Computer Science

Carlos José Díaz Baños
Erik Andreasson

Information Visualization

Bachelor's Project

C2012:09

Information Visualization

Carlos José Díaz Baños
Erik Andreasson

This report is submitted in partial fulfillment of the requirements for the Bachelor's degree in Computer Science. All material in this report which is not our own work has been identified and no material is included for which a degree has previously been conferred.

Carlos José Díaz Baños

Erik Andreasson

Approved, 2012-06-05

Advisor: Johan Garcia

Examiner: Donald F. Ross

Abstract

Reasoning graphs are one of many ways to visualize information. It is very hard to understand certain type of information when it is presented in text or in tables with a huge amount of numbers. It is easier to present it graphically. People can have a general idea of the information and if it is necessary to see the details, it is possible to have a way to add more information to the graphical display. A graphical visualization is able to compress the information, which represented in text can be thousand of lines, to be shown in only one image or in a set of them. Therefore it is a very powerful to transmit information in a blink of an eye, and people will not waste time reading many lines, values or numbers in text or tables.

The developed application has a graphical user interface in 3D which shows the information in a reasoning graph. The user can navigate through the graph in a 3D way, expand the nodes to see more information and change the position of them to restructure the graph.

Acknowledgements

We would like to thank our advisor Johan Garcia for his support and advice. Without his help the project would not be handed in on time. His guidelines and corrections were very important.

Also, we want to thank all the people who read this document more than twice, in special, Antonio Álvarez Bermejo, whose corrections helped.

Contents

List of Figures

1 Introduction

Nowadays, the study of knowledge and intelligence is on the rise. Humanity always has wanted to grow in those fields and, for that, attempted to formalize and standardize everything that we find in the world, and everything that we have in our mind. It can be seen that this formalization and standardization makes it easier to reach the proposed goals. Computers are very useful in order to do those tasks as well as they, in a sense, make our mind grow. They can help to calculate huge amounts of operations, store huge amounts of information and create a new world of possibilities. It could be said that computers extend and complement our mind, since they can do things that our mind can do with more effort or slower, or even things that our mind is unable to do.

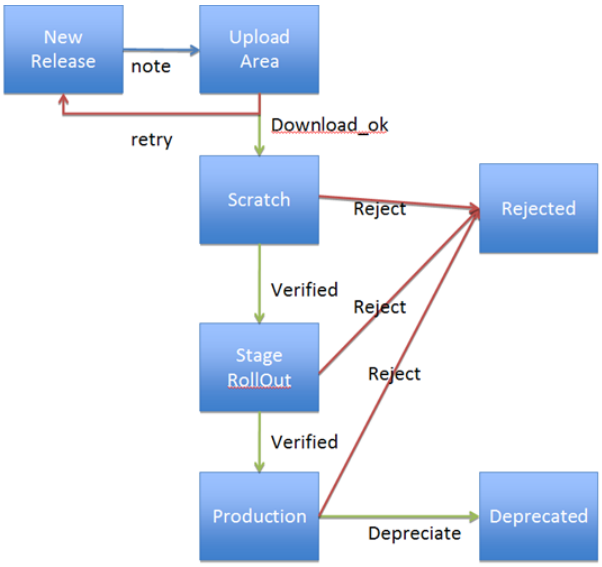

Figure 1.1: An example of reasoning graph[4].

A reasoning graph, also known as conceptual graph, is anything else than a formalism for knowledge representation[10]. We could think that reasoning graphs are only an entity

in computers, but they could be seen as an entity in our mind. They could be like pieces of information, which try to describe how the mind can flow from one state of thinking to another one. In reasoning graphs, we show those flows as lines, and states as boxes, when we try to explain them to some one else by sketching. In a formal way they are nodes and edges of a graph. Computers can be used to store those reasoning graphs, as well as to calculate results of taking different reasoning paths, and so on.

Due to this desire of the knowledge and the usefulness of computers for that purpose, we have to go deeply into details with the study of reasoning graphs, the applications that manage them, and the systems in which they are included.

The ideal application which handles reasoning graphs, should have a user-friendly graphical user interface, since reasoning graphs could be very hard to understand and manipulate if they are big. It should allow the user to be able to perform some operations, like creating new reasoning graphs, saving, loading, changing color and shape of the content. Some other less important features could be a tactile, 3D or multiscreen interface. But all these features should be present by one interested in a commercial deployment or way to use.

There are many formats in which information can be stored. Standardization, mentioned above, leads to the selection of a markup language like XML (Extensible Markup Language), which is, in fact, a standardized language and enables the specification of any type of information. Since this language does not use specific keywords to describe elements and attributes, it is possible to name every element and attribute according to the demand of the particular circumstances. This way, it is very easy to define an infinite amount of different objects of the world, which can contain, in turn, more objects.

Presenting the graphs and their properties in a textual way could be also interesting. If the application allows the export of documents, they can be used as input by other

applications or simply as a summary. It was decided to use XML language to code this functionality since word processors like the one of Microsoft Office, or the one of Open Office, are able to deal with a specific type of XML files. Both applications can open this type of file, as well as make changes in them. That is a very useful and fast way to obtain a formatted document understandable by humans. It can be shown fonts with different color, size and style like italic and bold, which makes the parts of the document more discernible.

1.1 Related work

Applications that we can find in the market right now, meet the requirements cited above. Their interfaces provide a wide set of different boxes to represent states of the graph and arrows to represent edges of the graph. They allow to choose among different shapes and colors for boxes, arrows and fonts. Text can be placed inside the boxes, close to the boxes, over the arrows and even in the arrowhead. It is possible to embed a graph inside the box of other one in order to create a simpler way to see the graph when the omitted parts are less important. And it is also possible to expand these boxes to see the hidden parts when it is desirable to see the graph in detail. This is also interesting when it is needed to join two or more graphs and the resulting graph is very big. Boxes are expanded in a different way to show the information of the nodes represented by them. That information is placed in different fields. These fields are usually title, definition, explanation, summary or even implementation. They depend on the purpose of the software, but some applications allow to specify the purpose and create mentioned fields according to the requirements. Anyway, simple fields are enough to define many different types of graphs for different purposes.

These applications have many modules. Most of them have support for development of diagrams in 2D. For example, one of these modules checks if arrows to and from a node are right connected to it in the container box and present in the inside contained boxes, when

a graph is inside the box of another one. Other module checks if there are disconnected arrows or untitled boxes when we take the graph for finished. Some modules run programs like browsers when we click on a link in a box or like scripts which perform some task. An example of a task would be to generate a text document with all the features and information within the graph.

Examples of software developed with some of this features is explained in the Chapter 2.3.

Some of these features are implemented in the application that we have developed. This application has a 3D engine which makes visual context stronger than 2D, allowing the user to navigate through the graph. It also has a module capable to generate a formatted text document which shows important information about the graph and its components in detail.

1.2 Document layout

In this document are explained what is developed, its purpose and features, which tools are used to implement the application and how is its structure.

What has been developed is an application that displays reasoning graphs which purpose is to represent information in general on screen. This is explained in Chapter 2. The tools used to develop this application are a 3D library, a programming language called Python and an integrated development environment. The study of the available tools and the selection of them are done in Chapter 2.2. The expected functionality for the application is explained in Chapter 2.1 through several requirements. In addition, the features of the application are compared, in the Chapter 2.3, with similar features of other application. The way the reasoning graphs are represented in the application in 3D and how the navigation through it is done, this is explained in Chapter 3. How is implemented

the GUI and the XML modules and how is the architecture of modules and files is explained in Chapter 4. The evaluation of the application features, which of the requirements have been met, is in Chapter 5. And the Chapter 6 explains how the experience was and what could be done in the future.

2 Background

It is necessary to make a research or in-depth study in order to have a more clear idea of what has to be done, what is already done and could be used.

The ***Information Visualization*** is a concept not so hard to imagine, but also not so easy. When we read the words *information* and *visualization* separately, we are able to get the idea about what we are talking, but possibly is not an accurate idea. So, it is necessary to define that concept. Here, we have a definition[14]:

"Information visualization is the interdisciplinary study of the visual representation of large-scale collections of non-numerical information, such as files and lines of code in software systems, library and bibliographic databases, networks of relations on the internet, and so forth"

The word *interdisciplinary* refers to the *Information visualization* as a field of study which includes many other disciplines. One of those is the so related concept of *Data information*. The following definition is one of many others[11]:

"Data visualization is the study of the visual representation of data. Data means information that has been abstracted in some schematic form, including attributes or variables for the units of information"

It is quickly noticed that the words *information* and *data* are very related, but have different meaning as well as their definitions are similar, but not equal.

This definition can be extended to reach a part where the graphs are mentioned. Graphs are not anything else that a way to represent the information that must be visualized, for being displayed in a 2D or 3D way on screen.

"Information visualization deals with representing concepts and data in a meaningful way. Depending on the medium used, information can be visualized in either static (e.g. a graph on a printed page) or dynamic forms."[8]

There are several ways to present the information. For example, a 3D graph is the one used in this application as 2D graph in System Architect. Other applications use a different method as 2D Radial visualization[7], cladograms[9] and dendrograms[12]. Dendrograms are tree-shaped representations. They are very similar to the graphs, since trees are a subset of them. Radial visualization is a method which present a tree with a radial structure, where the root is in the center and its children nodes are the adjacent boxes and so on. The Figure 2.1 is an example.

