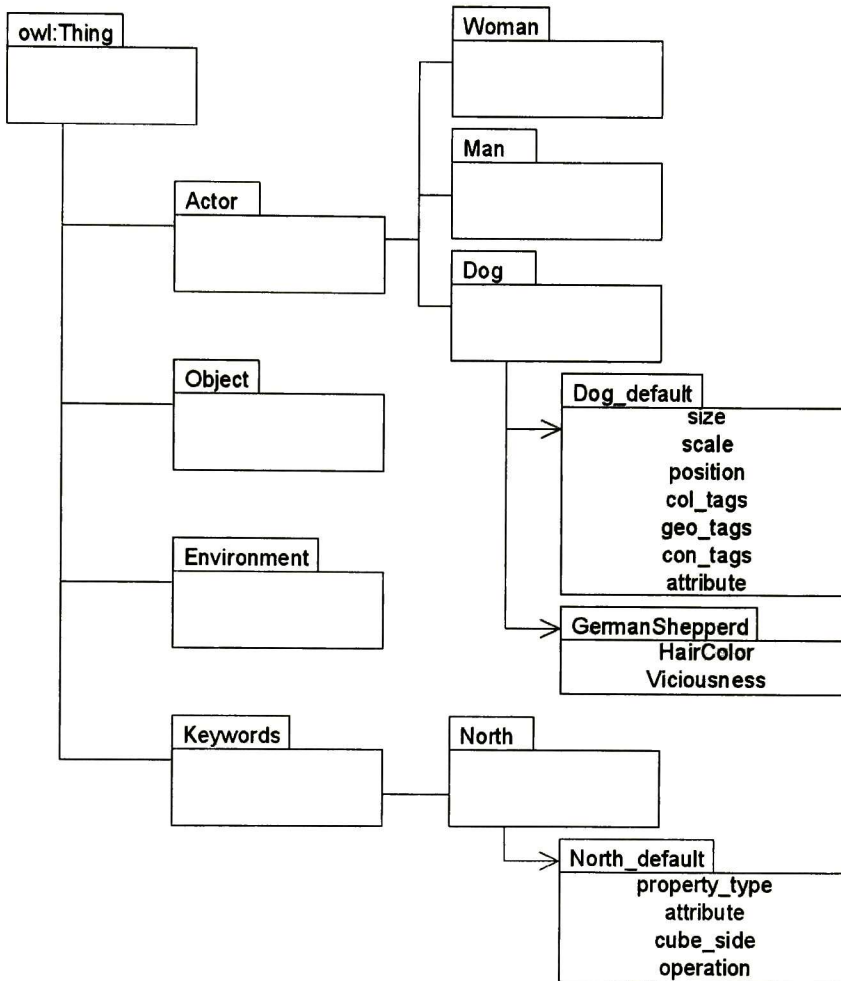


Figure 2.4: Ontology example.

VE are transformed into a set of constraints, which must be verified and transformed to assure the correct configuration of any solution. To solve any conflict between constraints, we employ an algorithm capable of finding a configuration in the solution space for finding a solution satisfying all constraints. An algorithms with those properties is known as Constraint Satisfaction Problem (CSP), and is defined as follows [29]:

Definition 2.4 (CSP Definition). A tuple $\langle X, D, C \rangle$, where:

$X = \{X_0 \dots X_n\}$, is the set of variables for the problem.

D is a domain for each variable X_i .

$C = \{C_0 \dots C_n\}$, the constraints set, where each C_i specifies a subset of X_i and the acceptable values for that subset.

A variable X_i is considered *instantiated* when it has been assigned a value from its domain D_i . Those variables which have not been assigned with any value are called *non instantiated*. We also can refer to both variables as *past* variables and *future* variables, respectively. The notation " $X_i = x_j$ " means that the variable X_i is assigned with the value x_j . The act of the instantiation is denoted by " $X_i \leftarrow x_j$ ". The variables in a CPS are instantiated in a given order, denoted by " \vec{X}_i ".

A variable is in a *dead-end* state, if there is no value in its domain consistent with \vec{X}_{i-1} . There are two kinds of dead-ends: *leave*, if there are constraints that forbid each value in its domain, and *interior*, if there are some values making the domain compatible with \vec{X}_{i-1} , but the subtree with root in the variable does not have a solution.

A problem state is the *assignment* of values to one or all of the variables, from the domain value sets of each variable, $\{X_i = v_i, X_j = v_j \dots\}$. If an assignment does not violate any restriction, is called *consistent* or legal. A solution to a CSP is a value assignment to all variables in such a way that none of the constraints is violated. A problem which presents a solution is considered satisfiable or consistent, otherwise is called *unsatisfiable* or *inconsistent*.

To solve a CSP, there can be used two approximations: search and deduction. Both are based on the idea "*divide and conquer*" that is, transforming a complex problem into a simpler one. Search generally consists of selecting an action to develop, maybe with the aid of a heuristic, which will take us to the closest state for the intended objective. Tracking is an example of searching for CSP. The operation is applied to the value assignation of one or more variables. If a variable can no longer be assigned in such way that can keep the consistency for the solution, it reaches a dead-end, and tracking is executed.

It is a good idea to visualize a CSP as a *constraint graph*, the nodes corresponding to the problem variables and the arcs to restrictions. Dealing with a CSP allows to generalize the successor function and goal test for adapting to any CSP, as well as to apply efficient and generic heuristics to avoid including additional information or domain expertise. Also, the graph structure can be used to simplify the solving process.

A CSP can be designed following an incremental formulation, as in standard search problems, starting with an empty assignment, this is, no variables assigned yet. A successor function will assign values to variables as long as there are conflicts with previously assigned variables. A test function is designed to find a complete variable assignment, and a constant step cost.

The simplest case of CSP implies discrete variables and finite domains, for example, the Boolean CSP, formed by variables that can be either true or false. For a maximum domain size of d in any CSP, the possible number of complete assignments is $O(d^n)$, where n is the number of variables, in the worst case. If the domain for the discrete variables being handled by the CSP is infinite, it is not possible to describe restrictions by enumeration of the possible values combinations, but a restriction language can represent it. In the case of continuous domains, the most common in the real world, we find that the most studied cases are linear programming problems, which are solved in polynomial time.

The constraints can be unary, when the constrained values affect only one variable; binary, when the constraint involves two variables; or higher order, implying three or more variables. Constraints can also be absolute, meaning that the violation of any of them excludes a potential solution. Some CSP include preference constraints, which indicate what kind of solution is preferred. Several methods have been proposed to solve CSP; next, we present some of them:

2.4.1 Backtracking

The simplest algorithm for solving CSP is backtracking [30]. This algorithm starts with an empty set of consistently assigned variables, and tries to extend it by adding new variables and values for them. If the inclusion of a new variable does not violate any constraint, the process is repeated until all variables are included. If the newly added variable makes the solution inconsistent, the last variable added is instantiated with a new value. If there are no more possible values for that variable, it is removed from the set and the algorithm starts the backtracking again.

An important element of the backtracking algorithm is the review of consistencies; which is conducted frequently, and makes an important part of the algorithm's work. The time used to conduct this revision is in agreement to the representation of the constraints. Those representations can be a list of the allowed tuples to maintain the constraints free and keep only the incompatible tuples, making use of a Boolean values table, or executing a procedure.

2.4.2 Backmarking

This method reduces the cost of consistency checking while backtracking is conducted [29]. It requires that the consistency review be executed in the same order, as the variables were instantiated. Proceeding in this way, the algorithm avoids previously tested and later rejected combinations, increasing the efficiency. However, this approach is restricted to binary CSPs and static variable ordering.

This method requires two additional tables, $M_{i,v}$ used to register the first variable with

a failed consistency check $X_i = x_v$. If $X_i = x_v$ is consistent with the previous variables, $M_{i,v} = i$. L_i records the first variable that has changed its value since $M_{i,v}$ was assigned for X_i with any value v from its domain. If $M_{i,v} < L_i$, the variable pointed by $M_{i,v}$ has not changed and $X_i = x_v$ will fail when being checked against $X_{M_{i,v}}$, therefore the consistency check is not required and x_v can be rejected. If $M_{i,v} > L_i$, $X_i = x_v$ is consistent with all the variables before X_{L_i} and those consistency checks can be avoided.

2.4.3 Backjumping

This is proposed as a way for reducing the amount of *trashing*, which is the act of finding the same dead-end several times [31]. This algorithm is capable of “jumping” from a dead-end to a previous variable that causes the dead-end with its current instantiation. A variable causes a dead-end, if in conjunction with zero or more variables preceding it in the ordering are instantiated in such way that a restriction will not allow the assignment of values to the variable in conflict.

An array J_i , $i \leq 1 \leq n$ is used to find variables that cause dead-ends. J_i stores the last variable test for consistency with some value from X_i . If X_i does not get into a dead-end, then $J_i = i - 1$. If X_i is inconsistent, then each value in D_i was tested for consistency with the past variables until an instantiation failed to pass test, J_i contains the index of the variable inconsistent for some value in D_i . Is important that the consistency review of instantiated variables be in the same order as the instantiation. If X_i is in a dead-end, we can guarantee that conducting the tracking between X_{J_i+1} and X_{i-1} will be unprofitable, since the cause of the dead-end in X_i is not marked. The partial instantiation \vec{X}_{J_i} will cause that any value for X_i generates a dead-end on some restriction, so modifying the values after X_{J_i} will not solve the dead-end.

2.4.4 Backjumping based on graphics

This method is another variation of backtracking. It jumps over variables as a response to a dead-end [32]. Unlike backjumping, which only responds to leave dead-ends. Backjumping based on graphics can respond to previous dead-ends. In order to accomplish this, it reviews the *set of parents* P_i for the dead-end variable X_i , where a parent of X_i is any variable connected to X_i through the constraint graph and precedes X_i in the instantiation order.

If X_i is a dead-end variable, the algorithm jumps to the last variable in the set of parents. If X_i is in a previous dead-end, the new set is formed by the union of the parent set of X_i and those from the dead-end variable found after X_i in the search tree. The algorithm then jumps to the last variable of the inducted set J_i . The algorithm requires to update J_i after every unsuccessful inconsistency review, requiring up to $O(n^2)$ space and $O(ec)$ time, where

c is the number of constraints and e is the maximum number of variables for each constraint. The other disadvantage of using the parents is using less refined data of the causes of the dead-end.

2.4.5 Forward Checking

Forward Checking is a variation of backtracking, which acts by instantiating a variable and removing any conflicting value in the domains of future variables. This algorithm rejects any value that can lead to the removal of the last value in the domain of future variables. The values are not removed permanently, but stored in the set D' , which contains the narrowed domains. The action of removing values from D' is called filtering [33].

The algorithm works filling D' with all the compatible values from each domain for each variable, and continues looking for at least one compatible value for D'_{cur} with future variables. It is not necessary to review previous variables, since D'_{cur} only contains compatible values with \vec{X}_{cur-1} .

Most methods make decisions on how the variables have been instantiated, and then modify values to continue searching for solutions. We know part of the solution space at the beginning of the search, having access to information of possible positions, relations that can be or not be changed, as well as ranges of correct values for the variables. Directed methods such as backjumping based on graphics or forward checking suit our needs better, since the inclusion of metaheuristics based on knowledge can lead to quicker solution finding, dead-end solving, and corrections in the search direction.

2.5 Virtual Environment

One of the best definitions for VR we have found is “Virtual Reality is a way for humans to visualize, manipulate and interact with computers with extremely complex data” [34]. Visualization means that the user can perceive the outputs generated by the computer. This means, the actions achieved in the world represented inside the computer. The perception can be visual, auditory, sensory or a combination of these. The world being represented can be a CAD model, a scientific simulation, or the view of a database. Interaction means that such world can evolve autonomously, be means of either the objects inside the world or the properties of the world itself. This interaction triggers the evolution (animation), through some process, of either physic simulations or simple animation scripts [35].

