School of Mathematics and Systems Engineering

Reports from MSI - Rapporter från MSI

Implementation of 3D Kiviat Diagrams

Student: Guo Yuhua
Supervisor: Andreas Kerren

Oct
2008

MSI
Växjö University
SE-351 95 VÄXJÖ

Report 08115
ISSN 1650-2647
ISRN VXU/MSI/DA/E/--08115/--SE

Abstract

In this thesis, a 3D approach to visualize software metrics is presented. Software metrics are attributes of a piece of software or its specification. They generally contain a set of multivariate time-series data and can be displayed, for example, as a Kiviat diagram consisting of axes and polylines. The aim of this work is to design a Win32 application that can load multivariate time-series data from a file and visualize it as an interactive 3D Kiviat diagram.

There has been an approach that can display software metrics by using 2D Kiviat diagrams, but there are still some drawbacks on it. Since a better visualization of software metrics can help the developer to control the quality of software products more easily, this thesis improved the existing approach by extending 2D Kiviat diagram to 3D Kiviat diagram.

Keywords of this report are:
Kiviat diagrams, Star Plots, Star Glyphs, Information Visualization, Software Visualization

Content

1. Introduction

This chapter introduces the problem and the objective that should be fulfilled in this thesis.

1.1 Problem

A software metric is a measure of some property of a piece of software or its specifications. Since quantitative methods have proved so powerful in other sciences, computer science practitioners and theoreticians have worked hard to bring similar approaches to software development. Tom DeMarco stated, "You can't control what you can't measure."[16] Moreover, only measure is not enough. We should also analyze the data which we have measured and get some useful information from that. Then, we will have a good control on software development. The following table is an example of several software metrics.

Nr.	Metric	Release 0.92	Release 0.97	Release 1.0	Release 1.2	Release 1.4	Release 1.6	Release 1.7
1	nrACouples	6	12	18	24	30	36	42
2	entropy	115738	165695	189391	212873	247570	318324	337721
3	nrMRs	30046	45600	58060	67631	83344	106523	113253
4	nrCouples	15286	25809	34073	41112	53573	67170	71901
5	in_nrACalls	6	6	6	6	6	6	6
6	in_nrCallers	886	972	769	772	768	859	835
7	in_nrCalls	1256	1307	1116	1109	1099	1459	1560
8	nrAttrs	906	988	1118	1236	1292	1316	1293
9	nrClasses	459	476	528	566	595	607	609
10	nrDirs	44	45	50	50	50	50	49
11	nrFiles	397	405	443	464	477	485	492
12	nrFuncs	10135	10275	10634	11148	11445	11464	11398
13	nrGlobalFuncs	333	880	288	325	341	334	330
14	nrGlobalVars	219	227	234	262	250	237	229
15	nrMeths	9802	9395	10346	10823	11104	11130	11068
16	nrPackages	0	0	0	0	0	0	0
17	nrVars	1125	1215	1352	1498	1542	1553	1522
18	out_nrACalls	4	3	3	3	4	4	3
19	out_nrCallers	566	597	575	575	623	638	429
20	out_nrCalls	1339	1631	1640	1657	1761	1808	1309

Table 1.1: Measures of source code and evolution metrics of 7 releases of Mozilla's DOM module

In Table 1.1, the metric names are using abbreviations. For example, "nrACouples" means number of abstracted logical coupling relationships, and "nrMRs" means number of modification reports involved in the logical coupling relationships. By measuring the metrics of the previous 7 releases, the trend of future releases was revealed. With these data, software engineers can estimate the effort that will cost in future development and thus the quality of software can be controlled. However, it will be not only boring, but also inefficient if we just list the data we have measured and hope to get something from a mass of digits. A good representation of software metrics can help us to pick up

useful information from data source easily and efficiently. For example, according to the increase of modification reports, the software engineers may decide to use some appropriate tactics to improve the modifiability, and see how the result is going to be.

Generally, one single software metric has a two-dimensional data set, including the value and the time it was measured. Kiviat diagrams (or Star Glyphs, Star Plots) are widely used in two-dimensional visualization. As shown in Figure 1.1, they have axes for each attribute, and the axes are arranged like a star. The values of attributes are dots on the corresponding axes. Each dot is connected to another two dots near by. Such diagrams can not only represent the value of each attribute exactly, but also can help us to have an overview of the whole object.

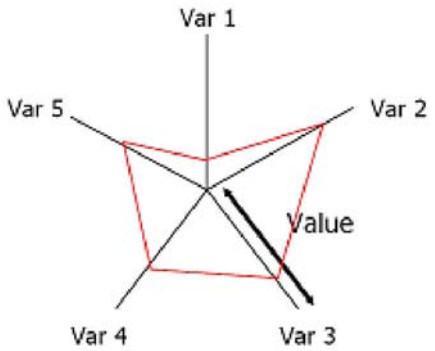

Figure 1.1: A typical 2D Kiviat diagram (Taken from [1])

However, Kiviat diagrams are often not suitable for visualizing the entire data set when the values of attributes change during a period of time. In a 2D Kiviat diagram, there is only one axis for each attribute, i.e., for each metric in our case, so that it is difficult to represent how each of them is changed. Moreover, in 2D Kiviat diagrams, the area of the shape indicates the integrated performance of an object. Generally, bigger area means better integrated performance. It is even difficult to represent the evolution of the integrated performance because it is rather hard to compare many shapes with each other in one two-dimensional space.

Pinzger et al. [2] presented the RelVis visualization approach that concentrates on providing integrated condensed graphical views on source code and release history data of up to n releases by using 2D Kiviat diagrams. As shown in Figure 1.2, the Kiviat diagram shows 20 metrics of software and how their values are changed from release 1 to release 7. Each point of intersection on the axes exactly points out its relevant value. For example, the blue bar shows the difference of the metric values between release 1 and release 2 and the red bar shows the difference between release 6 and release 7. The advantage of this diagram is that the changes between values are emphasized by colors so that it is easy to get both the current state and the trend of each metric.

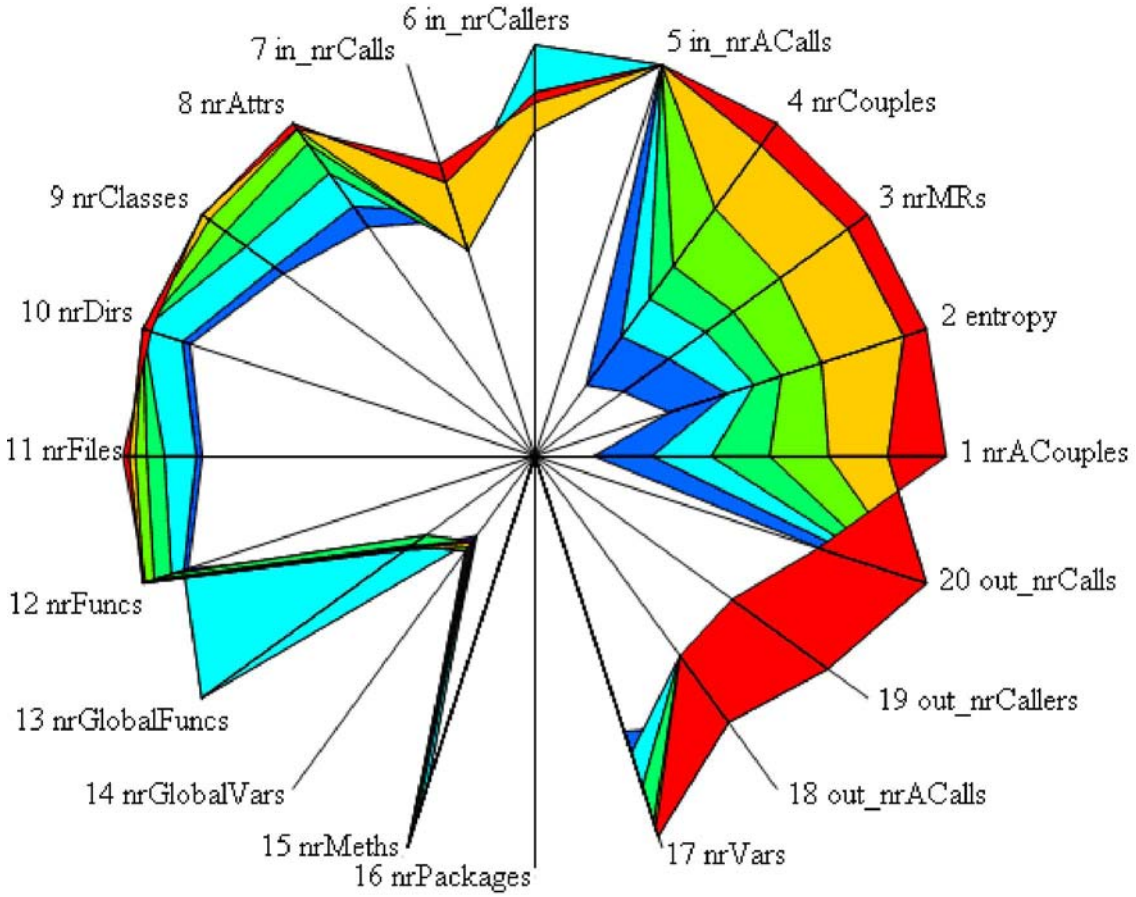

Figure 1.2: A Kiviat diagram presenting software metrics (Taken from [2])

There are mainly two drawbacks in this Kiviat diagram:

- **Overlapping**
  When an increasing value is going to decrease or a decreasing value is going to increase, the previous information will unavoidably be overlapped by new ones. As the Figure 1.2 shows, overlapping occurs obviously on metric 7, 18, 19, and 20.
- **Direction Problem**
  It is hard to recognize the direction from release *n* to release *n+1*. The direction of the change during the recent two releases is even ambiguous. Figure 1.3 gives as a detailed view on this problem. If we focus on the change of metric No. 13, which is named "nrGlobalFuncs", between release 2 and release 3, the direction is rather confusing. It is hard to make out whether the value was increased (as shown in the right part) or decreased (as shown in the left part) during that period of time. In such kind of diagrams, when the value changes distinctly, the corresponding color will be easy to recognize. Otherwise, the corresponding color will be almost invisible. We have to recognize the direction with the help of other colors next by, so when there is only one distinct color on one axis, like metric 13 in Figure 1.3, the direction problem will happen. As shown in Figure 1.4, the direction problem can also be caused by overlapping. Two different data sets have just the same representation. In the upper case, all metrics increased from release 1 to release 2, and then they increased again from release 2 to release 3. In the lower case, all metrics increased more largely from release 1 to release 2, but then they decreased from release 2 to release 3. But coincidently, the two different cases have the same visualization. So, if we have a diagram like that, we can not recognize which case it should be.

