Visualising Memory Graphs: Interactive Debugging using Java3D

Darius Bradbury

May 19, 2008

Abstract

This report describes a new way of visualising Java run-time objects, and their associated memory graphs. Using the Eclipse debugging framework, alongside the Java3D platform, it aims to describe methods for extracting useful debugging information from a running program and displaying this information in a three-dimensional space. The focus of this report deals with how using a three-dimensional space can enhance the debugging experience, introduce interesting visualisations of programs, and create a basis for future debugging in this way. The result is a user-friendly, efficient system whicb can visualise large programs in a relatively small amount of screen real-estate. This report shows that three-dimensional visualisation cau be a useful tool for debugging, program analysis, and a viable alternative to traditional solutions.

Contents

1 Introduction

Debugging solutions currently available offer a wide range of information to the end-user. This information is typically displayed in a textual or 2-dimensional graphical manner. At the low level, we have available debuggers which provide insight into the state of a running program by allowing insertion of breakpoints, and displaying a summary of the program's stack at any point [14]. However, such a display of information makes it difficult for a nser to follow pointers, and references within a live program. This is true of any text-based system, whatever the graphical front-end [8]. Debuggers do go further than this however, with some allowing graphical output to display graph structures of the running program [9]. It is these kinds of systems which we build upon.

Visualisation solutions on the other hand seem to be available in an off-line format. A program is processed by means of its source code, and various layouts produced. These kinds of systems map source code to visual representations of that sonrce code [12]. These systems provide some help in debugging systems, but are much more frequently used to aid the understanding and planning of larger programs. When representing our run-time objects in 3D space, it would be useful to consider how these same principles can be applied. We aim to provide much the same information, but with respect to the run-time environment of a program, rather than its static counterpart.

Current visualisation solutions seem to be moving toward three-dimensional interfaces, examples of this can be seen in the work of Knight and Munro [11], Callaghan and Hirschmüller [6], and Maletic, Leigh, and Marcus [12] to name but a few. The reason for this is that it turns a two-dimensional piece of screen estate, into a three-dimensional world. This allows us to display many more items in a smaller amount of space. It also provides an immersive interface for exploring a program, allowing certain paths to be followed, and alleviates the problem of traversing huge two-dimensional graphs.

This project aims to combine aspects from both these fields of work, in order to visualise Java run-time objects in a three-dimensional manner. This will allow for visualisation of any program, but from a debugging perspective. This so-called three-dimensional debugger will provide users with a way of stepping through their code, visualising it on-the-fly, and providing them with a new way at looking at the program they have created. This can be either to debug it, enhance it, or simply aid understanding. This project will therefore deal with displaying this disconnected, dynamically changing set of objects, and the multiple links between them.

The idea underpinning these visualisations is that of a "Memory Graph" [17]. A memory graph, as the name suggests, represents the contents, and current state, of memory in a given system. It is this information that we aim to extract and display to the user. Memory graphs allow for an instant understanding of a program state, looking at links between objects, the number of objects, the disconnected or connected nature of the system, and a visual representation of how the program 'grows'.

The system we aim to produce will analyse Java programs in particular. As such, we will use the widespread IDE (Integrated Development Environment) 'Eclipse' as our starting platform. In particular, we will make use of JDT, the Java Development Tools and one of its main components, the built-in Java debugger [1].

This project assumes a framework sitting between the Java debugger and itself, allowing for useful extraction of the current memory state, object and primitive information, and provide prompting of state changes. We then aim to build an Eclipse plug-in which will allow for our three-dimensional space interface to sit side-by-side with the JDT debugging interface. The aim is to give the user additional debugging opportunity, as well as program visualisation, side-by-side with a comprehensible set of tools already available in the JDT framework.

1.1 Formal Definition

In order to continue from here we must formally define what is meant by a "memory graph", and what relations can be drawn from such a graph. This will provide impetus for its use, and how to go about visualising it in a three-dimensional space.