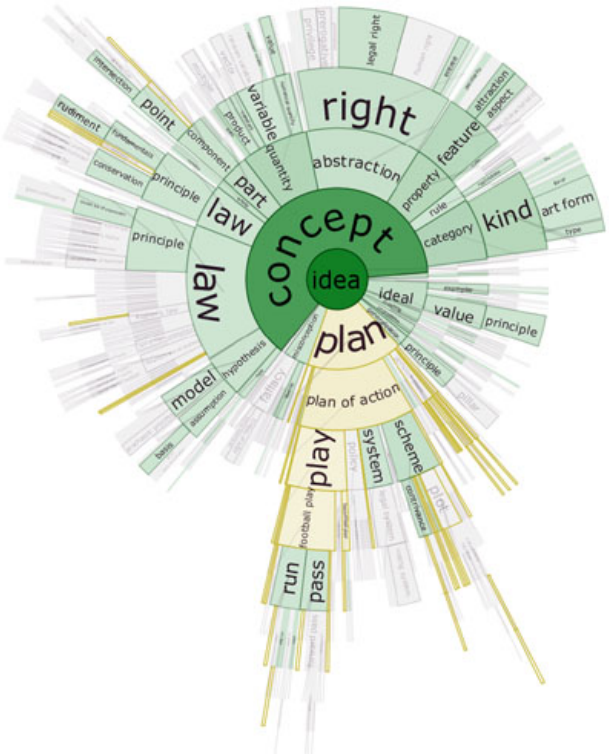

Figure 2.1: Radial visualization[7].

Once the idea of *Information visualization* is taken, it is time to establish some requirements to the application that has to be developed, what tools choose to implement its functionality, what libraries are interesting to make easier the development. For that, in the following sections, different tools will be explained as well as the possible functionality of the application and similar applications will be analyzed.

2.1 Requirements

This section establishes the expected requirements for the application. These requirements try to define different goals or functionality the application should accomplish. They are the following:

1. The application shall be able to represent reasoning graphs in a 3D screen. This implies the boxes must have three coordinates in the space.

2. The application shall be able to represent the information within the nodes and edges on screen. A short description will be displayed inside the box that represents the node, and a long description will be displayed in an extra box that should appear below the aforementioned box.

3. The application shall be able to allow the user to navigate through the graph. This implies to zoom in and zoom out, move the camera up, down, to the right and left, and tilt.

4. The application shall be able to load a graph from a file. It must display the information on screen. It should transform the information of a graph in the file into the appropriate structured representation understandable by the user-interface module.

5. The application shall be able to save a graph in a file. All relevant information of a graph in the current memory of the process which runs the application, must be saved in a file.

6. The application shall be able to generate a file in DOC format. The content of this file will be the principal information of the nodes and edges, omitting some features as 3D representation or identification parameters.

7. The application shall be able to recognise special characters for formatting the text of certain features of a graph such as its descriptions. Those characters must be parsed and omitted when the text is shown on screen or in the document file in DOC format.

2.2 Tools

For developing this kind of application, it is necessary to have a set of tools. It is recommended that the toolkit contains, at least, an *IDE (Integrated Development Environment)*, a *3D Library* which performs the calculations and displays of the graphic part of the system, and, of course, a programming language to implement the modules which handles the *3D Library* and the input and output.

2.2.1 3D Library

Some popular engines that are in active development are *WorldViz*, *Python-Ogre*, *PyGame* and *Panda3D*. All are open-source except *WorldViz*. There are a lot more engines around but this are some of the more active engines. Because *WorldViz* is a commercial engine, there is no interest in that engine for that project. For the rest of the engines there are positive and negative opinions as described in the following paragraphs.

Python-Ogre

Python-Ogre is a 3D engine that has been ported from *C++* to *Python*. The documentation for this engine has a high requirement of knowledge on 3D engines. This is due to the fact that there is not a huge amount of basic tutorials to get you up on your feet. There is also a high level of abstraction which can be both good and bad.

Panda3D

This 3D engine is based on *C++* with a Python wrapper. This 3D engine has a lot of tutorials, example code and almost complete documentation on everything. This engine is pretty easy to get started with and have a lot of information about what is under the shell. The community seems to be active.

PyGame

PyGame is an engine that is really easy to get started with. This is due to all tutorials you can find. There is even some books about it. The problem with PyGame for this project is that it's not a real 3D engine. To achieve 3D some plug-ins will be required.

Selected graphic engine

For this project was panda3D chosen because it has the largest amount of documentation and has a large active community. Panda3D has also not reinvented the wheel for standard libraries like Python-Ogre. Pygame was not a true 3D engine and was therefore not a very interesting engine. When looking at the performance, there was not real test trying the performance between engines, this had not any impact on the choice of the engine.

2.2.2 Programming

Python

Python is a high-level programming language. It is very powerful and readable. It has a large standard library which code is also very readable and not very hard to understand. It supports several programming paradigms like object-oriented, imperative and functional. Automatic memory management and dynamic type system are some of its features, which make work more easily. It can be used even as a scripting language. For more information about this language see [16].

There are many more languages which could be suitable for developing this application such as Java, C, C++, C# or Ruby, but for this project Python 2.7 will be used because Panda3D is using it in their version 1.8.0. For future use of this code it will be written in a 3.0 compatible way. This means that the code will be compiled with following flag *-3*. This flag will print any warning when the Python converter 2to3 cannot handle the syntax.
Some of the changes in Python 3 is the print. In Python 2.7 you could write print "hello world" but in 3.0 it will be like print("hello world").

User interface

The interface of the software will be designed in a 2D/3D manner. It will be simplistic with only basic features for changing the layout of the graph. The displayed content will focus on nodes with a short summary and the dependencies between the nodes.

Interaction in the software will be movement with the mouse, and zooming capabilities will be controlled by the scroll. When clicking on a node the extended information will be displayed.

The 2D interface will only let all argument nodes be displayed on a flat surface, the camera will be locked on a predefined distance from the surface. In 3D it may be placed

anywhere but always facing the same direction. The camera angle will be locked so it never can rotate around its own axis. This will make the navigation less confusing.

2.2.3 Integrated Development Environments

When choosing an integrated development environment (IDE) the following were examined: *Geany*, *Eclipse+Pydev* and *Netbeans+Python* plug-in. These are, listed in the order of lightweight to more extensive IDEs.

Geany

Geany is a lightweight IDE built upon C and GTK+. The graphical user interface (GUI) is minimalistic and serves it purpose. The only GUI feature is a symbol table containing information about variables and classes from the current file. Geany supports syntax highlighting and has the ability to execute the code. On the other side, Geany has not support for debugging, and support for projects.

Eclipse with Pydev

Eclipse is an IDE built with java. The GUI is split in two part where the first is code writing and secondly is a debug layout which is a mess of windows. In the GUI for coding, there is a symbol table displaying variables, classes and functions.

There is support for syntax highlighting and code completion. The *GUI* is constantly looking for errors in the code which gives a feeling that the code is sound. When debugging the code, there is built in terminal to show output and a call-stack. And there is support for stepping through the code line by line.
Negative stuff about the IDE is that it can be a bit heavy for weaker laptops. The setup of a project is a bit trickier because you may need to manually choose the paths to Python.

Netbeans with Python plug-in

Netbeans is an IDE built with Java. The GUI has a clean layout that is strict. There is support for syntax highlighting and code completion. When debugging there is a call-stack and a variable list. The code can be stepped through line by line. The big problem with Netbeans and Python is that this plugin is still in beta phase and are not yet supported on windows. No symbol table have been implemented and the IDE can be heavy for laptops.

Selected IDE

The chosen IDE for this project was in the end eclipse with pydev this because it has all features looked for and it is supported for multiple platforms. Netbeans was the second most interesting IDE looked at mostly for it also used plug-ins, but its downfall was in the end that the plugin for python are only in beta and supported right now only windows. Geany was the most lightweight IDE looked at and lacked some useful features like auto-complete.

2.3 Related work

There are several applications which share much functionality with the one that we have developed. One of them is **IBM Rational System Architect**[17]. This application is a powerful tool in order to model operations and systems used by many corporations and governments. It has the option to choose between several frameworks such as *TOGAF*, *DoDAF*, *MODAF* and *NAF*, as well as has support for many different standards of modelling as UML (Unified Modeling Language) or BMM (Business Motivation Model).

