A Solar System Metaphor for 3D Visualisation of Object Oriented Software Metrics

Hamish Graham, Hong Yul Yang, Rebecca Berrigan
Software Engineering School
Computer Science Department
University of Auckland
Auckland, New Zealand
rebecca@cs.auckland.ac.nz

Abstract

Early detection of problems within a code base can save much effort and associated cost as time progresses. One method of performing routine assessment of code with a view to pre-emption of a decline in quality is to collect software metrics associated with code size and complexity. Despite the best efforts of the last decade to establish this type pf empirical analysis as best practice, it is not yet a standard activity in software production. One way of potentially increasing empirical analysis activity on this realm is to contemplate visualisation as a means to readily analyse either static or evolving code to perceive in real time suspected areas of risk within the code base. This paper presents a first attempt at 3D visualisation of software metrics by using a familiar metaphor to present empirical concepts.

Keywords: Object Oriented Metrics, Information Visualisation, Visualisation Metaphors, OpenGL, Java, GL4Java.

1 Introduction

Finding ways to objectively measure software in order to provide an indication of quality has been steadily gaining ground over the past decade. There is an attraction toward quantitative empirical data that reflects code complexity, especially when correlated with software production external attributes such as maintenance effort or required rework. If impending costs can be associated with measurable software traits, these can potentially be pre-empted or controlled.

Optimum value can be realised through early detection of unhealthy trends (risks) in the evolution of the code. While metrics collection and interpretation is a lively area of research and debate, the presentation of the resulting data to everyday stakeholders such as software developers and architects has not received much attention. In the interests of early risk detection in software, we postulate that the ability of the responsible person to identify ***through visualisation technology*** areas of code that may need attention could be of great value.

It is important at this stage to emphasise that this work is not a treatise of the software metrics themselves. We have drawn on metrics that have been thoroughly referenced in academic literature, and are collected by commercial products where they are advertised to assist in improving design quality, refactoring, software evolution *etc.*. In visualising these numbers we refrain from passing any judgement on the validity of the numbers themselves. Our aim in this work is to demonstrate the value added by visualising the data, even if the validity of that data is sometimes hotly debated in current empirical software development literature.

The paper is organised as follows. The relevance of software metrics in software production is presented in the following section, where certain 'mainstream' metrics are highlighted as being appropriate for visualisation in this work, and the tools utilised to do so are presented. Section 3 then concentrates on the visualisation in presenting related work, our chosen metaphor and a description of the tool that has been developed using GL4Java (*GL4Java* 2003) . The architecture of the system is briefly outlined in Section 3.4, and then future plans for research in this area as well as relevant conclusions complete the paper.

2 Software Metrics

Software metrics associated with code have been discussed for decades(Yourdon & Constantine 1979, Fenton & Pfleeger 1997), yet the advent of object oriented languages prompted awareness that this type of programming may require more diverse measurement techniques.

First emergence from Chidamber and Kemerer (Chidamber & Kemerer 1991) of a proposed suite of measures designed for object oriented systems (C-K OO metrics) paid attention to the hidden complexities that arise in object oriented code. Research focusing on these ***hidden dependencies*** has progressed to indicate that complexities and relationships in code can exist as a 'trail' of interconnections (Z. Yu 2001). The aspect of these connections that is deemed to be ***hidden*** is our inability to detect these dependencies by simply reading the source code (as one does during routine maintenance).

The potential of software metrics is tangible yet not fully realised in real world scenarios. Software metrics provide clear qualitative measures, and could also provide early detection of risks to quality - whether this be at design or development level. Intuition suggests that tracking these metrics over the lifetime of product provides insight into the way that the product is growing and changing. One challenge (of many) in this field is to lower the interpretive barrier for everyday practitioners, and in this make routine empirical assessment part of production culture.

In this work we have targeted technology that makes metrics more 'visible' to the everyday software practitioner. For the purposes of demonstration, however, we have required real metrics from real systems. The following sections outline the textbook metrics selected for

collection from moderately sized open source code bases. The tools used to collect these metrics are also presented. We are presenting this information for completeness, not to imply that the metrics collected or the tools themselves are of sufficient quality for real world, enterprise system analysis. Rigorous re-examination of this part of the work will be required before real world, large scale studies (the ultimate assessment for this type of tool) are undertaken.

