TViz: A Taxonomy Visualization Tool

Pedro Maroun Eid

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

April 2005

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: Pedro Maroun Eid

Entitled: TViz: A Taxonomy Visualization Tool

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair
Dr. Vasek Chvatal

_____ Examiner
Dr. Sudhir Mudur

_____ Examiner
Dr. Nematollaah Shiri

_____ Co-supervisor
Dr. Volker Haarslev

_____ Co-supervisor
Dr. Peter Grogono

Approved by _____
Chair of Department or Graduate Program Director

_____ _____
Dr. Nabil Esmail, Dean
Faculty of Engineering and Computer Science

Abstract

TViz: A Taxonomy Visualization Tool

Pedro Maroun Eid

The Semantic Web is an emerging science in the world of computer information processing. RACER, a robust semantic web engine, represents the core of a semantic web system. The engine offers various inference services that return information about a given ontology, for instance, to compute its associated taxonomy of concept names. The taxonomy is generated in a form called the "TBox subsumption hierarchy" although it is not necessarily a true hierarchy. Since humans are more comfortable reading a graphical structure than a textual one, RACER needs a visualization tool in order to visualize taxonomies after processing them. Also, a graph delivers statistical information in addition to its easy interaction capabilities.

In this thesis, we present TViz, a tool capable of visualizing large numbers of nodes representing the TBox subsumption hierarchy, using the Cone Tree layout. TViz, customized for taxonomy visualizations, gets its input from a text file that is created after streaming the information from RACER. TViz consists of three components: (1) a parser that parses the information obtained from the RACER server using a specific grammar, (2) an engine that simplifies and changes the non-hierarchical structure into a hierarchical tree, and (3) a graphics engine that graphs the hierarchical tree using the Cone Tree layout. The graphics engine handles the graphics, the user interface, and tools and is written using OpenGL and GLUI multiplatform libraries. TViz implements useful tools for an easy exploration of a dense environment such as the Compass, the Local View and the Information Window and is implemented using standard C/C++ multiplatform libraries.

Acknowledgements

I take this opportunity to thank all those who accompanied me all through these years of study as well as provided me with moral, intellectual, and financial help.

A special thanks to my great supervisors Dr. Peter David Grogono, Computer Graphics, and Dr. Volker Haarslev, Semantic Web, for their support, guidance, and encouragement towards the realization of this research thesis.

And, finally, many thanks to the Concordia Community, my great and lovely home Montreal, and all my friends, fellow students, and supporting family.

"*Do not follow where the path may lead.*
Go, instead, where there is no path
and leave a trail,"
Ralph Waldo Emerson

To Chawki, Josette, Georges, and Honorée Claris…

…with all my love and appreciation.

Table of Contents

Table of Figures

Table of Tables

Table of Equations

1 Introduction

1.1 Objective

"The Semantic Web is an extension of the current web in which information is given well-defined meaning, better enabling computers and people to work in cooperation." -- Tim Berners-Lee, James Hendler, Ora Lassila, The Semantic Web, Scientific American, May 2001.

The Semantic Web is the combination of large amounts of information and meanings to create an efficient representation of data that can be shared among machines. This is useful mostly to allow machines to efficiently find the information required by web users. The Semantic Web is mainly composed of four components: ontologies, semantic web engines, knowledge bases, and agents. “An ontology defines the common words and concepts (meanings) used to describe and represent an area of knowledge, and so standardizes the meanings.” [32] An ontology is processed using the semantic engine’s inference services to yield a consistent knowledge base. A knowledge base is a formal version of the ontology. Users will then interact with the engine, through an agent, which carries their query and retrieves the relevant data from the engine. Ontology creators now need some tool that gives them an idea about the overall structure of their ontologies in order to verify whether or not they are well elaborated and meaningful. The ontology is passed to the semantic engine which reveals the consistency and the latter produces an associated knowledge base. A knowledge base is normally constituted of two important parts: the ABox and the TBox. The ABox, standing for Assertional knowledge box, holds information about individuals or instances of concepts. The TBox, standing for

Terminological knowledge box, holds information about the concepts and the subsumption hierarchy information. The best way to know if the representation of the original ontology is correct or to look at whatever data it holds is to examine some kind of representation of that TBox. The semantic web engine has a function that streams the TBox subsumption hierarchy. This stream is then dumped into a text file, known by the extension ".tree". This text file can contain many thousands of nodes and can be as large as one gigabyte. The user will eventually get confused when reading this large text file. The dataset size of a manual interaction of a user trying to figure out this set's relations or connections is known to be at most 30 nodes [10]. So, eventually, there is a need for a tool that shows this dataset visually along with the connections between its nodes. Also, it would be used to deliver some statistics about the nodes to be graphed like the concentration of nodes on different levels.

1.2 Approach

The goal of this research is to visualize a huge number of nodes which represent a semantic structure. This structure is generated by RACER, a semantic web engine that is given an ontology as input and after processing it yields a non-hierarchical tree-like representation. A node in this tree-like structure may have multiple parents. The implementation of this project was performed using a component based design. First, a parser was created that parses the input according to a specific grammar. This input, if correct, would be available in a table called the "symboltable" using a list data structure. The symboltable holds a set of triples as defined by the requirements of the input and accumulates all information concerning a certain concept in its corresponding triple node

in the symboltable. A triple is defined to be a record that holds information about the node, its parents, and its children. Next, the symboltable is processed by the engine of the application and is checked for logical consistency; the structure should be changeable to a hierarchical entity. This non-hierarchical structure is then converted to a hierarchical tree that will be used in the layout algorithm. This is done by removing all the factors that make it non-hierarchical, e.g. any multiple inheritance should be made to follow a single parent. Finally, this hierarchical entity is graphed in a certain layout. One of the best layouts suitable for huge hierarchies is the Cone Tree hierarchical layout [14]. This layout models the nodes, their parents and their children in a cone shaped layout.

The main layer that binds all the components of the project is the "Grapher". This code has the function of passing an input data file representing the tree to the parser which in turn parses the input, that is, it checks the input for lexical and grammatical errors, and as a result, stores it in the symboltable. Then the "Grapher" passes this table to the Cone Tree Interface where it is processed and made hierarchical. The Cone Tree Interface initializes a Cone Tree Structure with static attributes, ready to be graphed. Finally, the display function of the "Grapher" graphs the tree and maintains a user interface to allow users to perform some common user tasks for browsing and exploring the tree. Some tools are added to the main hierarchy viewer to make users comfortable with their exploration. The idea of these tools came from the needs that adventurers have when they go on any common exploration of an unfamiliar region. The three most important things these explorers would need to have are a map, a locator that gives them their location and a datasheet that specifies specific data about important points to look for. Using the same kind of reasoning, tools for the system were created and these

include: the Compass which acts as a locator, the Local View which acts as a detailed view of the node in focus, and the Information Window that displays some text info about the actions users are taking and the node that they are focused on. Many more features are implemented in the main viewing window, for example, picking and searching for a specific node. As a final step to test the whole, a synthetic data generator, called the "Stress Generator", was created to stress the application and detect its limits. Finally, a test application was created to test multiple instances of the Stress Generator on TViz. These resulted in a set of output values which are presented for usability statistics. All of these are discussed further in this thesis.

1.3 Background

Previous research that is relevant to this project concentrated mainly on the display of large hierarchies in 3D computer graphics. Various methods were suggested in different technical papers, in addition to many cognitive studies, that deal with the human mind's understanding of graphical and especially large graphical structures. A lot of efficient algorithms and layouts work very well in 2D, however, concerning the capacity of nodes to be displayed, it is obvious that 3D can visualize much more; so 3D visualization was adopted. Previous research suggests efficient layouts for displaying huge hierarchies (>1000 nodes). Now it is clear that one can fit many more nodes into the space using 3D rather than 2D, however, the question is, will there be user comprehension of this dense visual space? Using a specific layout, the graph can be visualized in a structured way. The idea is to visualize something useful but to be careful not to overtax the user's cognitive processing capabilities. In the following, we review

most of the commonly used graphing layouts and methods to visualize large datasets in a structured way along with similar research that was conducted around the same topic.

1.3.1 Hyperbolic Trees

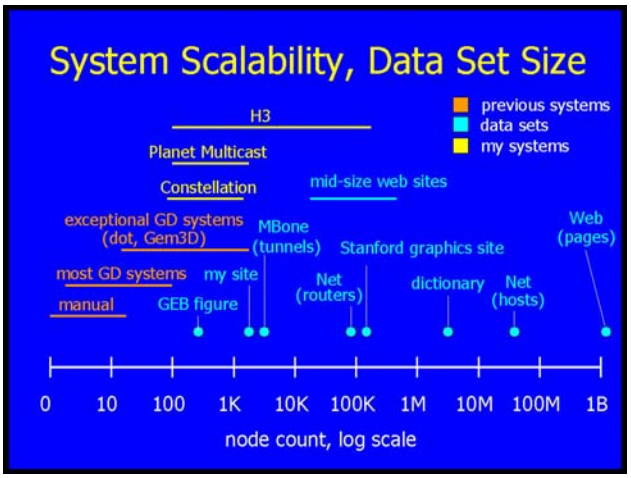

Figure 1: System scalability and dataset size

The Hyperbolic tree layout, a new and well known layout to display large hierarchies as explained by Hong [17], draws very large structures, up to around 120,000 nodes. Hyperbolic trees have been under study since 1995 by Lamping et al. [19]. A pioneer in hyperbolic trees is Tamara Munzner [10 and 20]. Munzner, in 2000, developed a highly stable system; referred by H3, as her PhD implementation. H3 mainly graphs internet hyperlink structures of websites. It scaled to datasets of over 100,000 nodes by carefully choosing a spanning tree as a layout backbone. As a performance rating, H3 displayed 110,000 edges in 12 seconds given DFS input [11]. The scalability of H3 with respect to other existing systems is shown in Figure 1 taken from [10]. As Munzner described, larger trees tend to display as large neighborhoods instead of the global overview which might lead to cognitive disorientation of the user. Hyperbolic trees have innovative algorithms; they are simple structures that make it fast and easy to process large hierarchies. The main idea behind hyperbolic trees is to map the hierarchical structure onto a hyperbolic function or on hyperbolic space. H3 uses a projective model to project the infinite space onto finite 3D

Euclidean space. Another model would be the conformal model, also called Poincaré disc, which preserves angles while plotting the connecting lines onto arcs.

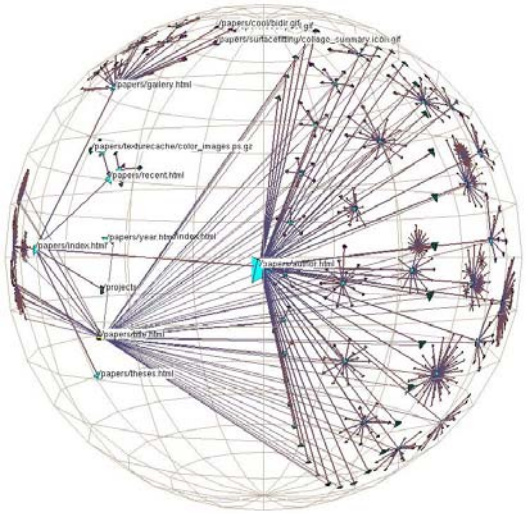

Figure 2: A graph in 3D Hyperbolic space (H3)

The projective model distorts angles while keeping lines straight as shown in Figure 2 also courtesy of [10]. It allows the encoding into 4x4 matrices that can be used to represent hyperbolic transformations. The main reason for using this model is to allow 4x4 matrix operations to be done using the optimized 3D libraries on modern computers and graphics processors. This mapping behaves as a Focus + Context view that shows the local connections of the parents and children of the node in focus while still giving some information about the general layout of the tree. Far nodes appear packed on the circumference. All what is in between has a kind of interpolation effect due to the hyperbolic function mapping. Hyperbolic trees have to be graphed using hyperbolic space in 3D or hyperbolic functions in 2D. Although they are fast and can manage the visualization of many nodes, they tend to disorient users because the layout does not give them a sense of where their position is with respect to the whole space or tree. The root node would appear similar to any child node with the same number of children.

1.3.2 Tree Maps

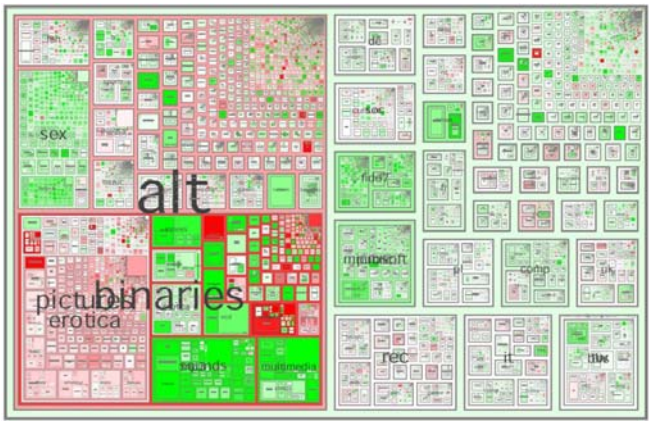

Figure 3: Tree Map Layout and Enclosure Property

Tree Maps, another famous 2D layout introduced in 1991 by Johnson et al. [12], divide the space of each node among its children and then subdivide recursively until the whole screen space represents the given nodes as shown in Figure 3 [33]. However, this layout is best used in software visualizations and business. Some implementations of tree maps have been realized in 3D and these allow the viewing of layers representing the underlying levels as shown in Figure 4 [31].

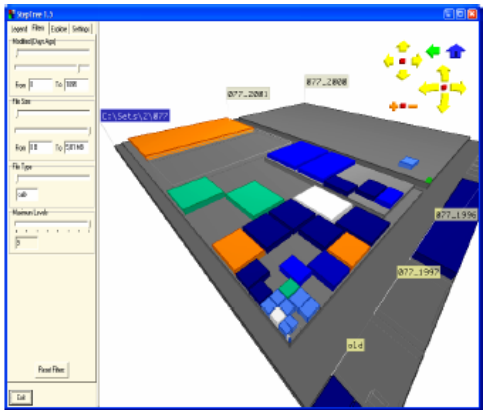

Figure 4: The 3D StepTree Application

The drawback in this layout is that it does not present connections between the nodes. This layout optimizes the use of the available space but it makes it much harder for the viewers to understand the relational structure of the whole entity. Tree Maps are considered to be an enclosure kind of graphing since they include sub-nodes in the node space of their parents usually represented by rectangular areas for optimized area use.

1.3.3 Cone Trees

Cone Trees [14] has been initiated in 1991 by Robertson et al. It has been under development for many years and many improvements have been suggested.