In a VW we found entities that dwell inside it, which are named commonly *avatars*. Avatars represent animated entities having complex behavior, for instance animals (persons or other type), or other imaginary animated creatures. The avatars can perform a variety of

actions, according to the world they have been put in, as well as represent human emotions ([36],[37]) and perform varied behaviors [38]. Those avatars have different levels of complexity and detail, depending on the overall representation of the world-taking place. We can take as an example a person: in simple simulations, it is not needed great details of the body, for instance the skin, the hair, the face details are represented by very simple models (Figure 2.5). In contrast for movies, a virtual character must have a high level of detail (a detailed face, modeling individual hairs). The level of detail comes at computational cost. Currently several methods are used to provide better levels of representation in real time, such as ray tracing or bump-mapping [39].

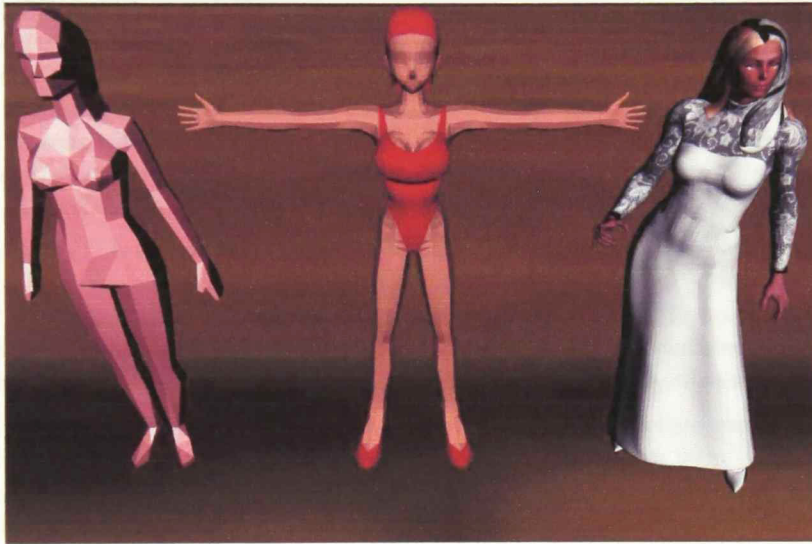


Figure 2.5: Different levels of detail for avatars.

Among some of the applications for VE we can find: the creation of VW for video games, where the player can travel through different environmental settings, interact with diverse characters and perform a set of actions. In movies, VW are used to re-create ancient worlds, fantasy settings or impossible situations. VE are also used in applications such as architecture [40] or city-planning [41][42], as well as for applications such as story telling [43], as a support method in surgery and medical education [44] or as educative tools.

2.6 Related Works

Several works have exploited the methodology of DM. Before commencing the review of some of them, it is necessary to state that since we are focusing on using DM to create VEs, all the works presented in this document are also oriented towards the creation of virtual 3D models. In these works one of the problems solved is validating the disposition of the objects that conform the VW using diverse techniques, whereas other modelers focus on validating other aspects of the model, such as congruency or efficiency (Repast Symphony System [45], Joseph System [46]). Of the two tasks, the most demanding in computing processing is validating the correct positioning and orientation.

Some of the reviewed works focus on architectural or urban design, such as FL-System [40]. This system is focused on the generation of complex city models, parting from a specialized grammar, and using a variant of the Lindenmayer System [47], called Functional L-System, replacing the generation of terminal symbols by generic objects. It uses VRML97 to visualize the models, and works generating individual buildings and then incrementally working the rest of the city block, and later the entire city.

CityEngine [41] is a system capable of generating a complete city model, using small sets of statistical and geographical data, contained in geographical maps (elevation, land, water maps) and socio-statistical maps (population maps, zoning maps). It works creating a first layer of roads, using L-Systems, and then creating city blocks. The final step is the generation of geometry and visualization, using first a real time render, and then a raytracer.

Wonka et al. in [48] presents a method for automatic architecture modeling, using a spatial attribute design grammar, or split grammar as input for the user. The input is used to create a 3D layout which is the base for the building in creation. The facade is created next, splitting it into structural elements at the level of individual parts (windows, cornices, etc). The resulting model is shown to users through a real-time render.

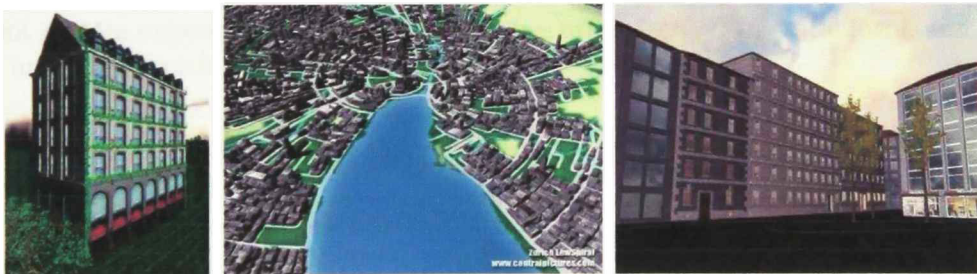


Figure 2.6: FL-System, City Engine and Instant Architecture

None of the previous presented works is oriented toward creating a model of a VE, but

just a 3D model of a description, composed in all the cases by a specific grammar, either raw geographical data, architectural designs or three-dimensional design data. These projects do not allow any kind of interaction with the environment, due to completely automated model generation. In those works, once the VW is generated, it is not possible to interact it beyond positioning the camera. If a modification is needed, it is mandatory to modify the description in order to change the output of the system. Other works are oriented to create VS, where different elements are positioned in the VW, and their properties can be modified. In the next subsections we will review some of them.

2.6.1 WordsEye: Automatic text-to-scene conversion system

Bob Coyne and Richard Asproad at the AT&T laboratories developed WordsEye [49]. This system allows the generation of a 3D scenario from a description written on natural language, for instance: “the bird is in the birdcage. The birdcage is on the chair” The text is initially marked and analyzed using part-of-speech taggers and statistical analyzers. The output of this process is an analysis tree, which represents the structure of the sentence. Next, a depicter (low level graphic representation) is assigned to each semantic element. Those depickers are modified to match with the poses and actions described in the text, through inverse kinematics. After that, the implicit and conflicted constraints of the depickers are solved. Each depicter is then applied, while keeping its constraints, to incrementally build the scene. The final step includes adding the background environment, the terrain plane, the lights, and the camera. Then, the scene is rendered and presented to users. If the text includes some abstractions or descriptions that does not contain physical properties or relations, the system employs several techniques, like textualization, emblemization, characterization, lateralization, or personification. This system accomplishes the text-to-scene conversion by using statistical methods and constraints solvers, and also has a variety of techniques to represent certain expressions. However, the scenes are presented in static form, and the user has no interaction with the representation.

2.6.2 DEM²ONS: High Level Declarative Modeler for 3D Graphic Applications

DEM²ONS has been designed by Ghassan Kwaiter et al [50] offering to the users the possibility to easily construct 3D scenarios in natural way and with a high level of abstraction. Two parts constitute it: a modal interface and, the 3D scene modeler. The modal interface allows the users to communicate with the system. It uses simultaneously several combined methods provided by different input modules (data globes, speech recognition systems, spaceball, mouse). The syntactic analysis and Dedicated Interface modules analyze and control the low-level events to transform them in normalized events. DEM²ONS uses ORANOS as 3D scene modeler, a constraint solver designed with several characteristics allowing the

expansion of DM applications, like generality, breakup prevention and dynamic constraint solving. The GUI (Graphic User Interface) is based upon the Open Inventor Toolkit [51] and Motif library [52]. These two modules render the objects, and provide support to present the menus and tabs. DEM²ONS allows the user to interact with the objects in the scene. Also, it solves any constraint problem, but only allows static objects, with no avatar support.

2.6.3 Multiformes: Declarative Modeler as 3D sketch tool

William Ruchaud et al. presents Multiformes [53], a general purpose DM, specially designed for 3D scenario sketches. As any DM, the work over the scenario with MultiFormes is handled essentially through a description (the way the user inputs all of the scenario geometric characteristics of the elements, and the relationships between them). The most important feature in MultiFormes is the ability to automatically explore all the possible variations in a scenario. Unlike most of the existent sketch systems, Multiformes does not present only one interpretation of each imprecise property. Starting with a single description, the designer can obtain several variations of the same scenario as a result. This can lead the users to choose a variation not considered previously. A constraint solver supports this process. The description of the scenario includes two sets: *the geometric objects set* presents in the scenario, and *the set of existent relationships between the geometric sets*. To allow the progressive refinement of the scenario, MultiFormes uses hierarchical approximations for the scenario modeling. Following this approach, a scenario can be incrementally described at different levels of detail. Thanks to its constraint solver, MultiFormes is capable of exploring diverse variations of sketch, satisfying the same description.

The geometric restriction solver is the core of the system and is used to create a hierarchically decomposed scenario. Even when this system obtains its solutions in incremental ways, and is capable of solving the constraints requested by the user, the system requires the list of actions needed to construct the scenario. This requirement makes the use of the system restrictive.

2.6.4 CAPS: Constraint-based Automatic Placement System

Ken Xu, et al. presents CAPS [54], that is a positioning system based on restrictions. It makes possible modeling big and complex scenarios, using a set of intuitive positioning restrictions that allow manipulation of several objects simultaneously. It also employs semantic techniques for the positioning of the objects, using concepts such as fragility, usability or interaction between the objects. The system uses pseudo-physics to assure that the positioning is physically stable. CAPS uses input methods with high levels of freedom, such as Space Ball or Data Glove. The positioning of the object is executed one at the time. Allows direct interaction with the objects, keeping the relationships between them by means of pseudo-physics

or grouping. These methods and the tools integrated into this system make it a design tool, mainly oriented towards scenario visualization, with no capabilities for self-evolution.

2.6.5 ALICE

This is a tool for describing the time-based and interactive behavior of 3D objects, developed at the Carnegie Mellon University [55]. Described by its authors as “a 3D interactive, animation, programming environment for building VW, designed for novices” provides a creation environment where users can create, use, and animate 3D objects to generate animations. The users select an object from a 3D database and then arranges its position in the world. After the object has been positioned, the user can select primitive methods, which send messages to the object. These methods are arranged to form program sentences which are interpreted by a Python Interpreter, which works as a scripting service. The system uses Microsoft 3D Retained Mode (D3DRM) to render the scene. Users can add new content and methods, since the system supports many 3D modelling formats. This project focuses on educational/instructional areas, but still focusing on programming paradigms. The users need to actively create the scenario, and specify the scene using a programming-like tool. Also, the software does not make any context verification.

Chapter 3

Proposal

Abstract

This section is devoted to explain the approach taken to solve problems posed by the inclusion of knowledge in DM. Also, a detailed explanation of the procedure and the methodology is presented.

3.1 Interaction Language: A Review of VEDEL

To allow interaction between the modeler and the user, is necessary an interface that allows easy specification of the desired properties for the model intended. There are different input methods, but we choose to let the user express himself in a natural-like language the characteristics of the VE.

Our objective in defining a declarative method to describe a vs was to provide users a structured, easy to follow method to compose the description of a scenario, in the form of a declarative scripting language, which we called Virtual Environment Description Language, or VEDEL [56] that is a tag language. A description in VEDEL is composed by three sections, each of them devoted to describe one of the three possible types of elements in the VE: the **Environment**, that is, the general settings for the scenario; **Actors or avatars**, which are those entities capable of perform and react to actions, display emotions, represent behaviors, and react to the changes in the environment and other entities; and **Objects**, entities that can be subjected to actions, but cannot act on their own. Each section is delimited by *section tags*, which start with a keyword for that section (*ENV*, *ACT*, or *OBJ*) respectively enclosed in brackets ([]). A slash (/) before the keyword makes the tag close a section.