Some of the first work in useful graphical debugging was achieved by Zeller and Lütkehaus in developing their DDD (Data Display Debugger) front-end for UNIX debuggers [16]. The concept was to display data structures in the form of graphs, these graphs would be representations of the run-time components of the running program. Formally, each value in memory is considered a vertex (node), and each edge is considered to be a pointer between two such values, or in this case, vertices. In the DDD system, clicking on a node resulted in its expansion, displaying the values it references. This idea has been developed greatly, notably by Zimmermann and Zeller in their paper on 'Visualising Memory Graphs' [17].

They describe a memory graph as, "a basis for accessing and visualizing memory contents." This differs from the DDD solution as they propose an automated method for creating the whole graph. The formal definition of the structure is defined in their paper [17]; however, I will outline it below:

1.1.1 The Memory Graph

Consider a graph defined by G, where G = (V,E,root). Namely, the graph consists of a set of vertices, or nodes, a set of edges, and a dedicated root node.

Vertices V: Each vertex in the set V consists of a triple. This triple is made up of the value, type, and address of the object in memory. In Java, this can be both primitives and object instances.

Edges E: Each edge also consists of a triple, this time made up from two vertices, notably, the related vertices, and an operation. The operation relates to how we construct a name for the edge, given the parent and descendant vertices. Edges in this graph are directed, one value is referencing another, and hence, one node, is pointing to another.

Root node: In Zimmermann and Zeller's interpretation of a memory graph, the root node is a dedicated vertex which references all base variables. In other words, every variable in the scope of the memory graph is accessible from root. What this entails is that the description of a memory graph used here, creates a directed and connected graph. This project aims to generalise this requirement, allowing for a disconnected graph whereby a specified root node is not required.

1.1.2 Beyond The Memory Graph

The above describes the definition of a memory graph; however, as explained we may not always want to decide on a root node. Instead, we look to create a more general graph, but allow users to select nodes for which they would like to make the root. This then allows us to continue looking at the work of Zimmermann and Zeller, and continue to use their memory graph concept. Their paper then continues on the automatic construction of memory graphs which simply consists of creating a connected graph linking the root to all the base variables, considering the path of references which must be undertaken to get there [17].

Building on this memory graph structure is the notion of forward and backward traces. What we have already explained is the notion of a disconnected memory graph. No root node is specified, but we can ascertain the links between objects and primitives. Program slicing deals with how a certain bit of code is relevant to a particular program [2]. In particular, dynamic slicing, whereby we only consider a specific execution of a program, or in our case, the current run-time state of the program [15].

What we propose is a method similar to slicing, which I will call *tracing* to differentiate it from program slicing. We use the disconnected memory graph we have available to us, and then continue to construct a connected memory graph by picking a particular root. In other words, we are looking to centre focus on one object, and see what role it plays in the program. This provides the user with the ability to see all the relationships within a large program, but then narrow focus down to a particular object.

The notion of a trace is very much similar to that of a memory graph, in fact, Zimmermann and Zeller consider tracing to provide sub-graphs of the overall memory graph [7]. Forward tracing looks at all the references made from our user-selected 'root' node. A backward trace does the opposite, it traces all the objects that reference it, and then the objects which reference those objects, and so forth. In other words we can look at the path of referencing from a particnlar object, and the sequence of referencing to obtain a particular object, at any point in a program run.

Forward tracing can be seen as a way to look at how a particular object influences other objects and variables, in particular, showing its effect on the system as a whole. Backward tracing provides the insight into finding out which other objects inflnenced the resulting value of this object or primitive. Notably we now have an underlying connected graph structnre representing the effect of, or the objects/primitives which effect, a certain object.

This tracing system allows us to free np the notion of a memory graph, removing the necessity for the graph to be connected, and allowing us to visualise the run-time state of a whole program. Then, if the user wishes to select focus on a single object, we provide tracing options to do just that.

1.2 Project Road Map

This project aims to access a framework sitting on top of the Eclipse JDT debugger, providing access to all the underlying objects and primitives in the system. This framework will provide information as to the values of an object, as well as the links to other objects. It is our job therefore to display this information as coherently and intuitively as possible to the end user. We must be sure that the system created is capable of dynamically updating any graph in view, when the underlying program changes. In other words, we subscribe to the underlying debug model to notify us when the user steps to a different point in the program, and update our model accordingly.