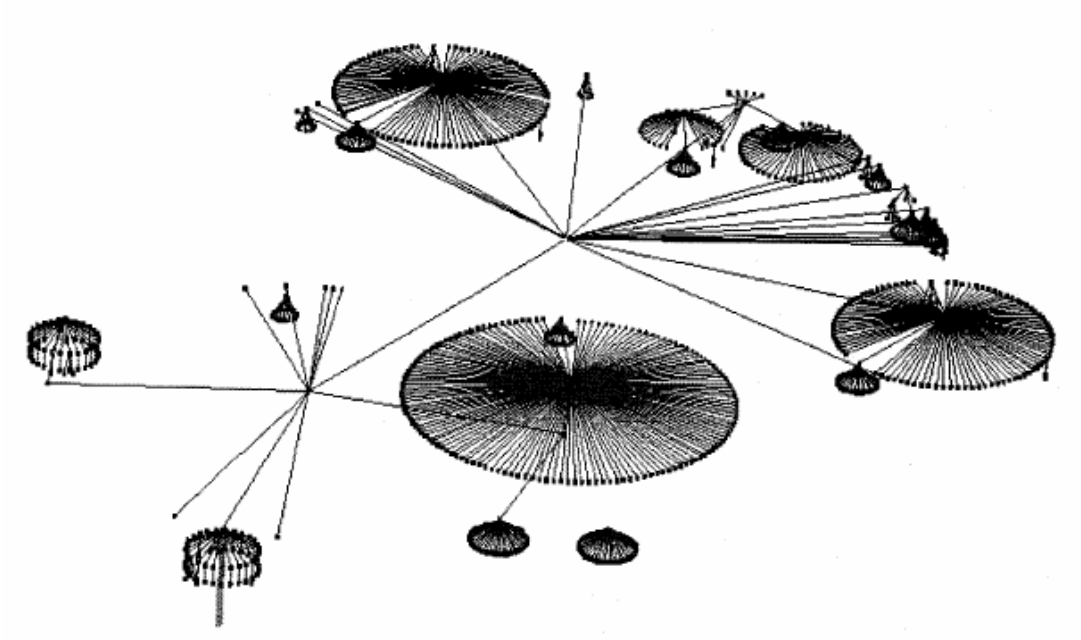

Figure 5: Cone Tree layout

Cone Trees give a straightforward hierarchical representation of the entity in question. They do not alter any of the visual hierarchy representing the semantic meaning of the text. This property is very appealing visually and logically to the user. The drawback using this kind of layout is that it is harder to graph nodes in 3D than it is in 2D so the algorithms are a little more complicated and take more time in processing. The largest number of nodes graphed for a directory structure using the Cone Tree layout given by Carrière and Kazman in [15] is 5000 and, as the authors described, the far nodes with respect to the viewer's eye are no longer identifiable starting with datasets larger than 1000 nodes. Tulip, "a huge graphs visualization framework" [16], is being developed in the University of Bordeaux 1, France since late 2002. Tulip is built to handle 1,000,000

elements theoretically and is able to visualize 110,000 nodes representing a UNIX file system in less than a second as stated by the author. Tulip implements an improvement to the Carrière and Kazman algorithm by finding the optimal enclosing circle that is an amelioration to the enclosing circle used in the Carrière and Kazman algorithm. Finding the optimal enclosing circle is a special case of the general "Smallest enclosing ball" problem as described in [16] and is out of the scope of our research.

1.3.4 Other Techniques

Many other techniques described in different technical papers [25 – 30] have been developed but either they present too much information which makes users unable to clearly reach their objectives or the nodes are drawn in a non-hierarchical layout, which is not what cognitive science recommends for the understanding of graphs and information visualizations. These can only be used by experienced users and eventually they will also face problems using it.

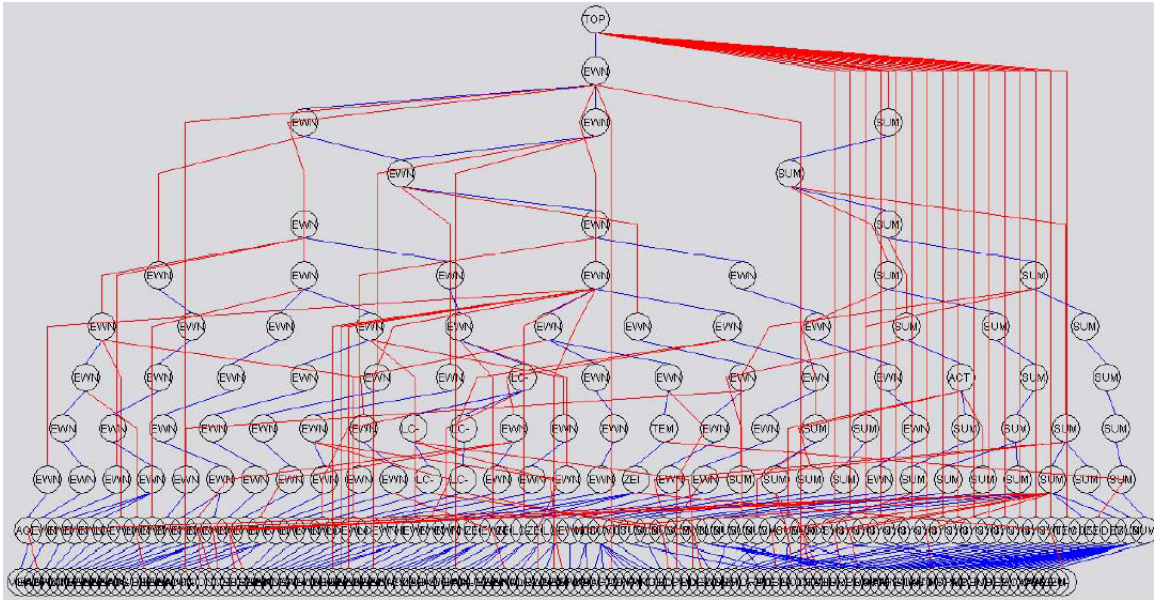

Figure 6: Zarrad's TBox Visualization Tool

Concerning the study of visualizing semantic web taxonomies, some research has been done in the field. Some visualize RDF-based information and ontologies such as in [5 and 6] and others conducted the same research but with unsatisfying results for large datasets. Zarrad [7], in his master thesis, tried to visualize the same datasets but using a layer layout structure in 2D. He encountered some obvious problems such as cluttering, dense spaces and cognitively unreadable graphs when the datasets exceeded 200 nodes as shown in Figure 6 taken from his research thesis.

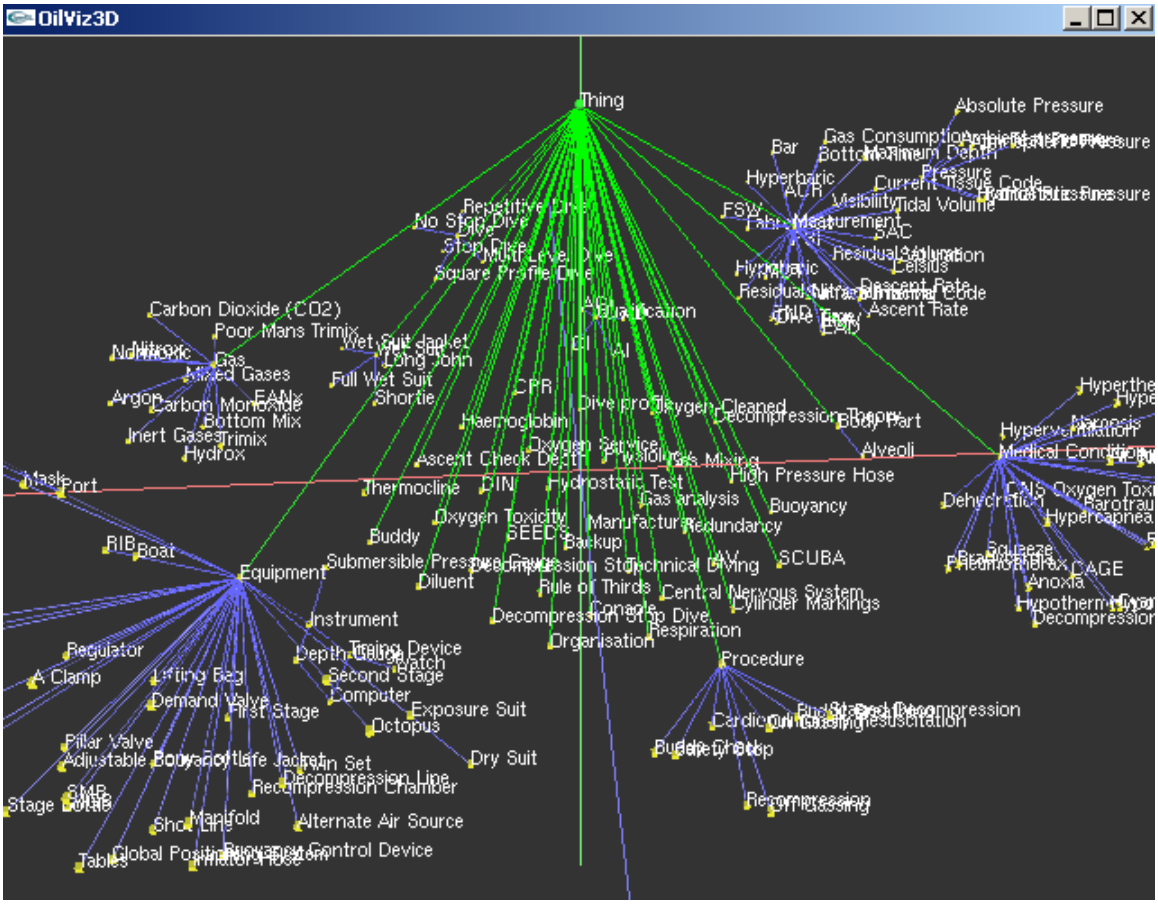

Figure 7: OilViz3D Prototype for a 179 nodes tree

Some other work is also being done in parallel to this study in Manchester University. The work involves the same subject and is closely related to this thesis. Khalid [34] is working on the same concept but with a different approach. The

implemented visualization program, OilViz3D presented in Figure 7, has been integrated into OilEd®. After testing the prototype, we noticed that the layout might grow infinitely in Euclidean space which makes it hard and sometimes impossible for users to perceive large graphs. A spring embedder algorithm is used to stabilize the graph according to connections and densities of nodes after the tree is drawn with the appropriate connections between the children. This stabilization procedure, after reaching its balanced state, should give the smallest possible tree. Moreover, their graph exploration technique is slow since the user can only browse by hierarchy jumping from the parent to the child and vice versa. This technique needs more computational resources than other available exploration techniques and becomes worse as users browse deeper in large trees. Hence, it does not take advantage of the powerful graphics hardware currently present in modern computers. Also, OilViz3D does not implement tools such as searching and picking. It uses OpenGL and Java for rendering the graph.

1.3.5 Summary

Our review summarized the works and the different methods used to visualize large datasets. The applications already created to visualize semantic web taxonomies do not accomplish their task in an efficient way with large datasets and there is room for better tools. More elaborate methods for graphing large datasets have been discussed such as the Hyperbolic Tree, the Tree Map, and the Cone Tree layouts. As a first attempt to our solution, another kind of implementation was tried and is discussed in the following section. Later, we adopted the Cone Tree layout for its capability to handle very large datasets in very little time as compared to the other layouts. It also conserves the look of the hierarchical structure and allows easy graph manipulation and exploration.

This layout needs a hierarchical dataset; to solve this problem, we had to change the non-hierarchical entity into a hierarchical one. The way this will be executed will be discussed later in this thesis.

1.4 A First Attempt

Previous versions to our current solution were constructed as part of experimentation that led to the final result of this research. Basically the parser component had very little changes throughout the versions because it was well defined since the beginning. Previous versions all questioned the effectiveness of the resulting graph layout or visualization. The two previous versions were fruitful experiments but insufficient towards reaching the goal of our research. These experiments had problems in their effectiveness as a viewing layout and were unusable for large trees. A lot of cluttering resulted along with immense processing times because of the graph layout that was not using the screen space effectively and the recursive algorithms that overexploited system memory. Studies and changes were made until the current version was considered to be a good tool for visualizing TBox subsumption hierarchies created by RACER.

One of the previous versions was a 2D connected graph that represents the taxonomy in question. In this layout, the nodes were put on random positions on the screen, and connecting lines linked the nodes together. This graph layout was a test platform for more research to have a lookout of common problems we could encounter. It showed to be unreadable in the case of more than around 30 nodes. Another layout had to be considered for our goal of efficiently visualizing around 15,000 nodes.

Another version then was thought of and realized. Since the taxonomy starts with top and ends with bottom, the whole layout could be represented as a circle in 2D or as a sphere in 3D. The top node would be placed on the North Pole, the bottom node would be placed on the South Pole, and all the in-between nodes would be placed consequently somewhere in between. In this case, a good layout would be a layered layout that could sort the nodes each on its respective layer and finally graph these nodes on static layers constructed on the circumference of the sphere as shown in Figure 8.

