Vrije Universiteit Brussel
Faculty of Science
Department of Computer Science

From Knowledge Representation to Virtual Reality Environments

Wesley Bille
Academic Year
2001 – 2002

Essay brought forward to achieve the degree
of Licentiate in the Applied Computer Sciences

Promoter: Prof. Dr. Olga De Troyer

Vrije Universiteit Brussel
Faculteit Wetenschappen
Departement Informatica

Van Kennis Representatie naar Virtuele Realiteit Omgevingen

Wesley Bille
Academiejaar
2001 – 2002

Verhandeling voorgedragen tot het behalen van
de graad van Licentiaat in de Toegepaste Informatica

Promoter: Prof. Dr. Olga De Troyer

Abstract

Virtual Reality is used in lots of different applications and in solving different problems. The filming industry has been using the possibilities of Virtual Reality for years. If they hadn't used Virtual Reality in the film 'Titanic' then the film would have looked very different as what we saw now. Virtual Reality is also used in medical research to map the different parts of the human body in detail. Thanks to this detailed data they can track all sorts of diseases and physical handicaps more easily and faster. Even for preparing a complex surgery they use this data. The industry also uses Virtual Reality to develop new products and prototypes. Car industry for example uses it for simulating crash test. In Formula One they use Virtual Reality for the visualization of their measurements. This visualization brings the efforts of the cars during a race using a 3D-interface. Also on the World Wide Web we can find Virtual Reality. Some sites are using the VRML technique, which stands for 'Virtual Reality Modeling Language'. VRML gives the possibility to picture for example a complete city in which the visitors of the Web site can walk around. Another important, and probably the most known application of Virtual Reality are computer games. 3D computer games have more or less become some standard in the gaming world. The gaming industry gave an enormous impulse to the development of 3D-techniques.

But what exactly is Virtual Reality? The term Virtual Reality (VR) is used by many different people and in many different meanings. Aukstakalnis and Blatner gave the following definition: *"Virtual Reality is a way for humans to visualize, manipulate and interact with computers and extremely complex data."* Visualization refers to the computer generating visual, auditory or other sensual outputs to the user of a world within the computer. This world can be a scientific simulation or a view into a database. The user can interact with the world and directly manipulate objects within the world. Other processes, perhaps physical simulations, or simple animation scripts animate some worlds. Interaction with the Virtual World, at least with near real time control of the viewpoint, can be seen as a critical test for 'a virtual reality'.

There are currently quite a number of different efforts to develop VR technology. There are two major categories for the available VR software; these are toolkits and authoring systems. Toolkits are programming libraries that provide a set of functions with which a skilled programmer can create VR applications. Authoring systems are complete programs with graphical interfaces for creating worlds without resorting to detailed programming. They usually use some sort of scripting language in which to describe complex actions. So they are not really non-programming. The programming libraries are generally more flexible and have faster renders than authoring systems, but you must be a very skilled programmer to use them.

In this thesis we want to try to generate Virtual Worlds based on some general physical description of the objects in the world that somebody wants to create. We will show the possibility of it as being a simple manner for developing Virtual Reality environments without losing the advantages of using the programming libraries. However, this will be demonstrated on a very basic level, but detailing this approach just follows the general line introduced in this document.

Samenvatting

Virtuele Realiteit wordt gebruikt in veel verschillende applicaties en bij het oplossen van problemen. De filmindustrie gebruikt de mogelijkheden van Virtuele Realiteit reeds jaren. Indien men geen gebruik had gemaakt van Virtuele Realiteit in de film 'Titanic', dan zou de film er geheel anders hebben uitgezien dan wat we nu zagen. Virtuele Realiteit wordt ook gebruikt in de medische wetenschap om de verschillende onderdelen van het lichaam in detail in beeld te brengen. Dankzij deze gedetailleerde gegevens kan men allerlei ziektes en afwijkingen makkelijker en sneller opsporen. Dit kan zelfs gebruikt worden voor het voorbereiden van een ingewikkelde operatie.Ook de industrie gebruikt Virtuele Realiteit voor het ontwikkelen van nieuwe producten en prototypes. De autoindustrie maakt er bijvoorbeeld gebruik van om crash testen te simuleren. In Formule 1 maakt men gebruik van Virtuele Realiteit voor het visualiseren van metingen. Deze visualisatie geeft de prestaties van de auto's gedurende de wedstrijd weer, gebruik makend van een 3D–interface. Ook op het World Wide Web vinden we Virtuele Realiteit terug. Sommige sites maken gebruik van de Virtual Reality Modeling Language, ofwel VRML. Een mogelijk van VRML is bijvoorbeeld om een complete stad te bouwen waarin bezoekers van de website kunnen rondwandelen. Een andere belangrijke en waarschijnlijk één van de meest gekende applicaties van Virtuele Realiteit zijn computerspellen. 3D computerspellen zijn min of meer een standaard geworden. De computerspel-industrie heeft een enorme impuls gegeven aan de ontwikkeling van 3D-technieken.

Maar wat exact is nu Virtuele Realiteit? De term Virtuele Realiteit (VR) wordt door veel verschillende mensen gebruikt voor veel verschillende betekenissen. Aukstakalnis en Blatner gaven de volgende definitie: *"Virtuele Realiteit is een manier voor mensen om te visualiseren, manipuleren en interaheren met computers en extreem complexe data."* Visualiseren refereert naar de generatie van visuele, auditorische en andere zintuigelijke output afkomstig van de computer, bestemd voor de gebruiker van een wereld binnenin een computer. Deze wereld kan een wetenschappelijke simulatie zijn of evengoed een zicht op een databank. De gebruiker kan interaheren met de wereld en kan rechtstreeks objecten binnen de wereld manipuleren. Andere processen, mogelijk fysische simulaties of simpele animatie scripts, animeren de wereld. Interactie met de Virtuele Wereld met een bijna onmiddelijke controle over wat zichtbaar is, kan gezien worden als een kritische test voor 'Virtuele Realiteit'.

Tegenwoordig zijn er een aantal verschillende inspanningen voor het ontwikkelen van VR technologie. Er zijn hoofdzakelijk twee categoriën binnen de beschikbare VR software, namelijk toolkits en authoring systemen. Toolkits zijn programma bibliotheken die een verzameling van functies aanbieden die door een getrainde programmeur aangewend kunnen worden voor het creëren van VR applicaties. Authoring systemen zijn volledige programma's met een grafische interface voor het creëren van werelden zonder te moeten terugvallen op gedetailleerd programmeren. Meestal maken ze gebruik van een eenvoudige scripting taal die gebruikt kan worden voor het beschrijven van complexe acties. Eigenlijk is deze benadering dus niet echt zonder programmeren. De programma bibliotheken zijn meestal veel flexibeler en hebben vluggere renderers dan authoring systemen, maar er zijn getrainde programmeurs nodig om deze te hanteren.

In deze thesis willen we proberen om Virtuele Werelden te genereren vertrekkende vanuit een algemene fysische beschrijving van de objecten in de wereld die iemand wil creëren. We gaan de mogelijkheid ervan aantonen als zijnde een eenvoudiger manier voor het ontwikkelen van Virtuele Realiteit omgevingen, zonder de voordelen van het gebruik van programma bibliotheken te verliezen. Dit zal aangetoond worden op een basis niveau, maar het uitbreiden van deze benadering volgt uit de algemene lijn die geïntroduceerd wordt in dit document.

Acknowledgement

I would like to thank Prof. O. De Troyer for making it possible to realize this thesis and for her suggestions and help during the development of this document. I would also like to thank Peter Stuer for all the suggestions and the discussions we had, which contributed to the quality of this thesis. Furthermore I would like to thank all professors I had during my four-year education at the Vrije Universiteit Brussel for all things they taught us and for the patience they had to answer all kinds of questions we had. Thanks also go to Peter Plessers, with who I had a lot of interesting discussions and with who I did a lot of collaborative work during my education. Lots of thanks also go to my family and friends, for making it possible for me doing these studies, and for being patient and trying to understand me during some of my bad mood periods.

TABLE OF CONTENTS

LIST OF FIGURES

OUTLINE

Nowadays there are two different approaches in developing a Virtual Reality application. These are toolkits and authoring systems. Toolkits are programming libraries that provide a set of functions with which a skilled programmer can create VR applications. This is the most flexible approach and provides faster rendering. Authoring systems are complete programs with graphical interfaces for creating worlds without resorting to detailed programming. However, the implementer of the virtual environment still needs a notion of programming as complex actions need to be described in some sort of scripting language.

The goal of this thesis is to set a new dimension in creating Virtual Reality environments. We want to create VR environments based on some general physical description of the components of which the world consists of. This must bring the advantage that domain experts can express the world they want to generate using their own familiar domain vocabulary without having to be familiar with programming. This will lead to fast Virtual World design and it will provide consistent VR world interactions and behavior as eliminating programming for leeks directly eliminates programming errors that otherwise are very likely to exist.

Now, for doing this we need two things. First of all we need some knowledge representation system in which somebody can declare the physical description of the VR world components. The research towards such a representation mechanism is explained in the first and second chapter. The first chapter introduces knowledge representation mechanisms that exist in the field of artificial intelligence, while the second chapter introduces techniques used in the development of the semantic web.
The second thing we need is some technique for visualizing the described world. In this approach we use a programming library, or a 3D-engine. Chapter three introduces some of these libraries and gives an overview of the engine we are using here in the solution.

Onces we have these two technologies, we can start out thinking about some mechanism for mapping the knowledge expressed in our representation formalism towards a visualization rendered by our toolkit. The complete solution is explained in chapter four.

Finally, chapter five describes the general conclusions we can draw from this research.

1 KNOWLEDGE REPRESENTATION IN ARTIFICIAL INTELLIGENCE

1.1 INTRODUCTION

To give computers access to structured collections of information and sets of inference rules that they can use to conduct automated reasoning, artificial intelligence researchers have studied such systems in detail. In AI, a representation of knowledge is a combination of data structures and interpretive procedures that, if used in the right way, will lead to knowledgeable behavior. Work on knowledge representation in AI has involved the design of several classes of data structures for storing information in computer programs, as well as the development of procedures that allow 'intelligent' manipulation of these data structures to make inferences.

One of the assumptions underlying most work in artificial intelligence is that intelligent behavior can be achieved through the manipulation of symbol structures representing bits of knowledge. Knowledge representation languages have been developed to make this easier. These are special notations that make it easy to represent and reason with complex knowledge about the world. This chapter will introduce the main approaches to knowledge representation.

Before I will talk about these different languages, I want to give an overview of the requirements for representation languages. First of all, a knowledge representation language should be able to represent adequately complex facts in a clear, precise and natural way. It should also allow you to deduce new facts from the existing knowledge. All the requirements are examined below.

Representational adequacy is the ability to represent adequately complex facts. Indeed, some facts are hard to represent in a way that allows those facts to be reasoned with. For example, a fact like "Joe believes no-one likes crashing computers", how can we reason with this representation and conclude that Joe believes that Harry doesn't like crashing computers?
A *well-defined syntax and semantics* is necessary to represent knowledge in a clear and precise way. We have to know what the expressions are we can use in the language and what they mean. However, it is not enough to have a precise syntax and semantics if this means that your representation scheme is non-intuitive and difficult to use and understand. This is why there is the requirement for our representation scheme to be *reasonably natural*.
Inferential adequacy is our final requirement. It means that it must be possible to deduce new facts from existing knowledge. We cannot explicitly represent everything that the system might ever need to know, some things should be left to be deduced by the system when needed in problem solving. However, making arbitrary deductions from existing knowledge is a complex process. Generally, there is a trade-off between what we can infer and how quick it can be inferred. However, no representation language satisfies all these requirements perfectly. I will now give an overview of the main approaches to knowledge representation.

1.2 SEMANTIC NETWORKS

1.2.1 Introduction to semantic networks

Semantic nets were originally developed to represent the meanings of English words. The term dates back to Ross Quillian's Ph.D. thesis in 1968, in which he first introduced it as a way of talking about the organization of human semantic memory, or memory for word concepts. The idea of a semantic network, a network of associatively linked concepts, is very much older. Anderson and Bower (1973) claim to be able to trace it all the way back to Aristotle.
In semantic nets, all the information is represented as a set of nodes connected to each other by a set of labeled arcs, which represent relationships among the nodes. The most important relations between concepts are *subclass* relations between classes and *instance* relations between particular object instances and their parent class. There are however many other relations allowed, such as *has-part, color,* …The following example demonstrates the subclass relationship and the instance of relationship.

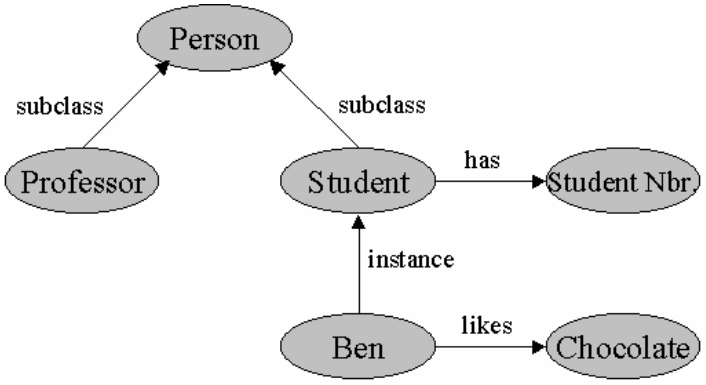

Figure 1: example of a simple semantic network

As we can see in the above example, subclass relations define a class hierarchy. Also, the subclass and instance relationship may be used to derive new information that is not explicitly represented. We should for example be able to derive that Ben has a student number.
Notice that each property stores a one-way link, such as the arc from Student to Student Nbr. To store bi-directional links, it is necessary to store each half separately. If we want to be able to answer the question "Who owns a student number?", we would need an arc from Student Nbr. to Student.
From what we have seen so far it is clear that semantic networks can be used to represent relationships that would appear as binary predicates in predicate logic (see later). But knowledge expressed by other predicates can also be expressed in semantic nets. For example, one-place predicates can be thought of as binary predicates, using some very general-purpose predicates. In the following example we express the one-place predicate "Man(Ben)" in a semantic network.

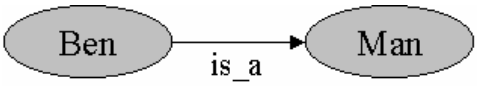

Figure 2: example of a unary fact represent by a binary relation

Three or more place predicates can also be transformed into a binary form by creating one new object representing the entire predicate statement and then introducing binary predicates to describe the relationships to this new object. Suppose we want to represent the following information: "3-4 was the score of England against Belgium", then we can do this as in the following example.

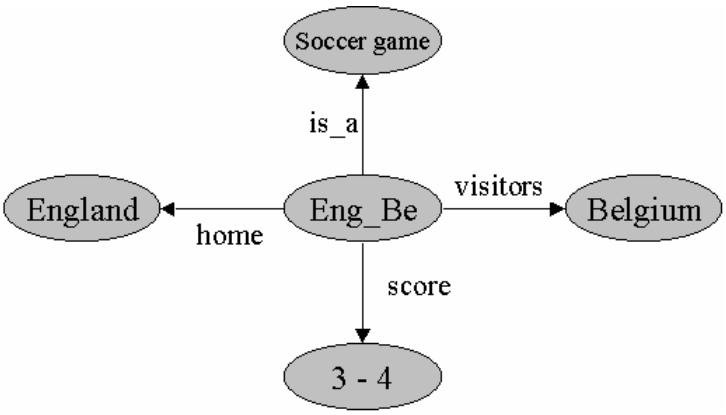

Figure 3: representation of a n-ary predicate by some binary predicates

When semantic networks became popular in the 1970's there was much discussion about what the nodes and the relations really meant. People were using them in subtly different ways, which led to much confusion. As we mentioned in the introduction, it is important to state, as precisely as possible, the semantics of a representation language. This way we know exactly what expressions mean. One simple way to describe the meaning of nodes and links in a semantic network is in terms of set theory. We can interpret a class node as denoting a set of objects. So the student node (figure 1) denotes the set of all students. Nodes such as Ben denote individuals. The instance relationship can thus be defined in terms of set membership. The subclass relationship can be defined in terms of a subset relation.

1.2.2 Advantages of semantic networks

Semantic networks allow us to represent knowledge about objects and relations between objects in a simple and intuitive manner. The conventional graphical notation allows us to see how the knowledge is organized.

1.2.3 Disadvantages of semantic networks

The sorts of inferences that are normally supported are very restrictive. So semantic networks are not likely to be adequate if you have to draw a wide range of different sorts of inferences and not just inferences based on inheritance of properties.

1.3 FRAMES AND SCRIPTS

1.3.1 Introduction to frames and scripts

Frames were originally proposed by Minsky as a basis for understanding visual perception, natural language dialogues and other complex behaviors. Scripts are frame-like structures that were specifically developed for representing sequences of events. They have been developed by Schank and Abelson. Frames and scripts are both methods of organizing the knowledge representation in a way that directs attention and facilitates recall and inference.

The following example is a frame representing the same knowledge as we have represented in figure 1.

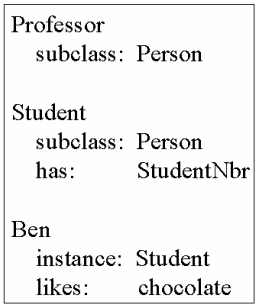

Figure 4: frame example

Professor, Student and Ben are objects in the frame system, properties such as likes and has, are sometimes referred to as slots and chocolate and Student Nbr as slot values. It is straightforward to translate between semantic networks and frame based representations. Nodes become objects, links become slots and the node on the other end of the link becomes the slot value.
Both slot values and slots may themselves be frames. Many systems allow also for slots to include procedures. It is often useful to describe what should be done whenever a slot is filled or how a value for a slot can be computed. The use of such procedures embedded within an otherwise declarative structure is referred to as 'procedural attachment'.

Most frame systems allow you to state which properties (or slots) that are just typical for a class, with exceptions allowed, and which must be true for all instances. The value of a slot that is only typical for a class is a default value, which can be overridden by giving a different value for an instance or a subclass. So objects and classes inherit the default slots of their parent classes unless they have an individual slot which conflicts with the inherited one.
Now, most systems also allow multiple inheritance which makes things a little more complex. Multiple inheritance means that more than one parent is allowed. Suppose we have a class Teacher that has a slot status with the value 'academic personnel' and we have a class Student that has a slot status with the value 'daystudent'. Now if we have a class Teaching Assistent, then this class can inherit both from Student and from Teacher. But what is now the status for a Teaching Assistant? Therefore a frame system must have some mechanism to decide which value to inherit if there are such conflicts. Sometimes it is better to create a new class that does not inherit from both parent classes.

With all these features mentioned here, it may be hard to predict exactly what will be inferred about a given object just by looking at the set of frames. We will also have to know something about how the underlying frame system is implemented.

Before finishing about frames, I will give some more explanation about scripts. A script is a structure that describes a stereotyped sequence of events in a particular context. A script also has a set of slots. The definition of a script is very similar to that of a frame. However, a script has some important components. These components are:

- *Entry conditions*: these are conditions which must be satisfied before the events described in the script can take place
- *Result*: these are the conditions that will be true after the events described in the script took place
- *Properties*: these are slots representing objects that are involved in the events described in the script
- *Roles*: these are slots representing people who are involved in the events described in the script
- *Track*: this mentions a specific variation on the more general pattern that is represented by the script
- *Scenes*: this represents the actual sequences of events that occur.

The events described in a script form a giant causal chain. The beginning of the chain is the set of entry conditions. The end of the chain is the set of results.

1.3.2 Advantages of frames

Frames provide a fairly simple and clear way of representing properties of objects and categories of objects, just like semantic networks. A basic type of inference is defined, whereby objects may inherit properties of parent objects.

1.3.3 Disadvantages of frames

There are many things that cannot easily be represented when using frames. It is hard to express negation, disjunction and certain types of quantification, that is, representing that something is true for ‘all’ or ‘some’ of the category of objects.

1.4 LOGIC

Logic is one of the first representation schemes used in AI. Predicate logic is probably one of the most important knowledge representation languages. We also have propositional logic, which is much simpler. A logic is a formal system which is defined in terms of it's syntax, its semantics and its proof theory which says how we can draw new conclusions given some statements in the logic. I will first give a brief overview of propositional logic, followed by an overview of predicate logic and concluding with the use of logic in AI.

1.4.1 Propositional logic

First of all, I will give an overview of the syntax of propositional logic. The alphabet of propositional logic contains three sorts of symbols. A proposition, which is represented by a propositional character, is a statement expressed as a sentence. We can for example have the proposition 'Ally likes green beans'. We can represent this with the symbol P, which we then call a proposition letter. Sometimes propositional characters are also referred to as atomic formulas or just atoms.
Second sorts of symbols are the logic symbols. There are five different logic symbols, $\neg$ (negation), $\wedge$ (conjunction), $\vee$ (disjunction), $\rightarrow$ (implication), $\leftrightarrow$ (equivalence). These symbols are also called logic connectives. The following example illustrates the use of logic connectives. We can have the symbol P that means 'Amy is crying' and the symbol Q that means 'Amy is running'. Then, $P \wedge Q$ will mean that 'Amy is crying and Amy is running'.
Third sorts of symbols are the help symbols like brackets.
With what we have seen now, we can give the definition of an alphabet in the propositional logic.

Definition

An alphabet from propositional logic consists of:

a. a set of propositional characters
b. logic symbols: $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$
c. help symbols:) and (

Now we can go further with the semantics of propositional logic. The values true and false are called truth-values. Sometimes people use 0 if they mean false and 1 if they mean true. For atoms we cannot analyze the truth further. We have to assume that they are true or false. But how can we now calculate the truth-value of a non-atomic formula like for example $P \wedge Q$? $P \wedge Q$ will be true if both P and Q are true otherwise it will be false. All of these calculations can be represented in a table, called a truth table. Figure 5 shows all the truth tables for negation, disjunction, conjunction, implication and equivalence.