We will make use of the theory of memory graphs, as well as forward and backward traces. However, we must also design intuitive ways in which to display the overall run-time environment, as well as making the job of a debugger simpler by highlighting the more interesting nodes.

With this in mind, I will now continue to explain the program in the following order. Firstly, we must cover some essential 3D modelling background information, providing us with the knowledge to build a 3D universe. Then we will look at the requirements of our system, providing the foundations for our design phase. In explaining the design I will aim to leave out uninteresting intricacies, whilst detailing the more interesting and important methods. As such, I will cover the initial steps necessary to creating a 3D universe within Eclipse, as well as having access to the underlying debugging information. Following that, I will discuss the main classes in my design, detailing their roles, and any interesting methods. I will finish the design stage by providing a graphical overview of the program as a whole, both in terms of the Java3D scene graph, and the actual classes created. Following the design stages, I will aim to make use of stringent testing to fully explore the options, and usability, of the resulting program. Finally, I will look to draw conclusions based on the design and testing stages, as well as the theories we have already discussed.

2 Background - 3D Modelling in Java

The 3D modelling system required for this project must be cleanly accessible from within our Java code, allow for dynamic changes to the 3D world, and provide a high-level intuitive interface for doing so. What we require is a system which can interface cleanly with the Eclipse window, and allow user-interaction with the underlying 3D objects.

One such three-dimensional modelling language satisfying these requirements is Java3D. The reason for this is that it provides a way to create a three-dimensional scene, completely in Java, and in a high-level manner. Whilst giving much control to the designer, Java3D abstracts away from the intricacies of 3D modelling present in many other systems [3]. The designer does not have to worry about rendering, which is done efficiently and automatically. The designers aim is to construct a scene graph, which consists of instances of Java3D objects. These Java3D objects can consist of a variety of different components, including transforms, shapes and groups of trausforms and shapes. It is the job of these objects to define the geometry, lighting, location, orientation and appearance of all the visual objects in the virtual universe.

The Java3D API consists of over a hundred classes present to aid the construction of this three-dimensional universe. The use of these will be crucial in designing a clear and concise Java3D program. In order to begin describing my approach to the creation of a memory graph, we mnst first explain the basic construction of a 3D uuiverse, in the Java3D environment.

As already stated, Java3D looks to create an underlying tree structure which is subsequently rendered. The minimal such tree in order to create a 3D universe is explained in figures 1 and 2 taken from the Java3D API guide [3].

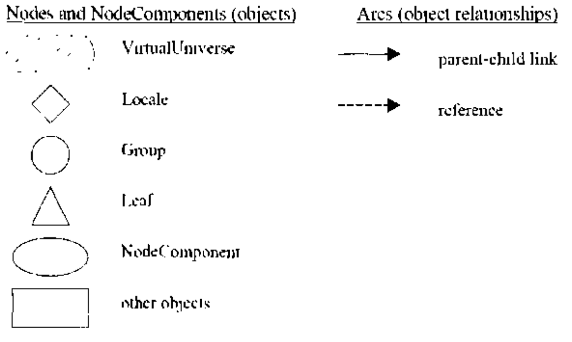

Figure 1: Key for symbols in Scene Graph [3]