The sections are composed by sentences, each of them formed by comma-separated statements, and ended by a dot (“.”). The first statement must be a *concept word*, this is, a word that explicitly defines the idea that will be represented, followed by a single-word proper name for that entity, which must be unique for every entity (the environment does not need a proper name), which can be of any length and can use special symbols. The rest of the sentence is composed by the entity properties, which must begin with the property name, followed by the values for that specific property. Numeric values must be enclosed in parenthesis, and it is possible to define specific ranges by enclosing the values in brackets ({ }), and separated by commas. This last option is a technical feature of the lexical-syntactic parser, and its use is not intended for final users.

The basic structure of a description composed in VEDEL follows the pattern:

```

``[ENV]''
  Environment keyword, [ environment settings ], ``.''
``[/ENV]''
``[ACTOR]''
  { Actor Class, [ Actor Identifier ], [ Actor Properties ], ``.'' }
``[/ACTOR]''
``[OBJECT]''
  { Object Class, [ Object Identifier ], [ Object Properties ], ``.'' }
``[/OBJECT]''

```

The revised formal Description Structure is defined as follows:

```

Description ::= environment, actors, objects;
environment ::= "[ENV]" environment sentence "[/ENV]";
actors ::= "[ACT]" [ { actor sentence } ] "[/ACT]";
objects ::= "[OBJ]" [ { object sentence } ] "[/OBJ]";
environment sentence ::= environment class, [ environment properties ], dot;
actor sentence ::= entity class, [ actor properties ], dot;
object sentence ::= entity class, [ object properties ], dot;
properties ::= { separator, properties };
environment class ::= entity class;
entity class ::= class, [ identifier ];
environment properties ::= { characteristic };
actor properties ::= { comma, characteristic | property | position };
object properties ::= { comma, characteristic | position };
characteristic ::= class, value;
position ::= class, [ [ number ], identifier ];
property ::= class, [ value ];
class ::= word;
value ::= word | number | range;
identifier ::= letter — digit, [ { letter | digit | special symbol } ];
word ::= { letter };
number ::= "(" [ minus ], { digit }, [ dot { digit } ] ")";
letter ::= "A" to "Z" | "a" to "z";
digit ::= "0" to "9";
special symbol ::= "@" | minus | "_";
minus ::= "-";
comma ::= ",";
dot ::= ".";
range ::= "f" number | word [ { comma, number | word } ] "g";

```

<pre> [ENV] House, night, rain. [/ENV] [ACTOR] Man Albert, old, sit FrontCouch, reading to- dayNewspaper. Woman Beth, old, sit RockingChair, knitting. [/ACTOR] [OBJECT] FrontCouch, left Fireplace. Chair RockingChair. Paper todayNewspaper. [/OBJECT] </pre>	<pre> [ENV] Farm, day, hot. [/ENV] [ACTOR] Man Albert, close BigTree. Woman Beth, sit PicknickTable, talking Car- rie. Carrie, sit PicknickTable, next Beth, talking Beth. [/ACTOR] [OBJECT] Table PicknickTable. [/OBJECT] </pre>
---	--

Figure 3.1: VEDEL examples.

A description can contain any number of sentences per section; the number of properties allowed per sentence is only restricted to the number of properties associated with the entry of the entity in the KB. The lexical-syntactic parsing method is presented in depth the next section.

3.2 Parsing Methodology

Descriptions given by the user are analyzed by a lexical-syntactic parser, which is a *state machine* that searches for invalid characters, and verifies if descriptions are composed following the rules established by the language definition. It also formats the entry into a data structure, which the model creator can use.

The first step is searching for *tokens*, individual words which are strings of characters delimited by a space, a comma, a dot, or a carriage return. Then, the *syntactic analyzer* sets the token to a data structure designed to allow easy exploration by the model creator (figure 3.2). This data structure is a hierarchical tree, with branches which start with entity class and the unique name of the entity, if it founded. The leaves are filled with the properties requested for that entity, starting whit the name of the property, and followed by the values requested for that particular attribute. When a dot is found, the newly parsed sentence is stored and the process continues in the next sentence. If the following token is a *section tag*, the parser verifies tag parity, then proceeds the analysis of the next section or reports the corresponding error state. The parser reports any error to the *Error Manager*, and discards the faulty token, statement, sentence, or section. When the closing objects sections tags is

reached, or when the end of the description string is found, the parser ends its task, and sends the parsed entry and the error report to the next module, where the modeling process continues, the necessary actions are taken in order to complete the process and, the user is informed with the errors found so far.

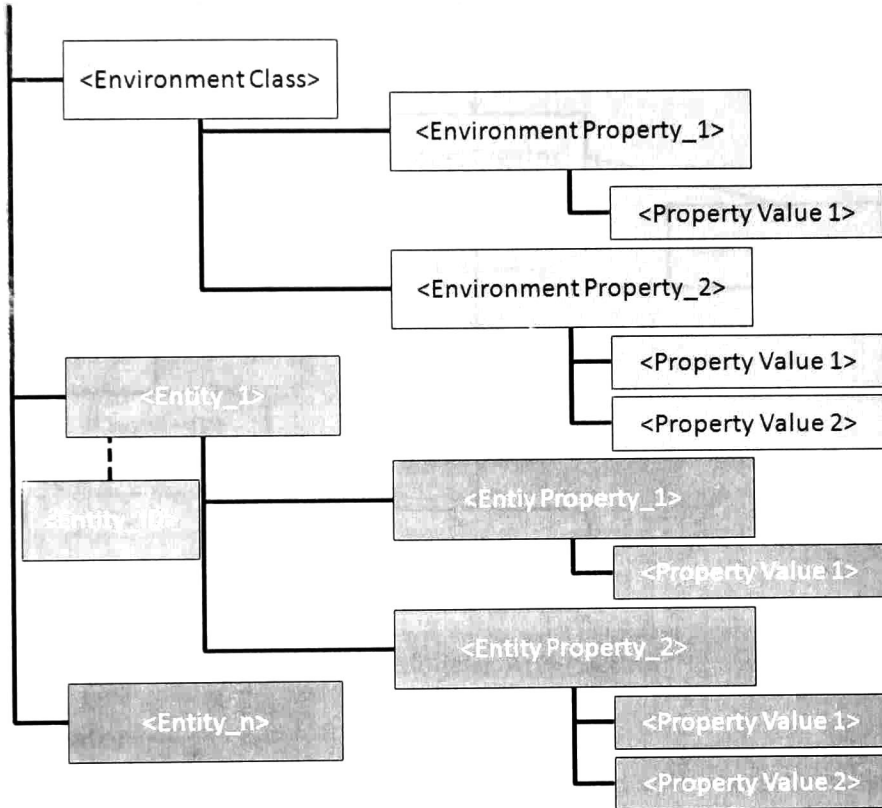


Figure 3.2: Parsed Entry Structure

The parser has been designed, so individual sentences or even individual statements can be sent for verification and conversion, allowing to modify the model previously constructed, by adding, deleting, or changing the attributes for new or existing entities.

3.3 Modeler's Architecture

The modeler is composed by five modules, as shown in figure 3.3. The Lexical-Syntactic Module was explained in the previous section (3.2), while the rest of the modules is presented

in depth in the following sections. This section highlights the functioning of the modeler and resumes briefly each module.

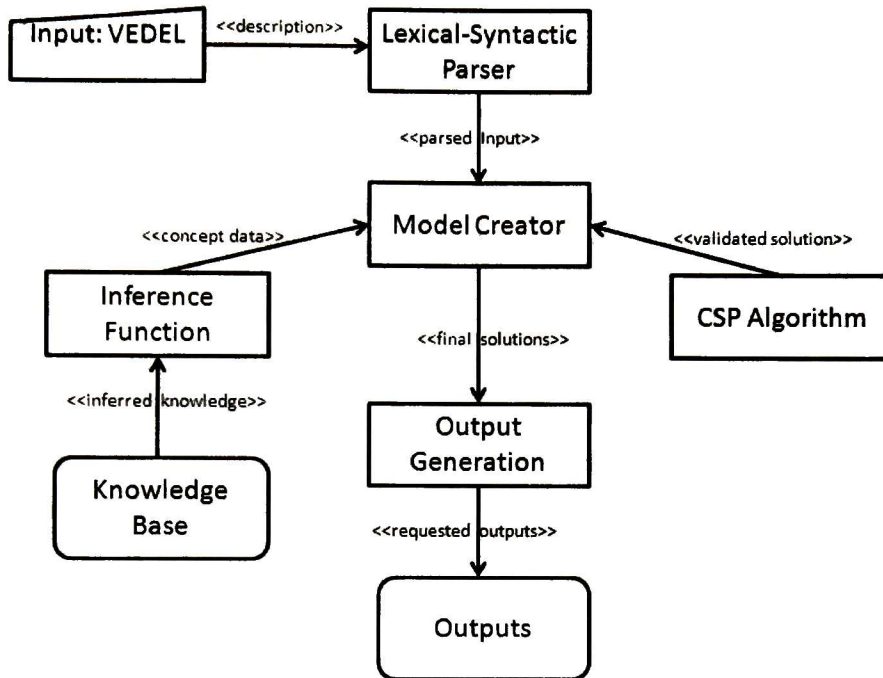


Figure 3.3: Modeler Architecture

Model Creator

The *Model Creator* receives the parsed entry to proceed with the generation of a model satisfying the constraints stated in the description. It makes use of the inference module, and formats the information obtained through it to generate the model. The modeler also creates a list of entities in the environment and their dependencies, so the positioning process can use this information in order to facilitate the task. It also does collision verification in order to solve conflicts involving the geometry of the entities intersecting each other, and positioning validation, to assure the position of the entity corresponds to description constraints.

Inference Module

The inference machine access the KB to gather the necessitated data contained within it. It uses the OWL API to obtain specific knowledge, and formats the data so the modeler can use the values during the model creation. It also validate the semantic values of the constraints, as well as the overall composition of the description.

CSP Algorithm

This module solves the conflicts that arise from the positioning of the elements in the scenario. The CSP Algorithm verifies the model for detecting: collisions, incorrect positioning, or invalid disposition of the elements, to correct any conflict found. It returns a valid model to the *Model Creator*, or the list of unsolved conflicts.

Output generator

The requested outputs are generated by a *Model-View Controller*, which receives the model created by the *Model Creator* to generate the outputs to be used by the underlying architecture. The templates are formatted accordingly to the needs of the architecture or the systems that the model will use.

3.4 Creating the Model

Once the description has been analyzed and the model creator received the output from the lexical-syntactic parser, the actual creation of the model begins. This creation starts with a *default-valued* initial model, which we called the *zero-state model*. This model is refined progressively to finally obtain a model which contains all the values requested by the users and also satisfies the constraints, explicit or implicitly, requested by the users.

3.4.1 Model Data Structure

The obtained model is basically a hierarchical structure. The top elements correspond to the class of the entities, which are obtained from the KB. The leaves on the same level correspond to the unique name, that is, the type of entity (environment, actor or object), and its properties. The structure representing the environment model is presented in figure 3.4.