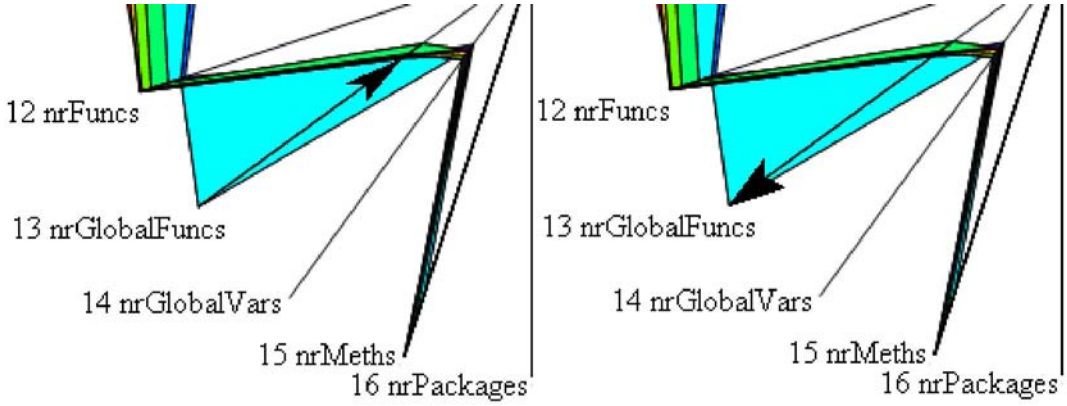

Figure 1.3: Direction problem in 2D Kiviat diagram
(Taken from [2] and modified)

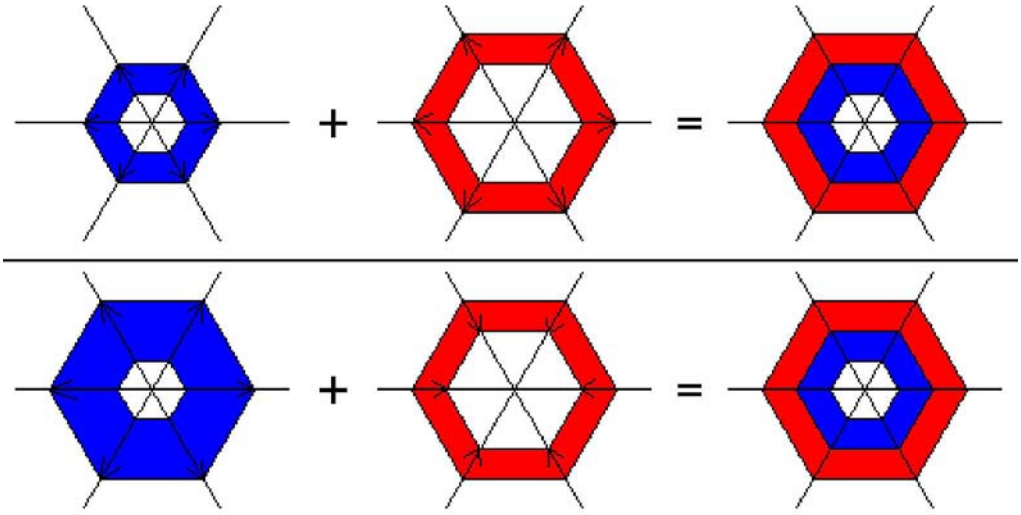

Figure 1.4: Another direction problem in 2D Kiviat diagram

1.2 Motivation

Software visualization encompasses the development and evaluation of methods for graphically representing different aspects of software, including its structure, its execution, and its evolution [17]. Visualization of software metrics reveals the structure and evolution of software. A good approach for the visualization of software metrics is helpful in controlling the quality of software products during the development and maintenance. For example, if a developer wants to know the growth of classes or the increasing/decreasing of coupling dependencies between two classes, he/she can get information from a comprehensive diagram and get a better understanding of the current state and future trend of a source code entity.

1.3 Goal

The goal of this thesis is to implement an interactive 3D Kiviat diagram for visualization of integrated condensed graphical views on source code and release history data of up to *n* releases. The result is a **Win32 application** that can open a data file and visualize an appropriate and understandable view of the whole data set. The initial representation should be a 2D Kiviat diagram as shown in Figure 1.2. Then, the user can rotate the diagram by direction keys on the keyboard, select axes by mouse

click, unfold the selected axes and adjust the transparence of unselected axes by keyboard or by the mouse wheel. And finally, the user can reset the whole diagram to the initial 2D view by keyboard.

1.4 Criteria

The solution for software metrics visualization supports a user in comprehending software metrics. The visualization tool is a Win32 application; its development was driven by the following criteria.

- **Generality**
  The solution is composed of components for data source, data model, data processing and data presentation. It is appropriate for representing all kinds of multivariate time series data.
- **Usability**
  The program has a good GUI so that the user can interact with the program easily.

1.5 Restrictions

In this thesis we assume that the measures of software metrics have been completed. We just use the data directly and focus on the representation of 3D Kiviat diagrams.

The solution accepts only one file format as data source. The data file has to be prepared before running the program. It can be edited in any text editor, and the format of it is restricted.

1.6 Structure of the Thesis

This thesis reports about the implementation of a 3D Kiviat diagram, which improves the 2D approach and give a better visualization of software metrics. The report contains five chapters each reporting about a phase in developing the visualization tool.

At the beginning, a series of reading related articles was discussed. Thereafter follows a discussion about possible visualization approaches and feasibility evaluations. The third chapter describes the design and implementation of the visualization tool. Finally, a chapter concludes the report and describes future work.

2. Related Work

Parallel Coordinates [5, 6] and Star Glyphs [7, 8, 9] are both well known multi-dimensional visualization techniques. Parallel Coordinates have parallel axes for each attribute and connect adjacent value points with polylines. Star Glyphs are similar to Kiviat diagrams. They are small Kiviat diagrams and look like icons. There are still many researches about how to extend them and improve the set of interactions possible with them [9, 10, 11, 12, 13, 14].

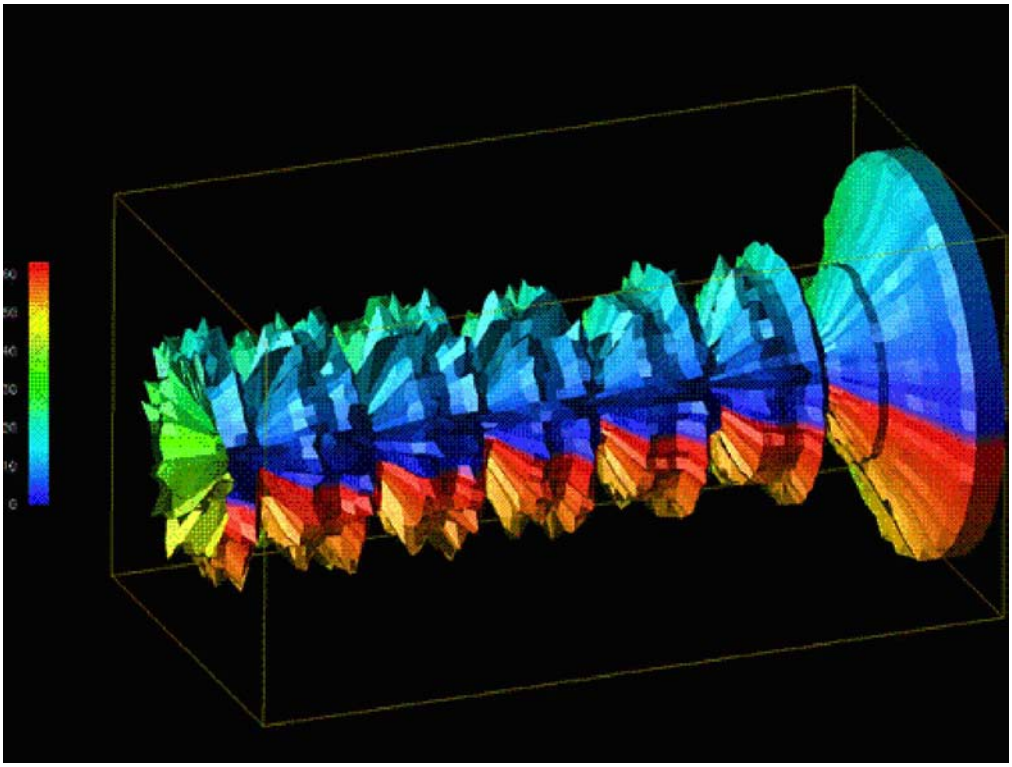

Figure 2.1: Kiviat Tube
(Taken from [15])

Kiviat tubes, as shown in Figure 2.1, are obtained by displaying Kiviat diagrams along a time axis and rendering the surrounding surface as suggested by Hackstadt and Malony [15]. The surface that connects all Kiviat diagrams emphasizes the shape of the tube and conceals information about individual data items.

Fanea et al. [3] extended 2D Parallel Coordinates into 3D Parallel Coordinates, as shown in Figure 2.2, by unfolding Parallel Coordinates and thus arraying 2D Kiviat diagrams in a cylinder. The result of this approach is similar to Kiviat Tubes. But the quantity of the data set is still limited because when there are too many data, it is still hard to recognize them and compare them with each other.

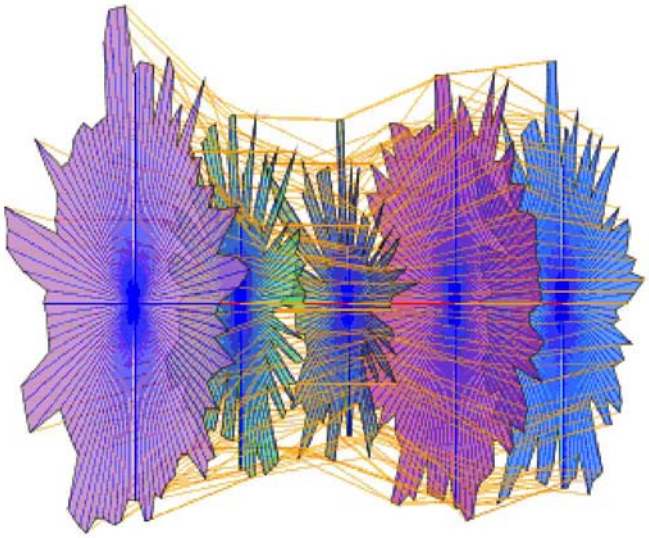

Figure 2.2: 3D Integration of Parallel Coordinates and Star Glyphs (Taken from [3])

Tominski et al. [4] discussed some approaches for interactive axes-based visualizations that can be used to explore and analyze multivariate time series data. The main idea is to extend the axes to a square and lay a time plot on it, see Figure 2.3.

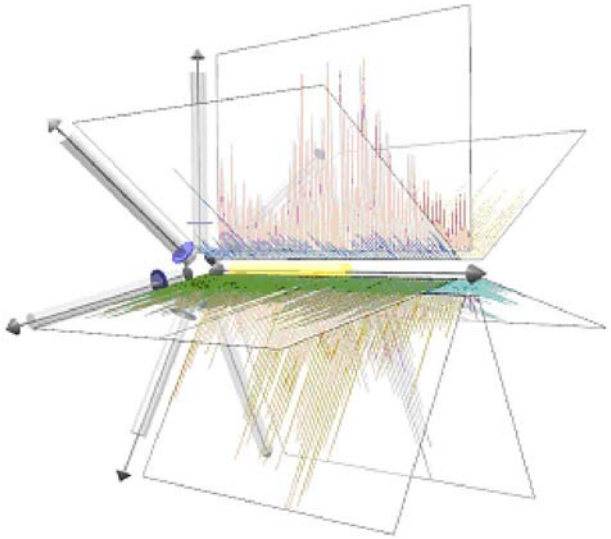

Figure 2.3: An axis-based 3D Multi comb (Taken from [4])

2.1 Chapter Summary

We can conclude that most of the existing approaches for extending 2D Kiviat diagrams to 3D diagrams are combinations of 2D Kiviat diagrams and Parallel Coordinates. Like the two diagrams shown in Figure 2.1 and Figure 2.2, they connect the points by polylines. The position and direction of a line is hard to recognize in a 3D space and illusions will be caused by perspective. So, it is difficult to recognize the change between the near by two Kiviat diagrams. In Figure 2.3, the state of each attribute is very good, but they are comparatively isolated from each other, and thus, we cannot have an overview of the whole entity.

3. Visualizations

As stated in Section 1.3, the objective of this work is to implement an interactive 3D Kiviat diagram for visualization of software metrics. The result is a program that can read a specified data file and visualize an appropriate and understandable view of the whole data set. This chapter indicates the idea of the visualization approach.

3.1 Data Source

The model of data source is a two-dimensional table as shown below.

Metric Name	value 1	value 2	……	value n
metric 1				
metric 2				
……				
metric m				

Table 3.1: The model of data source

As shown in Table 3.1, each row contains information of one single metric. Value *n* indicates the exact value of the metric at release *n*. The number of metrics *m* and the number of values for each metric *n* are both unrestricted. Therefore, we should have a data file, which contains a data model above, as the input file for our application.