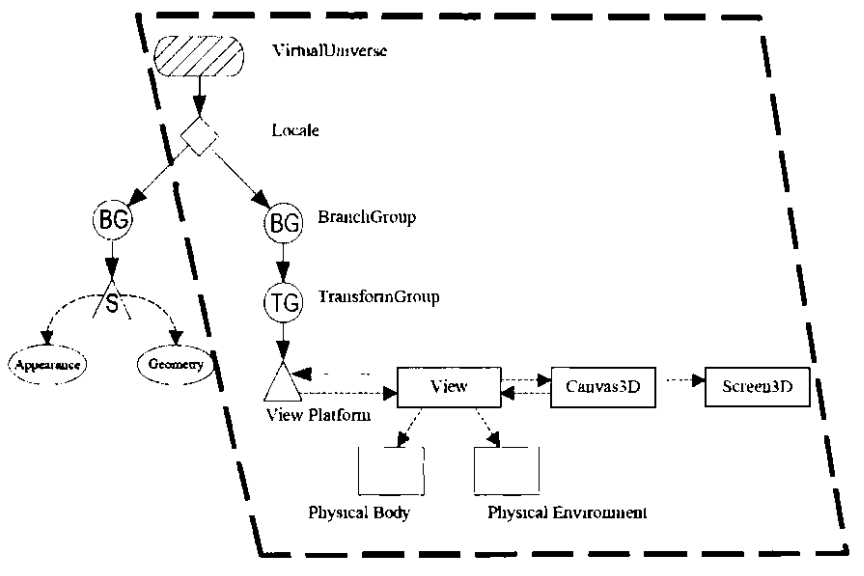

Figure 2: Scene Graph Data Structure, with minimal tree highlighted [3]

The diagram in figure 2 shows us the minimal scene graph, with an additional group node, consisting of a shape and associated appearance and geometry. Rendering this scene graph would produce a 3D environment, with a single object in it. This is a very basic Java3D program, and the intricacies are far more apparent when some code is produced. Options are available to the designer detailing the exact viewing angles and platform that are output, interactions with the physical environment (i.e. User input), and detailed construction of 3D objects.

Java3D allows a vast array of options; however certain construct rules must be adhered by. For example, looking at figure 2, we could have created a Branch-Group node, consisting of further BranchGroup nodes, each consisting of a shape. Each BranchGroup node can then have an associated transform which controls the positioning of all the objects below it.

Two relationships occur in the scene graph creation of a Java3D program. The first is a parent-child relationship used in creating the graph. This relationship must adhere to a number of rules. Namely, a group node can have any number of children, but only one parent. A leaf node can have one parent and no children. In other words, a tree with no backward links. The second relationship is known as a reference, and associates a 'NodeComponent' object, with a scene graph node. These 'NodeComponent' objects are there to define the geometry and appearance attributes used to render their associated visual object.

The tree created can be described as having a single root, being acyclic, with no backward links. This means that each leaf can be fully described based on it's so called "scene graph path". The path from the root node, to the leaf. In this way, the Java3D renderer is able to configure the most efficient render order, for the leaves of the graph. It should be noted that this is the case for the parent-child relationships, the reference relationships may go between branches, but in essence they are not dependent on this tree structure, and simply define 'shareable' attributes (such as appearance and geometry). This overall tree definition describes the construction patterns used in creating a scene graph which is renderable by the Java3D renderer, and gives a general idea as to the processes required to making a three-dimensional user interface.

3 Requirements

In designing any program, one must consider the requirements, in terms of fulfilling and achieving certain goals, whilst also adhering to the requirements in efficiency and usability enforced by an end-user. I will now discuss what these requirements are:

Accuracy: One of the most important aspects of such a program is that no objects are displayed incorrectly. We must be sure to carefully produce any code, following design patterns if possible, to ensure that what appears on the screen is consistent with the underlying model.

Efficiency: When debugging, a user may step through many breakpoints, and thus, many visualisations will be generated. As such, we must ensure that this generation process is as efficient as possible. Some efficiency considerations must come into play when considering what we expect the renderer to do, but we must also ensure that the calculation of positions, sizes, rotations of any of our objects is done efficiently also. We will be forced to keep track of possibly thousands of run-time objects, and as such, we should use appropriate data structures.

Usability: We must constantly consider the usability of the program during the design phase. After all, this program is designed as an interface to an underlying model. As such, it should be intuitive, simple, and yet allow for much variety in the need of the user. Be it for visualisation of the memory graphs, or for thorough analysis.

Extensibility: The code should be designed to allow for extensions to be made. For example, I propose the design of a layout manager to handle the positioning of objects. As such, it should be simple to create new layout patterns, without having to completely restructure the code. There should be a separation of concerns in this respect.

