Graphical Interface to Ontologies

Mark Ib Alameddine, c974440
Supervisors: Hans Bruun & Jørgen Fisher Nilsson
Institut for Information Teknologi, DTU

2nd August 2001

A ma mère qui a tellement souffert pendant sa vie.

A mon père qui m'a toujours soutenu.

A un professeur, à un père, à monsieur Hans Bruun.

A un grand professeur, à monsieur Dines Bjørner.

A monsieur Flemming Stassen, l'homme de fer .

A tous mes amis, surtout à Fadi Indraous et Marc Etienne.

Abstract

Dansk: Projektet omhandler etablering af grafiske grænseflader for ontologier. Ontologier er abstrakte begrebsmodeller for et valgt domæne. I projektet har vi først undersøgt de forskellige forslag til grafisk visning af grafer og lignende strukturer, som de foreligger beskrevet i faglitteraturen. Desuden skal undersøges forskellige visningsformer for begrebsstrukturer, herunder entitets–relationsmodellen.

Derefter har vi udarbejdet et forslag til grafisk system for visning af ontologier, som de anvendes i OntoQuery-projektet. Herunder skal tages hensyn til ønskede funktionaliteter, såsom mulighed for at navigere i ontologien.

English: This project is about constructing graphical user interfaces for ontologies. An ontology is an abstract model of concepts for a specific domain. During the project we first examine the various ideas about displaying graphs and similar structures that are described in the references. We also examine various visualizations for concept structures including the entity-relationship model.

This work results in a proposal for a graphical system to display ontologies as there are used in the OntoQuery project. The desired functionalities are examined like, for instance, the possibility to navigate the ontology.

graphical system.

Acknowledgment

I would especially like to thank my two supervisors Jørgen Fischer Nilsson and Hans Bruun for their help during this project. I would also like to thank Asger Eir for helping me with ontology theory. I will not forget to thank a great friend Jesper Rasmussen, Erik Bo Simonsen for advices on the programming, and Nikolaj Oldager for reading my report. Finally, I would like to thank a dear friend Claus Zilmer for his support and patience.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Visualization of Ontologies

Although an ontology is an abstract model of concepts for a specific domain. Generally speaking, ontology is the "study of existence"

This project is about constructing graphical user interfaces for ontologies.

During the report we first talk about the notion of ontology and as an example of ontologies, a linear representation will be presented "conceptual grammar".

Then we examine the various ideas about displaying graphs in 2 and 3 dimensions, we will look carefully at the information visualization, one of the important ascpects that can be used in ontology, because a great deal of **ontology** is[1] for "structuring our knowledge", from where we can retrieve information efficiently from our structure.

More functionalities will be also examined, like for instance the possibility to navigate the ontology.

Generally we found a great amount of articles that can be the key to implement this graphical system for ontologies, but nothing has been located specifically about "visualization of ontologies."

1.2 The Report Outlines

The report is organized as follows:

Chapter 2: Serves as an introduction for ontology theory, where we will present the notion of ontology, a little introduction to lattice theory, The

[1]If we may say.

notion of *concept* will also be presented, that help us to define the notion of conceptual grammar, its form, and its restriction.

Chapter 3: This chapter is the main chapter in our report, we will discuss graph drawing in 2 dimensions and 3 dimensions, compare them by talking about their advantages/disadvantages. Then we will introduce the idea behind visualization of programs, and graph visualization, from where there will be taken decisions for the layout of the structure of **c**onceptual **g**rammar.

Chapter 4: In this chapter we will introduce **Java 3D API**, describe how we implement the graph layout by using this potential tool, and explain why it has been chosen. The graphical user–interface is called **ONTOVIZ**, that visualize a graphical system to display ontologies.

Chapter 5: This chapter is devoted to testing the parser, and the Ontoviz system to display ontologies.

Chapter 6: We will finish this report with a conclusion that includes the summary of the achievement, and the future work where a description of improving areas in our implementation.

Chapter 2

Ontology

2.1 What is an Ontology?

The word "ontology" seems to generate a lot of controversy. It has a long history in philosophy, in which it refers to the subject of existence . It is also confused with epistemology, which is about knowledge and knowing.

The discipline of O*ntology*, is defined in the literature mainly as "the study of existence" or " whatever exists".

Sowas definition:

" ***Ontology*** is the study of the categories of things that exist or may exist in some domain."

A short answer from Gruber:

An *O*ntology is a specification of a Conceptualization[1]

The notion of ***Ontology***, has created dispute between whom wants to actually define it.

In the context knowledge sharing, we may use the term ontology to mean a *explicit specification of a conceptualization.* That is, an **ontology** is a description (like a **formal specification of a program** of the concepts and relationships that can exist for an agent or a community of agents. This definition consistent with the usage of ontology as *set-of-concept-definitions*, but more general. And it is certainly a different sense of the word than its use in philosophy.

What is important is what an **ontology** is designed for the purpose for enabling knowledge sharing and reuse. In that context, an **ontology** is a specification used for making ontological commitments. Practically, an

[1] **O***ntologies, by B.C. Vickery, University College London, 1997..*

ontological commitment is an agreement to use a vocabulary, in a way that is consistent with the respect to the theory specified to ontology. We build agents that commit to ontologies. We design ontologies so we can share with and among these agents.

A body of formally represented knowledge is based on a *conceptualization:* the objects, concepts and other entities that are assumed to exist in some area of interest and the relationships that hold among them. A conceptualization is an abstract, simplified view of the world that we wish to represent for some purpose.

2.2 Conceptualization and Conceptual Grammar

Before presenting what is *conceptual grammar*?, we need to discuss what we understand or seem to understand by the notion of the **concept**.

We may understand a *concept* as some sort of **abstract entity**, which is referred to when we[2] communicate. A *communication* can be between two individuals (e.g persons) or it could be between a person and him-self, i.e a person iteratively expressing and reflecting own thoughts; This kind of communication can be *self-communication.*

We may say, that a **concept** is used to describe an *entity*. Entities are classified into *categories*, but the categories are first and foremost a classification of the way people think and talk about in the world. The nature of the world itself directly affects the human perception and indirectly the categories they use.

But then we are coming back to the idea of *communication*[3].

When humans communicate, they need to be precise, so they could be understood. For this purpose, they distinguish objects by properties, giving attribution to concepts.

So what is an *attribute* ?

John Sowa defined an *attribute* as the properties of a *continuant*, which are usually described by adjectives, verbs. Attributes include entities like colors, shapes, sizes and weights. The part of *continuant* are called *pieces*. Examples of pieces includes the doors and walls of a house, the states or provinces of a country.

[2] humans

[3] We refer to **humans** as an example of communication, but the reader may say that animals are able to communicate, don't forget nature !, and what about the Universe?. Here we try to focus on the understanding the notion of a concept.

So we may conclude that the notion of *conceptual grammar*, or *ontogical grammar* is to emphasize the notion of concept described or attached by attribute values.

After a brief discussion about understanding the notion of a *concept*, we will give a basic introduction to *lattice theory*.

In the next section, we will start our presentation of *lattice theory* with the notion of partition and classification[4], emphasizing one of the most important characteristics of lattices, namely that they facilitates *multiple inheritance*.

2.3 Classification and partition

We take our starting point in the Boolean algebra, considering a universe, a domain, i.e. a (possibly infinite) set $\mathcal{U}$ of concepts and a (possibly infinite) set $\mathcal{X}$ of objects. For every concept t in $\mathcal{U}$ there is a set S which is a subset of $\mathcal{U}$. We call S a model of t. t is a type in the conceptual system, and S is a (possibly infinite) set of objects.

By a *classification* we understand dividing our domain of objects into sub–domains, which we designate with some names or symbols, as t was a designation for S.

We can give an example:

```
STUFF ::= Cvitamin | Avitamin | Bvitamin ;
```

The union of set of concepts, Cvitamin, Bvitamin, Avitamin is equal to our set STUFF.

2.4 Multiple Inheritance

A classification, using the above approach, yields a hierarchical order of the sets of objects in the domain. A concept c_1, is said to be *more* general than another concept c_2 if and only if the set denoted by c_2 is a subset of the set denoted by c_1; we write $c_2 \leq c_1$.

Such a hierarchy is a tree structure where the root of the tree holds the most general concept; that is, the set of all objects in the domain ($\mathcal{U}$). However, such a tree structure is inadequate (or at least inconvenient) if we wish to classify on different, incomparable levels. To facilitate this, lattices are used in most knowledge based systems as lattices add the facility of multiple inheritance. The lattice structure represents a partial ordering of concepts.

[4] Included from *Comparing Top–Down and Bottom–Up Concept Analysis in Ontology, Unpublished, by* ***Asger Eir***.

The least general of two concepts a and b is called *The Least Upper Bound* (the *meet* of a and b, denoted $a + b$), and the concept which inherits from both a and b is called *The Greatest Lower Bound* (*join*) of a and b, denoted $a \times b$. The most general concept is called *Top* ($\top$) and the most specialized concept, inheriting from all other concepts resulting in *absurdum* is called *Bottom* ($\perp$).

Usually, lattices are defined by the following axioms:

Definition 1 *A lattice is a non–empty set L together with two binary operations $\times$ (join) and $+$ (meet) such that for all elements x, y, and z in L:*

$$
\begin{aligned}
&\textit{Commutativity} \quad & x + y &= y + x & (2.1)\\
& & x \times y &= y \times x & (2.2)\\
&\textit{Associativity} & x + (y + z) &= (x + y) + z & (2.3)\\
& & x \times (y \times z) &= (x \times y) \times z & (2.4)\\
&\textit{Idempotence} & x + x &= x & (2.5)\\
& & x \times x &= x & (2.6)\\
&\textit{Absorption} & x &= x + (x \times y) & (2.7)\\
& & x &= x \times (x + y) & (2.8)
\end{aligned}
$$

In addition, as $\perp$ is neutral to $+$ and null–element to $\times$; $\top$ is neutral to $\times$ and null–element to $+$:

$$
\begin{aligned}
&\textit{Bounds} \quad & x \times \perp &= \perp & (2.9)\\
& & x \times \top &= x & (2.10)\\
& & x + \perp &= x & (2.11)\\
& & x + \top &= \top & (2.12)
\end{aligned}
$$

A partial ordered set yields a *Hasse–diagram*, and this Hasse–diagram is a lattice if and only if equation 2.1 through equation 2.12 are valid. A partially ordered set (a *poset*) satisfies the axioms of *Reflexivity*, *Anti symmetry* and *Transitivity*.

We believe that the possibility of classifying on various incomparable levels (i.e. working with multiple inheritance and thus using the strength of the lattice structure) holds the key to the question of how ontology construction differ.

2.5 The general form of a Conceptual Grammar

Before digging into the general form of ***conceptual grammar***, we need to introduce the general form of a grammar.

2.5.1 The general form of a Grammar

Informally, a **grammar** consists of:

- A set of productions rules and a start symbol.

- A **production rule** consists of two parts, a left-hand side and a right-hand side, separated by a left-to-right arrow.

- The **left-hand side** is name of the syntactic construct; the **right-hand** side shows a possible form of the syntactic construct. An example of production rule is:

```
expression -> '(' expression operator expression ')'
```

- The right-hand side of a production rule can contain two kinds of symbols, a **terminal symbols** and **non-terminal symbols**.

- A **terminal**[5] **symbol** (or **terminal** for short) is an end point of the production process, and can be part of the strings produced by the grammar.

- A **non-terminal symbol** (or **non-terminal** for short) must occur as the left-hand side (the name) of one or more productions rules, and cannot be part of the strings produced by the grammar.

- Non-terminals and terminals together are called **grammar symbols**.

- The grammar symbols in the right-hand side of a rule are collectively called **members**; when they occur as nodes in a syntax tree they are often called its **'children'**.

[5]Terminals are also called **tokens**, especially when they are part of an input to analysed.

As we defined above the form of a grammar, we resume the idea of the form of the conceptual grammar, that has the form of a **BNF** grammar.

the ***conceptual grammar rule*** has a **non-terminal-concept** on the left, a *defining marker* "::=" in the middle, and a **terminal concept** that endes with **';'**. A **terminal concept** can be either a **simple non-terminal concept** or a **complex concept** that includes a **non-terminal concept**, a terminal symbol **[**, a list of attributes that links to **non-terminal concepts** and a terminal symbol **]**.

Here is the general form of *conceptual grammar rule:*

```
concept_ID::=
      concept_ID|...|concept_ID[attribute_VALUE:concept_ID]|...|
      concept_ID[attribute_VALUE:concept_ID, attribute_VALUE:concept_ID]|...|
      concept_ID[attribute_VALUE:concept_ID[attribute_VALUE:concept_ID[...]]]|...|
      concept_ID[attribute_VALUE:concept_ID,attribute_VALUE:
               concept_ID[attribute_VALUE:concept_ID[...]]]
      concept_ID[attribute_VALUE:concept_ID[...,attribute_VALUE:
               concept_ID[attribute_VALUE:concept_ID[...,...  ]]]];
```

here is some examples:

```
Occur ::= State[CBY:Occur];
State ::= Symptom | Disease | missing[WRT:Stuff];
Stuff ::= Cvitamin | Avitamin;
```

`XX ::= Y[LENGTH:Yy,A:B[h:r]];` The ids in this example are none relevant.

After defining the *general form of* ***c****onceptual* ***c****rammar*, let us ask ourselves this question:

Is conceptual grammar a special case of a BNF grammar ?

Short answer: ***yes, it is.***

The *conceptual grammar* has the properties of a *BNF grammar*, i.e same form, but in the **conceptual grammar** the non-terminals of the grammar (concept classes) correspond to ontological classes. So that the grammar is to reflects a domain ontology, that includes (**ONTOLOG**), emphasizing the role of lattices for classification. So our interest is to represent only lattices defined as concepts, linked in between by relationship, presented as a list of attributes or features.

Another obvious difference between ***conceptual grammar*** *and* ***BNF grammar****, is in addition of the representation of the BNF grammar that is hierarchal, the conceptual grammar can be represented cyclic, and with a top and a bottom (lattice). Note that, we are not interested to visualize in the conceptual grammar, the* terminal symbols *like '***,***', '[', '|', ']', ';'.*

2.6 Wellformedness of Conceptual Grammar

After defining the form of the **conceptual grammar**, we can notice that in every production rule, there is concepts, defined on the left-hand side as **non-terminal concepts** and also on the right-hand side as **terminal concepts**.

Terminals concepts have two categories, the first categories are **the complex concepts**, and they are beginning with a concept followed by an opening bracket **'['** and ended by a closing bracket **']'**. In between brackets a list of features/attributes followed by concepts again. The other categories are simple–concepts which are concepts of course.

What are the restrictions of presenting the same concepts ids on the left-hand and right hand of the defining marker '::='?

- A mutual recursion is not allowed, meaning that, the concept–id defined on the left-hand, can not be the same concep–id, in the complex concept before the opening bracket, defined on the right-hand. So in between brackets is completely allowed.

Here is an example:

`Occur ::= Occur[...];` This is illegal.

`Occur ::= State[WRT:Occur];`This is legal.

Here is another example:

`Occur ::= State[...]; ...  State ::= Occur[...];`

This is also a from of mutual recursion, and of course completely illegal.

- All the concept–ids defined in the complex concept must be defined in the productions rules.

here is an example:

`Occur ::= State[...];`

The concept id State must be defined in the production rule.

```
State ::= ...
```

Here is another example:

```
State ::= missing[WRT:Stuff];
```

The concept id Stuff must be defined in the production rule.

```
Stuff ::= Cvitamin | ...;
```

- The **simple-terminal** concepts defined on the right–hand side of the *defining marker* '**::=**', can never be defined in the production rules.

here is an example:

```
State ::= Symptom | Disease ...;
```

Symptom and disease can not be defined in the production rules, meaning:

`Symptom ::= ...;` This is illegal.

`Disease ::= ...;` This is also illegal.

- The *feature_id* must be an element of the list of attributes, see table 2.1.

Role	Legend
TMP	temporal aspects (generic role)
LOC	location, position
PRP	purpose, function
WRT	with respect to
CHR	characteristic (property ascription)
CUM	cum (i.e., with accompanying)
BMO	by means of, instrument, via
CBY	caused by
CAU	causes
CMP	comprising, has part
POF	part of
AGT	agent of act or process
PNT	patient of act or process
SRC	source of act or process
RST	result of act or process
DST	destination of moving process

Table 2.1: The List of Attributes.

*Wellformedness will be discussed in the chapter of implementation using **Java 3D API** for the **grammar** constructed from our conceptual grammar general form, by using **RAISE Specification tool***

In the next chapter, we will dig in graph drawing in 2D, 3D their advantages/disadvantages, then we will discuss visualization of programs, graph visualization and finally we will take decisions for the visualization of our conceptual grammar.

Chapter 3

Visualization

3.1 Graph Drawing in 2 Dimensions

The Visualization of complex conceptual structure is a key component of support tools for many applications in science and engineering. A graph is an abstract structure that is used to model information.

Graphs are used to present information that can be modeled as objects and connections between these objects. In the **Graphs Drawing**[1] **,Algorithms for the Visualization of Graphs**, many algorithms has been described for generating clear and readable diagrams of complex **conceptual structures**.

In the following it will be presented an essential aspect, whose goal is to produce clear and readable drawings of graphs.

3.1.1 Straight line, Polyline and orthogonal drawings

Various graphic standards have been proposed for the representation of graphs in the plane. Usually, the vertices are represented by symbols such as circles or boxes, and each edge (u,v) is represented by a simple open curve between the symbols associated with the vertices u and v. A drawing such that each edge is represented by a polygonal chain is a **polyline drawing** see figure 3.1.

There are two common special cases of this standard. A **straight line drawing** maps each edge into a straight line segment, see figure 3.2.

This standard is commonly adopted in graph theory texts. An ***orthogonal drawing*** maps each edge into a chain of horizontal and vertical segments see figure 3.3.

[1]By: Giuseppe Di Battista, Peter Eades, Roberto Tamassia, Ioannis G. Tollis.

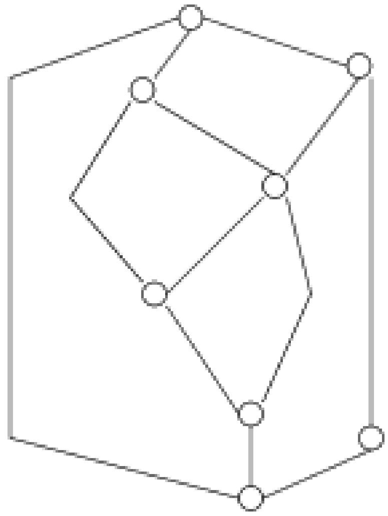

Figure 3.1: Polyline Drawing

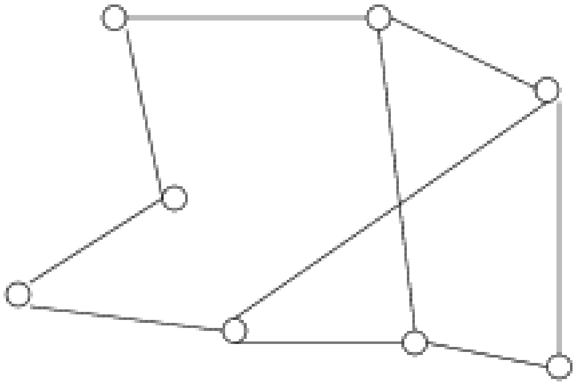

Figure 3.2: Straight line Drawing

Entity Relationship diagrams in data base design are usually drawn according to this standard. Note that polyline drawings can be modified to give drawings with nicely curved edges. A drawing is planar if no two edges intersect. A polyline drawing is a ***grid drawing*** if the vertices and the bends of the edges have integer coordinates.

A graph drawing algorithm reads as input a combinatorial description of a graph G, and produces as output a drawing of G according to a given graphic standard. The drawing is described in terms of graphics primitives such as draw line and fill circle, which can be interpreted on a physical graphics device. Within a graphic standard, a graph has infinitely many different drawings. However, in almost all data presentation applications, the useful-

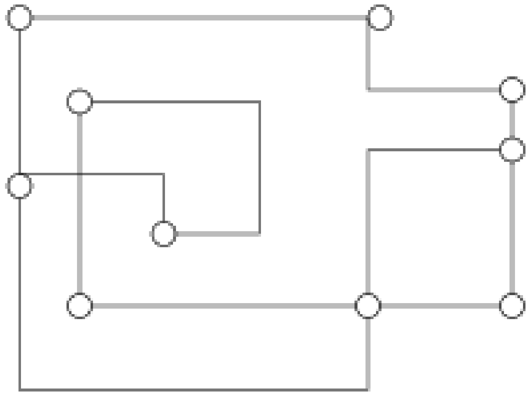

Figure 3.3: Orthogonal Drawing

ness of a drawing of a graph depends on its readability, that is, the capability of conveying the meaning of the diagram quickly and clearly. Readability issues are expressed by means of aesthetics, which can be formulated as optimization goals for the drawing algorithms. In general, the aesthetics depend on the graphic standard adopted and the particular class of graphs of interest. A fundamental and classical aesthetic is the **minimization of crossings between edges**.

Other aspects for graphs aesthetic can be achieved by:

- **Area:** Minimization of the area of the drawing.
- **Total Edge Length:** Minimization of the sum of the lengths of the edges.
- **Maximum Edge Length:** Minimization of the maximum length of an edge.
- **Uniform Edge length:** Minimization of the variance of the lengths of the edges.
- **Total Bends:** Minimization of the total bends of the edges.

...

In ***polyline drawings*** it is desirable to avoid bends in edges. In grid drawings, the area of the smallest rectangle covering the drawing should be minimal. In all graphic standards, the display of symmetries is desirable. It should be noted that aesthetics are subjective and may need to be tailored to suit personal preferences, traditions and culture. For example, although the cube graph is planar, it is traditionally drawn with crossing edges, as shown in figure 3.4.

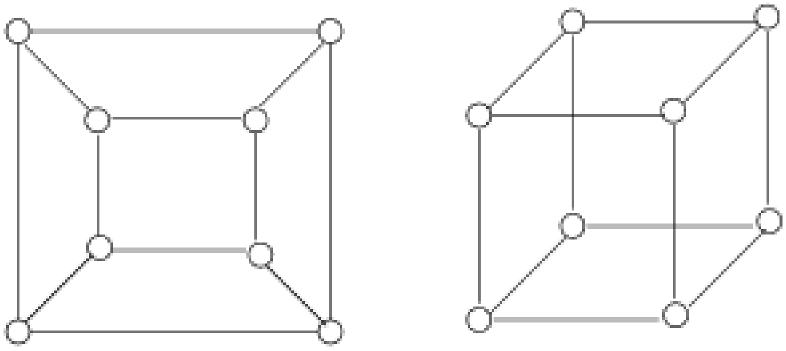

Figure 3.4: Two drawings of the cube graph

3.1.2 Rooted Trees

Rooted trees[2] are often used to represent hierarchies such as family trees, organization charts, and search trees. Planar straight–line drawings and orthogonal polyline drawings are commonly used to represent rooted trees see figure 3.5.