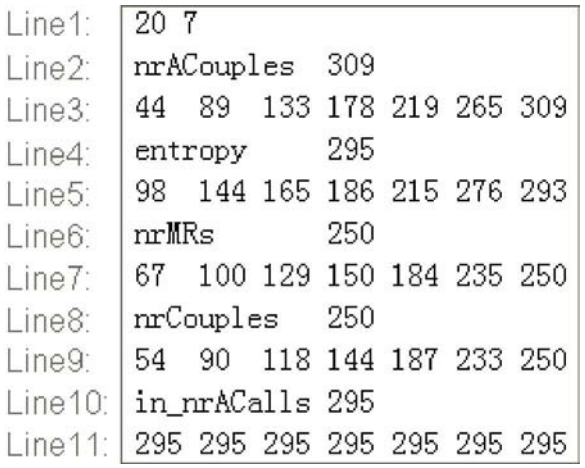

Figure 3.1: An example of the input file

Figure 3.1 shows the format of the input file. In the first line, the two integers 20 and 7 indicate the number of metrics and the number of values for each metric. Then, "nrACouples" is the metric name and "309" indicates the maximum value to show (this will be discussed in Section 3.2). After that, the remaining 7 numbers are the 7 values at 7 different releases of metric "nrACouples". From line 4 to line 5 is the information of another metric. For space saving, this picture only shows 5 metrics. Actually, there should be 20 metrics totally in the input file for our running example.

3.2 Initial View

The initial view of our program is a 2D Kiviat diagram which has been originally implemented in RelVis [2], see Figure 3.2.

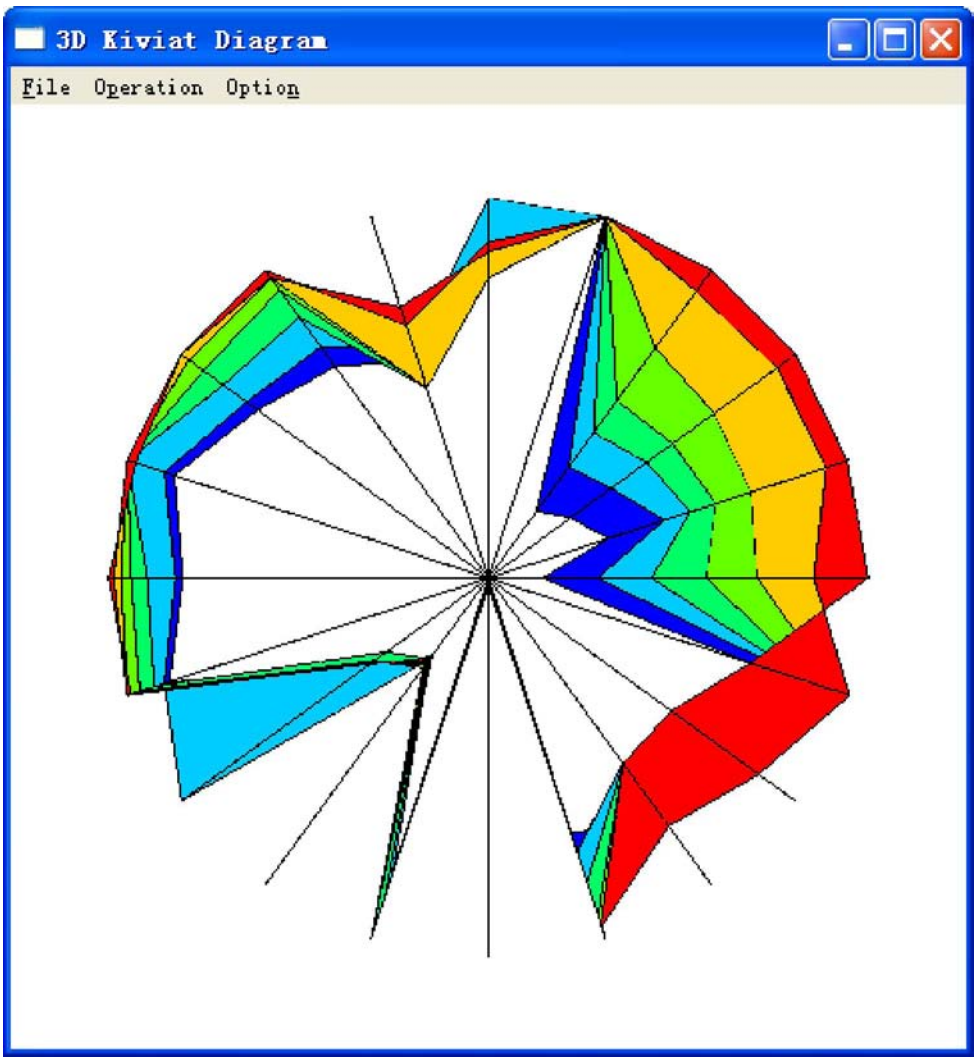

Figure 3.2: The initial view of the program

Twenty axes arranged like a star are mapped with twenty metrics separately. Rainbow color coding is used to indicate the change between two releases. Blue is for low releases and red is for high releases. The only difference is that the labels are hidden at the beginning. Labels can be shown if the user select corresponding axes (this will be discussed in Section 3.6).

From Figure 3.2, we also notice that the value on top of each axis may not be the maximum value among all values of the corresponding metric. This value can be indicated by the user in the input file discussed in Section 3.1. The user can set this value as an ideal value and see how far it is from ideal to reality. Additionally, we assume that the minimum value should always be 0 in any case.

3.3 Extend to 3D Kiviat Diagram

Considering in common 2D Parallel Coordinates and Star Glyphs, we found that if we rearrange the axes of a Parallel Coordinate, we will get a Star Plot as shown in Figure 3.3. While the two diagrams representing a same data set, the Star Glyph is comparatively space saving if the number of axes becomes huge.

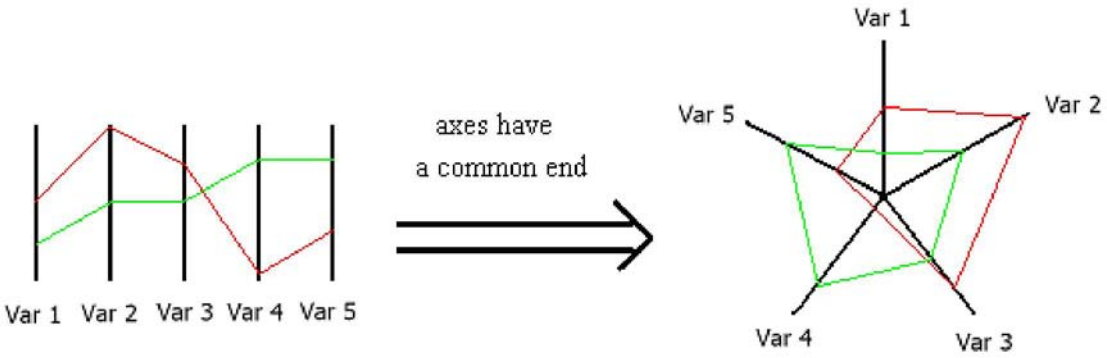

Figure 3.3: From Parallel Coordinate to Star Glyph

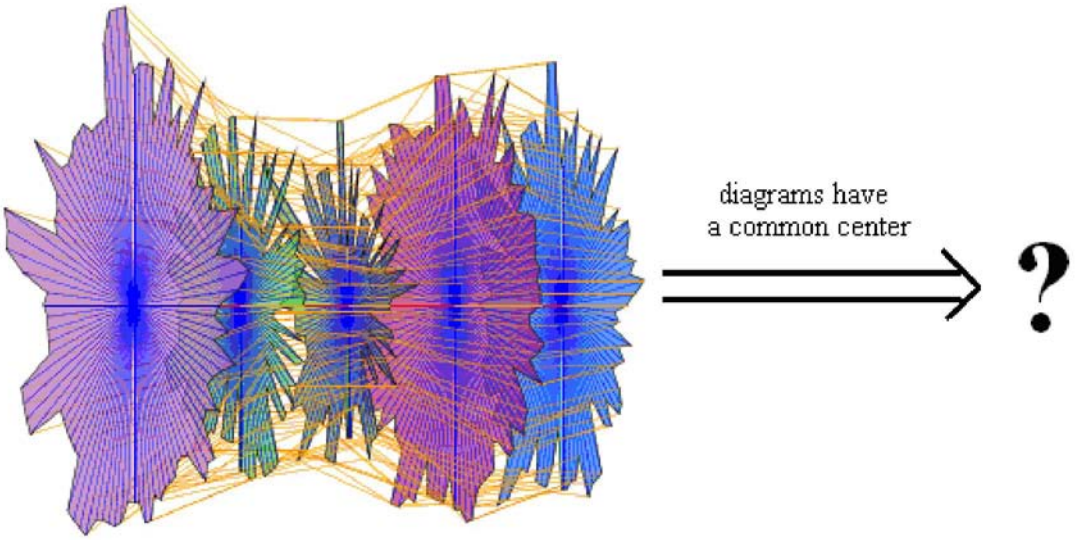

Figure 3.4: From 3D Parallel Coordinate to 3D Kiviat diagram

Similarly, if we do the same thing with a 3D Parallel Coordinate [3], i.e., if we rearrange each single 2D Kiviat diagram and make them have a common center, we will hopefully get a space saving 3D Kiviat diagram.

In that case, we found that for extending a 2D Kiviat diagram to a 3D Kiviat diagram, the combination of 2D Kiviat diagrams and Parallel Coordinates, as discussed in Section 2.1, is not the only approach. We can also combine 2D Kiviat diagrams with 2D Kiviat diagrams. Each axis of a 2D Kiviat diagrams can be consist of several sub-axes. These sub-axes should not be arranged like a star, because otherwise, they will interfere with each other and hard to be recognized. But these sub-axes can be arranged like a sector. Thus, we found that we can use axes unfolding to show a 3D Kiviat diagram.

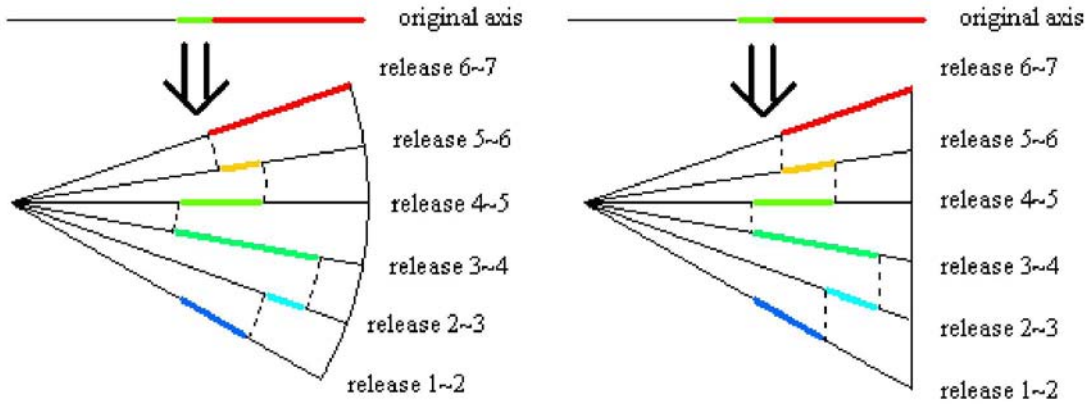

Figure 3.5: Two approaches for axis unfolding

There are two approaches for unfolding an axis. As shown in Figure 3.5, the axis is unfolded to a sector and to a triangle respectively. By doing this, the overlapped information is distributed to different sub-axis, so that the user will be able to see the overlapped detail by rotating the diagram. The two approaches are very similar in practice. Additionally, the dashed lines just indicate the same levels between adjacent sub-axes.