Integration: The program should provide seamless integration into the Eclipse framework. It would make sense to create the three-dimensional environment in a frame which sits alongside the debugger. A view which is only available during debugging, perhaps.

This list prescribes themes which should feature throughout the design process, whilst giving an overview of what we plan on achieving. We will now continue to describe various aspects of the design which aims to meet these requirements.

4 Design

The design of this 3D Debugger consists of a number of aspects. Firstly, we must consider the creation of an Eclipse plug-in, allowing for a side-by-side view of the JDT debugger, and our final three-dimensional view. Secondly, we must consider interfacing with the underlying framework that sits between the JDT debugger, and the program to be described. Thirdly, we must look at the intricacies in creating a virtual 3D universe which allows for dynamic behaviours, and user manipulation. Finally, we must consider our layout manager, the layouts we wish to display, and the general data structures in place for keeping track of all the objects in the system. In the remainder of this section I will highlight some of the more interesting aspects of the code, including code examples. Any details that are omitted will be available in the code listings in the Appendix. A general overview of our design can be found in §4.7.

4.1 Preliminaries

This section will discuss the methods used in setting up a framework to allow for the dynamic placement of 3D visual objects.

4.1.1 Creating the Eclipse Plug-in

Creating an Eclipse plug-in is a straightforward process. Dave Springgay gives a good outline of the processes necessary [13]. However, essentially we are concerned with creating an Eclipse 'View'. We give the user the option of opening this view whilst debugging an application, and thus, sticking to our usability requirements. Once we have a general Eclipse view framework set-up (for which Eclipse does most of the work), we can add any SWT (Standard Widget Toolkit) components into it. In our case, this will comprise of adding our 3D canvas to the frame.

```
/**
 * This is a callback that will allow us to create the view perspective and
 * initialise it
 *
 * What is expected is that we create a frame based on the input Composite
 * object, which will contain our view
 */
public void createPartControl(Composite parent) {

  // Create new Composite object given parent node
  Composite composite = new Composite(parent, SWT.EMBEDDED);
  // Set the 2D layout manager as a FillLayout
  composite.setLayout(new FillLayout());
  // Create a frame to add our canvas into, along with any
  // other components we wish to display
  f = SWT_AWT.new_Frame(composite);
  // Set the internal frame layout to a FlowLayout
  f.setLayout(new FlowLayout());

  /*
   * Create an Update handler object to deal with all underlying change
   * notifications. Subscribe the update handler to our intermediary
   * debugging framework
   */
  UpdateHandler uh = new UpdateHandler(this);
```

```
  DebugModelContainer.INSTANCE.addListener(uh);

  // Initialise the view (Create a virtual 3D universe and a physical
  // canvas)
  init();

  // pack the resulting frame
  f.pack();

  // Deal with maintaining the correct aspect ration during resizing
  composite.addControlListener(new ControlAdapter() {
    public void controlResized(ControlEvent e) {
      canvas3D.setSize((int) (f.getBounds().height * wideScreenRatio), f
          .getBounds().height);
    }
  });
  // Set the initial size
  canvas3D.setSize((int) (f.getBounds().height * wideScreenRatio),
      f.getBounds().height);
}
```

Listing 1: Creating the View plugin

Listing 1 shows us the implementation of the callback method used by Eclipse to generate the view. What we expect to happen is that Eclipse will call this method when the view needs to be created, providing the Composite object in which to place our 3D view. At this point we must also subscribe to the aforementioned framework, sitting on top of the JDT debugger, which will provide detailed information regarding the underlying model. This will further be explained in §4.1.2. Our initialisation method will then be discussed in greater detail in §4.1.3.

Also of interest here is the resizing procedure we create. In order to allow our 3D interface to be resized, we pass on arguments from the surrounding frame, to the underlying Canvas3D object. This is the component of the Java3D scene graph which controls the physical view output to the user. By passing on this information we can dynamically change the size of this canvas.