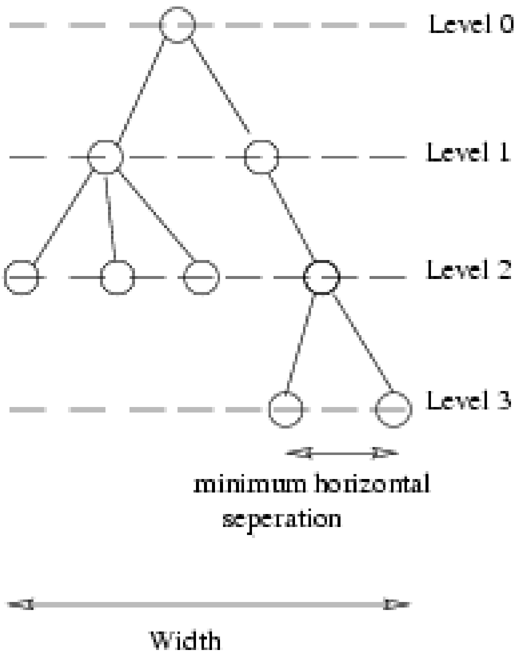

Figure 3.5: Drawings of a rooted tree in straight line and polyline orthogonal.

[2] ***Rooted trees are interesting layout for our conceptual grammar***

The following additional aesthetics are often adopted.

- Vertices are placed along horizontal lines according to their level (graphŋtheoretic distance from the root).
- There is a minimum separation distance between two consecutive vertices on the same level.
- The width of the drawing is as small as possible.

Further, for ordered binary trees such as search trees, it is required:

- left and right children of each vertex v are placed to the left and right of v, respectively.

3.1.3 General Graphs

There are several aesthetics for obtaining attractive drawings of general undirected graphs. The main such aesthetics are:

- Display symmetry.
- Avoid edge crossings.
- Avoid bends in edges.
- Keep edge lengths uniform.
- Distribute vertices uniformly.

3.1.4 Planarization

Most of the techniques for drawing general undirected graphs are heuristics. Given the wealth of techniques available for drawing planar graphs, a sensible strategy for drawing a non planar graph is to first planarize the graph, and then apply a planar graph drawing algorithm. In general, planarization seeks to transform a non planar graph into a planar graph with a small number of well defined operations.

The most common planarization operation is edge deletion: one must find a small number of edges whose deletion yields a planar graph. This is equivalent to finding a planar subgraph with a large number of edges. Finding a planar subgraph with a maximum number of edges is NP–hard. However, a maximal planar subgraph can be found efficiently.

Another planarization technique is to find a drawing with the minimum number of crossings. Again, this problem is NP–hard.

3.1.5 Planar Graphs

A graph is planar if it admits a planar drawing. Planar graphs play an important role in graph theory and graph algorithms.

Clearly, planar drawings are aesthetically desirable. Algorithms for drawing non planar graphs often begin by planarizing the graph, and then by applying a planar graph drawing algorithm.

Planarity Testing and Planar Representations

A planar representation is a data structure representing the combinatorial adjacencies between the faces of a planar drawing. Most planar graph drawing methods proceed as follows:

Step 1: Test planarity.

Step 2: (if the graph is planar) Construct a planar representation.

Step 3: Use the planar representation to draw the graph according to some graphic standard.

3.1.6 Visibility Representations

A visibility representation for a planar graph G consists of representing the vertices of G by horizontal segments, and the edges of G by vertical segments, so that the edgeŋsegment associated with each edge (u,v) intersects exactly the vertexŋsegments associated with u and v, and no other vertexŋsegment (see Figure 3.6).

In the following, we would like to introduce graph layout[3] for representing our **c**onceptual **g**rammar.

3.1.7 Acyclic Digraphs: Directed Graphs

Acyclic digraphs are widely used to display hierarchical structures. It is customary to represent these graphs so that the edges all flow in the same direction, e.g., from top to bottom, or from left to right (see Fig. 3.7. Namely, we say that a drawing of a digraph is upward if each arc is a curve monotonically increasing in the y–direction.

An important class of acyclic digraphs are covering digraphs of partially ordered sets. These digraphs are commonly represented by upward straightŋline drawings, called order diagrams, Hasse diagrams, or simply diagrams.

[3] ***After presenting Rooted trees.***

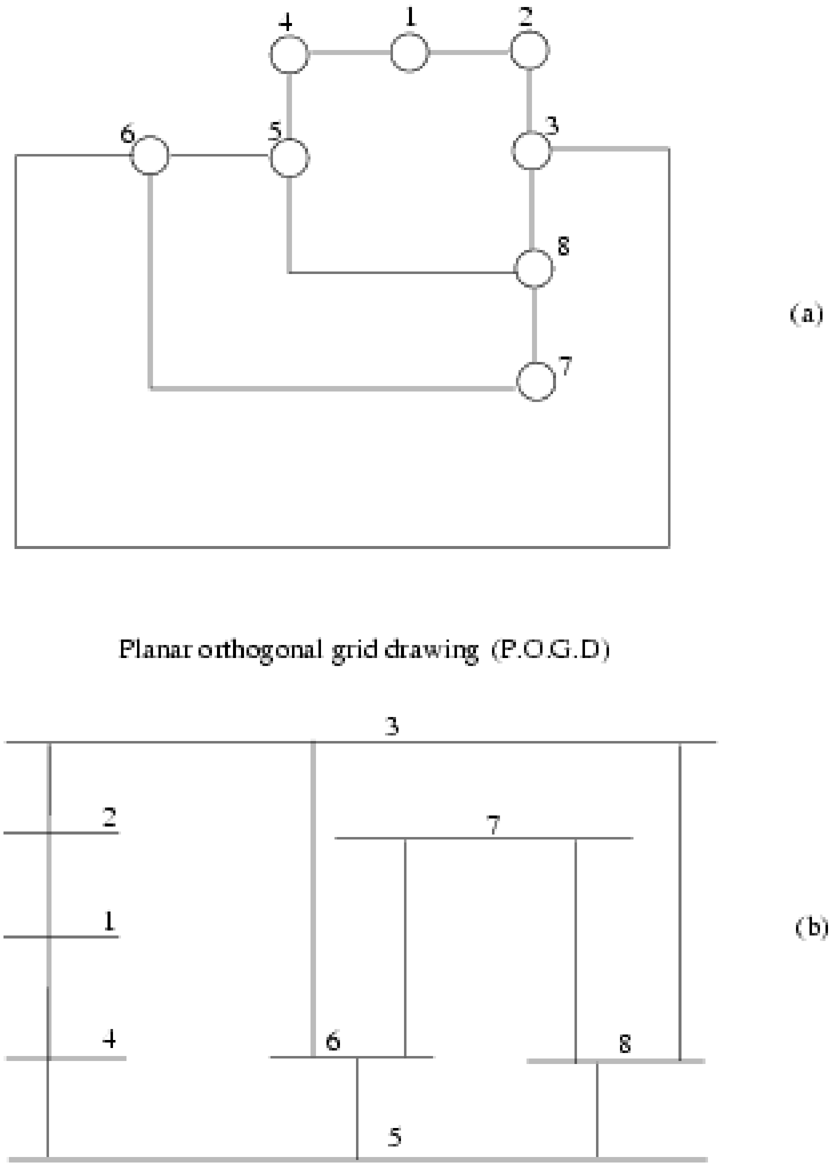

Figure 3.6: Visibility representation in (b) for the graph shown in (a)

3.1.8 Upward Planarity

The notion of planarity of undirected graphs has a corresponding notion of "upward planarity" for directed graphs. A digraph is said to be upward planar if it admits a planar upward drawing (see Fig. 3.7).

Note that an upward planar graph must be acyclic and its underlying undirected graph must be planar; however, there are planar acyclic digraphs

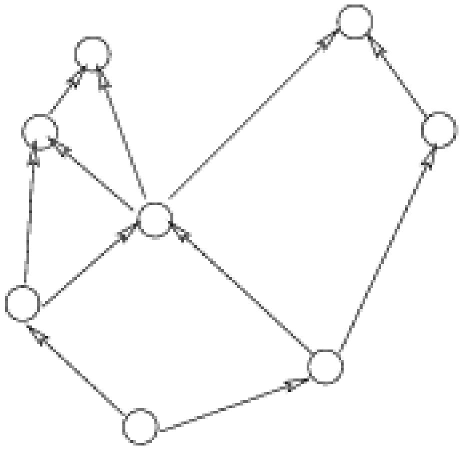

Figure 3.7: Planar Upward drawing of an acyclic digraph

which are not upward planar: see Fig. 3.8

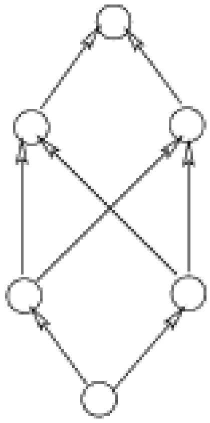

Figure 3.8: Planar acyclic digraph which is not upward planar

3.1.9 Hierarchical Drawings

A hierarchical drawing of an acyclic digraph is an upward polyline drawing where the vertices and bends are constrained to lie on a set of equally spaced horizontal lines, called layers (see Fig. 3.9). In some applications the assignment of vertices to layers is given, e.g., by the semantics of the graph. Such graphs are called layered digraphs, or hierarchies.

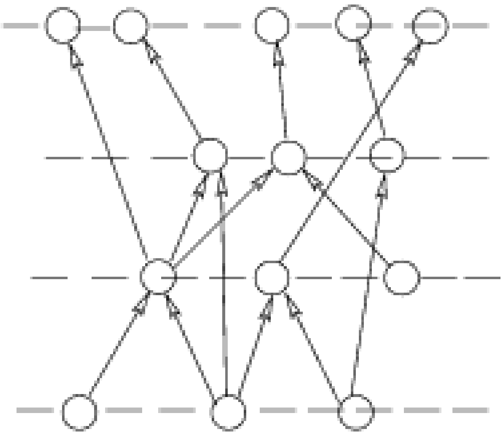

Figure 3.9: Hierarchical drawing

3.1.10 Conclusion

Scope Graph drawing addresses the problem of constructing geometric representations of abstract graphs and networks.

The automatic generation of drawings of graphs has important applications in key computer technologies such as software engineering, database design, and visual interfaces. Recent progress in algorithm design, computational geometry, topological graph theory, and order theory has considerably affected the evolution of this field.

In this book, many algorithms have been described to achieve visibility and aesthetic for graphs, in 2 dimensions.

But, the problem of displaying the large structure of the conceptual grammar, and earning more **"Space"** comes the idea for an extra dimension, i.e. presenting my graph/structure in 3 dimensions and for many other reasons proposed in the next section, where it has been detailed the advantages/disadvantages of 3 dimensions.

By choosing the extra dimension, a ***question*** rises in our mind:

Is it possible to apply visibility and aesthetic algorithms on our structure taken from the graph drawing book?

Answer: In 3 dimensions none of these algorithms can be applicable, because by trying these technics of minimization of the crossings edges, bends etc, on one specific 2 dimensions area, have no guaranty that these minimization are always achieved on another angle.

In the next section, we will discuss the *advantages and disadvantages of 3D*[4]

[4]Dimensions

compared to 2D.

3.2 Advantages of the 3D of software vizualisation

Although there are potential disadvantages, three dimensions offer numerous advantages over a traditional two dimensional style of visualization.

3.2.1 Potential Advantages of 3D

- The average distance between entities in 3D space would be less than in an equivalent 2D space. This is due to the fact that 2D diagrams are forced to be spread out, where as 3D has the potential to be more compact. As an example, consider a map of the world see figure[5] 3.10.

  The distance between countries on opposite sides of the map would be significant if the map were to be displayed in detail on a computer screen. A user would be forced to scroll for some time to move from one country to the other. Now imagine if this map were rolled up to make a cylinder. That is, the two most distant edges were brought together. By doing this the average distance between countries is reduced.

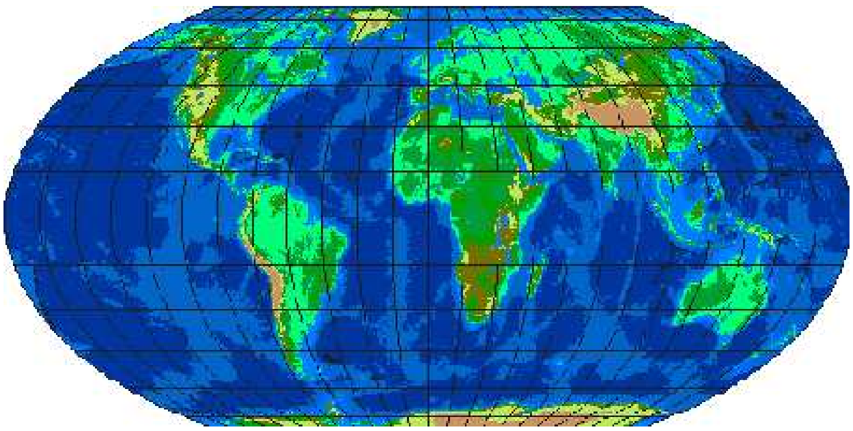

Figure 3.10: The world in sphere

The obvious downside to having a densely packed diagram is that it may become cluttered so some balance must be struck between spread and space saving.

[5]http://www.spatial.maine.edu/ucgis/testproc/mulcahy/mulcahy.html

- Links are less likely to cross (or come close) than in 2D diagrams. In a complex 2D diagram, in which entities may be linked to other entities to show relationships, it is often impossible to avoid the crossing of links. If this happens often enough diagrams can become very confusing and virtually unreadable. 3D diagrams avoid this potential problem entirely. There is no single plane that links are restricted to; links can travel in any direction. It is relatively rare for links to need to cross in a 3D diagram.

- 3D characteristics of entities can represent properties of underlying objects. For instance a spheroid may represent class X, a pyramid class Y, etc. Although a similar regime is possible in 2D, 3D allows a much greater scope, since the potential complexity of shapes is much greater. The greater the number of means of representation available, the more meaningful and **information dense** our diagrams become.

- The positions of objects in space may have relevance to properties. For example, objects below object X have property P, objects behind have property K, etc. In 2D diagrams, this is limited to above, below, left and right. 3D adds depth, allowing more information density. We can now connect meaning to an entity being behind another, or in front of it.

- 3D also offers more interesting options as far as actual viewing is concerned. Where as a user can scroll around a 2D diagram, they may wish to fly around a 3D view, or walk, or simply stay put and rotate the world around them. When a model is built, the most appropriate means of navigation will have to be considered. For small diagrams, the best solution may be to have the model surround the user, and allow them to study the entire world simply by spinning it around on some axes. For a large diagram there may be tiers of information, the top level may contain different information to the second level. Therefore it is feasible that the user may walk around the levels, and fly between them.

- The fact that a 3 dimensional visualization may impart a sense of **being there** to a user, can also be seen as an advantage. If the user feels as though they are traversing a virtual world, they are more likely to become immersed in the world, and be more receptive to details portrayed in the world.

3.2.2 Potential disadvantages of 3D

Potential disadvantages of 3D diagrams are conceivable.

- Firstly, the frame rate (the rate at which a browser displaying a virtual world can update its screen) must be considered as a potential disadvantage if it cannot be kept to a realistically high rate. Reasons for a low frame rate are predominantly technological, however, and must be counted as the major factor as to why little research has been done historically in the 3D field when 2D has sufficed. If a diagram is unpleasant for users to navigate then they are unlikely to gain anything from it. They are also unlikely to want to use it again. Fortunately, 3D on the average desktop is now quite feasible.

- Disorientation is another potential shortcoming of 3D. A user may become lost within a diagram, by facing in an inappropriate direction, or by moving too close to some object or too far from the model in general. Disorientation problems are preventable, however, with sensible model design. A user can be prevented from getting too far away from a model by encasing it in walls. Signposts can also be positioned to give the user an idea of where they are. Other more complex help aids may also be devised. For instance, one may imagine programmed help agents which appear to inform the user of their current location. Similar problems also exist in 2 dimensions - these are being addressed by such techniques as fisheye views[6]

- Text labels pose a problem in 3D. In a 2D diagram entities are labelled and there is no issue of the angle at which they are being viewed, or the distance at which they are being viewed (unless zooming is allowed). It is assumed that labels are readable no matter where a user should scroll too. However in 3D there are some important considerations. Should entity labels always face the user? Should they scale as the user moves towards or away from them? Should they appear only when the user comes within proximity? Label density is also an issue. If text is too densely packed it may obscure objects in the background or make the entire display confusing.

3.2.3 Potential 3D engines

There exist numerous tools, libraries, and languages which could be used to generate 3D worlds. like Java 3D.

- **Java3D:** There now exists for Java an API specially for 3D development. An advantage of this would be easy integration with other Java programs and portability across platforms, including the Internet.

[6]Graphical fisheyes views are popular techniques for focus+context. Fisheye views imitate the well–known fisheye lens effect, by enlarging an area of interest, and showing other portions of the image with successively less detail.

In the Chapter **Implementation using Java 3D API**, we will dig deeply in Java 3D, and gives all the advantages that explain why it has been chosen as our tool, to represent the structure of the ***c**onceptual **g**rammar.*

In the following, we will introduce visualization of programs *by considering the conceptual grammar as a program with many production rules, we want to visualize*. Further on, there will be a section about *graph visualization and navigation in information*, where we will discuss possibilities for 3D layouts and techniques i.e. **focus+context techniques**, used for zooming and retrieving information from the graph. Finally, we will end by a discussion on thoughts to represent the structure of the ***c*onceptual *g*rammar**.

3.3 Visualization of Programs

Software visualization (SV) has been around in many forms. It has been used for various purposes, common ones being program debugging and program structure visualization.

3.3.1 What is Visualization

Visualization is defined by the Oxford dictionary as the act or process of interpreting, in visual terms, or of putting into visible form. This is a very broad definition, non–specific to any particular discipline. Although it captures the essence of visualization, I would prefer a more scientific definition.

A taxonomy for classifying program visualizations, describes four axes along which visualizations can be classified. These axes are:

Scope: What aspect of the program is being visualized? For example, may be structure, inheritance, or methods.

Abstraction: To what degree is the visualization an abstraction of the underlying system. For example, a pretty printing is only a mild abstraction of the underlying system. Conversely, a flow diagram is a considerably more abstract means of representing a program.

Specification Method: How flexible is the visualization? Does it allow alteration of mappings to provide varying models, which may be advantageous to a user?

3.3.2 Why do we need Software Visualisation?

When approaching an unknown piece of software for the first time, human beings find anything but the most trivial software something of a challenge to comprehend.

Understanding software is a major problem in industry and research. It costs millions of dollars each year to employ people to analyse software for modification, alteration, bug fixing, and other tasks. Certain measures such as meaningful comments in code can go some way to remedying this problem. The recent rise in popularity of object orientation was supposed by many to be a solution to the problem of software understandability . OO[7] brought a richer model to programming. However, it is not a total solution to the problem.

Software is now too large and complex for a single human to digest in a practical amount of time. Traditional means of navigating software (e.g. text editors) have not aided the learning process. Procedural languages, such as C, have functions and data structures, and that is the extent of their structure. Simple diagrams could be produced showing what functions call and are called by other functions, but ultimately there was little high level complexity, compared to OO. OO brings with it many features designed to make program writing and maintenance simpler, and, in general, this is the case. We can make OO an even better tool though by using visualisation to learn about the high level structure of programs. The figure 3.11 is a proof how rich is the structure of OO architecture.

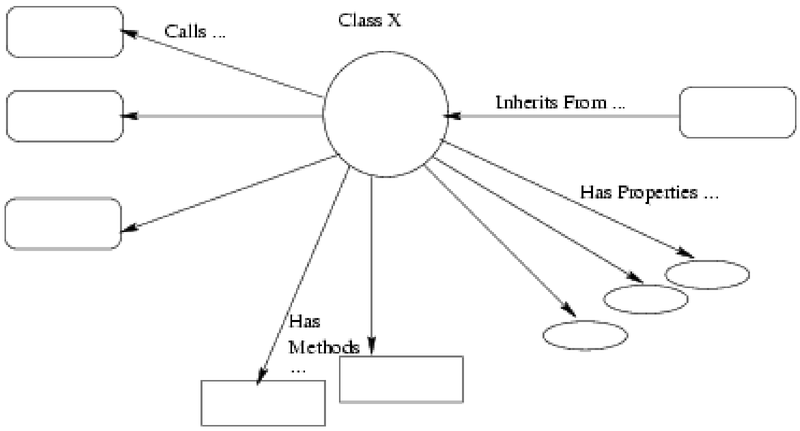

Figure 3.11: The Rich Structure of OO Architecture

Software visualisation should be considered, at its best, equivalent to having the software's original author available to discuss the program structure. That is, if a user new to a piece of software has some confusion over part of the software, or needs clarification of some process, the quickest solution would, in general, be to consult the original author. The goal of visualisation is to provide to a tool which to the user seams analogous to having the author available to focus on the particular part of a program the user wishes to know

[7]Object–Oriented

about, and perhaps to give less emphasis to parts which are irrelevant to the users current requirements.

The ultimate aim of visualisation is to speed up and improve the production and maintenance of software. Software is very expensive to create and maintain, so obviously the more efficiently these processes are, the better. Also, from a more academic perspective, development should progress faster if programs are easier to comprehend.

The type of visualization that I am dealing with here is static[8]. Dynamic visualizations are more likely to be used by a user who is already familiar with a system, but requires debugging or profiling information.

Noteworthy here is a definition of the term USER. This term has thus far been used rather loosely, and may conjure images of a single programmer dealing with an OO system. This is not the case; when USER is mentioned, it could entail any one or many persons involved in the development, design, or implementation roles of a system. It may also be the case that the job of creating visualizations is a specialised task, and is not handled by the person who will be analysing the visualizations at all.

Traditionally, visualizations as an aid to system understanding may have gone through multiple experts: The system designer requiring the visualization, the visualization designer, implementor and tester, then finally back to the original user.

After giving a general view about the visualization of programs, we can resume an important issue the simplicity and richness of object–oriented to be used for programming.

The reader has noticed, that OO[9] is a richer model to programming. In the next chapter *JAVA 3D API*, will let us decide to use *Java* as OO, for our implementation.

Anyway by considering our *conceptual grammar* as a program with an amount of production rules, what we are interested to visualize in it, is: **concept_ids and attribute_values** and nothing else.

Then, how to display them[10] in a graph?. This issue will be discussed in the next section.

[8]Only a static analysis of software systems shall be performed. That is, any information that can be derived from a system's source (or by reflection) may be visualized.