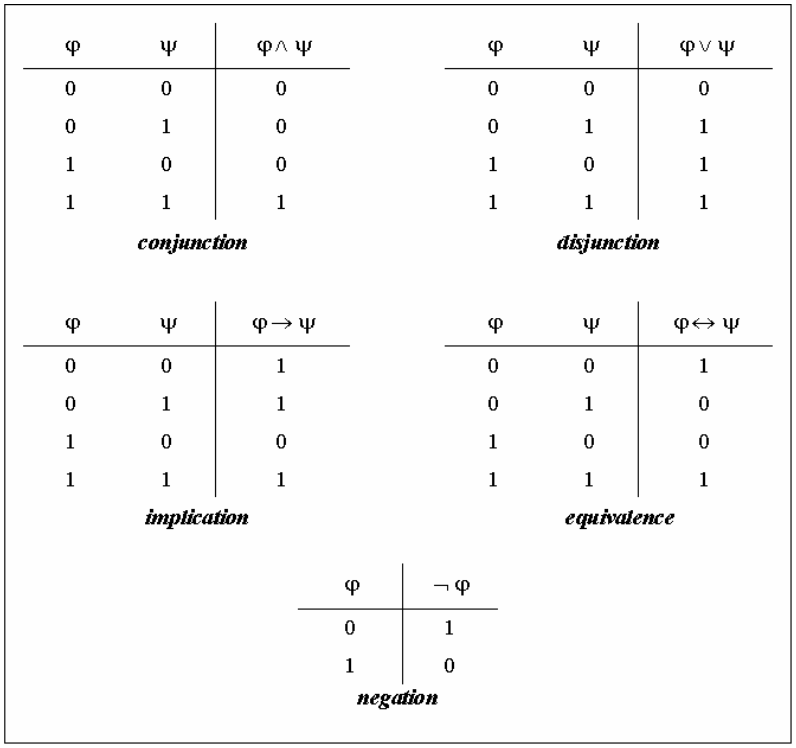

Figure 5: Truth tables in propositional logic

Now we have the truth tables for the standard logic connectives, we can calculate the truth-value of much more complex formulas. Figure 6 shows an example of the calculation for $((p \vee q) \wedge \neg r)$.

p	q	r	$(p \vee q)$	$\neg r$	$(p \vee q) \wedge \neg r$
0	0	0	0	1	0
0	0	1	0	0	0
0	1	0	1	1	1
0	1	1	1	0	0
1	0	0	1	1	1
1	0	1	1	0	0
1	1	0	1	1	1
1	1	1	1	0	0

Figure 6: truth table of complex formula

Now I will go a little bit in detail about proof theory in propositional logic. In order to infer new facts in logic, we need to apply inference rules. The semantics of the logic will define which inference rules are universally valid. This brings the proof theory in logic. Figure 7 shows an example of the use of such inference rules. Suppose we know that $\varphi \wedge \psi$ and $\varphi \rightarrow \varpi$ are true, then we can infer that ϖ must be true.

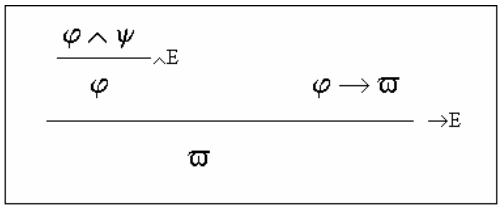

Figure 7: example of a deduction using inference rules

In the above example (figure 7) we made use of two inference rules. The first is the ∧-elimination rule. If we know that φ ∧ ψ is true, then we can say that φ must be true. That is the first deduction we made in the example. The second inference rule we used is called the →-elimination rule. If we know that φ is true and that φ → ϖ holds, then we can conclude that ϖ must be true. Next to these two rules we also have the ∧-insertion, the →-insertion, the ∨-elimination, the ∨-introduction, the ¬¬-elimination and the ¬¬-introduction rules. As it is not the intention of writing here a complete reference of propositional logic, I will not describe all these rules in detail here, but refer the interested reader to Van Benthem, Van Ditmarsch, Ketting and Meyer-Viol.

Now that we have an idea about propositional logic we can continue with an introduction about predicate logic.

1.4.2 Predicate logic

For the purposes in AI, propositional logic is not very useful. To capture our knowledge of the world adequately in a formalism, we need not only to be able to express propositions which are true or false, but we need also to be able to speak of objects, relationships between objects and to generalize the relationships between these objects. The trouble with propositional logic is that it is not possible to write general statements like 'Jack drives every car he owns'. We would need lots of rules, one for every car Jack owns. However, such things are possible in predicate logic. As in the chapter about propositional logic we will look here very short to the syntax, semantics and proof theory of predicate logic.

The predicate calculus is an extension of the notions of the propositional calculus. Instead of looking at sentences that are of interest merely for their truth-value, like in propositional logic, predicate logic is used to represent statements about specific objects or individuals.
"Predicates" are statements about individuals. A predicate is applied on a number of arguments and has a value that is either true or false when individuals are substituted into the arguments. An example of such a predicate is the predicate *is-student*. When applied to Wesley, the predicate turns out to have the value true. Note that predicates can have more than one argument, like for example *is-family of*, which takes two arguments.
Next to predicates we have also "quantifiers". To work with quantifiers, we also need the concept of a variable, which is a placeholder of some constant to be filled in. There are two quantifiers. The first is '∀', which means 'for all …', and it can be used to state that something is true for every object. The second is the quantifier '∃', which means 'there exists …', and it can be used to state that something is true for at least one object. These quantifiers can thus be used to indicate how any variables in the sentence are to be treated. The following are sentences illustrating the use of quantifiers.

(1) $\forall x\,(Man(x) \rightarrow Footballs(x))$

(2) $\exists x\,(Bike(x) \wedge Red(x))$

The first example (1) illustrates the use of the predicate 'for all'. This sentence means that every man is playing football. The second example illustrates the use of the quantifier 'there exists'. This sentence means that there exists at least one thing that is a bike and is red. Note that predicate logic uses symbols for predicates. These symbols are called predicate constants. Every predicate constant has a fixed number of arguments. So we can write 'Man(x)' as 'Mx'. The two previous examples then become as follows:

(1) $\forall x\,(Mx \rightarrow Fx)$
(2) $\exists x\,(Bx \wedge Rx)$

Note that a sentence must have all its variables quantified. This means that each variable must be in the range of some quantifier. Formulas with all their variables quantified are also called 'closed formulae'. Note that it is also possible to use more than one quantifier in one sentence, for example '$\forall x \exists y$ …', which means 'for all x there exists at least one y for which …'.
Next to predicates, we also have examples out of scientific and computational practice of basic assertions that introduce another aspect. For example an algebraic equation like '(x + y) * z = x*z + y*z', is a binary sentence of equality between two objects that can be described in different composed manners. Therefore we also want to represent functions in the language, next to relations and objects. We can also find these functions in natural language like 'Karin's mother' or 'Barry's age'. With al the things we have seen now, we can give the definition of the alphabet of a predicate logic.

Definition

The alphabet of a predicate logic L consists of:

a. a set C of individual constants
b. a set P of predicate constants
c. a set F of function constants
d. logic symbols $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$, $\forall$, $\exists$
e. individual variables
f. help symbols) and (

Now we can go further with the semantics of predicate logic. As in propositional logic, the semantics of predicate logic is defined in terms of the truth-values of sentences. We can use truth tables to find the truth-value of sentences using logical connectives from the truth tables of the smaller parts of the sentence. This is just like in propositional logic. However, now we have to deal also with predicates, arguments and quantifiers. What we need is an interpretation layer that gives meaning to the parts of the language. To define an interpretation function we first have to define what is meant by a structure.

Definition

A structure D is a three-tuple <D,R,O> that exists of a non-empty set D (the domain), a set R of relations on the domain and a set O of operations on the domain.

Definition

Take D=<D,R,O> to be a structure. An interpretation function I assigns to each individual constant c of the predicate logic a special object I(c) ∈ O, to each predicate constant P a relation I(P) ∈ R and to each function f an operation I(f) ∈ O.

Lets look at an example. Take D to be the set of natural numbers IN, R = {<} and O = {1, 3, 5, +}, and the interpretation function as follows I(P)='<', I(f)='+', I(a)='1', I(b)='2' and I(c)='5'. Then Pf(a,c)f(b,c) can be interpreted as 6<8. This returns the value true. Note that this is here only to give a notion of the semantics of predicate logic, but there are lots of other theories about the semantics here. I would refer the interested reader to Van Bethem, Van Ditmarsch, Ketting and Meyer (see also references). However, for our purpose we can just assume that a truth-value can be assigned to a sentence such as is-student(Wesley).
Now, the semantics of ∀ can be defined in terms of whether some sentence is true for all objects in the domain. Suppose we have a domain with three students, called Wesley, Ally and Barry, and take L to be defined as the predicate drives-brand. Then the sentence ∀x L(x, Honda) will be true if L(Wesley, Honda), L(Ally, Honda) and L(Barry, Honda) hold in the domain and thus are all true.
The semantics of ∃ can be defined in terms of whether some sentence is true for at least one of the objects in the domain. So, in our previous example, if L(Wesley, Honda) holds, but the others don't drive a Honda, then the sentence will evaluate to true. Only if none of the people in our domain drive a Honda, the sentence will evaluate to false.
Now I will talk a little bit about proof theory in predicate logic. Inference rules in predicate logic are similar to those for propositional logic. To illustrate this, we will look at the →-elimination rule. We would like to be able to conclude mortal(Wesley) out of the facts ∀x (man(x) → mortal(x)) and man(Wesley). To do this we can use the →-elimination rule, but we have to allow that sentences are matched with other sentences. In this example, man(x) can be matched with man(Wesley), having x = Wesley.

1.4.3 Use of logic in AI

Predicate logic provides a powerful way to represent and reason with knowledge. Some things that cannot be easily represented using frames, such as negation, disjunction and quantification, can be easily represented using predicate logic. The available inference rules and proof procedures mean that a much wider range of inferences are possible.
Now that we have a notion about propositional and predicate logic we can look at the use of logic in the field of artificial intelligence. In this section I want to give an overview of some AI systems that use logic to represent knowledge.
The first system I want to mention is STRIPS, the Stanford Research Institute Problem Solver. It was designed to solve the planning problems faced by a robot in rearranging objects and navigating in a cluttered environment. Since the representation of the world must include a large number of facts dealing with the position of the robot and objects, simpler methods used for puzzles and games would not suffice. Therefore they used predicate logic. [Fikes,Nilsson]
Another system is QA3. This is a general–purpose, question-answering system that solved simple problems in a number of domains. The system could solve problems in chemistry, robot movement, puzzles and automatic programming. Here also, the representation mechanism for the knowledge in the system was predicate logic. [Green]
The third and last system I want to talk about is FOL. FOL was a proof checker for first-order logic. Using FOL involved two steps: describing the representation of some

domain in first-order logic and sequentially stating the steps of a proof of a desired conclusion. FOL would check to make sure that all of these steps were valid. [Filman]

1.4.4 Advantages of logic

First of all, logic seems a natural way to express certain notions. The expression of a problem in logic often corresponds to our intuitive understanding of the domain. A second advantage is that logic is precise. There are standard methods of determining the meaning of an expression in a logical formalism. A third advantage is that logic is flexible. Since logic make no commitment to the kinds of processes that will make the deductions, a particular fact can be represented in a single way, without having to consider its possible use. A last thing that's also an advantage is its modularity. Logic assertions can be inserted into a database independently of each other. The knowledge can grow incrementally as new facts are discovered and added.

1.4.5 Disadvantages of logic

The major disadvantage of logic is the separation between representation and processing. The difficulty with most current AI systems lies in the heuristics part of the system. This means in determining how to use the facts stored in the system's data structures, and thus not in deciding how to store them.

1.5 PRODUCTION SYSTEMS

The term 'production system' is used to describe several different systems based on a general, underlying idea of condition-action pairs. These pairs are called 'production rules'.
A production systems consists of three parts: a 'rule base' that is a set of production rules, a buffer-like data structure, which forms a set of facts, named 'context' and an 'interpreter' which controls the application of the rules given the context. Figure 8 shows the architecture of a rule-based or production system.

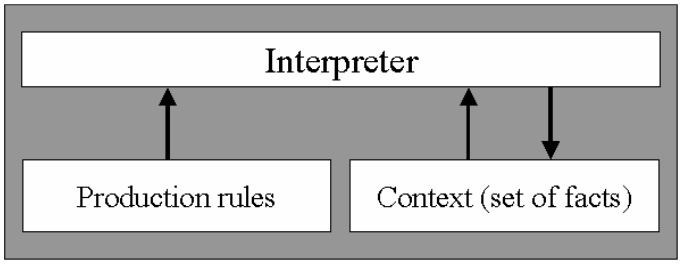

Figure 8: production system architecture

A production rule is also called an IF-THEN rule. The IF part of the production rule is called the condition part. It states the conditions that must be true for the production to be applicable. The THEN part is called the action part. It represents the action to take when the conditions are true. A production whose condition is satisfied can 'fire', this means that it can have its action part executed by the interpreter.
There are two kinds of interpreter: 'forward chaining' and 'backward chaining'. In a forward chaining system there are some initial facts in the context and rules are used to draw new conclusions given those facts. A backward chaining system starts with the goal to prove and keeps looking for rules that would allow concluding the goal. It is possible that sub-goals are set. I will now give a small introduction about these two systems.

1.5.1 Forward chaining systems

The facts in such a system are held in the working memory as they are continually updated by the invoked rules. The actions in the production rules are usually adding or deleting facts in the working memory. However, there are also other actions possible.
The system works as follows: the interpreter looks for all the rules whose conditions hold, given the facts in the working memory. It then selects one of the rules that can be fired. The selection of a rule is based on a particular strategy, called a 'conflict resolution strategy'. Finally, the action of the selected rule is performed and the cycle starts again until no more rules can be fired.
It is important which rule is chosen to fire, as this may influence the final conclusions that are drown. Therefore there are some common conflict resolution strategies to decide which rule to fire. One can prefer to fire the rule with the most specific condition while another can prefer to fire the rule that involves the most recently added facts to working memory. I will now give a small example of a forward chaining system. Suppose we have the following productions in our rule base:

R1: IF January OR February OR March THEN ADD winter
R2: IF April OR May OR June THEN ADD spring
R3: IF July OR August OR September THEN ADD summer

R4: IF October OR November OR December THEN ADD autumn
R5: IF winter THEN ADD high-heating
R6: IF spring THEN ADD low-heating
R7: IF summer THEN ADD no-heating
R8: IF autumn THEN ADD medium-heating
Suppose also we have the following facts in our context:

- January

Then the forward chaining interpreter will start looking at the known facts. Here it is simple as there is only one fact. Now, the interpreter looks in the rule base for rules that can be fired, that is, rules for which the conditions are satisfied by the known facts. Here there is one such rule, namely rule R1. There is no need for conflict resolution here as there is only one applicable rule. The action of R1 is performed, so now we will have the following facts in our context:

- January
- winter

Now by adding the new fact there is a new rule that becomes applicable, namely R5. But R1 stays also applicable so that conflict resolution is needed here. We will give the preference here to the rule that involves the most recently added facts. The interpreter will now perform the action of R5, so we will have the following facts in our context:

- January
- winter
- high-heating

Now, given the facts in the context, there are no rules anymore that are applicable that can lead to new facts in the context, so the systems stops and we can conclude that the heating must be high.

1.5.2 Backward chaining systems

A backward chaining system starts with the goal to prove. The system will then check if there are any facts in the working memory that matches the goal. If there are such facts, then the goal is proven. Otherwise, the system will look for rules with conclusions that can match the goal. One of the applicable rules is then chosen and the conditions of this rule are set as sub-goals to prove. The cycle can then start again to prove the sub-goals. Note that a backward chaining system must keep track of the goals that it needs to prove in order to satisfy the main hypothesis. Therefore, such a system can be implemented using a stack to keep track of the goals that still must be satisfied. I will now give a small example of a backward chaining system. Suppose we have the same rules as we had in the example of a forward chaining system:

R1: IF January OR February OR March THEN ADD winter
R2: IF April OR May OR June THEN ADD spring
R3: IF July OR August OR September THEN ADD summer
R4: IF October OR November OR December THEN ADD autumn
R5: IF winter THEN ADD high-heating
R6: IF spring THEN ADD low-heating
R7: IF summer THEN ADD no-heating
R8: IF autumn THEN ADD medium-heating

Now, if our goal to prove is that there must be no heating and we have in our context the fact ‘July’ then the system can start searching the rules in which the action can satisfy the required goal. This leads to rule R7 in which the action adds our goal to the context. The system will now set up the sub-goal to prove, namely the fact ‘summer’. Now, the system will again search in the rule set for rules that can satisfy our sub-goal. This will give rule R3. The system will see that the condition of this rule is satisfied by the initial fact in our context. So the system has proven that there must be no heating.

1.5.3 When to use which system

The approach of forward chaining is most useful when all the initial facts are known, but the conclusions or goals to be satisfied are not known. If it is known what the goal to satisfy is then the forward chaining approach can be inefficient. The forward system is likely to do a lot of irrelevant work in this case, adding uninteresting conclusions to the context. If we have some particular hypothesis to test, the backward chaining approach can be more efficient, as it avoids drawing conclusions from irrelevant facts. Sometimes backward chaining systems can also be wasteful as there can be many possible goals and for each one there can be many ways to try to prove it, few of which result in a conclusion.
So the choice of the approach depends on the properties of the rule set and initial facts and how many possible different hypothesis there are.

1.5.4 Advantages of production systems

First of all, a good quality of production systems is that the individual productions in the rule base can be added, deleted or changed independently. They are like independent pieces of knowledge. Changing one rule can be accomplished without having to worry about direct effects on the other rules, since rules only communicate with each other by means of the context.
Another advantage of production systems is the uniform structure of the knowledge in the rule base. All information must be encoded within the structure of production rules, so it can be more easily understood by another person or even by another part of the system itself.
A third advantage is its naturalness, namely the ease with which we can express important kinds of knowledge. Statements about what to do in predetermined situations are naturally encoded in production rules.

1.5.5 Disadvantages of production systems

One of the disadvantages of production systems is inefficiency of program execution.
A second disadvantage is that it is hard to follow the flow of control in problem solving. Although situation-action knowledge can be expressed naturally, algorithmic knowledge is not expressed in this way. The two factors responsible for this problem are the isolation of productions; this means that productions don’t call each other, and the uniform size of the productions. There is nothing like a subroutine hierarchy in which one production can be composed of several sub-productions.

1.6 SUMMARY

We have seen the most important knowledge representation systems used in AI. These are semantic nets, frames and scripts, predicate logic and rule-based systems or production systems. In this summary I will give a comparison between these systems.
Frames and networks are useful for representing declarative information about collections of related objects and concepts. They are very useful in particular where there is a clear class hierarchy, and where you want to use inheritance to infer the attributes of objects in subclasses from the attributes of objects in the parent class. However, frames and networks are unlikely to be used when you want to draw a wide range of different sorts of inferences, and especially not just inferences based on inheritance. When we want to do this, it is better to use the full power of predicate logic, along with a theorem prover. Logic-based approaches allow us to represent fairly complex things and have a well-defined syntax, semantics and proof theory. However, logic based approaches may be inflexible. Any inference rule in the logic must be validated by a precisely stated semantics for expressions in the language. This may be the reason why many commonsense inferences and conclusions aren't allowed and so reasoning may be inefficient. Rule-based systems provide more flexibility, sometimes allowing arbitrary procedures within rules. More emphasis may be placed on how the reasoning is controlled, but the semantics of the rules may be unspecified. The conclusions drawn by the system just depend on how the rule interpreter works.

Now, as I stated before in the disadvantages of logic, the major disadvantage of most current AI systems is the separation between representation and processing. This is the reason why I started looking at the techniques used in the semantic web. Next chapter will give a broad overview of the features in the semantic web.

2 THE SEMANTIC WEB

"The Semantic Web is an extension of the current web in which information is given a well-defined meaning, better enabling computers and people to work in cooperation."

T. Berners-Lee, J. Handler & O. Lassila

2.1 INTRODUCTION

The Web can only reach it's full potential if it becomes a place where data can be shared and processed by automated tools as well as by people. The Semantic Web is a vision: the idea of having data on the Web defined and linked in a way that it can be used by machines not just for display purposes, but for automation, integration and reuse of data across various applications.
The Semantic Web has the intention to bring structure to the meaningful content of Web pages, by creating an environment where software agents roaming from page to page can readily carry out sophisticated tasks for users. The Semantic Web will not be a separate web, but an extension of the current one, in which information will be given a well-defined semantics. The semantics will be encoded into web pages. Like the Internet, the Semantic Web as decentralized as possible.

For the Semantic Web to function, computers must have access to structured collections of information and sets of inference rules that they can use to conduct automated reasoning. As we have seen in the previous chapter, artificial-intelligence researchers have studied such systems since long before the Web was developed. Tim Berners Lee[1] states that these are clearly good ideas and that some very nice examples exist, but that it hasn't changed the world yet. To reach its full potential, it must be linked into a single global system.

Traditional knowledge representation languages like we have seen before are typically centralized. It requires everybody to share exactly the same definition of common concept. Increasing the size and scope of such a centralized system rapidly becomes unmanageable. These systems usually carefully limit the question that can be asked (see inference in semantic networks), so that the computer can answer reliably. Therefore, traditional knowledge representation languages have their own narrow set of rules for making inferences about their data.

On the other hand, Semantic Web researchers accept that paradoxes and unanswerable questions are the price to pay to achieve versatility. This is what Gödel's theorem from mathematics says: "Any system that is complex enough to be useful also encompasses unanswerable questions." The challenge of the Semantic Web is to build a language that expresses both data and rules for reasoning about the data and that allows rules from any existing knowledge representation formalism to be exported onto the Web.

Two important technologies for developing the Semantic Web already exist. These are the eXtensible Markup Language (XML) and the Resource Description Framework (RDF). Before going further, I will give an overview of these technologies.

[1] Tim Berners Lee is known as the father of the internet in the form it exists nowadays.

2.2 XML

2.2.1 Why XML?

XML stands for the Extensible Markup Language. It is a markup language developed by the World Wide Web Consortium, mainly to overcome the limitations in HTML.
HTML is an immensely popular markup language. It is supported by thousands of applications including browsers, editors, email, databases and more.
Originally the web was a solution to publish scientific documents. Today, the web is an interactive medium because it supports applications such as online shopping, electronic banking and trading and forums. Therefore, HTML has been extended over the years. Many new tags have been added. For example, the first version of HTML contained a dozen tags while the latest version has close to hundred tags. Furthermore, a large set of supporting technologies has been developed, like JavaScript, Java, Flash, CGI, ASP, streaming media, MP3, …
However, it is not all rosy with HTML. It has grown into a complex language. The combinations of tags are almost endless and the result of a particular combination of tags might be different from one browser to another. The biggest problem is that besides all existing tags, more tags are needed. Electronic commerce applications need tags for product references, prices, name, addresses and lots more. Search engines need more precise tags for keywords and description, security needs tags for signing and so on. However, adding more tags to an overblown language is hardly a satisfactory solution. Although many applications need more tags, some applications would greatly benefit if there were less tags in HTML. An example of such an application is accessing the web from a personal digital assistant. These machines are not as powerful as PC's and thus cannot process a complex language like HTML. Another problem is that sometimes you need many tags to format a page. Some pages contain more markup than content. Such pages are slow to download and to display.
However, XML is unlikely to replace HTML in the near or medium future. Some of the areas where XML will be useful in the near-term are: large website maintenance by using XML behind the scene to simplify the creation of HTML documents, exchange of information between organizations, electronic commerce applications where different organizations collaborate to serve a customer and lots more. Because HTML is successful, XML incorporates many successful features of HTML. However, XML breaks new ground where it is necessary.
In the following section I will give a short introduction to XML.