The classes defining the root branches in the model are Environment, Avatar and Actor. The *Environment* contains the details for the setting of the scenario, as well as the laws that will rule the entities in it. The information contained in the highest level correspond simply to environment size and descriptor, all the rules and properties of the environment are stored in the leaves of the sub-tree with root on the entry Environment. *Avatar* corresponds to the basic information regarding the entity, or the “*default*” entity, which can be viewed as the base model for all the entities of the same type. Example: a “man” avatar contains information about the conformation of a human, i.e., arms, head, legs, as well as the properties for the entity, such as hair, skin color, stamina, mood, with all of them having specific values set in the KB. Also, this class can be used to control the number of avatars of the same type, so the modeler can set the individual names for those entities without a specific identifier. The

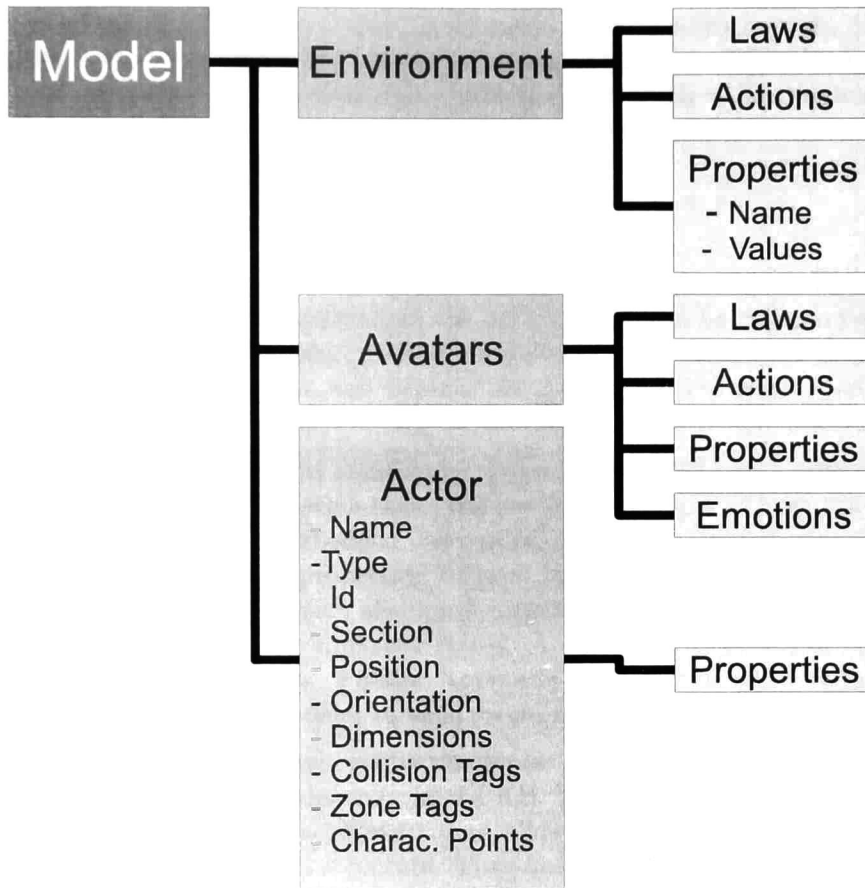


Figure 3.4: Model Structure.

elements of this class are used as a template; this means that all of the properties for new entities are copied from the values stored in this branch. Finally, the last branch contains the avatars instantiations, the *Actor* class instances, where details for each entity are stored. Those details include properties such as size, scale, position, initial action or validation tags. These values are stored as basic data types, since this branch is accessed by other modules of the architecture.

3.4.2 Modeler procedure

The modeler explores the parsed entry, queering the inference machine for every term found, with the exception of individual identifiers or numerical values. The information provided

by the inference machine is processed and stored in the corresponding sections of the model, keeping the linguistic terms for later references or validation procedures during the direct modification of the model. The modeler also conducts the semantic validation using the KB, dropping the values, terms or concepts found to be invalid or out of context.

Knowledge Exploitation

The *Knowledge Base* holds four basic classes: **Actors**, **Environment**, **Keywords** and **Objects**. Each of these classes contains all the entities that can be represented, either by the underlying architecture, or by any external software. For each class there must be at least one individual, named as the class, and followed by the “_default” suffix. Individuals are the instantiations of the classes, and contain all the data accessed by the inference module.

Each individual contains several mandatory properties such as size, position, entity descriptor, concept ranges, and validation tags. The environment must at least contain properties for size, and descriptor (an embedded description that will represent the environment). Actor and Objects must contain properties for size, initial position, collision tags, characteristic points, and attributes. Actors also must contain the action to be performed by the entity if not action is explicitly or implicitly stated. Actors without a default action are set to idle at the beginning of the scene. Finally, Keywords must include value ranges, property type, and operation type. The modeler to validate properties uses this information.

Knowledge bases can contain two properties types: Object and Datatype. *Object* properties express relations amid the elements in the KB. This type of property is used by the modeler to extract information about the values allowed for each property, as well as for formatting data for the model data structure. They are also used to determine the semantic validity of statements made in description. *Datatype* contains raw values that define concepts. These values can be: character strings, numerical, date, or Boolean values, which are stored at the end of the modeling process in the data structure. The Output Generator or the underlying architecture accesses this information.

The inference machine starts queering the KB for a particular concept. If the information exists, then it subtracts the information regarding the type of concept, and proceeds to explore the values stored for that particular entry. If the values are found to be object type, the inference process continues deeper, subtracting values for objects referenced, and then exploring the values stored for each of these, until a raw datatype value is found. To exemplify this process, consider the following request: “Chair, color white. The “color” property must be first a valid characteristic for the entity Chair, which is verified when the inference function subtracts the values for the avatar. Then the inference machine continues looking for possible values that can be assigned to the property of the entity. If any of the values correspond to the request made, the inference machine then proceeds to obtain the value

assigned to the concept, in this case, “*white*” The inference machine then subtracts the datatype value named RGB stored in the individual “*white*”, and passes the raw value stored as the character string “1 1 1” to the modeler, which uses it to create a leave for the entity in the model.

Depending on the concept being processed, the inference module also conducts conversions between datatypes. For example, the Keyword individual for “*left*” contains the property **range**, which expresses the values to be used to set the position of an entity. The value is stored as a character string in the KB, for instance, as “{ 30,0,0 }” This value cannot be used directly by the modeler, which requires a float-type array. Thus, the inference module conducts the conversion from string to array, and returns it to the modeler.

Since all the modeling process is based on the exploitation of the KB, the data stored can lead to unexpected or invalid values, from the users’ perspective expected results, so management of information should be left to a system administrator or an experienced user.

Initial Model

The modeler starts with an empty model, which is updated as the parsed entry is evaluated. The first element to be updated is the environment. The modeler request the inference machine for the *default* values for the environment where the scenario must be constructed and which are used to create a temporary leave. This leave is updated first with the instructions set for the environment, and later with the user’s requests. When the environment has been fully processed, it is assigned to the model.

The scenario can contain several *zones*, which have no visual representation, but are still used for referencing specific areas in the environment as referents for absolute positioning. These areas can be *landmarks*, specific delimitations inside the environment, *walls*, used to define the limits for landmarks, and *doors* which indicate the zones that allow the entities to pass from one zone to another. It is during the initial model generation where these zones are established, thus gives the rest of the entities the referencing points for proposing a positioning.

The procedure for the entities differs in one step. When the modeler finds an actor or object type entity, a search for the corresponding avatar is conducted. If the avatar is already stored in the model, the process to update the values for the properties is started, using a copy of the avatar as template for the current entity. If the avatar is not found, the modeler requests information for that specific entity type, and stores the corresponding avatar in the model, creates a template copy, updates it with the user’s request, and adds the new actor to the model.

The next step is to gather information to position the entities in the scenario, and the relations between them. The parsed entry is queried to subtract the entities, which are stored

in a FIFO list. When a new element is entered in the list, all the entities that have some relation with it are reviewed, and the entities that in turn hold a relation with the current are reviewed. This recursive process allows the modeler to set those entities which have dependencies on them, which we called *pivots*, first and later add the elements that make reference to these pivots.

Any error found during the creation of the model is reported to the *Error Manager*, which stores the corresponding error data and provides the corresponding course of action to solve the error state and continue. If the modeler is set to stop when finding an error, the process is halted and the error presented to the user. Otherwise, the process continues, and the list of errors is presented at the end of the modeling procedure.

Properties Values Assignment

As presented in section 3.4.2, an entry in the KB for any entity contains linguistic terms as well as raw data type terms, and the relations with other elements in the KB. The conversion works over the non-type terms to convert them to basic raw data values. This conversion is made so that the final output can be read by the other modules in the architecture, adapted to their own input languages, or to create output files readable by 3D render machines.

When the Inference Machine is processing and needs a property concept, the KB is queried for the term, and the raw information is in turn processed. If any of these raw data items is a linguistic term, it is processed, and the cycle continues until the low-level data types are obtained.

The values stated for the properties in the description are first validated against the restrictions set in the KB, then converted in the case of linguistic terms. Any inconsistency found during this process is reported to the *Error Manager*, which decides if the property can be removed or must be set to a default value.

Some of these data items, such as “*furniture*” for the environments, or previously generated VEs, can be VEDEL sentences or statements, which are processed by the Lexical-Semantic parser to include objects belonging to the environment or the entities. These items do not disrupt the generation process, since they are processed after the parent entity is completed. In this way, it is not necessary to extend the positioning list to include these new elements, since the parent acts as pivot element in turn.

The raw values obtained from linguistic terms are returned to the model creator, which assigns them to the entity in creation, and then continues with the following property. Once all properties have been validated, the modeler adds the entity to the model, and continues with the model generation task if it is needed.

3.4.3 Geometrical Validation

Once all data properties have been processed, the modeler continues with the process for verifying the positioning of all elements. This is handled by a CSP algorithm, which uses the collision and positioning tags to solve conflicts or invalid positions.

Each entity in the VE is represented as a set $X_i = \{P_i, O_i, S_i\}$, where $P_i = \{x_i, y_i, z_i\}$, $O_i = \{\alpha_i, \beta_i, \gamma_i\}$, $S_i = \{Sx_i, Sy_i, Sz_i\} \forall X_i \in X$, corresponding to position, orientation and scale for that entity.

Default position corresponds to the VE's center, this is, $\{0, 0, 0\}$. Each entity starts with a position corresponding to the VE center at the first modeling steps. If a specific position is requested, the model creator sends a query the to Inference Function, which returns a vector $V = \{x, y, z\}$, corresponding to the request. V can be either an absolute position, or a point in relation with another entity or environment component. In that case, V is calculated as $Rot(V - (X(P) * X(S)), X(O))$, where $Rot()$ is a rotation function on $X(O)$.

The CSP used by the modeler is defined as follows:

- The set of variables $X = \{X_1, X_2, \dots, X_n\}$ composed by the entities in the scenario, $X_i = \{X_i \cup \{Co_i, Ch_i\}\}$, where $Co_i = \{Sp_1, Sp_2, \dots, Sp_n\}$ representing a set of collision tags assigned to the entity, where $\forall Sp \in Co_i, Sp_i = \{Sp_x_i, Sp_y_i, Sp_z_i, Sp_r_i\}$, and $Ch_i = \{Pe_1, Pe_2, \dots, Pe_n\}$ corresponds to a set of characteristic points for the entity, where $\forall Pe \in Ch_i, Pe_i = \{Pe_x_i, Pe_y_i, Pe_z_i\}$.
- The domains for the variables are defined as follows $D(P_i) = [-\infty, \infty]$, $D(O_i) = [0, 2\pi]$, $D(S_i) = [0, \infty]$, $D(Co_i) = [-\infty, \infty]$ and $D(Ch_i) = [-\infty, \infty] \forall X_i \in X$
- The constrains set is formed by the following equations:

$$(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2 - (r_1 + r_2) \leq t_1 \quad (1.a)$$

$$\left(\frac{x}{p}\right)^2 + \left(\frac{y}{q}\right)^2 - 2z = t_2 \quad (1.b)$$

$$\left(\frac{x}{dx}\right)^2 + \left(\frac{y}{dy}\right)^2 + \left(\frac{z}{dz}\right)^2 - 1 = t_3 \quad (1.c)$$

Thresholds t_1, t_2 and $t_3, \{t_1, t_2, t_3\} \in \mathfrak{R}$, are values set by the system administrator. These values allow modifying the strictness for the constraints. Each collision tags set is stored in the KB as a vector, which represents the position and radius for the sphere.