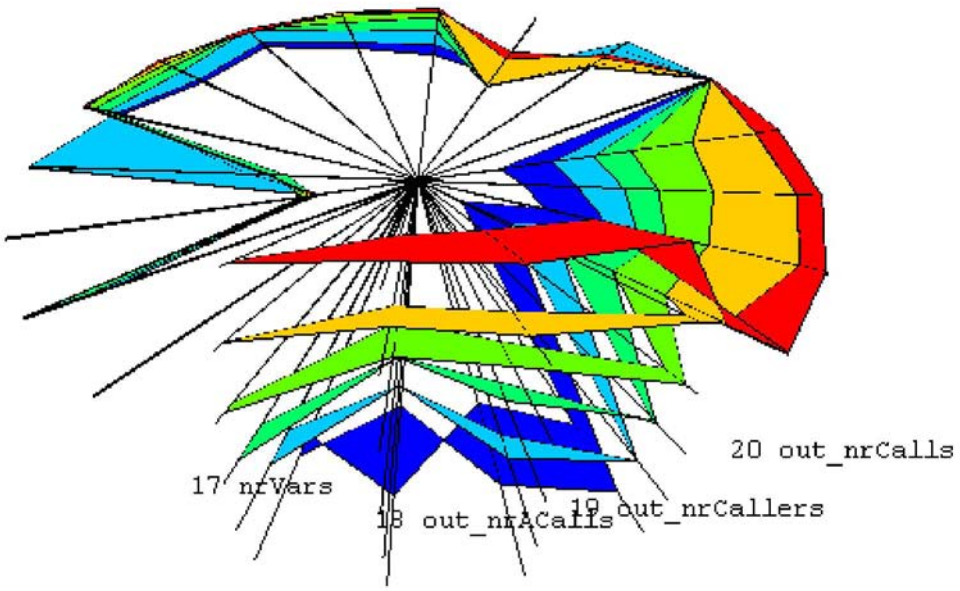

Figure 3.6: A 3D Kiviat diagram view in our application. Four axes have been unfolded to sectors.

As shown in Figure 3.6, the overlapped information from axis 17 to axis 20 in Figure 1.2 now becomes visible by unfolding the axes and rotating the diagram. Originally, the diagram shown in Figure 3.6 is a 2D Kiviat diagram. We rotate the whole diagram and unfold axis 17, 18, 19 and 20, thus we have a simple 3D Kiviat diagram.

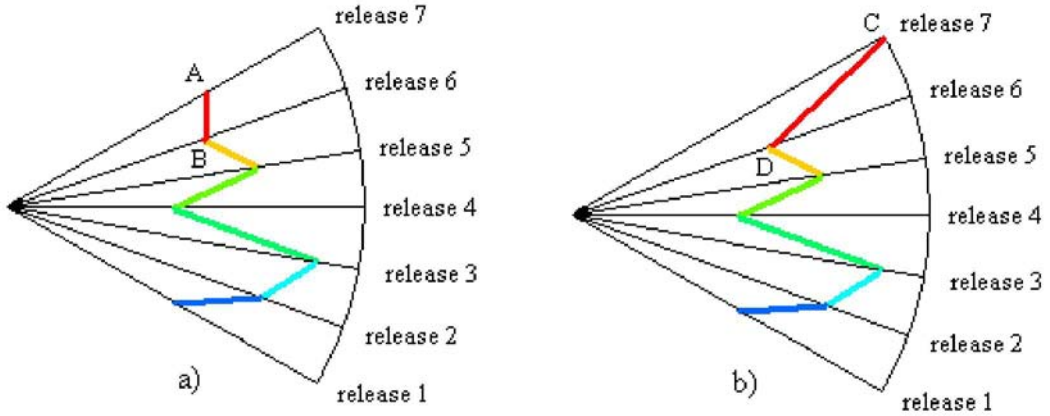

Figure 3.7: Another unadopted approach for axis unfolding.

There is another approach for unfolding axis, as shown in Figure 3.7. But this approach was not adopted. In Figure 1.2 and Figure 3.6, we can see that the adjacent points are connected to each other and form a colored block. If two unfolded axes are adjacent to each other, in most cases, the connection between them cannot be a single block. For example, in Figure 3.7, if Sector a) and Sector b) are adjacent sectors, point A, B, C,

and D are obviously not in the same plane. In such cases, the connection between them becomes complex. In order to make things simple, we decide to adopt unfolding approaches shown in Figure 3.5.

3.4 Color Coding

As stated in Section 1.1, it is hard to recognize the direction of the change from release *n* to release *n+1*. For example, in Figure 3.6, although we can see the overlapped blue blocks, we still don not understand whether the value of metric 18 was increased or decreased between release 1 and release 2 (blue). The origin of this problem is that we have to recognize the direction with the help of adjacent colored blocks.

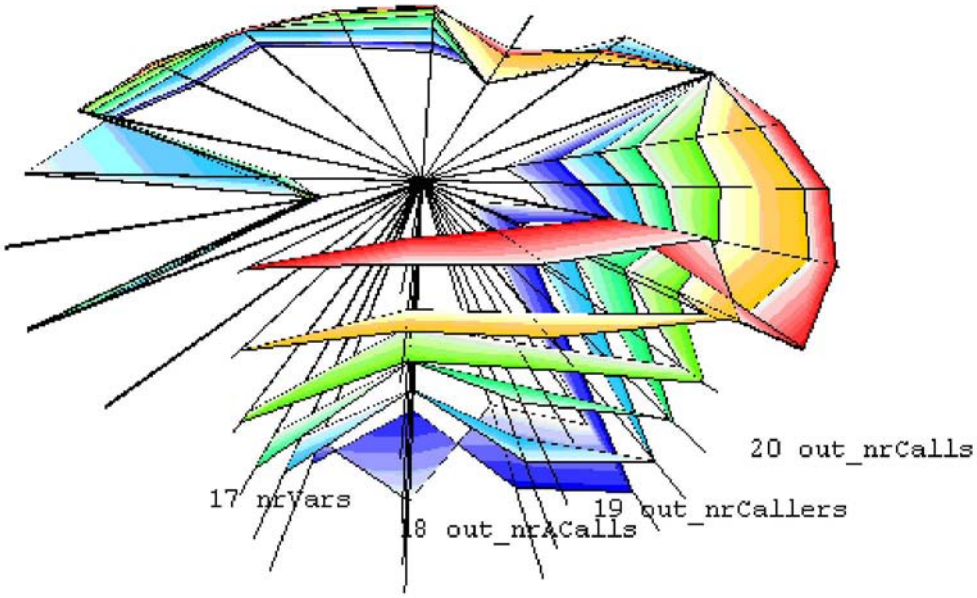

Figure 3.8: Color coded 3D Kiviat diagram view

If we can perceive the direction without the help the adjacent colored blocks, the direction problem would be solved. The gradual change of color is a good approach for direction indication. Since we have used rainbow color to distinguish different releases, we could use gradual change of brightness to indicate the direction. From a low release to the following high release, we change the color from white or gray to the corresponding rainbow color. As shown in Figure 3.8, we can recognize that the value of metric 18 was decreased from release 1 to release 2.

3.5 Focus on Specific Metrics and/or Releases

By implementing the approaches we discussed above, the user should be able to have a rough impression on the software metrics visualized by the 3D Kiviat diagram. But sometimes the user may just want to focus on some specific metrics. In such cases, the diagram shown in Figure 3.5 will be very helpful. However, simply rotate the whole diagram can not get the side views because the axes are overlapped by other axes and colored blocks.

In order to reduce clutter and display the information of several specific metrics in detail, we can let some part we don not care be transparent. As shown in Figure 3.9, except for axis 18, the rest part of the diagram has faded out. The value of metric 18 was decreased from release 1 to release 2 and almost stable through release 2, 3 and 4. It

was increased in release 5 and 6, but still decreased back in release 7. The release labels are not shown here as in Figure 3.5 because the rainbow color has indicated the release number.

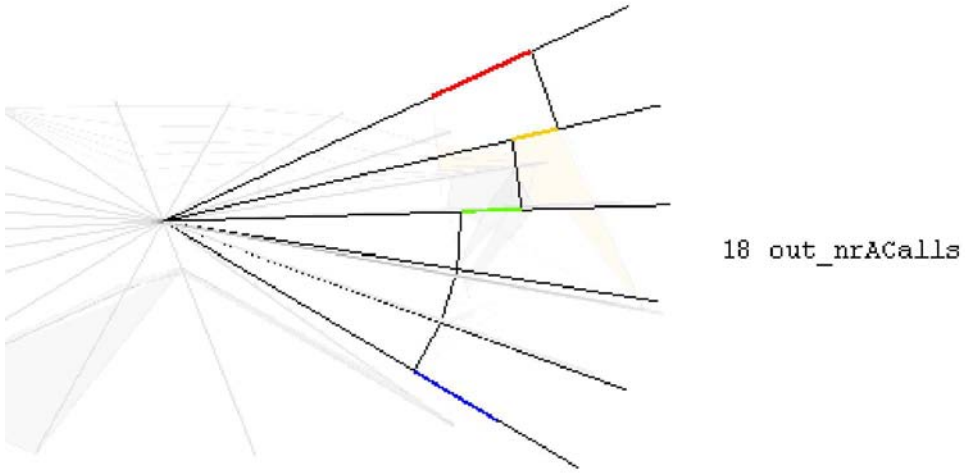

Figure 3.9: A 3D Kiviat diagram view focused on metric 18

Similarly, the user also may want to focus on some specific releases. As shown in Figure 3.10, the user is focusing on 14 metrics and 3 releases (release 1, 2 and 3). We can also have a good overview of some specific releases by focusing on all axes and only two or three releases.

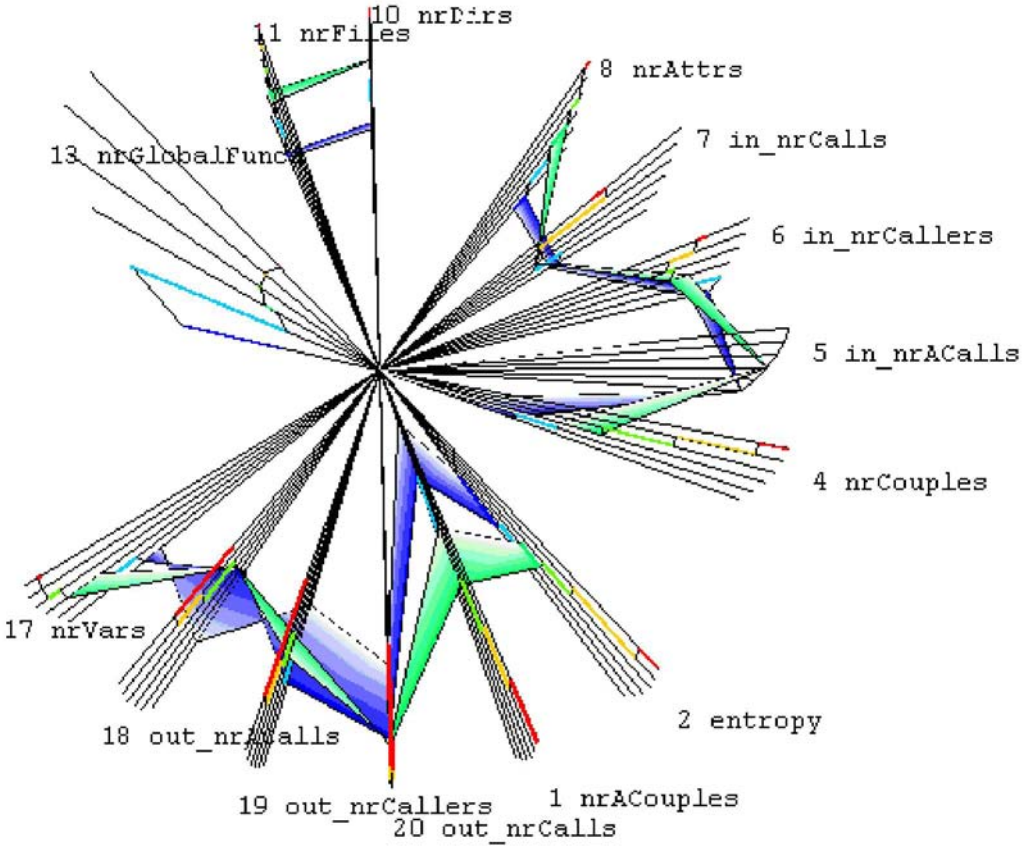

Figure 3.10: A 3D Kiviat diagram view focuses on several metrics and releases.