[9]Object–Oriented

[10]I.e. concept_ids and attribute_values

3.4 Graph Visualization

Graphs appear in numerous applications such as web browsing, state transition diagrams, and data structures. Large and abstract graphs often cause many problems, like how to visualize them? And how is it possible to navigate in the graph without being lost? Anyone who has been navigating in a large structure, experienced a few problems involved in graph visualization: "Where am I?", and "Where is what I'm looking for?"

The size of the graph to view is **a key issue** in graph visualization. Large graphs pose several difficult problems, because of the large of number of elements to view. This can compromise performance or even reach the limits of the viewing platform. Is it possible to view all the elements on one screen?.

After discussing, in the beginning of this chapter, about the graph drawing in 2 dimensions, now we are more interested to focus on 3D layouts.

3.4.1 3D layouts: Cone trees

The hope as we said previously, for an extra dimension would give literally, more "space", and that this would ease the problem of displaying large structures, let us search for a possible 3D layouts for our large structure.

Cone trees are ones of the best known 3D graph layout techniques in information visualization. They are also, an appealing *interactive 3D visualization technique for hierarchical data structures.* They were developed directly for 3D, instead of generalizing another 2D algorithm; Originally, they[11] were intended to **maximize** effective use of available **screen space**, and to better exploit the abilities of the human perceptual system.

Mathematically, the layout is **quite simple**. Nodes are placed at the apex of the cone with its children placed evenly along its base. Navigation in the cone tree by rotating it, is an important **visual aspect, that makes it very usable for a USER**[12] **.**

3.4.2 3D layouts: Hyperbolic Layout

The *hyperbolic layout* of graphs (mainly trees) is one of the new forms of graphs layouts which has been developed with *graph visualization and interaction in mind.*

Hyperbolic view, which can be implemented in either *2D* or *3D*, provide a distorted view of the tree. It resembles to the effect of using **fisheye** focus

[11] Cone trees

[12] We have given in the previous section, the definition of a **USER**

technique, that enlarge an area of interest on traditional layouts, and showing other portions of the image with successively less detail. This distortion helps to display large trees, specially nowadays structure becomes larger and hyperbolic view is suitable for this kind of application.

Even we talk about a big advantage previously, *hyperbolic views* are surrounded by a **sort of mystery**, because few people in the information visualization community really understand the mathematics of hyperbolic visualization. Another disadvantage is that hyperbolic settings, yields sometimes to quiet different results then what one expected, and furthermore, it is quiet difficult to reproduce the results. Unfortunately, none of the people in this community, i.e. information visualization community, are didactic enough to reveal the mystery.

- **Decision:** For all the reasons that have discussed above, we have decided to use the **simple** *cone tree* layout for our structure of conceptual grammar.

3.5 Navigation and Interaction

Navigation and **interaction** are essential to overcome the problems raised by large sizes of the graphs occuring in the visualization applications. They, i.e. navigation and interaction facilities, help us to reveal the *structure* of the graph.

3.5.1 Zoom and pan

To explore a large graph structure, zoom and pan[13] are traditional tools in visualization.

There is two forms of zooming:

- **Geometric zooming:** Geometric zooming provides a blow up of the graph content.

- **Semantic zooming:** Semantic zooming means that information content changes and more details are shown when approaching a particular area of the graph.

[13] We may say, also **match**.

3.5.2 Focus+Context Techniques

The term *focus+context* techniques do not replace *zoom and pan*, but rather complement them.

One of the most popular technique is the **Fisheye Distortion** for *focus+context*.

- **Fisheye Distortion:** Fisheye views imitate the well–known fisheye lens effect, by enlarging an area of interest in the graph, and retrieve details. We may compare it, i.e. *fisheye distortion* to a **loop**. An advantage of the fisheye technique is that it is independent of the layout algorithm and is defined as a separate processing step on the graphical layout of the graph. Interacting with fisheye, is done by changing the position of the focus point and/or modifying the distortion value.

For more details about all these techniques please refer to, "*Graph Visualization and Navigation in Information Visualization : a Survey, by **Ivan Herman, Guy Melancon, and M.Scott Marshall***".

The idea behind these techniques, will be implemented and discussed in the next chapter.

3.6 Visualization of Conceptual Grammar

In this section, we will try to express thoughts about how to visualize the graph for our **c**onceptual **g**ammar.

In visualization of our conceptual grammar, two structures has to be taken into consideration the *concept* and *attribute* structures.

Here we are talking about the design, i.e. the look of our display.

For example:

```
Occur ::= State[WRT:Stuff];

State ::= Symptom | Disease | missing[WRT: Stuff];

Stuff ::= Cvitamin | Bvitamin | Avitamin;
```

See figure 3.12:

In the figure 3.12, the concepts ids are displayed in circles and the black is a rectangle box that via attributes values links concepts between each other.

This was a first thought that comes into our mind, by displaying all the concepts objects randomly in the graph, without any consideration to any decisions taken previously, and make a black box where attributes values links concepts objects to each other.

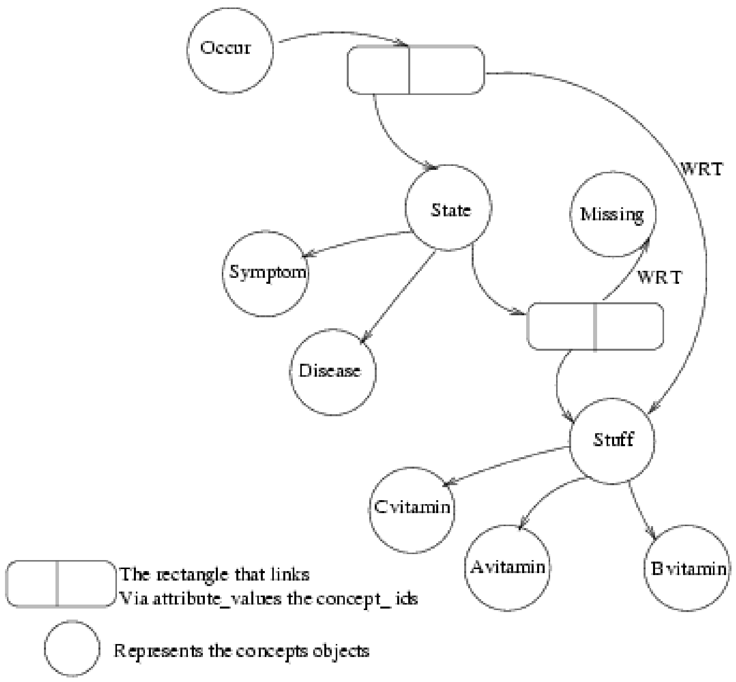

Figure 3.12: The Graph Layout for the Conceptual Grammar Structure

Now by taking into consideration of displaying the graph, using *cone tree.* The layout will completely be different from the previous figure.

How to display the new layout?

Conceptual grammar has an amount of productions rules. All production rules has rule_ids[14] on the left–hand–side of the *defining marker* '::=', will be placed at the apex of the cone with their children, i.e. all what is on the right–hand–side of the *defining marker* '::=', are placed evenly along the rules_ids base and will be displayed on a circle, if they are of the same **type**[15].

In figure 3.12, we used a rectangle to link all the concept_ids between each other. Now we will display a sphere for the same purpous as we used the rectangle box, but also to avoid repeating concept_ids or attribute_values.

As a result we come up with this structure, using the same example above, see figure 3.13:

In the next chapter we will discuss the implementation, by using *Java 3D API*. Further on there will be a testing of parser and the graph layout of the ***conceptual grammar***.

[14] *These* ***rule_ids are defined_ concept_ids.***
[15] simple_concept, attribute and concept type

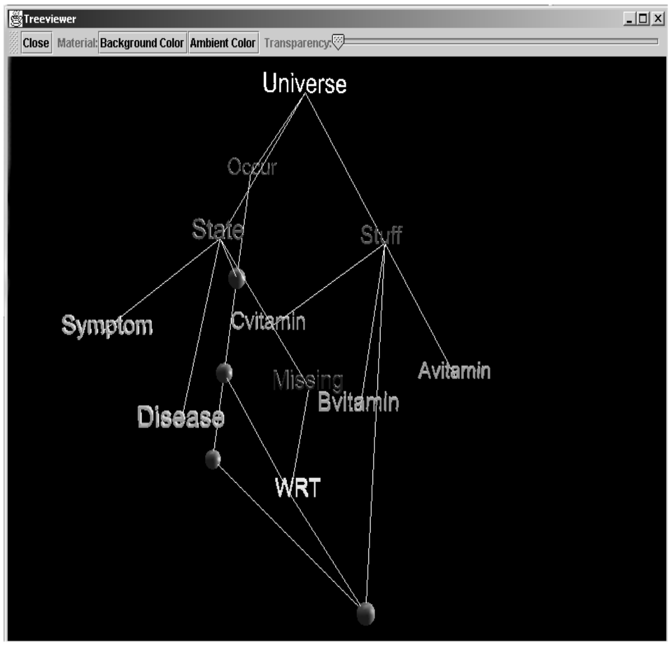

Figure 3.13: The new graph layout, using cone trees

Chapter 4

Implementation of ONTOVIZ system

In this chapter, we will discuss the implementation of the **ONTOVIZ** system.

4.1 An Overview

To get an overview of the entire application, here is a summary about our implementation.

- The **Cg class** first read the grammar accumulates it in a single string named *"s_grammar"*.

- The **Cg class** then creates a branch group (a *BranchGroup* instance) that will hold the entire grammar tree.

- Now it is time for the **Parser class** to step into action. It takes *"s_grammar"* and the *BranchGroup* instance (rootBranchGroup) as input arguments. The Parser adds nodes to the branch group while parsing the grammar.

- The nodes added by Parser are all instances of the **GrammarTree class**. One may note that the *GrammarTree.GrammarInsert(...)* method is used to create new instances of the **GrammarTree class**.

- When the **Parser** returns after creating the grammar tree the **Cg class** moves on to displaying the grammar tree. This is done by the following statement:

```
...
new MainFrame(new Ontoviz(rootBranchGroup), 800, 600);
...
```

I.e. an **Ontoviz** instance is placed on a window visible on the computer screen.

- From the **Ontoviz constructor** one may see that the branch group argument is stored in a instance variable only. The visualization process starts later on when the Java system calls the *Ontoviz.init()* method. It is important to notice that the **Ontoviz class** is an extension of the **JApplet class**.

  This is a base class in the so–called *Swing* or *JFC* (Java Foundation Classes) frame work which is different from the basic frame work in the **Java 3D API**. The Java 3D API is based on the now somewhat old *AWT classes* that don't allow for tooltips (pop–up help when the mouse cursor rests on some components) and other more advanced features. Since tooltip help texts are desired we have mixed the *Swing* and the *AWT* functionality[1]. This is not a simple issue but basically most problems can be avoided by making a slight change to the way that tooltips and pop–up menus are displayed in Swing. Two statements are all that's needed for this change:

  ```
  ...
      JPopupMenu.setDefaultLightWeightPopupEnabled(false);
      ToolTipManager.sharedInstance().setLightWeightPopupEnabled(false);
  ...
  ```

- Note that a *"canvas"* is the drawing surface in the **Ontoviz class**, and it occupies most of the window's area on the computer screen. In order to point out 3D objects with the mouse (so–called *"picking"*) we need the mouse cursor's coordinates. This is achieved by catching mouse clicks on the canvas through a call to the *addMouseListener(...)* method.

- The key points of getting the *grammar tree* onto the *canvas* really happens in the *createSceneGraph()* method where the branch group instance (*rootBranchGroup*) previously mentioned is added.

[1]The fundamental difference between *AWT* and *Swing* is that each visual *AWT component* have a Java representation matched with a parallel operating system equivalent component (a Window handle in Microsoft Windows), while visual *Swing components* are all drawn on a single operating system component.

Thus there is less processing involved in displaying Swing components, hence the nickname *"light weight components"*.

4.2 Implementation of the grammar of Conceptual Grammar

The first step after we defined the general form of our *conceptual grammar*, we have to try to extract the grammar of the *conceptual grammar* so we can construct the parser, scanner etc.

In chapter 2, in section 2.5, we have the general form for the conceptual grammar see below:

```
concept_ID::=
      concept_ID|...|concept_ID[attribute_VALUE:concept_ID]|...|
      concept_ID[attribute_VALUE:concept_ID, attribute_VALUE:concept_ID]|...|
      concept_ID[attribute_VALUE:concept_ID[attribute_VALUE:concept_ID[...]]]|...|
      concept_ID[attribute_VALUE:concept_ID,attribute_VALUE:
               concept_ID[attribute_VALUE:concept_ID[...]]]
      concept_ID[attribute_VALUE:concept_ID[...,attribute_VALUE:
               concept_ID[attribute_VALUE:concept_ID[...,...  ]]]];
```

By examining this general form of the conceptual grammar, we come up with its grammar defined as:

```
      Conceptual-Grammar ::= Rule { Rule }
      Rule ::= ID "::=" Concept { "|" Concept };
      Concept ::= ID | ID "[" Feature { Feature } "]"
      Feature ::= ID ":"  Concept
```

We note that ID is an identifier.

4.2.1 Wellformedness of the grammar.

We can try to describe informally the wellformedness of this grammar by the following:

- A conceptual grammar consists of a none empty sequence of rules.
- A rule/defined–concept has an identifier with a none empty sequence of concepts.

- A concept has an identifier and can be a defined–concept[2], a terminal–concept[3] or a concept–with-a–feature.

- A terminal–concept is none related to feature.

- A concept–with–a–feature includes features.

- A feature has an identifier and must be an element of a feature set. This feature set has been defined as a table in chapter 2, in section 2.6.

- A concept–with–a–feature must be defined as a rule, or defined–concept in the production rules of the conceptual grammar.

- A mutual recursion is allowed for concepts defined in relation with feature in a concept–with–a–feature.

- The set of terminal–concepts have to be disjoint with the set of defined–concepts or the set of rules, in the production rules of the conceptual grammar.

- The set of terminal–concept have to be disjoint with the set of concept–with–a–features, i.e only the set of concepts retrieved from the set the set of concept–with–a–features[4]

...

More formally, we try to redefine the grammar of *conceptual grammar* using ***RAISE*** specification:

type

CG :: rules : R^*,
R :: rname : Id
 concepts : C^*,
C = Id | IDF,
IDF :: fname : Id
 concept: C,
Id :: id : Identifier,
Identifier,

[2] We call it here Rule.

[3] We stop constructing in our graph, whenever a terminal–concept–identifier is being introduced in the productions rules.

[4] *They include also the set of features/attributes..*

```
        Features = ID-set

value
        is_wff: CG → Features → Bool
        is_wff(cg)(fs) ≡
                len(rules(cg)) ≥ 1
        ∧     (∀ r : R • r ∈ elems (rules(cg))
                                    ⇒
                        len(concepts(r)) ≥ 1
                )
        ∧

        ≡−wff: R → Bool
        is_wff(r)  ≡
                len(concepts(r)) ≥ 1
        ∧     (∀ c : C • c ∈ elems (concepts(cg))
                                    ⇒
                        is_wff(c)
                )

        is_wff : C → Features → Bool
        is_wff(c)(fs) ≡
                case c of
                        mk_Id(i) → mk_id(i) ∉ fs,
                        mk_IdF(id, fl) → len fl ≥ 1 ∧
                                    id ∉ fs ∧
                                    is_wff(fl)(Fs)
                end

        ...
```

We thought about using the PrintError class for checking wellformedness of our grammar, unfortunately this has not been achieved, but after the informal and formal descritption about wellformedness of our grammar, we think it will be easier to implement the PrintError class that can be included as a task for future work.

Here is the figure 4.1, where we include the stage of wellformedness.

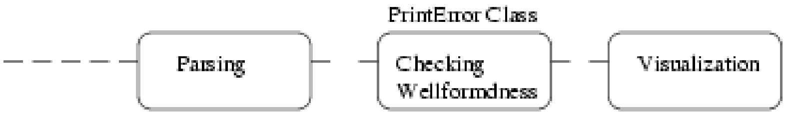

Figure 4.1: Wellformedness in the implementation

4.3 Implementation of the parser

After defining the grammar of *conceptual grammar*, the next step is to implement the parser.

Syntax: The way in which words are put together to form phrases, clauses, or sentences, By *Webster's Dictionary.*

Parsing: It is the process of finding the structure in the flat stream of *tokens.*

Parser: It is a module that performs this task, i.e **parsing**.

As a skeleton the paper, *Programming in C++ for those who know Java* of Hans Bruun, was a great inspiration for building my parser that is a top–down parsing with 1–symbol lookahead.

So we built a little state machine for our grammar that helps us to construct our parser, see figure 4.2

In the figure 4.2, *RuleId, RuleDef, ConceptId, Semicolon, Or, Leftbr, FeatureId, Colon, Comma and Rightbr* are all tokens.

Another issue is checking the bracket–counter, as you can see, in the same figure, a `brCounter` is used for displaying an error in parsing, in case of missing right–bracket .

As a result we parse the production rules one by one, we recognize the end of every rule by checking the semi–colon ";".

The lookahead symbol is called in Parser class as `nexttoken`. We used switch and case for parsing the production rules.

```
// handle one token at a time (tok) with look--ahead to the next token (nexttok)
while ((tok=sc.nextToken(false))!=Tokens.SEMICOLON)
 {
   nexttok=sc.nextToken(true);

   switch (tok)
    {
      case Tokens.ERROR:
```

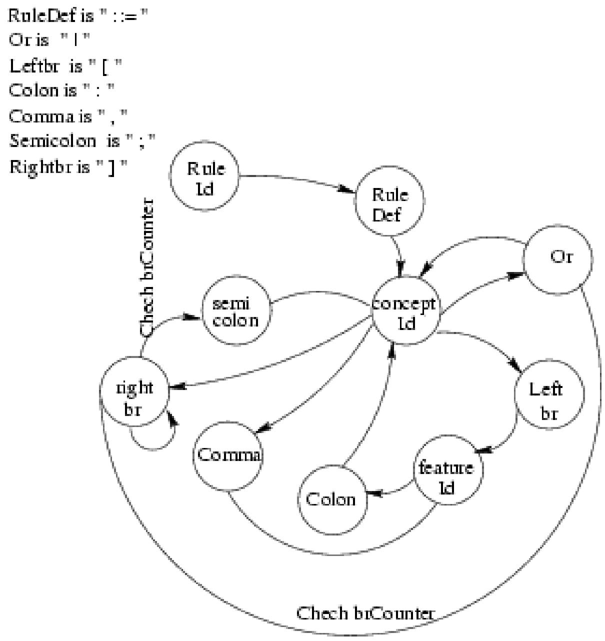

Figure 4.2: A little state machine for our parser

```
      errorMsg="Scanner Error !"; error=true;
      break;

case Tokens.ID:
switch (nexttok)
{
  case Tokens.DEFRULE :
  System.out.println("RuleId="+sc.getLexeme());
  currule=sc.getLexeme();
  insertv.addElement(sc.getLexeme());
  // insert the node in the tree
  gt = gt.GrammarInsert((Vector)insertv.clone(),GrammarTree.RULE,gt,null);
  break;
        ...
```

When we look at the code above, we can notice that we are calling a method *getLexeme(...)*. This method, is part of the *Scanner* class, it returns the tokens after scanning them, by calling another method *Lexeme(...)*.

4.4 Implementation of the Scanner

In our implementation of our scanner, that you can find in **Scanner** class, we break our source in the production rules into words and tokens.

In the figure 4.3, we define the structure of our scanner.

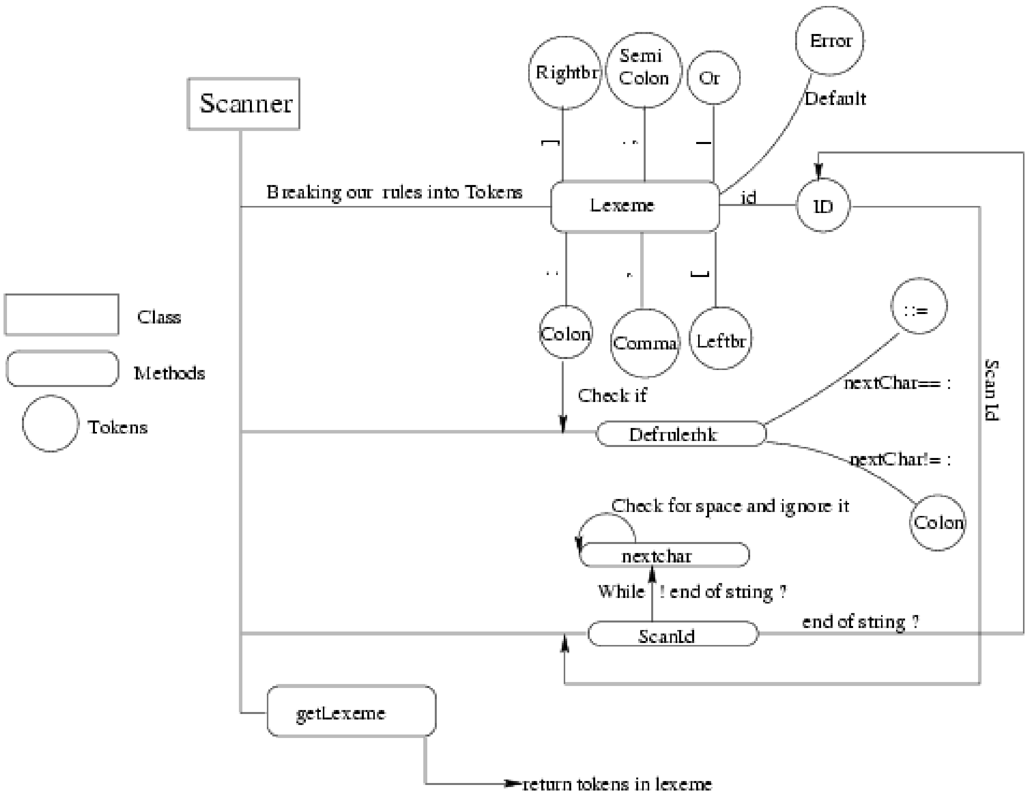

Figure 4.3: The Structure of the Scanner

The scanner includes five methods:

- **Lexeme method**: Its role is to break the source into words/tokens. If the source is an identifier then it will call **ScanId method**, else if the source is a colon ':' then it will call **Defrulechk method**. The default if the source is not recognizable by the scanner it will return an *error token.*

- **ScanId method**: It scans the identifier for rules, concepts, simpleconcepts, and features, and put it in one string and return as an *Id token.*

- **nextchar method**: Its role is to ignore all the space, this method is called by **ScanId method**.

- **getLexeme method**: is to initiate the process of scanning by calling **Lexeme method**.

- **Defrulechk method**: Its role is to check every time there is a *defrule token* '::=' or just a *colon token*, ':'.

In the next we will present **Java 3D** in our implementation.

4.5 Implementation Using Java 3D API

There now exists for Java an API specifically for 3D development. An advantage of this would be easy integration with other Java programs, and portability across platforms, including the Internet. A disadvantage is the speed of the 3D engine, since it is written in interpreted Java.