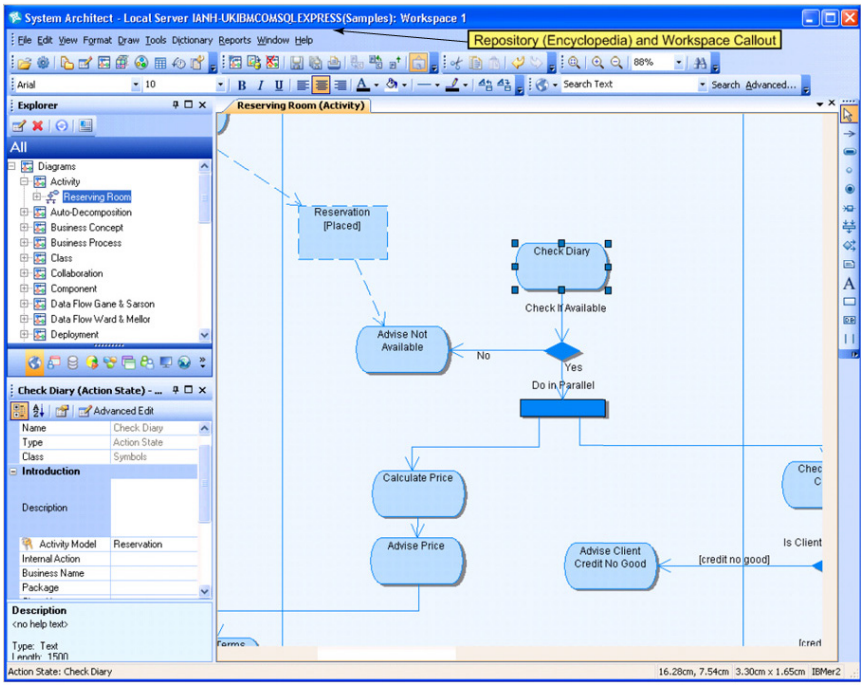

Figure 2.2: Screenshot of System Architect[5].

Why an application to model systems is similar to this?. The modelling of a system using *System Architect* is done by a graphic module. This module represents all the information of the system on screen using graphs. In every system, we can find entities which perform a certain task. They send information to other entities. The flow of this information can be seen as links or edges of a graph. In fact, they are. The interface of this huge application has internally an structure defined by nodes and edges as our required application. That is the similarity between them, but not the only one. *System Architect* has a module that generates a document file in *DOC* format as summary or report of the system. This is a development requirement of our application. A difference is that System Architect has not the 3D navigation that our application has.

There are additional interesting features of *System Architect*. For example, it is possible a node contains a embedded graph, so it can be expanded or contracted depending on

the amount of information we want to see in the display. There is a module which check if the links are connected to that node are present in the subgraph that it contains, and if the links which flow out the subgraph are present in the node as links that flow into another node. Other module checks if there are links disconnected or boxes untitled. In summary, they make easier the modelling task. For more information visit the web of System Architect in IBM[6].

2.4 Summary

This section has provided a background on Information Visualization and has taken a look at some of the more popular IDEs. They have been compared and tested. Finally, *Eclips+PyDev* was chosen for this project. There was a similar look at the 3D libraries where *Panda3D* was chosen because of its active community. Furthermore, an overview about similar work was done and established the expected requirements for the application.

3 Overview of general 3D technology and Panda3D

This chapter will introduce the basics of 3D and how text is handled and how a camera works. There will also be a description on how this is handled in panda3D and with its engine. The most weight for this chapter will be placed upon how the rendering of an object works, how text can be generated with infinite size and some information on panda3D.

3.1 General technology 3D overview

3.1.1 Object representation

To show any given object in a 3D space, it must be abstracted into an object with only flat surfaces. Of course, a ball cannot be represented with a single flat surface, but by making it a cube it will be similar, but not completely equal. When the cube is divided into 12 sides it is rounder, but when it is divided into a 100 sides it will almost look like a ball. By abstracting objects into a lot of flat surfaces, any given solid object can be represented.

To represent a object in 3D it is necessary to use a polygon mesh. This mesh is built up with only triangles. Any flat surface with four corners will be broken down into triangles. Each triangle will have one surface or face represented by the three corners, which is called vertices. Every vertex is a point in the space positioned by three values x, y and z, and is connected to other vertices to form edges.

3.1.2 Camera

To represent the scene it is necessary to use a camera like object with a lens, which will project the world to a 2D surface. This lens will decide field of view and the depth of field. The camera will decide the position, direction and tilt; and will create an 2D projection of the space in front of it.

There are two kinds of camera lenses. These are orthogonal and perspective. These two types are quite different. The orthogonal camera view will make all objects of the same size equally large. When drawing two lines parallel to each other, they will never merge. Using a perspective camera and placing two objects with different distances from the camera, the one closer to the camera will be bigger. If two lines were drawn, they will eventually look like they have merged. In most cases, the perspective lens has more advantages and is used when you want to show 3D with a feeling for depth. The orthogonal lens is mostly used when 2D is used or when it is wanted to keep the proportions.

3.1.3 Text

There are two ways to create a glyph. The simplest way is to represent the letter with a bitmap, while the other method is the use of vector graphic. The bitmap glyphs have some big disadvantages against vector based glyphs like they are not scalable, because each letter size must be defined with a pre-created bitmap of all letters, where vector based letters are defined for all sizes.There are several big libraries for creating glyphs. The biggest is TrueType fonts. Others are Type 1 and OpenType.

In a computer the most common way to create a glyph is to use vector based graphic which will help to generate glyphs with unlimited precision and size. To achieve this with TrueType, Bezier curves are used. For more specific the quadratic Bezier curves are used for a better performance.

To describe a glyph in precision so-called control handles are used. These control handles are fixed points where the line will be drawn through. For more fine-tuning so called “off curve” control points are used, their purpose is to make the drawing of the line more controlled. These two points create the basic foundation of representing a glyph. In Figure 3.1 you can see control handles as boxes and off curves as circles.

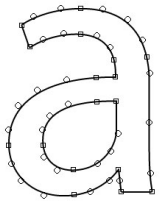

Figure 3.1: Drawing of a letter with Bézier curves.

To represent it on a bitmap all needed to be done is to color all pixels whose centers are inside the lines black. This will create the glyph, but it may be too thick or thin compared to other places on the glyph. This may look bad when the glyph size is small. This can be fixed by using hints to adjust the lines. This will lead to use more appropriate width and leave the glyph correctly sized, but very edgy. By using AA (anti aliasing) the letter will look better and easier to read, see Figure 3.2.

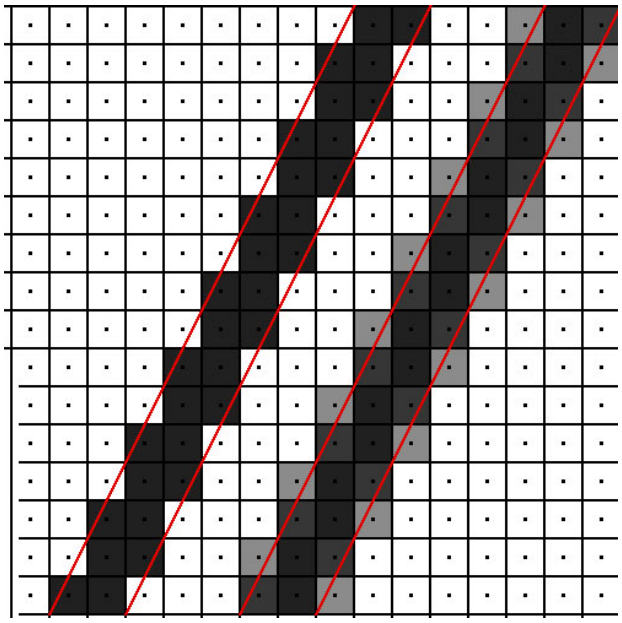

Figure 3.2: No AA on the left and with on the right.

3.2 Panda3D

3.2.1 Introduction to Panda3d

Panda3D is a 3D engine that will help the programmer to hide the more complex parts of programing with 3D. This will help the programmer to create 3D applications faster without re-inventing the wheel and therefore save a lot of time.

Panda3D is built as a scene graph engine which means that all objects that are going to be displayed are added into a graph. When you do an operation on one node, that operation will be sent to the children, while if you do the operation on a scene, all objects in that scene will do the same.

The task manager in panda3D works by doing one callback every frame to all functions added to the list. During this update there can be changes done to the scene graph.

All inputs to the game are done through callbacks that may occur once every frame update. If looking behind the scene of panda3D, it will be seen that the engine is composed of three major parts: app, cull and draw. The app part is all code that is written for the project. This part may be split up into multiple threads. Cull is view-frustum culling, which is the part that will check if an object in a scene will be rendered to the screen. The part draw will do all calls to the graphic card and is, in normal case, the bottleneck that will set the performance. These parts are implemented in different threads which make them work in parallel, see Figure 3.3. The reason why they are shifted one step to the right is because they can only work one at a time on the scene graph in a specific order. There are three instances of the scene graph called draw, cull and app. This will allow them to work simultaneously. This give some latency to the program because it will take two more frames before you can see the effect of your inputs.

Figure 3.3: This shows the execution order in Panda3D.

3.2.2 ScenGraph

The scene graph is a way to represent everything that is in the world. Panda3D uses two different Scene graphs. The first is called render and the other one is render2d. The scene graph render is used to render all the 3D and render2d is used for rendering GUI which always will be placed in-front of render and on the same location.

The scene graph is built up with nodes called PandaNode. Each node will always think it is placed in the center of space, also called origo. All actions done to one node will have affect on all its children indirectly. The effect done to one node will not need to be applied to the sub nodes because their global placement is based upon the parent, this makes them behave as a single object.

3.2.3 3D system

To represent a point in the space, the Cartesian coordinate system are used. The default usage of each axis in panda3D is that the x-axis is width, y-axis is depth and z-axis is height. This is not appropriate for the system in this project because, in most cases, the space will be in 2D representation and then x and y are the default axis normally used. For this project the y and z axes are swapped, which leads to the x-axis are used as width, y-axis for height and z-axis for depth.