2.1 Metrics Collection Tools Used

Open source tools were used to collect metrics from open source systems, allowing the bulk of our effort to be spent concentrating on the visualisation technology and a suitable metaphor. Code to be assessed was chosen to be Java due to a wide range of open source test cases, existing tools and our familiarity with the language.

Java NCSS (*Java NCSS* 2003) collects data relating to code size and complexity whereas Dependency Finder was used to extract coupling and inheritance data from the system. Neither tool provides more than only basic console style tabular reporting of the data. These tools were powerful enough, however, to be used as collectors to aggregate and store for our 3D visualisation.

Open source offerings can also take form of IDE plug-ins. IBM's Eclipse IDE (*Eclipse IDE* 2003) for Java and Sun's equivalent, NetBeans (*NetBeans IDE* 2003), both offer plug-ins for metrics collection from code managed at the time within the IDE. Data in both plug-ins appear as a scrollable paned table in place of the output console within the IDE. Version based snapshots are not possible. Visualisation of metrics is an option inside the Eclipse plug-in, and this is further discussed in Section 3 (see 3. We contemplated an embedded IDE visualisation, yet preferred at this stage not to tie the tool in to a particular technology or provider.

2.2 Metrics Selected For Visualisation

Table 1 below lists the C-K OO metrics that we chose to collect for the purpose of visualisation. These metrics were selected from a standard software engineering text (Pressman 2001) which itself draws on foundational empirical OO literature of the last 1-2 decades.

Metric	Tool
Lines of Code(LOC)	NCSS
Number of Children(NOC)	DF
Weighted Methods (WMC)	NCSS
Coupling Between Objects(CBO)	DF
Depth of Inheritance Tree (DIT)	DF

Table 1: CK OO metrics listed, with those collected by open source tool where NCSS = Java NCSS (*Java NCSS* 2003) and DF = Dependency Finder (*Dependency Finder* 2003) .

Our selection of metrics for visualisation was bounded by two key factors

1. **Relevance** The metrics selected are commonly reported in literature as well as marketed in commercial realms. Open source tools were used to collect these metrics, so availability of these tools were a contributing factor in selection.

2. **Efficacy in Visualisation** There are undoubtedly some metrics that are more 'spatially' conceptual or conceptual through relationship to other entities. Those that were concurrent with the aforementioned metrics in (1) were selected in anticipation of their visual impact.

Metrics were collected and stored in an embedded McKoi database (*McKoi Database* 2003). Persistent storage of the data was foreseen to cater for large data sets and provide the opportunity to collect data for successive releases of software for evolutionary studies. The architecture for collection is shown in Figure 1.

3 3D Visualisation of Software Metrics

Our choice of 3D visualisation over 2D was driven mostly by novelty, but also through recognition of the delicate task of trying to represent relationships inside complex code. The following sections outline our known visualisations of code status as well as some rationale behind our choice of metaphor for a 3D visualisation.

3.1 Related Work

Existing metrics visualisation tools generally target the code in production at the time of analysis. The most prominent offering is from WebGain (*Web Gain Quality Analyser* 2003) as part of the QualityAnalyser product. This product is fully featured and clearly built for large data sets, yet the visualisation of the data is 2D graph and chart oriented.

An early open source tool called JMetric (*JMetric* 2003) used a kiviat or spider chart to try and provide a single view of at least 6 metrics in one view. This approach has been enveloped in latest offerings from Borland in their Together ™ControlCenter ™package (*Borland Together Control Center* 2003), targeting local empirical changes in code as the developer is refactoring.