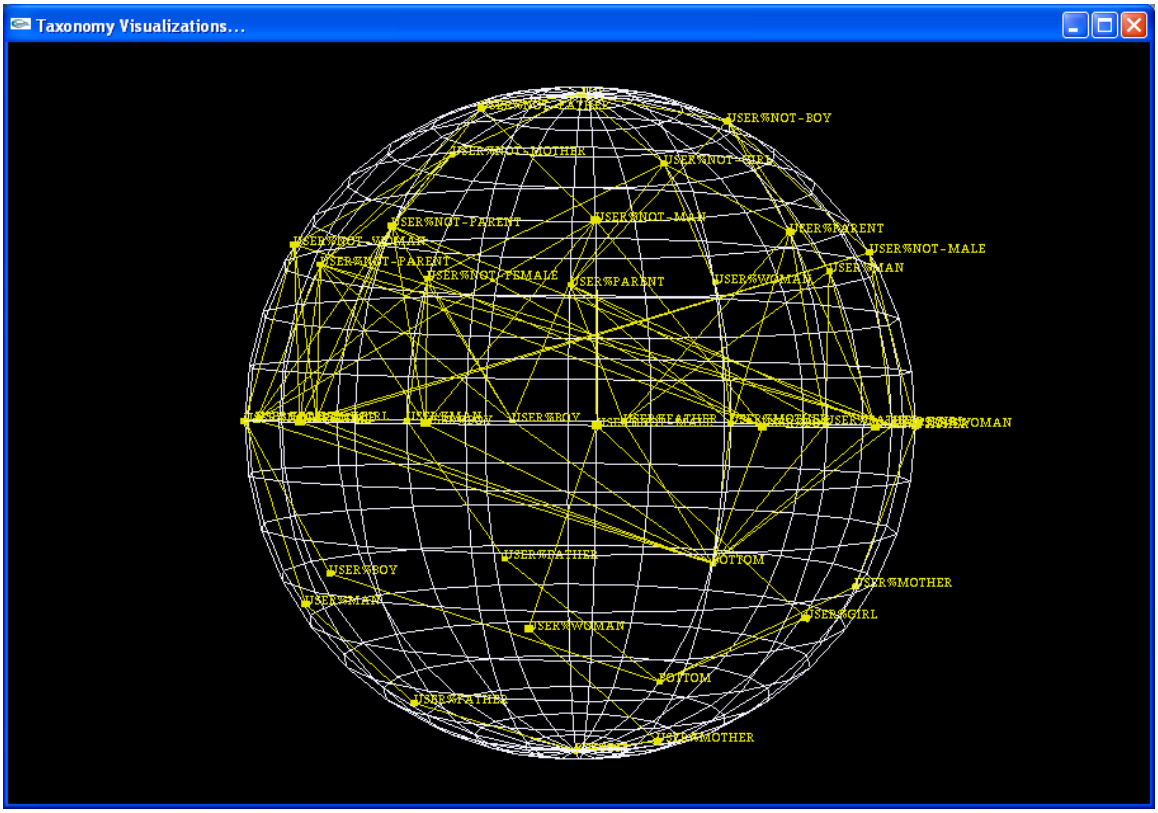

Figure 8: Previous version of TViz

However, we faced three main problems dealing with this layout. The first problem was the occasional existence of a heavily densed layer when the tree had many of its children appearing at a certain depth (on this layer). This heavy layer was considered to be good statistical information but it was not easy to interact with. This also made the sphere

layout unusable for certain datasets. The second problem was that, since our layout represents nodes as cubes and relationships of nodes as edges and the interconnections between the nodes were an important part of the graph, the sphere layout was not showing these explicitly; the edges were hard to order and this resulted in too many crossings. When the nodes are sorted into their layer list even though all the lists were sorted in the same manner, their logical connections were unreadable. A node on one side of the sphere was to be connected to a lower layered node on the other side of the sphere while its parent is also on the other side. Fixing this was a complex problem and needed calculations about how to graph the layer list along with some decisions about where to start on the circumference of the sphere. More code had to be added to the already complex algorithms. The sphere layout easily handled 1000 nodes with their connections, but they were not perceivable. The third problem was the processing time of our algorithms. The algorithms for the layered sphere layout were not affordable with respect to what has been found in the review about other visualization methods. They were recursive algorithms to go into a certain depth and extract all the nodes in the tree which are considered to be at this depth. In case of multiple inheritance, the solution was that a node already put on a layer is not put in any layer again. For large trees of more than 2000 nodes, our algorithms made Windows-based machines run out of memory after hours of processing times. This layout also made trees of more than 500 nodes very cluttered and heavy to the cognitive mind of the user. The sphere layout was simply not interesting for our problem and certainly cannot handle the large datasets that we were targeting.

1.5 Current version

The current version of the project is the fruit of all the previous experiments. It is built by interconnected components written all in C/C++ being able to be used again in similar projects. Throughout the whole process of construction, we kept in mind further portability of the project to other system platforms, mainly UNIX/X-windows and Mac OS®. This is why we built the project with standard libraries and OpenGL along with GLUT and GLUI. All of these components are recognized to be platform independent.

The project is intended to be used as a tool for visualizing taxonomies as described under our goal section above. Subjects of ontologies vary from personal to big companies and their number is constantly increasing. So, it is clear that the market of the project is broad and could lead to a very interesting state of requests. Therefore, the tool should be as a whole usable, fast, and efficient to meet its needs and the users' expectations. It should be customized for semantic web datasets. It should also provide an easy interface for novice users and advanced tasks for experts. This whole version is discussed in details in the following sections of this thesis.

2 The Parser

2.1 The Grammar

The first component of the project is a parser that is capable of reading a consistent input from a file. Optimally, this parser would have to parse input from a network stream. An example of this input would be:

(WOMAN (PERSON) (OLDLADY))

Figure 9: a Triple example

This example represents a simple Triple that defines a notion called WOMAN which has PERSON as a parent node and OLDLADY as a child node. The problem with this representation is that there may be further details discovered later while parsing, for example, another child for WOMAN. In this case, the parser should not crash when finding the concept already defined. Throughout this thesis, we will denote a concept name(s) by NodeNames represented by one unique name that defines the node and its Synonyms, if they exist. A node might have many parents as well as many children. It is at a later phase where multiple inheritance is dealt with to ensure proper layout. The parser is constructed using the BisonFlex compiler generator. According to [1 and 2], Bison is one of the most efficient compiler generators ever created. BisonFlex was chosen because of its ease of use, easy debugging, and capability of handling very large sets of information in reasonable time. RACER outputs the information to be visualized in text mode through a network connection by simply querying it with "(taxonomy)" over a TCP connection. The result of this query is then dumped into a file. Our compiler then reads from this file. In case of a network connection, a small application would be used to

connect to RACER and dump the taxonomy into a file which is then given to TViz. We define the grammar that is used to read the input and to ensure its syntax in Table 1.

```
1   Triples: /*empty*/
2       | Triples '(' NodeNames Parents Children ')'
3       ;
4   Parents: NAME
5       | NAMETAG
6       | '(' NodeList ')'
7       ;
8   Children: /*empty*/
9       | NAME
10      | NAMETAG
11      | '(' NodeList ')'
12      ;
13  NodeList: RNodeNames
14      | NodeList RNodeNames
15      ;
16  RNodeNames: NAME
17      | NAMETAG
18      | '(' RNameList ')'
19      ;
20  RNameList: NAME
21      | NAMETAG
22      | RNameList NAME
23      | RNameList NAMETAG
24      ;
25  NodeNames: NAME
26      |  NAMETAG
27      | '(' NameList ')'
28      ;
29  NameList: NAME
30      | NAMETAG
31      | NameList NAME
32      | NameList NAMETAG
33      ;
```

Table 1: Grammar used to parse the tree files

Line 2 in Table 1 shows the grammar definition of a Triple. Triples are defined one after the other and are recognized by their enclosing parentheses. They hold three main components: the NodeNames, the Parents, and the Children. NodeNames is made of a Name or a NameTag along with some synonyms, if they exist, as shown in the definition starting on Line 25 of Table 1. A Name or a NameTag both represent one single name. The only difference between these notions is that NameTags are names enclosed by "|", e.g. "|dog|". NameTags can also have anything in them including a quotation mark, a parenthesis, or any irregular character. A Name, on the other hand, is a strict combination of letters and digits. Name and NameTag definitions are presented in Table 2. Parents, if made of one Name or one NameTag, could appear without being enclosed by parentheses. However, once the definition is made of many nodes, it would be called a

nodelist and it should be enclosed by parentheses. A nodelist means a definition of one or many nodes. Now a node as said before could be a Name or a NameTag, however, it could also be made of a namelist. A namelist is a node that has many names separated by an empty space defining a unique nodename and its synonyms. A namelist has to be enclosed in parentheses. For instance, the definition of a node that has only one parent but this parent is a namelist would be: (nodename ((oneparent1 oneparent2))). If the parentheses are omitted, it would be considered to have two different parent nodes rather than one with two names. A namelist is made of one or many Names or Nametags. Children have exactly the same definition as Parents do with only one exception. A node or Triple has to have at least one parent defined whereas it might have no children defined. Since different actions should be taken for a namelist that defines a nodename and for a namelist that defines a parent or child, then these two definitions are split. As shown in Table 1, a namelist defined as a nodename and a namelist defined as a parent or child have both the same grammatical structure but are only split in order to allow the consequent actions to be called respectively by the parser.

```
%%
delimiter       [ \t\n]
ws              {delimiter}+
comments        (;.*\n)
NameTag         "|"([^"|"]|(\"))+"|"
Name            [^ \t\n\(\)]+
%%
{comments}      {/*delete || do nothing*/}
{NameTag}       {TreeParserlval.nm = (char *) strdup(yytext); return NAMETAG;}
{Name}          {TreeParserlval.nm = (char *) strdup(yytext); return NAME;}
{ws}            {/*delete || do nothing*/}
.               {return TreeParsertext[0];}
%%
```

Table 2: Lexer definitions

Each definition in the grammar has a different associated action. A namelist that is found as a nodename is dealt with differently than a namelist that is found as a parent or child. Table 2 defines the lexer regular grammar, this grammar is just used to recognize Names and NameTags and return them to the parser. It is also made to discard white spaces denoted by 'ws' and comments. Comments are known to be starting with a ';' at the beginning of the line and ending at the end of the line. Anything other than those defined are returned to the parser and the parser would judge if the input is relevant or not. For example, enclosing parenthesis other than those found inside NameTags are not taken care of in this lexer, they are passed as is to the parser which decides their relevance and meaning.

The parser is a tool that recognizes certain strings and performs an associated action when one is found. Therefore, we should also carefully define the actions that should be taken in the case of a token being detected. The name of a node is considered to be unique. A namelist is considered to be basically a unique name with some synonyms. For instance, when a new triple is found, the name, also known as nodename, of this node is checked if it is already defined and either locates its definition or defines it. In case of a namelist, the parser searches also the synonyms because the same namelist can be ordered differently at every occurrence. If we take the example explained in the previous paragraph, the namelist can be either a definition of a nodename or a parent or child. If it is a definition of a nodename, then the parser should look if this namelist is already defined or not. If it is a parent or child then we need to take the located nodename and check its data. Then, using the location of the node, the parser inserts new found data under that node's position. If, for example, a node appears with one of its children once

and then it appears again with another child, each child is inserted when found under the same node space. When a node's child is to be inserted, the parser checks if that child does not already exist in the children list of that node. The child is inserted in the node's list of children and, at the same time, the node's child inserts the node's name as one of its parents. This would result in a lot of repetitive data but would also make sure that none of the nodes in the tree is missing information that belongs to it. The nodes are inserted in a list data structure, called the "symboltable", in the way they are ordered in the file or in the way they are read from the input. Since we are using a list structure and adding nodes as we are finding them in the input, the result is an unsorted list which is slow to process by having to deal with the parents and children links that connect the nodes together. Thus, further sections in this thesis will describe the ways used to translate to an array structure and how a linktable is used to sort this array for faster node position calculations. These topics will be discussed in the engine part of the research which is responsible for translating the non-hierarchical unordered structure into an ordered hierarchical dataset ready to be graphed. The next section will describe the symboltable mentioned above and its details.

2.2 The Symbol Table

The parser interacts with one structure called the "symboltable". This structure is a list data structure which holds and can handle all the tasks needed to define a node with its attributes and its relations. The symboltable is a globally defined table, using the 'static' scope. It is used in the whole project to access node details. The symboltable is, however, a dynamically sized linked list that is capable of holding any number of nodes

given. There are two main interfaces for the symboltable in this project. One is the use of the list structure to define and maintain the concepts and notions found in the input by the parser, and the other is the fetching of information to be used in the layout calculations and in the information display. After the system runs through the parsing stage, the symboltable is never altered again and is only used as a reference.

2.2.1 The Structure

The first interface of the symboltable includes functions such as creating a new record which is basically a Triple in the table. This Triple is always added to the end of the list if it is not defined earlier. The reason for this is that if the list search for this Triple resulted in no node defined earlier, then adding it at the end is basically the fastest way to add a node to this list without having to change any previous index references. When a new Triple is inserted, the newly created id is returned. Otherwise, if the node is already defined in the symboltable, then the node's id is returned by the lookup function.

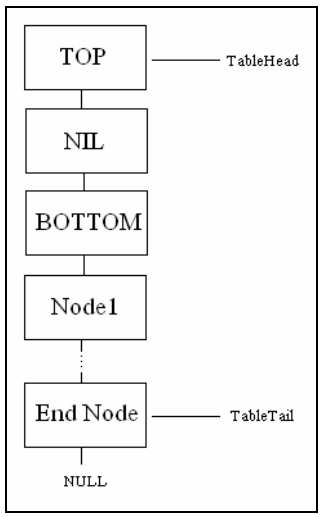

Figure 10: The Symbol Table

In both cases, an id is returned and this permits other functions to use this id to enter related information in that node's space. The node's id is just a data type of type 'long' that defines the position of this node in the respective linked list. The id's '0', '1' and '2' are reserved for the 'TOP', 'NIL' and 'BOTTOM' nodes respectively which should exist physically or logically in any hierarchical structure. This is done to preserve the sequence of appearance in the symboltable of these critical nodes. 'NIL' only links 'TOP' and 'BOTTOM'. If bottom-up layout is adapted, then 'BOTTOM' is the active starting point

and if top-down layout is adapted, then 'TOP' is the active starting point. This flexibility makes the symboltable easily usable by any layout graphing engine. Triples are defined in the following section. Figure 10 shows the symboltable's definition. The second interface is a direct access to the information where a pointer to the whole symboltable is passed to the other system tools. Those would be able to use it to fetch information they need about a specific node, e.g. the number of children this node has. However, rather than dealing directly with the linked list, the symboltable has methods to give a sense of abstraction to the tools using it. These methods also include some useful functionality such as error and integrity checking. These would be used to extract statistics such as the number of nodes on a specific level and to manipulate instances of nodes such as inserting a new node.

2.2.2 Triples

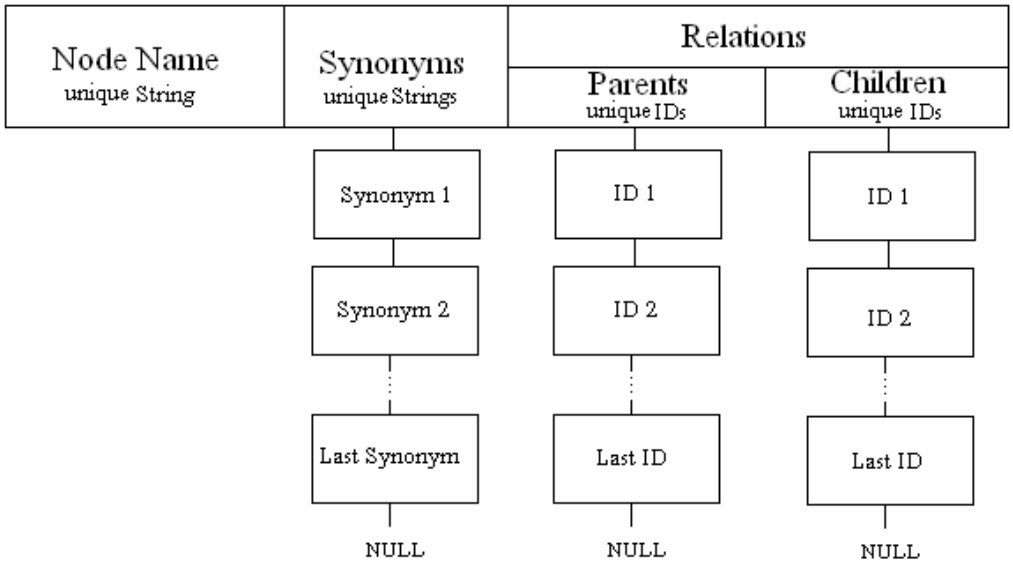

Figure 11: A Triple

The symboltable, as said before, is a linked list of triples. Each node in the symboltable is made of three parts: the node name, the list of synonyms found for this

node name, and the relations of this node. These are shown in Figure 11. The nodename string and its synonyms can be made of any number of characters as long as they fit in the user's active system memory. These are stored using their full strings to be able to refer to them later and be able to compare them with other instances that hold similar names or synonyms. The synonyms of a node are represented by a linked list because we cannot foretell how many synonyms a node has before the end of the parsing stage. In case that a node does not have any synonyms, then no list is created at all. Synonyms might be found as we proceed in reading the input file as is the case with any new information. Consequently, the symboltable is easily manipulated and controlled by only inserting what is new. The relations of a node include two more linked lists and these are called the children list and the parents list. The parents list will eventually contain at least one instance by the end of the parse of a single triple because of the requirements of the grammar. It is assumed that the parser would take care of checking the input before inserting it into the symboltable. The relations' lists are both stored as ids. This means that we cannot change the sequence of the nodes in the symboltable without checking the parents and children lists and making the necessary corrections. The parents and children lists are built by trading memory with speed. We allow redundancy only to have faster data access.