The second set of characteristic points is based on the geometry of the entity, selected if they help to represent the contour of the entity, or correspond to a peculiar feature. These



Figure 3.5: Collision Tags

points are used along equations 1.b and 1.c to verify positioning. Both collision and characteristic points set are updated as the entity moves or rotates, using the *Rot()* function. The radius of collision tags is also modified when the entity is scaled up or down.

The constraints are satisfied if:

1. Be $(X_i, X_j) \in X$, constraint 1.a is satisfied $\leftrightarrow C_1(Sp_m, X_j(Sp_n)) \leq t_1, \forall Sp \in X(Co)_{i,j}$.
2. Be X_i an entity whose position was calculated as $P(X_i) = F(V, X_j)$, constraint 1.b is satisfied $\leftrightarrow C_2(Pe_n, X_j) \leq t_2, \forall Pe \in X(Ch)_i$.
3. Be X_i an entity with a position calculated as $P(X_i) = F(V, X_j)$ and $P(X_i) - P(X_j) < 1$, constraint 1.c is satisfied $\leftrightarrow C_3(Ph, X_j) \leq t_3$, where $Ph \subseteq Ch, \forall Pe \in X(Ch)_i$, and $0 < dn < 1, dn \in \{dx, dy, dz\}$.
4. Be X_i an entity with a position set as $P(X_i) = V$: constraint 1.c is satisfied $\leftrightarrow C_3(Pe_n, Env(S)) \leq t_3, \forall Pe \in X(Ch)_i$, where $Env(S)$ is the environment or area measures.

A verification is taken previously to search for collisions: if the euclidean distance between two entities is less or equal to the sum of the diagonal of a box formed by the dimensions of each entity, collision verification is carried out. This can be defined as follows:

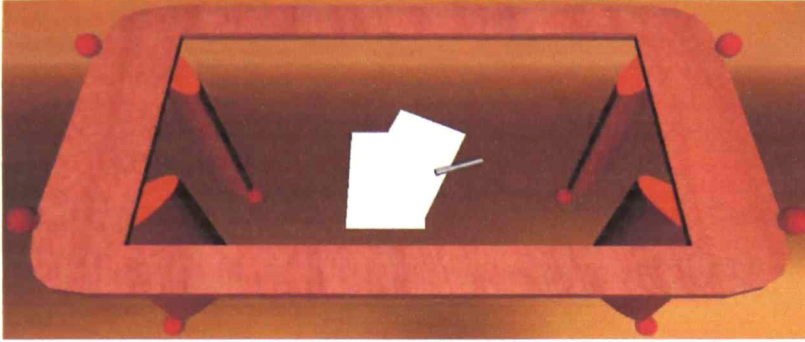


Figure 3.6: Characteristic points

Be $X_i, X_j \in X$, collision verification is executed if

$$\sqrt{(X_i(P_x) - X_j(P_x))^2 + (X_i(P_y) - X_j(P_y))^2} \leq \sqrt{X_i(S_x)^2 + X_i(S_y)^2} + \sqrt{X_j(S_x)^2 + X_j(S_y)^2}$$

When a collision occurs, there are two routes of action:

1. If one of the entities is a *pivot*, this is, is set to an absolute position, such as north, south or east, the other is moved in a perpendicular line described from the center of the pivot entity and with a magnitude equal to the outcome of equation 1.a.
2. If none of the entities is a pivot, one or both entities are moved in a perpendicular lines described form both entity center, and with a magnitude equal to half of the outcome of equation 1.a.

The second option necessitates carrying some verification before making any change in the position of the entity. First, if both entities in collision make reference to the same entity, an algorithm similar to the one used by the FL-Systems is employed to set the new position for each entity, making use of equation 1.b to validate these new positions. If the entities are unrelated, the one with the smaller volume is moved first, since smaller elements are more likely to be set in valid positions.

In some special cases, the tags assigned to a concept correspond to areas on the entity, for example, the over keyword. When the system deals with those cases, the system administrator can set the number or even the exact characteristic points that should be inside that volume, so the position can be declared valid. The equation 1.c is used in those cases.

When it is required that the entity be positioned inside another, or in a specific geographic location, the volume generated by equation 1.c is used to match most or all of the space.

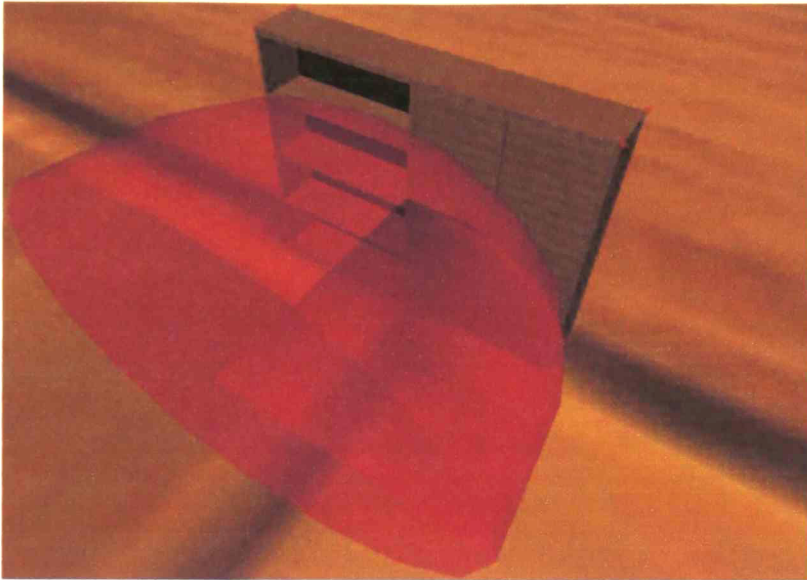


Figure 3.7: Validation volume for equation 1.b

In these cases, the system administrator can also set the characteristic points that must be contained for the position to be valid.

Previous modifications are conducted on the entities that can represent poses, using a specialized module in the architecture. This module, called the *Planning Module*, part of the research presented in [37], uses a self-conscious entity to modify the entity elements in order to represent the desired pose. Sitting, running or holding is computed by means of knowledge which describes the skeleton of the entity, and can be applied to any entity that hold the same structure. This structure can be modified to represent the lack of any extremity or element, allowing characteristics such as limping or lacking one arm. The modifications are carried out through synergy movements and using the KB to verify the new pose.

After all collisions have been solved, the CSP continues its work to verify if the new position of the entities is valid. If it verified that the entities current position passed test using constraint equations 1.b and 1.c. If that is truth, the process ends and the user is informed. Furthermore, the previous position of the entities is recorded, and the process continues. When the CSP falls in a dead end, the recorded positions are used to create a new state, and continue the constraint solving process. If the dead end cannot be solved, previous model values are used, moving the pivot entities if needed, and then verifying the new state. The previous positions are also used to determine if the new positions computed are forming

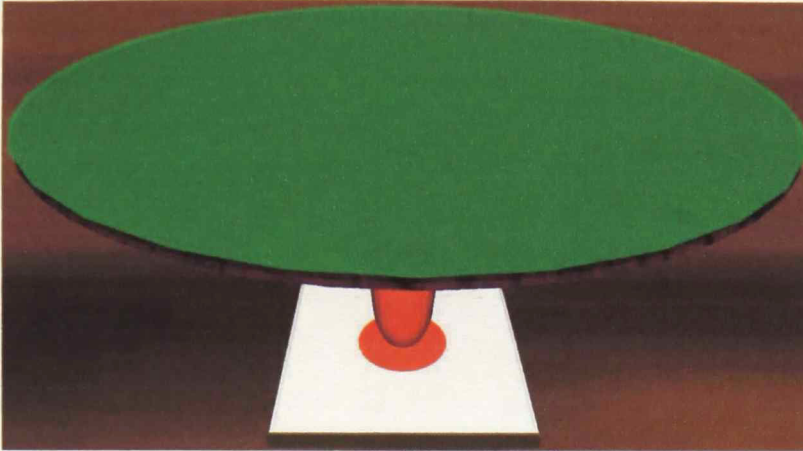


Figure 3.8: Special case: against

a cluster, and then compute a new position far away from it, thus preventing falling into local minima, where apparently there are no solutions, or local maxima, solutions with stiff constraint values, which does not allow further modifications and lead to dead ends for other entities. Other methods used to find solutions are: the total re-arrangement of the entities, rotate an entity that is being referenced by other entity in conflict, or dropping some of the entities.

If the CSP can not find a solution, an error state is raised, and the modeler can either stop the process and present the error status to the user, or can drop the violating entities (to eliminate the conflicts) and continue, presenting an error report at the end of the process.

3.5 Generation of the Outputs

When the *Model Creator* has finished constructing the model, the next step is to create the output that will be sent to the underlying architecture. This step is carried out by a MVC (Model-View Controller), which pre-process a series of templates, and then receives the model to fill the templates with model values.

3.5.1 Model-View Controller

A *Model-View Controllers* allows easy translation from data structures to file or character stream outputs. It was choose to ease communication between the VEE and the rest of

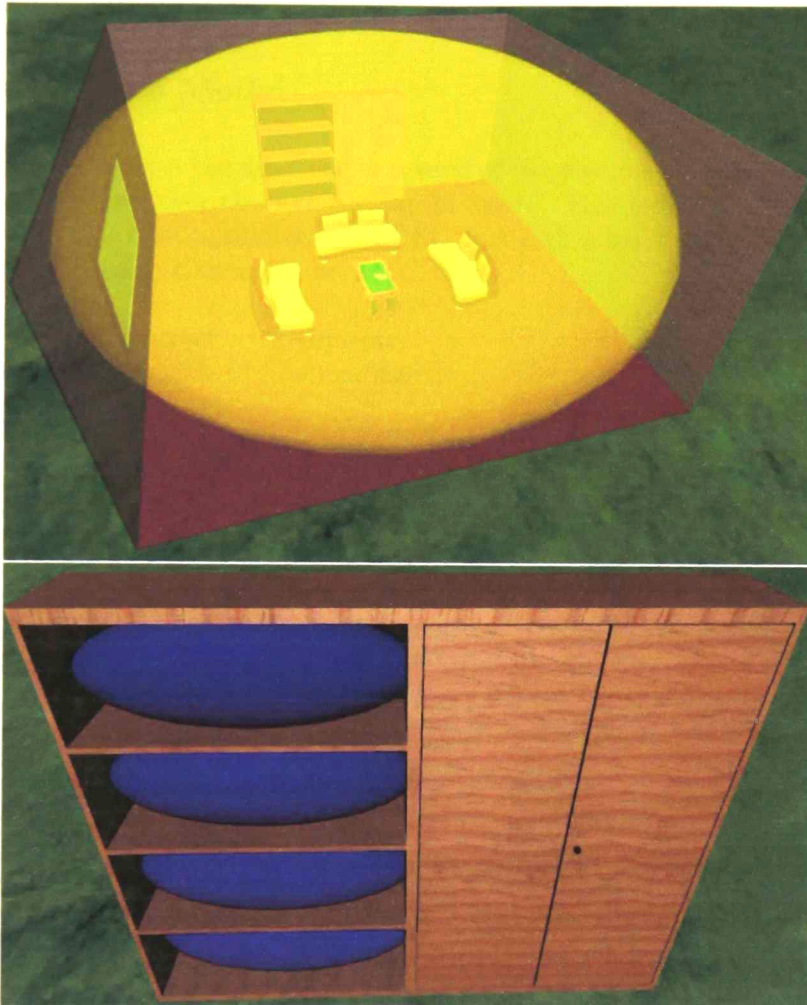


Figure 3.9: Special case: inside

the modules in the architecture. Each module requires a specific portion of the information stored in the model, and each of the modules has its own input language. It would have been difficult if the modeler had created, generated and maintained these inputs by itself. **Instead, the MVC uses a series of templates to create the outputs, using the KB to fill any additional request. This method also allows to modify the type, quantity and destination of each of these outputs.**

3.6 Modifying the Model

Once the modeler has shown the users one or several of the possible solutions for the description, direct modifications over the scenario can be made. This is handled through specific commands written in a syntax similar to VEDEL, but with a few modifications. The basic structure of a modification commands is:

```
<command ><entity identifier ><arguments ... >  
arguments ::= [ <new position > ] | [ <modifier > ]
```

A command corresponds to either position (command **M**) or properties (command **C**). The command is sent to the model, which then takes the necessary actions to ensure its execution. If the modified model fails any of the construction validations, the model is returned to its previous state, and the error is reported to the user.