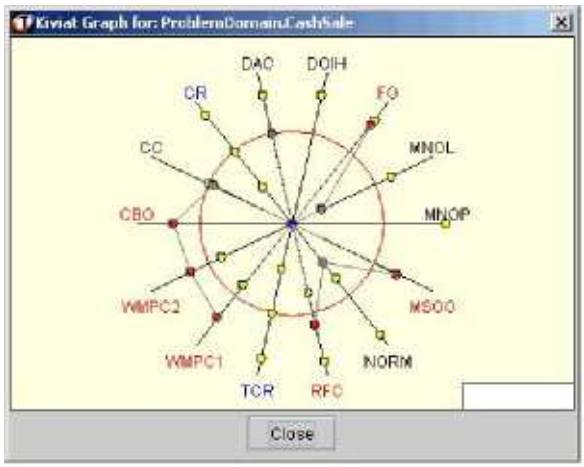

Figure 2: Kiviat charting as seen in the Borland's TogetherJ (*TogetherJ* 2003)

Visualisation of *large* data sets is by no means new and is well investigated by many institutions worldwide. Of these, the University of Maryland's HCI group (*Human Computer Interaction Laboratory* 2003) has undertaken work to visualise code size in a 2D fashion which has proved effective even for large code bases like the Linux kernel (*Treemap* 2003). This type of visualisation could be very powerful in nightly build situations where a simple metric like defects/LOC can be effectively visually reported for each module.

The Eclipse IDE (*Eclipse IDE* 2003) will accept a plug-in to collect standard OO metrics for code accessed by the IDE. This is displayed in a table in the first instance, yet the user can request a visualisation in 2D by viewing an embedded 2D graph viewer called TouchGraph (*TouchGraph* 2003). TouchGraph appears to use a spring

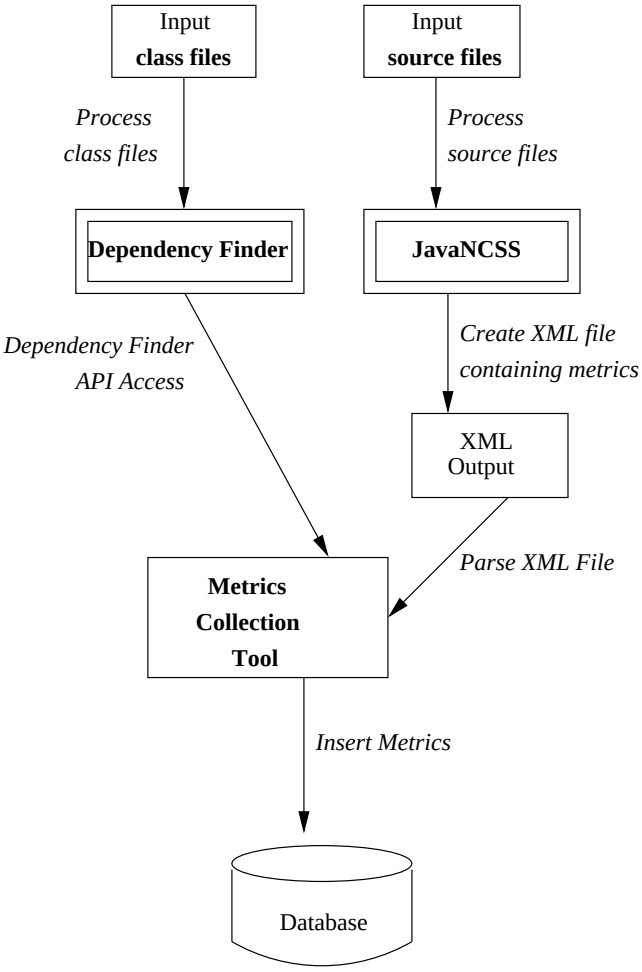

Figure 1: Metric collection system architecture for our study.

algorithm to render the view each time a node is selected, spacing the nodes so that obscuration does not become an issue. This is a powerful visualisation, yet the limits of the tool's ability to cater for large data sets or filter data sets for exceptional features seems limited at this time. Figure 3 shows the metrics plug-in in operation, with TouchGraph (darker panel) displaying nodes representing packages and lines as dependencies. Each time a node is selected it is centered in the view and the other nodes repositioned algorithmically as to avoid obscuration.

The efficacy of these models wanes somewhat when trying to capture changes in the code over time. Evolutionary visualisation of the Linux kernel in 3D has been performed (*A 3D animation of Linux kernel development* 2003) and the resulting data animated to be viewed as a movie. The visualisation model is a basic ball and stick model to represent files (nodes) and function calls (sticks). This technique is powerful in elucidating 'dense' areas of code (nodes attributed to certain modules in the code) but can afford no extra information about code size or complexity and suffers occasionally from obscuration. The efficacy of this model for its relative simplicity, however, was an influence in our choice of metaphor as we decided how code metrics should be portrayed for users to best interact with.