3 The Cone Tree

3.1 The Cone Tree Structure

The Cone Tree Structure is a strict hierarchical structure that contains the same number of elements existing in the symboltable. Since the symboltable holds all the possible existing elements whether they are linked to something or not and that the parser would take care of inserting elements that correspond to the grammar definitions stated above in Section 2.1, then at least one instance of each element of the existing symboltable entries should be found in the Cone Tree Structure, if there is no break in the logical hierarchy.

3.1.1 The Hierarchy

Any inconsistency found while parsing relating to the grammar will be reported but the logical interpretation of the tree cannot be found at parsing time and should be dealt with while constructing the Cone Tree Structure. For example, there might be a set of nodes generating a loop that might be disconnected from the whole graph. In this case, the parser cannot detect this situation but the Cone Tree Structure will recognize it. The symboltable is an unordered, non-hierarchical structure as it can hold multiple inheritance and is created as the file is being read. To graph using a Cone Tree layout, we need a strict hierarchical structure. In our implementation, to transfer the non-hierarchical symboltable to the hierarchical Cone Tree Structure, the algorithm in Table 3 is used. This algorithm yields a sorted list of nodes in the order of processing so that no dependent node is processed before processing the nodes it is depending on. This

algorithm sorts out the parsed nodes into a drawing list. The resulting list defines a top-down layout because the topmost nodes are all drawn and any further repetitions of these nodes in the logical structure would be omitted. The total number of nodes that the sequence list is initialized with is defined by the symboltable size to ensure that all the defined nodes will appear in the sequence list. The sequence list is actually a logical queue data type, where the nodes are inserted at the end and the processing happens from the front.

```
sequencelist = top node;
node = sequencelist[0];
while(not end of sequencelist)
{
    for(all children of node)
    {
        if(child is not already drawn)
        {
            mark child as drawn in the drawnnodes list;
            add child to the end of sequencelist;
        }
    }
    node = next node of sequencelist;
}
```

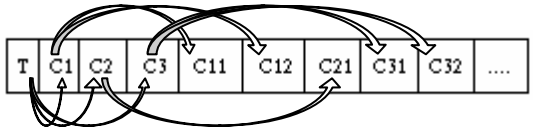

Table 3: Sorted Sequence

This simple algorithm is used to transfer the logical hierarchy into a strict hierarchical one; no node is to appear twice in the graph. This algorithm supposes that as long as the total number of nodes is not reached, then the list always contains a child inserted from another node at the end of the list. If a break in the tree exists in the logical structure, the index used to add the children at the end of the sequence list and the index where the current node is being processed may overlap. A simple test condition would ensure that, if the two indexes overlap, a logical break in the tree would be detected. This overlap would happen in logical hierarchies which contain cycles or some unconnected set of

nodes to the whole tree. In this stated case, the Cone Tree Structure would return an error and no graphical tree is generated; the input is considered erroneous. The algorithm in Table 3 is easily upgradeable and with only changing the first assignment of topnode to bottomnode along with checking the parents instead of the children in the internal loop, this algorithm would yield in a bottom-up sequence list which could be used through the whole process again to yield a bottom-up graphic representation. However, a bottom-up layout was not implemented in our current solution.

3.1.2 The Layout

The Cone Tree Structure uses the Carrière and Kazman algorithm [15] to set up attributes to graph the resulting hierarchical structure. The Cone Tree Structure is an array of nodes with the same size as the symboltable size. Each node is assigned a 3D position and a list of its graphical children as well as some attributes that contain important information to be used for graphing the layout and for some statistics. The Cone Tree Structure has two main methods to yield in a 3D position for each node. Both methods use the node sequence array that was constructed using the sorting algorithm shown in Table 3. Before any of these methods are accessed, an algorithm, explained later, binds the symboltable linked list nodes with their corresponding positions in the sequence list. This binding algorithm creates a structure called the linktable. It is much faster to access nodes using their pointers rather than having to browse through the linked list to reach the node in question. This linktable makes it much faster to access the corresponding node in the symboltable and binds the node's pointer with its occurrence in the sequence list. It is used throughout the Cone Tree Structure.

3.1.2.1 The Cone Base Radius

The first method, called generateRadiusBasedTreeLayout(), is called to setup the Cone Tree with the linktable as an argument. It starts by initializing the sequencelist which has one and only one instance of each of the symboltable nodes. This sequence is then used in the whole Cone Tree Structure and by the next function, described later in this text, as a guide to the order of processing of the nodes. This makes each of these functions run as a linear process rather than having to manipulate the n-branched tree. ComputeNodePositions() is a function that assigns the positions to the nodes based on the Carrière and Kazman algorithm. This function is split into two parts. The first part calculates the radius of each cone, which represents a node and its children, and stores the largest child radius of each node needed for further calculations. The radiuses are calculated based on a predefined initial radius. Since our tree is logically ending by the bottom node always, then a good method of calculation would be to set the bottom node to the initial radius. All graphically connected nodes to 'BOTTOM' would be assigned that initial radius and calculations continue further on up the tree using the equations in the following table:

$C_{n-1} \cong 2\sum_{i} r_{i,n}$ $r_n = \frac{C_n}{2\pi}$	$R \cong \frac{\sum_{\forall i} r_i}{\pi}$ $BR = R + lcR$

Equation 1: Carrière and Kazman and our version

On the right of Equation 1, $\sum_{\forall i} r_i$ represents the sum of the children's radiuses which yields in an approximation to the half of the required circumference of the current cone.

R would be the radius that the children nodes need to be graphed on. We call this radius the rendering radius. R is basically summing up the two equations found on the left-hand side of Equation 1 that are presented by Carrière and Kazman. When added with the largest child radius of the node that is currently being processed, R would ensure the sufficient space needed by the node's cone so that cones from other nodes would not overlap with it.

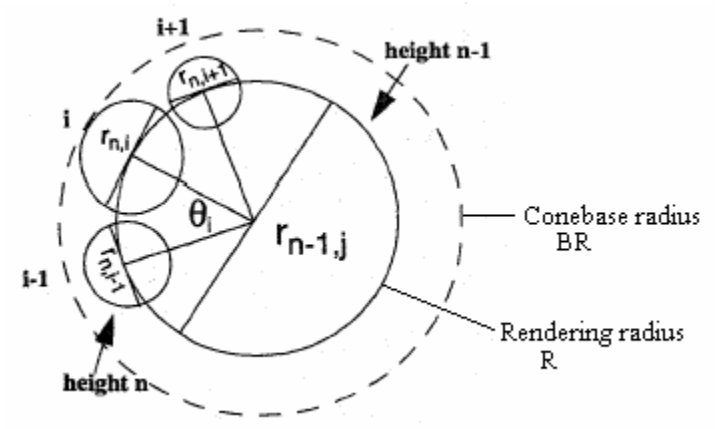

Figure 12: Carrière and Kazman

We denote the largest child radius by lcR. We call this enclosing radius the conebase radius denoted by BR. The graphical difference between the two radii, R and BR, is presented in Figure 12.

3.1.2.2 3D Node Position

The second part calculates the positions of each node starting from the top position. The top node has a statically defined position. The children of 'TOP' would be the next to be assigned positions and so forth until the end of the tree is reached using the sequence list as a guide. The position of each node is determined by first calculating an angle ϑ between two nodes, then assigning the node position with respect to the parent position with a height value that goes down the tree. The angle between two children nodes is determined by the arc that these children are taking on the circumference of the rendering radius of their parent. The angle ϑ between two nodes and the position of the

child being processed are linearly computed by a polar coordinate system assignment using the formulas on the right of Equation 2. These functions ensure the proper positioning of children with respect to their parent according to the Carrière and Kazman algorithm. However, with this layout, a tree can grow deep and wide to a state where it passes much beyond the bounds of the viewing frustum of the user. Therefore, the depth of the tree is made static by assigning a static position to the bottom node and calculating h as the distance between the top and bottom nodes divided by the total number of levels obtained for the whole tree graph.

$s_i \cong r_{i-1,n} + r_{i,n}$ $\theta_i = \frac{s_i}{r_n} \cong \frac{r_{i-1,n} + r_{i,n}}{r_n}$	$\vartheta_{i-1,i} \cong \frac{BR_{i-1} + BR_i}{R}$ $P_{x,i} = P_x + R \times \cos(\vartheta_{i-1,i})$ $P_{y,i} = P_y + R \times \sin(\vartheta_{i-1,i})$ $P_{z,i} = P_z - h$

Equation 2: Calculating Positions

This technique makes the height of the tree constant for all trees on a given coordinate system. The width of the tree cannot be determined by a similar technique because it depends on the set of radiuses that were calculated and is basically summing them up the tree in the way they were constructed and computed. Altering the initial radius would yield in smaller widths of the whole tree but to know what the initial radius should exactly be needs extensive study based on the number of nodes on one level and their layout. A way to make the radius of the whole graph semi-static is bounding it by considering that all the nodes in the tree are under the top node. This means that there is one single parent. If this is the case then the largest circumference would be known since it is defined by Equation 3. And this would simply lead to the maximum radius this tree

would occupy in space (worst case). Therefore, one might think that choosing an initial radius would be good using Equation 4 where R would be the static radius of the whole tree. However, this calculation often results in a bad layout since most subsumption hierarchies have rather large subtrees. Our experiment lead to a bound tree but which is distorted by height. The strategy of using the total number of nodes to determine what the initial radius should be remains an open problem which surely involves further research. However, for the time being, we chose to preserve the nice looks of the tree rather than fixing the radius which results in a not-so-interesting visualization.

$$C_{\max} = 2 \times initialRadius \times totalNumberOfNodes$$

Equation 3: Maximum Circumference

$$initialRadius = \frac{\pi \times R}{totalNumberOfNodes}$$

Equation 4: Initial Radius

If any error occurs during any of the discussed processes, the called function would return -1 as an error notification. This error would be most of the time due to logical errors in the input. One of the best features the generateRadiusBasedTreeLayout() and the ComputeNodePositions() have is that they are decoupled in a way that makes it possible to easily implement other layouts such as the bottom-up graph layout discussed earlier or even other kinds of layouts. The two functions are both linear in terms of execution process; each node is processed only once. The parent node determines its radius from the sum of its children's radiuses which are supposed to be already computed because the sequence list ensures that no node which is a child of a node appears before its parent. And, similarly, the node positions are computed using the same architecture by specifying the top position and then generating positions for its children and so on as shown in Equation 2.

3.2 The Cone Tree Interface

The Cone Tree Interface is an interface to create, manipulate, and graph the interpreted tree created using the Cone Tree Structure. The Cone Tree Structure ensures that the tree is a strict hierarchy. One of the most important jobs that the Cone Tree Interface does is decoupling the actual symboltable from the Cone Tree Structure by constructing a linktable that links the nodes in the symboltable with their corresponding Cone Tree nodes. This is done by keeping a pointer in the linktable to every node in the symboltable at the setup time of the Cone Tree Interface. This makes all nodes fast and easily accessible through one interface and renders both structures coherent. It is more sensible to use the linktable rather than accessing both structures independently. The Cone Tree Interface need only be given a symboltable structure. The interface then sets up the linktable to bind the corresponding nodes of the symboltable and of the Cone Tree Structure together. It then passes this linktable to the initialization of the Cone Tree Structure. An earlier version initialized the Cone Tree Structure first using the symboltable as a reference and then constructed the linktable to be only used in the graphing function for fast rendering access. This method was found to be slow when constructing the Cone Tree Structure so a better way is to initialize it by using the linktable instead of the symboltable. This technique resulted in huge performance differences in the construction of the Cone Tree Structure and in a good software engineering practice. In case the symboltable is changed, only the Cone Tree Interface would need to be updated. So in the current version, The Cone Tree Interface passes the

linktable to the Cone Tree Structure to initialize it and to build up the Structure corresponding to the symboltable.

3.2.1 The Tree Graph

The Interface then performs some internal computations and sets up a graphing function to graph the whole tree similar to any basic OpenGL graphics function. This function, when called with some attributes such as size, should display the Cone Tree Structure in 3D using OpenGL. This function is straightforward in its graphing procedure because the nodes stored in the Cone Tree Structure have all been assigned a 3D position in space. After a node is graphed, it is linked by a straight line to its parent using the parent's position. The graphing function also takes care of displaying statistics as colors in the graph. The graphing function has three attributes. The first attribute called the radius is used as a scale for the whole graph. The second attribute, the object size, determines the size of the object, currently a cube, representing a node in the tree. Previous experiments showed that a cube is one of the fastest objects to render versus other 3D objects. And finally, the last attribute, called labels, displays the node id which is the node's index in the array. The node names are not displayed in this function because they use a lot of processing, take too much graph space, and make the graph very slow to interact with. Other ways that display nodes names will be discussed later in this thesis. The Tree graphing function uses color to help users in graph exploration. Using color, the user is able to easily identify and recognize important parts of the tree. The basic color model used is divided in two parts. The first is the node color. This color is given only by the node's depth in the tree. The top node has a green-blue color, the bottom node has a pure blue color, and the middle nodes are given colors according to

their position in the tree. The second is the edges colors. The edge colors which are clearly easier to distinguish than node colors because they occupy more space in the graph are used to display some statistics such as the largest node and its depth. The colors would range between dark red for the largest node in the tree to very light red for the smallest. This color is combined with equal green and blue colors defining the depth of the node. The edges colors are based on the statistics of the parent, so all the edges of one parent are displayed with the same color. This makes the cone of the largest deepest node appear in dark red and that of the biggest node closest to top being close to white.