4.5.1 Creating Interactive 3D Scene Using Java 3D

The new Java 3D API can be used to create three–dimensional (3D) web applications. It also investigates the capabilities of Java 3D API and talks about its advantages and disadvantages.

Web is a growing information media that contains tremendous information sources. The number of web users also increases quickly. They obtain information and knowledge from web every day. Web designers also need to make their web sites more attractive, interactive and professional.

Interactive 3D environment and 3D animation is a good technique to make the web sites more attractive and interactive. 3D animation is more realistic than 2D animation which has been used widely in most web sites.

Interactive 3D environment is also very useful in some applications, such as education. For example, interactive 3D environments can be used by teachers to demonstrate solar system to students.

Geometry teachers can also use an interactive 3D environment to demonstrate the geometry theorems. There are several technologies that can create 3D environments and animations.

VRML: It is an acronym for Virtual Reality Modeling Language. VRML is analogous to HTML, but it can be used to create three–dimensional (3D) spaces, often referred to as "virtual worlds."

VRML first took shape in the spring of 1994, at the First International Conference on the World Wide Web in Geneva. More specifically, VRML is referred to as a "scene description language," because it tells a VRML viewer, or browser, how to construct a 3D environment. It is a lightweight and efficient way to create an online 3D virtual reality environment. However, VRML files can be quite large, and a special VRML viewer is needed, or browser to view the file. Also, compare to Java Language, VRML provides users with limited features.

Java AWT: Java AWT API can also be used to create 3D scene. However, Using Java AWT to create 3D application is very complicated and it is very hard to achieve good 3D effects.

Other plug–ins: Some 3D tools, such as Strata 3D, can also create online 3D scene. But users need special and non–standard plug–ins or viewers to view the 3D scene. And their features depend on the manufacturer of the tools. In order to provide web designers an easier and more powerful way to create online 3D scene and other 3D applications, Sun Microsystems Inc. built the Java 3D API that combines the 3D features and powerful Java language. Sun made first Java 3D 1.1 implementation (Alpha) available in1998. And it proves a natural choice for any Java programmer wanting to write an interactive 3D graphics program. **"Write once, view everywhere"** nature of the new Java 3D API means that an application or applet written using the Java 3D view model can be executed in any computer platform and render images to a broad range of display devices, including standard computer displays, multiple–projection display rooms, and head–mounted displays.

The Java 3D API can build a broad range of 3D graphics applications. Developers have already used Java 3D to build applications in a variety of domains including mechanical CAD, molecular visualization, scientific visualization, animation previews, geographic information systems, business graphics, 3D logos, and educational offerings.

Programming Paradigm

Java is an object–oriented programming language. So the programming in Java 3D is also based on objects. Applications construct individual graphics elements as separate objects and connect them together into a treelike structure called a scene graph.

The normal process of creating and executing a Java 3D program is that a programmer constructs a scene graph containing graphic objects, such as lights, sounds, environmental effects, and behaviours that handle interactions or modify other objects in the scene graph. Then the programmer can hand the scene graph to Java 3D for execution. Then Java 3D starts to render objects and execute behaviours in the scene graph. There are three kinds of rendering modes in Java 3D API: immediate mode, retained mode, and compiled retained mode.

Immediate mode renders scene graph using the high–speed Java 3D renderer. It is fast and shows the rendering result immediately. However, it leaves little room for global optimization at the scene graph level. It is fast but the quality is low.

In retained mode, Java programmers can construct a scene graph and specify which objects in the scene graph may change during rendering. Thus the renderer can render the scene graph using different rendering strategy for objects that may change during rendering and objects that do not change during rendering.

Compiled–Retained Mode assembles retained mode. Additionally, it can compile the scene graph into an internal format. It provides the highest performance. Java programmer can choose the rendering modes according to their needs.

4.5.2 Scene Graph

Scene graph is the basic data structure in Java 3D API. A scene graph is a tree structure that contains data arranged in a hierarchical manner. The scene graph structure consists of a set of nodes and leaves. The nodes have parent nodes, called Group nodes, which are used to control the way Java 3D process their children. Leaves are the objects that are displayed or used in the 3D scene. The leaves can be 3D shapes, lights or materials for 3D shapes.

The virtual universe is the root of the `"tree"`. It is a three–dimensional space with an associated set of objects. It serves as the largest unit of aggregate representation. Virtual universes can be very large and serve an application's entire needs.
As in figure[5] 4.4, locales are children of Virtual Universe, which use high–resolution coordinates to define a point in the virtual universe. In the high–resolution coordinate system, the origin is defined at the center of the screen. The X axis runs form left to right and the Y axis runs from bottom to top and the Z axis runs from inside of the screen to outside of the screen. X, Y and Z are 256–bit fixed point numbers, with the fixed point at bit 128. The value 1.0 is defined to be exactly 1 meter. Thus the coordinate system is sufficient to describe a universe of several hundred billion light years, while it can still define objects smaller than a proton.

Attached to each Locale, there are a set of Branch Groups. A Branch Group is the root of a sub–graph of a scene that may be compiled as a unit, attached to a virtual universe, or included as a child of a group node in another sub–graph. They contain the 3D objects to render, the lights objects to apply and the behaviour nodes to execute.

[5] http://java.sun.com/products/java–media/3D/forDevelopers/j3dguide/SceneGraphOverview.doc.html

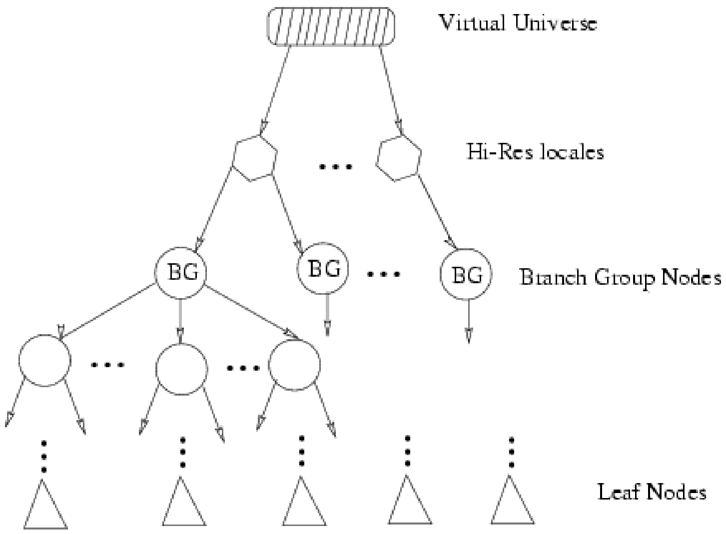

Figure 4.4: The scene graph structure

There are two kinds of Branch Groups: **the view branch and the content branch**.

The content branch contains nodes describing the actual objects that should be rendered and how the objects should behave interactively.

The view branch contains everything about how to view the 3D scene. In practice, the view branch is quite smaller than the content branch. The group nodes control the position and appearance of their children and the leaves are the actual objects to be shown. The method createSceneGraph, implemented in *Ontoviz Class* is used to construct the scene graph.

```
public BranchGroup createSceneGraph()
{

  // Create the root of the branch graph
   BranchGroup objRoot = new BranchGroup();

   TransformGroup objScale = new TransformGroup();
   Transform3D transform = new Transform3D();
   transform.setScale(bgScale);


   objScale.setTransform(transform);
   objScale.addChild(bg);
```

```
   ...

   BranchGroup scene = createSceneGraph();
   ...
}
```

After creating the scene graph we add all the children/leaf–nodes on the scene graph, *ambient, directional lights, the background etc.*

```
 // In the scene graph we add all the nodes like ambient light,
 // directional light etc.
 // Set up the background
 bgNode = new Background(bgColor);
 bgNode.setApplicationBounds(bounds);
 bgNode.setCapability(Background.ALLOW_COLOR_WRITE);
 objRoot.addChild(bgNode);

 // Set up the ambient light
 ambientLightNode = new AmbientLight(ambientColor);
 ambientLightNode.setCapability(Light.ALLOW_COLOR_WRITE);
 ambientLightNode.setInfluencingBounds(bounds);
 objRoot.addChild(ambientLightNode);

 // Set up the directional lights


 Color3f light1Color = new Color3f(0.3f, 0.3f, 0.4f);
 Vector3f light1Direction  = new Vector3f(4.0f, -7.0f, -12.0f);


 Color3f light2Color = new Color3f(0.7f, 0.7f, 0.8f);
 Vector3f light2Direction  = new Vector3f(-6.0f, -2.0f, -1.0f);

 light1 = new DirectionalLight(light1Color, light1Direction);
 light1.setInfluencingBounds(bounds);
 objRoot.addChild(light1);

 light2 = new DirectionalLight(light2Color, light2Direction);
 light2.setInfluencingBounds(bounds);
 objRoot.addChild(light2);
...
```

We will later on explain the *ambient light, directional light.*

The Canvas3D object, inherited from Java AWTŠs Canvas class, is the final component object that is a surface on which the scene graph is rendered. The Canvas3D object is usually the leaf node in the view branch.

```
//In Ontoviz class
private Canvas3D canvas;

...
canvas = new Canvas3D(SimpleUniverse.getPreferredConfiguration());
content.add(canvas, BorderLayout.CENTER);
...
```

Shape3D: The Shape3D leaf node specifies all geometric objects. It contains a list of one or more Geometry objects and a single Appearance object. The geometry objects define the shape node's geometric data. The appearance object specifies that object's appearance attributes, including color, material, texture, and so on.

But we notice that, when we rotate the *Ontoviz* tree, all the nodes in the graph structure rotate in all different directions. So we used instead **OrientedShape3D** leaf node that inherits from **Shape3D**. The extra advantage is that we can control the nodes in the tree. It is reasonable that the nodes will always face the user, so he/she will not be confused.

Appearance: The appearance object defines the Shape3D object's appearance attributes, including color, material, texture, rendering, etc. In the NodeText class the appearance of the textShape is defined:

```
...
NodeShape textShape = new NodeShape(tree);
// put it all together
textShape.setGeometry(txt);
textShape.setAppearance(apText);

textShape.setAlignmentMode(OrientedShape3D.ROTATE_ABOUT_POINT);
textShape.setRotationPoint(p);
...
```

In the *NodeText* class, we define the color of the nodes, i.e. *rule_id* in *red*, concept_ids in *blue*, *simple–concept_ids* in *green* and *attribute_ids* in *blue*. We include also the shape , font of all these nodes that will be inserted in the tree displayed on the 3D Canvas.

```
...
 // set the text color
```

```
  switch(type)
   {
    case GrammarTree.RULE : nodecolor=new Color3f(1.0f, 0.0f, 0.0f);break;
    case GrammarTree.CONCEPT: nodecolor=new Color3f(0.0f, 0.0f, 1.0f);break;
    case GrammarTree.SIMPLECONCEPT:nodecolor=new Color3f(0.0f, 1.0f, 0.0f);break;
    case GrammarTree.FEATURE:nodecolor=new Color3f(1.0f, 1.0f, 0.0f);break;
    default : nodecolor=new Color3f(1.0f, 1.0f, 1.0f);
       }
 ...
```

Material: The Material object defines the appearance of an object under illumination. The properties of the material are ambient color, diffuse color, specular color, emissive color and shininess.

Ambient color: is the color of the ambient light reflected off the surface of the material.

Diffuse color: is the color of the material when illuminated. Specular color is the color of the high–lighted part of the material.

Emissive color: is the color of the light that the material emits itself.

Each of the colors are defined as RGB colors and each color channel is defined by a variable of type float in the range [0.0, 1.0]. Shininess defines the material's shininess with a float in the range [1.0, 128.0] with 1.0 being not shiny and 128.0 being all shiny.

All what we have defined above is included in *ToolBox* class.

A default material has been created.

```
objMat = new Material();
objMat.setLightingEnable(true);
. . .
app.setMaterial(objMat);
```

Light: The Light class is a super class in Java 3D API. There are several kinds of light classes inherited from the super class.

An AmbientLight object defines an ambient light source, which defines the light all over the 3D universe.

A DirectionalLight object defines an oriented light with an origin at infinity so that the light rays are parallel to one another. It is good for simulating natural sunlight.
A PointLight object defines a point light source located at some point in space and radiating light in all directions.

A SpotLight object defines a point light source located at some point in space and radiating in a specific direction.

Before creating the light objects, it is important to create a Bound object to define the influence area of the lights.

```
BoundingSphere bounds =
new BoundingSphere(new Point3d(0.0, 0.0, 0.0), 10000.0);
```

In *Ontoviz* class, we create a sphere shape area in the virtual universe with the center at the origin and radius 10000.

4.5.3 Interactions

To interact with the Java 3D scene or make animations, the Java programmers can add the behaviour node to the scene graph.

The behaviour nodes provide the means for animating objects, processing keyboard and mouse inputs, reacting to movement, and enabling and processing pick events (picking objects by mouse clicks, for instance). A Behaviour object contains a scheduling region and two methods: an initialize method called once when the behaviour is activated and a process stimulus method processing the behaviour messages and called whenever appropriate by the Java 3D behaviour scheduler. The scheduling region defines the scheduling of behaviour nodes.

Interpolator Behaviour objects are used to create animations.

Three most interesting behaviour objects are: **MouseRotate, MouseZoom** ***This object behaviour can be considered as Focus+Context technique and ressemble to the Fisheye Distortion*** **and MouseTranslate**. They are used to react to the user's mouse operations and define the behaviour for those operations, such as rotation, zooming and movement. It is also important to define the Bound object before define those behaviours. The Bound object will be used to define the scheduling region of the behaviour.

```
...
 // add orbit behaviour to the viewing platform
 OrbitBehavior orbit = new OrbitBehavior(canvas, OrbitBehavior.REVERSE_ALL);
 BoundingSphere bounds =
new BoundingSphere(new Point3d(0.0, 0.0, 0.0), 10000.0);
 orbit.setSchedulingBounds(bounds);
 viewingPlatform.setViewPlatformBehavior(orbit);
```

...

4.5.4 Information Zooming, using PickRay

In chapter 3, we talked about *Zoom and pan technique* in section 3.5.1, the idea behind pan and zooming specially *Semantic zooming* is introduced in the following, where *PickRay* allows us to pick up information from our graphical representation.

Introduction: Picking an object is the combination of two simple actions: handling a mouse click or movement and mapping a 2D screen coordinate into the 3D world. The last part is just the opposite of the 3D rendering process, but a lot more work needs to be done by the user rather than being automated.

One Click, Many Objects

Say we have the situation illustrated in figure 4.5. The user clicks on the screen at point A. The object that is picked is very easy to determine as there is only one along the projected vector indicated by the dashed line (note that the drawing does not indicate the effects of perspective). A simple casting of the vector into the scene graph will reveal what object has been selected and our pick system will return a reference to it.

On the other hand, consider what happens when the user clicks at point B. Now there are three objects that could have been picked. What should the system return to the user when asked? It depends really on what the application requires. Sometimes we only want the closest object, other times we might require all of them. If we want all of them we may want it sorted by depth or not care too. All of these options are provided by the Java3D API, so we should not have any problems building the type of user interface that we need.

Nominating Pickable Objects

The act of picking requires a reasonably large amount of calculation to be performed for every object. Worlds that have lots of objects could conceivably spend most of their time just calculating picking information. Obviously this isn't a desirable outcome, so the Java3D API makes some large optimisations. Anything that is not pickable can be discarded. Although the Shape3D object is pickable by default.

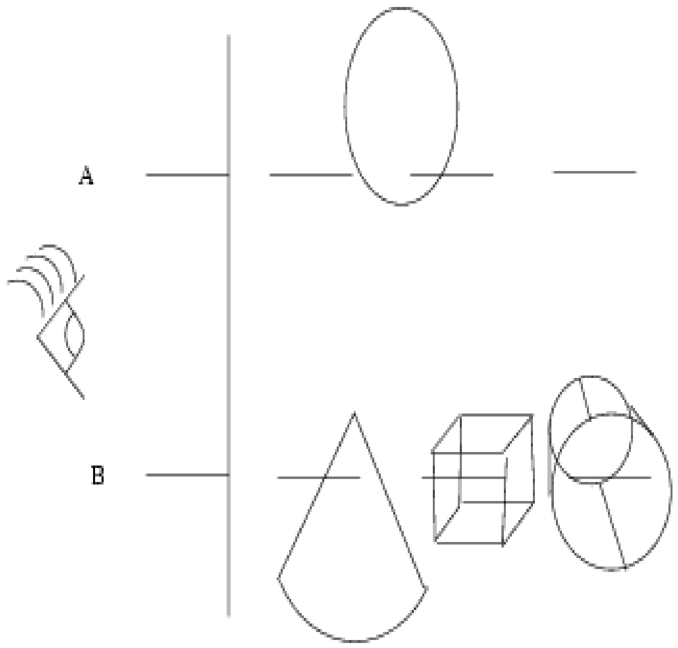

Figure 4.5: Different forms of picking objects in a 3D world.

Setting the pick state is through the use of the **setPickable()** method of the Node class. That allows any scene graph node to be pickable. Because picking is like other optimisable parts of the scene graph, it also requires the use of the capabilities:

ALLOW_PICKABLE_WRITE: allows the user to change the pick state on the fly. We might use this if the object is only sometimes pickable. This only needs to be set if we are changing the pick state on the fly, not when first constructing a dead/uncompiled scene graph.

ALLOW_PICKABLE_READ: allows the user to read what the current pick state of an object is. Less likely to be used than the write capability, but generally useful when dealing with large numbers of anonymous objects.

ENABLE_PICK_REPORTING: is used only by grouping nodes. Because grouping nodes don't define a visible object on the screen, normally they are tossed aside in the picking process. By enabling this option, when we get the path to the picked object, it will also include the enabled group (but not every group in the path) in that path.

The capabilities are only needed if we wish to change the pick state at run–time. If we don't need to change or know what the state is, then all we need to do is call **setPickable(true)** on the object at creation time and then forget about it.

Describing Picked Objects

Because Java3D does a lot of internal optimisation of our scene graph when it gets compiled, our representation and the internal representation may not even look vaguely similar. This can cause a lot of problems when trying to access the node of the scene graph that was picked. To overcome this problem, all path information is represented as a **SceneGraphPath object**. This abstract representation is used to hide the internal representation from the external code and make it easy to deal with picking.

When picking an object, typically we are not interested in the entire scene graph tree to that object, just the actual object that was picked. Since the object is a visible item, it has no children and the end of the path is the picked object. The picked item can never be an item in the middle of the path.

The **SceneGraphPath** class can be very useful for dealing with picked objects. As mentioned above, we usually don't need to know anything more than the object that was actually picked. However, for those that do, there is quite a bit of useful information available through the following methods:

getObject(): returns the instance of the object that was picked.

getNode(): method allows us to step through all of the objects in the path from the root locale to the clicked object. If we have not enabled pick reporting for any of the groups (like transforms) then we end up with a path that contains a single element – the picked object. We may want to use this in conjunction with the nodeCount() method that will return the number of objects in the path.

getTransform(): returns the location of the object in world coordinates at the time the object was picked. This is not effected by the setting of the **ALLOW_LOCAL_TO_VWORLD_READ** capability of the picked node. Note that if the object is being animated that the transform may have changed by the time we get to actually process some of the data.

isSamePath(): can be used to see if two paths represent the same object. For example, if we have picked the same object twice in a row during an animation, the path is the same, but a number of other parts like the transforms are different. A pure equals comparison would return false because the transforms are different, when really it is the same object that has been picked.

Pick Mouse behaviour

Assuming that the scene graph consists of a number of pickable objects, we now need to do something with them. The typical use is with the mouse by clicking on it. When the mouse is clicked, we'll return a SceneGraphPath to the selected object.

The pick behaviour is derived from the mouse click behaviour. Depending on what we want for pick behaviour, a click may not be the appropriate input. For example, a drag to move object would really need to combine picking with the mouse drag behaviour. For this example the simple click on an object notification is all that we need to illustrate the basic principles.

When picking an object, we have to get hold of the mouse position, this has been implemented in our code as the following:

```
canvas.addMouseListener(new MouseAdapter()
{
  public void mouseClicked(MouseEvent ev)
   {
     clickAction(ev.getX(), ev.getY());

   }
});
```

With the mouse point available, we now need to cast that into the scene as a 3D ray from the user's position. Note that we say a ray, not a vector. By definition, a vector is infinitely long that has no beginning and no end. Thus, if we used a vector in the scene, clicking on the screen would also select objects behind us as well. Even though the matrix math is exactly the same, we stick to the more precise mathematical description of a ray, which is a starting point and an infinitely long line in one direction only. In this case "outwards" from the viewer's position in the direction they are looking.

We need to transform a mouse position to a ray but it is not exactly obvious with Java3D. Here is the code that we need to use to turn a mouse position into a ray:

```
 /** Get the 3D ray (from user's eye to infinity) at the mouse position
 */
private PickRay createPickRay(Canvas3D canvas, int x, int y)
{
 Point3d eye_pos = new Point3d();
 Point3d mouse_pos = new Point3d();

 canvas.getCenterEyeInImagePlate(eye_pos);
 canvas.getPixelLocationInImagePlate(x, y, mouse_pos);

 Transform3D motion = new Transform3D();
```

```
 canvas.getImagePlateToVworld(motion);
 motion.transform(eye_pos);
 motion.transform(mouse_pos);

 Vector3d direction = new Vector3d(mouse_pos);
 direction.sub(eye_pos);

 return new PickRay(eye_pos, direction);
} // end createPickRay
```

After establishing a couple of points for eye and mouse positions, the first thing that must be done is to establish where our mouse is in 3D space and where the user is in 2D space. In both of these queries, the Point3d object is passed in and the values placed in them. Note that we go through this third party object called an image plate. This object is used as the interface between the 3D renderer and the 3D code.

Having established the basic renderer specific information the next task is to convert both the mouse and eye position into real virtual world coordinates (forming the next 4 lines of code). The two transform lines are used to modify the image plate coordinate systems into the virtual world coordinate system.

Finally, we need to establish a direction based on the user's eye position and the position of the mouse. By subtracting the eye position from the mouse position, we end up with the direction of the pick ray. The last line is the final point of constructing the PickRay object from the eye position and direction.

Once the pick ray is established we need to find what this ray intersects with in the scene. There are two places we can do this – **Locale** and **BranchGroup**. The **BranchGroup** is located in our implementation in *Ontoviz* class, the method implemented that allows us to pick objects, is defined as :

```
/** This method controls what will happen when the user clicks
    on the canvas
*/
private void clickAction(int x, int y)
{
  PickRay ray = createPickRay(canvas, x, y);

   //OBS: need to fix, we don't get the right node
  SceneGraphPath path = bg.pickAny(ray);
```

```
Node node;
...
String message = "Node=" + node +"  key=" + tree.getKey()
                              + "  type=" + tree.getTypeString();
```

As a result we decided as a test to get the key/id of the picked object and its type, i.e concept–id, simple–concept–id, feature–id, rule–id and an unknown type as a default.