The measurement for objects in the space is by default -1 to +1 for the default location of the camera, where -1 to +1 represent the distance of 1 screen length. When representing

a distance in the engine some default values for a game for example may be, 1 unit in the engine will be the same as 1 meter in the real world. But now there is no such things that can be measured in this project, by this fact it was chosen that a distance of 1 will represent 1 pixel when camera is in its default position.

3.2.4 Rendering in 3D

As mentioned before in Chapter 3.1.1 all meshes are built up with triangles. To draw an object in 3D is necessary to know the location of the three corners and the camera angle. This is because, when drawing the triangle, the face normal will be chosen, which is the direction the triangle will be visible from. This direction is based upon if the triangle is drawn clockwise or counterclockwise. If it is drawn counterclockwise the face normal will be directed toward the default camera angle and the surface will be visible for the camera.

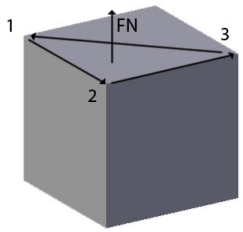

Figure 3.4: Direction for face normal.

When drawing two or more surfaces on the same plane, there will be problems with z-fighting if no counter measurement is used. This will show up as a flickering of colors,

based upon what surface that wins the z-test for that current frame. There is a solution for this problem and that is to use a z-buffer value, which in basic tells the rendering system what surface will have the highest priority. This is in more detail the order that the surfaces will be rendered in the graphic card.

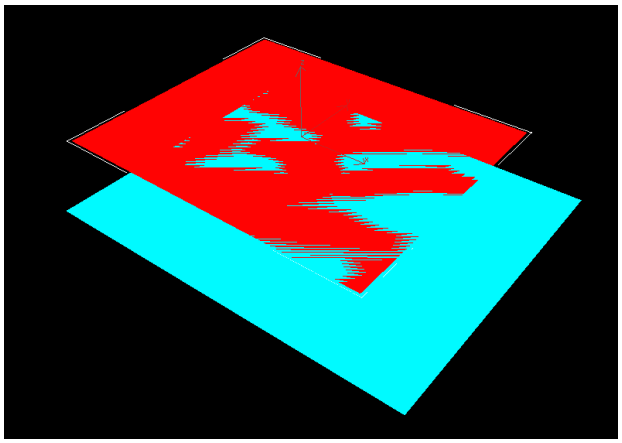

Figure 3.5: Z-fighting two faces in the same space.

3.2.5 Rendering text

The rendering of text in panda3D is done through a library called FreeType. For more information on rendering text read chapter 3.1.3. When the rendering of a glyph is done, it will be represented by a texture. This texture is actually two triangles. When any text is being generated by panda3D, the engine will take control over the rendering systems framebuffer and do one rendering pass with the given text message. This is done because rendering a lot of text is expensive. Specially considering a text consisting of 500 characters. This will be represented with $500 * 2 = 1000$ triangles because each letter will have to be represented by a texture and every texture is built up with two triangles. The drawing of a 1000 triangles over and over again will take its toll on the fps. This can be fixed by using a framebuffer. The framebuffer can convert all those triangles with textures into one single texture that will look identical to what previously was represented by 500 textures.

3.2.6 Bottlenecks

Because python is an interpreted language and not compiled there is obviously some bottlenecks that may occur if no caution are taken. Some of these things are of course heavy or repeated calculation.

When fixing bottlenecks on heavy calculation there are some ways of fixing these. The first solution is of course to perform the calculation using compiled code from the panda3D library. This is often possible when the problem is 3D related like collision testing and text creation.

One big bottleneck for the 3D engine is the information sent over the bus between CPU and the GPU. This may happen because every frame all information of what meshes that will be used must be sent over that bus. This bottleneck will show up when the CPU or GPU are not maxed out, but you still get an FPS drop. This bottleneck can be fixed by removing meshes for rendering or combining them to bigger meshes which leads to less information needed to be sent over the bus.

3.3 Summary

This chapter has described the basic of 3D with information on how text is being rendered. And how the panda3D works and keep itself updated during the rendering, view-frustum culling and application part. Other things that are mentioned in this chapter are bottlenecks, z-fighting, scene graph and face normal.

4 Implementation

The implementation of the graphical user interface is done by a module called *GUI-module*. This module manages the representation of the elements of the graph on screen. It places the boxes that represent the nodes in their respective positions and calculates the points from and to a line that represents an edge has to be drawn. It controls the behavior of the boxes and lines when the boxes are moved as well as the movements of the camera to navigate through all the graph. It also parse the text to show its appropriate format on screen.

All the details about the implementation of the GUI are explained in Chapter 4.1.

The implementation of the input and the output of the file is done by a module. This is called *XML-module*. That is because it handles loading and saving information of graphs and their components in a *XML-file*. The file is named this way due to XLM is its codification language. The details about the XML-file will be explained in Chapter 4.2.3.

When the user wants to open a previously saved graph from a file, the interface commands a loading which make the module performs a reading of the XML-file. This reading is called parsing. The parsing is made by an external module or library. When the parsing is done, a structure is created. That structure is the *XML-tree*. It contains all the information collected from the file properly ordered and ready to be requested. This way, it is easy to get the information about any element of the graph and do changes on it. This information will be used by the GUI-module to represent the graph on screen. The details about the XML-tree will be explained in Chapter 4.2.4.

When the user wants to save all the changes, the interface command a saving which make the module performs a writing of the information of the structure on the XML-file.

The old XML-tree is rejected and built again in order to update all the changes, write the content and save the file.

This module also provides a way to create an empty graph. Then, the user can add and remove nodes, edges and their features.

All the details of the implementation are explained in the Chapter 4.2

4.1 GUI-module

4.1.1 Architecture

The architecture for this software is built upon the main class Engine. This class purpose is to set up the window with all of its properties and launch all other classes that will be needed. After this initialization, the only purpose left is to manage the updates. The Engine class uses three other classes to be able to manage everything. These classes are MyCamera, InputManager and BoxManager. The MyCamera class purpose is to move the camera directly or over time. The InputManager class manages the input given and translate it into more understandable data for the rest of the program. It also manages some basic movement for the camera. The BoxManager controls all boxes in the space and the input given to these. The BoxManager needs a list of nodes that are of a type Node. This class contains all information about a certain box and acts like interface between the GUI-module and the XML-module.

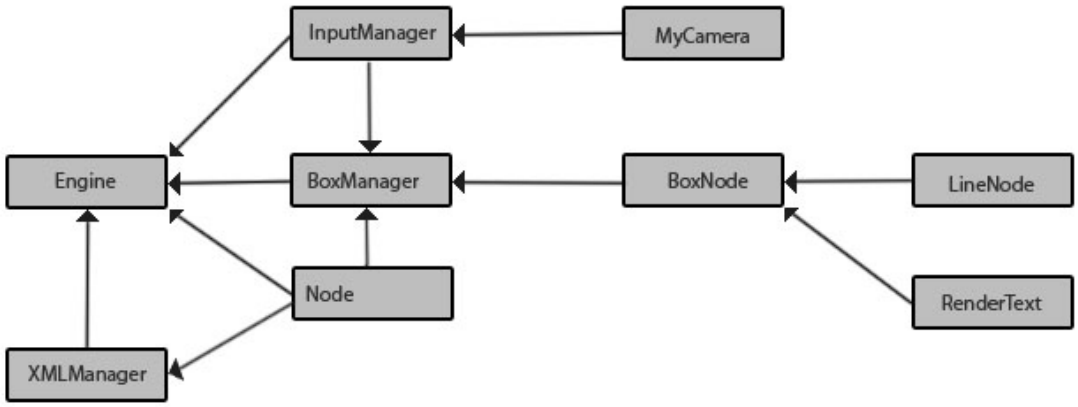

Figure 4.1: Shows the class structure of the program.

4.1.2 Node representation

The nodes are represented with four types of boxes, these are created with polygon meshes. All the boxes will have a variable height, width and borders. These boxes also got

a depth so they can be displayed in real 3D. To be able to draw these boxes some function for drawing meshes where needed. For this a method for drawing circles was made this method works by having a start point and a list of points where the edges are. From this information triangles will be drawn and in the end a large mesh can be created. This basic mesh have some limitation, but is just what's needed to create flat surfaces like the boxes in this project.

4.1.3 InputManager

The InputManager purpose is to manage all inputs given to the project and in more detail tell what the actually input for a key is. This can be done by accepting inputs from panda3D, by using a method that binds a key event to a function with certain preselected parameters. So when a key is pressed it will send an event to panda3D that in turn will call the prearranged function with specified parameters.
When it comes to manage the mouse some special features must be used, because we are in an actual 3D environment the mouse input position will only tell us where the mouse is in relation to the camera view. So to actually tell if the mouse is over a box we have to draw a line from the camera to the position of the mouse. This line can now be used to do a ray-collision with surfaces representing the boxes so the actual position of the mouse is revealed.
The last function for this class is to move the camera around with the inputs that is not used by any other class. This is so that if you click on a box you will not move the camera.