3.2.2 OpenGL Picking

The same function discussed above also implements OpenGL picking. Picking is a keyword for object selection using an input device such as a mouse. A user picks something when he clicks on it or when the object is the nearest in the area of selection with respect to the user's viewpoint. The Cone Tree Interface has two rendering modes. The first is GL_RENDER where the graph of the Cone Tree Structure is displayed as any static object that can be manipulated through standard modeling transformations. The second is GL_SELECT where the Cone Tree Structure is displayed as it would be for GL_RENDER but with identifier assignments to selectable objects. glLoadName() is called for each object that the user can select. When the Cone Tree Interface is set with RenderMode() to GL_SELECT, the graph is drawn in selection mode as described and this makes it easy for the picking matrix to interpret the exact position of the node the user picked. Usually picking is called only once when the mouse button is clicked, but it is up to the driver program to ensure that the right rendering mode is set. When a node is selected by the picking mechanism implemented in OpenGL, a blue sphere is drawn

around the picked node in the graph and its label is activated to make it easy to be distinguished in the whole tree.

3.2.3 Local Node View

Another main function available in the Cone Tree Interface is the graph of a single node (local). However, this graph is not a graph using the Cone Tree Structure because of multiple inheritance in which very significant information is lost when the strict hierarchy is created. Therefore this node graph needs to be graphed in a different layout than the Cone Tree layout. A simple layout is used supposing that the graphed node is displayed in the middle of the window with two cones one heading upwards and the other downwards. The upwards cone represents the parents of the node, in case of only one, that parent is drawn right above the node being graphed. Similarly, the downwards cone represents the children of the node. The graph of a node also uses the same color model as the tree graph. This function is also used as one whole object that the user can manipulate such as any standard OpenGL object graphing function. The graphselectednode() function uses the graphnode() function which uses the Local View layout described to graph the selected node that was picked using OpenGL picking. Some useful set and get functions are also available to help the user get important information used for graphics and maintenance.

3.2.4 Node Searching

The last important feature the Cone Tree Interface holds is the search function where two kinds of searches can be done. The first is using the requested node id. This search only checks if the requested id exists in the symboltable and, if it does, it sets it as

the selected search node. The second type of search is using the node name. This search tries to find the requested string as a name or as a synonym and sets the node found that holds that name or synonym as the selected search node. The search node function returns an information string that could be used to display the status of the search. For example, if a node is found to contain the requested string, a message would be returned saying: “Node with ID 345 is found to contain string...". If a search result is found, it is assigned to the Cone Tree Interface which changes the tree graph; a red sphere is put to encircle the selected search node. The Cone Tree Interface is considered to be a bridging class between the Cone Tree Structure and the Grapher which will be discussed in the following section.

4 The Grapher

To display the graph and manipulate it interactively, a component that takes care of graphics and user input, called the Grapher, was implemented. It has six main parts: a main view window called Taxonomy Visualization, a controls window containing some interaction capabilities to the main view, a tool called the Compass, the Local View window, the Information window along with the Console, and, finally, the driver program.

When explorers are wandering in some region, they basically need three things to carry with them; a compass, a map, and some description about special places they may encounter in that region. This metaphor is used here in our project. The main view and its controls represent the real world or region with the explorer being able to move to where he wishes. The Compass combines two tools for the explorer and these make him aware of where he currently is located and how the whole region looks like. The Information and Local View windows give the ability to the wanderer to get relevant information and a local detailed view about points in the space that he is interested in.

4.1 The Main View

The main view window is the main visualization window that the user interacts with using the main view controls. This window is rendered using OpenGL and is capable of visualizing more than 100,000 nodes with their edges at a rate of 5fps on a 1.4 Centrino processor with 512MB RAM and an NVidia GeForceFX 5200GO running WindowsXP®. An example can be seen in Figure 13. With OpenGL, it basically relies

on the graphics hardware and how many million texels it can render per second. Texels is a term for texture elements. The main view works best in full screen mode were OpenGL has control over the whole screen; there are no graphical tasks to be given to other processes.

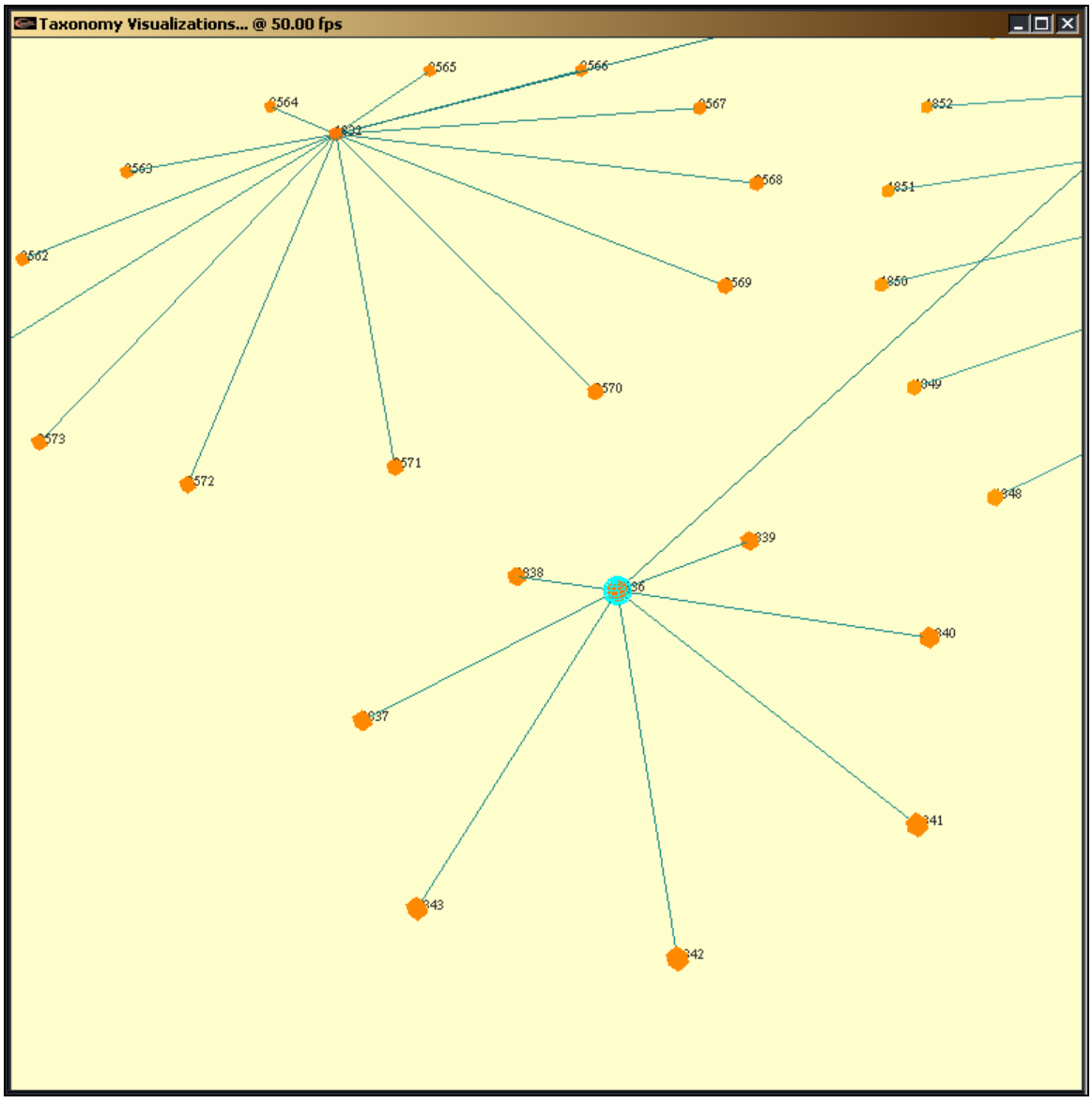

Figure 13 : Main View of "galen.tree"

The main view sets up the viewing volume and the transformations needed for the user to interact with the graph. It has an implemented camera structure prototype that takes care of camera style movements used from CUGL, Concordia University Graphics Library

[35]. Although, this camera style movement code is still a prototype and under test, it is still useful for our project. The main view also has special capabilities when nodes are selected using a search or using OpenGL picking. The main view directly centers the selected node so that the user would know what he selected and this would be a shortcut to easier navigation to that node. With some appropriate transformations, the main view uses a single call to the Cone Tree Interface class, specifically to the graphConeTree() method with the appropriate attributes it needs. This allows us to give users the right view they are requesting or expecting from the interface. The graphConeTree() function as described above is a linear function that processes each node once and, using the node's statically calculated position in space, renders it along with its edge that is connecting it to its parent. The selected and searched nodes are distinguishable in the whole tree since they appear by specially colored spheres rather than cubes along with their visible automatically activated labels. In Figure 13, the main view shows a focus on selected node #1836 of "galen.tree" which yielded in a total number of 2,752 nodes graphed at a rate of 50fps. The tree graph could have an overall cone view with the top node being on the top of the cone and all sub cones forming the rest of the cone. The nodes are displayed as cubes and the edges are direct lines connecting the node to its graph parent. The graph parent is surely one of the parents of the node. The right parent for a node is selected when the node sequence is created (see Cone Tree Structure). Finally, this function gives a dynamic fps rate to keep the user informed about the current rendering complexity of the graph. This rate is recalculated at every single display procedure that OpenGL performs and is shown in the title bar of the main view window.

4.2 The Main View GLUI Controls

GLUI, Graphics Library User Interface, is totally compatible with GLUT, Graphics Library Utility Toolkit. As explained in [8 and 9], it is platform independent and works in conjunction with GLUT. The main view GLUI controls window is totally built with GLUI. It implements user interface that only affect the main view window explained in the previous section. This window has four main interaction sections: the graphics controls section, the switches section, the search section and the application controls section. The graphics controls section presented in Figure 14 has user interaction controls to allow the user to interact with the tree graph in a logical way.

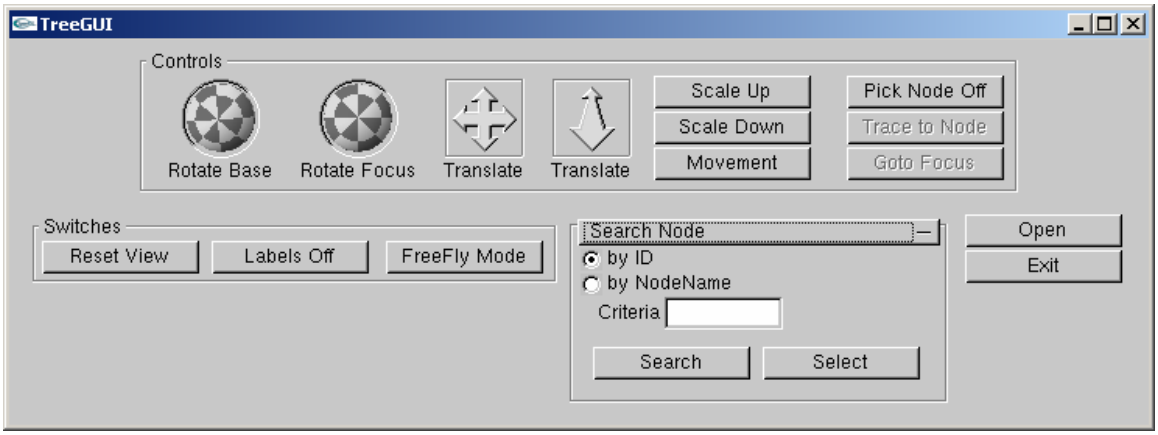

Figure 14: The GLUI Controls Window

These controls include rotations and translations of the whole graph in all the respective axes. Rotate Focus, however, does not work unless a focus or node is selected using picking. Buttons to scale up or down are also available to the user if he would like to have a fast general view of the structure of the tree. Movement here is a control that affects all graphics controls in their movement interval. When Movement is pressed, the movement interval is multiplied by 10. This could be useful in very large graphs. Pick Node is a control that exploits OpenGL picking to pick a labeled node using the mouse click. When a node is effectively selected, the main view window translates this node to

the middle of its screen. Moreover, the Rotate Focus, Trace to Node and Goto Focus buttons are enabled. The picking matrix selects all nodes that appear within a small area under the cursor of the mouse click and stores all hits in a special array. The node that our system selects would be the nearest node to the viewer in terms of depth at that area under the mouse cursor. For the time being the Goto Focus and Trace to Node buttons are not fully functional. The Goto Focus button uses the camera class in CUGL to smoothly animate the translation sequence from the point of start to the point of finish when a node is picked. The Trace to Node button should distinctively color edges that form a path to the selected node from the top node.

The switches section holds controls for general windows options. These include: the reset view button, the labels button and the Free Fly Mode button. The reset button is supposed to reset and center the view in the main window wherever the user is and whatever he is trying to do. This is a very useful button for lost users. This button basically reset all graphics parameters to their initial values. If a node is picked, this node will be centered and viewed from the top when the reset button is clicked. If no node is selected, the main view window will show the initial view where the eye is somewhere on the positive y-axis and pointing to the origin of the graph. The (0, 0, 0) position or origin is midway between the top and the bottom nodes. The Labels button switches the labels in the main window on and off. This is a nice feature for small graphs to directly know the nodes and what they are connected to, but is not advised in large graphs because of its huge impact on frame rate performance. For this reason, the Labels are switched to off initially. The label of a node that appears next to its cube representation in the main view window is basically its unique identifier. The last switch is meant for users

who would like to manually explore the tree without having to use the GLUI controls window. When Free Fly Mode is enabled, the main view window uses the mouse and the arrow keys to perform user requested actions. The mouse rotates the graph and the arrow keys use the camera class to move in the space. More keyboard controls are available and are explained in the Console window that accompanies the application. Basically everything the GLUI controls can perform is tunneled down through a keyboard function with its assigned keyboard switch to perform the action. So, for example rather than clicking on the pick node button, users can just tap the 'p' key on the keyboard and this would do the same thing. This is done to give the user an advanced and probably faster way to navigate through the structure. This is also very useful in full screen mode where interaction is much smoother due to OpenGL being in control of the whole screen.

The search node section contains a radio group to select the method of search, a text box input field to allow users to enter information relevant to their choice in the radio group, a search button, and, finally, a select button. There are two search mechanisms currently available: 'by id' or 'by name'. By id basically means that users are expected to enter a number representing the index of a node. Using the input, the system will search and get the node having this number as its identifier in the list. The id of a number is basically its index in the array. 'By name' means that the user is expected to enter a string that identifies a node using its name or one of its name's synonyms. When the search button is clicked, the system searches all strings it holds for the requested set of characters. If a node is found, the searched node is assigned to the Cone Tree Interface which in its turn renders the tree using this extra attribute. The select button would then

be enabled to allow users to assign the searched node as a selection to the system. Users would then be able to perform similar actions as if they selected the node using picking.