By far the most related and most recent report that is relevant in the context of this study is work undertaken by Irwin and Churcher (Irwin & Churcher 2003) in carefully collecting metrics from systems, performing adept filtering and then piping the data to the screen to be visualised in class 'clusters'. The quality of the data collection in this work far exceeds our efforts, yet differs from this work in its direct pipe of data from memory to screen. The

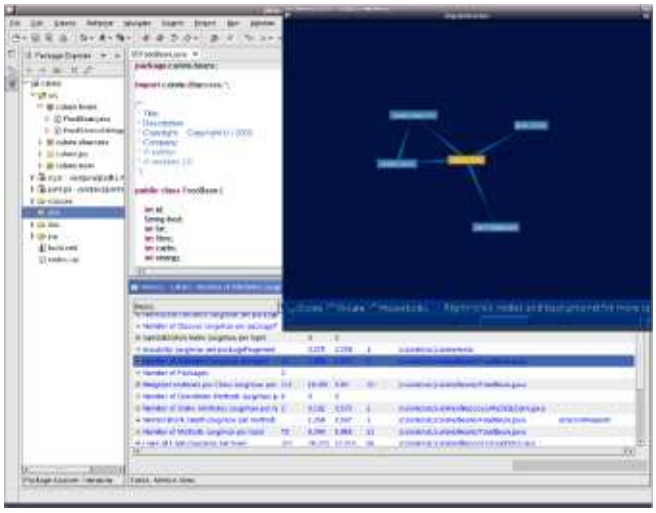

Figure 3: Metrics visualisation provided by the Eclipse metrics plugin (*Eclipse Metrics Plug-in* 2003) and TouchGraph (*TouchGraph* 2003).

visualisation in VRML is deft, and approaches that task of representing code as such, rather than departing as we have in this work to consider a rich and familiar real world metaphor.

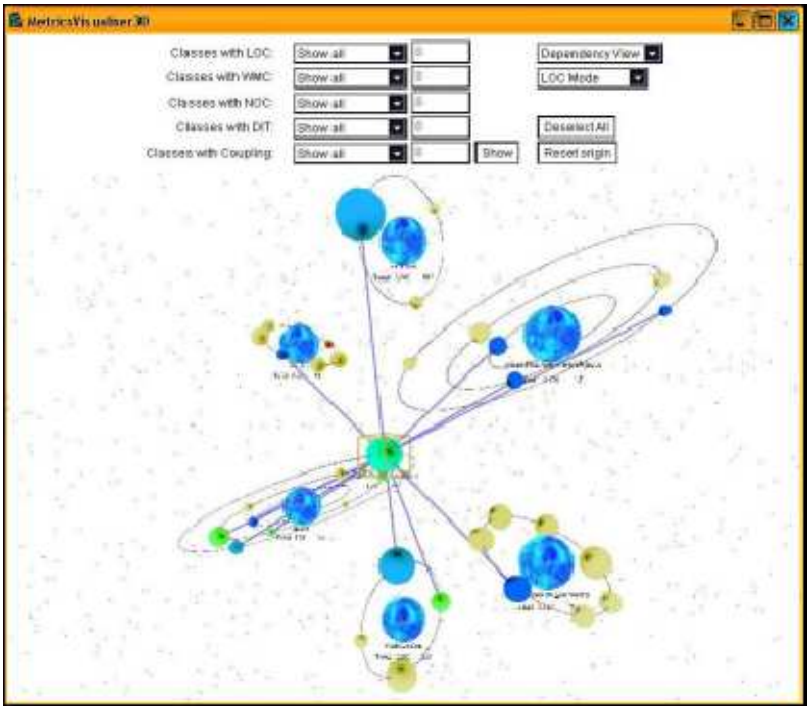

Figure 4: A screenshot of the solar system metaphor as applied to the source code of the tool itself. The true colour scheme is a real starscape, but colours have been inverted for clarity in publication.

3.2 The Solar System Metaphor

Software metrics hold great value for any environment prepared to make best use of them, yet like any other diagnostic activity that involves complex analysis and swathes of seemingly similar data, resistance to regular and rigorous analysis is high in real world contexts. To present these metrics in a way that involved familiar objects, concepts and relationships in a 3D virtual space, a visualisation metaphor was desired.