A problem occur, when we click on our structure. We always get *Universe*[6] as a key and type unknown, see figure 4.6.

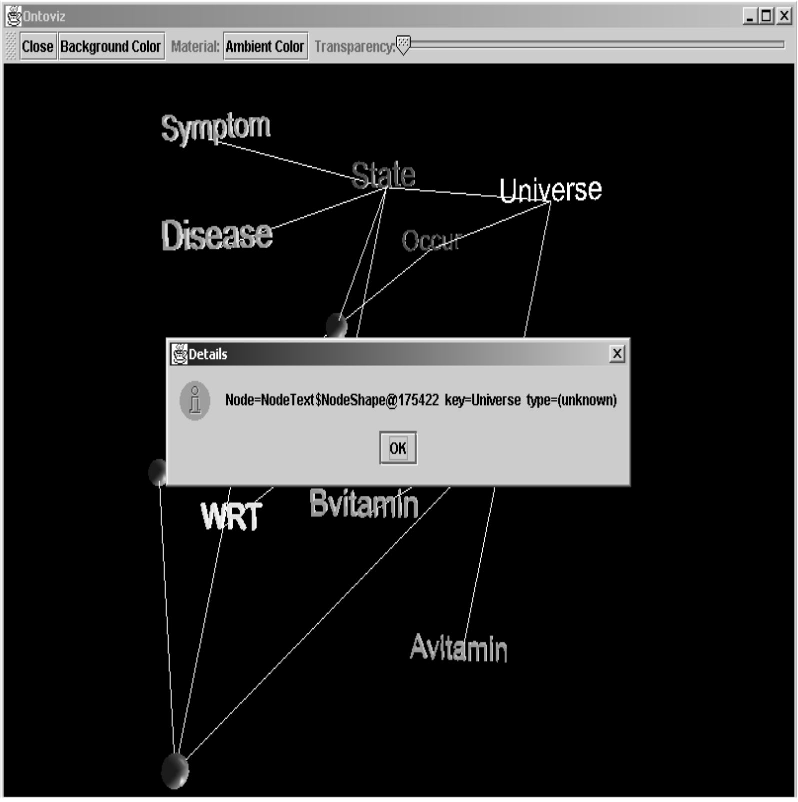

Figure 4.6: Information Zooming, using PickRay

So are there problems to be fixed. This can be included as future work. The problems seems to be in the **createPickRay(...)** method.

Another use of *picking* is for **collision detection**. Construct a bound around our object describing the *collidable area*. Then, on each frame check to see if we can pick anything within that area. If yes, then we have collided with

[6]Universe is the top of our *Ontoviz tree*

something and need to react. If there are no picked objects then keep going as we were before.

4.5.5 Detecting Collision Per Object

The most simple collision system is based on object to object collision. The default implementation of the Node object sets the collision system on by default (just like picking).

Based on this, whenever one piece of geometry collides with another, a behaviour could be executed to deal with the results. Simple or complex interactions can then take place.

Because collisions notifications are delivered by the behaviour mechanism, unless we register a behaviour, we will never know if two objects collide. That is, the default implementation is just to issue a message to say that two objects have collided, but do nothing about it. It is possible for two objects to pass straight through each other and the rendering engine will not do anything about it.

There are three types of notification that the system will deliver for collisions:

- **WakeupOnCollisionEntry:** When two objects first collide this condition is executed. We might use this to stop the movement of one object through another. One point of this is that we can construct a bound that is just outside the object so that as collision occurs we can prevent situations where it appears that the object moves just in and out of collision with the other.

- **WakeupOnCollisionExit:** When two objects stop colliding this condition is executed. This uses similar conditions to the entry condition allowing a bounding area to work.

- **WakeupOnCollisionMovement:** Once two objects have started colliding, as they move relative to each other, this criteria is executed until the exit condition.

To end our discussion about picking, we will introduce in the next section the limitations of picking.

4.5.6 Picking Limitations

Despite all the good points about the use of picking, there are a number of limitations.

The first limitation is:*If we can't see it, we can't pick it.* Back faces or transparent objects cannot be picked. If we have a completely transparent object, such as a window, attempting to pick it will result in nothing being returned.

The second, and most difficult limitation is that, by default we cannot get the point information about where on an object we picked it. That is, we only get the whole object and no further information.

In the next section we will talk about the advantages/disadvantages of **Java 3D API**, to explain to the reader why it has been chosen.

4.6 Advantages of Java 3D API

The advantages of Java 3D API are:

- **Platform independent:** This is also the feature of Java. Web designers and Java programmers can write their code and make their code run on different systems.

- **Flexible:** The scene graph model is very flexible to fit different applications. The programmer creates objects containing geometric data, attribute information and viewing information, which are connected through scene graph. Node and Node component derived from the SceneGraphObject class.

  The instances of the subclasses of the Group and Leaf classes will form the scene graph and will contain references to instances of the subclasses of the Node component. A leaf node's state is defined by the nodes in a direct path between the scene graph's root and the leaf. Because a leaf's graphics context only relies on a linear path between the root and that node, the Java 3D renderer can decide to traverse the scene graph in whatever orders it wishes. It can traverse the scene graph from left to right and top to bottom, in level order from right to left, or even in parallel. The only exceptions to this rule are spatially bounded attributes such as lights and fog. In addition the sub–graphs can be shared and cloned.

- **Object–Oriented:** Java 3D API uses the scene–graph model to provide web designers and programmers a high–level, object–oriented view of 3D graphics. Java 3D allows the programmer to think about geometric objects rather than about triangles about the scene and its composition rather than about how to write the rendering code for efficiently displaying the scene. This approach is especially good for programmers without much graphics or multimedia programming experience. It is straightforward enough for any experienced Java programmer to learn.

- **Optimized for 3D graphics:** The Java 3D runtime uses the fastest possible renderer to render the 3D scene graph. It may affect the quality of 3D display. So Java 3D API is better for interactive 3D environments than for high–quality graphics applications.

- **Import 3D objects:** A large and growing number of 3D loaders are available to import content into the Java 3D runtime. Thus Java 3D API is capable of loading 3D objects created by other 3D creators. This approach provides programmers an option to create professional 3D objects outside of Java 3D and then display them using Java 3D.

4.7 Disadvantages of Java 3D API

There are also some problems of the current implementation of Java 3D API. In the current implementation, Java 3D is mixing a lot of Java code with OpenGL or Direct3D calls. However, many of the existent Windows machines emulate Direct3D and OpenGL with pure software drivers. Thus, actually, Java 3D programs are slow. when users rotate the *Ontoviz* tree, there is apparent flicking, which degrades the quality of the 3D graphics. Another problem is that Java 3D API is a standard extension API and it has severe availability constraints. Now the only major vendor providing a Java 3D implementation is Sun. Also, the users have to install proper Java 3D runtime to run the 3D application in their computers. Thus the cross–platform portability of Java 3D code looks questionable. Also Java 3D programmers have problems to control low level rendering processes. Java 3D hides rendering–pipeline details from programmers. It is because Java 3D is a high–level API. It intentionally hides details of the rendering from the Java programmers. So the Java 3D API is not suitable for the application in which quality of graphics is important and where Java programmers need a customized rendering process.

Finally, Java 3D has no file format. So the applications have to define their own graphics file formats. So the future implementations of Java 3D API should contain much more native optimizations to improve the performance of Java 3D. And Java 3D API could be incorporated in the standard Java APIs, which allows Java 3D applets to run under regular Java web browsers.

4.8 Conclusion

Java 3D is an extension of the Java platform with advanced capabilities for three–dimensional graphics. Though the Java 3D API is an easy to use but powerful API to create interactive 3D applications.

In the future, Java 3D API may be very successful in the education market. Online educational web sites can use Java 3D API to create interactive 3D scenes quickly. And Java 3D programs can use the powerful features of the Java language to access databases, parse XML, etc.

In chapter 5, we will discuss the testing of our **parser** and the **Ontoviz system**.

Chapter 5

Testing

This chapter includes two parts:

- Testing our parser.
- Testing the Ontoviz system.

5.1 Testing the Parser

For testing our parser we took this example:

```
Occur ::= State[WRT:Stuff];
State ::= Symptom | Disease | Missing[WRT: Stuff];
Stuff ::= Cviatmin | Bvitamin | Avitamin;
```

The output after testing the compiler is under the name `Y.res`, where we include in the file, the test and the result of this test. You can find all theses output in the appendix under section B.

Tests
1: Occur := State [WRT:Stuff];
2: Occur ::=State [WRT:];
3: State :: = Symptom | Disease | [WRT:Stuff];
4: State :: = Symptom | | Missing WRT:Stuff];
5: Occur ::= State [:Stuff];
6: :: = Symptom | Disease | Missing[WRT:Stuff];
7: State :: = Symptom | Disease | Missing WRT:Stuff];
8: State :: = Symptom | Disease | Missing [WRT:Stuff;
9: Stuff ::= Cvitamin | Bvitamin | Avitamin

Table 5.1: List of the Tests

The result of the nine tests introduced in the table 5.1, are shown in table 5.2.

Result of Testing
1: Error Parsing : Syntax error.
2: Error Parsing Occur : Missing ConceptId after ':'.
3: Error Parsing State : Missing ConceptId/SimpleConceptId after '|'
4: Error Parsing State : Missing ConceptId/SimpleConceptId after '|'
5: Error Parsing Occur : Missing FearureId after '['.
6: Error Parsing Occur : Missing RuleId.
7: Error Parsing Occur : Syntax error.
8: Error Parsing Occur : Missing ']'.
9: Missing last ';' The rule will be not taken in consideration

Table 5.2: Result of Testing

The results in table in 5.2, shows that our parser is working.

The parser can be improved, see chapter 6, in section 6.2, where it has been included for future work.

5.2 Testing Ontoviz System

Three major problems has been discovered by testing our layout:

- **Mutual Recursion:** When we execute this test:

  ```
  Occur::= State[CBY : Occur]
  ```

  There will an infinite loop, without any visualization of the layout.

- **Collisions of objects:** The second problem is when we execute this test:

  ```
  State ::= Symptom | Disease | Missingvitamin | Treatment;
  Stuff ::= Cvitamin | Avitamin | Bvitamin | Evitamin;
  ```

  The result of this test is shown in figure 5.1:

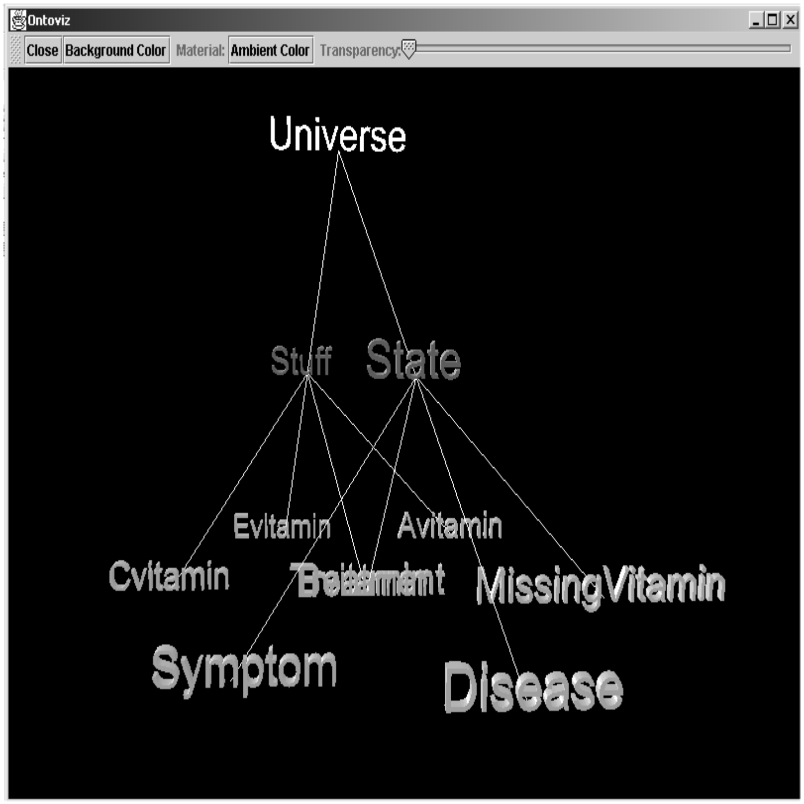

Figure 5.1: Collision in the Layout

Research has been made to avoid collisions in **Ontoviz system**, and we have found that **Java 3D API** has its own methods for detecting collisions.

By trying to implement the collision method, we have noticed that this method depends on the picking method that is already is not working. For more information please refer to chapter 4 in section 4.5.5.

The temporary solution that we can propose for the moment is by allowing *transparency* on the **Ontoviz** tree.

Another *kind* of collision of objects is when they have large *identifiers*, this problem can be avoided by deciding to show only the first 3 letters of every identifier and when we click on the object, the identifier will be shown for a laps of time on the canvas, and then it will be hidden again.

Another solution will be to make the radius of the circle changeable depending on the size of object_ids.

- **Minimizing edge crossings:** The third major problem that we have found by testing the layout, is the *edge crossing*.

  When we have decided to use **Java 3D API**, as we thought, was for using more *space* and specially minimizing the edge crossing problem. This problem, i.e. minimizing crossings was a hard task because of the structure of concepts that presents lattices(join and meet), the *mutual recursion*, the relationship in between concepts, and our decision[1] to show one identifier **only** once in the layout.

  It is impossible to eliminate edge crossing but we can try to minimize it, by permitting *attribute_ids* to be shown several times, or to avoid showing the *attribute_ids* on the *canvas*, and instead showing them, i.e. *attribute_ids* on the edges between *concept_ids*, and that can be done by popping up a frame, see figure 5.2 as an example.

  In figure 5.2, when the user moves the mouse on the red–line, it will be popped up a little window showing him/her that the relationship between concept_ids **X** and **B** is the attribute_value **WRT**.

  There is also a simple solution, by using *tooltip*, but unfortunately this cannot be achieved, we discussed this issue in chapter 4 in section 4.1.

  Personal Opinion: In reality the problem of **edge crossing** in our layout is also caused by showing our 3D structure on a *flat screen*. When we display the layout on a view where it seems that we have edge crossing, it will be eliminated when we try to look at our structure on another angle, and vice–versa.

- **Independent Platform:** We execute our implementation on Windows98 and Windows–2000–Pro, on two different computers, our application is working on both, but it is faster on one of the two computers and this depends on the porcessor. We notice another little difference that our visualization is flicking also on one of the two computers, and this problem is depending on the video–card.

  Notice that if the user wants to run our application, here what he/she has to do :
  1: Download the last version of **Java2** software development kit (**sdk**).

[1] This decision has been taken because of the large structure that caused by the large amount of production rules

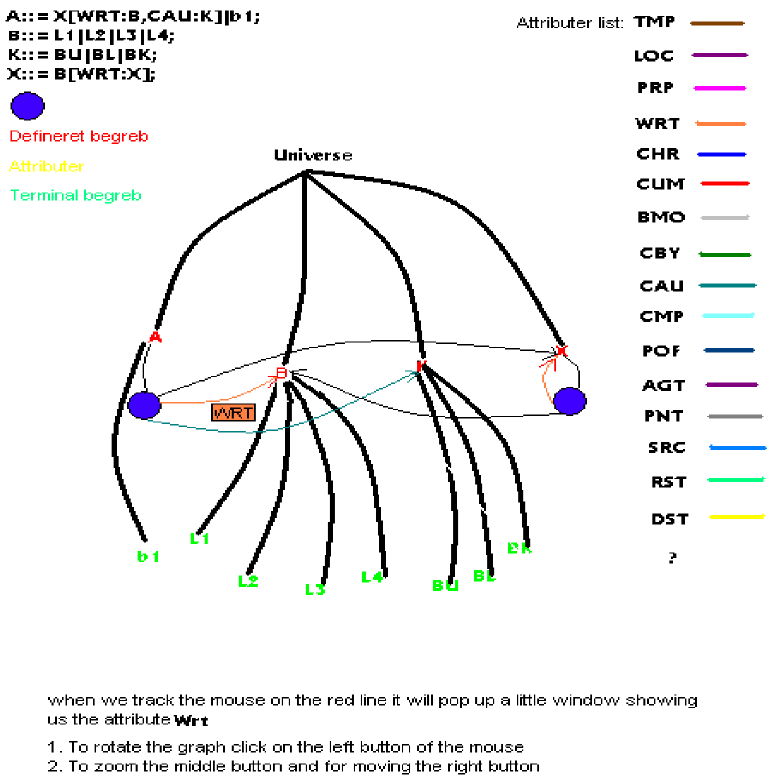

Figure 5.2: A New Layout for Ontoviz Tree

2: Download **Java3D** for **win32 directx** version, and install it under Java sdk directory.
3: Include our application under **Java2 sdk** and go to the directory where you include our application, under **dos command prompt**, then compile our application by typing "compile" and execute it by typing "cg".

In chapter 6 we will finish our report with a summary of our implementation and how it[2] can be improved as future work.

[2]I.e. our implementation

Chapter 6

Conclusion

This chapter includes two sections, the first one is a discussion about our project and our implementation, the second section includes the future work that can be done like improving our implementation etc.

6.1 Discussion

English :During this project we construct graphical user interface, (we call it *ONTOVIZ*), for **ontology**.

An ontology is an abstract model of concepts for a specific domain. The main use of **ontology** is for "structuring our knowledge", let us give a concrete example, Internet searching is based on **syntax**, i.e. text search for finding what we are looking for, the result of this search will include a lot of documents that we doesn't need.

Now imagine that we structure our Internet by building a hierarchy of objects, i.e. concepts that are classified. So instead, Internet searching will be based on "**semantic**" searching. This will lead to big probability of having the right hit, by minimizing the amount of results, when we enter our key to be found.

Our purpose, by giving this example, is to show the reader that all the theories behind *ontology* will be forgotten, when we deliver this thesis.

Going back to our project, a linear representation have been invented[1] to describe our domain for ontologies, and to present the ontology by using concepts and attributes, that related these concepts to each other.

This linear representation has been called "**Conceptual Grammar**".

[1]By professor Jørgen Fisher Nilsson.

The conceptual grammar is a special case of *BNF grammar*, because what we really are interested here is to visualize **only** concepts, attributes and nothing else.

The conceptual grammar includes a large amount of productions rules. A user reading these rules can be confused, so what is the solution?

To help to make ourself clear, human–being use images, figures, graphs etc. So the reader can conclude that we want to visualize our conceptual grammar's structure.

Here comes the hard part. i.e. the visualization.

Questions:

- How we will visualize our structure, is it in 2 dimensions or 3?

- What kind of tool we want to use to implement our application?

- **The hardest** part is the layout of *Ontoviz tree*. How the tree will look like?

Because of our large structure and earning more **"Space"** comes the idea for an extra dimension, that is why we decided to present our structure in 3D[2] instead of 2D.

What about the layout of *Ontoviz* tree?. The article *Graph Visualization and Navigation in Information*[3]*.* has been an inspiration for us to decide layout for our structure using the simple **cone trees** layout.

The tool to implement with our application, was decided to be Java 3D API[4], one big *advantage* of using *Java 3D*, is that because of **platform independency** "write once, used everywhere", also flexibility etc, for further information please read advantages/disadvantage about **Java3D** that have been discussed in chapter 4, in sections 4.6 and 4.7.

Further on I thought that it will be a good idea to include a **tool–box** menu where the user can interact with the visualization of our structure, for example change colors of the nodes of *Ontoviz tree*, the background, transparency, diffuse color, ambient color so we can achieve a pretty layout of our structure and for interaction.

Information zooming, was another area that we have tried to achieve so the user can retrieve information from the *Ontoviz tree*, that helps him/her to better understand the tree layout.

[2] *Dimensions*

[3] By Ivan Herman, Guy Melancon, and M.Scott Marshall.

[4] **Application Programming Interface.**

I am quiet satisfied with my result, but there is more work to be done like **minimization of edge crossing** and other problems that will be discussed in the next section.

Frensh : Lors de ce projet, on a construit une construction graphique afin de viser l'ontologie.

L'ontologie est un model abstrait de notions pour un domaine specific. La principale utilisation de l'ontologie est afin de "structurer notre connaissance", permettez nous de vous donner un exemple concret de cette utilisation ;

La recherche sur Internet est basée sur la recherche par "syntaxe", je voudrais dire par cela on cherche par mot, le résultat de ce genre de recherche va aboutir à un nombre illimité des documents, dont on n'a pas besoin.

Maintenant imaginez que nous structurons notre Internet par la construction d'une hiérarchie d'objets qui représente les notions (concepts) qui sont classifiés.

Cela va aboutir à une grande probabilité de minimiser les documents et économiser notre temps.

Cet exemple a un but de faire voir à notre lecteur que la théorie derrière l' ontologie, ne sera pas oubliée dès qu'on termine cette thèse.

Revenons à notre projet, une représentation linéaire a été inventée[5] pour décrire notre domaine de l'ontologie, et de présenter la notion des (concepts et des attribut es), qui lient ces (concepts) entre eux.

La représentation linéaire a été nommée "conceptuel grammar".

"Conceptuel grammar" est un ensemble des règles de production, de notions de "concepts" reliés entre eux par la notion des "attribut es".

Cette représentation peut rendre le lecteur assez confus, alors la solution c'est de représenter cette structure graphiquement, et bien sûr ce qui nous intéresse c'est de viser les notions des "concepts" et comment ils sont reliés entre eux.

pour rendre les choses encore plus claire, un humain utilise les images, afin de se faire comprendre. Alors maintenant le lecteur peut conclure que nous voulons construire un graph afin de viser notre "conceptuel grammar".

Et là commence la partie la plus difficile.

Questions :
- Comment veut–on viser notre structure, est–il en 2 ou 3 dimensions ?
- Quel genre de programme on veut utiliser pour implémenter notre structure ?

[5]Par le professeur Monsieur Jørgen Fisher Nilsson

– Quel genre de layout qu'on va appliquer à notre **Ontoviz arbre** ?

Suite à notre large structure et pour la possession de plus de "**Place**", on a eu une idée d'utiliser une dimension de plus, donc on a décidé de présenter notre structure en 3 dimensions au lieu de deux dimensions.

Et alors à propos de notre présntation de **l'arbre Ontoviz**. L'article *Graph Visualization and Navigation in Information*[6]. a été une grande inspiration pour nous décider de la présenter en utilisant la simple idée de l'arbre des cônes.

Le programme qu'on a utilisé pour implementer notre structure est **Java 3D**, l'une des raisons essentiels c'est l'indépendance de platform, nous voulos dire par là, "écrire un fois, utiliser partout", également pour la flexibilité etc, pour plus d'infos veuillez lire les avantages/désavantages de **Java 3D** dans le chapitre 4, et précisement dans les sections 4.6 and 4.7.