3.6 User Interface

Interaction between user and the program is very important in this thesis because almost every operation is used frequently. A friendly and comfort user interface can improve the usability.

- **Diagram Rotation**
  A user can rotate the diagram by dragging the background or press direction button on keyboard.
- **Axes Selection**
  A user can select/deselect axes by left button click directly on the diagram. Labels of selected axes will be displayed. Considering the number of axes may be huge, Ctrl + A can select all axes and relatively, Ctrl + I can make a negative selection. Shift + left button click can make a sequence selection.
- **Color Coding Switch**
  A user can press Space to switch on/off color coding, as stated in Section 3.3.
- **Unfold Mode Switch**
  A user can press Tab to switch the two different modes for axis unfolding, as mentioned in Section 3.2.
- **Axis Unfolding**
  A user can unfold selected axes simply by rotating the wheel on the mouse or press Page Up/Page Down button on keyboard.
- **Transparence Adjustment**
  A user can adjust transparence of the unselected part of diagram by pressing the Ctrl button and with the same time, rotating the wheel on the mouse.
- **Release Selection**
  A user can select releases by pressing the Ctrl button and with the same time, left button click the corresponding colored blocks on the diagram. Sequence selection can also be supported by pressing Shift button simultaneously.

Additionally, the user can open a data file and configure the system by dialog boxes.

3.7 Conclusion

In this chapter, we discussed a possible approach for Software Visualization. This approach improves the existing 2D approach and has solved two problems: overlapping and direction problem. Furthermore, comparing this approach to other approaches discussed in Chapter 2, it can display the detail as well as give a better overview. The color coding emphasize the change between two releases and also point out the trends of future development. Additionally, it is comparatively space saving.

4. Implementation

Based on the discussions about the data source in Section 3.1 and possible visualization in Chapter 3, it is clear that the final software solution is mainly made of three components: data source, axes layout calculation, and the core component which controls the previous two components and responds to user interaction. This chapter will explain the data structure as well as internal algorithms of the three components and how the components are connected.

4.1 Component Diagram

As shown in Figure 4.1, *Data Source* loads a data file and the data is constant during the whole runtime because this program is a pure visualization tool and does not support data modification. *Axes Layout Calculation* provides the variable data during interaction with the diagram. Finally, the *Core* component uses both the constant data as well as the variable data, and provides layout functions to display the diagram on the screen.

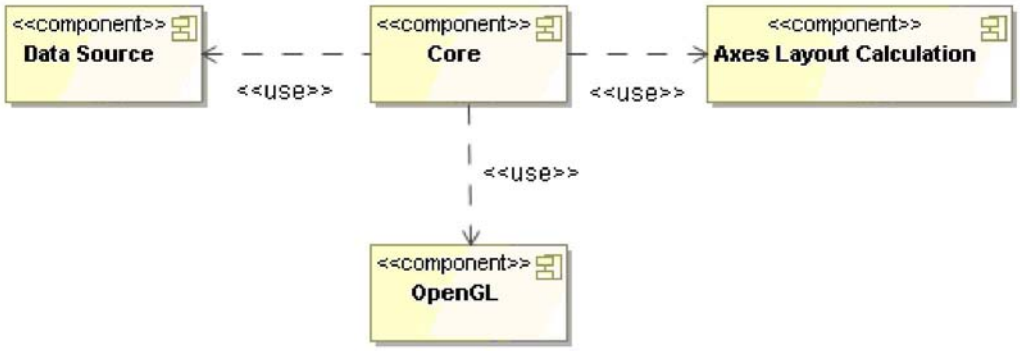

Figure 4.1: Component diagram of the visualization tool

Furthermore, the visualization application is written in C++ because the student in charge is familiar with C++ programming. One requirement of the final implementation is to make this approach be appropriate for all kinds of data, so the size of data can be changed (scalability).

4.2 Class Diagrams and Data Structure

The three components as shown in Figure 4.1 are implemented with three corresponding classes as shown in Figure 4.2.

Figure 4.2: The relationship between three main classes

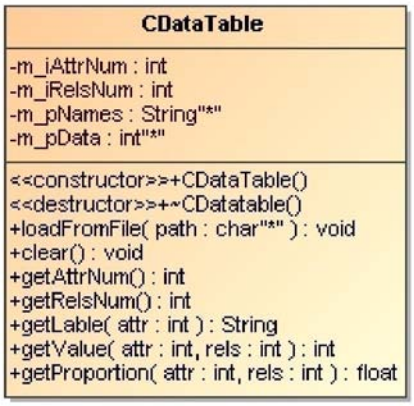

Figure 4.3: Class diagram of CDataTable

As shown in Figure 4.3, this class uses the method *loadDataFile()* to open the file and load all constant data in a dynamic array *m_pData*. Other classes can get information by the five "get" methods. The method *getProportion()* will return a float number which equals the specific value on an axis divided by the top value on the axis. This value can help us to locate the value on the axis exactly.

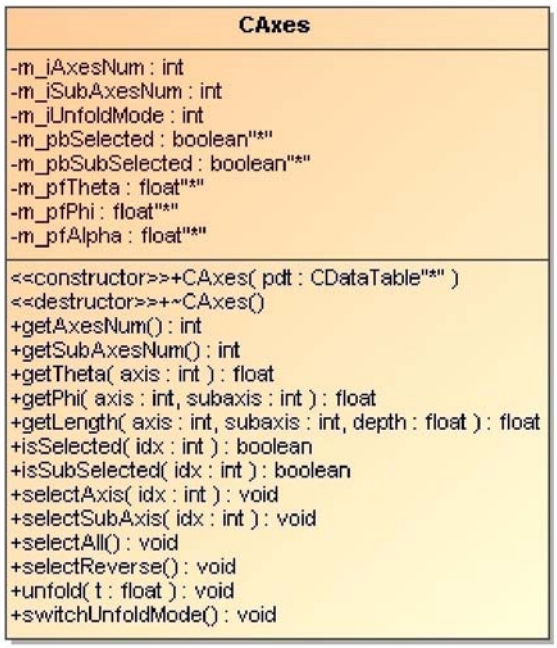

Figure 4.4: Class diagram of CAxes

Class *CAxes* is used to store the dynamic data of the interactive axes. As shown in Figure 4.4, the most frequently used method is *getTheta()*, *getPhi()*, and *getLength()*, by which we can get the exactly position of any unfolded sub-axes.

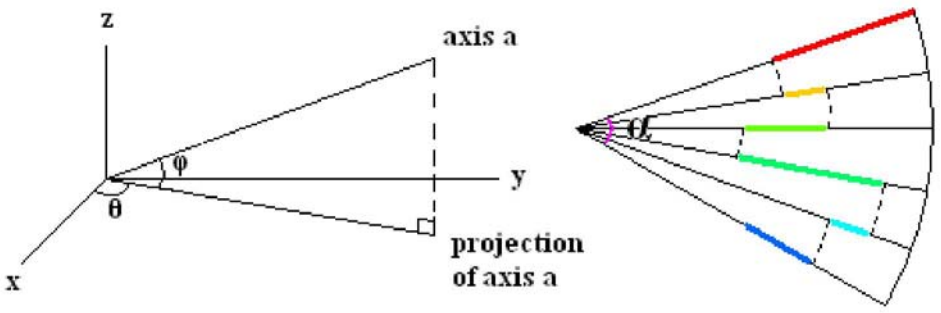

Figure 4.5: θ, φ and α in the 3D coordinate

In the 3D coordinate system as shown in the first part of Figure 4.5, we use θ (the angle between x axis and the projection of axis a), φ (the angle between x-y plane and axis a) and length (the distance between origin point and the end of axis a) to represent the original axes start with origin point in the 3D space. Additionally, we use α to indicate how big the axis is opened as shown in the second part of Figure 4.5.

The unfolded sub-axes and the corresponding original axes have the same θ value, and it can be calculated as:

$$\theta(i) = 2\pi \frac{i}{n}$$

While *i* is the index of axis and n is the total number of axes.

Although the φ values of the original axes are always 0 in this application (the original axes are always on the x-y plane), we still take this value into consideration for improving the extensibility of this application. The φ value of sub-axes can be calculated as:

$$\varphi(i, j) = \frac{j}{m - 1} \alpha(i) - \frac{\alpha(i)}{2} + \varphi(i)$$

While *i* is the index of original axis, j is the index of sub-axis, m is the total number of sub-axes and φ(i) is the φ value of the original axis. It should be pay attention that if m = 1, the denominator will be 0. So, we should check whether m is bigger than 1. If m = 1, φ(i, j) =φ(i);

Finally, the length values of original axes are stable in this application. But the length values of sub-axes depend on the unfolding mode. If we unfold the original axes into a sector, they will have the same length value because the length equals the radius of the sector. On the other hand, if we unfold the original axes into a triangle, the length value of sub-axes can be calculated as:

$$Length(i, j) = \frac{Length(i)}{\cos(\varphi(i, j) - \varphi(i))}$$

While *i* is the index of original axis, *j* is the index of sub-axis and Length(i) is the length value of the original axis. Moreover, the “select” methods and *unfold()* from the *CAxes* class control the interaction of axes.

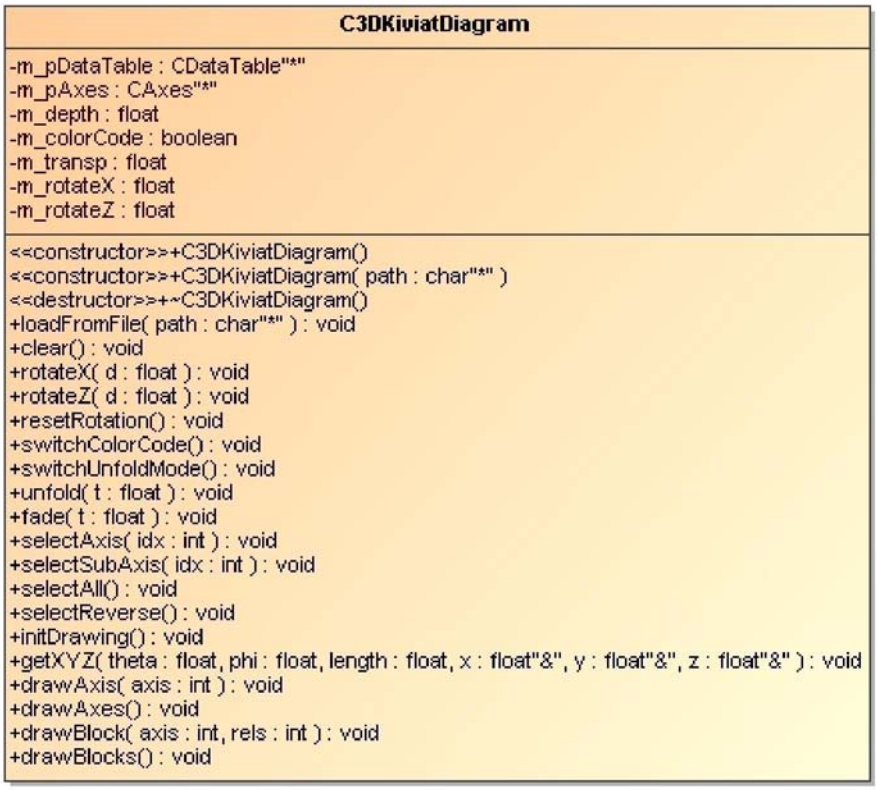

Figure 4.6: Class diagram of C3DKiviatDiagram

The *C3DKiviatDiagram* is a control class, which encapsulates the other two entity classes. It is directly used by the main program. This class provides services such as rotation, selection, unfolding, fading, and, most important, the "draw" methods for representing the diagram on the screen.