One of the most common metaphors for 3D visualisation is a city (Panas, Berrigan & J.C.Grundy 2003) but the city metaphor (where buildings, streets *etc.* are effectively grounded to a 2D plane) reduced our ability to present complex relationships, increased obscuration and required seemingly too much visualisation effort for relative informational output regarding the state of the code being represented.

A simpler model was constructed of a **solar system**, derived from the effective Linux kernel 3D evolutionary animation. The solar system was an attractive metaphor due to its ability to effectively represent size and inheritance (through orbits and/or rings). One of the greatest challenges in the OO model is to represent coupling between objects in a way that does not crowd or detract from the metaphor. The solar system does not provide a means native to the metaphor to express connections or relationships (gravity seemed a little too esoteric) but it was decided that basic ball and stick representations may still represent the necessary information.

3.3 The Solar Metric Visualisation Tool : Description

A screenshot of a visualisation of the source code of the visualisation tool itself is presented in Figure 4. The true colour scheme is a real starscape, but colours have been inverted for clarity in publication. The tool uses the solar system metaphor to represent the structural and relational aspects of the code associated with the collected metrics in the following way.

Suns Represent a Java package

Planets Represent Java objects or classes. Colours of classes dictate class vs interface, and in 'Coupling Mode', afferent and efferent coupling.

Orbits Represent the inheritance level within the package. Planets in each orbit represent an inheritance hierarchy.

Planet Size Represents LOC in a class. Actual figures are presented as the user clicks on the planet with the mouse.

Further features are implemented as such

Raw Data The raw metrics from which the visualisation is derived is displayed as the user selects a planet with the mouse.

Connectors Connections are displayed between planets to represent coupling between those classes or inheritance views.

Thresholds Filtering can be effected by setting thresholds on any of the collected metrics. This is especially useful when looking for relative extremes within large data sets where small differences are difficult to detect visually.

Figures 5 and 6 demonstrate the contrast of size and complexity in visualisation. These two figures present the same view of the code, yet focus on LOC and Coupling respectively. It is apparent to the naked eye in real time that sheer size of the code base does not reflect the nature of the interconnectivity between the respective classes.

The visualisation performs adequately on code such as JUnit v3.8.1 (431KB) and JEdit v 4.2 (1.3 MB). Metric collection performs at the order of minutes, yet unlike many available tools for this kind of analysis, the metrics need only be collected once for a designated version or release. User interaction with the GL4Java Visualisation is very promising as the screen is highly interactive and responsive at this data scale. A detailed comparative performance focused study is underway to establish the limits for size and complexity of systems that could be presented in this way.

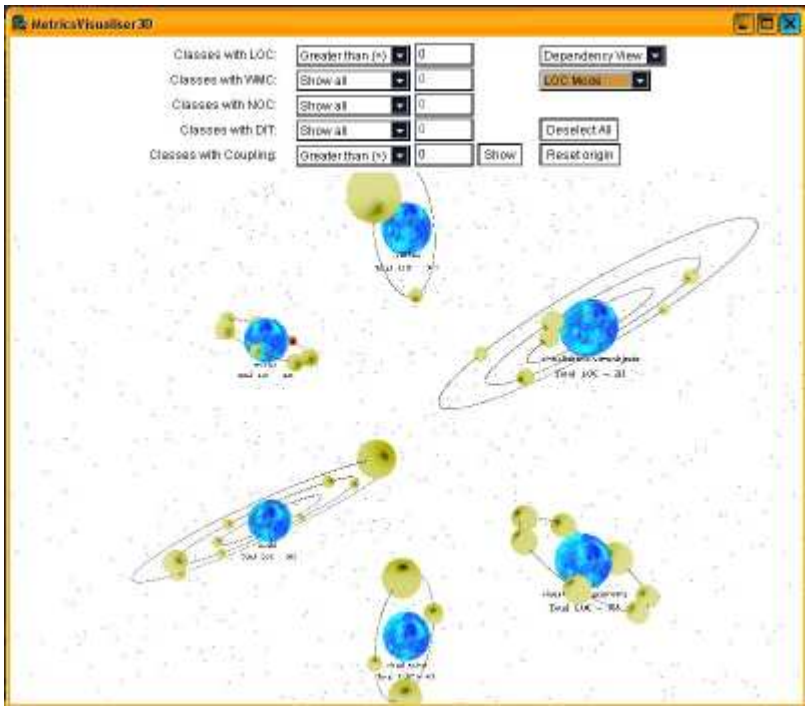

Figure 5: Visualisation in 'Lines of Code mode' indicates relative size of each class.