To comply the modifications with the model, an interface that directly modifies the model was implemented. This is simply a VEDEL translator; the parser processes. The translator receives the command that is translated into a VEDEL-compliant sentence. Then, the parsed command is sent to the *Model Creator* module, which carries out the same verifications made for verifying the model during its creation. If the new model is tested as valid, a new output is created and sent for processing.

Chapter 4

Research Outcome

Abstract

We present the results obtained from two different prototypes, as well as some observations made during the course of the research.

4.1 Virtual Environment Editor Prototypes

During the course of the research, the methodologies proposed were applied in the implementation of a VEE, based on DM. This VEE receives a description written on the VEDEL specification and then proceeds to generate the model, presenting the output, if successful, in a X3D-compliant viewer.

This prototype was developed in the Java language, using the OWL Protégé API and the FreeMarker library, for access to KBs and using MVC methods. So far, our focus has been on modeler functionality, rather than final-user interface. Therefore the GUI for the VEE is still in early development, but is fully functional (figure 5.1).

The selected MVC was Freemarker [57], version 2.3.15, a “*template engine*”. This is a tool that creates a text output based on templates, programmed in Java and with a Java API. Although FreeMarker has some programming capabilities, it is not a fully programming language, akin to PHP, but more a data display generator. This first study case uses X3D-formatted templates that generate an output that can be viewed using any VRML97 or X3D compliant viewer.

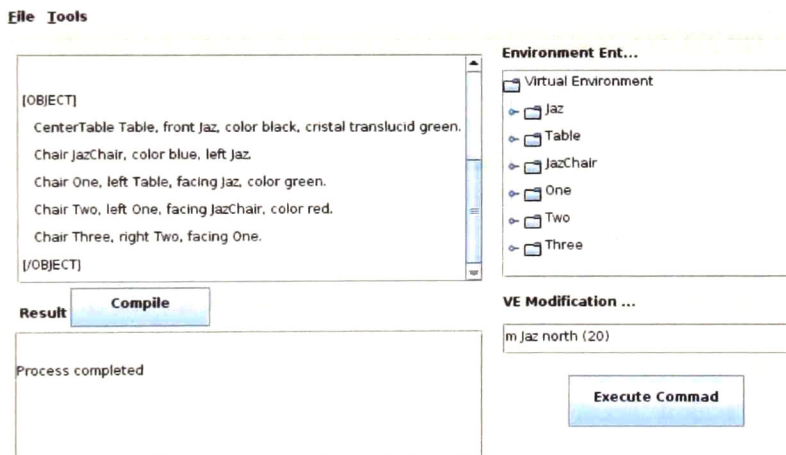


Figure 4.1: Virtual Environment Editor GUI

The KB used by this first prototype contains several entries for environments, actors, objects and keywords, and is an extension of the KB used for the prototype presented in the previous research ([56]), as presented in figure 4.2. This was a work based on the GeDA-3D prototype developed by Gutierrez [58].

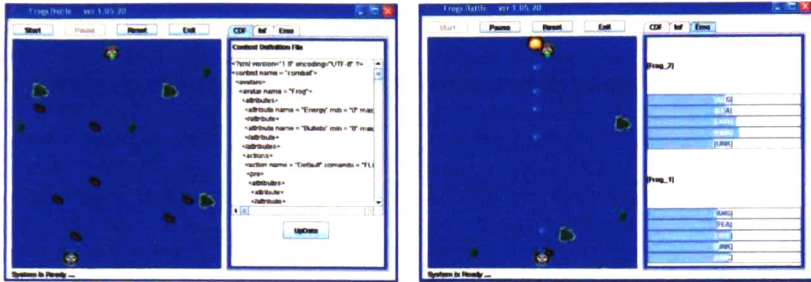


Figure 4.2: Previous Prototypes: Battle of the Frogs



Figure 4.3: Previous Prototypes: Earlier version of GeDA-3D

This first use case was a prototype for the GeDA-3D kernel, which used an earlier version of the modeler to model the initial state of the simulation, which also included the emotional machine by Razo [36]. This earlier basic modeler allowed to initialize the actors values, as well as setting its emotional behavior.

A second prototype was developed, in order to test the functionality of the earlier GeDA-3D kernel, the VEE and the Render Module, a work by Matinez [59]. This prototype sent the final output for both the kernel and the render module, which presented a limited amount of entities, letting us prove also some of the expectations for the architecture (figure 4.3).

This earlier prototypes were the basis for the current efforts, from which we obtained some models presented next.

4.1.1 GeDA-3D Virtual Environment Editor Prototype

Example 1:

[ENV]

room.

[/ENV]

[ACTOR]

ManSuit, left one, facing Table.

woman, right one, facing Table.

[/ACTOR]

[OBJECT]

bookshelf, againts NorthWall.

CenterTable Table, color black, cristal translucent gray.

Sofa one, behind Table.

Sofa 2, left Table, facing Table.

Sofa tri, right Table, facing Table.

Chair One, behind (2) ManSuit0, color green, facing ManSuit0.

Chair Two, behind (0) woman0, color red, facing woman0.

Puff seat, front Table, color black.

[/OBJECT]



Figure 4.4: Example 1: Top-Down view

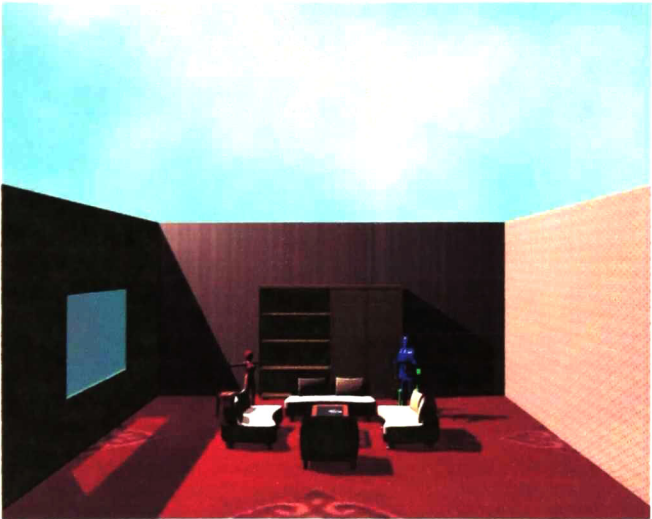


Figure 4.5: Example 1: General View

Example 2:

```
[ENV]
```

```
house.
```

```
[/ENV]
```

```
[ACTOR]
```

```
ManSuit, anywhere Kitchen.
```

```
woman, anywhere Garden.
```

```
[/ACTOR]
```

```
[OBJECT]
```

```
CenterTable Table, color black, cristal translucid gray, front (50) bed0.
```

```
[/OBJECT]
```



Figure 4.6: Example 2: House environment

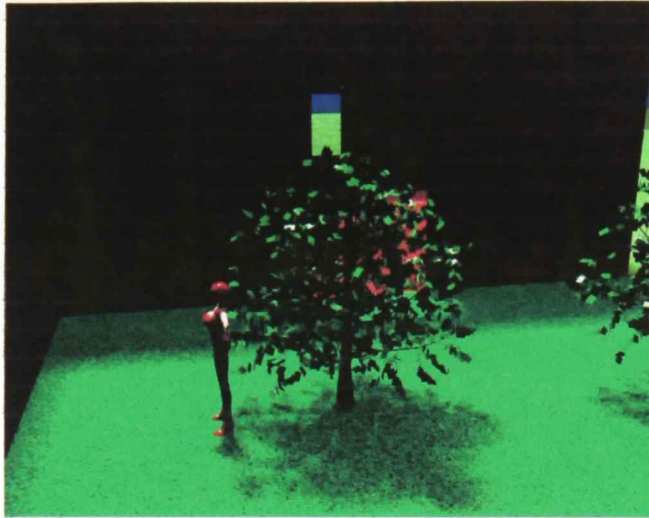


Figure 4.7: Example 2: House environment

Example 3:

```
[ENV]
  forest.
[/ENV]

[ACTOR]
  Knight One, center.
  Knight, near One.
  Knight, near One.
  Knight, near One.
[/ACTOR]

[OBJECT]

[/OBJECT]
```



Figure 4.8: Example 3: Detail view



Figure 4.9: Example 3: Top-Down View

4.1.2 DRAMA Project Module DRAMAScène

As part of the collaboration with the Institut de Recherche en Informatique de Toulouse, IRIT, we adapted part of the VEE to form part of the DRAMA Project [60].

DRAMA is a multimodule project, consisting of DRAMATexte, a tool for indexing theatrical texts, highlighting the most important elements for the director or the scenarist, and DRAMAScène, a set-in-scene visualization tool, which allows the theater company to work on a possible view for the play.

Both modules work together in the following way: first, the system receives a theatrical piece as input, and then analyzes it and tags all the characteristic elements, such as dialogues or introductions. The user then can add new tags, which will help to make a full indexation for the non-explicit elements in the play, such as movement, mood or illumination. The indexed text is then used to generate an entry for the DRAMAScène module, in which a DM takes the task of creating the visual representation for the play, and later allowing the user to modify the proposed visualization.

The DM is based on our own VEE, being the main difference the concepts stored in the KB. For this project, strong emphasis was made on the context for the model. Context plays an important role since depending on the type, era or style of the play, the model changes significantly. For example, whereas a modern play involves the actors making direct eye contact during dialogues, the style of older plays involve the actor addressing the audience all the time. Technology available in the era for the play is also taken into account the, as well as costumes and props.

In figure 4.10, we present some examples of models obtained through this variation of our VEE.