4.3 Axes Visualization

As we discussed in Section 4.2, we can get the exact position of any axes by the methods provided by *CAxes*, but we still need to convert them to an (x, y, z) value, because we are using a Cartesian coordinate system. This method is provided by *C3DKiviatDiagram*. We can easily convert (θ, φ, length) to (x, y, z) by the formulas below.

$$x = \cos(\theta)\cos(\varphi)\,\text{length}$$
$$y = \sin(\theta)\cos(\varphi)\,\text{length}$$
$$z = \sin(\varphi)\,\text{length}$$

Furthermore, when the user focuses on some specific metrics as discussed in Section 3.4, the unselected axes should fade out and the selected axes should show additional lines indicating the same value between two adjacent sub-axes, just like the dashed lines shown in Figure 3.5. This method is implemented in *C3DKiviatDiagram::drawAxis()*.

4.4 Colored Blocks Visualization

Connecting adjacent dots on the axes and rendering a colored quad can communicate the trend how metrics changed and make the diagram more readable. The color coding function has been implemented as:

```
bool colorCoding(float percent, float& r, float& g, float& b)
{
    if (percent >= 0.0f && percent <= 0.25f)
    {
        percent *= 4.0f;
        r = 0.0f;
        g = percent;
        b = 1.0f;
        return true;
    }
    else if (percent > 0.25f && percent <= 0.5f)
    {
        percent = (percent - 0.25f) * 4.0f;
        r = 0.0f;
        g = 1.0f;
        b = 1.0f - percent;
        return true;
    }
    else if (percent > 0.5f && percent <= 0.75f)
    {
        percent = (percent - 0.5f) * 4.0f;
        r = percent;
        g = 1.0f;
        b = 0.0f;
        return true;
    }
    else if (percent > 0.75f && percent <= 1.0f)
    {
        percent = (percent - 0.75f) * 4.0f;
        r = 1.0f;
        g = 1.0f - percent;
        b = 0.0f;
        return true;
    }
    else
    {
        r = g = b = 1.0f;
        return false;
    }
}
```

With the percent change from 0.0 to 1.0, we get a rainbow color from blue to red. The percent parameter should equal the sub-axis index divided by the total number of sub-axes.

Secondly, the (x, y, z) position of the dots on axes should be calculated. Since the dots are on the axes, they will have the same θ value as well as ϕ value, and length value is the only difference. The length value here equals the length value of the corresponding axis multiplied by a proportion number, which is mentioned in Section 4.2. Then we can convert the coordinate in the same way discussed in Section 4.3 and draw a quad on the screen.

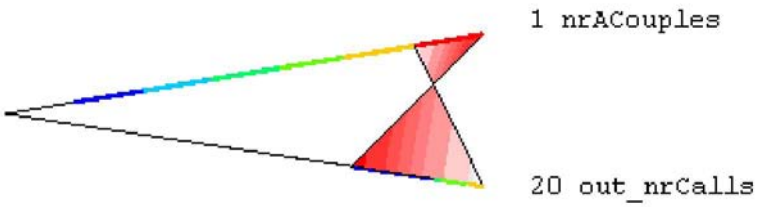

Figure 4.7: A special colored block

There may be a special situation as shown in Figure 4.7: metric 1 increased but metric 20 decreased. In such cases, we should render two triangles instead of one quad. The key is to calculate the position of the intersection point. We can project the two lines to x-y, x-z and y-z plane and then calculate three intersection points respectively. Finally,

we can get the final intersection by considering the three results. Moreover, the color of the intersection point should be a blending of rainbow color and the common color.

Finally, when the colored blocks fade out, we should render a colored cylinder on the axis, so that we can focus on details of several individual metrics as shown in Figure 3.5 above. This method is implemented in *C3DKiviatDiagram::drawBlock()*.

4.5 Axes Selection and Releases Selection

As some important operations, such as unfolding and fading, are based on object selection, the most convenient and comfort operation to select an object is to use mouse click. The selection mode supported by OpenGL can help us to achieve this efficiently.

In selection mode, nothing is drawn to the screen. Instead, information about objects rendered while in selection mode will be stored in the selection buffer. Firstly, we prepare a name stack, and then we select the projection matrix so that we can restrict drawing to the area just under the cursor. After that we switch to the model-view matrix and draw the axes or blocks by calling *drawAxes()* or *drawBlocks()*. We assign different IDs for different logical objects. For example, although we have six sub-axes when the original one is unfolded as shown in Figure 3.5, the six sub-axes are representing a same metric, and on logical they are the same object and they should have a same ID. So, we come to the result that if we click on a single sub-axis, we will select a group of axes. Releases selection is just the same. After drawing the objects, we switch back to the projection matrix and pop the stored matrix off the stack, and switch back to the model-view matrix. Finally, we switch from selection mode to render mode so that objects are displayed to the screen from the buffer. Then, we get the result about how many objects we have hit in the small area we restricted before.

4.6 Final Remarks

Finally, the three components, data source, axes layout calculation and presentation, are connected to each other over a core implementation. Altogether, our software solution improves the existing 2D Kiviat diagrams, which have been developed in RelVis [2]. To finish this chapter, the view on source code and release history data of up to *n* releases is explained as follows:

Before running the application, the user must prepare a data file in the way explained in Section 4.2. Right after the user has started the application a data file can be chosen over the *File* menu. Having done this, the *C3DKiviatDiagram* instance creates a *CDataTable* instance and passes the path to it. Then the file is opened and all the data is stored into the *CDataTable* instance and returned to *C3DKiviatDiagram*. After that, a *CAxes* instance is created by *C3DKiviatDiagram* and all the parameters are initialized and preparing for display.

Every time a *WM_PAINT* message is queued to the message queue, the *initDrawing()* method, *drawAxes()* method and *drawBlocks()* method will be called. These methods retrieve necessary information from the *CDataTable* instance and *CAxes* instance and use their own algorithms to show the diagram on the screen.

Other occurrences of messages, such as *WM_LBUTTONDOWN, WM_ MOUSEMOVE, WM_KEYDOWN* and *WM_MOUSEWHEEL*, will cause the execution of corresponding message handlers. These handlers will call other methods provided by *C3DKiviatDiagram*, and thus, the parameters of the diagram are changed. Finally, the visualization will be changed responding to the user interaction.

The interaction continues until the user terminates the application. Then, the destructor of each instance is called layer by layer and no memory leak will occur.

5. Conclusions and Future Work

This chapter makes a conclusion over the whole thesis and discusses some future work should be done after this thesis.

5.1 Conclusion

The goal of the thesis was:

Implement an interactive 3D Kiviat diagram for visualization of integrated condensed graphical views on source code and release history data of up to n releases by improving an existing 2D approach and solve the overlapping and direction problem.

At first, the solution was described and defined by criteria and restrictions in Chapter 1. The final visualization helps a software developer to have a comprehensive view on software metrics. The user can get information through the diagram about the history and current state as well as the future trend of the software. The user can also focus on some specific details and have an overview through interaction with the application.

At the beginning of this report, the advantages and disadvantages of the existing 2D approach were analyzed and two main problems, the overlapping and the direction problem, were revealed. Several existing 3D visualization approaches are investigated and some inspiration was picked up from them. After this, the feasibility of several approaches was discussed and the final approach was confirmed.

At the beginning of the implementation part, a 2D Kiviat diagram was firstly implemented. Color coding and diagram rotation as two new features were added to the application. Then axes selection and unfolding was made available by using keyboard buttons. Several new possibilities of interaction were discussed in this thesis.

In order to improve the extensibility and maintainability of the application, the source code was rearranged. The origin entity class was divided to two entity classes and store stable and changeable data separately. A better user interface was implemented. Some new features, such as fading and sub-axis selection, were introduced. With that, the criterion of being interactive is fulfilled.

5.2 Future Work

The application can only accept one kind of data file as stated in Section 3.1. Even a little mistake, for example, adding some unnecessary characters, may cause big problems in the final representation. It would be much better if the application can access database or can retrieve data from xls-files. It would also be nice if this application can cooperate with other applications which can measure software metrics and provide the data set.

Furthermore, the arrangement of labels is clumsy as shown in Figure 5.1. The labels are overlapped and in a mess. One solution is to add a little offset to the labels being plotted. We can also rotate each label and keep it being parallel with the corresponding axis. Another solution is that only the index numbers are shown beside the axes and the metric names are listed in a table in another view.

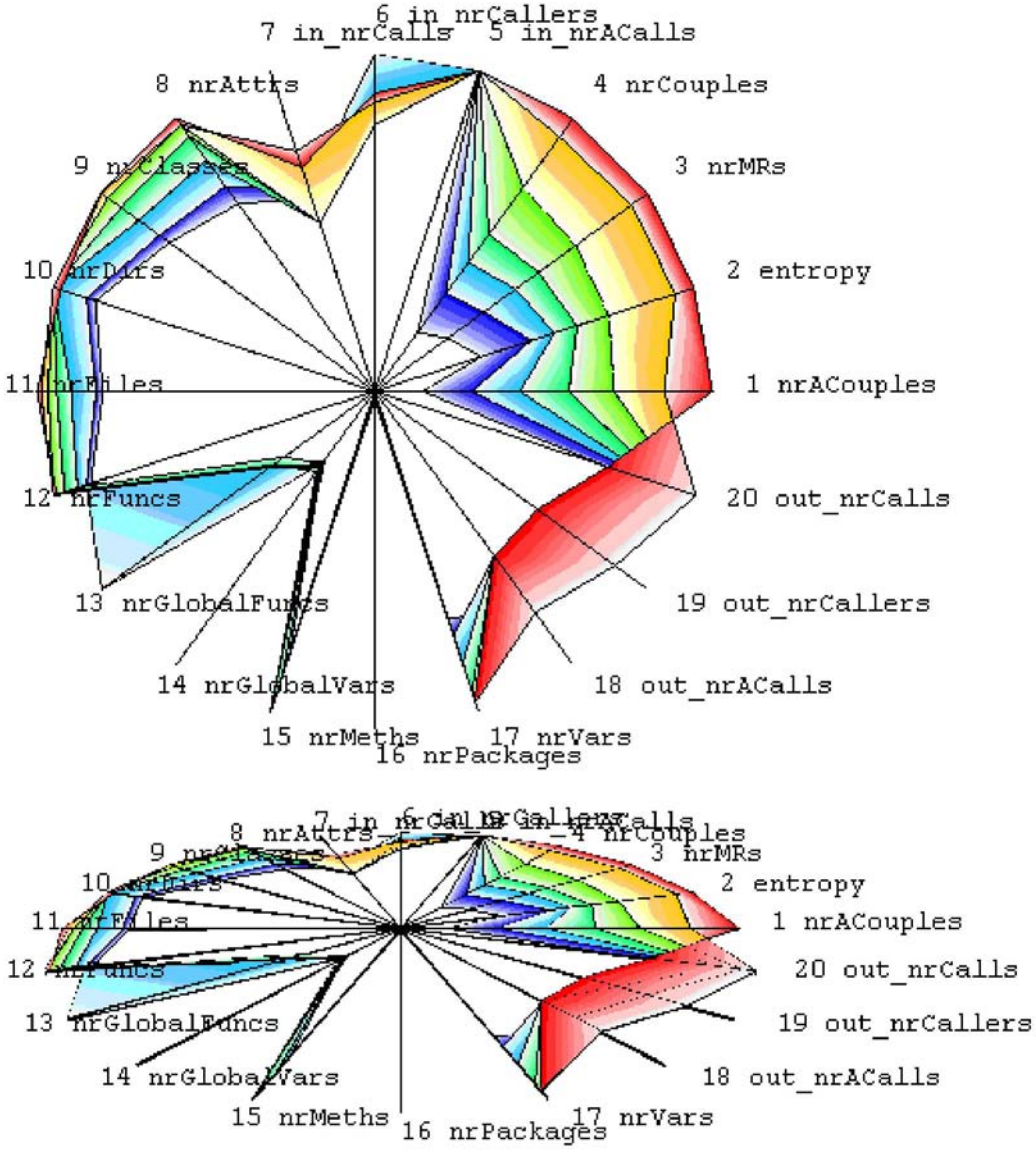

Figure 5.1: A jumbled arrangement of labels

Furthermore, it is also impossible for us to see the exact value of a metric on the diagram. It can be solved if we use the following approach: when the cursor stops on the intersection, the value will be shown beside the cursor.