4.1.4 Lines routing

This class purpose is to draw a line between two boxes as good as possible. This line will represent the connection between two nodes. So the line must have some indicator that represent what direction a connection has. One important thing for this class is to make the line so simple as possible and not too thin in order to it can be seen. To draw

a line between the boxes, in the beginning, a simple method based upon outward angles from the center of the box, were used. This method was simple, but had some big flaws like that it did not take into account that if the boxes were too close to each other, or that the box placement was extreme in another way. Then, a new plan was created by ideas from path-finding in games. The idea was to test all possible ways to draw the lines like in Figure 4.2 where the grey box is the start and the other are boxes to stop at, and have penalty cost on not so desirable features like turns and steep angles. This makes the lines more dynamical with less thought required.

This plan has done so that this class uses four methods to calculate the line. It uses the

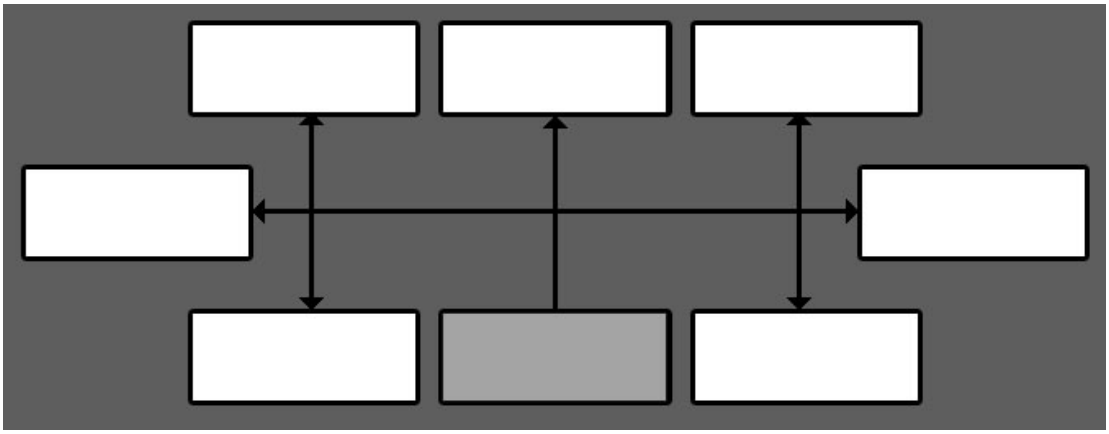

Figure 4.2: Shows possible line test from one start position.

methods in a strict hierarchy order to keep it simple.

The first method takes in parameters about the first boxes. This information is used to calculate all possible ways to draw a line from the predetermined starting positions. These possible lines are tested using the next method.

The second method takes in parameter from where the lines are starting and calculates if these positions are legal. This is done by checking distances between the corners of the boxes, this way any collision happens between these two boxes. Those lines passing this test goes on to the next method.

The third method calculates the lines positions and sends it to the last method.

This last method calculates the cost depending on the line length and the numbers of corners or angles.
This leads to much better auto-generated lines that have a better look and less extreme angles, and allowed for restrictions when drawing lines. This helps the user to more easily see connections between argument nodes.

4.1.5 Text formatting

One thing that were asked for the text was that it would be able to contain links. This requirement was satisfied by adding in a special text formatting class. This class main purpose is to format the text so it fits inside a certain width and add links that can be clicked. But during the development it was realized that this formatting could be more extended with just some small amount of extra work. So what was added finally was a markup language that supports the following commands.

Command	Description
==Header1==	Creates a header with font size 24 and text style bold
===Header2===	Creates a header with font size 18 and text style bold
====Header3====	Creates a header with font size 12 and text style bold
″text″	This will do so the text is displayed as italic
‴text‴	This will do so the text is displayed as bold
⁗text⁗	This will do so the text is displayed as bold and italic
– – –––	This will draw a line across the text field
[[name|a link]]	this will create a click-able link with the text name

This markup language will give a better control over how the text is displayed. So the user can get a fast overview about what is showed.
This markup language has also some limitations like the appearance of the headers is fixed and cannot be changed by adding italic. Other features not added is a method to change

the size of the text or color. But the most important features have been added.
The text is parsed through with a parser of the type LR(1). This means that it parses the text from left to right with a lookahead depth of one to avoid backtracking. This makes the text simple to parse and errors can be found. This implemented parser have no error correction, so when an error is detected, it will simply stop parsing and print out "ERROR" in red text and exit. For example if we take this raw example text.

```
==Main header==Some text.
\'\'Some italic text\'\'
\'\'\'Nice Bold text'\'\'
\'\'\'\'\'ItalicBold link\'\'\'\'\'
----
Some normal [[link|www.link.com]]. some large amount of text.
```

Then put it through the rendering of the text. It will give the following result as in following Figure 4.3.

Figure 4.3: A text rendered inside the program.

4.1.6 Box

This is the class that represents a box in the 3D space. The class in itself will not work because its do not contain the shape of the box that will be rendered, so it will have to be overloaded with a second class containing the shape. These boxes are, in essence, very customizable because when they are rendered the class uses only some predefined data object whose information is given from the XML files like border color, background color, height, depth, width, text margin.
The most important function for this class is the render. This method will only be called once during the startup of the program. So the main thing will be to render the box itself and the text. The box itself has actually no knowledge of its own height until the text is rendered. This is because the text can be formatted with the markup language that was explained in the previous chapter. When the height is known the actual rendering of the box can begin. This is done by rendering it around a starting point. This method will render all shapes that have been planned for this project and will generate the minimum amount of triangles. When it comes to render the border of the box it was just a simple fact to expand the existing shape with the size of the border. This was done by going through the list and taking three points in the list and doing the following calculation.
When the rendering of this box is done the only task left for this method to do is to create ray-collision planes for the InputManager to use. The planes created are for the clicking on the box and clicking on the links.

4.1.7 Camera and input

The purpose of the camera class is to show of the 3D space that is rendered in a so good way as possible so you can easily understand the data. This goal can be obstructed if the movement of the camera is too complex and gives the user to much freedom. So having looked camera angles is not always a bad thing, for this reason the base camera will mostly be looking towards the argument nodes in a fixed fashion. This decision has made some

big impact on what has been prioritized first that involves 3D and camera movement. The most useful thing added for movement have been the throw function. It works by pressing the left mouse button and then moving the mouse and releasing. The average speed is then calculated and the camera moves with that speed in the direction of the mouse.

4.2 XML-Module

4.2.1 Languages

In this section appears which languages are used to code the application and their features in general.

The XML-module module is coded using Python as programming language and the language used to code information on files is XML.

XML is a markup language. It establishes some rules for creating documents understandable by humans and machines. It provides a standard representation for many types of definitions with arbitrary structure. The structure can be specified by *schemas*. Many organizations have its own schema definition. XML has become even in the standard representation format of many applications as *Office Open XML* of Microsoft Office. For more information about this language see [18].

4.2.2 Minidom Library

The library is the best way to handle a file with XML format, since it can read a XML-file and create the XML-tree only by invoking a parsing method. It can also provide an empty tree to create a new reasoning graph and methods to create new elements, add them to the tree as nodes, and remove them.

That library uses the *xml.dom* library of Python, which is based on DOM, to create a tree with all the information about the content of the graph coded on a file in XML. The Minidom library is a simple subset of methods of xml.dom library which helps to easily handle files in XML. For more information see the specification of the library online[2].

The ***DOM* (Document Object Model)** is a way of representing objects coded in several formats such as XML and HTML. Any application which wants or needs to handle files in those formats does not need to implement the parsing and the classes which define the objects in the XML or HTML world. Those application can directly use the *DOM API (application programming interface of DOM)* for parsing the files. This API provides classes and methods to handle the parsing of the files as well as the creation and removal of content of the structure formed by DOM. For more information about DOM visit [13].

The structure created by DOM in its version for Python will be explained later in the section 4.2.4. But, as a summary, it contains classes as Document, Node, Element, Attr (attribute) and Text. The *Document* class defines the structure of a XML or HTML document by a unique convergent structure. The *Node* class defines the basic container of each object in the document. The nodes properly placed and linked form a hierarchy. That hierarchy is a tree. Each node has two different classes. One of those classes is, of course, Node. For example, the root node of the tree is an object of Node and Document classes. The *Element* class represents each node that is not the root or a leaf of the tree. The *Text* class represents all the leaf nodes of the tree since they have no children. A text object is very similar to an attribute of an element, but it is not the same, so they are treated in different way. The *Attr* class defines all needed about the attributes of the document and the elements. For more details visit the web of Python[1].

The Minidom library provides several methods to handle files. The method used to parse the XML-file is *xml.dom.minidom.parse* which returns an object of Document class. So, it returns the root of a XML-tree. Navigating through its links, all the nodes can be reached, so it is returned an entire XML-tree in wide context. And the methods used to write a XML-file could be *writexml*, *toxml* or *toprettyxml*. The three are similar, but the second performs a writing without taking care of the indentation of the code. For more details visit the web of Python[2].

4.2.3 XML-File structure

A graph is formed by nodes and edges which are represented in the XML-file as ***elements***. An element of XML is like a container. It is represented in plain text between inequality signs. It has attributes and can contain other elements. The attributes of an element are represented in plain text with an = symbol between the name of the attribute and its value.

The elements in the XML-file are nodes in the XML-tree. From now on, the elements that represent nodes of a graph will be named *xnodes*, and these ones which represent edges will be XML-edges.