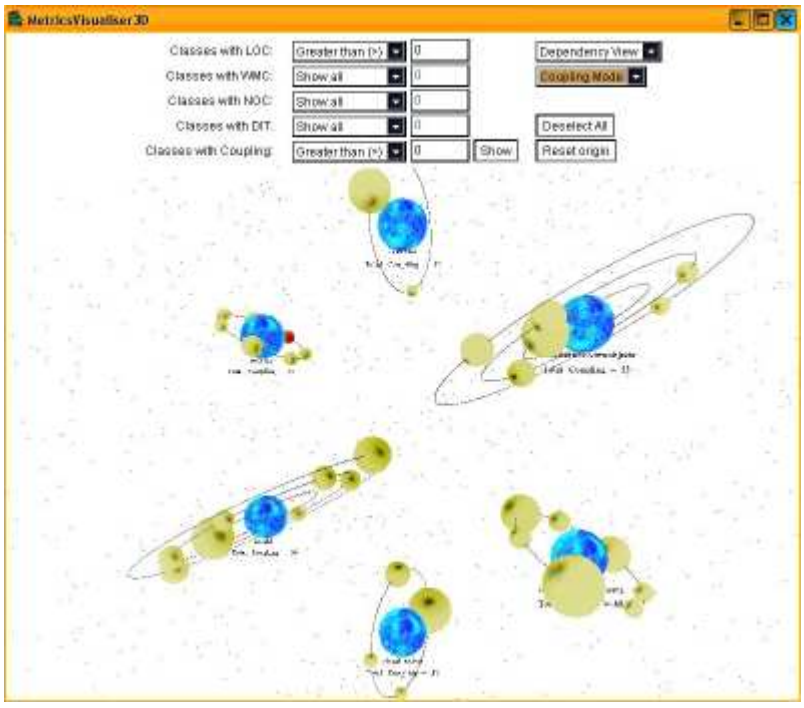

Figure 6: Visualisation in 'Coupling Mode'. Note the contrast in relative size of the planets in some orbits.

3.4 Visualisation Tool Architecture

The underlying tool architecture is portrayed in Figure 7. The Model-View-Controller architecture has been adhered to as it allows for changes in collection methods or even visualisation technology. Java code that binds to OpenGL has been encapsulated in view specific code, referencing data-centric control classes that are fed from the database. Database population has not impinged on the source of open source collection tools, providing the opportunity for a range of tools to be used in conjunction with the ones currently employed.

4 Future Work

Aspects of this prototype that can be further developed :

- Improve the quality of the metrics collected for visualisation. The efficacy of the metaphor can only be fully realised through visualising valid data.
- Develop the metaphor further to cater for more obscure and complex empirical concepts like indirect coupling and cohesion.
- An ability to visualise evolutionary changes through a code base using this metaphor.
- The ability to 'map' external attributes such as maintenance effort to selected features in the metaphor (the mention of black holes for associated cost has been suggested).
- Real world testing of this system to prove its efficacy as well as validate the metrics themselves.

As the size of our test systems increases the need for more sophisticated navigation and rendering will most probably be required. Enhancements such as user defined axes of rotation as well as compact 'bird's eye' views of the entire system that allows for a system wide perspective and ready access to class level information.