2.2.2 The syntax

As we can see in figure 9, an XML document is textual in nature. The document consists of 'character data' and 'markup'. Both things are represented by text. We can recognize the markup because it is enclosed in angle brackets. The markup is important because it records the structure of the document. Note that the text without the markup would be more readable. However, for a machine it is exactly the opposite. We need to tell software what the name is, what the address is, and so on. The markup breaks the text into its constituents so software can process it.

```
<?xml version="1.0"?>
<!- - a simple example - ->
<adress-book>
  <individual>
    <name>Wesley Bille</name>
    <address>
      <street>65 VR Square</street>
      <postal-code>46758</postal-code>
      <city>Toronto</city>
      <country>United States</country>
    </address>
    <email href="mailto:wbille@vub.ac.be"/>
  </individual>
  <individual>
    <name>
      <first-name>Brad</first-name>
      <last-name>Smith</last-name>
    </name>
    <email href="mailto:brad.smith@nonsense.com"/>
  </individual>
</address-book>
```

Figure 9: a simple XML document

2.2.2.1 XML Declaration

The XML declaration is the first line of the document. The declaration identifies the document as an XML document. This declaration also states the version of XML being used in the document. It can contain other attributes to support features such as character set encoding.

However, the XML declaration is optional. If the declaration is included, then it must start on the first character on the first line of the XML document. The XML recommendation suggests including the declaration in all XML documents.

2.2.2.2 XML Elements

The standard building block of XML is 'element'. Each element has a name and content. The content of an element is delimited by special markups called the 'start tag' and the 'end tag'. The start tag is the name of the element in angle brackets while the end tag is the name preceded by a slash character also in angle brackets. Unlike in HTML, both start and end tags are required. An example of such an element is the street element:
<street>65 VR Square</street>

Element names must follow certain rules. Names in XML must start with either a letter or the underscore character. The rest of the name must consist of letters, digits, the underscore character, the dot or a hyphen. However, spaces are not allowed in names. Finally, names cannot start with the string 'xml' because it is reserved for the XML specification itself. Note that by convention, XML elements are frequently written in lowercase and when a name consists of several words, the words are normally separated by a hyphen.

Elements that have no content are known as ‘empty elements’. There is a shorthand notation for such elements, the start tag and end tag merge and the slash from the end tag is added at the end of the opening tag. A simple example is the following:

<email href=”mailto:wbille@vub.ac.be”></email>

Which is identical to the following element in shorthand notation:

<email href=”mailto:wbille@vub.ac.be”/>

2.2.2.3 Nesting of XML elements

Element content is not limited to text. Elements can contain other elements that in turn can contain text or elements and so on. Such elements are called complex elements. An XML document can be seen as a tree of elements. There is no limit to the depth of the tree and elements can be repeated. You can see in figure 9 that there are for example two *individual* elements in the element *address-book*. We name an element that is enclosed within another element a ‘child’. The element a child is enclosed in is named the ‘parent’. Start and end tags must always be perfectly balanced and children are always completely enclosed in their parents. So it is not correct if the end tag of a parent precedes the end tag of one of its children.

At the root of a document there must be one element. So all elements in a document must be children of a single element. In figure 9, the *address-book* element defines the root element.

2.2.2.4 XML Attributes

It is possible to add additional information to elements in the form of ‘attributes’. Attributes have a name and a value. Elements can have one or more attributes in the start tag, and the name is separated from the value using the equal character. The value of an attribute is enclosed in double or single quotation marks. An example can be an attribute that states whether a telephone number is private or public. The following example shows a telephone number that is known to be private.

<tel private=”true”>02-655-33-35</tel>

Note that XML insists on the quotation marks. The XML processor would reject an element if there are one or more quotation marks missing.

2.2.2.5 XML Namespaces

A single XML document may contain elements and attributes that are defined for and used by multiple software modules. If such a markup vocabulary exists which is well-understood and for which there is useful software available, it is probably better to reuse this markup rather than to reinvent it.
However, documents containing multiple markup vocabulary can pose problems of recognition and collision. Software modules must be able to recognize the tags and attributes which they have to process, even if collision occur when markup intended for some other software package uses the same element type or attribute name.

This requires that we should have some mechanism to accomplish the fact that document constructs should have universal names, who's scope extends beyond their containing document.
First of all, I will give the definition of a name space defined by the World Wide Web consortium.

Definition

*An **XML namespace** is a collection of names, identified by a URI reference, which are used in XML documents as element types and attribute name.*

I will now give a small example of how to use namespaces. Suppose we have an XML document that has the URI http://wise.vub.ac.be/WofSy/example.xml that describes an element 'student'. If we now want to construct a new XML document that uses this element, we will have to construct a namespace to identify this element.
The declaration associates the URI with a prefix. Lets take here the prefix 'ex'. Namespace declaration is done through attributes with the prefix xmlns followed by the prefix we want to associate with the given URI. Figure 10 shows an example of a XML document using namespaces.

```
<?xml version="1.0"?>
<!-- a namespace example -->
<adress-book xmlns:ex="http://wise.vub.ac.be/WofSy/example.xml">
  <ex:student>
    <name>Wesley Bille</name>
    <address>
      <street>65 VR Square</street>
      <postal-code>46758</postal-code>
      <city>Toronto</city>
      <country>United States</coutry>
    </address>
    <email href="mailto:wbille@vub.ac.be"/>
  </ex:student>
  <ex:student>
    <name>
      <first-name>Brad</first-name>
      <last-name>Smith</last-name>
    </name>
    <email href="mailto:brad.smith@nonsense.com"/>
  </ex:student>
</addrss-book>
```

Figure 10: XML document using namespaces

In summary we can say that namespaces are a mechanism to unambiguously identify who has developed which element. It is a small topic but an essential service.
Note that the namespace name is the URI and not the prefix associated with it. When an XML application compares two elements, it uses the URI, not the prefix.

Now we have an overview of what XML is, we can go one step further. In the next section we will look at what XML Schema is used for.

2.3 XML SCHEMA

XML Schemas express shared vocabularies and allow machines to carry out rules made by people. They provide means for defining the structure, content and semantics of XML documents.
XML Schema is an XML based alternative to DTD, which stands for Document Type Definition. However, XML Schemas are the successor of DTD's. As DTD's do not fall in the scope of this document, I would refer the interested reader to the World Wide Web Consortium website.
What exactly is an XML Schema? The purpose of an XML Schema is to define the legal building blocks of an XML document. XML Schema was originally proposed by Microsoft, but has become now an official W3C recommendation since May 2001.
Before looking at how we have to use XML Schema's there rests one question to answer: Why would XML Schema be better than it's ancestor DTD?
Experts mention several reasons here. First of all, XML Schema has support for data types, which makes it for example easier to work with data from a database. A second reason is that XML Schema uses XML syntax, so that we don't have to learn another language. A third argument is that XML Schemas are extensible. This makes it for example possible to use our Schema in another Schema.

2.3.1 How to use XML Schemas

Let's take a new simple XML document like in figure 11.

```
<?xml version="1.0"?>
<email>
  <from>wbille@vub.ac.be</from>
  <to>dsmith@vub.ac.be</to>
  <subject>Party tomorrow</subject>
  <body>Hello! Don't forget to come
        to the party tomorrow evening!
        See you there!
  </body>
</email>
```

Figure 11: another simple XML example

In the above example, the 'email' element is a complex element as it contains other elements. All the other elements are simple elements. Now, figure 12 represents a simple XML Schema that defines the elements of the above example.

```
<?xml version="1.0" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:element name="email">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="from" type="xs:string"/>
      <xs:element name="to" type="xs:string"/>
      <xs:element name="subject" type="xs:string"/>
      <xs:element name="body" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

</xs:schema>
```

Figure 12: an XML Schema example

The schema in figure 12 defines the element 'email' to be a complex element containing a sequence of other simple elements. The syntax for defining simple elements is as follows:

<xs: element name="name" type="type"/>

We can define complex elements using the following syntax:

```
<xs:element name="name">
  <xs:complexType>
        …
        … Element content
        …
  </xs:complexType>
</xs:element>
```

Note that the type used in figure 12 for the simple elements is 'string'. However, there are a lot of other XML Schema data types like boolean, decimal, float, double, time, date, … and so on.
Supposing that the above schema is named *email.xsd* we can rewrite the XML document of figure 3. The new version is shown in figure 13.

```
<?xml version="1.0" ?>

<email xmlns:xsi=
        "http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation=
        "http://wise.vub.ac.be/WofSy/email.xsd">

  <from>wbille@vub.ac.be</from>
  <to>dsmith@vub.ac.be</to>
  <subject>Party tomorrow</subject>
  <body>Hello! Don't forget to come
        to the party tomorrow evening!
        See you there!
  </body>
</email>
```

Figure 13: XML document using an XML Schema

Now to conclude this section I want to answer the question of why to use XML Schemas. The majority of XML documents are well formed rather than valid. 'Well-formed' means that the document follows the XML syntax exactly. A valid document is well formed and conforms to a specified set of element rules. To validate an XML document, some form of validation rules need to be provided. Providing an XML schema can do this.

2.4 THE RESOURCE DESCRIPTION FRAMEWORK

2.4.1 Introduction

It is very hard to automate anything on the web, and because of the volume of information it contains, it is not possible to manage it manually. The proposed solution here is to use metadata, that is, data about data, to describe the data contained on the Web.
The Resource Description Framework (RDF) is a W3C recommendation designed to standardize the definition and use of metadata.
Now, someone may raise the question: "Why using RDF instead of XML?". XML allows us to invent tags that may contain both text data and other tags as we have seen in the previous section. XML has a build-in distinction between 'element types' like the element type 'IMG' in HTML and 'elements' like <img source="example.jpg">. This corresponds naturally to the distinction between properties and statements. So it seems as we can use XML documents as a natural vehicle for exchanging general-purpose metadata. However, using XML raises a problem. The order in which elements appear in an XML document is significant and often very meaningful. This is highly unnatural in the world of metadata. Data oriented applications are less concerned about serial order. Therefore it is better to use RDF.
Now I will briefly introduce what's RDF all about.

2.4.2 The Basic RDF Model

The foundation of RDF is a model for representing named properties and property values. RDF properties may be thought of as attributes of resources and in this sense correspond to traditional attribute-value pairs. RDF properties also represent relationships between resources. Using the terms in object oriented programming, we can say that resources correspond to objects and properties correspond to instance variables of objects.

The basic data model in RDF consists of three object types:

1. Resources

All things being described by RDF expressions are resources. A resource can be an entire Web page, such as an HTML document 'http://wise.vub.ac.be/WofSy', but it may also be a part of a Web page, for example a specific XML element within a document. A resource can also be a whole collection of pages like an entire Web site. But a resource doesn't need to be something that is accessible via the Web, it can also be an object like a printed book. Resources are always named by using Uri's, because the extensibility of Uri's allows the introduction of identifiers for any possible entity.

2. Properties

A property is a characteristic, attribute or relation used to describe a resource. Each property has a specific meaning, defines its permitted values, the types of resources it can describe and its relationship with other properties. You will find out how to express the characteristics of properties in the next chapter about RDF Schema.

3. Statements

A specific resource together with a named property and the value of that property for that specific resource is called an 'RDF statement'. The three parts of such a statement are called respectively 'the subject', 'the predicate' and 'the object'. Note that the object of a statement can be another resource or a literal; this is a string or other primitive data type defined by XML.

I will now give some small examples to get used with the terms previously mentioned. Consider the following sentence: "Wesley is the owner of the resource http://wise.vub.ac.be/WofSy." This sentence has the following parts: the subject or resource is http://wise.vub.ac.be/WofSy, while the object is "Wesley". The predicate or property is Owner.
An RDF statement can graphically be represented by a direct-labeled graph. We use nodes to represent resources; arcs to represent named properties and rectangles will be used to represent string literals. So the representation of the above example is like in figure 14.

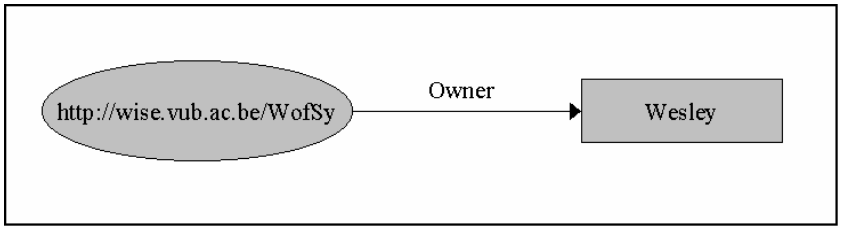

Figure 14: a simple RDF statement

Note in figure 14 that the direction of the arrow is important. The arc always goes from the subject to the object of the statement. We can then read the above diagram as "http://wise.vub.ac.be/WofSy is owned by Wesley".
Now we will consider a more specific example. Suppose we want to represent "The resource http://wise.vub.ac.be/WofSy has creator the individual referred to by student rolnumber 60968 is named Wesley and follows the program Applied Computer Sciences" in RDF, than we can do this with the diagram represented in figure 15.

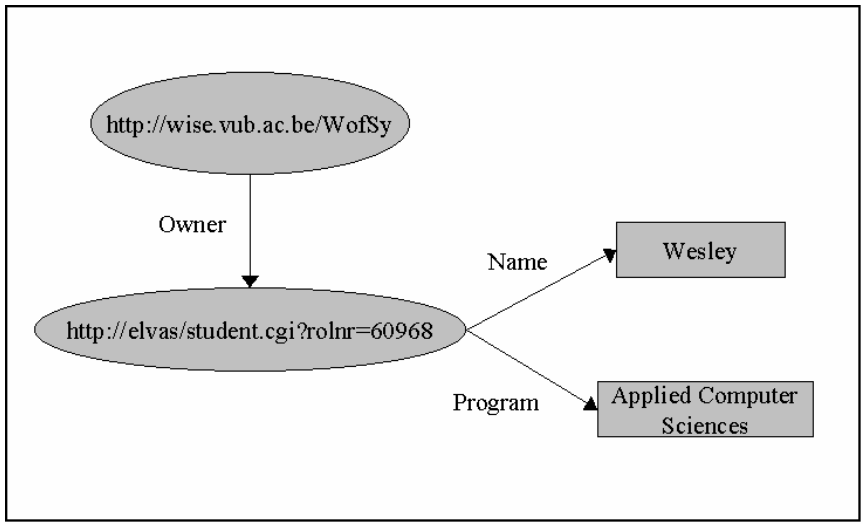

Figure 15: another RDF example

2.4.3 The Basic RDF Syntax

The RDF data model provides an abstract, conceptual framework for defining and using metadata. There is also needed a concrete syntax to create and exchange this metadata. This specification of RDF uses XML as its interchange syntax. RDF also requires the XML namespace facility to associate each property with the schema that defines the property.
I will not give here the complete overview of the RDF syntax but just demonstrate how to write the above examples using the RDF syntax. The interested reader can look at the Resource Description Framework Model and Syntax Specification to have an overview of the complete syntax expressed in Backus-Naur Form.

The first example sentence from figure 6 can be represented in RDF/XML as follow:

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:sch="http://wise.vub.ac.be/WofSy/Schema/">
    <rdf:Description about="http://wise.vub.ac.be/WofSy">
        <sch:Owner>Wesley</sch:Owner>
    </rdf:Description>
</rdf:RDF>
```

We use the 'RDF element' to mark the boundaries in an XML document what part of the content is intended to be mapped into an RDF data model instance. Next we see there are some namespaces declared. These are needed to identify where for example the property Owner is defined. As the Owner property is preceded by the prefix 'sch', we know that this property is defined in the namespace referred to as 'sch'. This is similar to what we have seen in XML and XML Schema.
Next we have the 'Description' element. The Description element can be though of as a place to hold the identification of the resource being described. When using the 'about attribute' in the Description element, the statements in the Description refer to the resource whose identifier is determined in the about.
The rest of the elements used in the previous example are properties that are used within a Description. The properties used here are all properties for the resource declared in the Description element.

Now I will show how to express the second example from figure 15 in RDF/XML. Note that I left the namespace declaration away and just described the Description element for the RDF statement of interest.

```
<rdf:RDF>
    <rdf:Description about="http://wise.vub.ac.be/WofSy">
        <sch:Owner rdf:resource="http://elvas/student.cgi?rolnr=60968"/>
    </rdf:Description>

    <rdf:Description about="http://elvas/student.cgi?rolnr=60968">
        <sch:Name>Wesley</sch:name>
        <sch:Program>Applied Computer Sciences</sch:Program>
    </rdf:Description>
</rdf:RDF>
```

Note that this is not the only notation we can use. There is also an abbreviation syntax we can use. I recommend the interested reader to look at the RDF Model and Syntax Specification document for an overview of this.
As I mentioned before, it is frequently necessary to refer to a collection of resources, for example for describing the list of students in a course, or software modules in a package. Therefore we can use RDF containers to hold such lists. There are three types of containers used to hold such a list of resources or literals. However, I will only discuss one here as an example of RDF containers. I will give an example of the list of people working at the research group http://wise.vub.ac.be. Therefore we can use the RDF container 'Bag'. A bag is an unordered list of resources or literals. Now suppose we want to represent the following sentence in RDF: "Prof. De Troyer, Sven Casteleyn and Peter Stuer are working on http://wise.vub.ac.be" This can be written in RDF/XML as follows:

```
<rdf:RDF>
  <rdf:Description about="http://wise.vub.ac.be">
    <sch:researchers>
      <rdf:Bag>
        <rdf:li resource="http://wise.vub.ac.be/Members/Olga/default.html"/>
        <rdf:li resource="http://wise.vub.ac.be/Members/Sven/default.html"/>
        <rdf:li resource="http://wise.vub.ac.be/Members/Peter/default.html"/>
      </rdf:Bag>
    </sch:researchers>
  </rdf:Description>
</rdf:RDF>
```

I hope this gives the reader of this thesis a good overview of what RDF is and how it can be used. However, these are only the main topics of RDF as the complete details of RFD go way beyond the scope of this thesis. Like we had XML Schema associated with XML, we will see RDF schema in the next section.

2.5 RDF SCHEMA

In the previous section we used some properties in the RDF statements shown in the examples. In the first example, we used the Owner property to connect a website to some person. I have chosen this name myself and I use it in all of my metadata. Someone else might have chosen another name to denote the same property, like the Dutch equivalent 'Eigenaar'.
For the metadata to be exchangeable, we need to define a common name for it. RDF Schema is meant to give such definitions. With RDF Schema, we can create a schema that gives us a language to use in our RDF data.
In such a schema we can define specific classes of resources and subclasses of these classes, we can define the properties that we need for the particular domain we are working with and we can add domain- and range-constraints on properties.

The RDF Schema Specification consists of some basic classes and properties and can be extended by others to fit in nearly every given domain. Classes are arranged hierarchically and the use of properties can be constrained to certain classes. The root of the class hierarchy is rdfs:Resource. The rdfs:Class is a subclass of rdfs:Resource. Properties are defined by the rdf:Property class and can be seen as attributes that are used to describe resources by assigning values to them. The RDF Schema Specification defines four specific properties, which are rdfs:subClassOf, rdf:type, rdfs:range and rdfs:domain. These four properties are used to define other RDF schema constructs. Figure 16 shows the class hierarchy for the RDF Schema taken from the World Wide Web Consortium.

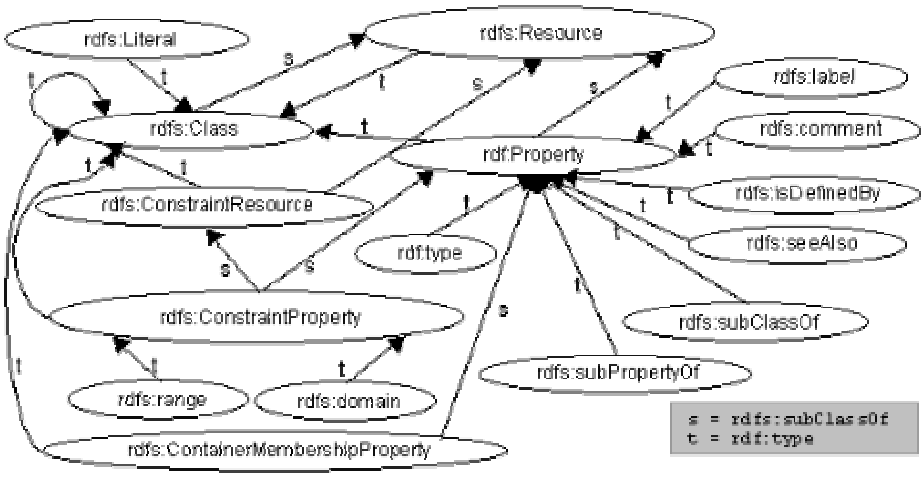

Figure 16: Class hierarchy for the RDF Schema

As explaining the whole RDF Schema Specification goes way beyond the scope of this thesis, I will conclude this section by giving a small example of the usage of RDF Schema.

I will not go into detail in this example, but just transform a real world situation into some RDF Schema. Suppose we want to define a class person that is a subclass of the class animals already defined. We also want that a person has a maritalStatus that can be

either Married, Divorced, Single or Widowed. Next to that every person has an age. The RDF Schema shown in figure 17 does this.