On ne s'est pas arrêté là, pour plus d'interactions on a crée un menu "**ToolBox**", afin d' aboutir à une bonne présentation de notre graph. Également on a travaillé sur la partie où on peut avoir plein d'infos en cliquant sur le graph, afin de plus comprendre ce qui se passe dans notre structure, cela a introduit une vitalité.

Je termines par dire que je suis bien satisfait de mon implementation, mais le travail n'a pas encore fini, il y a des problèmes dont il faut faire face. Cette partie sera discutée dans la section suivante.

6.2 Future Work

The implementation is far away from being finished. In the report we point to many areas, to be reviewed, maybe redone or improved.

Here an overview about the areas that can be improved:

- **Mutual recursion:** This problem has been discussed previously in chapter 5, in section 5.2.

  We think that the problem is caused by *GrammarTree.GrammarInsert(...)*, it needs to be checked.

- **Parser Improvement:** We think that the parser implementation can be improved, by replacing all the *switch–case*, by routines for example.

- **Finishing the ToolBox Menu:** In the *ToolBox* menu, there is still to achieve linking the *shininess , diffuse and emissive color buttons*, to the *Ontoviz system.*

[6]Par Ivan Herman, Guy Melancon, et M.Scott Marshall.

At the end the menu displayed in *Ontoviz system* will look like in figure 6.1.

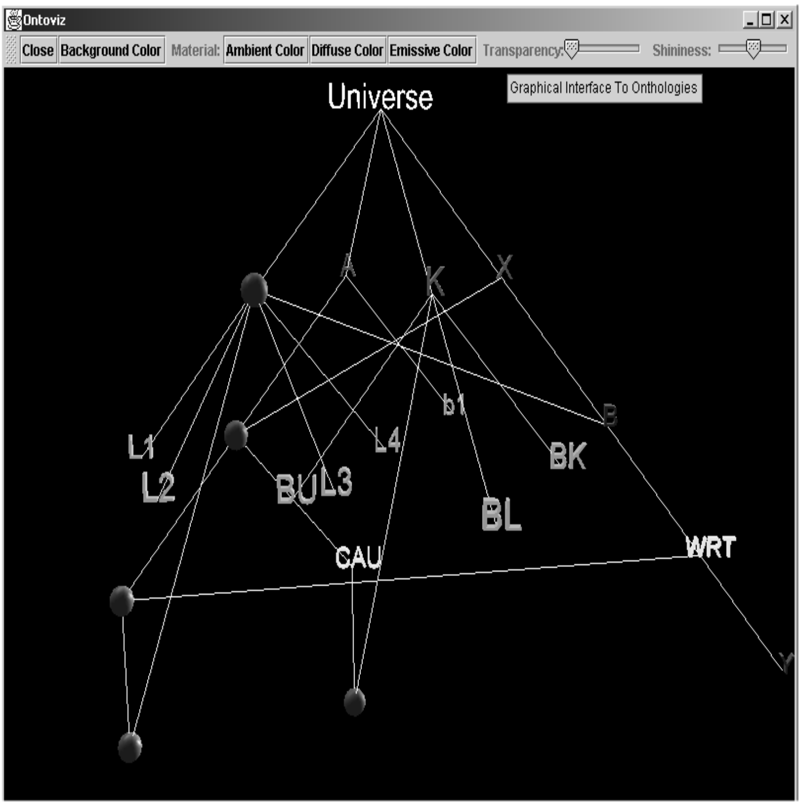

Figure 6.1: Finishing the ToolBox menu and Poping The Tooltip test it.

- **Object Collisions:** This problem has been discussed previously in chapter 5, in section 5.2, and solutions have been proposed. For more information about **Java 3D's collision system** please refer to chapter 4 in section 4.5.5.

- **Minimizing Edge Crossing:** This problem has been also discussed in chapter 5, in section 5.2, and solutions have been also proposed.

- **Tooltip problem:** This problem has been discussed in chapter 4 in section 4.1, we propose a solution, but unfortunately it is not popping up any *tooltip* on the *canvas*, instead it does pop our test on the *ToolBox* menu, when you drag the mouse onto the menu and not on the button created on the menu itself.

  See figure 6.1.

- **Information Zooming using PickRay:** This problem has been discussed in chapter 4 in section 4.5.4, we think that the problems seems to be in the **createPickRay(...)** method, it can checked and see if the problem comes from this method.

Bibliography

[1] Jørgen Fisher Nilsson. *A logico-Algebraic Framework for Ontholgies.* Department of Computer Science, Technical University of Denmark 2001.

[2] B.C Vickery. *Ontholgies.* University College London, Uk January 1997.

[3] Ivan Herman, Guy Melancon, and M.Scott Marshall. *Graph Visualization and Navigation in Information.* Center of Mathematics and Computer Sciences, Amsterdam, The Netherlands.

[4] Sun Microsystems Inc.*Java 3D API Documentation and Specification.* "http://www.java.sun.com/products/java-media/3D/releases.html" .

[5] Sun Microsystems Inc. *Java 3D Tutorial* "http://java.sun.com/products/java-media/3D/collateral/"

[6] Giuseppe Di Battista, Peter Eades, Roberto Tamassia and Ioannis G. Tollis.*Graph Drawing.* Prentice Hall, 1999.

[7] Dines Bjørner.*Advanced Formal Software course.* Department of Computer Science, Technical University of Denmark 2001.

[8] Asger Eir. *Comparing Top–Down and Bottom–Up Concept Analysis in Ontology, Unpublished.* Department of Civil Engineering,and Informatics and Mathematical Modelling, Technical University of Denmark.

[9] H.M. Deitel and P.J. Deiltel. *Java, How to program.* Prentice Hall, 1998.

[10] Hans Bruun.*Programming in C++ course.* Department of Computer Science, Technical University of Denmark 2001.

[11] Jørgen Fisher Nilsson. *Conceptual Grammer.* Department of Computer Science, Technical University of Denmark 2001.

[12] Andrew W. Appel. *Modern Compiler implementation in SML.* Cambridge, United Kingdom 1999.

Appendix A

The Source Code

A.1 Cg.java

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

import javax.swing.*;

import java.io.*;
import com.sun.j3d.utils.applet.MainFrame;
import javax.media.j3d.*;


public class Cg extends JApplet
{

public static void main(String args[])
{
String s_grammar = "";  // accumulator for the entire input (where new-lines are ignored)

if (args!=null)  // check if there are any arguments on the command line
{
try{
if (args[0]=="") {System.out.println("No inputfile specified...");System.exit(1);}
}
catch (Exception e) {System.out.println("No inputfile specified...");System.exit(1);}

String inline = "";
File grammarfile = new File(args[0]);
try
{
  BufferedReader in = new BufferedReader(new InputStreamReader(new FileInputStream(grammarfile)));
  while ((inline=in.readLine())!=null)
                      // accumulate the contents of the entire input file in the string s_grammar
                      s_grammar += inline;
  in.close();
}
catch (Exception e)
{
  new PrintError("Error opening or reading file : " + args[0]);
}
```

```
}

//Global BranchGroup
BranchGroup rootBranchGroup = new BranchGroup();
//Start the parser
Parser p = new Parser(s_grammar, rootBranchGroup);
//Open the display frame
new MainFrame(new Ontoviz(rootBranchGroup), 800, 600);


// obs: this is actually not an error (it prevents the program from exiting)
new PrintError("Hit RETURN or close the window...");
} // end main


/*not used
public Cg()
{
} // end constructor
*/


// for a browser
public String getAppletInfo()
{
return "Name: cg\r\n" +
       "Author : Mark A.";
}


// for a browser
public void init()
{
     resize(320, 240);

}


/*not used
    // for a browser
public void destroy()
{
}
*/

/*not used
public void paint(Graphics g)
{
}
*/

/*not used
// for a browser
public void start()
{
}
*/

/*not used
// for a browser, always called before destroy()
public void stop()
{
```

```
}
*/

} // end class Cg
```

A.2 GrammarTree.java

```
import java.util.*;
import javax.vecmath.*;
import javax.media.j3d.*;
import com.sun.j3d.utils.geometry.Sphere;

public class GrammarTree
{

    public final static int RULE=1;
public final static int CONCEPT=2;
public final static int SIMPLECONCEPT=3;
public final static int FEATURE=4;

private final static int RADIUS=10;
private final static int DELTA_Z=10;



// shared transparency object used by all nodes
    static private TransparencyAttributes transparency = new TransparencyAttributes();

    static
    {
transparency.setCapability(TransparencyAttributes.ALLOW_MODE_READ);
transparency.setCapability(TransparencyAttributes.ALLOW_MODE_WRITE);
transparency.setCapability(TransparencyAttributes.ALLOW_VALUE_READ);
transparency.setCapability(TransparencyAttributes.ALLOW_VALUE_WRITE);
// set to non-transparency (opaque) initially
transparency.setTransparencyMode(TransparencyAttributes.NONE);
transparency.setTransparency(0.0f);
    }


// input value:  0.0 <= alpha <= 1.0
    static public void setTransparency(float alpha)
    {
        transparency.setTransparencyMode((alpha < 0.1)
           ? TransparencyAttributes.NONE
           : TransparencyAttributes.NICEST);
transparency.setTransparency(alpha);
}


// returned value:  0.0 <= alpha <= 1.0
    static public float getTransparency()
    {
//setupStatic();
return transparency.getTransparency();
}
```

```
private static GrammarTree root;

protected Vector children;
    protected GrammarTree parent;
    protected String key;
protected int type;  // i.e. RULE, CONCEPT, SIMPLECONCEPT or FEATURE

private Point3f pos;
private float currentz = 0;
private String depth = "";
    private double deltaphi = 0;

private NodeText nodetext=null;
private LineArray linestochildren=null;
private Sphere knot=null;
private static BranchGroup branchgroup;



public String getTypeString()
{
switch (type)
{
case RULE:  return "rule";
case CONCEPT:  return "concept";
case SIMPLECONCEPT:  return "simpleconcept";
case FEATURE:  return "feature";
default: return "(unknown)";
}
}




    ////////////////////////
// GrammarTree Constructor.
//
// Parameters :
// treeTitle : rootnode title
// nodetype : the type
// par : the parent tree
//
////////////////////////

public GrammarTree(String treeTitle, int nodetype, GrammarTree par)
{
children = new Vector();
parent = par;
key = treeTitle;
pos = new Point3f(0,0,0);
type = nodetype;
}

////////////////////////
// GrammarTree Constructor.
//
// Parameters :
// treeTitle : rootnode title
```

```
// nodetype : the type
// bg : the java3D branchgroup
//
////////////////////////

public GrammarTree(String treeTitle, int nodetype, BranchGroup bg){
children = new Vector();
parent = null;
key = treeTitle;
pos = new Point3f(0,0,0);
branchgroup = bg;
type = nodetype;
root = this;
}

    //Copy the tree
    public void copy(GrammarTree t)
    {
        GrammarTree gt = (GrammarTree)t;
        for (int i=0; i<gt.children.size(); i++)
          children.addElement(gt.children.elementAt(i));
        parent = gt.parent;
        key = gt.key;
    }

    // returns the name of the ID
public String getKey()
{
        return key;
    }

    // sets the name of the ID
public void setKey(String k)
{
        key=k;
    }

    // get the node type
public int getType()
{
        return type;
    }


    // get the position of the node
public Point3f getPos()
{
return pos;
}

// set the position of the node
public void setPos(Point3f p)
{
pos=p;
}

// search for a child by name, and return it
public GrammarTree getChild(String s)
    {
        for (int i=0;i<children.size();i++)
if ( ((GrammarTree) children.elementAt(i)).getKey().equals(s))
     {return (GrammarTree) children.elementAt(i);}
```

```
        return null;
    }

// search for a child by an index, and return it
public GrammarTree getChild(int i){return (GrammarTree) children.elementAt(i);}

    // search for a child by name, and return the index
public int getchildIndex(String s)
    {
        for (int i=0;i<children.size();i++)
if ( ((GrammarTree) children.elementAt(i)).getKey().equals(s))
 return i;

return 0;
    }


// return parent
public GrammarTree getParent()
{
        return parent;
     }

// count children
public int numChildren()
{
if (children==null) return 0;
return children.size();
}

// visualize the tree
// create the nodetext member for all nodes,
// place them correctly and make the edges for the tree

public void printChildren(GrammarTree tree, double phi, Point3f parentpos)
{

if (tree.pos.x==0.0f && tree.pos.y==0.0f && tree.pos.z==0.0f)
{
  if (phi!=0)
{
 tree.pos.x=(float) (RADIUS*Math.cos(phi))+parentpos.x;
 tree.pos.y=(float) (RADIUS*Math.sin(phi))+parentpos.y;
 tree.pos.z=currentz;
}
  else
{
 tree.pos.x=parentpos.x;
 tree.pos.y=parentpos.y;
 tree.pos.z=currentz;
}
nodetext = new NodeText(this, tree.getKey(),tree.pos, tree.type, branchgroup, transparency);
}
 else
  {
    if (tree.key==" ")
{
 //GrammarTree temptree=tree.getNode(tree.getKey(),root);
 //if (temptree!=null) temptree.setPos(tree.pos);

  //knot = new GrammarSphere(tree, 1.0f);
  knot = new GrammarSphere(this, 1.0f);
```

```
  Appearance apSphere = new Appearance();
  Material m = new Material();
  m.setLightingEnable(true);
  m.setAmbientColor(new Color3f(0.0f, 0.0f, 1.0f));
  apSphere.setMaterial(m);
            apSphere.setTransparencyAttributes(transparency);  // use the shared transparency object

  knot.setAppearance(apSphere);
  TransformGroup sphereGroup = new TransformGroup();
  sphereGroup.setPickable(true);
  Transform3D sphereXform = new Transform3D();
        sphereXform.set(new Vector3f(tree.pos));
  sphereGroup.setTransform(sphereXform);
  sphereGroup.addChild(knot);
  branchgroup.addChild(sphereGroup);

}
  }

if (tree.parent!=null)
  System.out.println(depth+tree.getKey()+" x:"+tree.pos.x+" y:"+tree.pos.y+" z:"+tree.pos.z+

                                                                              "   Parent: "+tree.parent.getKey());
else
  System.out.println(depth+tree.getKey()+" x:"+tree.pos.x+" y:"+tree.pos.y+" z:"+tree.pos.z);

for (int i=0;i<tree.numChildren();i++)
{
if (i==0)
{
depth+=".";
currentz-=DELTA_Z;

if (tree.numChildren()!=1)
{
tree.deltaphi=((2*Math.PI)/tree.numChildren());
}

tree.linestochildren = new LineArray(tree.numChildren()*2, LineArray.COORDINATES  );


}

printChildren((GrammarTree)tree.getChild(i),tree.deltaphi*(i+1),tree.pos);

if (tree.numChildren()-1==i)
{

depth=depth.substring(0,depth.length()-1);
currentz+=DELTA_Z;

if (tree.numChildren()!=0)
{
for (int j=0;j<tree.numChildren();j++)
{
tree.linestochildren.setCoordinate(j*2,tree.pos);
tree.linestochildren.setCoordinate(j*2+1,tree.getChild(j).getPos());

}
branchgroup.addChild(new Shape3D(tree.linestochildren));
     }
```

```
}
}

} // end printChildren


// return a GrammerTree node
public GrammarTree getNode(String name, GrammarTree tree)
{
if (tree.getChild(name)!=null) return tree.getChild(name);

for (int i=0; i<tree.numChildren(); i++)
{
GrammarTree t;
if ((t=getNode(name,tree.getChild(i)))!=null)
return t;
}
return null;
}


// insert a new node, if it does not exist. And return the new tree
public GrammarTree GrammarInsert(Vector v, int type,GrammarTree tree, GrammarTree par)
{

if(tree == null)
{
GrammarTree node = new GrammarTree((String)v.elementAt(0),type,par);
GrammarTree tempnode = getNode((String)v.elementAt(0),root);
if (tempnode!=null)
{
tempnode.setKey(" ");
node.children.addElement(tempnode);
}
return node;
}

        if (v.size()==1)
{
if (tree.getChild((String)v.elementAt(0))==null && (String)v.elementAt(0)!=tree.getKey())
tree.children.addElement(GrammarInsert(v,type,(GrammarTree)tree.getChild((String)v.elementAt(0)),tree));
}
else
{
String newData=(String) v.elementAt(0);
v.removeElementAt(0);
tree.children.setElementAt(GrammarInsert(v,type,(GrammarTree) tree.getChild(newData),tree),
                                                            tree.getchildIndex(newData));
}
return tree;
    }



// inner class to give us mouse connection to GrammarTree instances
class GrammarSphere extends Sphere
{
private GrammarTree node;

public GrammarSphere(GrammarTree node, float fl)
{
super(fl);
```

```
this.node = node;
}

public GrammarTree getTreeNode() { return node; };

    } // end GrammarSphere
}
```

A.3 NodeText.java

```
//
//
// NodeText
//
//
import javax.media.j3d.*;
import javax.vecmath.*;
import java.awt.*;

class NodeText
{

////////////////////////
// NodeText Constructor.
//
// Parameters :
//
// tree : the node in the non-graphical tree representation
// textString : the text
// p : the rotation (middle of the text) point
// type : the node type (SIMPLECONCEPT,CONCEPT....)
// objRoot : the java3D branchgroup
// transp : the shared transparency object for all nodes
//
////////////////////////
public NodeText(GrammarTree tree, String textString,Point3f p, int type, BranchGroup objRoot,
TransparencyAttributes transp)
{
Color3f nodecolor;
// set the text color
switch(type)
{
case GrammarTree.RULE : nodecolor=new Color3f(1.0f, 0.0f, 0.0f);break;
case GrammarTree.CONCEPT: nodecolor=new Color3f(0.0f, 0.0f, 1.0f);break;
case GrammarTree.SIMPLECONCEPT:nodecolor=new Color3f(0.0f, 1.0f, 0.0f);break;
case GrammarTree.FEATURE:nodecolor=new Color3f(1.0f, 1.0f, 0.0f);break;
default : nodecolor=new Color3f(1.0f, 1.0f, 1.0f);
}



int sl=textString.length();
// set the appearence of the text
Appearance apText = new Appearance();
Material m = new Material();
        m.setLightingEnable(true);
m.setAmbientColor(nodecolor);
apText.setMaterial(m);
        apText.setTransparencyAttributes(transp);  // use the shared transparency object
```

```
////////////////////////////////

// create the font
// make the characters slightly deeper(default is 0.2 here we set it to 0.5)
java.awt.geom.Line2D shape = new java.awt.geom.Line2D.Float(0f, 0f, 0.5f, 0.5f);
FontExtrusion fontExtr = new FontExtrusion(shape);

Font3D f3d = new Font3D(new Font("Arial", Font.PLAIN, 2), fontExtr);
// make a Point3f
Point3f textPt = new Point3f(-sl/2.0f+p.x, p.y, p.z);
        // create the Text3D object
Text3D txt = new Text3D(f3d, textString, textPt);



// create the OrientedShape3D object
//OrientedShape3D textShape = new OrientedShape3D();
//Shape3D textShape = new Shape3D();
NodeShape textShape = new NodeShape(tree);

// put it all together
textShape.setGeometry(txt);
        textShape.setAppearance(apText);

textShape.setAlignmentMode(OrientedShape3D.ROTATE_ABOUT_POINT);
textShape.setRotationPoint(p);
     ////////////////////////////////

// add the textShape to the branch group
objRoot.addChild(textShape);
}



// inner class to give us mouse connection to TreeNode instances
class NodeShape extends OrientedShape3D
{
private GrammarTree node;

public NodeShape(GrammarTree node)
{
this.node = node;
}

public GrammarTree getTreeNode() { return node; };

    } // end NodeShape
}
```

A.4 Ontoviz.java

```
//
//
// Ontoviz
//
//
import java.applet.*;
```

```
import java.awt.*;
import java.awt.event.*;

import javax.swing.*;


import com.sun.j3d.utils.universe.*;
import javax.media.j3d.*;
import javax.vecmath.*;
import com.sun.j3d.utils.behaviors.vp.*;
import com.sun.j3d.utils.geometry.Sphere;


public class Ontoviz extends JApplet
{
private SimpleUniverse u;
private BranchGroup rootBranchGroup;
private AmbientLight ambientLightNode;
private Canvas3D canvas;

private DirectionalLight light1;
private DirectionalLight light2;


private double bgScale;
private Color3f bgColor;
private Color3f ambientColor;
//private TransparencyInterpolator transp;
private TransparencyAttributes ta;
private Background bgNode;

      private Ontoviz instance;


public Ontoviz(BranchGroup b)
{
rootBranchGroup = b;
} // end constructor


public BranchGroup createSceneGraph()
{

// Create the root of the branch graph
BranchGroup objRoot = new BranchGroup();

TransformGroup objScale = new TransformGroup();
Transform3D transform = new Transform3D();
transform.setScale(bgScale);


objScale.setTransform(transform);
objScale.addChild(rootBranchGroup);

TransformGroup objTrans = new TransformGroup();
objTrans.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
objTrans.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
objRoot.addChild(objTrans);

BoundingSphere bounds =
new BoundingSphere(new Point3d(0.0,0.0,0.0), 10000.0);
```

```
// Set up the background

bgNode = new Background(bgColor);
bgNode.setApplicationBounds(bounds);
bgNode.setCapability(Background.ALLOW_COLOR_WRITE);
objRoot.addChild(bgNode);

// Set up the ambient light
ambientLightNode = new AmbientLight(ambientColor);
ambientLightNode.setCapability(Light.ALLOW_COLOR_WRITE);
ambientLightNode.setInfluencingBounds(bounds);
objRoot.addChild(ambientLightNode);

// Set up the directional lights
//Color3f light1Color = new Color3f(1.0f, 1.0f, 0.9f);
Color3f light1Color = new Color3f(0.3f, 0.3f, 0.4f);
Vector3f light1Direction  = new Vector3f(4.0f, -7.0f, -12.0f);

//Color3f light2Color = new Color3f(0.3f, 0.3f, 0.4f);
Color3f light2Color = new Color3f(0.7f, 0.7f, 0.8f);
Vector3f light2Direction  = new Vector3f(-6.0f, -2.0f, -1.0f);
light1 = new DirectionalLight(light1Color, light1Direction);
light1.setInfluencingBounds(bounds);
objRoot.addChild(light1);

light2 = new DirectionalLight(light2Color, light2Direction);
light2.setInfluencingBounds(bounds);
objRoot.addChild(light2);


objRoot.addChild(objScale);
objRoot.compile();
return objRoot;
} // end createSceneGraph




public void setAmbientColor(float r, float g, float b)
{
Color3f color = new Color3f(r, g, b);
ambientLightNode.setColor(color);
}


public void setEmissiveColor(float r, float g, float b)
{
Color3f color = new Color3f(r, g, b);
//material.setColor(color);
}


  public void setTransparency(float alpha)
{
GrammarTree.setTransparency(alpha);
}


public void setBackgroundColor(float r, float g, float b)
{
Color3f color = new Color3f(r, g, b);
bgNode.setColor(color);
```