The application controls section holds controls to perform general application actions such as opening a new file and quitting the application. The Exit button simply cleanly shuts down the application and erases all used memory. The Open button has the ability of opening another .tree file, parsing it, and graphing it. However, this control is currently disabled.

4.3 The Compass

The Compass is a window that keeps a general view of the tree and the position of the user so that he would always know where he currently is and what he is looking at. As explained before, an exploring user needs this kind of tool to be able to easily navigate in a huge space without getting lost. The Compass gives a general view of the whole layout of the graph tree along with an object representing the eye position of the viewer (user) and where it is heading (focusing at).

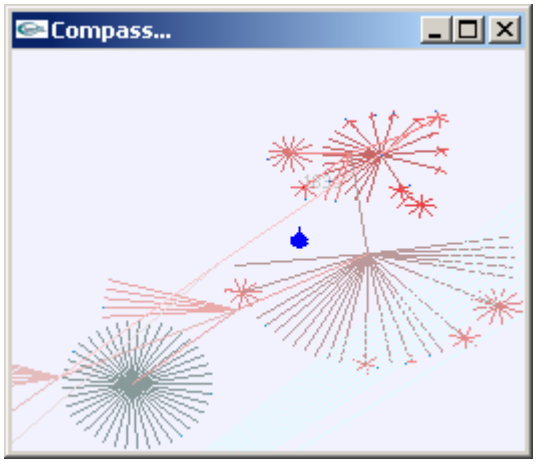

Figure 15: The Compass Window (local)

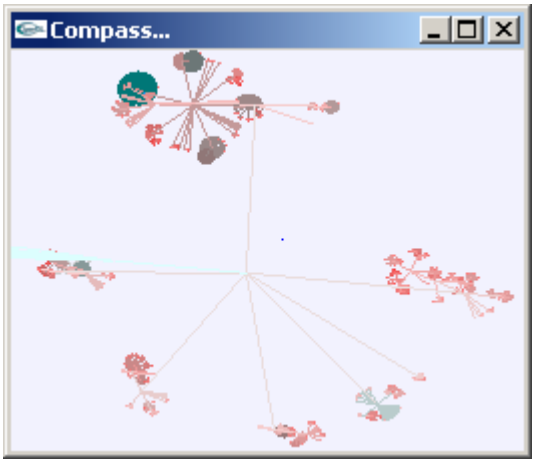

Figure 16: The Compass Window (general)

Figure 15 shows the focus and the eye position along with the surroundings. Figure 16 shows a general view of the same tree with the eye position marked as the blue dot. The Compass window has some independent controls which allow users to customize the view of the window using the arrow keys and the mouse. The mouse rotates the Compass display. The arrow keys translate it in the global x and y planes. The page up and page down buttons scale the display in the Compass window to give users more control on the details they would like to see. The Compass is a very useful tool to lost users since by just looking at it, they would know where they are in the space and what they are looking at and with some easy Main View manipulations, the users would adjust the view to point to what they need to see.

4.4 The Local View

Along with the Compass window, users need one more important tool that provides a detailed description of what they are looking for. The Local View uses the information captured at compilation time of the taxonomy to provide some important but omitted details while drawing the Cone Tree. The Local View is basically made of three parts: the parents, the focused node, and the children. The backbone structure of this view is basically two cones where one is inverted on the top of the other corresponding to the same reasoning that a node can have one or many parents and a node may have one or many children as shown in Figure 17. We have mentioned before that a node may have no children; in this case, we link the node to the logical “bottom” node and it is considered to have one child. If a node has one parent, then this parent is drawn in the center of the base of the inverted top cone. If the node has more than one parent, these are

drawn on the circumference of the base of the inverted top cone. The same rules apply to the bottom cone. The resulting layout would be logically easy to understand and, hence, users would be comfortable in discovering relationships in the graphed tree.

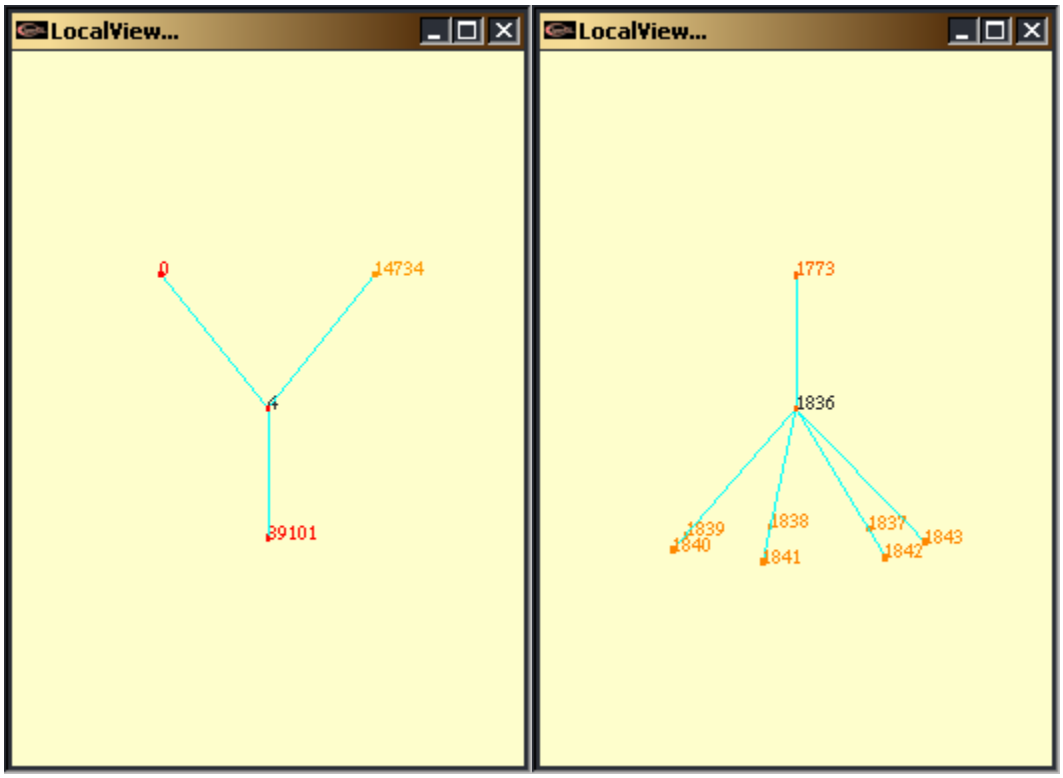

Figure 17: The Local View Window

This view uses the colors used in the main view corresponding to each node to clearly identify relationships between the nodes. Therefore, if a node has two parents, those parents will have the same colors assigned to them in the main view window. This presents important information, for example, a node that has a dark and light parent means that this node has a path in both the top part and bottom part of the tree. Therefore, processing would lead to this node in two different ways. Similarly, the children nodes are graphed on the base of the bottom cone using the same properties. A node can have children which are not visible in the tree graph. As mentioned above, no node can appear twice and, in case of multiple inheritance, a node would exist under two parents. The

main view shows this node under the parent that is assigned to it using the sequence list described in Section 3.1.1 of this thesis. On the other hand, when looking at the Local View of that node, both parents will be apparent each with its respective properties with colors indicating their positions in the tree. And when the Local View of the parent is viewed, this parent will hold all the children including the one that graphically belongs to another parent up in the tree. Graphical information is not considered to be enough for text based datasets. Name strings and statistical data are considered important. This is why an information window and a console are explained in the next section to provide further useful information to users.

4.5 The Information window and the Console

The Information window is a graphically generated text window that displays text using OpenGL and GLUT. The first line in the Information window provides information about the selected node. The last line in the window is considered to be a status line that informs the user about the action that was currently taken and some properties.

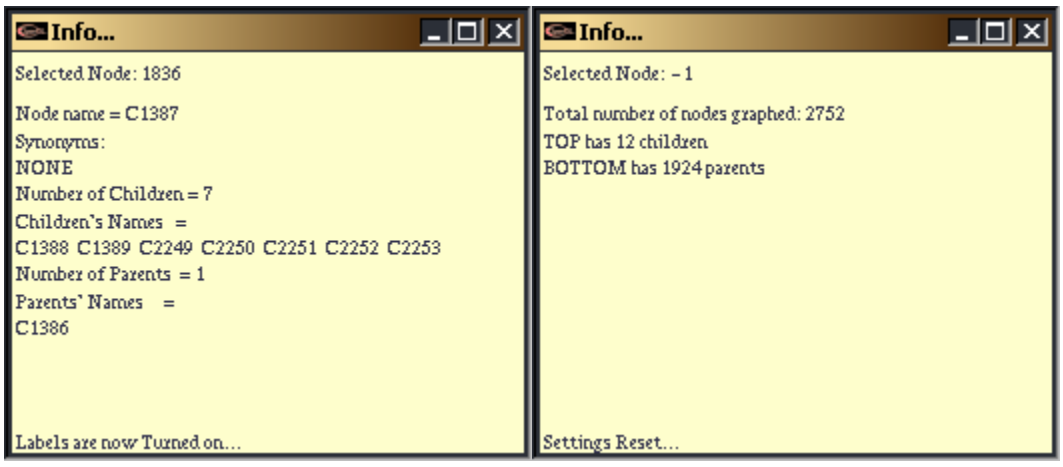

Figure 18: The Information Window

Two screenshots of the Information window display in different situations are shown in Figure 18. When no node is selected, the Information window displays the status of the tree, including the total number of nodes, the top node's children number, and the bottom node's parents number. When a node is selected, information about that node is displayed. The selected node ID on the first line changes to display the corresponding node ID of the selected node. If no node is selected, -1 is displayed to say that no node is selected. The Information window then continues to present users with information about that node.

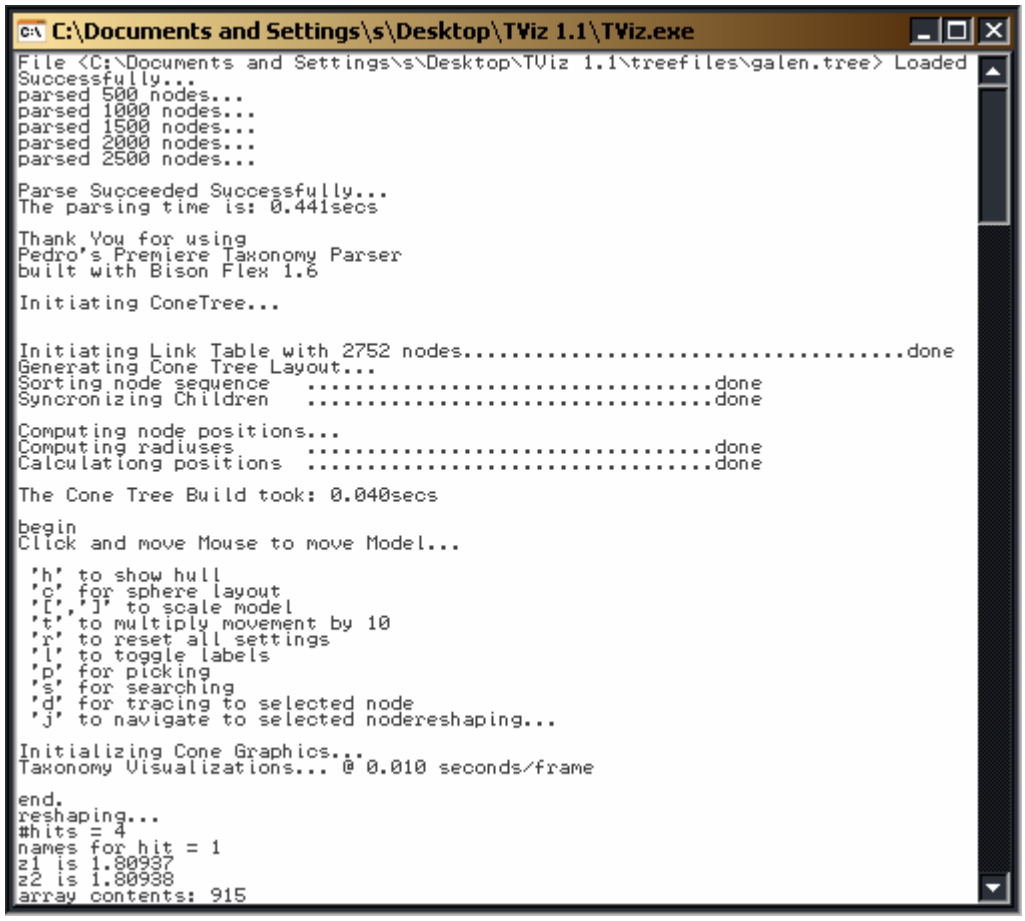

Figure 19: The Console Window

The node's string name is displayed then any known synonyms are written below. The number of parents that the selected node has and a list of the string names of those

parents are then displayed underneath. Similarly, the children number is displayed along with a string list of those children. The Information window is still a basic window that uses OpenGL capabilities to display text. This is why a more elaborate information tool that follows the progress of the application, its preprocessing, and its status is used. The Console is the command line tool that the application uses to run. The Console is an operating system tool and is shown in Figure 19. The Console window, even before graphing and using OpenGL is able to provide the user with some information about the current state and processing and about the preprocessing status. The Console has been implemented with progress fields that inform the user about each 3% of the application's sub-processes. The Console also provides information about current actions and keyboard controls that users can use to interact with the graphics without having to use the GLUI window provided.

4.6 The Driver Program

The application uses a driver program that first takes the path and name of a file from the arguments given to the program and then assigns it to the constructed parser. The parser, when called, processes this file according to the rules assigned and performs actions on found patterns. The parser creates a global symboltable that would be used throughout the application. The driver program then passes a reference of the symboltable to the Cone Tree Interface which in its turn creates the Cone Tree Structure and binds the nodes in the symboltable with the corresponding nodes in the Cone Tree Structure. Creating the Cone Tree Structure takes three steps: assigning the node sequence to be used to graph the layout, finding the radius of each node, and computing

the position of each node in 3D space. All these are done using the Carrière and Kazman algorithm as explained thoroughly in Section 3.1 above. When the Interface has finished all processing and computations, the driver program creates the graphics window along with its control tools and information tools and the graphics is ready to interact with. If any problem while executing these processes is found, may it be logical or an error in processing, the driver program specifies the error encountered and exits by returning control to the operating system.