The *xnodes* are represented in the XML-file by the keyword ***inode*** and grouped by the keyword ***information*** as it shown by the following example:

```
...
<information>
   <inode>
       ...
   </node>
   <inode>
       ...
   </node>
</information>
...
```

The XML-edges are represented in the XML-file by the keyword ***lnode*** and grouped by the keyword ***links*** as it shown by the following example:

```
...
```

```
    <links>
       <lnode>
           ...
       </lnode>
       <lnode>
           ...
       </lnode>
    </links>
    ...
```

Information about the nodes and edges are represented in the XML-file as ***attributes***. An attribute in XML is a property of an element. Those attributes in the XML-file are attributes of the nodes of the XML-tree.

Graphs, nodes and edges have certain properties, so they have attributes in their XML representation. The following example shows this:

```
<graph baseNode="2213e067-1082-400a-b08e-de833cf535a7"
      id="0694e229-2f88-447f-a711-450ed0612302">
    <information>
       <inode id="2213e067-1082-400a-b08e-de833cf535a7" type="standard">
           ...
       </node>
       <inode id="ea411072-5e61-48ac-8ab2-64a5be2437c4" type="standard">
           ...
       </node>
    </information>
    <links>
```

```
        <lnode id="ea411072-5e61-48ac-8ab2-64a5be2437c4">
            ...
        </lnode>
        <lnode id="ea411072-5e61-48ac-8ab2-64a5be2437c4">
            ...
        </lnode>
    </links>
</graph>
```

The XML-nodes have some features represented as elements contained by *inode* elements. The most important are those that describe the node. Some of these elements have their values in plain text embrace by two tags with the same value (the second one with a slash right before) others in attributes. The following example shows that:

```
    ...
    <inode id="2213e067-1082-400a-b08e-de833cf535a7" type="standard">
        <shortDescription>
            First
        </shortDescription>
        <longDescription>
            This is the first node
        </longDescription>
        <strength value="0"/>
        ...
    </inode>
    ...
```

The XML-edges contains some elements which have their own attributes. The most important elements contained by the XML-edges are ***fromInode*** and ***toInode***. They have the two nodes which are connected by the link. This is an example:

```
...
<lnode id="ea411072-5e61-48ac-8ab2-64a5be2437c4">
   <fromInode id="2213e067-1082-400a-b08e-de833cf535a7"/>
   <toInode id="ea411072-5e61-48ac-8ab2-64a5be2437c4"/>
   <Applicability value="45"/>
   ...
</lnode>
...
```

There are some features of the presentation, related to the GUI-module, which are represented in the XML-file. For example, the positions of the nodes on the 3D space in coordinates x, y, z. These features are represented grouped by an element named ***presentation*** as the following example shows:

```
...
<presentation type="3D">
   <inode id="69031f64-4d43-48b3-a911-0c6fe5ee73d3">
       <xpos value="0.0"/>
       <ypos value="17.0"/>
       <zpos value="23.0"/>
       ...
   </inode>
</presentation>
...
```

There are three different types of representation. Every one of them has their own elements and attributes in the XML-file, due to the different features that they have. The representation in XML format for Microsoft Office (MS-DOC) is stored in the XML-file this way:

```
...
<presentation type="DOC">
   <inode id="69031f64-4d43-48b3-a911-0c6fe5ee73d3">
       <extraformatting value="w:b"/>
       ...
   </inode>
</presentation>
...
```

This is very useful in order to generate a pretty printed document. It is possible to define the use of different types of font, color, size and format like bold, italic or underlined text.

And it is also possible to specify a way of presentation for *HTML5*.

4.2.4 XML-Tree structure

In this section, it will be explained how the XML-tree, created after parsing the XML-file or right before writing it, is structured.

Its structure is defined by the module xml.dom.minidom of Python. So, it does not admit any change. But it provides a very powerful and easy way to handle documents in XML format.

The root of the XML-tree must be always an object of ***Document*** class. The most important attributes of this node of the XML-tree are *version*, *encoding* and *standalone*.

They will be present in the header of any XML-file. They get their values from the file when it is loaded, and those values are *1.0*, *utf-8* and *no*, respectively, by default, when the file is written. The details of these attributes will be explained later. As all the nodes in the XML-tree, the Document one has the appropriate attributes. Those attributes are *childNodes*, *parentNode*, *previousSibling* and *nextSibling*. They shape the tree since they act like links connecting the nodes.

Other type of node in the XML-tree is the one known as ***Element***. The nodes of this type are the children of the root of the tree. The fact of adding more than one child to the root is considered an error. But, the child of the root is a node which admits more than only one node as child. The elements in the XML-file like graph, information, links, inode, lnode and presentation are xnodes of type Element in the XML-tree. This type of node has different attributes. It depends on the type of element each one represents. This is marked by an attribute named *tagName*. For example, an inode element in the XML-file will be represented as a node of type Element in the XML-tree and the value of its attribute tagName will be inode. As the rest of types of node, they have the attributes childNodes, parentNode, previousSibling and nextSibling as well.

Some of the attributes of the previous nodes Document and Element are defined by attributes of the objects of a class. But, they are some attributes which are variable. The name of those attributes depends on the used in XML-file. They are allocated in a list. The list of attributes is, in fact, an attribute of the objects of a class, which is not explained in this document because they are not very important. That list of attributes is inherited from this mentioned class by the rest of classes defined to represent element nodes.

Elements which contains other elements are related to them by the links formed by aforementioned attributes childNodes, parentNode, previousSibling and nextSibling. So, one element has the others as children and they, inversely, has that one as parent, and

between them as siblings. For example the xnode information has as children inode element nodes in the XML-tree, those inode element nodes has information as parent, and between them as siblings. The following example shows that:

```
XML-file:                          XML-tree:
...                                    ...
<information>                          information node:
   <inode id="1"...>                      children:
   </inode>                                   inode node:
   <inode id="2">                                 attributes: [id,1]
   </inode>                                       parent: information node
</information>                                     nextSibling: inode node (id=2)
...                                            inode node:
                                               attributes: [id,2]
                                               parent: information node
                                               previousSibling: inode node (id=1)
                                               ...
```

Sometimes elements does not contain elements. For example the attribute *strength* of the element node inode in the XML-file is a node of type Element in the XML-tree but it does not contain more elements. That is why it is represented with a final slash right before the less than symbol which closes the declaration of the element. This can be seen in that example:

```
XML-file:                          XML-tree:
...                                ...
<inode ...>                        inode node:
   <strength value="0"/>              children:
</inode>                                  strength node:
```

```
...                                          attributes: [value,0]
                                            parent: inode node
                                ...
```

There is another type of element node in the XML-tree that is the one which represents the plain text elements in the XML-file. The element node *longDescription* in the XML-file has plain text as value of the attribute of the element is representing. This plain text forms an element node of type ***Text*** in the XML-tree as shows the following example:

```
XML-file:                                XML-tree:
...                                      ...
<inode ...>                              inode node:
   <longDescription>                        children:
       Something                                text node:
   </longDescription>                                data: Something
</inode>                                              parent: inode node
...                                      ...
```

4.2.5 DOC-File structure

In this section is explained how the DOC-file is structured.

There are several types of DOC-files. The native file formats uses the *.dot* extension, but they are different. The native formats are Word for DOS, Word for Windows 1 and 2 (Word 4 and 5 for Mac), Word 6 and Word 95 for Windows (Word 6 for Mac), Word 97, 2000, 2002, 2007 and 20010 for Windows (Word 98, 2001, X and 2004 for Mac). The new type has the extension *.docx* which means Office Open XML and it is the international standard for Word 2007 and 2010 for Windows, Word 2008 and 2011 for Mac and can be

used for other applications like Open Office Writer and open source[15]. The new extension is the one used by our application.

The following example shows how a DOC-file would look like:

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<?mso-application progid="Word.Document"?>
<w:wordDocument
xmlns:w="http://schemas.microsoft.com/office/word/2003/wordml">
    <w:body>
       <w:p>
           <w:r>
               <w:t>
                   Short description of the node
               </w:t>
           </w:r>
           <w:br/>
           <w:r>
               <w:t>
                   Long description of the node
               </w:t>
           </w:r>
       </w:p>
    </w:body>
</w:wordDocument>
```

Notice the DOC-file looks like a XML-file. That is because, in fact, it is a XML-file but with different elements. These elements are specific for applications like Microsoft Office

and Open Office. Those applications are able to read XML-file documents and display them in screen like a DOC-file.

As every XML-file, the header is present with the typical values of version, encoding and standalone.

The element named ***mso-application*** specifies the application which the operating system should run in order to open the file. If that line is not in the file, the operating system would run the commonly used application for open files of XML type or a plain text editor instead.

The element ***w:wordDocument*** embraces all the elements found inside the DOC-file. Its attribute ***xmlns:w*** describe the schema used for this type of structure. It is defined by Microsoft.

The element ***w:body*** is used to specify that the contained elements are concrete of a DOC-file, not common to all the XML-file documents.

The element ***w:p*** specifies that its content is inside a paragraph in the DOC-file and must be displayed that way.

The text will be placed in element ***w:t***. The element ***w:r*** opens a new area which can contain several w:t elements. It is also possible to specify some childs of w:r for formatting the text. The format options can change the size, color and type of the text like boldface or italic. The format is applied to all the w:t element contained in the same w:r element.

The element ***w:br*** is used to show a break (new line and/or carriage return) in the *DOC-file*.

There are more elements. Most of them used to add information about format. This format could be about text, paragraph, pages and the whole document. They will not be explained.

All this information can be see in detail in *rep.oio.dk*[3].

4.2.6 Input

The input, in this section, refers to the capability of loading a XML-file. When the user want to open an existing reasoning graph, the parser of Minidom library comes in. It reads the file completely and creates a tree with all the information. Then, from this tree, the information is collected and a structure is created which is used by the GUI-module.

The *loadXML* method is the one which handles this loading. It is explained here in a high-level language in order to understand how it works in detail.

When *loadXML* is called, this method uses the *parse* method provided by Minidom library to parse the input file. Then, the XML-tree is created and returned by this method as its root in an object of Document.

There is something necessary to do that. Creating the structure used by GUI-module. It could happen, when the file is read, the indentation creates a lot of blank nodes. Those nodes are objects of *Text* class of Minidom and they have only space and tabular characters in the *data* attribute, which is the attribute that has the text information. So, those blank nodes must be removed for the printing or writing methods to not print more spaces or tabulator than necessary. The method which performs that action is *_removeEmptyNodes*. It iterates over the XML-tree removing all the blank nodes using the method *removeChild* provided by the Node class of the Minidom library.

When all the blank nodes have been removed, the GUI-module has to call the *getAllNodes* method of the *XMLManger* class. This method will extract all the information held in the XML-tree and create a structure of nodes linked with each others, which is the structure understandable by the GUI-module.

The method getAllNodes goes into a loop. In this loop, for each inode element node of information element, it extracts the id, type, short and long descriptions and strength. If one of those values are missing, the method ignores it and goes on the extraction assuming a default value. When all the information of a node is collected, an object of the Node class is created. This node is the one required by the GUI-module, and it is appended to the returning list. And the method ends the loop.

The next step of the method getAllNodes is to collect the information of the element presentation. There are three different elements named like this, distinguished by their attribute *type*. The one which has *3D* as value of this attribute, has three elements *posx*, *posy* and *posz*. Those three elements have the position of the node on screen. Those values are updated in the node object. For that a searching is performed by a loop. When the node is found and update, the method ends the loop.

The final step performed by the getAllNodes method is the one which search for the links of the nodes. That is done by a loop. For each lnode element child of links element, the method collects the id numbers of the node which the link starts from and the node which the link goes to. Those id numbers are in *fromNode* and *toNode* elements as attributes. Then two loops performs the searching of the nodes in the returning list of nodes and when they are found, the list of children nodes of the from-node is updated appending the reference to the to-node. And, finally, the list of nodes is returned in order to the GUI-module print them on screen.

4.2.7 Output

The output of the XML-module can be two different files: the XML-file and the DOC-file. The first one is the file in which all the information necessary to represent the graph is saved. The DOC-file, however, is only a generated document where can be found a summary of the most important information of the graph.

There are two similar methods which each one handles a type of file. They are saveXML and saveDOC. Both methods creates an XML-tree with all the data from a list of nodes just before writing the file.

Then, the saveXML method is explained in a high-level language in order to understand how it works in more detail.

When the method saveXML is called, first of all, it starts by opening the output file and creating the root of the XML-tree is created: the Document node. The attributes version, encoding and standalone are written. They specify the version of the XML-file, the encoding of the characters and the possibility that the document has relationship with other documents. That three attributes form the header of the XML-file.

Then, the method creates the element *graph*, and their two attributes *baseNode* and *id*. The first has the number that identify which node is treated as the base of the graph. This node marks where to start the fetching of the information by the GUI-module to print the whole graph in screen. The second attribute identify uniquely the graph from the rest.

The next element node created is the one named information. This element node is used to group all the inode element nodes. These nodes have the information that can be added to the nodes of the graph. The information element node has to be child of the graph element node.

In like manner, lnode element nodes, which has the information of the edges of the graph, are grouped in an element node named links. This is the next element node created by the method. The same way information became child of graph before, links has to be too.

Then, three similar element nodes are created. They are presentation element nodes. Each one with a different attribute type with one of these three values: 3D, DOC or HTML5. The three ones are children of graph element node.

Now, the method goes into a loop. For each node in the list of nodes, it has to create a xinode element node. It has the attributes id and type. The method also creates three more element nodes: shortDescription, longDescription and strength. Short and long descriptions has a plain text node as child, in which the short and long descriptions text is placed. The strength has not a plain text node. It has an attribute instead. It is value which is a number. The three ones element nodes are children of xinode element node. And every xinode element node created in the loop becomes child of information which is the element node that groups all the nodes with information.

While the forementioned loop is taking place, the method starts another one. For every child in the list of children of the currently evaluated node, the method has to create a xlnode element node. The only attribute of this type of element node is id. But, it contains at least three element nodes. They are fromInode, toInode and applicability. They are also created by the method. The fromInode and toInode has id attribute. They refer to the nodes connected from the first one to the second one, forming a link or edge in the graph. The applicability element node has an attribute named value with a number. The three element nodes are children of xlnode element node created in each loop. And every xlnode element node becomes child of links element node, which is the one that groups all the xlnode ones.

Without leaving the loop, three xinode element nodes are created. Each one will be child of a type of forementioned presentation element node. If the type attribute of presentation element node is 3D, *xpos*, ypos and zpos element nodes will be created and become children of xinode element node. The xpos, ypos and zpos element nodes has an attribute called value with the coordinates x, y and z of the position in where the nodes of the graph are located in the screen. The xinode will be, then, child of presentation element node with the attribute type with the value of 3D. If the forementioned attribute has the value of DOC, then, the content of xinode will be a extraformatting element node with an attribute named value which specifies the format of the information of the node in the DOC-file.

And finally, the method writes all the content of the XML-tree on the file and closes the file.

It is possible to save the content in the same file as the loaded before. That is possible because the file is closed right after the loading and open again right before the saving. So, that provokes the overwriting of the file.

4.2.8 Interface

The interface of the XML-module is defined by the methods used to communicate with *GUI-module*. These methods are very simple and allow to hide the more complicated input and output methods used to handle the files.

The class which implement the XML-module is called ***XMLManager***. Once an object of this type is created, it allows to use some public methods. They are loadXML, saveXML, saveDOC, getAllNodes and getBaseNode. The private methods _createTextElement, _createXinode, _createXlnode, _createDocument, _createTextRun has been created to help keep the code more readable and to refactorize.

The method ***loadXML*** is the one which calls the parse method of xml.dom.minidom library. It creates a XML-tree with all the information of the graph represented in the XML-file.

Once the method loadXML is called and the XML-tree is created, the methods ***getAllNodes*** and ***getBaseNode*** can be called. The first one provides all the nodes in the XML-tree placed in a list of nodes. That list contains objects of the Node class. This class was created only for being used by both modules XML-module and GUI-module in order to communicate. This class is not the same as the one in xml.dom.minidom library. And the second method cited: getBaseNode is useful to get the node known as base, which is the first checked by the GUI-module to create the structure on screen. The value returned by this method is the value of the attribute baseNode of the graph element node, which is present in the XML-tree and in the XML-file.