```
}



public void init()
{
JPanel content = (JPanel)getContentPane();
//JPanel content = this;
content.setLayout(new BorderLayout());

// adding toobar
ToolBox toolbar = new ToolBox("Toolbox", this);
content.add(toolbar, BorderLayout.NORTH);


// tooltip test
toolbar.setToolTipText("Graphical Interface To Onthologies");


// obs: advice from http://www.daniel-hirsch.de/RubiksCube/description/problems.html
JPopupMenu.setDefaultLightWeightPopupEnabled(false);
ToolTipManager.sharedInstance().setLightWeightPopupEnabled(false) ;

// Not necessary:
//ToolTipManager.sharedInstance().registerComponent(pan);



// canvas that the 3D is drawn upon
canvas = new Canvas3D(SimpleUniverse.getPreferredConfiguration());
content.add(canvas, BorderLayout.CENTER);
//TEST pan.add(c, BorderLayout.CENTER);


            /*MAYBE USED LATER (gives the mouse position over the canvas)
canvas.addMouseMotionListener(new MouseMotionAdapter() {
                public void mouseMoved(MouseEvent ev)
                {
                  System.out.println("x=" + ev.getX() + " y=" + ev.getY());
                }
              });
            */

        instance = this;  // to center the message dialog over the Ontoviz instance
canvas.addMouseListener(new MouseAdapter() {
                public void mouseClicked(MouseEvent ev)
                {
  clickAction(ev.getX(), ev.getY());
                }
              });

bgScale = 0.05;
bgColor = new Color3f(0.0f, 0.0f, 0.0f);
ambientColor = new Color3f(1.0f, 1.0f, 1.0f);
// Create a simple scene and attach it to the virtual universe
BranchGroup scene = createSceneGraph();
u = new SimpleUniverse(canvas);

// add mouse behaviors to the ViewingPlatform
ViewingPlatform viewingPlatform = u.getViewingPlatform();
```

```
// This will move the ViewPlatform back a bit so the
// objects in the scene can be viewed.
viewingPlatform.setNominalViewingTransform();

 // add orbit behavior to the viewing platform
OrbitBehavior orbit = new OrbitBehavior(canvas, OrbitBehavior.REVERSE_ALL);
BoundingSphere bounds =
new BoundingSphere(new Point3d(0.0, 0.0, 0.0), 10000.0);
orbit.setSchedulingBounds(bounds);
viewingPlatform.setViewPlatformBehavior(orbit);

u.addBranchGraph(scene);
} // end init




    /** This method controls what will happen when the user clicks
        on the canvas
    */
    private void clickAction(int x, int y)
{
PickRay ray = createPickRay(canvas, x, y);

//OBS: need to fix, we don't get the right node
// SceneGraphPath path = rootBranchGroup.pickAny(ray);
SceneGraphPath path = rootBranchGroup.pickClosest(ray);

Node node;

/*maybe used later
if (path!=null  &&  (node=path.getObject())!=null  )//&&  (node instanceof GrammarTree.GrammarSphere))
{
            GrammarTree tree = ((GrammarTree.GrammarSphere)node).getTreeNode();
*/
if (path!=null  &&  (node=path.getObject())!=null  &&  (node instanceof NodeText.NodeShape))
{
            GrammarTree tree = ((NodeText.NodeShape)node).getTreeNode();
if (tree!=null)
{
String message = "Node=" + node +"  key=" + tree.getKey() + "  type=" + tree.getTypeString();
//String message = "Node=" + node;
JOptionPane.showMessageDialog(instance, message, "Details",
  JOptionPane.INFORMATION_MESSAGE);
// for testing
/*used later
                System.out.println("x=" + x + " y=" + y);
*/
}
}
} // end clickAction



    /** Get the 3D ray (from user's eye to infinity) at the mouse position
    */
private PickRay createPickRay(Canvas3D canvas, int x, int y)
{
  Point3d eye_pos = new Point3d();
  Point3d mouse_pos = new Point3d();
```

```
  canvas.getCenterEyeInImagePlate(eye_pos);
  canvas.getPixelLocationInImagePlate(x, y, mouse_pos);

  Transform3D motion = new Transform3D();
  canvas.getImagePlateToVworld(motion);
  motion.transform(eye_pos);
  motion.transform(mouse_pos);

  Vector3d direction = new Vector3d(mouse_pos);
  direction.sub(eye_pos);

  return new PickRay(eye_pos, direction);
} // end createPickRay



    // for a browser
public void destroy()
{
u.removeAllLocales();
}


} // end class Ontoviz
```

A.5 Parser.java

```
//
//
// Parser
//
//
import java.awt.*;
import java.util.*;
import javax.vecmath.*;
import javax.media.j3d.*;

class Parser
{

private Vector rulelist = null;
private Scanner sc = null;
private boolean error = false;
private String currule = "", errorMsg = "";

////////////////////////
// Parser Constructor.
// Starts parsing the grammer.
// Parameters :
// grammer : a string containing the entire contents of the "test.gra" file.
// bg : the Java3D branchgroup
//
////////////////////////

public Parser (String grammar, BranchGroup bg)
{
rulelist = new Vector();
```

```
int index = -1;

// chop the input string into segments separated by ';'
while ((index=grammar.indexOf(';'))!=-1)
{
// take the segment and add an element to the dynamic array
rulelist.addElement(grammar.substring(0,index+1));
// cut away the segment that we have already treated
// i.e. keep the remainder of the input string
grammar = grammar.substring(index+1,grammar.length());
}

// check the validity of rulelist
if (rulelist.size()==0)
new PrintError("Missing ';'");
if (grammar.length()!=0)
System.out.println("Missing last ';' The rule will be not taken in consideration ");

// create tree root
GrammarTree gt = new GrammarTree("Universe",0,bg);


for (int i=0, max=rulelist.size();  i<max;  i++)
{
// the Scanner splits the rule into tokens
sc = new Scanner((String)rulelist.elementAt(i));
int prevtok=-1, tok=-1, nexttok=1;
errorMsg = "";
error = false;
boolean gotDEFRULE = false;
boolean needFeature = false;
boolean gotConcept = false;

int bracketcount = 0;

// the vector path to the tree node to be inserted
Vector insertv = new Vector();


// handle one token at a time (tok) with look-ahead to the next token (nexttok)
while ((tok=sc.nextToken(false))!=Tokens.SEMICOLON)
{
nexttok=sc.nextToken(true);

switch (tok)
{
case Tokens.ERROR:
errorMsg="Scanner Error !"; error=true;
break;

case Tokens.ID:
switch (nexttok)
{
case Tokens.DEFRULE :
System.out.println("RuleId="+sc.getLexeme());
currule=sc.getLexeme();
insertv.addElement(sc.getLexeme());
// insert the node in the tree
gt = gt.GrammarInsert((Vector)insertv.clone(),GrammarTree.RULE,gt,null);
break;

case Tokens.OR:
```

```
System.out.println("SimpleConceptId="+sc.getLexeme());
insertv.addElement(sc.getLexeme());
gt=gt.GrammarInsert((Vector)insertv.clone(),GrammarTree.SIMPLECONCEPT,gt,null);
break;

case Tokens.LEFTBR:
System.out.println("ConceptId="+sc.getLexeme());
insertv.addElement(sc.getLexeme());
if (needFeature)
{
errorMsg="Syntax error.";error=true;
break;
}
gt = gt.GrammarInsert((Vector)insertv.clone(),GrammarTree.CONCEPT,gt,null);
gotConcept=true;
break;

case Tokens.COLON:
System.out.println("FeatureId="+sc.getLexeme());
insertv.addElement(sc.getLexeme());
if (!gotConcept)
{
errorMsg="Syntax error.";
error=true;
break;
}
gt = gt.GrammarInsert((Vector)insertv.clone(),GrammarTree.FEATURE,gt,null);
gotConcept=false;
needFeature=false;
break;

case Tokens.RIGHTBR:
System.out.println("ConceptId="+sc.getLexeme());
insertv.addElement(sc.getLexeme());
if (needFeature)
{
errorMsg="Syntax error.";
error=true;
break;
}
gt=gt.GrammarInsert((Vector)insertv.clone(),GrammarTree.CONCEPT,gt,null);
gotConcept=true;
break;

case Tokens.COMMA:
System.out.println("ConceptId="+sc.getLexeme());
insertv.addElement(sc.getLexeme());
if (needFeature)
{
errorMsg="Syntax error.";
error=true;
break;
}
gt=gt.GrammarInsert((Vector)insertv.clone(),GrammarTree.CONCEPT,gt,null);
gotConcept=true;
break;

case Tokens.SEMICOLON:
System.out.println("SimpleConceptId="+sc.getLexeme());
insertv.addElement(sc.getLexeme());
gt = gt.GrammarInsert((Vector)insertv.clone(),GrammarTree.SIMPLECONCEPT,gt,null);
break;
```

```
case Tokens.ERROR:
errorMsg="Scanner Error !";
error=true;
break;

} // end switch
break;


case Tokens.DEFRULE:
if (gotDEFRULE)
{
errorMsg="Can't define a new rule in a rule.";
error=true;
break;
}
if (prevtok!=Tokens.ID)
{
errorMsg="Missing RuleId.";
error=true;
break;
}
if (nexttok!=Tokens.ID)
{
errorMsg="Missing ConceptId/SimpleConceptId.";
error=true;
break;
}
gotDEFRULE=true;
break;

case Tokens.OR:
if (nexttok!=Tokens.ID)
{
errorMsg="Missing ConceptId/SimpleConceptId after '|' ";
error=true;
break;
}
insertv=removeElements(insertv,1);
break;

case Tokens.LEFTBR:
if (prevtok!=Tokens.ID)
{
errorMsg="Missing ConceptId before '['.";
error=true;
break;
}
if (nexttok!=Tokens.ID)
{
errorMsg="Missing FearureId after '['.";
error=true;
break;
}
bracketcount++;
break;

case Tokens.COLON:
if (prevtok!=Tokens.ID)
{
errorMsg="Missing FeatureId before ':'.";
```

```
error=true;
break;
}
else if (nexttok!=Tokens.ID)
{
errorMsg="Missing ConceptId after ':'.";
error=true;
break;
}
break;

case Tokens.RIGHTBR:
bracketcount--;
if (bracketcount<0)
{
errorMsg="No matching '['.";
error=true;
break;
}
if (bracketcount==0)
{
if (nexttok!=Tokens.SEMICOLON && nexttok!=Tokens.OR && nexttok!=Tokens.RIGHTBR)
{
errorMsg="Missing ']'.";
error=true;
break;
}
}
if (bracketcount>0)
{
if (nexttok!=Tokens.COMMA && nexttok!=Tokens.RIGHTBR)
{
errorMsg="Missing ']' or ','.";
error=true;
break;
}
}
insertv=removeElements(insertv,2);
break;

case Tokens.COMMA:
if (prevtok!=Tokens.ID && prevtok!=Tokens.RIGHTBR)
{
errorMsg="Missing SimpleConceptId or ']' before ','.";
error=true;
break;
}
if (nexttok!=Tokens.ID)
{
errorMsg="Missing FeatureId after ','.";
error=true;
break;
}
insertv=removeElements(insertv,2);
needFeature=true;
break;
} // end switch

if (error)
{
System.out.println("Error Parsing "+currule+" : "+errorMsg);
break;
```

```
}
prevtok=tok;

} // end while
} // end for loop
gt.printChildren(gt,0,new Point3f(0,0,0));

} // end Parser


private Vector removeElements(Vector v,int num)
{
for (int i=0; i<num; i++)
v.removeElementAt(v.size()-1);
return v;
} // end removeElements


public void print_rules()
{
for (int i=0; i<rulelist.size(); i++)
System.out.println(i+". "+rulelist.elementAt(i));
} // end print_rules

} // class Parser
```

A.6 PrintError.java

```
//
//
// PrintError
//
//
class PrintError
{
// default constructor (no action here)
public PrintError() {}

// Print the string, wait for the user to press the Enter key and then terminate the program
public PrintError(String s)
{
System.out.println(s);
try
{
// wait until the user hits the return key
System.in.read();
}
catch (Exception e)
{
}
System.exit(-1);
}

/** wait for the user to hit the Enter key
 */
public void waitforEnter() {try {System.in.read();} catch (Exception e){}}
}
```

A.7 Scanner.java

```
//
//
// Scanner
//
//
class Scanner
{
private final static String IDCHARS="0123456789abcdefghijklmnopqrstuvwxyzæøåABCDEFGHIJKLMNOPQRSTUVWXYZÆØÅ_";
private char[] charbuf=null;
private int bufpointer=1, savedbufp;
private String lexeme, savedlex;
private char curchar=' ', savedchar;
private int tok;

////////////////////////
// Scanner Constructor.
//
// Parameters :
//
// rule : the rule to be scanned
//
////////////////////////

Scanner(String rule)
{
charbuf = rule.toCharArray();
curchar = charbuf[0];
}

// returns the next token, and moves the bufferpointer if the lookAhead is not true
int nextToken(boolean lookAhead)
{
if (lookAhead)
{
savedbufp=bufpointer;
savedlex=lexeme;
savedchar=curchar;
}
Lexeme();
if (lookAhead)
{
bufpointer=savedbufp;
lexeme=savedlex;
curchar=savedchar;
}
return tok;
}

// find the next token
void Lexeme()
{
lexeme="";
while (curchar==' ') nextchar();

if ((IDCHARS.indexOf(curchar))!=-1) scanId();

else
{
switch(curchar)
```

```
{ case ':': nextchar(); tok=Tokens.COLON; defrulechk(); return;
case ',': nextchar(); tok= Tokens.COMMA; return;
case '[': nextchar(); tok= Tokens.LEFTBR; return;
case '|': nextchar(); tok= Tokens.OR; return;
case ']': nextchar(); tok= Tokens.RIGHTBR; return;
case ';': tok= Tokens.SEMICOLON; return;
default : tok=Tokens.ERROR; return;
}
}

}

// is it a ruledefinition (::=) or a colon (:)?
void defrulechk()
{
if (curchar == ':')
{
nextchar();
if (curchar == '=' ) {nextchar();tok = Tokens.DEFRULE;}
else tok = Tokens.ERROR;
}

}

// get the next char and strip the whitespaces
void nextchar()
{
if (curchar!=' ') {lexeme+=curchar; curchar=charbuf[bufpointer++];}
while (curchar==' ') {curchar=charbuf[bufpointer++];}
}

// get the ID
void scanId()
{
// check for valid chars
while ((IDCHARS.indexOf(curchar))!=-1) nextchar();
tok=Tokens.ID;
}

// return the lexeme
String getLexeme(){return lexeme;}
}
```

A.8 ToolBox.java

```
// this is the toolbar that appears at the upper edge of the window

import java.io.*;
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;
import javax.swing.*;
import javax.swing.event.*;
import javax.media.j3d.*;
import javax.vecmath.*;


class ToolBox extends JToolBar implements ActionListener, ChangeListener
```

```
{
private Ontoviz viewer;
private JButton bClose;
private JButton bBackgrColor;
private JButton bMatAColor;
private JButton bMatDColor;
private JButton bMatEColor;
private JPanel pAlpha;
private JLabel lAlpha;
private JSlider sAlpha;
private JPanel pShininess;
private JLabel lShininess;
private JSlider sShininess;

// Material colors, transparency and shininess
int ra = 51, ga = 51, ba = 51;
int rd = 255, gd = 255, bd = 255;
int re = 0, ge = 0, be = 0;
float alpha = 0.0f;
float shininess = 64f;


public ToolBox(String c, Ontoviz viewer)
{
super(c);
setFloatable(true);

this.viewer = viewer;

bClose = new JButton("Close");
add(bClose);

bBackgrColor = new JButton("Background Color");
add(bBackgrColor);

add(new JLabel("  Material: "));

bMatAColor = new JButton("Ambient Color");
add(bMatAColor);

bMatDColor = new JButton("Diffuse Color");
// add(bMatDColor);

bMatEColor = new JButton("Emissive Color");
// add(bMatEColor);

lAlpha = new JLabel("  Transparency:");
add(lAlpha);

sAlpha = new JSlider(JSlider.HORIZONTAL, 0, 20,
 (int)(GrammarTree.getTransparency()*20.0f));  // read current value
add(sAlpha);

lShininess = new JLabel("  Shininess:");
// add(lShininess);

sShininess = new JSlider(JSlider.HORIZONTAL, 0, 128, 64);
// add(sShininess);



bClose.addActionListener(this);
```

```
bBackgrColor.addActionListener(this);
bMatAColor.addActionListener(this);
bMatDColor.addActionListener(this);
bMatEColor.addActionListener(this);
sAlpha.addChangeListener(this);
sShininess.addChangeListener(this);
} // end constructor



// this method handles all clicks on the toolbar buttons
public void actionPerformed(ActionEvent e)
{
if(e.getSource() == bClose) // close button
{
System.exit(0);
}
else if(e.getSource() == bBackgrColor)  // Background Color
{
Color c = (new JColorChooser()).showDialog(this, "Choose Background Color", new Color(ra, ga, ba));
if(c != null)
{
ra = c.getRed();
ga = c.getGreen();
ba = c.getBlue();
viewer.setBackgroundColor((float)(ra / 255f), (float)(ga / 255f), (float)(ba / 255f));
}
}
else if(e.getSource() == bMatAColor)  // Ambient Color
{
Color c = (new JColorChooser()).showDialog(this, "Choose Material Ambient Color", new Color(ra, ga, ba));
if(c != null)
{
ra = c.getRed();
ga = c.getGreen();
ba = c.getBlue();
viewer.setAmbientColor((float)(ra / 255f), (float)(ga / 255f), (float)(ba / 255f));
}
}
else if(e.getSource() == bMatDColor) // Diffuse Color
{
Color c = (new JColorChooser()).showDialog(this, "Choose Material Diffuse Color", new Color(rd, gd, bd));
if(c != null)
{
rd = c.getRed();
gd = c.getGreen();
bd = c.getBlue();
//viewer.setDiffuseColor((float)(rd / 255f), (float)(gd / 255f), (float)(bd / 255f), alpha);
}
}
else if(e.getSource() == bMatEColor) // Emissive Color
{
Color c = (new JColorChooser()).showDialog(this, "Choose Material Emissive Color", new Color(re, ge, be));
if(c != null)
{
re = c.getRed();
ge = c.getGreen();
be = c.getBlue();
viewer.setEmissiveColor((float)(re / 255f), (float)(ge / 255f), (float)(be / 255f));
}
}
```

```
} // end actionPerformed


// this method handles both sliders in the toolbar
public void stateChanged(ChangeEvent e)
{
if(e.getSource() == sAlpha)
{
alpha = (float)((float)sAlpha.getValue() / 20.0f);
viewer.setTransparency(alpha);
}
else if(e.getSource() == sShininess)
{
shininess = (float)(sShininess.getValue());
//viewer.setShininess(shininess);
}
} // end stateChanged

} // end class ToolBox
```

A.9 test.gra

```
Occur ::= State[WRT:Stuff];
State ::= Symptom | Disease | missing[WRT: Stuff];
Stuff ::= Cviatmin | Bvitamin | Avitamin;
```

A.10 cg.bat

```
@echo off
if not x%javaDone% == x goto done

REM **** JDK 1.2 ****
SET java_home=c:\jdk13

SET CLASSPATH=.;%java_home%;%java_home%\lib;%java_home%\lib\tools.jar;c:\
SET PATH=%java_home%\bin;%PATH%
set javaDone=x
:done
echo on

java Cg test.gra
```

A.11 compile.bat

```
@echo off
if not x%javaDone% == x goto done

REM **** JDK 1.2 ****
SET java_home=c:\jdk13
```

```
SET CLASSPATH=.;%java_home%;%java_home%\lib;%java_home%\lib\tools.jar;c:\
SET PATH=%java_home%\bin;%PATH%
set javaDone=x
:done
echo on

del *.class

javac *.java
```

A.12 Tokens.java

```
//
//
// Tokens
//
//
import java.util.*;

class Tokens
{
// token constant declarations
public final static int ID=0;
public final static int DEFRULE=1;
public final static int OR=2;
public final static int LEFTBR=3;
public final static int COLON=4;
public final static int RIGHTBR=5;
public final static int COMMA=6;
public final static int SEMICOLON=7;
public final static int ERROR=-1;

Tokens() {}

}
```

A.13 Makefile

```
all:  dvi ps clean

clean:
rm *.log
rm *.aux
rm *.toc
rm *.dvi
rm *.idx
rm *~
ps:
dvips ontho.dvi -o ontho.ps
dvi:
latex ontho.tex
latex ontho.tex
latex ontho.tex
```

Appendix B

Test of the Parser

B.1 missing-simple-concept.res

```
Test->State :: = Symptom |  | Missing WRT:Stuff];
res ->Error Parsing State : Missing ConceptId/SimpleConceptId after '|'
```

B.2 missing-concept-id2.res

```
Test -> State :: = Symptom | Disease | [WRT:Stuff];
res  ->Error Parsing State : Missing ConceptId/SimpleConceptId after '|'
```

B.3 missing-semi-colon.res

```
Test->Stuff ::= Cvitamin | Bvitamin | Avitamin
res ->Missing last ';' The rule will be not taken in consideration
```

B.4 missing-feature-id.res

```
test->Occur ::= State [:Stuff ];
res -> Error Parsing Occur : Missing FearureId after '['.
```

B.5 missing-concept-id.res

```
test -> Occur ::=State [WRT: ];
res ->Error Parsing Occur : Missing ConceptId after ':'.
```

B.6 missing-left-bracket.res

```
test-> State :: = Symptom | Disease | Missing WRT:Stuff];
res -> Error Parsing Occur : Syntax error.
```

B.7 missing-right-bracket.res

```
test -> State :: = Symptom | Disease | Missing [WRT:Stuff;
res ->Error Parsing Occur : Missing ']'.
```

B.8 missing-rule-id.res

```
Test-> :: = Symptom | Disease | Missing[WRT:Stuff];
res->Error Parsing Occur : Missing RuleId.
```

B.9 missing-colon-in-defining-marker.res

```
test -> Occur := State [WRT:Stuff];
res -> Error Parsing  : Syntax error.
```

Appendix C

Figures

In this section, we will show different views of:

```
Occur ::= State[WRT:Stuff];
State ::= Symptom | Disease | Missing[WRT: Stuff];
Stuff ::= Cviatmin | Bvitamin | Avitamin;
```

C.1 The First View

When we execute **Ontoviz system**, we will get a *Top–Down* view, see figure C.1.

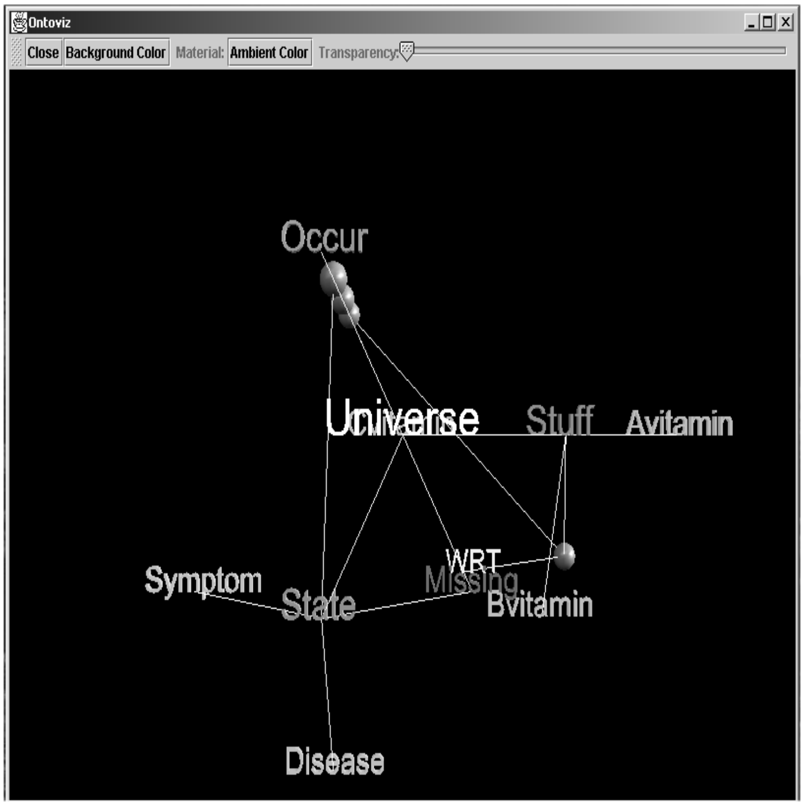

Figure C.1: A Top–Down View

C.2 The Bottom–Up View

By *clicking on the left button of the mouse* and *hold on it*[1], and then move with the mouse so we can see the *Ontoviz tree* from the bottom, we gwe will get a *bottom–up* view, see figure C.2.

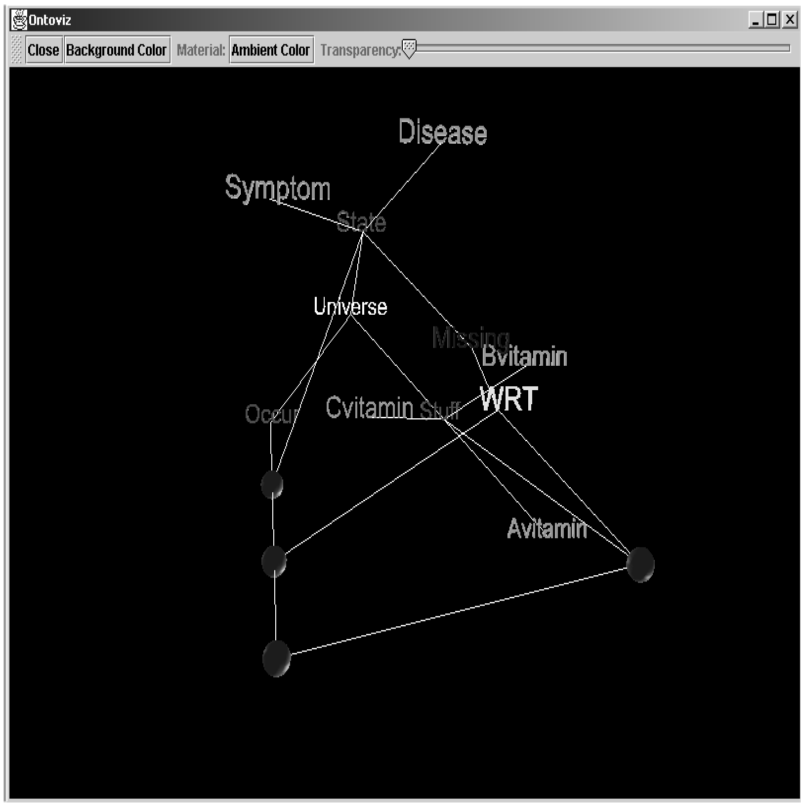

Figure C.2: A Bottom–Up View

[1] I.e. the button

C.3 A Left Rotate View

By *clicking on the left button of the mouse* and *hold on it*, and then move with the mouse, to the **left** for example, we get this figure C.3

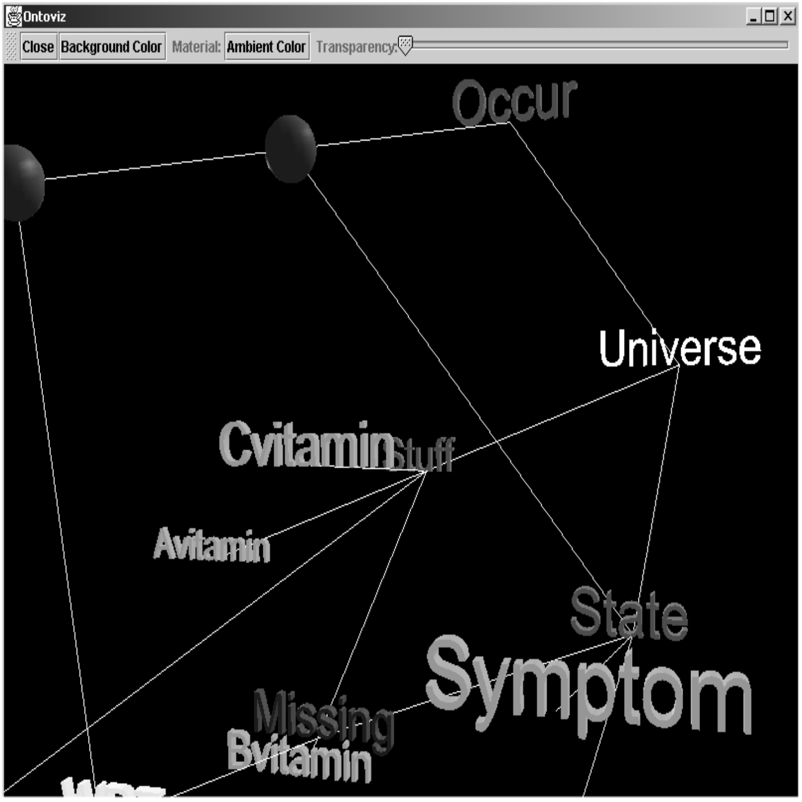

Figure C.3: A View by Rotating to the Left

C.4 A Right Rotate View

By *clicking on the left button of the mouse* and *hold on the button*, and then move with the mouse, to the **right** for example, we get this figure C.4

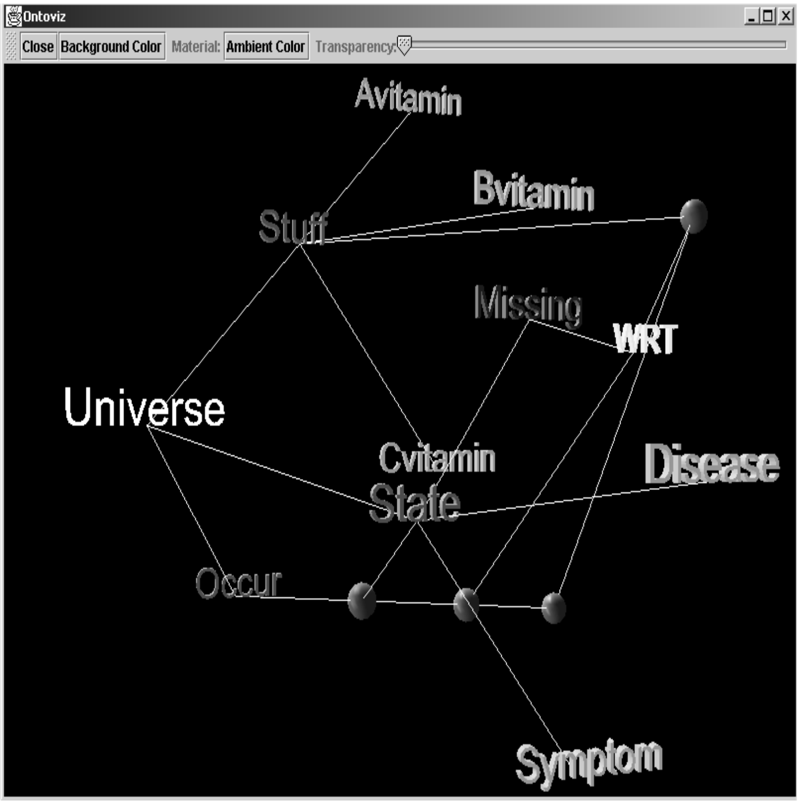

Figure C.4: A View by Rotating to the Right

C.5 A Zooming–Out View

By *clicking on the middle button of the mouse* and *hold on the button*, and then move with the mouse, to the **vertically to you** for example, we get this figure C.5

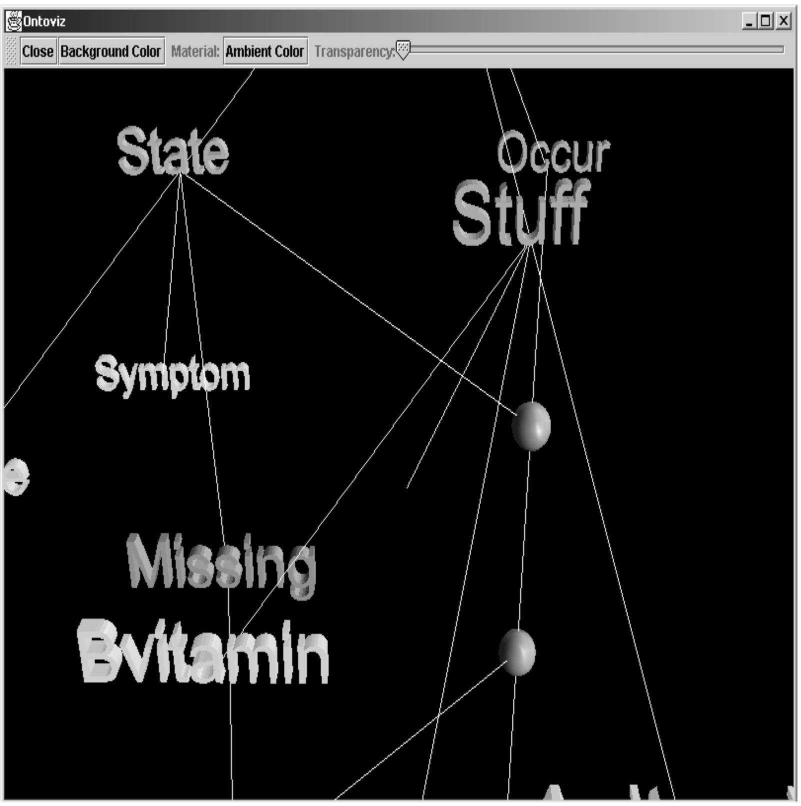

Figure C.5: A View by Zooming–Out to the Right

C.6 A Zooming–In View

By *clicking on the middle button of the mouse* and *hold on the button*, and then move with the mouse, to the **vertically opposite to you** for example, we get this figure C.6

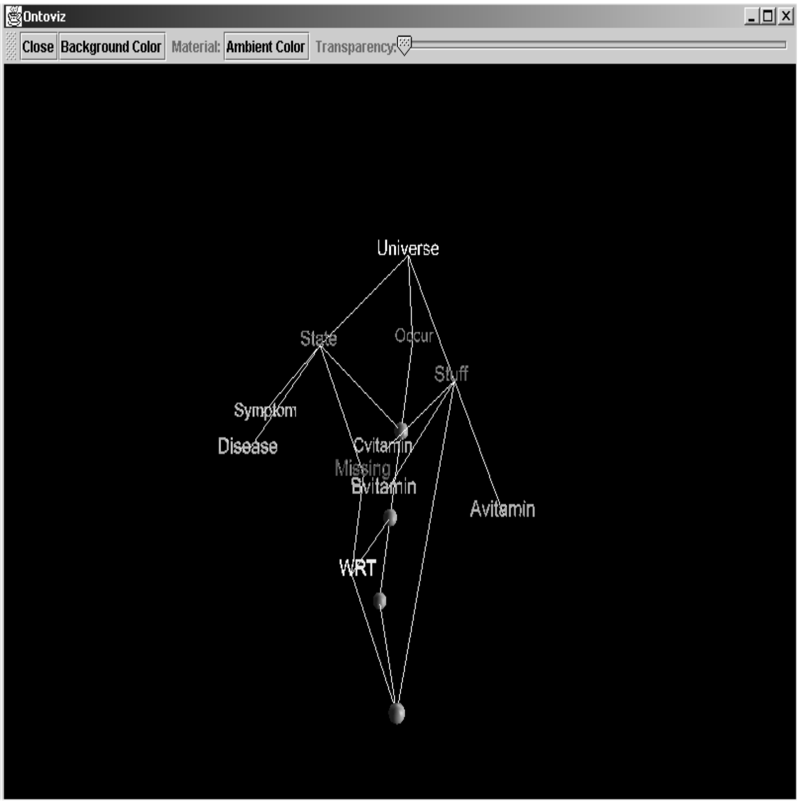

Figure C.6: A View by Zooming–In to the Right

C.7 Left Translating the Ontoviz Tree View

By *clicking on the right button of the mouse* and *hold it on the button*, and then move with the mouse, to the **to the left** for example, we get this figure C.7

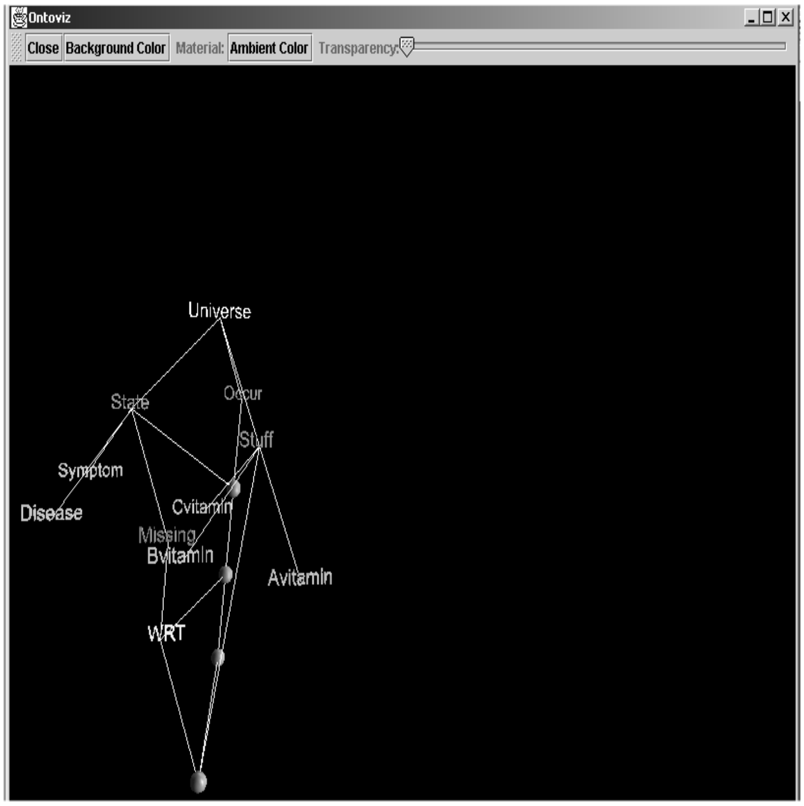

Figure C.7: A View by Left–Translating the graph to the left

C.8 Right Translating the Ontoviz Tree View

By *clicking on the right button of the mouse* and *hold on the button*, and then move with the mouse, to the **to the right** for example, we get this figure C.8

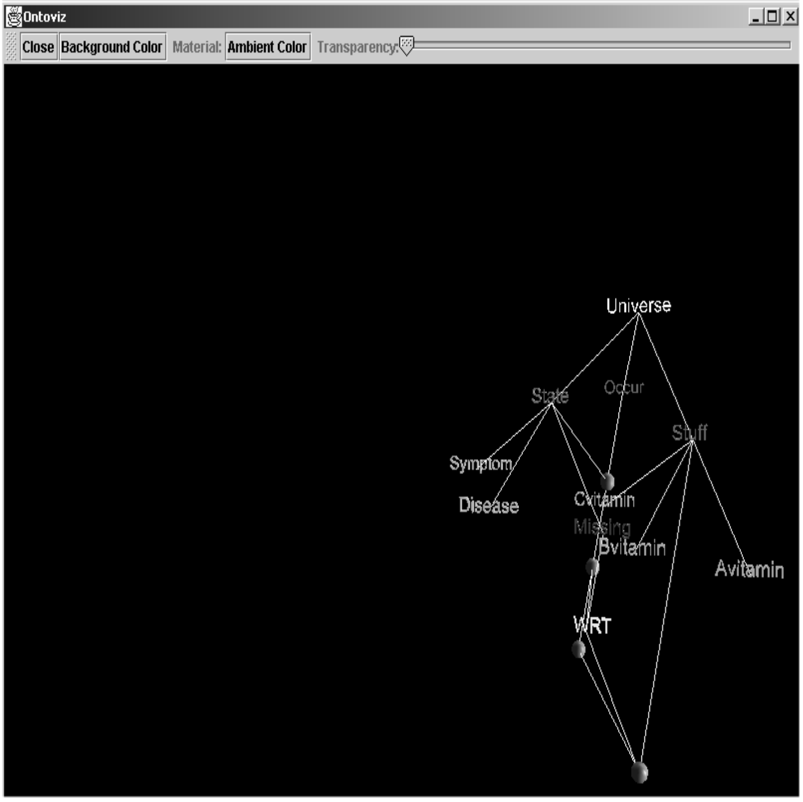

Figure C.8: A View by Right–Translating the graph to the left

C.9 Information Zooming of the Ontoviz Tree

By *clicking **once** on the left button of the mouse*, a little **information** window will appear, giving you some details. In the test we tried is that to get the *key of the node* and its *type*. This is achieved by unsing *PickRay* that pick objects, please refer to section 4.5.4, in chapter 4. The idea is to pick up objects *concept_Ids*, then unfortunately this is not working, we think there is a problem with *PickRay*.

Notice that by clicking with the *left button mouse*, everywhere on the canvas we get the key/id (Universe: the top), and its type is unknown. See figure C.9

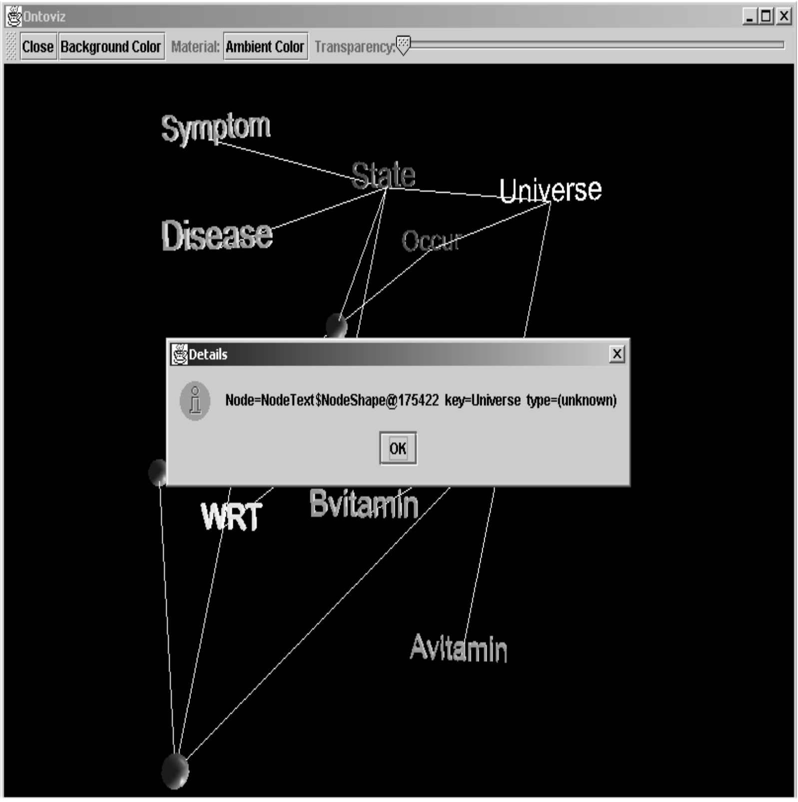

Figure C.9: Information Zooming, using PickRay

C.10 Poping the Tooltip test on Ontoviz Menu

We have made a test for poping a *tooltip test*, that shows this phrase **Graphical Interface To Ontologies**, when the user drag the mouse on the *toolbox menu.* see figure C.10

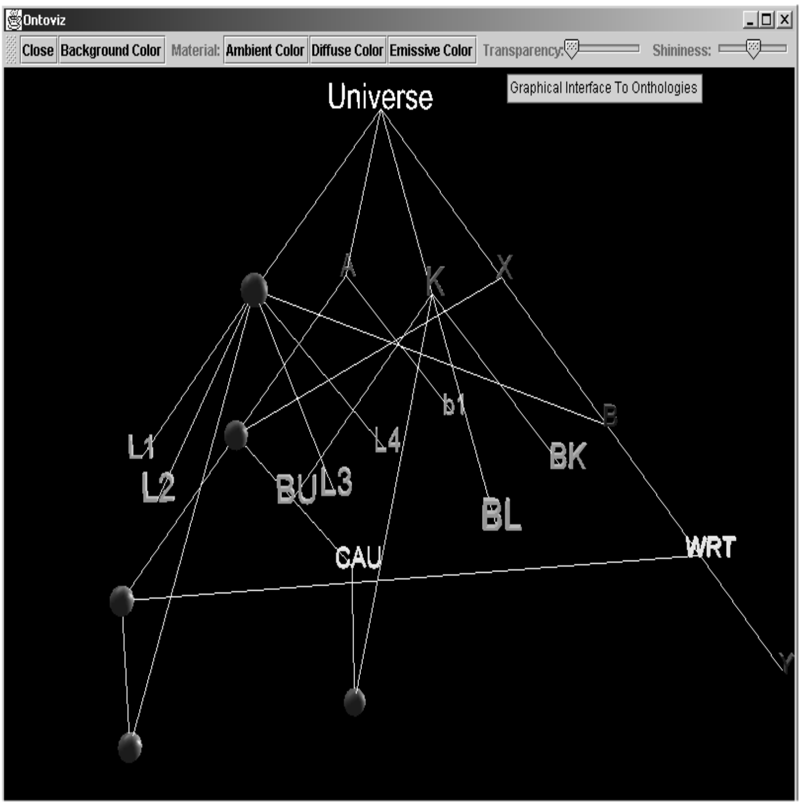

Figure C.10: Poping the Tooltip test on the Menu