```
<rdf:RDF xml:lang="eng"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">

<rdfs:Class rdf:ID="Person">
    <rdfs:subClassOf rdf:resource="http://www.w3.org/2000/03/example/classes#animal"/>
</rdfs:Class>

<rdf:Property ID="maritalStatus">
    <rdfs:range rdf:resource="#MaritalStatus">
    <rdfs:domain rdf:resource="#Person">
</rdf:Property>

<MaritalStatus rdf:ID="Married">
<MaritalStatus rdf:ID="Divorced">
<MaritalStatus rdf:ID="Single">
<MaritalStatus rdf:ID="Widowed">

<rdf:Property ID="age">
    <rdfs:range rdf:resource="http://www.w3.org/2000/03/example/classes#integer"/>
    <rdfs:domain rdf:resource="#Person">
</rdf:Property>

</rdf:RDF>
```

Figure 17: small example of RDF Schema

This section has given a quick introduction to RDF Schema and what it can be used for. In the next chapter we are going to talk about ontologies, which are more or less an extension to RDF Schema.

2.6 ONTOLOGIES

Of course the story of the semantic web does not end with XML (Schema) and RDF (Schema). Suppose we have two databases that use different identifiers for in fact the same concept. Lets say they use couchType and sofaType, which are exactly the same concepts. A program that wants to combine information across the two databases has to know that these two terms are being used to mean the same thing.
A solution to this problem is provided by the third basic component of the Semantic Web, namely collections of information called ontologies. An ontology is a document or file that formally defines the relations among terms.
With ontology pages, solutions to terminology problems begin to emerge. Pointers from the page to the ontology can define the meaning of XML codes used on a Web page. Of course, the same problems as before now arise if we point to an ontology that defines addresses as containing a 'zip code' and someone else points to one that uses 'postal code'. This kind of confusion can be resolved if ontologies provide equivalence relations. Then we can say that our concept 'zip code' is equivalent with the concept 'postal code'.
In it's most simple form, we can say that an ontology is an abstraction of a computer-based lexicon, thesaurus, glossary or some other type of structured vocabulary, suitably extended with knowledge about a given domain.

At this time there are some different ontology languages, some of which are seeming to be new, like DAML+OIL, other are descendents from conceptual schema languages of the 80s, like ORM or EXPRESS. Others are continuations of important and famous 'old' developments from artificial intelligence, like CYCL and KIF.

To give an idea of what an ontology language is, I will now give an introduction in DAML+OIL. As it will turn out to be the knowledge representation formalism to use in this project, I will go in detail in some of the topics.

2.7 DAML+OIL

DAML+OIL is a semantic markup language for Web resources. The DAML language is being developed as an extension to XML and RDF. The latest release of the language provides a rich set of constructs with which to create ontologies and to markup information so that it is machine readable and understandable. The language has a clean and well-defined semantics.
The DAML (DARPA Agent Markup Language) program formally began with a kick-off meeting in August 2000 in Boston. An integration contractor and sixteen technology development teams are still working to realize the DAML vision. Tools, data and other results of their efforts become available.
On the other hand, OIL [www-16] is a language built on a long history of research in description logics. Description Logic is a sub-field of knowledge representation and as such aims to provide a vehicle for expressing structured information and for reasoning with the information in a principled manner. OIL, which stands for Ontology Inference Layer, is an effort to produce a well defined language for integrating ontologies with web standards, in particular XML, XML Schema, RDF and RDF Schema. It is a web based representation and inference layer for ontologies using the constructs found in many frame languages and reasoning and formal semantics in description logics.
In December 2000, DAML and OIL were brought together as the DAML+OIL language specification. The latest version dates back to March 2001 and is the version on which I grounded my research. In this chapter I will give a systematic, compact and informal description of all the modeling primitives the language contains.

A DAML+OIL knowledge base is a collection of RDF triples, that is, a resource, a property and a literal. The relation between those three things is as follows: the "*literal*" has "*property*" "*resource*". As an example of this take the following: lets say that the resource is 'http://wise.vub.ac.be/WofSy", the property is 'creator' and the literal is 'Wesley', then we can say: "Wesley is the creator of the resource http://wise.vub.ac.be/WofSy. The following paragraphs will specify which collections of RDF triples constitute the DAML+OIL vocabulary and what the meaning of those triples is. Note that DAML+OIL triples can be represented in many different forms as with any set of RDF triples. Here we will use a specific RDF syntactic form for these triples.

A DAML+OIL ontology is made up of several components. Some of these components are optional, others may be repeated. A DAML+OIL ontology contains zero or more headers, followed by zero or more class elements, property elements and instances.

2.7.1 Headers

We will begin with the header. A **daml:Ontology** element contains zero or more version information and import elements. The **daml:versionInfo** element generally contains a string giving information about this version. This can for example be CVS keywords. Note that the version information element does not contribute to the logical meaning of the ontology. Each **daml:imports** element references another DAML+OIL ontology containing definitions that apply to the current DAML+OIL resource. Each reference contains a URI specifying where the ontology to import can be found. Note that it is also possible in the beginning of a DAML+OIL document to specify some namespaces like you have in RDF. Namespaces only provide a mechanism for creating unique names for elements and do not actually include definitions the way that imports does. On the other hand, imports do not set up shorthand notations for names. It is therefore common to

have import statements that correspond to each namespace. The following example is an example of a namespace specification:

```
<rdf:RDF xmlns:daml=”http://www.daml.org/2001/03/daml+oil#”>
```

The example on the previous page states that daml is the abbreviation for the total string http://www.daml.org/2001/03/daml+oil#. The following example is an example of a header.

```
<daml:Ontology rdf:about=””>
        <daml:versionInfo>
            $Id: test.html, v 1.0 2002/01/12 21:45:33 mdean Exp $
        </daml:versionInfo>
        <rdfs:comment>An example ontology</rdfs:comment>
        <daml:imports rdf:resource=”http://www.daml.org/2001/03/daml+oil”/>
</daml:Ontology>
```

2.7.2 Defining classes

Now we can start creating our ontology definitions. If we want to describe some objects, it will be useful to define some basic types, which we call here classes. Suppose we are working in the domain of the university, we will want to define a kind of thing called student. We will say that its id identifier is Student, to make it possible for others to refer to the definition of Student we are giving. Suppose our ontology is stored on http://wise.vub.ac.be/WofSy/example.daml, then somebody can refer to our definition of Student with the name http://wise.vub.ac.be/WofSy/example.daml#Student. The following example demonstrates the definition of the class Student.

```
<daml:Class rdf:ID=”Student”>
        <rdfs:comment> This represents the class of all students </rdfs:comment>
</daml:Class>
```

We can also define subclasses of a class. This is done by the element subClassOf. Suppose we have to types of students, we will call them DayStudents and EveningStudents. We can define those classes of students as follows:

```
<daml:Class rdf:ID=”DayStudent”>
        <rdfs:subClassOf rdf:resource=”#Student”/>
</daml:Class>

<daml:Class rdf:ID=”EveningStudent”>
        <rdfs:subClassOf rdf:resource=”#Student”/>
</daml:Class>
```

However, there is more to use than a subClassOf relation. We can also use daml:disjointWith, daml:disjointUnionOf, daml:sameClassAs, daml:equivalentTo relations.

2.7.3 Defining properties

Next, we are going to define some properties. A **rdf:Property** element refers to a property. DAML+OIL properties are generally divided into two sorts, those that relate objects to other objects, which are instances of **daml:ObjectProperty**, and those that relate objects to datatype values (see later), which are instances of **daml:DatatypeProperty**. The following example demonstrates an ObjectProperty that connect two students as being in the same year.

```
<daml:ObjectProperty rdf:ID=”hasClassMate”>
        <rdfs:domain rdf:resource=”#Student”>
        <rdfs:range rdf:resource=”#Student”>
</daml:ObjectProperty>
```

The **rdfs:domain** element states that we are defining *hasClassMate* as a property that applies to students. Similar to the domain, we also declare the range with the **rdfs:range** element. This means that we are defining *hasClassMate* as a property whose value can only be Students. Domains and ranges are global information about properties. So they should be used with care. Putting a too-restrictive domain or range on a property can cause all sorts of trouble.

The next example shows a datatype property. Let’s provide an age property. We will reference the XML Schema non-negative integers for this by referring to it’s standard location. Note that this is a unique property, it provides at most one single age, which is a non-negative integer.

```
<daml:DatatypeProperty rdf:ID=”age”>
        <rdf:type
rdf:resource=”http://www.daml.org/2001/03/daml+oil#UniqueProperty”/>
        <rdfs:range

rdf:resource=”http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger”/>
</daml:DatatypeProperty>
```

Note that we can do a lot more with properties. As with classes, we can define subproperties using the **rdfs:subPropertyOf** element. Further we can use daml:samePropertyAs, equivalentTo and daml:inverseOf relations. We can also use any of the following elements: daml:TransitiveProperty, daml:UniqueProperty or daml:UnambigousProperty. Each of these asserts additional information about the property.

2.7.4 Defining property restrictions

A property restriction is a special kind of class expression. It implicitly defines an anonymous class, namely the class of all objects that satisfy the restriction. A daml:Restriction elements contains a daml:onProperty element which refers to a property name and one or more of the following: daml:toClass, daml:hasValue, daml:hasClass, daml:Cardinality, daml:minCardinality and daml:maxCardinality. There are also others, but as this is just an introduction to DAML+OIL we will not discuss them. Now, we will redefine the Student class, requiring that a student must have exactly one age.

```
<daml:Class rdf:ID=”Student”>
        <rdfs:subClassOf>
                <daml:Restriction daml:cardinality=”1”>
                        <daml:onProperty rdf:resource=”#age”/>
                </daml:Restriction>
        </rdfs:subClassOf>
</daml:Class>
```

So now we have defined a Student as being a subclass of all objects that have exactly one age.

2.7.5 Using user defined data-types

It is also possible to use user-defined data types in DAML+OIL. Data type values are written in a manner that is valid RDF syntax, but which is given a special semantics in DAML+OIL. Suppose we want to create different categories of ages, then we will do this like in the following example:

```
<xsd:schema xmlns:xsd = ”http//www.w3.org/2000/10/XMLSchema#”>

<xsd:simpleType name=”over11”>
        <xsd:restriction base=”xsd:positiveInteger”>
           <xsd:minInclusive value=”12”>
        </xsd:restriction>
</xsd:simpleType>
```

So now we have defined a data type, named *over11*, which indicates all positive integers that are higher than 11.

2.7.6 Defining individuals or instances

We can also define individual objects in a class. Instances of both classes and of properties are written in RDF and RDF Schema syntax. You can see the specification of these languages for more detail about the various syntactic forms that are allowed. Now we will create a particular day student named Wesley:

```
<DayStudent rdf:ID=”Wesley”>
        <age><xsd:integer rdf:value=”22”/></age>
</DayStudent>
```

Note that it is also possible not to use an XML Schema data type along with a lexical representation of the value, by which the data type is used to parse the lexical representation into an actual value. Instead of this, we can also provide the lexical representation without the data-type. However, this is not recommended. The following example illustrates this.

```
<DayStudent rdf:ID=”Wesley”>
        <age>22</age>
</DayStudent>
```

From the range restriction in the age property definition we know that this is of the type xsd:nonNegativeInteger.

2.7.7 Daml collections

The last thing I want to discuss in this chapter is representing unordered collections of items, also known as bags or multi-sets. Whenever an element has the **rdf:parseType** attribute with value **daml:collection**, the enclosed elements must be interpreted as elements in a list structure. The following example illustrates this.

```
<oneOf rdf:parseType=”daml:collection”>
        <Thing rdf:resource=”#small”/>
        <Thing rdf:resource=”#large”/>
        <Thing rdf:resource=”#extralarge”/>
</oneOf>
```

The above must be seen as a list containing the three elements ‘small’, ‘large’ and ‘extralarge’.

2.8 FURTHER READING

This chapter is just an introduction to DAML+OIL. There are lots of other detailed features and combinations possible. I would recommend the interested reader to look at van Harmelen, Patel-Schneider and Horrocks’ “Annotated DAML+OIL Ontology Markup” and “Reference Description of the DAML+OIL Ontology Markup Language”, on which the content of this chapter has been based.

2.9 SUMMARY

After looking at different approaches in the context of the Semantic Web, we can conclude that an ontology approach is most suitable to use as knowledge representation formalism in the proposed solution (see later). If we would use RDF Schema, then we would need to define our own representations for example an equivalence relation. DAML+OIL is providing us such thing immediately as an extension of RDF Schema. Therefore we will use the DAML+OIL syntax, because it gives us all features we need, like collections that we can use for object composition and lots of other things.

However, this is not the end of the story. Now we have some representation formalism. This is one side of what we need in the solution. We want to transform this into Virtual Reality. Therefore we need something that we can use to represent Virtual Reality. In the next chapter I will give an introduction into the world of Virtual Reality and give the second part needed to propose a solution to the stated problem.

3 VIRTUAL REALITY

3.1 INTRODUCTION TO VIRTUAL REALITY

In Virtual Reality (VR), people concentrate on virtual worlds that are a good approximation to the real world. Other terms used to denote Virtual Reality are Artificial Reality, Simulator technology, Cyberspace or Synthetic Environments. However, there are many definitions of Virtual Reality that are used in many different contexts. Most of the definitions correspond with each other, but sometimes we can find some small differences.

Most people say that Virtual Reality is a technology that makes use of 3D graphics, simulation and special interaction devices to engage the senses more naturally and thereby creating an environment that facilitates more intuitive interaction with an application and its content. However, all these components are not necessary when talking about Virtual Reality, as we will see later on in this chapter. In the section types of VR systems we will see that VR is not just about interaction with a computer generated 3D graphical world. Like I already mentioned in the abstract of this thesis, the most complete definition of VR according to the style of VR we are talking about in this thesis comes from Aukstakalnis and Blatner. They defined Virtual Reality using the following definition:

"Virtual Reality is a way for humans to visualize, manipulate and interact with computers and extremely complex data."

We will now analyze the different part of this definition. The visualization part refers to the computer generated visual, auditory and other sensual outputs to the user of a world within the computer. This world can be a scientific simulation or even a view into a database. The user can interact with the world and directly manipulate objects within the world. Some of these worlds are animated by other processes, perhaps physical simulations or simple animation scripts. The almost real time interaction with the Virtual World is one of the critical tests to conclude if we can talk about VR or not.

There is a wide variety in applications being developed for VR. This goes from games to architectural planning and crash test simulations.

In the next section I'm going to give an overview of the existing types of Virtual Reality. After that I will give some overview of the aspects of VR programs. Then I will describe the different approaches to implement VR nowadays and giving some examples of these approaches. Finally I will conclude with the approach we will use for the generation of 3D graphics and simulation.

3.2 THE DIFFERENT TYPES OF VIRTUAL REALITY

3.2.1 Window on World Systems

This probably forms the most known form of Virtual Reality nowadays. Window on World (WoW) or desktop VR uses a conventional computer monitor to display the virtual world. Ivan Sutherland who laid out a research program for computer graphics introduced this type of VR in 1965. He proposed it as follows: “One must look at a display screen as a window through which one beholds a virtual world”.

3.2.2 Video Mapping

Video mapping is a variation on the WoW approach. It merges a video input of the user’s silhouette with a 2D computer graphic. The user watches a monitor that shows his body’s interaction with the world. An example of this is the Mandala Gesture Xtreme System from VividGroup. The system places the participant’s real time video image into Virtual Worlds. A motion-sensitive video camera captures the player’s image and processes relevant information such as height, arm-span, … to the software. The user’s body in turn is monitored by the software and used as the interface to control the Virtual Environment they occupy. The participant watches the mirror image on a large video monitor, moving within the Virtual World as they move in front of the camera. This way the player can control the game by just moving around. In fact, you are using your body as an input device.
Figure 18 shows a multiplayer version of some volleyball game using video mapping.

Figure 18: screenshot from a multiplayer volleyball game using video mapping

3.2.3 Immersive systems

These VR systems completely immerse the user's personal viewpoint inside the virtual world. These systems are often equipped with a Head Mounted Display, which is a helmet or a facemask that holds the visual and auditory displays. Figure 19 shows a picture of such a Head Mounted Display.

Figure 19: an example of a Head Mounted Display

A variation of immersive systems uses multiple large projection displays to create a room in which the viewer stands.

3.2.4 Telepresence

Telepresence links remote sensors in the real world with the senses of a human operator. Robots equipped with teleprocessing have already proved to be useful in deep sea and volcanic exploration. NASA also plans to use telerobotics for Mars exploration.
Telepresence can be described as the experience or impression of being present at a location remote from one's own immediate environment. Telepresence does not give a virtual world to the user, but it gives enough visual and audio senses to the user to make him feel to be there virtually.

3.2.5 Mixed Reality

Mixed Reality is the term used to denote the mix between telepresence and Virtual Reality systems. Computer generated inputs are merged with telepresence inputs and/or the user's view of the real world. An example is a surgeon's view of a brain surgery that is overlaid with images from earlier CAT scans and real-time ultrasound.

Next to the above five types of VR, people sometimes talk about distributed VR. This means that a simulated world runs on several computers connected over a network. People using these computers are able to interact in real time, sharing the same virtual world. This could have been the case in figure 18. Suppose that the users are at different location being filmed each by a separate camera, rather then all being filmed at the same place by the same camera, then we could have been talking about distributed VR.

The kind of Virtual Reality we want to generate in our approach will be in the domain of Window on World systems.

3.3 ASPECTS OF VR PROGRAMS

The basic parts of a system can be broken down into four different parts: an input processor, a simulation processor, a rendering process and a world database.

3.3.1 Input processes

The input processes of a VR program control the devices used to input information into the computer. There are a wide variety of possible input devices like a keyboard, mouse, joystick, a body suit, etc. Sometimes voice recognition is used. The input processing of a VR system is mostly kept simple, just getting the coordinate data to the rest of the system with minimal lag time, sometimes adding filtering and data smoothing processes.

3.3.2 Simulation process

The simulation process is the core of a VR program. This process knows about the objects in the world and the inputs coming from outside the world through an input device. The simulation process is responsible for handling the interactions, simulations of physical laws and determining the status of the world. The simulation is a process that is iterated one's for each frame or time step.
It is the simulation process that takes care for any task programmed into the world such as collision detection and dynamics, which we will see later on.

3.3.3 Rendering processes

The *visual renderer* is the most common process and it has a long history from the world of computer graphics and animation. The major consideration of a graphic renderer for VR applications is the frame generation rate. Twenty frames per second is roughly the minimum rate at which the human brain will merge a stream of still images and perceive a smooth animation. Visual renderers for VR use methods like the painter's algorithm, a Z-buffer or a scan line oriented algorithm. I would refer the interested reader to check books about computer graphics for more information about these algorithms.
The visual rendering process often has a series of sub-processes that are invoked to create each frame. Everything starts with a description of the world, the objects within the world, lightning and camera or eye position into the world. It starts by eliminating all objects that are not visible from the given camera position. Then the remaining objects have their geometry transformed into the eye coordinate system, which can be different from the world coordinate system. Then some hidden surface algorithm can be applied and finally the actual pixel rendering can begin.

Next to the visual renderer, we can also have some *auditory renderer*. A VR system is greatly enhanced by the inclusion of an audio component. This may produce mono, stereo or 3D audio. The last of these three is fairly difficult. Research into 3D audio has shown that there are many aspects of our head and ear shape that affect the recognition of 3D sounds. However, it is possible to apply a complex mathematical function, the Head Related Transfer Function, to a sound to produce this effect.

Next to that, we also have *"haptic rendering"*. Haptics is the generation of tough and force feedback information. Almost all haptic systems existing today have focussed on force feedback rather then on true touch sense.

3.3.4 The World Database

The storage of information on objects and the world is a major part of the design of a VR system. The primary things that are stored in the world database are the objects that inhabit the world, next to that we store scripts that describe actions of those objects or from the user. We also store information on lightning, program control and hardware device support.

So far we have seen a small introduction into the world of Virtual Reality. In the next section I'm going to describe some of the existing VR systems that can be used to generate a virtual environment.

3.4 *EXISTING VIRTUAL REALITY SYSTEMS*

There are two major categories for the available VR systems. The first category consists of toolkits. Toolkits are programming libraries that provide a set of functions for a particular programming language with which a skilled programmer can create a VR application. Programming libraries are generally flexible and have fast renderers, but the disadvantage is that you mostly need a fairly detailed knowledge of programming.
The second category is that one of the authoring systems. Authoring systems are complete programs with graphical user interfaces for creating worlds without falling back to detailed programming. Usually they include some scripting language in which the user can describe complex actions to happen in the generated world. So they are not really non-programming. Authoring systems are generally less flexible than programming libraries and have slower renderers.
I will now give an overview of some authoring systems and some programming toolkits.

3.4.1 Authoring systems

3D Construction Toolkit

This authoring system has an old-fashioned GUI and it includes a simple scripting language. You choose the shape you want from a list including hexagons, triangles, cubes, lines and pentagons. You then can stretch, shrink, turn and shade it and position it in your world. This way, complex buildings can be easily built up. Sensors can be added to the world to let something react in certain circumstances when the user of your world walks around. Once you have created the world for your simulation, you can start concentrating on the presentation. For example, the size of the window can be changed, the smaller the window the faster things will occur in your world. Figure 20 shows a screen shot of the 3D Construction Toolkit interface.

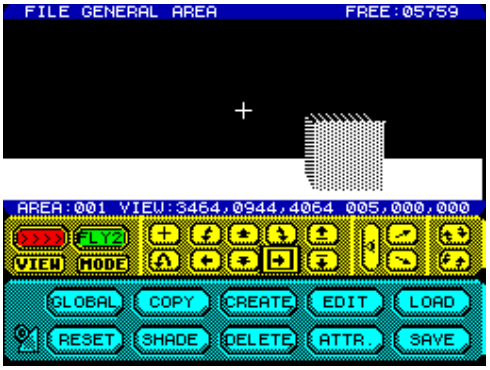

Figure 20: Screenshot from 3D Construction Toolkit

This program provides an easy way of generating 3D worlds. However, it does not correspond anymore to the requirements of the users of Virtual Worlds. We cannot implement sophisticated worlds like we are familiar with nowadays.

trueSpace

trueSpace is a widely used system that has gained industry recognition for advanced capabilities such as direct manipulation user interface. During the design process trueSpace provides real time feedback on virtually all operations through its advanced, direct manipulation user interface. The worlds generated with this engine have a frame generation rate of 30 frames per second, which is the same as used for the NTSC TV technology, which is the system of color telecasting used mainly in North America, Japan and parts of South America. This enables us to walk through the generated environments on a very realistic manner.
This system is frequently updated so it is not at all a problem to implement worlds that satisfy the user's requirements of a virtual world. Figure 21 shows a screenshot of the system while figure 22 shows a virtual environment generated with the system. As we can see, the generated world is very realistic.

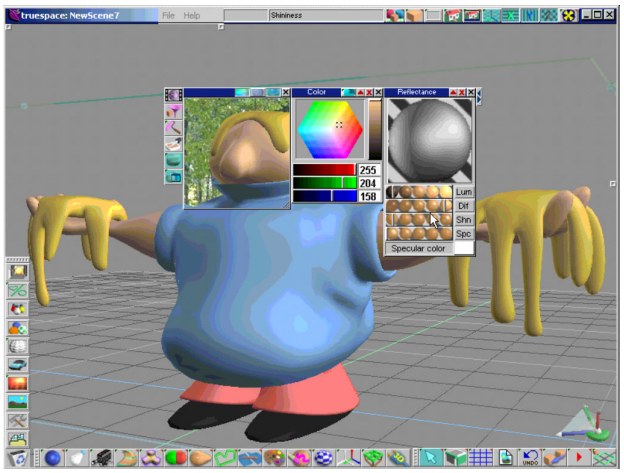

Figure 21: screenshot from the trueSpace system

Figure 22: a world generated with trueSpace

There are however a lot of authoring systems. It is impossible to describe all of them. I will just mention also VRCreator from VREAM Inc., VRT from Superscape and Virtual Home Space Builder from Paragraph International. In the next section I will introduce some programming libraries.

3.4.2 Programming Libraries

When looking at programming libraries I noticed that there are mainly three categories of programming libraries. First of all, we have the standard 3D engines, then there are 3D engines that provide collision detection and finally we also have 3D engines providing both collision detection and dynamics. I will introduce each of these concepts in this section and give some example libraries that fall in one of these categories.

3.4.2.1 Standard 3D Engines

The standard 3D engines are those that do not provide dynamics and collision detection. They just give you the power to create some virtual environment and it's up to the programmer to program what happens if two objects collide with each other and how some objects have a dynamic behaviour in the world.

The first engine in this category is Titan Engine. The Titan Engine was a project with the goal of showing up some OpenGL (see glossary) programming. The engine has been coded with extensibility, scalability and portability in mind as the main goals.
They have been working on the engine until May 2000, but since then, no updates or changes have been done. This engine is actually dead.

The second engine I will mention in this category is Hurricane. This engine is very primitive, but it may be possible that there will be updates in the future. It is an engine developed by an amateur-programmer. It has support for OpenGL and Direct3D (see glossary). It works only for Windows 98, but because of it's object oriented development it can be easily ported to other operation systems.

3.4.2.2 3D Engines with Collision Detection

Collision detection is the name used for the method that determines whether or not objects collided with each other in some virtual world. Collision detection can be handled in many different ways. The simplest way is to compare the bounding boxes of the objects. However, this is only efficient in case all objects are more or less boxes, but rigid objects can be very much more complicated. There are several algorithms for doing collision detection that I will not describe here. I will just mention a few like the Lin-Canny closest feature algorithm, V-Clip, I-collide and OBB-tree.

One engine in this category is Phoenix3D. Phoenix3D is a set of programming libraries and tools for the development of real time 3D software such as video games or multimedia software. It is written in C++, so it is object oriented and any programmer familiar with object-oriented programming can easily use it.
The Phoenix3D API is cross-platform. The same software code and data runs on MS-Windows, Sony Playstation 2 or MS X-Box.
The developer can export files in the Phoenix3D file format by using a plug-in for 3D Studio Max. The plug-in has a built-in viewer and a scene processor.

Another engine that falls into this category is Panard Vision. Panard Vision was an attempt in writing a fast generic high quality 3D renderer with support for the most common used rendering methods. Panard Vision has some strong points. First of all it is portable, which means that one can use the same code to have the same result on different operating systems. Next to that, it has a great level of abstraction so one can

benefit from hardware acceleration in a transparent manner. The Panard Vision API is simple to use and extensible. Panard Vision is free for non-commercial use.

3.4.2.3 3D Engines with Collision Detection and Dynamics

Dynamics in a 3D engine can be used to add believable, realistic and complex behaviour to real time 3D environments. Objects in the system can move or change state due to forces, torques or other causes. Most engines that provide dynamics let you produce imaginative simulations quickly and easily without immersing yourself in the complex details of the underlying mathematics and physics. Dynamics in a 3D engine is something with which we can manage the physics in our Virtual Reality environments.

A first engine that provides both collision detection and dynamics is Total Havok. Total Havok is a game SDK that includes all of the Havok Hardcore functionality and additionally offers fully integrated soft bodies, cloth, rope and deflectors. It also contains a rapid prototype layer. Rigid body response allows specification of mass, elasticity, friction and restitution. It also has highly efficient collision response models capable of simulating very large number of physics objects.
Total Havok is available for PC, Xbox and PlayStation 2. However, the main purpose of this engine is for developing games. Note also that this engine is not free to use.

A second engine that falls into this category of engines is the Vortex 3D Engine from Critical Mass Labs.
This engine provides advanced physics, rigid-body dynamics and collision detection. It delivers fast stable physics simulation suitable for use in interactive 3D applications. It is possible to create sophisticated environments using Vortex.
Vortex provides a set of well-structured components that can be used together as a complete physics engine. There are basically three part is the Vortex toolkit.
The fist is the Vortex Simulation library, which provides useful functions for creating and simulating objects using both the Dynamics and the Collision API's. In fact, the Simulation library forms the bridge between the Dynamics library and the Collision library. The library takes full advantage of the features available in both packages and uses them efficiently.
Next to that we have the Vortex Dynamics library, which has been engineered to satisfy both the rigorous requirements of physics modelling for applications like a ground vehicle simulator, and mixed environments with animated virtual humans. The library is equipped with a new physics solver and a series of modules all delivering the speed and stability critical to all interactive applications. The Vortex Dynamics library lets developers design and construct real time simulations efficiently.
The third component is the Vortex Collision library. It delivers the features of several academic packages. It provides a robust, efficient set of algorithms designed to deliver fast and accurate run time collision detection.
Vortex is a very complete 3D engine that seems very suitable for the work we have to do here. Other projects, like for example the work of Karl Sims about evolving 3D morphology and behavior, also used the Vortex engine.
Although the engine is not free to use, Critical Mass Labs provided us a free license we could use in this project.

3.5 SUMMARY

Finally we have all we need to start thinking about our own approach for creating VR applications.
In chapter 3 we already decided to use DAML+OIL as our knowledge representation technique for describing the worlds that will be generated. In this chapter I have explained why we decided to use the Vortex 3D engine as a mechanism for the implementation of our 3D applications.
As we have seen in this chapter, all existing methods for creating VR software require programming. Toolkits are completely programming based while authoring systems require some easy programming for using the scripting languages included.
What we now want to do is proposing a new approach that does not require programming at all. At this point, I don't know if this will be possible. It will certainly be possible to reduce the programming but is too early to decide that we can do it totally without programming. The proposed approach is completely described in chapter 5.

4 THE SOLUTION

4.1 INTRODUCTION

So far we have learned about knowledge representation formalisms and about Virtual Reality. What I will do in the rest of this thesis is outlining the approach I have developed for generating VR. The proposed solution here is a manner for generating VR where the developer doesn't need to program.
This will be done by going from a knowledge representation formalism, DAML+OIL, towards a VR implemented in C++ using the Vortex 3D engine. The developer will describe the world to be generated in DAML+OIL and based on this description a tool will generate the C++ code for the virtual world. Note that it is not necessary that the developer need to know DAML as he has the possibility to use a DAML+OIL editor. However, a notion of DAML+OIL will be an advantage.

4.2 OVERVIEW

First of all, I will explain the system of ontologies we will use in the approach. The tools I have developed and implemented are completely based on this system. Then I will explain the two tools that have been implemented. Finally I will give the conclusions that can be drawn from the work done in this thesis.

4.3 THE SYSTEM OF ONTOLOGIES

4.3.1 The Need for a System

The first step in the solution was to create a system of ontologies. One can raise the question why we need such a system and which information needs to be described by the different ontologies. Well, first of all, the main topic here is the mapping between objects described by means of DAML+OIL and objects in C++ (defined by Vortex). So we definitely need to describe what Vortex is and what DAML+OIL is to be capable to describe a general mapping mechanism between two such specifications. To describe Vortex and to describe DAML+OIL we will use an ontology. This results in the Vortex Ontology and the DAML Ontology. Also the mapping mechanism between DAML+OIL objects and Vortex objects will be described using an ontology: the Represented By Ontology. These three ontologies together form a kind of meta-level. They are independent of the specific virtual world we want to develop. The world for which a virtual world needs to be generated is also described in an ontology: the Domain Ontology. This ontology is specified using DAML+OIL and this is why the DAML Ontology is needed. The mapping of objects from the Domain Ontology to Vortex objects will be described in the Representable Domain Ontology, which can be regarded as an instantiation of the Represented By Ontology. An overview of the relations between the different ontologies is given in figure 23.

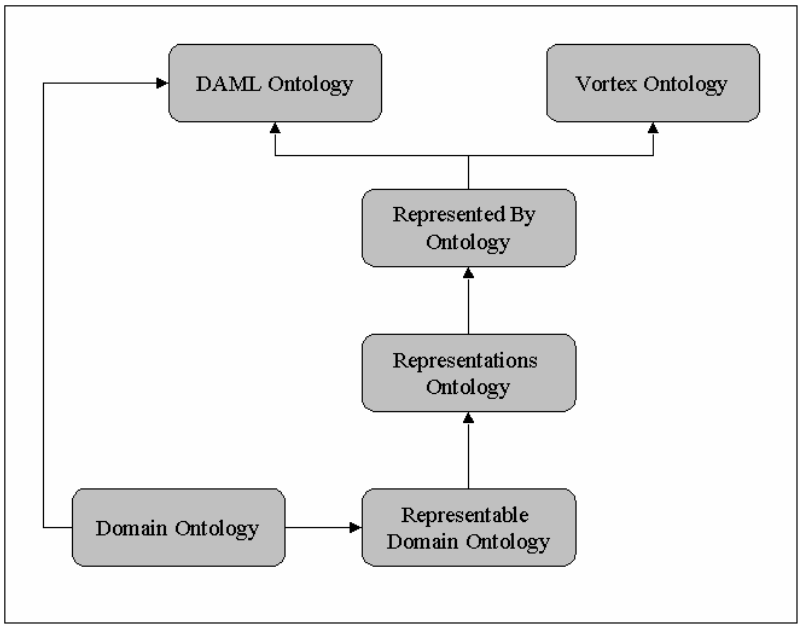

Figure 23: The System of Ontologies

In this figure, there is one more ontology that is not yet explained. This is the Representations Ontology. The role of this ontology will become clear when we explain the role of the different ontologies in more detail. After that, the complete system of ontologies is explained once more and also possible extensions are discussed. The best way to understand ontologies is to think object oriented, in other words, thinking in terms of objects, instances of objects and relations between objects.

4.3.2 The Vortex Ontology

The first ontology we need in the system is an ontology that gives the entire description of the Vortex engine. This ontology contains information like the simple objects that can be displayed using the 3D-engine. Also the different constraints and connections that are possible are described in this ontology.
To be able to generate code we also need to know which parameters are needed in order to create objects, for example to create a sphere. We need these to be able to ask the developer (later on) to give values for these parameters. Some examples of these parameters are radius, length and depth. We also need the types of the parameters to be able to control the input given by a developer.
I will now demonstrate the different parts explained above by giving samples from the ontology we have defined for Vortex. To specify the ontology DAML+OIL is used.

As explained, all simple object types in Vortex need to be specified in the ontology. An example of a simple object in Vortex is a sphere. The following part shows how we can describe a sphere.