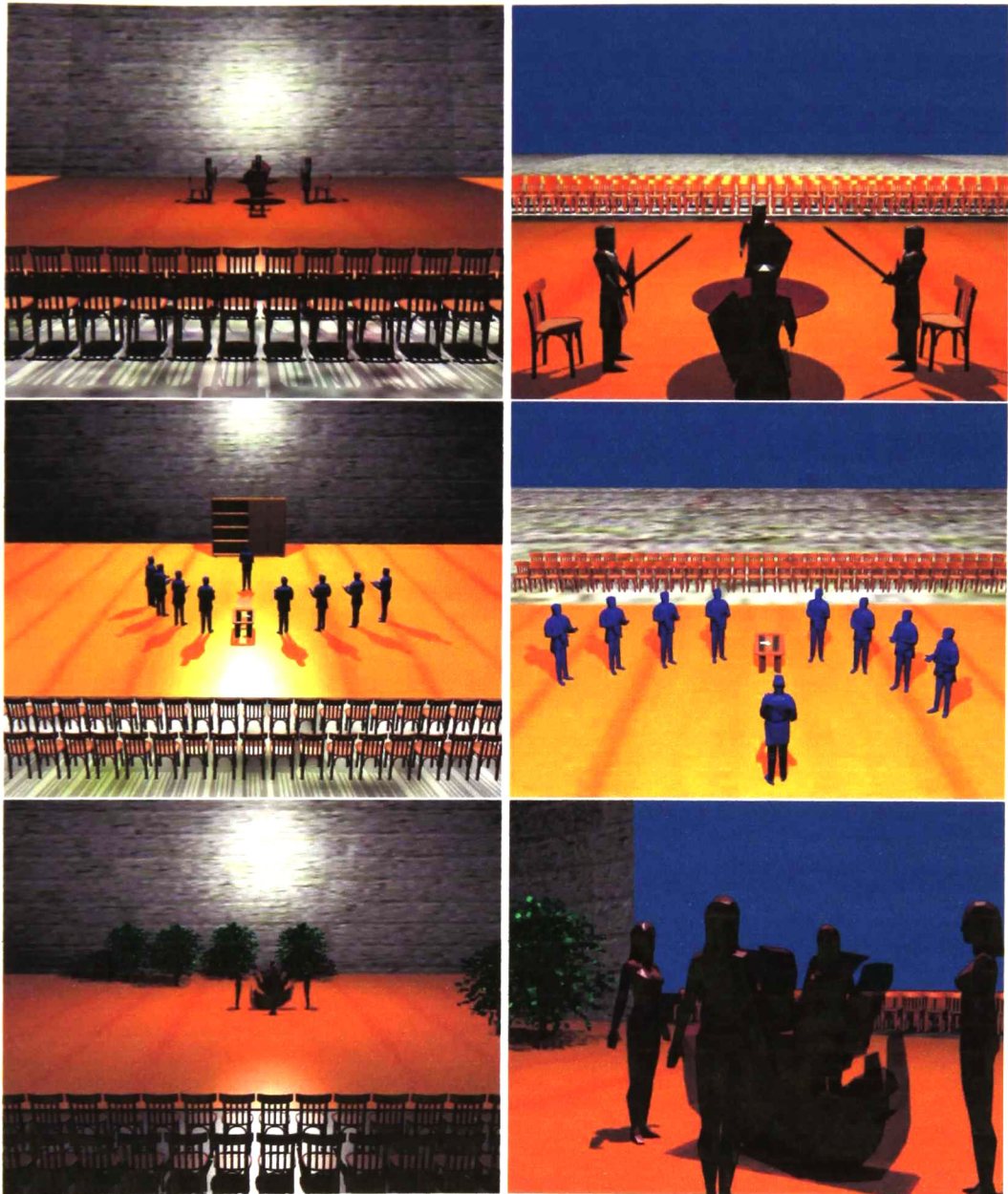


Figure 4.10: Examples of DRAMAScène Concepts

Chapter 5

Conclusions

Abstract

The last section of this documents presents the conclusions draw form the work and makes a comparison with other works. Finally, the future assets to be covered are presented.

5.1 Conclusions

Through the study of the available literature from related works, both on DM and in knowledge management, we can point several features that distinguish our research from others. First, most of the proposed input methods make use on specialized hardware, which can be intimidating for the non-experienced user, whereas we propose a direct method, which is also structured to aid in the composition of the scenario, supports both complex and simple entries. Also, they are based on the use of the mouse, which is a useful tool in 2D environments, but requires certain learning stepd to be used on 3D environments. Two researchrd propose using declarative methods: WordsEye and CAPS. WordsEyes relay on direct human language, using complex parsing methods to obtain the semantic tree used to construct the VS, leading to over-simplistic compositions in order to generate the desired output, although making it more natural to users, the web nature of the system does not allow to extend the existing object database, and even when several techniques are used to provide a visual representation of unknown concepts, the system does not make any semantic verification for the congruency of the scenario, so illogical or non-realistic situations can be created. Finally, the static output does not allow for any further interaction, and the system does not provide a method for further use of the model, that is, does not let the scenario develop into a scene, and does not provide interaction between the user and the entities created.

CAPS uses a specialized constraint declaration to construct the scenario, which is not fully presented in the literature, focusing only on the *placement* of objects, leaving their physical characteristic completely out of any modification. Also, the system uses a direct interaction method with the user: an object being placed highlights the possible surfaces that can be occupied, making the modeling process mechanic and allowing for non-logical positions, if that is the users' desire. It uses direct tags for allowing the placement of objects in cases such as "over" or "inside" but restricted to Boolean tags, which makes the positioning of new objects or incomplete entries difficult. Finally, the modeling process ends when all of the objects have been placed, without providing any methods for interaction between objects or with users.

Other projects, such as DEM²ONS, CityEngine or FL-System, are completely based on using specialized data or input methods, and provide static visualizations of the scenarios described by the users' input. Again, none of these researches deal with post-modeling tasks such as entity interaction, environment development, or the user's external input.

Our research not only focuses on the generation of a VS, that is, the positioning of elements inside the VW, but it also provides grounds for modifying most of the aspects for the scenario and the entities, either as characteristics visible through graphic representation, as well as implicit properties that can modify the way the entity behaves during the simulation run, in the form of a context associated with the model.

We do not only create the visual representation for the users' input. We also verify its congruency, and adapt the non-explicit values, based on a semantic-base (our main contribution of our proposal KB), to find conflicts, solve them, and only after this process has finished, then proceed with the visualization and animation of the world envisioned in the users' mind. None of the researches reviewed during the first phases deal with internal representation for the entities, or the rules of the worlds. The conjunction of these two parts, the visual output and the implicit representation for both the world and the entities, can work to create the simulation of a complex VE.

Our research showed, as in the figures presented in the previous sub-chapter, that a knowledge-based modeler could successfully construct a model that represents a VW, beginning with the description written on a near-natural language. This model can be integrated into the KB to be used further in the construction of more complex worlds, which can be assigned with new or complementary rules. Also, the rules dictating the construction and evolution of the VW can be modified according to the users' needs, by adjusting some values in the KB, or modifying the output-generation templates. This allows recreating almost any possible environment, with the only restriction that the elements, setting, and rules for that particular representation exist in both the KB and a 3D model database, and also, by stating the necessary information in the KB, the modeler can retrieve and even generate the necessary data to allow the evolution of the VE. In fact, that data must include the rules of the world, the entity behaviors, the relationship among entities or the values for the entity internal properties.

Project Features Comparison.

Project	Input Type	Modeling Type	Output Type	Creates Scenes	Editing Tools	Environment Navigation
GeDA-3D VEE	VEDEL Description	Declarative (CSP)	Multiple output MVC Based	Y	Y	Viewer/Renderer Based
WordsEye	Description in natural language	Declarative (Multiple methods)	Static image	N	N	Visible after render
DEM ² ONS	Multiple inputs	Declarative (CSP)	3D Render	N	Y	Yes
CAPS	Specialized Grammar	Semantic Techniques, Pseudo-physics	3D Render	N	Y	Y
ALICE	Graphic GUI	User-Based	3D Render	Y	Y	Y
FL-System	Specialized Grammar	Context Free L-System	VRML-97	N	N	Viewer/Renderer Based
City Engine	Statistical and geographical data	Extended L-Systems	Rendered Images	N	N	Y
Instant Architecture	"Split Grammar"	L-Systems	3D Render	N	N	N

5.1.1 Future Work

Several aspects must be explored in the interest of extending the reach of this research. For instance, to extend, updating and upgrading the databases, both the KB and the 3D object database must be a main objective for future researches, including a method for finding both characteristic points and the collision tags.

The other important aspect is the fully integration of the VEE with the rest of the GeDA-3D architecture, including updating the modeling process to support future movement and perception sub-modules inside the agents in charge of providing environmental self-evolution, as well as interacting with the parser module and agent community during the animation process. Finally, several arrangements can be made to the GUI for the VEE, to provide the user with a list of all the possible environments and entities available, as well as their most representative properties.

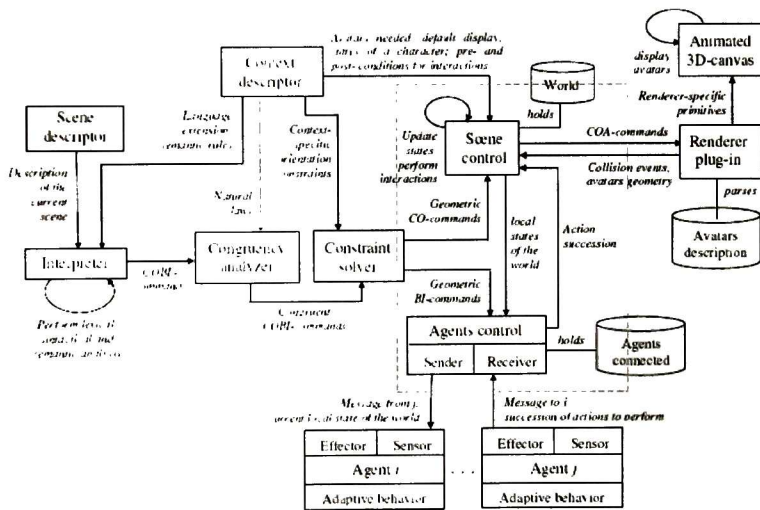


Figure 5.1: GEDA-3D Architecture

This task would help the research by:

- Allowing to design and create new use cases, giving the opportunity to verify novel concepts and the reach of the module.
- Ease the resources needed to create the model, increasing the efficiency of the modeling tool.

- Provide the necessary methods to explore VE self-evolution, and test the context-generation capabilities.
- Cover all possible aspects of DM, providing support for unspecified request and regional or local linguistic accidents.
- Present a refined GUI to final users, easing the interaction and providing end-user oriented features such as VE saving, on-the-fly rendering, online interaction and sharing, and end-user technical support.

Being part of the GeDA-3D project, this research is included as a module in the general architecture. This module, named the *Virtual Editor*, also includes a *Scene Editor* and the *Context Descriptor*. The module sends messages through the kernel during modeling time to several other modules (*Planing Module*, *Agents Module*), and sends custom-formated outputs to the rest of the architecture when the model has been approved by the user.

Bibliography

- [1] Demetri Plemenos, Georges Miaoulis, and Nikos Vassilas. Machine learning for a general purpose declarative scene modeller. In *International Conference GraphiCon'2002, Nizhny Novgorod (Russia), September 15-21, 2002*.
- [2] Véronique Gaildrat. Declarative modelling of virtual environment, overview of issues and applications. In *International Conference on Computer Graphics and Artificial Intelligence (3IA), Athènes, Grèce*, volume 10, pages 5–15. Laboratoire XLIM Université de Limoges, may 2007.
- [3] Felix Ramos, Fabiel Zúniga, and Hugo I. Piza. A 3D-space platform for distributed applications management. *International Symposium and School on Advanced Distributed Systems 2002. Guadalajara, Jal., México*, November 2002.
- [4] Dirk Fahland. Towards analyzing declarative workflows. In *Autonomous and Adaptive Web Services*, number 07061 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.
- [5] Antoine Spicher and Olivier Michel. Declarative modeling of a neural-like process. *Biosystems*, 87(2-3):281 – 288, 2007.
- [6] Dimitri Plemenos. Using artificial intelligence techniques in computer graphics. In *GraphiCon*, 2000.
- [7] Olivier Le Roux, Véronique Gaildrat, and René Caubet. Design of a new constraints solvers for 3d declarative modeling. In *International Conference on Computer Graphics and Artificial Intelligence (3IA), Limoges, 03/05/200-04/05/200*, pages 75–87, may 2000.
- [8] Olivier Le Roux, Véronique Gaildrat, and René Caubet. Using constraint satisfaction techniques in declarative modeling. In *Geometric Modeling Techniques, Applications, Systems and Tools*, pages 1–20. Kluwer Academic Publishers, 2004.