References

[1] A. Kerren, 2008,
Lecture on Information Visualization / , DA4035, Växjö University, Sweden

[2] M. Pinzger, H. Gall, M. Fischer, and M. Lanza, 2005,
Visualizing Multiple Evolution Metrics
In Proceedings of the 2005 ACM symposium on Software Visualization
Visualization of the software development process, Pages: 67 - 75

[3] E. Fanea, S. Carpendale, and T. Isenberg, 2005,
An Interactive 3D Integration of Parallel Coordinates and Star Glyphs
In Proceedings of the Proceedings of the 2005 IEEE Symposium on Information Visualization

[4] C. Tominski, J. Abello, and H. Schumann, 2005,
Interactive Poster: 3D Axes-Based Visualizations for Time Series Data
In Poster Compendium of IEEE Symposium on Information Visualization (InfoVis'05), Minneapolis, USA, 2005

[5] A. Inselberg and B. Dimsdale, 1990,
Parallel Coordinates: A Tool for Visualizing Multi-Dimensional Geometry
In Proceeding of IEEE Visualization 1990 (VIS 1990), pages 361-378, Los Alamitos, CA, 1990. IEEE Computer Society Press

[6] A. Inselberg, 1985,
The Plane with Parallel Coordinates
The Visual Computer, 1(2):69-91, October 1985

[7] W. Ribarsky, E. Ayers, J. Eble, and S. Mukherjea, 1994,
Glyphmaker: Creating Customized Visualizations of Complex Data
Computer, 27(7):57-64, July 1994

[8] C. D. Shaw, J. A. Hall, C. Blahut, D. S. Ebert, and D. A. Roberts, 1999,
Using Shape to Visualize Multivariate Data
In Proceedings of the 1999 Workshop on New Paradigms in Information Visualization and Manipulation, pages 17-20, New York, 1999. ACM Press

[9] M. O. Ward, 1994,
Xmdv Tool: Integration Multiple Methods for Visualizing Multivariate Data
In Proceedings of IEEE Visualization 1994 (VIS 1994), pages 326-333, Los Alamitos, CA, 1994. IEEE Computer Society Press

[10] Y. H. Fua, M. O. Ward, and E. A. Rundensteiner, 1999,
Hierarchical Parallel Coordinates for Exploration of Large Datasets
In Proceedings of IEEE Visualization 1999 (VIS 1999), pages 43-50, Los Alamitos, CA, 1999. IEEE Computer Society Press

[11] A. R. Martin and M. O. Ward, 1995,
High Dimensional Brushing for Interactive Exploration of Multivariate Data

In Proceedings of IEEE Visualization 1995 (VIS 1995), pages 271-278, Los Alamitos, CA, 1995. IEEE Computer Society Press

[12] W. Peng, M. O. Ward, and E. A. Rundensteiner, 2004,
Clutter Reduction in Multi-Dimensional Data Visualization Using Dimension Reordering
In Proceedings of the IEEE Symposium on Information Visualization 2004. IEEE Computer Society Press

[13] M. O. Ward, 1997,
Creating and Manipulating N-Dimensional Brushes
In Proceedings of Joint Statistical Meeting 1997, pages 6-14

[14] J. Yang, W. Peng, M. O. Ward, and E. A. Rundensteiner, 2003,
Interactive Hierarchical Dimension Ordering, Spacing and Filtering for Exploration of High Dimensional Datasets
In Proceedings of the IEEE Symposium on Information Visualization 2003 (InfoVis 2003), pages 105-112, Los Alamitos, CA, 2003. IEEE Computer Society Press

[15] S. T. Hackstadt and A. D. Malony, 1995,
Visualizing Parallel Programs and Performance
IEEE Computer Graphics and Applications, 15(4):12-14, July 1995

[16] T. DeMarco, 1986,
Controlling Software Projects: Management, Measurement and Estimation
Prentice Hall PTR, Upper Saddle River, NJ

[17] S. Diehl, 2007,
Software Visualization
LNCS 2269, Springer-Verlag New York, Inc.