```
<daml:Class rdf:ID="Sphere">
   <rdfs:subClassOf rdf:resource="#Object" />
</daml:Class>
```

The parameters that are needed to create an instance of a sphere are specified as follows:

```
<Sphere rdf:ID="SphereInstance">
   <HasAttributes rdf:resource="#Xpos" />
   <HasAttributes rdf:resource="#Ypos" />
   <HasAttributes rdf:resource="#Zpos" />
   <HasAttributes rdf:resource="#Color" />
   <HasAttributes rdf:resource="#Radius" />
   <HasAttributes rdf:resource="#Mass" />
</Sphere>
```

So an instance of a sphere has a position in a three dimensional space, represented by the x-position, the y-position and the z-position. Next to that, a sphere also has a color, a radius and a mass. Note that this is only an example and that the list of attributes can be expanded in the future.
Now we also need to describe what the attribute names mean. For example, what is an x-position? This can be done by using the following piece of DAML+OIL code:

```
<daml:Class rdf:ID="Xpos">
   <rdfs:subClassOf rdf:resource="#Attributes" />
   <HasType
         rdf:resource="http://www.w3.org/2001/XMLSchema#nonNegati
         veInteger" />
</daml:Class>
```

So an Xpos is an attribute that has the non-negative integer type.

Next to the descriptions of the objects that can be placed in a Vortex world, we also need to describe the connection types we can use to connect objects to each other in order to create complex objects. Vortex provides lots of different connection types, like

a ball and socket joint. The ball and socket joint is described as a connector. This is what the following code stands for:

```
<daml:Class rdf:ID="BallAndSocket">
   <rdfs:subClassOf rdf:resource="#Connector" />
</daml:Class>
```

Of course these connections also need attributes, like for example the objects to connect. This can be done in a similar way like for objects.

One very important aspect of this ontology is that we also need something that describes a 'thing' that cannot be represented. We call this 'thing' a NonRepresentable. It is described as being a WorldConcept:

```
<daml:Class rdf:ID="NonRepresentable">
   <rdfs:subClassOf rdf:resource="#WorldConcept" />
</daml:Class>
```

This is necessary because we also have things like gravity. Gravity is something that is part of a world but doesn't have a graphical representation. Note that everything is a WorldConcept, also a connector and an object. A WorldConcept is just a DAML class without attributes and connections. It is the simplest form of a class we can have in DAML+OIL.

This is how the Vortex Ontology looks like. The complete specification of the Vortex ontology is given in appendix A. Note that if someone wants to use the same approach for generating VR but with another VR engine, a similar ontology has to be developed for the engine that will be used.

4.3.3 The DAML Ontology

The second ontology in the system is the ontology describing DAML+OIL. This is an existing ontology, developed to be the normative reference on the precise syntax of the language constructs. In fact, it is the machine-readable RDF Schema definition of DAML+OIL.
In this ontology we can for example see that a DAML class and a DAML data type is a subclass of the Class concept in the RDF Schema. This shows that DAML+OIL is an extension on RDF Schema.
The following two pieces of code are samples taken from the DAML+OIL ontology, describing the properties I just mentioned of a DAML class and a DAML data type.

```
<rdfs:Class rdf:ID="Class">
    <rdfs:label>Class</rdfs:label>
    <rdfs:comment>
        The class of all "object" classes
    </rdfs:comment>
    <rdfs:subClassOf rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
</rdfs:Class>
```

```
<rdfs:Class rdf:ID="Datatype">
    <rdfs:label>Datatype</rdfs:label>
    <rdfs:comment>
        The class of all datatype classes
    </rdfs:comment>
    <rdfs:subClassOf rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
</rdfs:Class>
```

Note that this is just to illustrate the content of the DAML+OIL ontology. The complete specification of this ontology can be found on the website of DAML (see references).
Why do we need this ontology? Our initial goal is to map a concept described in the DAML syntax to concepts in a Virtual World. First we had the Vortex ontology, which can be seen as the set of destination objects, while we can say that the DAML+OIL ontology is the set of the domain objects for the mapping between the two. The mapping between the two formalisms is described in the third ontology of the model, which is called the 'Represented By' ontology. This ontology will be explained in the next section.

4.3.4 The Represented By Ontology

As I said before, this ontology describes the mapping mechanism between the two formalisms, namely between DAML+OIL concepts and Vortex concepts. This ontology contains only one DAML class describing the properties of such a mapping. This class is declared as follows:

```
<daml:Class rdf:ID="Representation">
    <rdfs:subClassOf>
        <daml:Restriction daml:mincardinality="1" daml:maxcardinality="1">
            <daml:onProperty rdf:resource="#Source" />
        </daml:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
        <daml:Restriction daml:mincardinality="1" daml:maxcardinality="1">
            <daml:onProperty rdf:resource="#Target" />
        </daml:Restriction>
    </rdfs:subClassOf>
</daml:Class>
```

As we can see, a representation has exactly one Source property and exactly one Target property. Source and Target are defined as follows:

```
<daml:ObjectProperty rdf:ID="Source">
   <rdfs:range rdf:resource="http://www.daml.org/2001/03/daml+oil#Thing" />
</daml:ObjectProperty>
<daml:ObjectProperty rdf:ID="Target">
   <rdfs:range
rdf:resource="http://wise.vub.ac.be/WofSy/VortexOntology.daml#WorldConcept" />
</daml:ObjectProperty>
```

So Source is an ObjectProperty that has a Daml:thing as resource. Note that everything in DAML is a Daml:thing, so this way, every DAML concept can be used to fill in the Source property. On the other hand, the Target, which is also an ObjectProperty, has as resource a WorldConcept from the Vortex ontology. Everything in the Vortex ontology is defined as being a subclass of the class WorldConcept. This definition allows us to use any concept from the Vortex ontology as a value for a Target property.
By this definition of Source and Target, we can use the Representation class to map any object from DAML onto any object from Vortex and this is exactly what we want.

4.3.5 The Representations Ontology

The next thing we want is some default representation for all objects in a described world. I have chosen to take a Sphere as default representation, but this is only a matter of personal taste. Someone else might even well have taken a cone or a box as default representation. This means that if we take some object from a world description, a Sphere will represent this object, unless there is some more specific representation that is specified by the developer. If there is a more specific representation, then this will be found in the Representable Domain ontology (see 5.7.3).
Now just for the completeness, I'll give the definition of the default representation as it can be found in the Representations ontology.

```
<rprs:Representation rdf:ID="ClassRepresentation">
  <rprs:Source rdf:resource="http://www.daml.org/2001/03/daml+oil#Class" />
  <rprs:Target
      rdf:resource="http://wendy.vub.ac.be/~wbille/VortexOntology.daml#Sphere" />
</rprs:Representation>
```

As we can see, the default representation is declared as being an instance of the Representation class. Note that 'rprs' is the prefix referring to the namespace declared as being the Represented By ontology. So the default representation maps a DAML class onto a Vortex Sphere.

4.3.6 The Domain Ontology

The Domain ontology is the ontology describing the virtual world someone wants to generate. So the developer who wants to construct a Virtual World creates this ontology (it is also possible that this ontology already exists). The Domain ontology is supposed to contain all objects that will be needed in the virtual world. To describe composite objects, the Daml:collection can be used, like we have seen in the chapter 3.7. Suppose we want to describe a personal computer as being a composition of a screen, a keyboard, a mouse and a computer unit, we can do this the following way:

```
<daml:Class rdf:ID="PersonalComputer">
    <daml:unionOf rdf:parseType="daml:collection">
        <daml:Thing rdf:about="#Screen"/>
        <daml:Thing rdf:about="#Keyboard"/>
        <daml:Thing rdf:about="#Mouse"/>
        <daml:Thing rdf:about="#ComputerUnit"/>
    </daml:unionOf>
</daml:Class>
```

Note that there also has to be a description of the objects in a collection. So somewhere we would need a description of Screen, Keyboard, Mouse and ComputerUnit.

4.3.7 The Representable Domain Ontology

The Representable Domain ontology contains instances of the Representation class described in the Represented By ontology. Each Representable Domain ontology is related to some Domain ontology. So a Representable Domain ontology describes representations for objects described in the related Domain ontology. Note that it is not necessary to have a representation for each object as we also have the default representation that can be used. Suppose we have an object Screen in the Domain Ontology and there is no representation for it in the Representable Domain Ontology, then we can say that a Screen is represented as a Sphere, which is the default representation described in the Representations Ontology.
The Representable Domain ontology is not created manually, but by using the first tool implemented for this thesis. (See 5.4. A first implementation: The Domain Transformer)

4.3.8 The Complete System

The system shown in figure 24 gives once again the connections between the different ontologies.

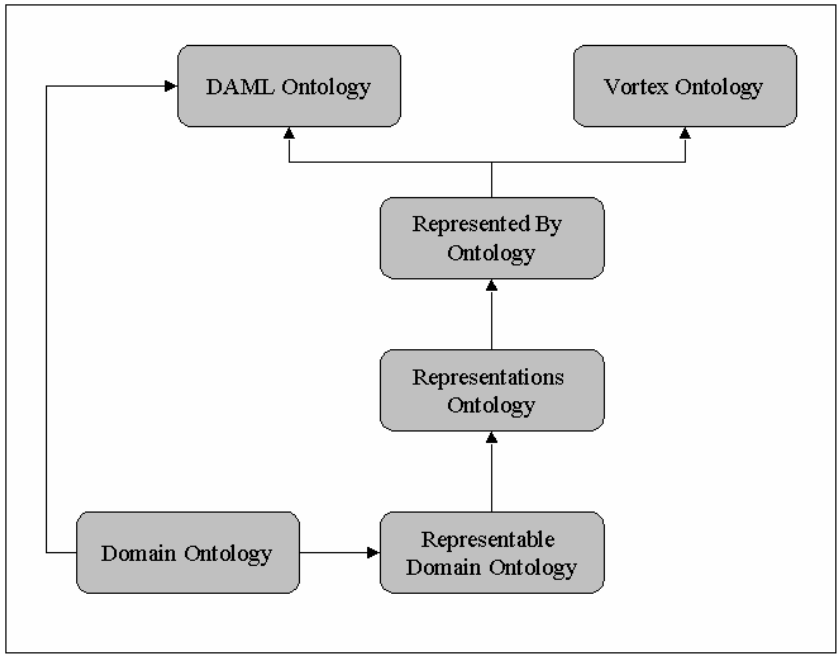

Figure 24: The System of Ontologies

As we can see in the figure, a domain ontology has a corresponding representable domain ontology that contains representations for objects defined in the domain ontology. The arrow between the two ontologies indicates the connection between the Domain Ontology and the Representable Domain ontology. If there is no representation for some object, then we can go one level up and use the default representation defined in the Representations ontology.
We can also extract from the model that the representation class described in the Represented By ontology maps things between Vortex and DAML. This is indicated by the arrow between the Represented By Ontology and the DAML ontology and by the arrow between the Represented By Ontology and the Vortex Ontology.

The arrow between the Domain Ontology and the DAML ontology indicated that everything in the Domain Ontology is described in means of DAML and so we can map a Domain Ontology to a Vortex Virtual World.

4.3.9 Layered System of Ontologies

The system described in the previous section can also be layered. Lets start with an example. Suppose someone creates a Domain Ontology A that contains a definition for 'wheel'. If I now want to create a Domain Ontology B that contains a description of 'car', then I don't have do describe the wheel again. I can just use the description from the ontology A. So my ontology B will contain a reference to the ontology A.
Suppose also that the person who built ontology A generated a Representable Domain Ontology for it. If I now want to generate a Representable Domain Ontology corresponding to the Domain Ontology B, then it is also not necessary to redefine the representation for 'wheel'. The Representable Domain Ontology for B can refer to the representation described in the Representable Domain Ontology corresponding with ontology A.
This promotes information reuse and consistency of representations. Note that it is not limited to one layer, we can expand the example by defining an ontology C using concepts of the ontology B, and so on. Figure 25 shows the layered model of ontologies.

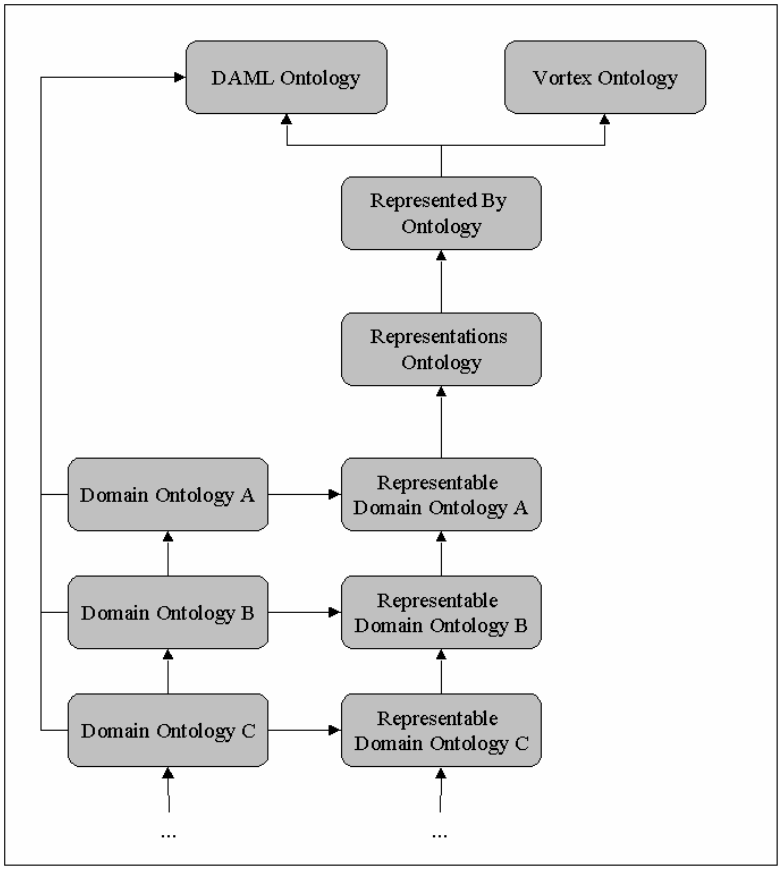

Figure 25: Layered System of Ontologies

4.4 THE DOMAIN TRANSFORMER

The Domain Transformer was developed in the context of this thesis. It is a program that can be used to produce a Representable Domain Ontology given a Domain Ontology. The user can give a Domain Ontology as input. Besides that, the user can also give some existing Representable Domain Ontology for the given domain. This way, it is possible to make small changes to the existing Representable Domain Ontology without having to rewrite it completely (see also section 5.3.9). The algorithm behind the tool is as follows (pseudo code):

Input: D: Domain Ontology, R: Representable Domain Ontology
Output: R: Representable Domain Ontology