- [9] Mathieu Larive, Olivier Le Roux, and Véronique Gaildrat. Using Meta-Heuristics for Constraint-Based 3D Objects Layout. In *International Conference on Computer Graphics and Artificial Intelligence (3IA), Limoges, France, 12/05/04-13/05/04*, pages 11–23, maY 2004.
- [10] Stephane Sanchez, Olivier Le Roux, H. Luga, and Véronique Gaildrat. Constraint-Based 3D-Object Layout using a Genetic Algorithm. In *International Conference on Computer Graphics and Artificial Intelligence (3IA), Limoges, 14/05/2003-15/05/2003*, may 2003.
- [11] Ghassan Kwaiter, Véronique Gaildrat, and René Caubet. Controlling object natural behaviors with a 3d declarative modeler. In *Computer Graphics International*, pages 248–, 1998.
- [12] Pierre Barral, Guillaume Dorme, and Dimitri Plemenos. Visual understanding of a scene by automatic movement of a camera. In *GraphiCon*, 1999.
- [13] Olivier Le Roux, Véronique Gaildrat, and Réne Caubet. Using constraint propagation and domain reduction for the generation phase in declarative modeling. In *IV '01: Proceedings of the Fifth International Conference on Information Visualisation*, page 117. IEEE Computer Society, 2001.
- [14] A. Winter, A. Strübing, L. Ißler, B. Brigl, and R. Haux. Ontology-based assessment of functional redundancy in health information systems. *Lecture Notes in Computer Science*, 5421:213 – 226, 2009.
- [15] Veronique Giudicelli and Marie-Paule Lefranc. Ontology for immunogenetics: The IMGT-ONTOLOGY. *Bioinformatics*, 15(12):1047–1054, 1999.
- [16] Mike Uschold, Mike Uschold, Michael Grüninger, and Michael Gruninger. Ontologies: Principles, methods and applications. *Knowledge Engineering Review*, 11:93–136, 1996.
- [17] Gutiérrez-García J. Octavio, Koning Jean-Luc, and Ramos-Corchado Félix F. An obligation approach for exception handling in interaction protocols. In *Workshop on Logics for Intelligent Agents and Multi-Agent Systems (WLIAMAS 2009) at IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology WI-IAT'09*, Milan, Italy, September 2009. IEEE CS Press.
- [18] Stephan Grimmp, Pascal Hitzler, and Andreas Abecker. Knowledge representation and ontologies logic, ontologies and semantic web languages. draft, 2006.
- [19] Maryam Alavi and Dorothy E. Leidner. Review: Knowledge management and knowledge management systems: Conceptual foundations and research issues. *MIS Quarterly*, 25(1):107–136, 2001.

- [20] Thomas R. Gruber. A translation approach to portable ontology specifications. *Knowl. Acquis*, 5(2):199–220, June 1993.
- [21] Phillip Breay. The social ontology of virtual environments – criticisms and reconstructions. *The American Journal of Economics and Sociology*, 62(1):269–282, January 2003.
- [22] A. Farquhar, R. Fikes, and J. Rice. The ontolingua server: A tool for collaborative ontology construction. Technical report, Knowledge Systems Laboratory, Stanford University, 1996.
- [23] John Dominguez. Tadzebao and webonto: Discussing, browsing, and editing ontologies on the web. In *In Proceedings of the Eleventh Workshop on Knowledge Acquisition, Modeling and Management, KAW'98, Banff, Canada*, April 1998.
- [24] E. Motik. *Reusable Components for Knowledge Modelling: Case Studies in Parametric Design Problem Solving*. IOS Press, Amsterdam, The Netherlands, The Netherlands, 1999.
- [25] W. E. Grosso, H. Eriksson, R. W. Ferguson, J. H. Gennari, S. W. Tu, and M. A. Musen. Knowledge modeling at the millennium (the design and evolution of protege-2000). Technical report, Stanford Medical Informatics, 1998.
- [26] Henrik Eriksson, Yuval Shahar, Samson W. Tu, Angel R. Puerta, and Mark A. Musen. Task modeling with reusable problem-solving methods. *Artificial Intelligence*, 79(2):293–326, 1995.
- [27] Deborah L. McGuinness and Frank van Harmelen. OWL Web Ontology Language Overview. W3C recommendation, World Wide Web Consortium, February 2004. <http://www.w3.org/TR/2004/REC-owl-features-20040210/>.
- [28] Graham Klyne and Jeremy J. Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C recommendation, World Wide Web Consortium, February 2004. <http://www.w3.org/TR/2004/REC-rfd-concepts-20040210/>.
- [29] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003.
- [30] Solomon W. Golomb and Leonard D. Baumert. Backtrack programming. *J. ACM*, 12(4):516–524, 1965.
- [31] John Gary Gaschnig. *Performance measurement and analysis of certain search algorithms*. PhD thesis, Carnegie-Mellon Univ. Pittsburgh Pa. Dept. Of Computer Science, 1979.

- [32] Rina Dechter. Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition. *Artif. Intell.*, 41(3):273–312, 1990.
- [33] Robert M. Haralick and Gordon L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artif. Intell.*, 14(3):263–313, 1980.
- [34] Steve Aukstakalnis and David Blatner. *Silicon Mirage, The Art and Science of Virtual Reality*. Peachpit Press, Berkeley, CA, USA, 1992.
- [35] Jerry Isadle. What is virtual reality?, a web-based introduction. WebPage, 1998. <http://vr.isdale.com/WhatIsVR/frames/WhatIsVR4.1.html>, Last visited 06/14/2007.
- [36] Luis Alfonso Razo Ruvalcaba. Algoritmos de comportamiento y personalidad para agentes emocionales. Master's thesis, Centro de Investigación y de Estudios Avanzados del IPN, Unidad Guadalajara, 2007.
- [37] Orozco H. R., Ramos F., Zaragoza J., and D. Thalmann. *Frontiers in Artificial Intelligence and Applications (Advances in Technological Applications of Logical and Intelligent Systems)*, volume 186, chapter Avatars Animation Using Reinforcement Learning in 3D Distributed Dynamic Virtual Environments, pages 67–84. IOS Press, Washington, DC, 2009.
- [38] Philippe Codognet. Declarative behaviors for virtual creatures. In *SIGGRAPH '99: ACM SIGGRAPH 99 Conference abstracts and applications*, page 237. ACM, 1999.
- [39] Samuel R. Buss. *3D Computer Graphics: A Mathematical Introduction with OpenGL*. Cambridge University Press, 2003.
- [40] Jean-Eudes Marvie, Julien Perret, and Kadi Bouatouch. The fl-system: a functional l-system for procedural geometric modeling. *The Visual Computer*, 21(5):329–339, jun 2005.
- [41] Yoav I. H. Parish and Pascal Müller. Procedural modeling of cities. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 301–308, New York, NY, USA, 2001. ACM.
- [42] Mathieu Larive, Yann Dupuy, and Véronique Gaildrat. Automatic generation of urban zones. In *WSCG (Short Papers)*, pages 9–12, 2005.
- [43] Stefan Göbel, Oliver Schneider, Ido Iurgel, Axel Feix, Christian Knöpfle, and Alexander Rettig. Virtual human: Storytelling and computer graphics for a virtual human platform. *Lecture Notes In Computer Science*, pages 79–88, 2004.
- [44] Riva G. Application of virtual reality in medicine. *Methods of information in medicine*, 5(5):524–534, October 2003.

- [45] M. J. North, T. R. Howe, N. T. Collier, and J. R. Vos. *A Declarative Model Assembly Infrastructure for Verification and Validation*, pages 129–140. Springer Japan, 2007.
- [46] R. Raymond Lang. A declarative model for simple narratives. In *Proceedings of the AAAI Fall Symposium on Narrative Intelligence*, pages 134–141. AAAI Press, 1999.
- [47] P. Prusinkiewicz and Aristid Lindenmayer. *The algorithmic beauty of plants*. Springer-Verlag New York, Inc., 1990.
- [48] Peter Wonka, Michael Wimmer, François Sillion, and William Ribarsky. Instant architecture. *ACM Transactions on Graphics*, 22(4):669–677, july 2003.
- [49] Bob Coyne and Richard Sproat. Wordseye: An automatic text-to-scene conversion system. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 487–496. AT&T Labs Research, 2001.
- [50] G. Kwaiter, V Gaildrat, and R. Caubet. Dem²ons: A high level declarative modeler for 3D graphics applications. In *Proceedings of the International Conference on Imaging Science Systems and Technology, CISST'97*, pages 149–154, 1997.
- [51] Issn x, D. Wang, D. Wang, I. Herman, I. Herman, G. J. Reynolds, and G. J. Reynolds. The open inventor toolkit and the premo standard, 1997.
- [52] "The Open Group" Motif 2.1-programmer's guide, 1997.
- [53] William Ruchaud and Dimitri Plemenou. Multiformes: A declarative modeller as a 3D scene sketching tool. In *ICCVG*, 2002.
- [54] Ken Xu. Constraint-based automatic placement for scene composition. In *In Graphics Interface*, pages 25–34, 2002.
- [55] Stephen Cooper, Wanda Dann, and Randy Pausch. Teaching objects-first in introductory computer science. In *SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on Computer science education*, pages 191–195, New York, NY, USA, 2003. ACM.
- [56] Jaime Alberto Zaragoza Rios. Representation and exploitation of knowledge for the description phase in declarative modeling of virtual environments. Master's thesis, Centro de Investigación y de Estudios Avanzados del Intituto Politécnico Nacional, Unidad Guadalajara, Guadalajara, México, 2006.
- [57] Mike Bayer Benjamin Geer. Freemarker: Java template engine libray. webpage, december 2008.

- [58] Alonso Gutierrez Aguirre. Núcleo geda-3D. Master's thesis, Centro de Investigación y de Estudios Avanzados del IPN, Unidad Guadalajara, 2007.
- [59] Alma Verónica Martínez González. Lenguaje para animación de creaturas virtuales. Master's thesis, Centro de Investigación y de Estudios Avanzados del IPN, Unidad Guadalajara, 2005.
- [60] Andriamarozakaniaina T., Pouget M., Zaragoza J., and Gaildrat V. Dramatexte: indexation et base de connaissances. Premier Colloque international sur la notation informatique du personnage, may 16-17, 2008, Toulouse, France. Publishing pending.



CENTRO DE INVESTIGACIÓN Y DE ESTUDIOS AVANZADOS DEL I.P.N. UNIDAD GUADALAJARA

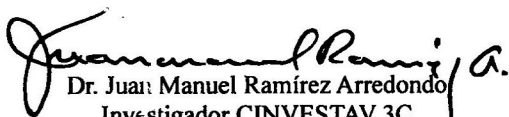
El Jurado designado por la Unidad Guadalajara del Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional aprobó la tesis

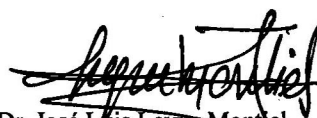
Modelado Declarativo Auxiliado por Conocimiento - Declarative Modeling Based On Knowledge


del (la) C.


Jaime Alberto ZARAGOZA RIOS

el día 15 de Diciembre de 2009.



Dr. Juan Manuel Ramírez Arredondo
Investigador CINVESTAV 3C
CINVESTAV Unidad Guadalajara

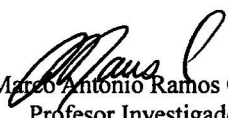

Dr. José Luis Leyva Montiel
Investigador CINVESTAV 3B
CINVESTAV Unidad Guadalajara


Dr. Luis Ernesto López Mellado
Investigador CINVESTAV 3B
CINVESTAV Unidad Guadalajara


Dr. Félix Francisco Ramos Corchado
Investigador CINVESTAV 3A
CINVESTAV Unidad Guadalajara


Dr. Jean-Luc Koning
Vice President of International
Relations
Grenoble Institute of Technology


M.C. Veronique Gaildrat
Maestra de Conferencias
Institute de Recherche en
Informatique de Toulouse, Toulouse
Francia


Dr. Marco Antonio Ramos Corchado
Profesor Investigador
Universidad Autónoma del Estado de México