If the user commands to save the current status of the graph in a file, the GUI-module should call the method ***saveXML***. It creates an XML-tree with the current information of the graph and saves all the content in the chosen XML-file.

The same is performed by the XML-module when the user wants to generate a DOC-file. In this case, the method called by the GUI-module should be ***saveDOC***. It also creates an XML-tree but with different tags and attributes, properly selected in order to generate a document understandable by the required external applications.

4.3 Summary

This chapter explained the basic implementation of the architecture of this application and some of its available features. There is some information about how the text features works and how this can be used to format text.

5 Evaluation

The functionality of the application has met almost all the previously established requirements. The estimations done at the beginning of the project for each functionality are explained later.

5.1 Details

The following sections describe the goals required for both modules XML-module and GUI-module.

5.1.1 XML-module

The following requirements were established as goals of the XML-module. They are explained in detail as well as the solution which satisfies each requirement.

Requirement:
The application shall be able to load a file in XML format. It must display the information in this XML on screen. It should transform the information of a graph in the file into the appropriate structured representation understandable by the user-interface module.

Result:
The XML-module provides a method called loadXML which performs the required functionality. It transforms the information in the file into a structured one, which is provided by the method called getAllNodes when the GUI-module asks for it.

Requirement:
The application shall be able to save a file in XML format. All the information of a graph in the current memory of the process which runs the application, must be saved in a piece of second or permanent memory: a file.

Result:

The XML-module provides a method named saveXML which performs the required functionality. It transforms the information of the graph in running memory and codes it in a file. The above-mentioned method loadXML will be used to open that file in order to recover the information on it.

Requirement:

The application shall be able to generate a file in DOC format. The content of this file will be the principal information of the nodes and edges, omitting some features as 3D representation or identification parameters.

Result:

The XML-module provides the saveDOC method for this requirement. It extracts information of the graph, such as nodes and their descriptions and strength parameter; and edges and their linked nodes and applicability parameter. It ignores the position of those nodes and edges in the screen and the identification number. So, those parameters are not printed in the document.

Requirement:

The application shall be able to recognise special characters for formatting the text of certain features of a graph such as its descriptions. Those characters must be parsed and omitted when the text is shown on screen or in the document file in DOC format.

Result:

The XML-module recognises special characters on the short and long descriptions of the graph in the XML-file by the method _parseText. It parses them in order to the GUI-module shows the descriptions on screen without those characters and with the appropriate format.

5.1.2 GUI-module

The following ones are the established requirements that the GUI-module must meet.

Requirement:

The application shall be able to represent reasoning graphs in a 3D screen. This implies the boxes must have three coordinates in the space.

Result:

This requirement was satisfied with the GUI-module by loading in data from the XML-module and create 3D objects of different shapes. To show the connection between these object a class for drawing lines were made.

Requirement:

The application shall be able to represent the information within the nodes and edges on screen. The short description will be displayed inside the box that represents the node, and the long description will be displayed in an extra box that should appear below the before mentioned box.

Result:

This was done with the class BoxNode. The class will generate two boxes with information from the XML-module that will give information about type, description, width and much more.

Requirement:

The application shall be able to allow the user to navigate through the graph. This implies to zoom in and zoom out, move the camera up, down, to the right and left, and tilt.

Result:

The MyCamera class can handle keyboard and mouse inputs that will make the camera able to move. It will only handle movements in the X and Y axis and changing the camera lens. This will satisfy all the need for movement in a 2D system, but not in complete 3D.

Requirement:

The application shall be able to recognise special characters for formatting the text of certain features of a graph such as its descriptions. Those characters must be parsed and omitted when the text is shown on screen or in the document file in DOC format.

Result:

This feature was fixed with a class RenderText that could parse through a text segment with tokens to represent headers, bold, italic, and links. This class formatted the text and then render it piece by piece until it was done.

5.2 Overall Evaluation

The look at the boxes was as hoped in the end. They are nice and sharp in the edges. They are easily customizable with different colors. The text became almost as sharp as hoped, but the cost in memory have made some limitations on how big the text size could be. This is because the limitations in smaller GPUs where the memory standards are limited to 64MB. But this is just a problem because of the zooming if the limit on the zooming is large, then the problem with the sharpness of the text goes away. The look of the software became as imagined in the beginning of the development and an example of the software can be seen in Figure 5.1.

The functionality expected in the XML-module has been accomplished. It was expected the module would be able to load and save documents in XML format for maintaining the information stored and for recovering, and it was achieved. It was also expected that

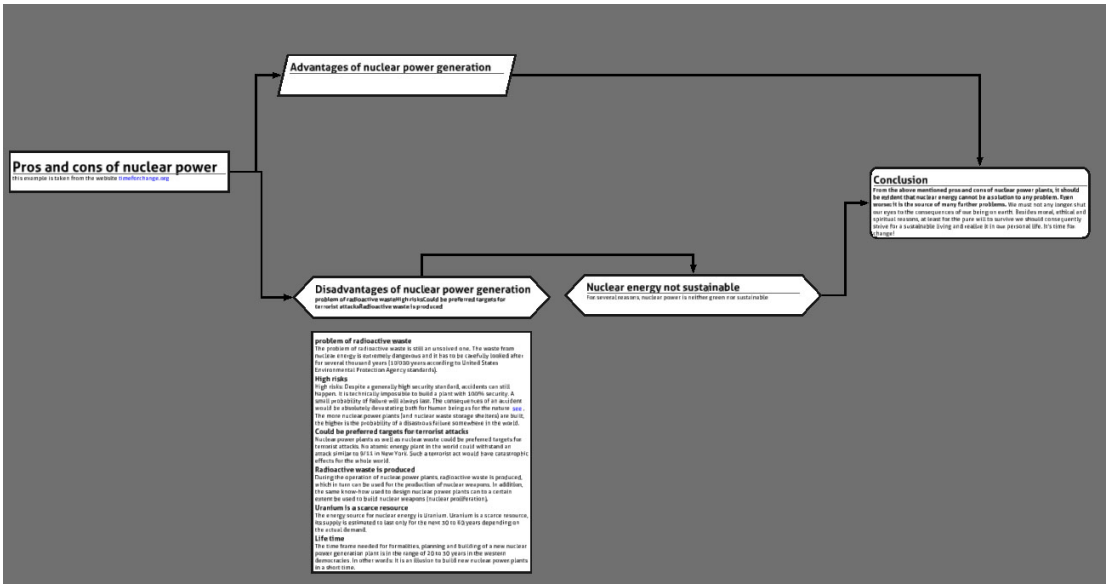

Figure 5.1: Shows a cropped screenshot of the software.

this module could generate files in XML format for text editors and also was achieved. However, not all types of format are supported by this last feature.

5.3 Summary

In this chapter, the solutions applied to meet the requirements have been explained separately for both modules and an overview of the most important issues of the development of the application has been done in the Chapter 5.2.

6 Conclusion

The goal for this project was to create a software that will help to visualize an argument chain. This will be good when trying to see the whole picture of an argument, from both sides of the view.
This chapter will contain some information on ideas for the future of this project and some experiences that have been gained during this project.

6.1 Experience

A lot of knowledge have been acquired when working with python mostly as a consequence of having no previous experience using it. But it was not a difficult language once you learned that there are several versions in use. Some new understanding was gained from panda3D as it uses a scene graph to represent its object for 3D. To get started with panda3D was not problematic because earlier experiences with 3D engines, the big step was using an interpreted language such as python.

6.2 Future development

With the base of the 3D implemented there is still stuff left to do and improve. This is because of the time constraints for this project is not infinite. So some interesting features have not been added that would have been nice to have. Some of those ideas will be explained here.

Loading bar

For the slower computers there is a need for a loading bar when some bigger argument chains are loaded. This loading bar is mostly for the text rendering part which can be a bit heavy to generate. The text rendering part can take several seconds on slower PC or just

a fraction on the more powerful PC. This is so the user will get a feeling that the software is working and have not crashed and show that something is still happening.

Mouse icons

This feature would replace the cursor icon with a hand, cross and an arrow. This will clarify for the user what action he/she will be able to do. And therefore make the program more user-friendly. And it will follow the more set up expectation on how a software should handle.

Optimization

One really desirable optimization would be to render the entire scene with a frame-buffer. This optimization could be used when using orthogonal camera or when a perspective camera is not moving. This optimization will do so that the graphic-card will not need to be used to maximum if there is a limit to the fps or just completely stall the updates.

Improved line drawing

To make lines drawn better and make less of them, we can do so they join up on certain spots. This can be done by adding intersection nodes in the lines and make the lines aware of where all other boxes placement. This will lead to less lines drawn and by that make the interface cleaner and easier to understand for the user.

Format information

The format supported by the application is very limited due to the time spent on developing. There are many additional information that could have been represented in the *DOC-file* with many different kinds of font, colors, sizes and font faces.

HTML5 support

It would be interesting to implement the necessary code to generate a document file for the HTML5 format. So, the browsers could open it and show the information of the graph in many different ways.

6.3 Summary

When looking at the initial requirements then the software have met its goals except for movement in 3D and some extra features were added like text, this will give the software a more polished touch.
Some new ideas for the future development have been described and will hopefully some day be implemented.

References

[1] *DOM implementation*. http://docs.python.org/library/xml.dom.html, 2012-05-16.

[2] *MINIDOM implementation*. http://docs.python.org/library/xml.dom.minidom.html, 2012-05-16.

[3] *Overview of WordprocessingML*. http://rep.oio.dk/microsoft.com/officeschemas/wordprocessingml_article.htm, 2012-05-16.

[4] Hellasgrid. *Design document*. http://wiki.hellasgrid.gr/wiki/bin/view/HellasGrid/EGIUMDRepository/DesignDocument, 2012-05-16.

[5] IBM. *System Architect*. http://www.ibm.com/developerworks/rational/library/10/system-architect-workspace-scenarios/fig02_lg.html, 2012-05-16.

[6] IBM. *System Architect*. http://www-01.ibm.com/software/awdtools/systemarchitect/, 2012-05-16.

[7] Infosthetics. *Document visualization*. http://infosthetics.com/archives/2007/04/ document_visualization.html, 2012-05-16.

[8] Robert Spence. *Information Visualization: Design for Interaction*. Prentice Hall, 2nd edition, 2007.

[9] Wikipedia. *Cladistics*. http://en.wikipedia.org/wiki/Cladistics, 2012-05-16.

[10] Wikipedia. *Conceptual graphs*. http://en.wikipedia.org/wiki/Conceptual_graph, 2012-05-16.

[11] Wikipedia. *Data visualization*. http://en.wikipedia.org/wiki/Data_visualization, 2012-05-16.

[12] Wikipedia. *Dendrogram*. http://en.wikipedia.org/wiki/Dendrogram, 2012-05-16.

[13] Wikipedia. *Document Object Model*. http://en.wikipedia.org/wiki/Document_Object_Model, 2012-05-16.

[14] Wikipedia. *Information visualization*. http://en.wikipedia.org/wiki/Information_visualization, 2012-05-16.

[15] Wikipedia. *Microsoft Word*. http://en.wikipedia.org/wiki/Microsoft_Word, 2012-05-16.

[16] Wikipedia. *Python*. http://en.wikipedia.org/wiki/Python_(programming_language), 2012-05-16.

[17] Wikipedia. *System Architect*. http://en.wikipedia.org/wiki/System_Architect_(software), 2012-05-16.

[18] Wikipedia. *XML*. http://en.wikipedia.org/wiki/Xml, 2012-05-16.