5 Stress Generator

The Stress Generator is an application that was constructed specifically for this project. The goal of this application is to provide synthetic datasets to give to TViz and test its capability to process very large datasets with random distribution of nodes in the logical tree. The Stress Generator has three input variables. The first is the token name that the user wants the node to start with; this can be one or many characters. All nodes created will start with this token followed by an automatically generated number that makes the node unique. Second is the branching factor which is a number that designates the maximum allowed number of children for a node. And third is the list size which when multiplied by the branching factor, gives the effective size of the dataset that will be created. The application starts by creating two static arrays; one Boolean and the other Integer. The Boolean array keeps track of nodes which have reached their branching limit and those not. The Integer array keeps track of how many children each defined node has been assigned. Since defining nodes with at least one parent is sufficient to render a dataset acceptable and conformant to our regular grammar, then only that property is used to make our program simple. Each node is assigned a single parent from the previously defined nodes. Choosing from the previously defined nodes preserves the correct overall logical connection of the entity created by eliminating any loops or cycles that may occur. The Integer array is updated accordingly by incrementing the chosen parent's children number by one. Upon each iteration cycle, the number of children that would be assigned for each parent is randomized and, if that number is reached, the Boolean array is updated and another parent is chosen. If the randomized number is not reached, the node is assigned a parent from the previously defined parents and the Integer array is

updated. When all nodes in the defined interval are processed, the Boolean array will have false values for nodes which have no children assigned. Therefore, we simply assign them as parents of the bottom node which is written as the last statement in the dataset. Using this technique, the Stress Generator is able to create very large datasets in a matter of milliseconds. The synthetic datasets were at first manually inspected for their underlying logic and were found to be totally compatible with our taxonomy datasets. However, one clear issue resulted in a delay in the processing time when the bottom node in large files is reached by TViz for processing. Because the number of connections that link two nodes together created by the synthetic generator is equivalent to the total number of nodes in the entity and that defined nodes are given more chance than those that are not defined yet, the Boolean array will hold a certain number of false values with respect to the whole entity. In the worst case, we would expect a massive n-1 parents to TOP if the branching factor is large enough. Therefore, the bottom node results in a very big string that has to be parsed by TViz. This huge string takes a bit of time using our Bison Flex parser. But eventually, the result will always be correct. In real taxonomy datasets, such huge bottom nodes are not common which make the parsing of the file reasonably faster. The user could clearly distinguish a stall period at the end of a large synthetic dataset before the parsing is announced as completed. This is not often the case with real sets. We can deduce from this fact that Bison Flex parsers work best with small strings. The Stress Generator is a very useful test tool that would test our research application and its limits. A test case has been formulated using the Stress Generator in the following section.

6 Test Case

A test program has been created to give test results and performance outputs for a set of inputs. This test program is a basic C++ program that creates datasets using our Stress Generator and gives the created dataset to TViz. The test program was run on a medium-end computer having the properties shown in Table 4.

System Specs:

Computer Name:	MEGAPULSE01
Operating System:	Microsoft Windows XP Professional (5.1, Build 2600)
Language:	English (Regional Setting: English)
System Manufacturer:	TOSHIBA
System Model:	Satellite M30
BIOS:	@Version 1.0
Processor:	Intel(R) Pentium(R) M processor 1400MHz
Memory:	510MB RAM
Page file:	234MB used, 1013MB available
DirectX Version:	DirectX 9.0b (4.09.0000.0904)

3D Device

Name:	NVIDIA GeForce FX Go5200
Manufacturer:	NVIDIA
Chip Type:	GeForce FX Go5200
DAC Type:	Integrated RAMDAC
Approx. Total Memory:	32.0 MB
Current Display Mode:	1280 x 800 (32 bit) (60Hz)
Monitor:	Default Monitor

Table 4: Test Node Specifications

While processing, TViz sent its output to a file rather than to the console and this collection of output files were then manually processed to yield in performance Table 5. The table shows that TViz is a reasonably fast tool for generating graphs from the synthetic data created considering the huge datasets given to it. The large strings

encountered while parsing the bottom node definition as explained in the Stress Generator section yields in some rundown in the performance of the Bison Flex generated Parser. Real datasets are surely to be even faster than the results declared in Table 5.

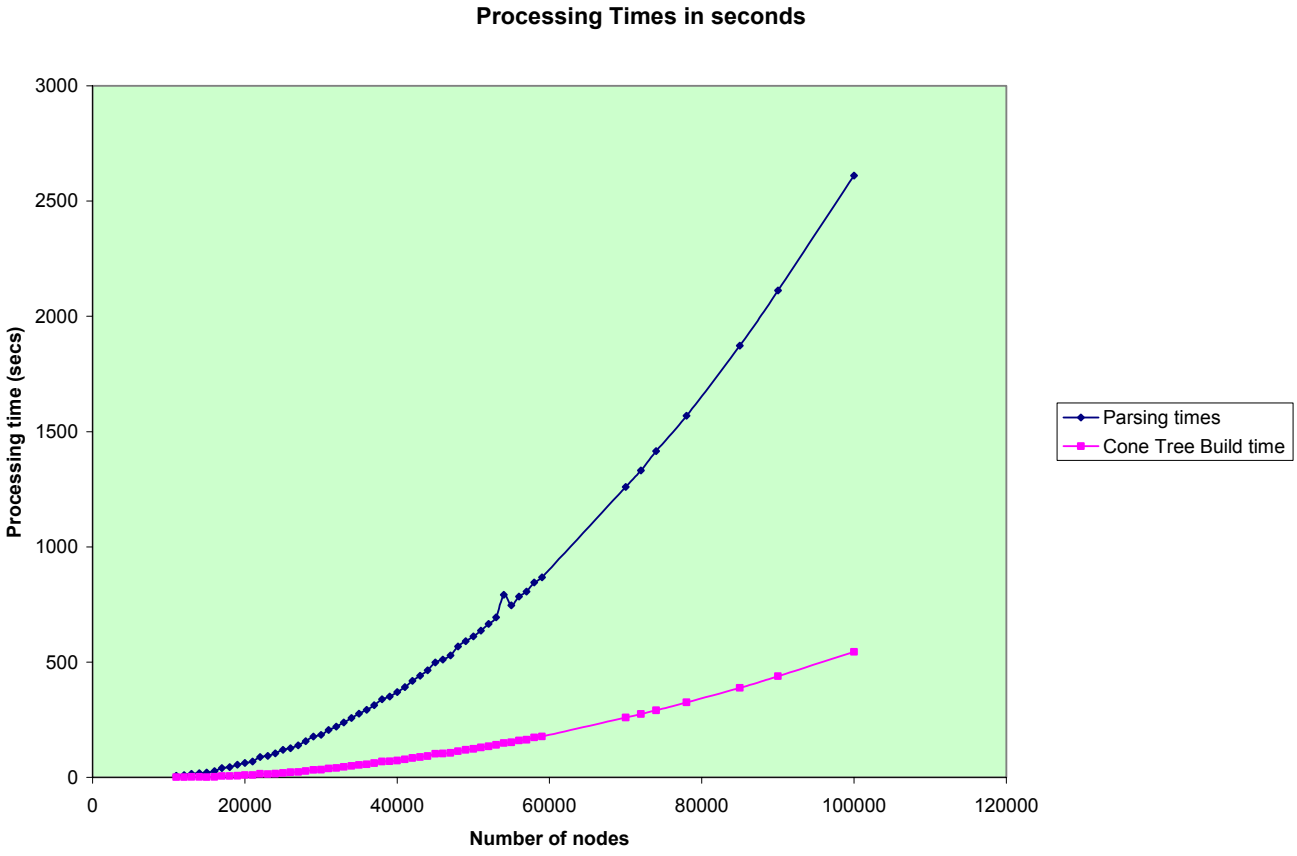

Table 5: Performance Table

An application is said to be usable if it satisfies the display frame rate performance standards for interactivity. The minimum acceptable frame rate per second or fps for a smooth interaction is 12fps. Table 6 shows the frame rate performance on our test hardware. It is clear that our algorithms run much faster than the parser and this is relative to the structure of the input files as described earlier, the carefully optimized processes used in our algorithms, and, finally, to fast memory access in contrast to hard storage devices such as hard disks that are used by the parser.

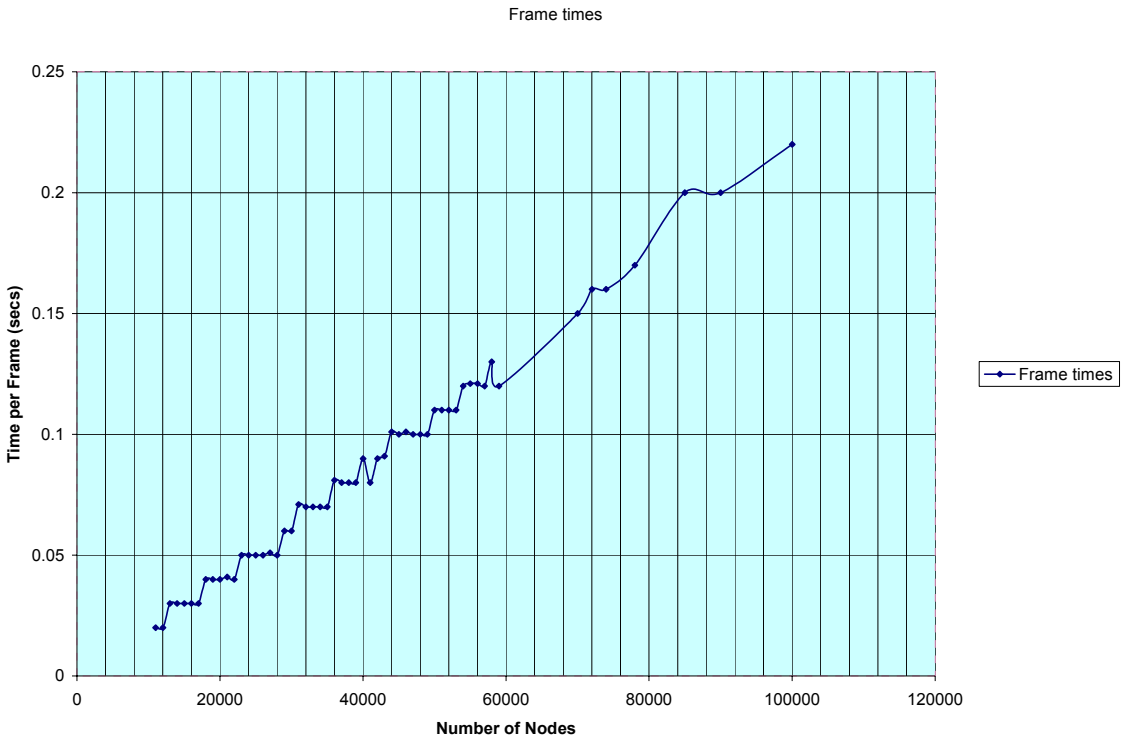

Table 6: Frame Rates

Table 6 shows that our application is usable with very good performance with up to 40,000 nodes datasets. 100,000 nodes give poor performance but the application is still considered usable since it responds at a 5fps rate. One can immediately notice the stepping in this table and this is due to the graphics hardware that has a static streaming capability. If the hardware is not able to stream all the required information in respective frame time, it automatically jumps to a lower frame rate and this next frame rate can handle a couple more tens of thousands of polygons. The frame rate is strictly dependent on the hardware. With better graphics hardware, a better frame rate will surely occur and hence better user interaction.

7 User Output

This project was distributed to a set of users to be tested and some feedback was returned. The main concerns of users were basically expected because of the domain of application of this system. Many users did not understand what the application serves for and why it should be used. The application is intended to be used by people who design and construct ontologies; in other words, for specialists in the field. People who do not know about Semantic Web and ontology design would have hard times guessing what the application would be used for. This document could be used as an introduction to what this application does, why it is used for and how it works.

Some interface issues were suggested as well, for example, the use of multiple windows versus the whole application being integrated into a single window. The way it is currently done was explained to be essential because advanced users would like to have a full screen view of the tree rather than being docked in a window. Moreover, this allows the user to have separate control over each window. We received good feedback about the GLUI Controls window as well as the presentation and the professional look of the application as a whole. The GLUI Controls window is a straight-forward interface that allows users to directly understand what each control does and is used for. The colors and the layout of the different windows of the application make it visually configurable and appealing suiting both professional and amateur users.