```
For each class C in D do
  Begin
    If(class C is not a collection) do
      Begin
        If(R is not NIL) do
          Begin
            Find representation r in R for C
            If(r is not NIL) do
              Begin
                Ask the user if he wants to keep r for C;
                If(not user wants to keep r for C) do
                  Delete r from R
                Else
                  Return; // nothing has to be done
              End
          End
        // Check the parent chain for a representation of one of the parents
        For each P parent of C do
          Begin
            If(P has a representation r) do
              Begin
                Ask the user if he wants to inherit r for C;
                If(the user wants to inherit r for C) do exit 'for each'-loop
              End
          End
          If(still r for C is empty) do
            Begin
              Ask the user for a geometric representation g for C;
              Create a Representation instance for the class C in the Representable
              Domain Ontology R using g;
            End
      End
  End.
```

In natural language this gives the following:

For each class in the Domain Ontology perform the following steps:

1. *Check if the class is defined as being a collection of other classes. If so, then we don't need to have a representation for this class as the representation for it is composed of the representation of the classes in the collection. All classes in the collection are described somewhere and if these classes are reached by the algorithm then there will be constructed a representation for it. Otherwise, if the class is not a collection go to step 2.*
2. *If the user also specified an existing Representable Domain Ontology, then look in this ontology if there is somewhere a representation for the class we are treating. If there is a representation, then display this representation and ask the user if he wants to change it. If the user decides not to change it, then nothing has to be done. If it has to be changed, then delete the existing representation from the domain ontology and go to step 3. If the user didn't give an existing representable domain ontology in the beginning, then we can not check if there already exist some representation for the class we are treating. Then we go to step 3.*
3. *At this point we didn't find a representation for the class we are treating. Now we will look if the class is a subclass of something else. If it is, and the parent is in the domain ontology, then we can ask the user if it is OK to inherit the representation from the parent. If the parent is not in the domain ontology, then we will have to ask the user for the URI of the representable domain ontology corresponding to the domain in which the parent is described. If there is no such representable domain ontology or there is one, but it doesn't contain a representation for the parent, then we can look at the parent of the parent, and so on. If in the end we didn't find a representation for one of the parents or we did find one but the user doesn't want to inherit it, then we go to step 4. If we found a representation for one of the parents that can be used, then nothing has to be done as we can find that representation afterwards by using inheritance.*
4. *So now we are sure that there is no representation for the class. We ask the user if it is OK to use the default representation. If the default representation can be used, nothing has to be done because afterwards we can find the representation back by using inheritance. In the other case, we will have to ask the user what geometric representation has to be used. Given this geometric representation we can now construct a new instance of the representation class and add it to our representable domain ontology.*

So we see that the algorithm takes advantage of the subclass relations used in the Domain Ontology in order to reuse information.

4.5 THE WORLD CREATOR

The World Creator will take care of transforming the Representable Domain Ontology into a working Virtual Environment. This is done by asking the developer for additional information about the objects in the domain, like for example their mass, their position, and so on. The World Creator creates an extended Representable Domain Ontology and a working Virtual World. It could have been split up in two parts: extending the Representable Domain Ontology and then the creation of a Virtual World by extracting information from the extended Representable Domain Ontology. However, I've chosen to put the two together in one application, but that's just a matter of personal taste.

The advantage of this approach is that the developer will not work anymore in terms of spheres, cylinders, boxes and so on, but thanks to the existence of the Representable Domain Ontology (from which we can extract the geometric information), we can for example ask the developer for the radius of a head, the mass of a head and so on. This is possible because we know (by looking into the Representable Domain Ontology) that a head is represented by a sphere. This way, the user really works in his knowledge domain and not in the domain of Virtual Reality and 3D.

I have split up the description of the solution in three parts. First of all, the generation of a *static* world containing only simple objects, then the generation of *dynamic* worlds containing only simple objects and finally the generation of worlds containing composite objects.

4.5.1 Generation of Static Worlds

4.5.1.1 How to generate the code

In this section I will describe the generation of static Virtual Worlds containing only simple objects. These simple objects are the ones described in the Vortex Ontology, namely spheres, cylinders, cones and boxes. To be able to automatically generate the C++ code for the world, we will have to ask our users to give values for all attributes needed for the simple objects. I will first give a table of the objects that can be used in the world and the attribute values needed for them.

	Sphere	***Cylinder***	***Cone***	***Box***
Xpos	X	X	X	X
Ypos	X	X	X	X
Zpos	X	X	X	X
Color	X	X	X	X
Radius	X	X	X	
Length		X		X
Height				X
Depth				X
LowerHeight			X	
UpperHeight			X	
Mass	X	X	X	X

Table 1: overview of simple objects

The attributes Xpos, Ypos and Zpos denote the x-, y- and z-coordinate in the 3D coordinate system. The x-axes point out of the screen, the y-axes point up and the z-axes points to the left. Figure 26 shows the coordinate system used in a generated Virtual World.

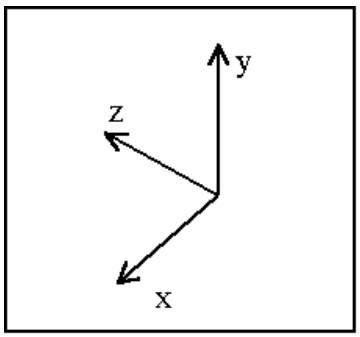

Figure 26: Coordinate system of a Virtual World

The position defined by Xpos, Ypos and Zpos is the position of the center of the object in our Virtual World.
The coordinate system can be changed using transformations and rotations. This can be done in Vortex using 4x4 transformation matrices. We can provide the user the utility for doing such changes.
We can also provide the option to our users to transform and rotate simple objects this way.

Note that thanks to the existence of the Representable Domain Ontology, we will not ask our users to give values for the attributes of a sphere, but we can work in terms of the domain. If there is for example a class 'head' in the Domain Ontology and someone decided during the generation of the Representable Domain Ontology to represent a head as a sphere, than we can now ask our user for the radius of the head, the mass of the head and so on.

I've also added an object 'environment' for which some attributes can be set. In the static worlds, we can provide the user the ability to change for example the size of the ground plane, the color of the ground plane or even whether there has to be a ground plane or not. It is also possible to change the camera position from which we view the world.

Once we have all the values for the attributes of all simple objects in the Virtual World and for the environment, we can start generating the 3D world and the related ontology containing all information about this world.

So how can we generate the C++ code for the Virtual World? First off all we start by generating the code for the include files. It are always the same include files that are needed, so we can simply insert it as a piece of text into the source file.
After that we generate the code for creating the World Renderer that is build-in in the Vortex 3D engine. The Vortex viewer API is a set of functions designed to provide Vortex developers with basic rendering functionality required to demonstrate their simulation code. First we have to create the renderer context that holds the state of the viewer. After that we have to declare pointers to the graphical structures that will appear in our world. This generated code is also pasted in the source file. Note that we also create a pointer to the ground plane. It is not possible to have a world without a ground plane, so if the world developer wants no ground plane, we can create a ground plane with height 0.

Once this is done we generate the code for a procedure called 'tick'. The renderer continuously calls this procedure. As we are generating static worlds here, nothing has to be done each clock-tick. So the tick-procedure will have an empty body.
When the user of a Virtual World closes the world, then a procedure named 'cleanup' is called. This procedure contains the code to destroy the renderer context and the code to destroy the world itself. So we also paste the code needed for this into the source file.
Optionally, it could be possible to generate the 4x4 matrices needed to perform the transformations and rotations of some objects in the world.
And then finally we have to generate the code for the main procedure. In the main procedure, first thing we do is initializing the renderer. Then we have to initialize the world by saying how much object it can maximum contain. Next to that we have to generate the code for initializing all parameters of the ground plane and all of the objects that are in the world. Note that we also have to transform the string representation we got from the user input for the color towards RGB-values.
Finally we can generate the code in the main procedure for setting the camera position, then we register the cleanup procedure and we end with the code that start the simulation loop.
Note that for a cylinder and a cone we will generate a standard 4x4 transformation matrix. This in because a cone and a cylinder are drawn in a lying position and it is more intuitive to draw them in a standing position. Therefore we introduce a standard transformation.

In the next section I will give an example of the creation of a world. There is also includes the source code that is generated, in which all different parts explained here are demonstrated.

4.5.1.2 An example generation

Suppose we want to generate a world in which we have a wall, a tower, a planet and a pyramid. We will create an ontology in which we define these four objects. Then we run the Domain Transformer to create a Representable Domain Ontology in which we define the following geometrical representations for our four objects: we represent the wall as a box, the tower as a cylinder, the pyramid as a cone and finally we represent the planet by a sphere.
So now we have our Representable Domain Ontology, and we can start running the World Creator. We will have to fill in the attributes for the objects in the world. Let's decide to have the default ground plane and also the default camera position. Table 2 shows the values we will fill in for the attributes of the objects, figure 27 shows a screenshot of the World Transformer, figure 28 shows the Virtual World itself and finally the generated C++ code is shown in Appendix B. Note that I changed the camera position of the world to have a better view of the objects present in the world.

	Wall	Tower	Pyramid	Planet
Xpos	3	-3	-3	0
Ypos	2	3	2	4
Zpos	0	-4	4	0
Color	Yellow	Blue	Green	Red
Radius		2	2	0.5
Length	10	6		
Height	4			
Depth	1			
LowerHeight			2	
UpperHeight			4	
Mass	20	40	30	10

Table 2: Values for an example Virtual World

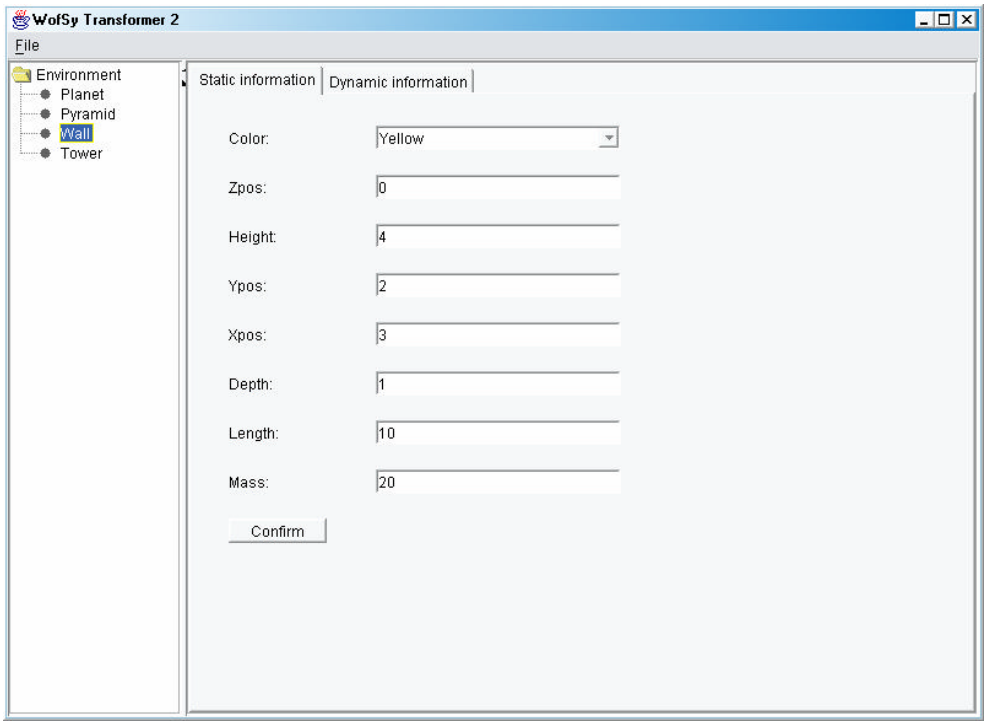

Figure 27: Screenshot from the World Creator

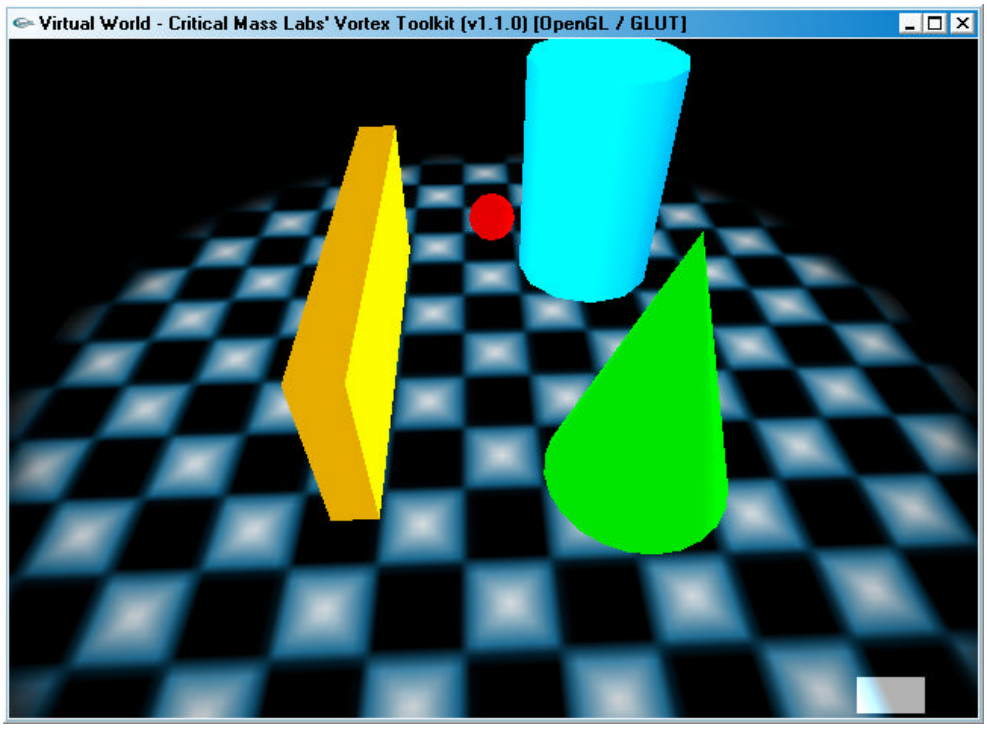

Figure 28: The generated Virtual World

4.5.1.3 The corresponding ontology

The World Creator does not only output the source file, but also an ontology that contains all information about the world. So by reading the generated ontology we can reconstruct the Virtual World. I will now show what will be written in the ontology.

As we will see, the ontology component for the simple elements is not that difficult. We just instantiate the geometrical representation used for the actual representation of the object. Let's look at the example of the wall. We can represent all the information about the wall as follows:

```
<vortex:Box rdf:ID="Wall">
   <vortex:Xpos>3</vortex:Xpos>
   <vortex:Ypos>2</vortex:Ypos>
   <vortex:Zpos>0</vortex:Zpos>
   <vortex:Length>10</vortex:Length>
   <vortex:Depth>1</vortex:Depth>
   <vortex:Height>4</vortex:Height>
   <vortex:Mass>20</vortex:Mass>
   <vortex:Color>Yellow</vortex:Color>
</vortex:Box>
```

It is not difficult to see that this contains all necessary information about the wall object. So we can do this for each object in our world, whether it is a cone, a box, a sphere or a cylinder. The only thing we still need to define in our ontology is the environment. We can do this using the following piece of code:

```
<daml:Class rdf:ID="Environment"/>
<daml:Class rdf:ID="Groundplane">
   <vortex:HasType rdf:resource="http://www.w3.org/2001/XMLSchema#boolean"/>
</daml:Class>
<Environment rdf:ID="EnvironmentInstance">
   <GroundPlane>true</GroundPlane>
   <vortex:Xpos>15</vortex:Xpos>
   <vortex:Ypos>-0.1</vortex:Ypos>
   <vortex:Zpos>0.5</vortex:Zpos>
</Environment>
```

Xpos, Ypos and Zpos here form together the position of the camera.
This is all we need to be able to reconstruct the world without any other information available.
At this point we are able to generate static worlds containing only simple objects and to generate the associated ontology. Now we can extend this solution in different ways. In the next section we will introduce dynamic worlds and after that, we will introduce world containing composite objects. Note that all the solution descriptions can be combined into one, namely the generation of dynamic worlds containing composite objects.

4.5.2 Generation of Dynamic Worlds with Collision Detection

4.5.2.1 How to generate the code

In this section I want to talk about the generation of worlds that have a dynamic behaviour. Having dynamics in a world also implies that we need some collision detection, because when objects can move around, they can also collide with each other. I will first give an overview of which dynamics we can introduce. In the next section I will show how we can save dynamics information in an ontology.

First of all, we can extract two sorts of dynamics, event-driven dynamics and continuous dynamics. Event-driven dynamics is when some dynamic behaviour is invoked due to some event happened in the world. Continuous dynamics is when dynamic behaviour is invoked constantly in the world without any reason. I will demonstrate the difference with a small example. Suppose we are developing a race game. Suppose also that the opponents in a race do not behave intelligent and just drive around on the circuit. Then the movement of the opponents is continuous. However, the car of the player may only move when some keys are pressed. For example when the upward arrow is pressed, then the car must drive forward and so on. This shows the difference between a continuous action and an event-driven action.

Now, how can we define such actions? There are different approaches for doing that. In this thesis I will explain how we can introduce actions by giving some objects an initial velocity or by putting some forces on objects. First we will look at the definition of actions using velocity.

Take for example the world we generated in 4.5.1.2. Suppose we want the planet to be thrown against the wall when spacebar is pressed. We will have to declare an event-driven action putting a velocity on the planet and an initial direction. If we want that the wall can be thrown over then we can also declare the wall as being movable. All other objects stay static in the world.

As we see in figure 29 and 30, the user can provide all this information using the World Creator. We see that there is a list of actions to which actions can be added and from which actions can be edited and deleted. We can also see that we can decide for each object whether it is fixed or movable. This is what is shown in figure 29. Figure 30 shows the user interface for defining new action. We can give a name to an action, we can attach a key event to an action and we can give the initial velocity for the object involved in the action.

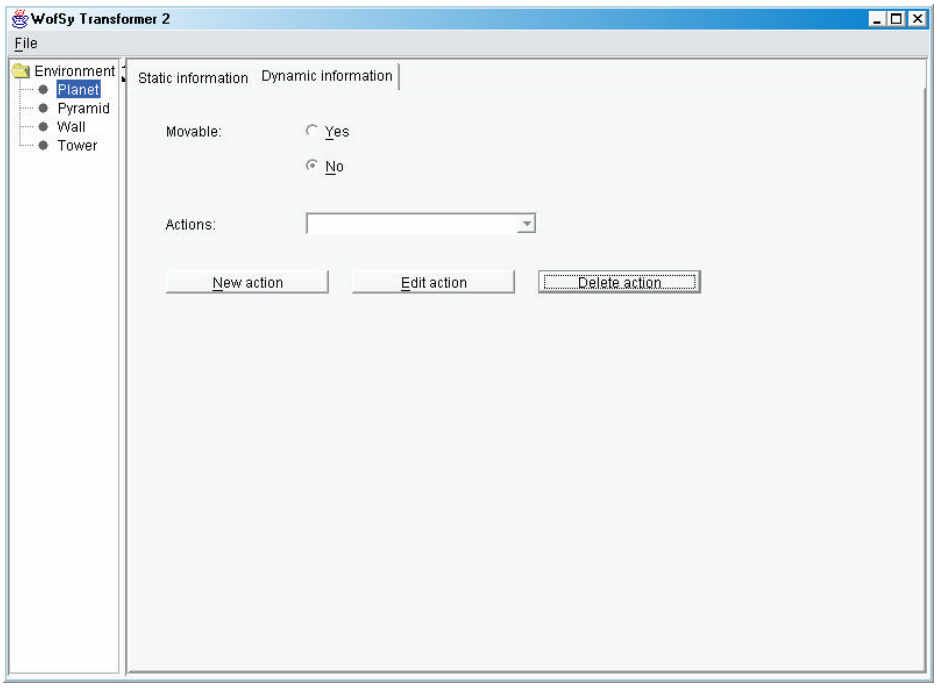

Figure 29: UI of dynamics in the World Creator

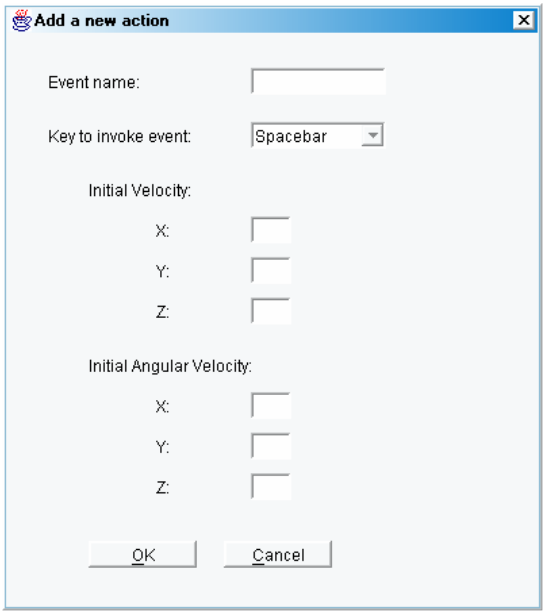

Figure 30: GUI for defining a new action

When generating the code, first thing we will have to do is defining a pointer to the collision space and a pointer to the bridge between the dynamical world and the collision space.
Next to that we will have to define pointers to the collision models for all objects in our world. Note also that we can now define into the environment variable whether there has to be gravity or not. If there has to be gravity we will have to generate the code for it.
Next thing we have to do is define a procedure that gives the dynamic behaviour. In the body of this procedure, we will have to enable the planet and the wall for movement, give the planet the wanted initial velocity and the initial angular velocity.

Next we can also define a procedure to reset the world after a dynamic behaviour has occurred.
In body of the tick procedure will not be empty any more. We will have to write a body that continuously updates the world. In the cleanup procedure we will also have to add code for destroying the collision models.
Now finally in the main procedure we will have to add code for the initialisation of the dynamic world, the collision space and the bridge between those two.
For each object in the world we also have to generate code for initialising its collision model. We also need to set the softness and restitution of contact. Finally we initialise the keyboard call-backs. With other words, we initialise the world so that when spacebar is pressed, the function containing the code for the dynamic behaviour of the planet is called. If we want, we can also generate code for displaying a help to the user of the generated Virtual World. Appendix C shows the code generated for this example.

As said before, we can also declare continuous actions. The only difference here is that we don't have to call a procedure using a registered keyboard call-back, but that we have to generate code that continuously calls the procedure defining the dynamic behaviour.

I also said that it is possible to add forces to some objects. This can be done by asking the user for the forces in the x direction, y direction and z direction and by using this information to generate the following piece of code:

MdtBodyAddForce(*body*, *xforce*, *yforce*, *zforce*);

By using this piece of code a force vector is added to a body. The force is applied at the centre of mass. We can also set the force to a specific point in the body using the following piece of code:

MdtBodyAddForceAtPosition(*body*, *xforce*, *yforce*, *zforce*, *xpos*, *ypos*, *zpos*);

Note that when we generate such code, we also have to generate the things we have seen for static worlds. It is also necessary to include other header files in our code, as will be the case also in the generation of worlds containing composite objects.

4.5.2.2 The corresponding ontology

In this section I will shows how we can save dynamic behaviour in an ontology. I will only show this for actions using initial velocity and initial angular velocity. However, it is similar when we start working with forces.

Using the following piece of DAML code we will define that when spacebar is pressed, the planet needs an initial velocity and an initial angular velocity:

```
<vortex:Ball rdf:ID="Planet">
    <vortex:Xpos>0</vortex:Xpos>
    <vortex:Ypos>4</vortex:Ypos>
    <vortex:Zpos>0</vortex:Zpos>
    <vortex:Radius>0.5</vortex:Radius>
    <vortex:Mass>10</vortex:Mass>
    <vortex:Color>Red</vortex:Color>
    <vortex:Movable>true</vortex:Movable>
    <vortex:Velocity>Initial Velocity</vortex:Velocity>
```

```
    <vortex:AngularVelocity>Angular Velocity</vortex:AngularVelocity>
    <vortex:Keypressed>Spacebar</vortex:Keypressed>
  </vortex:Ball>
```

Next piece of DAML code also defines the wall as being moveable:

```
<vortex:Box rdf:ID="Wall">
    <vortex:Xpos>3</vortex:Xpos>
    <vortex:Ypos>2</vortex:Ypos>
    <vortex:Zpos>0</vortex:Zpos>
    <vortex:Length>10</vortex:Length>
    <vortex:Depth>1</vortex:Depth>
    <vortex:Height>4</vortex:Height>
    <vortex:Mass>20</vortex:Mass>
    <vortex:Color>Yellow</vortex:Color>
    <vortex:Movable>true</vortex:Movable>
  </vortex:Box>
```

Note that it is not necessary to add to all other objects the Movable attribute and set it to false. We can accept intuitively when no Movable attribute is specified, then the object is not movable.

4.5.3 Generation of Complex Objects using the World Creator

4.5.3.1 How to generate the code

In this section I will describe how we can connect simple objects to form composites. First off all I will give an overview of all different connections we can use and how we can implement them in the World Creator, and finally I will show how we will describe this sort of things in our ontology describing the Virtual World.

When somebody defined a composite object in the Domain Ontology using a DAML collection, then the World Creator has to control whether all objects in that collection are attached to each other. When some objects are not attached, the World Creator can ask the user which connection type must be used for connecting the objects and what the physical properties of this connection must be.
Vortex provides a lot of joint types. I will describe a few of these here to give an idea of the power of the joint types. The first one I want to mention is the ball-and-socket joint. A ball and socket joint forces a point fixed in one body to be at the same position as that of a point fixed in another body. This joint is also known as the spherical joint in the scientific literature. An example for which a ball and socket joint could be used is to model a shoulder joint. Figure 31 shows an illustration of the ball and socket joint.

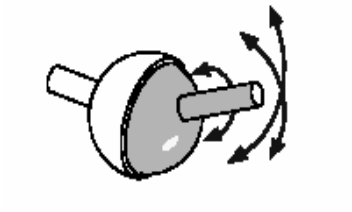

Figure 31: Ball and Socket joint

So what do we need to implement the possibility of attaching two objects to each other by using a ball and socket joint? We need a user interface in which we ask our users which objects must be connected, and what the position must be of the joint in the world. The World Creator would then have to generate the following code:

```
MdtBSJointID bs = MdtBSJointCreate(world);
MdtBSJointSetBodies(bs, body1, body2);
MdtBSJointSetPosition(bs, xpos, ypos, zpos);
```

The next joint type I describe is the hinge joint. A hinge leaves a pair of bodies free to rotate about a single axis, but otherwise completely fixed with respect to each other. Hinge joints are also known as revolute joints in scientific literature. An example for which we could use a hinge joint is to attach a door to a doorpost. Figure 32 shows the principle of a hinge joint.

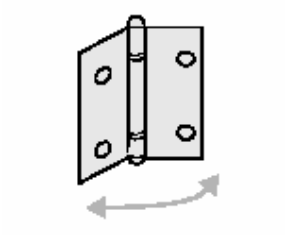

Figure 32: Hinge joint

To be able to generate code for such a connection, we will have to ask our user for the bodies that must be connected to each other by using the hinge joint. Another thing we will have to ask is the x-, y- and z-coordinate of the hinge axis. The hinge axis is determined by the point given by the user and the point (0,0,0). We can also ask for the limits for the relative rotation of the bodies and for the stiffness of the hinge rotation. Having this information, the World Creator could generate the following code:

```
MdtHingeID hinge = MdtHingeCreate(world);
MdtHingeSetBodies(hinge, body1, body2);
MdtBodyGetPosition(body1, pos);
MdtHingeSetPosition(hinge, xpos, ypos, zpos);
MdtHingeSetAxis(hinge, x, y, z);
```

If the user also specified limits and stiffness we can also generate the needed code for it. To conclude about joints I will describe the spring joint. This joint is used to attach one body to another. The spring joint tends to restore itself to its natural length by opposing any extension or any compression. The default behaviour is spring-like. Figure 33 shows the principle of a spring joint.

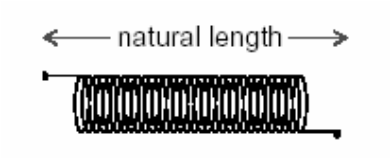

Figure 33: Spring joint

We can make the generation of such things possible by asking our user to give the objects that must be connected, the natural length of the spring and the stiffness of the spring. Having this information we can create the following piece of code that will take care of the simulation of a spring joint.

```
MdtSpringID spring = MdtSpringCreate(world);
MdtSpringSetBodies(spring, body1, body2);
MdtSpringSetNaturalLength(spring, NaturalLength);
MdtSpringSetStiffness(spring, stiffness);
```

There are lots of other joint types, like a prismatic joint, a universal joint, an angular joint, a car wheel joint, two sorts of linear joints, a fixed path joint, a fixed-position-fixed-orientation joint and a relative-position-relative-orientation joint. I would refer the interested reader to the Vortex Dynamics Developer Guide.

4.5.3.2 The corresponding ontology

The only thing that rests now is to show how these things can be placed in an ontology. First of all, I will show how we can represent a ball and socket joint in an ontology. Note that the two objects used in the joint will be defined as shown in 4.5.1.3 for simple objects. Once we know that the two simple objects will be defined that way, we can define the ball and socket joint as follows:

```
<vortex:BallAndSocket rdf:ID=”BS”>
    <vortex:Body>Body1</vortex:Body>
    <vortex:Body>Body2</vortex:Body>
    <vortex:Xpos>Xpos</vortex:Xpos>
    <vortex:Ypos>Ypos</vortex:Ypos>
    <vortex:Zpos>Zpos</vortex:Zpos>
</vortex:BallAndSocket>
```

For a hinge joint we can use the following to put in an ontology:

```
<vortex:Hinge rdf:ID=”hinge”>
    <vortex:Body>Body1</vortex:Body>
    <vortex:Body>Body2</vortex:Body>
    <vortex:Xpos>Xpos</vortex:Xpos>
    <vortex:Ypos>Ypos</vortex:Ypos>
    <vortex:Zpos>Zpos</vortex:Zpos>
    <vortex:Xaxis>x</vortex:Xaxis>
    <vortex:Yaxis>y</vortex:Yaxis>
    <vortex:Zaxis>z</vortex:Zaxis>
</vortex:Hinge>
```

And finally for a spring joint we can do it the following way:

```
<vortex:Spring rdf:ID=”spring”>
    <vortex:Body>Body1</vortex:Body>
    <vortex:Body>Body2</vortex:Body>
    <vortex:Length>NaturalLength</vortex:Length>
    <vortex:Stiffness>Stiffness</vortex:Stiffness>
</vortex:Spring>
```

For all the other joints I mentioned, we can use similar descriptions in the ontology. Note however that for the other joints, the Vortex Ontology has to be extended with the needed definitions.

4.5.4 Current State of the World Creator

First of all, I want to mention that the implementation of the World Creator developed in the context of this thesis is just a proof of concept. It does not cover all the aspects described here, but it gives an impression of the things that are possible using this approach.

With the current implementation it is possible to generate the static worlds and the associated ontology describing that world. The only thing described in the context of static worlds that is not possible yet is doing transformations and rotations, but that's not a big thing. It can be implemented by asking the users for the elements of the 4x4 transformation matrices. Another approach could be to ask the users for the degrees of rotations in the three coordinates, and based on that a 4x4 transformation matrix can be generated. This last approach is possibly the most user friendly one.

It is also possible to generate dynamic worlds. The user can say whether an object is movable, and it is possible to describe event-driven actions. Implementing continuous actions is almost the same as event-driven actions. The only difference is that the action must be performed continuously by putting it in the 'tick' procedure.

Using the current implementation it is not possible to create worlds with composite objects. However, it is not difficult to see how this can be done. We can give the user the possibility to attach one object to another by having a user interface that allows specifying the kind of connection (see 4.5.3) wanted and the objects that must be attached through it.

4.5.5 Future Work

In future the implementation can be extended in various ways. First off all, we can implement the missing parts described above, but we can also implement additional things. One thing that can be done is scaling of composite objects. Having this, somebody would have the possibility to say that given the current measures of the objects that compose a composite object, the composite object must be rescaled to for example 90% of it current size.
Another thing that can be extremely useful, for example in developing games, is automatic generation of different instances of some objects. Suppose somebody is developing a race game. When we look at the race games that exist nowadays, most of the time, all spectators look the same, except for the color of their hair, pull and their trousers. What we can do with this approach is giving the user the possibility to say that he wants for example 100 copies of some 'person' object, which must be scaled according to a normal deviation. This way, spectators would look mush more realistic, but it can also have advantages for lots of other things.
Another thing that can be done is the introduction of different materials for the objects. We could for example allow an object to be defined as having the material glass or wood. This allows someone to create a world in which collisions between objects become more realistic. When for example a ball of wood collides with a wall of glass, then it could be possible that the wall brakes into different pieces.
As you can see, their can still be done a lot in this research area and things can be extended with lots of different possibilities. In the next section I will describe the conclusions I can draw from this thesis.

5 CONCLUSIONS

I will conclude this thesis with some conclusions that can be drawn from it. First I will give a short overview of what we have seen in the different chapters. The conclusions will follow.

In chapter one we discussed some of the knowledge representation systems used in AI, but these seemed not very useful for what we needed here.
In chapter two we saw techniques used in the context of the Semantic Web and decided to use DAML+OIL in our approach.
Then in chapter three we had a small introduction about Virtual Reality.
Finally in chapter four, I described the solution to go from knowledge described in DAML+OIL towards a Virtual World.

The conclusions I can draw from this thesis are the following:

- It is possible to give the developer of a Virtual World a method for doing his work in terms of his domain knowledge instead of working in terms of 3D. Using the approach described above, the developer can just describe what he wants by defining the objects that must appear in the world and by defining physical properties of these objects to introduce movement in the Virtual World.

- Although I think that introducing this approach has reduced programming, I find that it is still impossible to do everything non-programming. An example of something that is still a major program to do non-programming is the usage of VR equipment like a head mounted display. It is probably still work for professionals to program Virtual Environment that are optimal for usage with such devices. Also implementing things like force feedback in games is something that is device-dependent and thus is more difficult to do non-programming.

- The final conclusion I want to draw from this thesis is that the answer to the question asked at the beginning of this research, namely: “Is it possible to generate Virtual Environments starting from a knowledge representation mechanism?”, is yes.

REFERENCES

References Knowledge Representation in AI

- Alison Cawsey, 1998, The essence of Artificial Intelligence, Prentice Hall (ISBN 0-13-571779-5)
- McGraw-Hill series in Artificial Intelligence, 1983, Artificial Intelligence, McGraw-Hill (ISBN 0-07-052261-8)
- A. Barr and E. Feigenbaum, 1981, The handbook of Artificial Intelligence (Volume 1), Pitman Books Limited
- Van Benthem, Van Ditmarsch, Ketting and Meyer-Viol, 1993, Logica voor Informatici (2de editie), Addison-Wesley (ISBN 90-7869-484-2)
- Minsky, 1975, A framework for representing knowledge
- Schank, R.C. and Abelson, 1977, Scripts, plans, goals and understanding, Hillsdale
- Green C., 1969, The application of theorem proving to question-answering systems
- Fillman R.E., Ascribing Artificial Intelligence to (simpler) Machines or When AI meets the real world, IntelliCorp Inc. California
- Fikes R.E, Hart P. and Nilsson N.J., 1972, Learning and executing generalised robot plans (Handbook of Artificial Intelligence Vol. 3), pp. 251 – 288
- What are semantic networks? A little light history, http://www.cogs.susx.ac.uk/local/books/computers-and-thought/chap6/node5.html

References Semantic Web

- T. Berners Lee, J. Hendler and O. Lassila, 2001, The Semantic Web: A new form of Web content that is meaningful to computers will unleash a revolution of new possibilities, in 'The Scientific American' May 2001
- B. Marchal, 1999, XML by example, Que (ISBN 0-7897-2242-9)
- T. Bray, D. Hollander and A. Layman, 1999, Namespaces in XML http://www.w3.org/TR/1999/REC-xml-names-19990114/
- The World Wide Web Consortium, http://www.w3c.org
- D.C. Fallside, 2001, XML Schema 0: Primer, W3C Recommendation

- D.C. Fallside, 2001, XML Schema 1: Primer, W3C Recommendation
- D.C. Fallside, 2001, XML Schema 2: Primer, W3C Recommendation
- Introduction to XML Schema, W3Schools Online, http://www.w3schools.com/schema/schema_intro.asp
- I. Stuart, 2001, XML Schema, a brief introduction
- O. Lassila and R. Swick, 1999, Resource Description Framework: Model and Syntax Specification, W3C Recommendation
- W. Nejdl, M. Wolpers and C. Capelle, RDF Schema Specification Revisited, Institut für Technische Informatik, Hannover
- F. Van Harmelen, P.F. Patel-Schneider and I. Horrocks, March 2001, Annotated DAML+OIL (March 2001) Ontology Markup
- F. Van Harmelen, P.F; Patel-Schneider and I. Horrocks, March 2001, Reference Description of the DAML+OIL (March 2001) ontology markup language
- The Darpa Agent Markup Language Homepage, http://www.daml.org
- Official definition of the language in RDF Schema form (The DAML Ontology) http://www.daml.org/2001/03/daml+oil

References Virtual Reality

- J. Isdale, What is Virtual Reality, http://www.isdale.com/jerry/VR/WhatIsVR/frames/WhatIsVR4.1.html
- Mandela System (Videomapping) http://www.vividgroup.com/products_main.html
- S. Aukstakalnis and D. Blatner, 1992, Silicon Mirage: The Art and Science of Virtual Reality, Peach Pit Press (ISBN 0-938151-82-7)

APPENDIX A: THE VORTEX ONTOLOGY

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!-- DAML+OIL Language, version 03/2001, generated by OntoEdit v2.0,
   ontoprise GmbH -->
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
            xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
            xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
            xmlns="http://wise/WofSy/VortexOntology.daml#">
   <daml:Ontology rdf:about="">
      <daml:versionInfo>$ http:// $</daml:versionInfo>
      <rdfs:comment>An ontology created by
            OntoEdit</rdfs:comment>
      <daml:imports
            rdf:resource="http://www.daml.org/2001/03/daml+oil" />
   </daml:Ontology>
   <daml:Class rdf:ID="Sphere">
      <rdfs:subClassOf rdf:resource="#Object" />
   </daml:Class>
   <daml:Class rdf:ID="LowerHeight">
      <rdfs:subClassOf rdf:resource="#Height" />
      <HasType
         rdf:resource=
         "http://www.w3.org/2001/XMLSchema#nonNegativeInteger" />
   </daml:Class>
   <daml:Class rdf:ID="RPROJoint">
      <rdfs:subClassOf rdf:resource="#Connector" />
   </daml:Class>
   <daml:Class rdf:ID="FixedPath">
      <rdfs:subClassOf rdf:resource="#Connector" />
   </daml:Class>
   <daml:Class rdf:ID="Linear1">
      <rdfs:subClassOf rdf:resource="#Connector" />
   </daml:Class>
   <daml:Class rdf:ID="Cylinder">
      <rdfs:subClassOf rdf:resource="#Object" />
   </daml:Class>
   <daml:Class rdf:ID="CarWheel">
      <rdfs:subClassOf rdf:resource="#Connector" />
   </daml:Class>
   <daml:Class rdf:ID="Linear2">
      <rdfs:subClassOf rdf:resource="#Connector" />
   </daml:Class>
   <daml:Class rdf:ID="Plane">
      <rdfs:subClassOf rdf:resource="#Object" />
   </daml:Class>
   <daml:Class rdf:ID="Angular3">
      <rdfs:subClassOf rdf:resource="#Connector" />
   </daml:Class>
   <daml:Class rdf:ID="Attributes">
      <rdfs:subClassOf>
         <daml:Restriction daml:mincardinality="2"
               daml:maxcardinality="2">
```

```
            <daml:onProperty rdf:resource="#HasType" />
            <daml:toClass
                rdf:resource=
                "http://www.w3.org/2001/XMLSchema#SimpleType" />
        </daml:Restriction>
    </rdfs:subClassOf>
</daml:Class>
<daml:Class rdf:ID="Zpos">
    <rdfs:subClassOf rdf:resource="#Attributes" />
    <HasType rdf:resource=
        "http://www.w3.org/2001/XMLSchema#nonNegativeInteger" />
</daml:Class>
<daml:Class rdf:ID="Stiffness">
    <rdfs:subClassOf rdf:resource="#Attributes" />
    <HasType rdf:resource=
        "http://www.w3.org/2001/XMLSchema#nonNegativeInteger" />
</daml:Class>
<daml:Class rdf:ID="Xaxis">
    <rdfs:subClassOf rdf:resource="#Attributes" />
    <HasType rdf:resource=
        "http://www.w3.org/2001/XMLSchema#nonNegativeInteger" />
</daml:Class>
<daml:Class rdf:ID="Yaxis">
    <rdfs:subClassOf rdf:resource="#Attributes" />
    <HasType rdf:resource=
        "http://www.w3.org/2001/XMLSchema#nonNegativeInteger" />
</daml:Class>
<daml:Class rdf:ID="Zaxis">
    <rdfs:subClassOf rdf:resource="#Attributes" />
    <HasType rdf:resource=
        "http://www.w3.org/2001/XMLSchema#nonNegativeInteger" />
</daml:Class>
<daml:Class rdf:ID="Ypos">
    <rdfs:subClassOf rdf:resource="#Attributes" />
    <HasType rdf:resource=
        "http://www.w3.org/2001/XMLSchema#nonNegativeInteger" />
</daml:Class>
<daml:Class rdf:ID="Mass">
    <rdfs:subClassOf rdf:resource="#Attributes" />
    <HasType rdf:resource=
        "http://www.w3.org/2001/XMLSchema#nonNegativeInteger" />
</daml:Class>
<daml:Class rdf:ID="Hinge">
    <rdfs:subClassOf rdf:resource="#Connector" />
</daml:Class>
<daml:Class rdf:ID="Radius">
    <rdfs:subClassOf rdf:resource="#Attributes" />
    <HasType rdf:resource=
        "http://www.w3.org/2001/XMLSchema#nonNegativeInteger" />
</daml:Class>
<daml:Class rdf:ID="Connector">
    <rdfs:subClassOf rdf:resource="#Representable" />
    <rdfs:subClassOf>
        <daml:Restriction daml:mincardinality="1">
            <daml:onProperty rdf:resource="#HasAttributes" />
```

```
            <daml:toClass rdf:resource="#Attributes" />
        </daml:Restriction>
    </rdfs:subClassOf>

</daml:Class>
<daml:Class rdf:ID="Width">
    <rdfs:subClassOf rdf:resource="#Attributes" />
    <HasType rdf:resource=
            "http://www.w3.org/2001/XMLSchema#nonNegativeInteger
            " />
</daml:Class>
<daml:Class rdf:ID=”Body”>
    <rdfs:subClassOf rdf:resource=”#Attributes” />
    <HasType
            rdf:resource="http://www.w3.org/2001/XMLSchema#string"
            />
</daml:Class>
<daml:Class rdf:ID="Object">
    <rdfs:subClassOf rdf:resource="#Representable" />
    <rdfs:subClassOf>
        <daml:Restriction daml:mincardinality="1"
                daml:maxcardinality="n">
            <daml:onProperty rdf:resource="#HasAttributes" />
            <daml:toClass rdf:resource="#Attributes" />
        </daml:Restriction>
    </rdfs:subClassOf>
</daml:Class>
<daml:Class rdf:ID="Color">
    <rdfs:subClassOf rdf:resource="#Attributes" />
    <HasType
            rdf:resource="http://www.w3.org/2001/XMLSchema#string"
            />
</daml:Class>
<daml:Class rdf:ID="Depth">
    <rdfs:subClassOf rdf:resource="#Attributes" />
    <HasType rdf:resource=
        "http://www.w3.org/2001/XMLSchema#nonNegativeInteger" />
</daml:Class>
<daml:Class rdf:ID="Velocity">
    <rdfs:subClassOf rdf:resource="#Attributes" />
    <HasType rdf:resource=
        "http://www.w3.org/2001/XMLSchema#nonNegativeInteger" />
</daml:Class>
<daml:Class rdf:ID="AngularVelocity">
    <rdfs:subClassOf rdf:resource="#Attributes" />
    <HasType rdf:resource=
        "http://www.w3.org/2001/XMLSchema#nonNegativeInteger" />
</daml:Class>
<daml:Class rdf:ID="Keypressed">
    <rdfs:subClassOf rdf:resource="#Attributes" />
    <HasType rdf:resource=
        "http://www.w3.org/2001/XMLSchema#String" />
</daml:Class>
<daml:Class rdf:ID="Xpos">
    <rdfs:subClassOf rdf:resource="#Attributes" />
```

```
   <HasType rdf:resource=
      "http://www.w3.org/2001/XMLSchema#nonNegativeInteger" />
</daml:Class>
<daml:Class rdf:ID="Spring">
   <rdfs:subClassOf rdf:resource="#Connector" />
</daml:Class>
<daml:Class rdf:ID="Height">
   <rdfs:subClassOf rdf:resource="#Attributes" />
   <HasType rdf:resource=
      "http://www.w3.org/2001/XMLSchema#nonNegativeInteger" />
</daml:Class>
<daml:Class rdf:ID="FPFOJoint">
   <rdfs:subClassOf rdf:resource="#Connector" />
</daml:Class>
<daml:Class rdf:ID="Length">
   <rdfs:subClassOf rdf:resource="#Attributes" />
   <HasType rdf:resource=
      "http://www.w3.org/2001/XMLSchema#nonNegativeInteger" />
</daml:Class>
<daml:Class rdf:ID="BallAndSocket">
   <rdfs:subClassOf rdf:resource="#Connector" />
</daml:Class>
<daml:Class rdf:ID="UpperHeight">
   <rdfs:subClassOf rdf:resource="#Height" />
   <HasType rdf:resource=
      "http://www.w3.org/2001/XMLSchema#nonNegativeInteger" />
</daml:Class>
<daml:Class rdf:ID="Box">
   <rdfs:subClassOf rdf:resource="#Object" />
</daml:Class>
<daml:Class rdf:ID="Cone">
   <rdfs:subClassOf rdf:resource="#Object" />
</daml:Class>
<daml:Class rdf:ID="Prismatic">
   <rdfs:subClassOf rdf:resource="#Connector" />
</daml:Class>
<daml:Class rdf:ID="Representable">
   <rdfs:subClassOf rdf:resource="#WorldConcept" />
</daml:Class>
<daml:Class rdf:ID="ConeLimit">
   <rdfs:subClassOf rdf:resource="#Connector" />
</daml:Class>
<daml:Class rdf:ID="NonRepresentable">
   <rdfs:subClassOf rdf:resource="#WorldConcept" />
</daml:Class>
<daml:Class rdf:ID="WorldConcept" />
<daml:Property rdf:ID="HasAttributes" />
<daml:Property rdf:ID="HasType" />
<daml:Property rdf:ID="RootRelation" />
<!-- Following definition only necessary for Generator to know which
      attributes are needed -->
<Hinge rdf:ID=”HingeInstance”>
    <HasAttributes rdf:resource=”#Body”/>
    <HasAttributes rdf:resource=”#Body”/>
    <HasAttributes rdf:resource=”#Xpos”/>
```

```
    <HasAttributes rdf:resource=”#Ypos”/>
    <HasAttributes rdf:resource=”#Zpos”/>
    <HasAttributes rdf:resource=”#Xaxis”/>
    <HasAttributes rdf:resource=”#Yaxis”/>
    <HasAttributes rdf:resource=”#Zaxis”/>
</Hinge>
<BallAndSocket rdf:ID=”BallAndSocketInstance”>
    <HasAttributes rdf:resource=”#Body”/>
    <HasAttributes rdf:resource=”#Body”/>
    <HasAttributes rdf:resource=”#Xpos”/>
    <HasAttributes rdf:resource=”#Ypos”/>
    <HasAttributes rdf:resource=”#Zpos”/>
</BallAndSocket>
<Spring rdf:ID=”SpringInstance”>
    <HasAttributes rdf:resource=”#Body”/>
    <HasAttributes rdf:resource=”#Body”/>
    <HasAttributes rdf:resource=”#Length”/>
    <HasAttributes rdf:resource=”#Stiffness”/>
</Spring>
<Sphere rdf:ID="SphereInstance">
   <HasAttributes rdf:resource="#Xpos" />
   <HasAttributes rdf:resource="#Ypos" />
   <HasAttributes rdf:resource="#Zpos" />
   <HasAttributes rdf:resource="#Color" />
   <HasAttributes rdf:resource="#Radius" />
   <HasAttributes rdf:resource="#Mass" />
</Sphere>
<Cylinder rdf:ID="CylinderInstance">
   <HasAttributes rdf:resource="#Xpos" />
   <HasAttributes rdf:resource="#Ypos" />
   <HasAttributes rdf:resource="#Zpos" />
   <HasAttributes rdf:resource="#Color" />
   <HasAttributes rdf:resource="#Length" />
   <HasAttributes rdf:resource="#Radius" />
   <HasAttributes rdf:resource="#Mass" />
</Cylinder>
<Plane rdf:ID="PlaneInstance">
   <HasAttributes rdf:resource="#Xpos" />
   <HasAttributes rdf:resource="#Ypos" />
   <HasAttributes rdf:resource="#Zpos" />
   <HasAttributes rdf:resource="#Color" />
   <HasAttributes rdf:resource="#Length" />
   <HasAttributes rdf:resource="#Mass" />
</Plane>
<Box rdf:ID="BoxInstance">
   <HasAttributes rdf:resource="#Xpos" />
   <HasAttributes rdf:resource="#Ypos" />
   <HasAttributes rdf:resource="#Zpos" />
   <HasAttributes rdf:resource="#Height" />
   <HasAttributes rdf:resource="#Color" />
   <HasAttributes rdf:resource="#Length" />
   <HasAttributes rdf:resource="#Depth" />
   <HasAttributes rdf:resource="#Mass" />
</Box>
<Cone rdf:ID="ConeInstance">
```

```
        <HasAttributes rdf:resource="#Xpos" />
        <HasAttributes rdf:resource="#Ypos" />
        <HasAttributes rdf:resource="#Zpos" />
        <HasAttributes rdf:resource="#Color" />
        <HasAttributes rdf:resource="#UpperHeight" />
        <HasAttributes rdf:resource="#LowerHeight" />
        <HasAttributes rdf:resource="#Radiuse" />
        <HasAttributes rdf:resource="#Mass" />
    </Cone>
</rdf:RDF>
```