Appendix: Source Code of the Most Important Classes

```
class CDataTable
{
    // Attributes:
private:
    int     m_iAttrNum;
    int     m_iRelsNum;

    string* m_pLables;
    int*    m_pData;
    bool*   m_pbOverlap;

    // Constructors:
public:
    CDataTable();
    ~CDataTable();

    // Operations:
public:
    void loadDataFile(wchar_t *path);
    void clear();
    int getAttrNum();
    int getRelsNum();
    string getLable(int attr);
    int getValue(int attr, int rels);
    float getProportion(int attr, int rels);
};


void CDataTable::loadDataFile(wchar_t *path)
{
    if (!(m_iAttrNum == 0 && m_iRelsNum == 0))
    {
        clear();
    }
    ifstream input(path, ios::in);
    input >> m_iAttrNum >> m_iRelsNum;
    m_pLables = new string[m_iAttrNum];
    m_pData = new int[m_iAttrNum * (m_iRelsNum + 1)];
    m_pbOverlap = new bool[m_iAttrNum];

    for (int i = 0; i < m_iAttrNum; i++)
    {
        input >> m_pLables[i] >> m_pData[i * (m_iRelsNum + 1)];
        for (int j = 1; j <= m_iRelsNum; j++)
        {
            input >> m_pData[i * (m_iRelsNum + 1) + j];
        }
    }
    for (int i = 0; i < m_iAttrNum; i++)
    {
        int dir = 0;
        m_pbOverlap[i] = false;
        for (int j = 1; j < m_iRelsNum; j++)
        {
            int x = getValue(i, j + 1) - getValue(i, j);
            if (dir == 0)
            {
                dir = x;
                continue;
            }
            if (dir * x < 0)
            {
                m_pbOverlap[i] = true;
            }
        }
    }
}
```

```
class CAxes
{
    // Attributes:
public:
    static const int TOSECTOR   = 0;
    static const int TOTRIANGLE = 1;
private:
    int     m_iAxesNum;
    int     m_iSubAxesNum;
    int     m_iUnfoldMode;
    bool*   m_pbSelected;
    bool*   m_pbSubSelected;
    float*  m_pfTheta;  // the angles between X axis and each logical axes
    float*  m_pfPhi;    // the angles between each logical axes and X-Y plane
    float*  m_pfAlpha;  // the angles how each logical will be unfolded
    // Constructors:
public:
    CAxes(CDataTable* pdt);
    ~CAxes();
    // Operations:
public:
    int getAxesNum();
    int getSubAxesNum();
    float getTheta(int axis);
    float getPhi(int axis, int subaxis);
    float getLength(int axis, int subaxis, float depth);
    bool isSelected(int idx);
    bool isSubSelected(int idx);
    void selectAxis(int idx);
    void selectSubAxis(int idx);
    void selectAll();
    void selectReverse();
    void unfold(float t);
    void switchUnfoldMode();
};


float CAxes::getTheta(int axis)
{
    return m_pfTheta[axis];
}

float CAxes::getPhi(int axis, int subaxis)
{
    float phi;
    if (m_iSubAxesNum > 1)
        phi = (float)subaxis / (float)(m_iSubAxesNum - 1) *
              m_pfAlpha[axis] - m_pfAlpha[axis] / 2.0f + m_pfPhi[axis];
    else
        phi = m_pfPhi[axis];
    return phi;
}

float CAxes::getLength(int axis, int subaxis, float depth)
{
    float length, phi;
    if (m_iSubAxesNum > 1)
    {
        if (m_iUnfoldMode == TOSECTOR)
            length = (float)depth;
        else if (m_iUnfoldMode == TOTRIANGLE)
        {
            phi = getPhi(axis, subaxis);
            length = depth / cos(phi - m_pfPhi[axis]);
        }
    }
    else
        length = (float)depth;
    return length;
}
```

```
class C3DKiviatDiagram
{
    // Attributes:
private:
    CDataTable* m_pDataTable;
    CAxes*      m_pAxes;
    CConfig     m_config;
    float       m_depth;
    bool        m_colorCode;
    float       m_transp;
    GLfloat     m_rotateX;
    GLfloat     m_rotateZ;

    // Constructors:
public:
    C3DKiviatDiagram();
    C3DKiviatDiagram(wchar_t* path);
    ~C3DKiviatDiagram();

    // Operations:
public:
    void loadDataFile(wchar_t* path);
    void clear();
    void rotateX(float d);
    void rotateZ(float d);
    void resetRotation();
    void switchColorCode();
    void switchUnfoldMode();
    void unfold(float t);
    void fade(float t);
    void selectAxis(int idx);
    void selectSubAxis(int idx);
    void selectAll();
    void selectReverse();
    void initDrawing();
    void getXYZ(float theta, float phi, float length, float &x, float &y, float &z);
    void drawAxis(int axis);
    void drawAxes();
    void drawBlock(int attr, int rels);
    void drawBlocks();
    void drawLable(int attr);
};


void C3DKiviatDiagram::initDrawing()
{
    glLoadIdentity();
    glTranslatef(0.0f, 0.0f, -3.0f * m_depth);
    glRotatef(m_rotateX, 1.0f, 0.0f, 0.0f);
    glRotatef(m_rotateZ, 0.0f, 0.0f, 1.0f);
}

void C3DKiviatDiagram::getXYZ(float theta, float phi, float length,
                              float &x, float &y, float &z)
{
    x = cos(theta) * cos(phi) * length;
    y = sin(theta) * cos(phi) * length;
    z = sin(phi) * length;
}
```

```
void C3DKiviatDiagram::drawAxis(int axis)
{
    float x, y, z, phi, theta, length;
    glColor4f(m_config.axr, m_config.axg, m_config.axb, m_transp);
    if (m_pAxes->isSelected(axis))
    {
        glColor4f(0.0f, 0.0f, 0.0f, 1.0f);
    }
    glBegin(GL_LINES);
    for (int i = 0; i < m_pAxes->getSubAxesNum(); i++)
    {
        phi = m_pAxes->getPhi(axis, i);
        theta = m_pAxes->getTheta(axis);
        length = m_pAxes->getLength(axis, i, m_depth);
        getXYZ(theta, phi, length, x, y, z);
        glVertex3f(0.0f, 0.0f, 0.0f);
        glVertex3f(x, y, z);
    }
    glEnd();

    if (m_transp < 0.5f && m_pAxes->isSelected(axis))
    {
        glColor4f(0.0f, 0.0f, 0.0f, 1.0f);
        glBegin(GL_LINES);

        float theta = m_pAxes->getTheta(axis);
        float lx, ly, lz, laxisl, lphi, llength;
        float hx, hy, hz, haxisl, hphi, hlength;

        glColor4f(m_config.axr, m_config.axg, m_config.axb, 1.0f);
        glBegin(GL_LINES);
        for (int i = 1; i < m_pAxes->getSubAxesNum(); i++)
        {
            laxisl = m_pAxes->getLength(axis, i-1, m_depth);
            lphi = m_pAxes->getPhi(axis, i-1);
            llength = laxisl * m_pDataTable->getProportion(axis, i+1);
            getXYZ(theta, lphi, llength, lx, ly, lz);
            haxisl = m_pAxes->getLength(axis, i, m_depth);
            hphi = m_pAxes->getPhi(axis, i);
            hlength = haxisl * m_pDataTable->getProportion(axis, i+1);
            getXYZ(theta, hphi, hlength, hx, hy, hz);
            glVertex3f(lx, ly, lz);
            glVertex3f(hx, hy, hz);
        }
        glEnd();
    }
}

void C3DKiviatDiagram::drawAxes()
{
    for (int i = 0; i < m_pAxes->getAxesNum(); i++)
    {
        glLoadName(i);
        drawAxis(i);
        if (m_pAxes->isSelected(i))
        {
            drawLable(i);
        }
    }
}
```

```
void C3DKiviatDiagram::drawBlock(int attr, int rels)
{
    int Attr[2] = {0};
    Attr[0] = attr;
    Attr[1] = attr + 1;
    int Rels[2] = {0};
    Rels[0] = rels;
    Rels[1] = rels + 1;
    if (Attr[1] == m_pAxes->getAxesNum())
    {
        Attr[1] = 0;
    }
    float r, g, b;
    if (m_pDataTable->getRelsNum() > 2)
    {
        colorCoding((float)(rels - 1) /
                    (float)(m_pDataTable->getRelsNum() - 2), r, g, b);
    }
    else
    {
        r = 0.0f; g = 1.0f; b = 0.0f;
    }
    float transp = m_transp;
    if (m_pAxes->isSelected(Attr[0]) && m_pAxes->isSelected(Attr[1]) &&
        m_pAxes->isSubSelected(rels - 1))
        transp = 1.0f;
    float axislength[2], length[4], theta[2], phi[2];
    float x[4], y[4], z[4];
    for (int i = 0; i < 2; i++)
    {
        axislength[i] = m_pAxes->getLength(Attr[i], Rels[0] - 1, m_depth);
        theta[i] = m_pAxes->getTheta(Attr[i]);
        phi[i] = m_pAxes->getPhi(Attr[i], Rels[0] - 1);
    }
    for (int i = 0; i < 4; i++)// calculate the four points
    {
        length[i] = m_pDataTable->getProportion(Attr[i/2],Rels[i % 2])*axislength[i/2];
        getXYZ(theta[i / 2], phi[i / 2], length[i], x[i], y[i], z[i]);
    }
    if ((length[0] - length[1]) * (length[2] - length[3]) >= 0)
    {   // simply draw a quad
        glColor4f(r, g, b, transp);
        glBegin(GL_QUADS);
        if (m_colorCode) glColor4f(m_config.r, m_config.g, m_config.b, transp);
        glVertex3f(x[0], y[0], z[0]);
        if (m_colorCode) glColor4f(r, g, b, transp);
        glVertex3f(x[1], y[1], z[1]);
        glVertex3f(x[3], y[3], z[3]);
        if (m_colorCode) glColor4f(m_config.r, m_config.g, m_config.b, transp);
        glVertex3f(x[2], y[2], z[2]);
        glEnd();
    }
    else
    {   // draw two triangles
        float xi, yi, zi;
        getIntersect(x[0], y[0], z[0], x[2], y[2], z[2],
            x[1], y[1], z[1], x[3], y[3], z[3], xi, yi, zi);
        glColor4f(r, g, b, transp);
        glBegin(GL_TRIANGLES);
        if (m_colorCode) glColor4f(m_config.r, m_config.g, m_config.b, transp);
        glVertex3f(x[0], y[0], z[0]);
        if (m_colorCode) glColor4f(r, g, b, transp);
        glVertex3f(x[1], y[1], z[1]);
        if (m_colorCode) glColor4f((m_config.r + r) / 2.0f,
            (m_config.g + g) / 2.0f, (m_config.b + b) / 2.0f, transp);
        glVertex3f(xi, yi, zi);
        glVertex3f(xi, yi, zi);
        if (m_colorCode) glColor4f(r, g, b, transp);
        glVertex3f(x[3], y[3], z[3]);
        if (m_colorCode) glColor4f(m_config.r, m_config.g, m_config.b, transp);
        glVertex3f(x[2], y[2], z[2]);
        glEnd();
    }
```

```
    glColor4f(m_config.axr, m_config.axg, m_config.axb, transp);
    glBegin(GL_LINES);
    glVertex3f(x[0], y[0], z[0]);
    glVertex3f(x[2], y[2], z[2]);
    glVertex3f(x[1], y[1], z[1]);
    glVertex3f(x[3], y[3], z[3]);
    glVertex3f(x[0], y[0], z[0]);
    glVertex3f(x[1], y[1], z[1]);
    glVertex3f(x[2], y[2], z[2]);
    glVertex3f(x[3], y[3], z[3]);
    glEnd();
    if (transp < 0.5f && m_pAxes->isSelected(attr))
    {
        glColor4f(r, g, b, 1.0f);
        glPushMatrix();
        float newtheta = theta[0] * 180.0f / pi + 90.0f;
        float newphi = -phi[0] * 180.0f / pi;
        float newlength = (length[1] - length[0]);
        if (newlength >= 0)
        {
            glTranslatef(x[0], y[0], z[0]);
        }
        else
        {
            glTranslatef(x[1], y[1], z[1]);
            newlength = -newlength;
        }
        gluQuadricDrawStyle(quadObj,GLU_FILL);
        gluQuadricNormals(quadObj,GL_FLAT);
        gluQuadricOrientation(quadObj,GLU_INSIDE);
        gluQuadricTexture(quadObj,GL_TRUE);
        glRotatef(90.0f, 1.0f, 0.0f, 0.0f);
        glRotatef(newtheta, 0.0f, 1.0f, 0.0f);
        glRotatef(newphi, 1.0f, 0.0f, 0.0f);
        gluCylinder(quadObj, 0.05, 0.05, newlength, 4, 1);
        glPopMatrix();
        glEnd();
    }
} // End of C3DKiviatDiagram::drawBlock()


void C3DKiviatDiagram::drawBlocks()
{
    for (int i = 1; i < m_pDataTable->getRelsNum(); i++)
    {
        glLoadName(i - 1);
        for (int j = 0; j < m_pDataTable->getAttrNum(); j++)
        {
            drawBlock(j, i);
        }
    }
}

void C3DKiviatDiagram::drawLable(int attr)
{
    float x, y;
    glColor4f(m_config.tr, m_config.tg, m_config.tb, 1.0f);
    x = cos(m_pAxes->getTheta(attr)) * m_depth * 1.1f;
    y = sin(m_pAxes->getTheta(attr)) * m_depth * 1.1f;
    glRasterPos2f(x, y);
    glPrint("%d %s", attr + 1, m_pDataTable->getLable(attr).c_str());
}
```

```
int Selection(int seltype)
{
    GLuint  buffer[512];
    GLint   hits;
    GLint   viewport[4];
    glGetIntegerv(GL_VIEWPORT, viewport);
    glSelectBuffer(512, buffer);
    (void) glRenderMode(GL_SELECT);
    glInitNames();
    glPushName(0);
    glMatrixMode(GL_PROJECTION);
    glPushMatrix();
    glLoadIdentity();
    gluPickMatrix((GLdouble) mouse_x,
                  (GLdouble) (viewport[3]-mouse_y), 5.0f, 5.0f, viewport);
    gluPerspective(45.0f, (GLfloat) (viewport[2]-viewport[0])/
                   (GLfloat) (viewport[3]-viewport[1]), 0.1f, 100.0f);
    glMatrixMode(GL_MODELVIEW);
    g_3dkd.initDrawing();
    switch(seltype)
    {
    case SEL_AXES:
        g_3dkd.drawAxes();
        break;
    case SEL_RELS:
        g_3dkd.drawBlocks();
        break;
    default:
        break;
    }
    glMatrixMode(GL_PROJECTION);
    glPopMatrix();
    glMatrixMode(GL_MODELVIEW);
    hits=glRenderMode(GL_RENDER);
    if (hits > 0)
    {
        int choose = buffer[3];
        int depth = buffer[1];
        for (int loop = 1; loop < hits; loop++)
        {
            if (buffer[loop*4+1] <= GLuint(depth))
            {
                choose = buffer[loop*4+3];
                depth = buffer[loop*4+1];
            }
        }
        switch(seltype)
        {
        case SEL_AXES:
            g_3dkd.selectAxis(choose);
            break;
        case SEL_RELS:
            g_3dkd.selectSubAxis(choose);
            break;
        default:
            break;
        }
    }
    return hits;
}
```

```
int getIntersect(float l1x1, float l1y1, float l1x2, float l1y2,
                 float l2x1, float l2y1, float l2x2, float l2y2,
                 float& x, float& y)
{
    //L1: a1x + b1y = c1
    float a1 = l1y2 - l1y1;
    float b1 = l1x1 - l1x2;
    float c1 = l1x1 * l1y2 - l1x2 * l1y1;
    //L2: a2x + b2y = c2
    float a2 = l2y2 - l2y1;
    float b2 = l2x1 - l2x2;
    float c2 = l2x1 * l2y2 - l2x2 * l2y1;
    float detab = a1 * b2 - a2 * b1;
    if(detab == 0)
    {
        float r;
        if(a2 != 0)
        {
            r = a1 / a2;
        }
        else
        {
            r = b1 / b2;
        }
        if(c1 == 0 && c2 == 0)
        {
            return 0;
        }
        if(r == c1 / c2)
        {
            return 0;
        }
        else
        {
            return 0;
        }
    }
    x=(c1 * b2 - c2 * b1) / detab;
    y=(a1 * c2 - a2 * c1) / detab;
    return 1;
}

int getIntersect(float l1x1, float l1y1, float l1z1, float l1x2, float l1y2, float l1z2,
                 float l2x1, float l2y1, float l2z1, float l2x2, float l2y2, float l2z2,
                 float& x, float& y, float& z)
{
    float tx, ty, tz, tx2, ty2, tz2;
    int r1 = getIntersect(l1x1, l1y1, l1x2, l1y2, l2x1, l2y1, l2x2, l2y2, tx, ty);
    int r2 = getIntersect(l1x1, l1z1, l1x2, l1z2, l2x1, l2z1, l2x2, l2z2, tx2, tz);
    int r3 = getIntersect(l1y1, l1z1, l1y2, l1z2, l2y1, l2z1, l2y2, l2z2, ty2, tz2);
    if (r1 != 0)
    {
        if (r2 != 0)
        {
            float d = (tx - tx2) / 2.0f;
            x = tx - d;
            y = ty;
            z = tz;
            if (d > 0.01f || d < -0.01f)
            {
                return 0;
            }
            return 1;
        }
        else if (r3 != 0)
        {
            float d = (ty - ty2) / 2.0f;
            x = tx;
            y = ty - d;
            z = tz2;
```

```
            if (d > 0.01f || d < -0.01f)
            {
                return 0;
            }
            return 1;
        }
        else
        {
            x = tx;
            y = ty;
            z = (l1z1 + l1z2 + l2z1 + l2z2) / 4.0f;
            return 1;
        }
    }
    else if (r2 != 0)
    {
        if (r3 != 0)
        {
            float d = (tz - tz2) / 2.0f;
            x = tx2;
            y = ty2;
            z = tz - d;
            if (d > 0.01f || d < -0.01f)
            {
                return 0;
            }
            return 1;
        }
        else
        {
            x = tx2;
            y = (l1y1 + l1y2 + l2y1 + l2y2) / 4.0f;
            z = tz;
            return 1;
        }
    }
    else if (r3 != 0)
    {
        x = (l1x1 + l1x2 + l2x1 + l2x2) / 4.0f;
        y = ty2;
        z = tz2;
        return 1;
    }
    else
    {
        return 0;
    }
}
```

Matematiska och systemtekniska institutionen
SE-351 95 Växjö

Tel. +46 (0)470 70 80 00, fax +46 (0)470 840 04
http://www.vxu.se/msi/