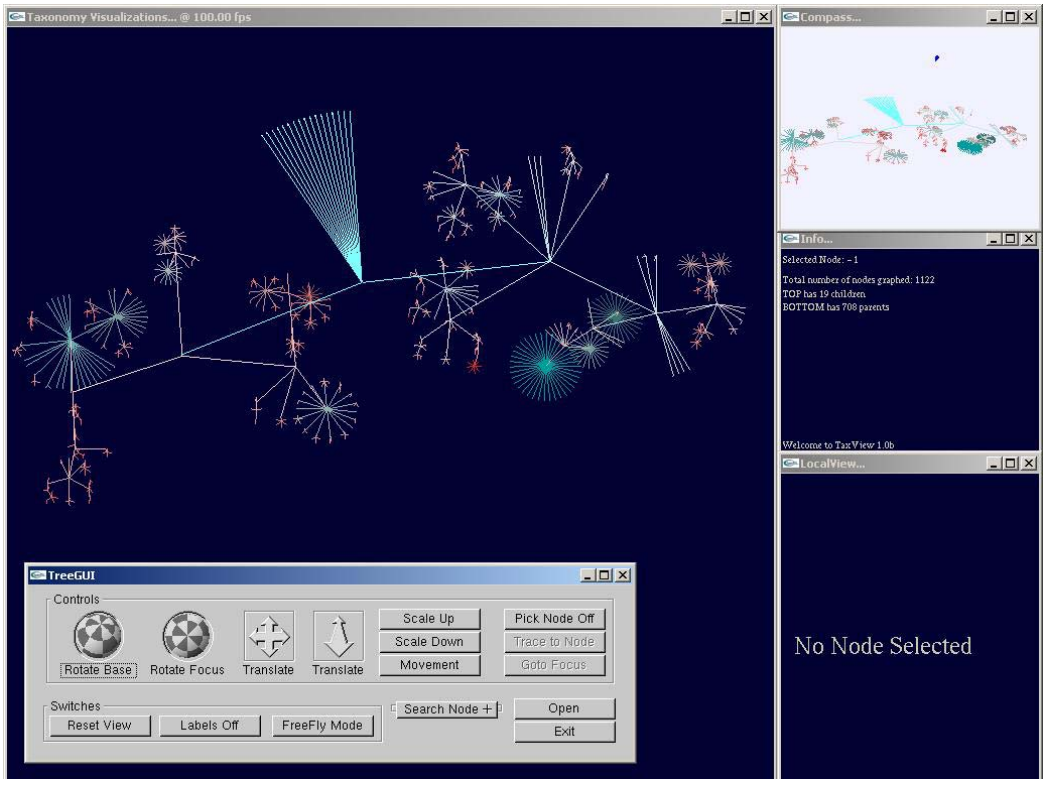

Figure 20: TViz: A Taxonomy Visualization Tool

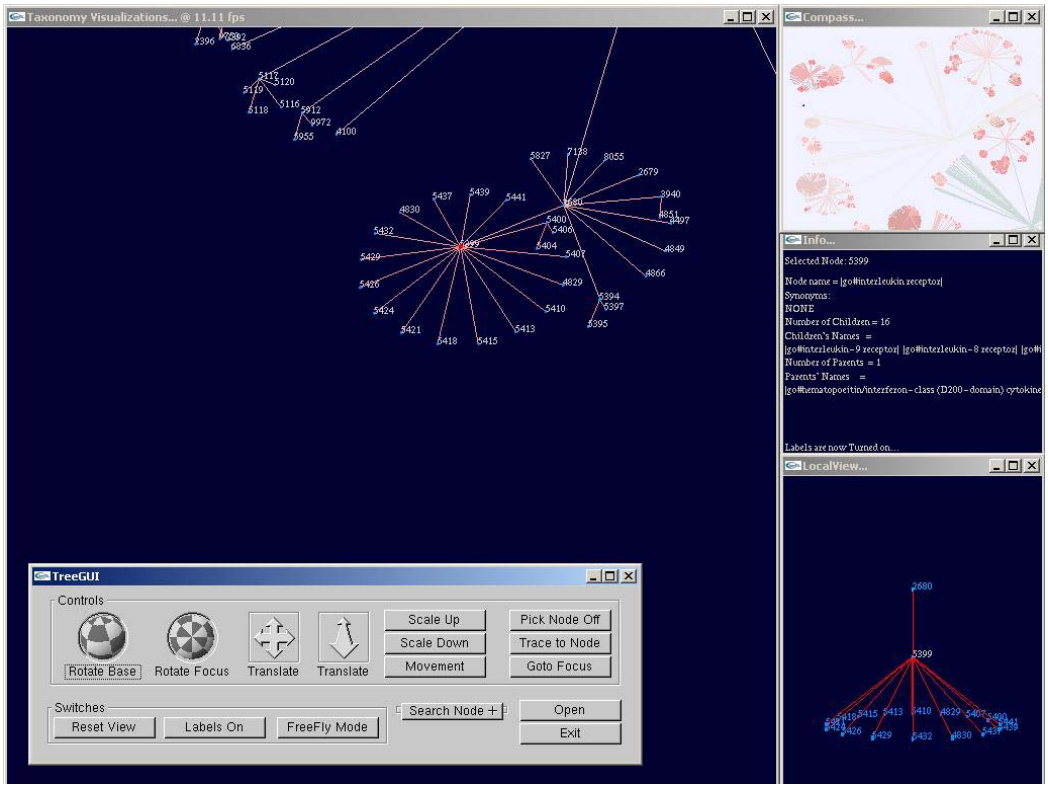

Figure 21: TViz: A Taxonomy Visualization Tool 2

8 Future Work

We categorized the improvements into three categories: the research, the layout, and the interface.

Concerning research, Cone Trees have proved to be effective for the hierarchical visualization of large datasets. Regarding the Carrière and Kazman algorithm and the Cone Tree algorithm in general, there was no reference in any of the papers read about how to graphically limit the tree in 3D space and not letting it grow as the datasets grow. In our study, the whole tree currently is static in its depth. The distance from the top node to the bottom node is set to '2' for all trees. All nodes within the tree have a depth position between '-1' and '1' where the center of the tree is considered to be at '0'. The initial view of the tree when the application is started is in the direction of '0' and having '-1' and '1' viewable in the view window. However, making the tree statically bound graphically is a more difficult problem. The tree computes the radius of a given node using the number and the size of the radii of each of the nodes that are children of this node. Thus, if we want to make the tree graphically bound by width, we have to carefully choose the initial radius set to the 'BOTTOM' node. Some experiments were discussed at the end of Section 3.1.2.2; however, it remains an open problem to find a suitable value depending on the layout of the tree being processed. This initial radius should also keep a balance between the height and the width of the tree without visually distorting it.

For very large instances in the layout, the same Cone Tree layout would be used but with a hemisphere rather than a circle base at the bottom of the cone representing each node. This would surely handle more nodes in a smaller space. Also, this would be especially useful if a lot of those children nodes had one or no children. The layout of the

tree as explained in the Cone Tree Structure section is built in a top-down manner by graphing the children of the top first and then their children and so on. This layout can be easily changed to a bottom-up approach that gives a different view of the same tree. This would be interesting in the case that sensitive information and statistics can be found in one of those layouts. So users would appreciate it if they could see the same tree in a different view or perspective. Also, the Tulip improvement to the Carrière and Kazman algorithm explained by Auber in [16] could be used to find the smallest enclosing circle and would provide a better compactness of the tree. Other kinds of layouts which are not cone tree layouts can be implemented such as the hierarchical hyperbolic tree layout which was researched and implemented in H3 by Munzner [11].

The interface of TViz uses OpenGL to perform graphics. OpenGL is a fairly low level language. The use of advanced graphics techniques such as the ones used in gaming and specifically modeling would be very efficient in visualizing large numbers of polygons. As mentioned, our layout represents a node by a cube and its link to its parent by a straight line. Some advanced graphics structures such as Binary Space Partitioning trees, or BSP trees, which minimize the number of elements to be drawn by not drawing any non-visible elements (not passing it into the graphics pipeline) would certainly give better performance in very large trees when the user is focusing on a specific part of the tree. Applying a texture to each cube object that shows its name or label would also lead to a good performance upgrade since, in our current implementation, each character of the label is rendered alone. Modern graphics hardware can apply textures faster than they can render the object being graphed. Another performance improvement when users are viewing the tree at large would be to approximate the look rather than drawing all the

elements of the tree which are most probably not visible each by themselves. Another method for testing if a node lies in the viewable area is to set a bounding box, or a frustum culling technique, configured by the graphics window and by the tree settings. This bounding box would set bounding plane values and would omit to draw any node and its children that lie outside the bounds of the viewing volume. This works similarly to the clipping planes to the viewing volume that are used in OpenGL's graphics pipeline. The only difference is rather than giving all the polygons to OpenGL and then letting it manage the view, we are reducing the number of polygons sent into the pipeline and this gives more performance to the application when viewing.

The Compass is a very useful tool as explained above. Since it provides an overview of the whole structure, users should be allowed to directly change their position by just specifying where they want to be on the Compass map. With a single mouse click the user can be placed at the point or within the proximity of the point they wish to be. This could also be done by smooth transition so that users would not get lost by the rapidly changing perspective and gives them a better understanding of the structure of the tree.

The Information window needs to be more dynamic and more text based. Currently, the window is rendered with OpenGL by using an orthogonal projection and text prints on raster positions on that window. That should be changed to a console-similar window that handles text or a sheet window that graphically shows information such as 100 children names for a selected node.

The interaction and malleability in the Cone Tree graph function is not optimized and could handle a lot more than it does now. Rotations of parts of the tree are

implemented now as rotations for the whole tree at the focused point. This makes the focused point fixed while the rest of the tree is rotating around it. A better implementation would be to include interaction capabilities in the body of the graph function and this would allow a rotation independent of the whole tree, in other words, only a sub tree would be rotating when a rotation of the focus is executed. This would be fully compatible with the Carrière and Kazman algorithm, since the rotation would not affect other subtrees. Other capabilities could also be included in the function body such as a choice of color to customize the appearance and a set of user operations imbedded in this function that allow users to have specific control at the time of drawing a specific node or edge.

9 Conclusion

The fruit of our research, TViz, is considered to be very useful and the tools implemented allow users to browse any taxonomy. Specifically, Cone Trees are a good way to represent hierarchical data since they provide a logical layout that embeds the hierarchy. Although faster applications have been implemented for other types of datasets, the user interface tools such as the compass used in this research are neither implemented nor discussed in those applications as to date. The use of taxonomy datasets, Compiler Design, Cone Trees, and OpenGL made TViz a unique application that would be used on different platforms to explore reasoning and interpretations of ontologies. TViz is expected to be used by Industry, corporate and individuals who would make use of it to explore their ontologies and TBoxes. Semantic Web is becoming more and more popular and has already taken steady places in important industry positions such as mechanical part searching, document searching and so on. TViz is capable of providing a visual information catalogue to browse the processing of the RACER semantic engine which is both efficient and fast. TViz does not need any high end machine and currently provides versions for Microsoft Windows®, Unix/X-windows, Solaris and Mac OS. A high end machine would give even better results and faster output in terms of both preprocessing time and display rates for large structures.

10 References

[1] A. A. Aaby, "Compiler Construction using Flex and Bison", Walla Walla College, Version of September 15, 2003.

[2] V. Paxson, "Flex: The Fast Scanner Generator Manual Version 2.5", *the Regents of the University of California*, 1990.

[3] S. Decker, M. Sintek, "The Semantic Web Portal", *www.semanticweb.org*, internet reference, 2003.

[4] W3 organisation, "Semantic Web", *www.w3.org/2001/sw*, W3 Consortium, internet reference, 2004.

[5] P. Mutton, J. Golbeck, "Visualization of Semantic Metadata and Ontologies", in *Proceedings of the 7th International Conference on Information Visualization*, IEEE, No. 1093-9547, 2003.

[6] A. Telea, F. Frasincar, G. J. Houben, "Visualization of RDF(S)-based Information", in *Proceedings of the 7th International Conference on Information Visualization*, No. 1093-9547, 2003.

[7] A. Zarrad, "The TBoxEditor", *Master Thesis, Concordia University*, 2004.

[8] D. Shreiner, M. Woo, J. Neider, T. Davis, "OpenGL Programming Guide", Fourth Edition, *OpenGL Architecture Review Board*, Silicon Graphics, 2004.

[9] P. Rademacher, "GLUI: A GLUT-Based User Interface Library", Manual, Version 2.0, June 1999.

[10] T. Munzner, "Interactive Visualization of Large Graphs and Networks", *PhD dissertation and slides, Stanford University*, June 2000.

[11] T. Munzner, "H3: 3D Hyperbolic", in *Proceedings of IEEE Symposium on Information Visualization*, 1997.

[12] B. Johnson, B. Shneiderman, "Tree-Maps: A Space-Filling Approach to the Visualization of Hierarchical Information Structures", in *Proceedings of IEEE Visualization,* pp. 284-291, 1991.

[13] O. Kersting, J. Dollner, "Interactive 3D Visualization of Vector Data in GIS", in *Proceedings of ACM GIS '02*, pp. 107-112, 2002.

[14] G. Robertson, S.Card, J. Mackinal, "Cone Trees: Animated 3D Visualizations of Hierarchical Information", in *Proceedings of CHI '91*, pp. 189-194, 1991.

[15] J. Carrière and R. Kazman, "Interacting with Huge Hierarchies: Beyond Cone Trees", *IEEE Symposium on Information Visualization*, pp. 74-81, 1995.

[16] D. Auber, "Tulip – A Huge Graphs Visualization Framework", *PhD thesis, University Bordeaux I*, France, 2002.

[17] S.H. Hong, "COMP4408 Information Visualization", *course slides, University of Sydney*, Australia, 2004.

[18] H. Koike, H. Yoshihara, "Fractal Approaches for Visualising Huge Hierarchies", in *Proceedings of 1993 IEEE/CS Symposium on Visual Languages*, pp. 55-60, 1993.

[19] J. Lamping, R. Rao, and P. Pirolli, "A Focus+Context Technique Based on Hyperbolic Geometry for Visualizing Large Hierarchies", in *Proceedings of ACM SIGCHI '95*, pp. 401-408, 1995.

[20] K. Lau, R. A. Rensink, and T. Munzner, "Perceptual Invariance of Nonlinear Focus+Context Transformations", in *Proceedings of the First Symposium on Applied Perception in Graphics and Visualization (APGV 2004)*, pp. 65-72, 2004.
[21] G. W. Furnas, "Generalized Fisheye Views", in *Proceedings SIGCHI '86*, ACM, pp. 16-23, 1986.
[22] M. Sarkar, and M. H. Brown, "Graphical fisheye views", *Communications of the ACM*, Vol.37 No. 12, pp. 73-84, 1994.
[23] C. Ahlberg, C. Williamson, and B. Shneiderman, "Dynamic Queries for Information Exploration: An Implementation and Evaluation", in *Proceedings of CHI '92*, pp. 619–626, 1992.
[24] B. Salomon, M. Garber, M. C. Lin, D. Manocha, "Interactive Navigation in Complex Environments Using Path Planning", *Communications of the ACM*, No. 1-58113-645-5, pp. 41-50, 2003.
[25] S. Benford, I. Taylor, D. Brailsford, B. Koleva, M. Craven, M. Fraser, G. Reynard , C. Greenhalgh, "Three Dimensional Visualization of the World Wide Web", *ACM Computing Surveys,* Vol. 31, No. 4es, 1999.
[26] H-Y. Lee, H-L. Ong, E-W. Toh, and S-K. Chan, "A Multi-Dimensional Data Visualization Tool for Knowledge Discovery in Databases", in *Proceedings of IEEE Conf. on Visualization*, pp. 26–31, No. 0730-3157, 1996.
[27] R.P. Klump, J.D. Weber, "Real-Time Data Retrieval and New Visualization Techniques for the Energy Industry", in *Proceedings of the 35th Hawaii International Conference on System Sciences*, IEEE, 2002.
[28] Q. V. Nguyen, M. L. Huang, "A Space-Optimized Tree Visualization", in *Proceedings of the IEEE Symposium on Information Visualization*, pp. 85-92, 2002.
[29] P. Mutton, P. Rodgers, "Spring Embedder Preprocessing for WWW Visualization", in *Proceedings of the 6th International Conference on Information Visualization*, No. 1093-9547, *2002.*
[30] E. Weippl, "Visualizing Content Based Relations in Texts", *IEEE*, No. 0-7695-0969-X, 2001.
[31] T. Bladh, D. A. Carr, J. Scholl, "Extending Tree-Maps to Three Dimensions: A Comparative Study", *in Proceedings of the 6th Asia-Pacific Conference on Human-Computer Interaction (APCHI'04)*, 2004.
[32] Michael Daconta, Leo Obrst, Kevin Smith, The Semantic Web, Wiley Publishing, pp. 166-167, 2003.
[33] B. Shneiderman, "Treemaps for space-constrained visualization of hierarchies," personal paper, December 26, 1998, updated May 18th, 2004.
[34] Atta-ul-Manan Khalid, "OilViz3D", University of Manchester, akhalid at cs dot man dot ac dot uk, personal communication, supervised by Dr. Ulrike Sattler, November 2004.
[35] P. Grogono, Concordia University Graphics Library resources, Concordia University, Department of Computer Science, release of May 10, 2004.