APPENDIX B: GENERATED CODE FOR STATIC WORLD

```
/* This program is generated by WofSy Transformater 1.0
Copyright (c) 2002 Vrije Universiteit Brussel - Wise Lab */

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "MeViewer.h"
#include "Mdt.h"

/*Rendering context*/
        RRender *rc;

/* Dynamics world */
        MdtWorldID world;

void MEAPI tick(RRender *rc, void *userData)
{
        /* Nothing has to be done */
}

/* Graphic representation groundplane */
RGraphic *planeG;

/*Transformation matrix for ground plane*/
MeMatrix4 groundTransform =
{
        {1, 0, 0, 0},
        {0, 0, -1, 0},
        {0, 1, 0, 0},
        {0, 0, 0, 1}
};

void MEAPI_CDECL cleanup(void)
{
        /* Memory release */
        RRenderContextDestroy(rc);

        /* free dynamics */
        MdtWorldDestroy(world);
}

/* Possible colors to use */
        float White[4] = {1.0f, 1.0f, 1.0f, 0.0f};
        float Green[4] = {0.0f, 1.0f, 0.0f, 0.0f};
        float Red[4] = {1.0f, 0.0f, 0.0f, 0.0f};
        float Blue[4] = {0.0f, 0.598f, 0.797f, 0.0f};
        float Yellow[4] = {1.0f, 0.75f, 0.0f, 0.0f};
        float Black[4] = {0.0f, 0.0f, 0.0f, 0.0f};

/* Graphic representation for Sphere Planet*/
```

The needed include files

Creation of the renderer context

Creation of the world space

The tick procedure

Pointer to the ground plane

Ground plane transformation

The cleanup procedure

Default color definition

```
        RGraphic *PlanetG;

/* Physical representation */
        MdtBodyID Planet;

/* Mass for the sphere Planet*/
        MeReal PlanetMass = 10.0f;

/* Position vector for the sphere Planet*/
        MeReal PlanetPos[3] = {0, 4, 0};

/* Graphic representation for Cone Pyramid*/
        RGraphic *PyramidG;

/* Physical representation */
        MdtBodyID Pyramid;

/* Mass for the Cone Pyramid*/
        MeReal PyramidMass = 50.0f;

/* Position vector for the Cone Pyramid*/
        MeReal PyramidPos[3] = {-3, 2, 4};

/* Transformation matrix for the Cone Pyramid*/
        MeMatrix4 PyramidTransform=
        {
                {1, 0, 0, 0},
                {0, 0, 1, 0},
                {0, 1, 0, 0},
                {-3, 2, 4, 1}
        };

/* Graphic representation for Box Wall*/
        RGraphic *WallG;

/* Physical representation */
        MdtBodyID Wall;

/* Mass for the Box Wall*/
        MeReal WallMass = 50.0f;

/* Position vector for the box Wall*/
        MeReal WallPos[3] = {3, 2, 0};

/* Graphic representation for Cylinder Tower*/
        RGraphic *TowerG;

/* Physical representation */
        MdtBodyID Tower;

/* Mass for the Cylinder Tower*/
        MeReal TowerMass = 50.0f;

/* Position vector for the Cylinder Tower*/
        MeReal TowerPos[3] = {-3, 3, -4};
```

Pointer to the planet's graphic and physics

Pointer to the cone's graphic and physics

Default cone transformation

Pointer to the wall's graphic and physics

Pointer to the tower's graphic and physics

```
/* Transformation matrix for the Cylinder Tower*/
        MeMatrix4 TowerTransform=
        {
                {1, 0, 0, 0},
                {0, 0, 1, 0},
                {0, 1, 0, 0},
                {-3, 3, -4, 1}
        };

int MEAPI_CDECL main(int argc, const char **argv)
{
        /* Initialise renderer */
        MeCommandLineOptions* options;
        options = MeCommandLineOptionsCreate(argc, argv);
        rc = RRenderContextCreate(options, 0, !MEFALSE);
        if(!rc)
                return 1;

        /*Create the dynamics world */
        world = MdtWorldCreate(10, 10);
```

Default cylinder transformation

Initialisation of the renderer

world initalisation

Ground plane initialisation

```
/* ground graphic */
planeG = RGraphicGroundPlaneCreate(rc, 24, 1, White, 0);
RGraphicSetTexture(rc, planeG, "checkerboard");
```

Planet initialisation

```
/* ball dynamics */
Planet = MdtBodyCreate(world);
MdtBodySetPosition(Planet, PlanetPos[0], PlanetPos[1], PlanetPos[2]);
MdtBodySetMass(Planet, PlanetMass);

/* ball graphics */
PlanetG = RGraphicSphereCreate(rc, 0.5, Red,
                                        MdtBodyGetTransformPtr(Planet));
```

Pyramid initialisation

```
/* Cone dynamics */
Pyramid = MdtBodyCreate(world);
MdtBodySetPosition(Pyramid, PyramidPos[0], PyramidPos[1], PyramidPos[2]);
MdtBodySetMass(Pyramid, PyramidMass);

/* cone graphics */
PyramidG = RGraphicConeCreate(rc, 2, 4, 2, Green, PyramidTransform);
```

Wall initialisation

```
/* box dynamics */
Wall = MdtBodyCreate(world);
MdtBodySetPosition(Wall, WallPos[0], WallPos[1], WallPos[2]);
MdtBodySetMass(Wall, WallMass);
```

```
        /* box graphics */
        WallG = RGraphicBoxCreate(rc, 1, 4, 10, Yellow,
                                  MdtBodyGetTransformPtr(Wall));

        Tower initialisation
        /* Cylinder dynamics */
        Tower = MdtBodyCreate(world);
        MdtBodySetPosition(Tower, TowerPos[0], TowerPos[1], TowerPos[2]);
        MdtBodySetMass(Tower, TowerMass);

        /* Cylinder graphics */
        TowerG = RGraphicCylinderCreate(rc, 2, 6, Blue, TowerTransform);

        /* camera position */
        RCameraSetView(rc, (float)15, (float)-0.1, (float)0.5);   } Camera position

        RRenderSetWindowTitle(rc, "Virtual World");

        /* Cleanup after simulation */
        atexit(cleanup);                                          } Registering the cleanup procedure

        /* Run the simulation loop */
        RRun(rc, tick, 0);                                        } Registering the tick procedure

        return 0;
}
```

Note that sometimes in the code we see some dynamic components. These are introduced in this stage because the objects are connected to the world through their dynamic components. So introducing dynamic components at this early stage simplifies the code that must be generated.

APPENDIX C: GENERATED CODE FOR DYNAMIC WORLD

```
/* This program is generated by WofSy Transformater 1.0
Copyright (c) 2002 Vrije Universiteit Brussel - Wise Lab */


#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "MeViewer.h"
#include "Mdt.h"
#include "McdFrame.h"
#include "McdPrimitives.h"
#include "Mst.h"
#include "McdInteractionTable.h"
#define MCDCHECK


/*Rendering context*/
        RRender *rc;


/* Dynamics world */
        MdtWorldID world;

/* Collision space */
        McdSpaceID space;

/* Bridge between dynamics and collision */
        MstBridgeID bridge;

/* Graphic representation groundplane */
        RGraphic *planeG;

/* Collision representation */
        McdGeometryID GroundPrim;
        McdModelID GroundCM;

/*Transformation matrix for ground plane*/
MeMatrix4 groundTransform =
{
        {1, 0, 0, 0},
        {0, 0, -1, 0},
        {0, 1, 0, 0},
        {0, 0, 0, 1}
};



/* Possible colors to use */
        float White[4] = {1.0f, 1.0f, 1.0f, 0.0f};
        float Green[4] = {0.0f, 1.0f, 0.0f, 0.0f};
        float Red[4] = {1.0f, 0.0f, 0.0f, 0.0f};
```

Some extra include files

Pointer to collision space

Pointer to bridge

Collision model for ground plane

```
        float Blue[4] = {0.0f, 0.598f, 0.797f, 0.0f};
        float Yellow[4] = {1.0f, 0.75f, 0.0f, 0.0f};
        float Black[4] = {0.0f, 0.0f, 0.0f, 0.0f};


/* Graphic representation for Sphere Planet*/
        RGraphic *PlanetG;


/* Physical representation */
        MdtBodyID Planet;

/* Collision representation */
        McdGeometryID PlanetPrim;
        McdModelID PlanetCM;

/* Mass for the sphere Planet*/
        MeReal PlanetMass = 10.0f;


/* Position vector for the sphere Planet*/
        MeReal PlanetPos[3] = {0, 4, 0};


/* Graphic representation for Cone Pyramid*/
        RGraphic *PyramidG;


/* Physical representation */
        MdtBodyID Pyramid;

/* Collision representation */
        McdGeometryID PyramidPrim;
        McdModelID PyramidCM;

/* Mass for the Cone Pyramid*/
        MeReal PyramidMass = 50.0f;


/* Position vector for the Cone Pyramid*/
        MeReal PyramidPos[3] = {-3, 2, 4};


/* Transformation matrix for the Cone Pyramid*/
        MeMatrix4 PyramidTransform=
        {
                {1, 0, 0, 0},
                {0, 0, 1, 0},
                {0, 1, 0, 0},
                {-3, 2, 4, 1}
        };
```

Collision model for Planet

Collision model for pyramid

```
/* Graphic representation for Box Wall*/
        RGraphic *WallG;

/* Physical representation */
        MdtBodyID Wall;

/* Collision representation */
        McdGeometryID WallPrim;
        McdModelID WallCM;

/* Mass for the Box Wall*/
        MeReal WallMass = 50.0f;


/* Position vector for the box Wall*/
        MeReal WallPos[3] = {3, 2, 0};


/* Graphic representation for Cylinder Tower*/
        RGraphic *TowerG;


/* Physical representation */
        MdtBodyID Tower;

/* Collision representation */
        McdGeometryID TowerPrim;
        McdModelID TowerCM;


/* Mass for the Cylinder Tower*/
        MeReal TowerMass = 50.0f;

/* Position vector for the Cylinder Tower*/
        MeReal TowerPos[3] = {-3, 3, -4};

/* Transformation matrix for the Cylinder Tower*/
        MeMatrix4 TowerTransform=
        {
                {1, 0, 0, 0},
                {0, 0, 1, 0},
                {0, 1, 0, 0},
                {-3, 3, -4, 1}
        };

/* World gravity */
        MeReal gravity[3] = {0.0f, -0.5f, 0.0f};

/* User help */
char *help[2] =
{
        "$ACTION2: reset",
        "$ACTION3: shoot"
};
```

Collision model for wall

Collision model for tower

Introduction of gravity

User help for dynamic actions

Procedure for dynamic behaviour of the planet

```
/* Dynamic procedure shoot */
void MEAPI shoot(RRender *rc, void *userData)
{
        MdtBodyEnable(Planet);
        MdtBodyEnable(Wall);
        MdtBodySetPosition(Planet, PlanetPos[0], PlanetPos[1], PlanetPos[2]);
        MdtBodySetLinearVelocity(Planet, 4.0f, -4.0f, 0.0);
        MdtBodySetAngularVelocity(Planet, 0, 0, 0);
}
```

Procedure to reset the world after a simulation

```
/* The reset procedure */
void MEAPI reset(RRender *rc, void *userData)
{
        MdtBodyEnable(Planet);
        MdtBodySetPosition(Planet, PlanetPos[0], PlanetPos[1], PlanetPos[2]);
        MdtBodySetLinearVelocity(Planet, 0, 0, 0);
        MdtBodySetAngularVelocity(Planet, 0, 0, 0);

        MdtBodySetPosition(Wall, WallPos[0], WallPos[1], WallPos[2]);
        MdtBodySetLinearVelocity(Wall, 0, 0, 0);
        MdtBodySetAngularVelocity(Wall, 0, 0, 0);
}
```

Revised tick procedure

```
/* Tick procedure */
void MEAPI tick(RRender *rc, void *userData)
{
        McdSpaceUpdateAll(space);
        MstBridgeUpdateContacts(bridge, space, world);
        MdtWorldStep(world, 0.2f);
}
```

Revised cleanup procedure

```
void MEAPI_CDECL cleanup(void)
{
        /* Memory release */
        RRenderContextDestroy(rc);

        /* free dynamics */
        MdtWorldDestroy(world);

        /* free collision */
        McdGeometryDestroy(GroundPrim);
        McdModelDestroy(GroundCM);
        McdGeometryDestroy(PlanetPrim);
        McdModelDestroy(PlanetCM);
        McdGeometryDestroy(WallPrim);
        McdModelDestroy(WallCM);
        McdGeometryDestroy(PyramidPrim);
```

```
        McdModelDestroy(PyramidCM);
        McdGeometryDestroy(TowerPrim);
        McdModelDestroy(TowerCM);
        McdSpaceDestroy(space);
        MstBridgeDestroy(bridge);
        McdTerm();
}
```

The main procedure of dynamic world

```
int MEAPI_CDECL main(int argc, const char **argv)
{
        /* KEA contact properties */
        MdtContactParamsID params;

        /* Initialise renderer */
        MeCommandLineOptions* options;
        options = MeCommandLineOptionsCreate(argc, argv);
        rc = RRenderContextCreate(options, 0, !MEFALSE);
        if(!rc)
                return 1;

        /*Create the dynamics world */

        world = MdtWorldCreate(10, 10);
```

Setting the world gravity for the dynamic world

```
        /* Set world gravity */
        MdtWorldSetGravity(world, gravity[0], gravity[1], gravity[2]);
```

Taking care that bodies are disabled for movement when they come to rest after a simulation.

```
        /* Bodies must be disabled when come to rest */
        MdtWorldSetAutoDisable(world, 1);
```

Doing some necessary initialisations

```
        /* Some necessary initialisations */
        McdInit(McdPrimitivesGetTypeCount(), 100);
        McdPrimitivesRegisterTypes();
        McdPrimitivesRegisterInteractions();
```

Initialisation of the collision space

```
        /*Create the collision space */
        space = McdSpaceAxisSortCreate(McdAllAxes, 10, 20);
```

Initialisation of the bridge between collision and dynamics

```
        /* Create bridge between dynamics and collision */
        bridge = MstBridgeCreate(10);
        MstSetWorldHandlers(world);

        /* ground graphic */

        planeG = RGraphicGroundPlaneCreate(rc, 24, 1, White, 0);
        RGraphicSetTexture(rc, planeG, "checkerboard");
```

Initialising collision model for the ground plane

```
/* ground collision */
GroundPrim = McdPlaneCreate();
GroundCM = McdModelCreate(GroundPrim);
McdSpaceInsertModel(space, GroundCM);
McdModelSetTransformPtr(GroundCM, groundTransform);

McdSpaceUpdateModel(GroundCM);
McdSpaceFreezeModel(GroundCM);


 /* ball dynamics */
Planet = MdtBodyCreate(world);

MdtBodyEnable(Planet);

MdtBodySetPosition(Planet, PlanetPos[0], PlanetPos[1], PlanetPos[2]);
MdtBodySetMass(Planet, PlanetMass);

/* ball graphics */
PlanetG = RGraphicSphereCreate(rc, 0.5, Red,
    MdtBodyGetTransformPtr(Planet));
```

Initialising collision model for the Planet

```
/* ball collision */
PlanetPrim = McdSphereCreate(0.5);
PlanetCM = McdModelCreate(PlanetPrim);
McdSpaceInsertModel(space, PlanetCM);
McdModelSetBody(PlanetCM, Planet);


/* Cone dynamics */
Pyramid = MdtBodyCreate(world);
MdtBodySetPosition(Pyramid, PyramidPos[0], PyramidPos[1], PyramidPos[2]);
MdtBodySetMass(Pyramid, PyramidMass);

/* Cone graphics */
PyramidG = RGraphicConeCreate(rc, 2, 4, 2, Green, PyramidTransform);
```

Initialising collision model for the Pyramid

```
/* Cone collision */
PyramidPrim = McdConeCreate(2, 4);
PyramidCM = McdModelCreate(PyramidPrim);
McdSpaceInsertModel(space, PyramidCM);
McdModelSetBody(PyramidCM, Pyramid);
McdModelSetTransformPtr(PyramidCM, PyramidTransform);


/* box dynamics */
Wall = MdtBodyCreate(world);

MdtBodyEnable(Wall);

MdtBodySetPosition(Wall, WallPos[0], WallPos[1], WallPos[2]);
MdtBodySetMass(Wall, WallMass);
```

```
/* box graphics */
WallG = RGraphicBoxCreate(rc, 1, 4, 10, Yellow,
        MdtBodyGetTransformPtr(Wall));
```

Initialising collision model for the Wall

```
/* box collision */
WallPrim = McdBoxCreate(1, 4, 10);
WallCM = McdModelCreate(WallPrim);
McdSpaceInsertModel(space, WallCM);
McdModelSetBody(WallCM, Wall);



/* Cylinder dynamics */
Tower = MdtBodyCreate(world);
MdtBodySetPosition(Tower, TowerPos[0], TowerPos[1], TowerPos[2]);
MdtBodySetMass(Tower, TowerMass);

/* Cylinder graphics */
TowerG = RGraphicCylinderCreate(rc, 2, 6, Blue, TowerTransform);
```

Initialising collision model for the Tower

```
/* Cylinder collision */
TowerPrim = McdCylinderCreate(2, 6);
TowerCM = McdModelCreate(TowerPrim);
McdSpaceInsertModel(space, TowerCM);
McdModelSetBody(TowerCM, Tower);
McdModelSetTransformPtr(TowerCM, TowerTransform);
```

Setting some contact parameters

```
/* Parameters for contact */
params = MstBridgeGetContactParams(bridge,
           MstBridgeGetDefaultMaterial(), MstBridgeGetDefaultMaterial());

MdtContactParamsSetType(params, MdtContactTypeFriction2D);
MdtContactParamsSetRestitution(params, 0.5);
MdtContactParamsSetSoftness(params, (MeReal)0.0005);
```

Building the collision space

```
/*Build the space */
McdSpaceBuild(space);



/* camera position */
RCameraSetView(rc, (float)15, (float)-0.1, (float)0.5);
```

Initialise the keyboard call-backs for invocation of dynamic actions

```
/* Keyboard callbacks */
RRenderSetActionNCallBack(rc, 2, reset, 0);
RRenderSetActionNCallBack(rc, 3, shoot, 0);
```

Displaying help text to the user

```
        /* Help text */
        RRenderSetWindowTitle(rc, "Virtual World");
        RRenderCreateUserHelp(rc, help, 2);
        RRenderToggleUserHelp(rc);

        /* Cleanup after simulation */
        atexit(cleanup);

        /* Run the simulation loop */
        RRun(rc, tick, 0);

        return 0;
}
```