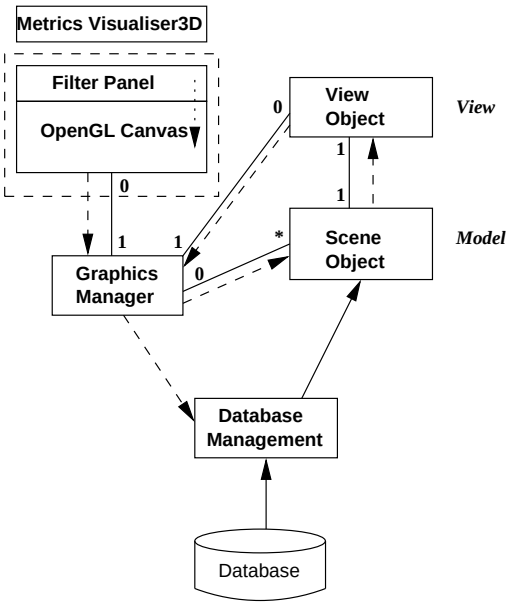

Figure 7: Underlying architecture of the 3D visualisation tool.

5 Conclusions

The tool presented in this paper provides a record of progress in taking early steps in the visualisation of software metrics associated with a code base. The validity of the selected metrics has not been the subject of this work, yet our choice of metaphor has provided a platform to further test the metrics, the metaphor and the scalability of this approach for large data sets. The tool has performed well in providing detailed data regarding a code base to the user without obscuration, as well as highlighting areas of the code that are relatively sever in size or complexity.

References

A 3D animation of Linux kernel development (2003), http://perso.wanadoo.fr/pascal.brisset/kernel3d/. (accessed 10th Oct 2003).

Borland Together Control Center (2003), http://www.togethersoft.com/products/index.jsp. (accessed 10th Oct 2003).

Chidamber, S. R. & Kemerer, C. F. (1991), Towards a metrics suite for object oriented design, *in* 'Proceedings of 6th ACM Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA)', pp. 197–211.

Dependency Finder (2003), http://depfind.sourcefourge.net/ (accessed 10th Oct 2003). (accessed 10th Oct 2003).

Eclipse IDE (2003), http://www.eclipse.org/. (accessed 10th Oct 2003).

Eclipse Metrics Plug-in (2003), http://sourceforge.net/projects/metrics. (accessed 10th Oct 2003).

Fenton, N. E. & Pfleeger, S. L. (1997), *Software Metrics: A Rigorous & Practical Approach*, second edn, PWS Publishing Company.

GL4Java (2003), http://www.jausoft.com/gl4java.html. (accessed 10th Oct 2003).

Human Computer Interaction Laboratory (2003), http://www.cs.umd.edu/hcil/. (accessed 10th Oct 2003).

Irwin, W. & Churcher, N. (2003), 'Object oriented metrics: Precision tools and confogurable visualisations', *Proceedings of the Ninth International Software MEtrics Symposium (METRICS '03)* .

Java NCSS (2003), http://www.kclee.com/clemens/java/ (accessed 10th Oct 2003). (accessed 10th Oct 2003).

JMetric (2003), http://www.it.swin.edu.au/projects/jmetric/. (accessed 10th Oct 2003).

McKoi Database (2003), http://www.mckoi.com/database. (accessed 10th October 2003).

NetBeans IDE (2003), http://www.netbeans.org/. (accessed 10th Oct 2003).

Panas, T., Berrigan, R. & J.C.Grundy (2003), A 3d metaphor for software production visualization, *in* E. Banissi, K. Börner, C. Chen, G. Clapworthy, C. Maple, A. Lobben, C. J. Moore, J. C. Roberts, A. Ursyn & J. Zhang, eds, 'Seventh International Conference on Information Visualization, IV 2003, 16-18 July 2003, London, UK', IEEE Computer Society, pp. 314–319.

Pressman, R. S. (2001), *Software Engineering: A Practitioner's Approach*, fifth edn, McGraw Hill.

TogetherJ (2003), TogetherJ, http://www.togethersoft.com/products/index.jsp. (accessed 10th Oct 2003).

TouchGraph (2003), http://www.touchgraph.com/. (accessed 10th Oct 2003).

Treemap (2003), http://www.cs.umd.edu/hcil/treemap/. (accessed 10th Oct 2003).

Web Gain Quality Analyser (2003), http://www.webgain.com/. (accessed 10th Oct 2003).

Yourdon, E. & Constantine, L. L. (1979), *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Prentice Hall.

Z. Yu, V. R. (2001), 'Hidden dependencies in program comprehension and change propagation', *Ninth IEEE International Workshop on Program Comprehension* .