4.1.2 The Underlying Framework - what we aim to build our visualisations from

As we have seen, our intention is to subscribe to a framework sitting on top of the JDT debugger. What we can expect here is that this system will communicate with the debugger, process the information received, and then make available to us information we may want. Our first requirement is to be notified of underlying model changes. In other words, we expect the model to provide us with a list of underlying objects, each time the user moves to a different debugging state in the JDT debugger. Thus, we subscribe, and what we receive are notifications each time the underlying system changes. These notifications consist of all the underlying objects which have been created, or are new, and all the underlyiug objects which have been modified, or had their state changed.

This subscription system provides us with a way of interfacing nicely with the intermediary framework. We will always receive new and changed objects, and these objects will provide us with access to the underlying model. The type we expect to receive in these updates is called an 'IDebugObject'. Our job is then to construct the 3D world from our collection of these IDebugObjects, and the information available for each one.

This is a summary of the interface provided for an IDebugObject, showing us the potentially useful methods which we have access to:

```
public interface IDebugObject {

  /**
   *
   * @return the underlying IJavaValue (either a IJavaPrimitive or an IJavaObject)
   */
  public IJavaValue getValue();

  /**
   *
   * @return links to objects, including the variable representing the link
   * @throws NullLinkException
   */
  public Map<IDebugObject, IVariable> objectLinks() throws NullLinkException;

  /**
   *
   * @return backlinks to objects, including the variable representing the link
   * @throws NullLinkException
   */
  public Map<IDebugObject, IVariable> backLinks() throws NullLinkException;

  /**
   * Gets the current calculated page rank
   * @return
   */
  public double getPageRank();
}
```

Listing 2: The IDebugObject Interface

It is this underlying framework, and update process, which we rely upon to provide us with accurate information for the model. The implemeutation used is a current project by Luke Cartey; however, any implementation adhering to this same interface would provide the same functionality. Hence, we model our 3D view without the absolute need for defining the underlying programming language. If we consider extensibility, it should be clear that the designs we continue to explain could in theory be portable to any programming language for which memory map style properties can be extracted.

4.1.3 Constructing the Three-Dimensional Environment

As we have seen in §2, Java3D requires a minimal scene graph to be built. This essentially constructs the 3D environment, and the viewpoint parameters. This is our first step in creating the overall visualisation, and is accomplished by setting up the minimal scene graph as in Diagram 2. The code which accomplishes this makes a call to the Java3D utility class for universe creation. This class generates our minimal scene graph structure which is required. However, our job is to construct a new branch, and then modify this when necessary. In other words, we instantiate a universe with the SimpleUniverse object, and then attach our own BranchGroup which will contain all of our visual objects. This main BranchGroup creation method is shown in listing 3.

```
/**
 * This method sets up the main BranchGroup parameters. This is the Branch
 * of the Java3D scene graph which will contain all of our run time objects
 * We set parameters including lighting, background colour, boundingSphere
 * and capabilities of the main BranchGroup node. We also assign this
 * BranchGroup an associated TransformGroup which will deal with the
 * Transforms made upon the whole universe
 *
 * @return The Main BranchGroup node. ie. A node to add all the visual 3D
 *         objects to.
 */
public BranchGroup createScene3D() {
  ...
  // Create the Main BranchGroup
  mainBranchGroup = new BrauchGroup();

  // Create the bounding leaf node
  // This specifies the size of the rendering space.
  ...
  mainBrauchGroup.addChild(boundingLeaf);

  // Create the background
  ...
  mainBranchGroup.addChild(bg);

  // Create the ambient light
  ...
  // Create the directional light
  ...
  // Create the transform group node
  mainTransformGroup = new TransformGroup();
  // Set the appropriate capabilities for the TranformGroup node
  ...
  // Set the appropriate capabilities for the main BranchGroup node
  ...
  // Add the main TransformGroup node to the main BranchGroup
  // This means the main transform group will be in charge of all the
  // transformations of the universe as a whole
  mainBranchGroup.addChild(mainTransformGroup);

  return mainBranchGroup;
}
```

Listing 3: Scene3D initialisation method

What we now have is a usable 3D environment. We can create Java3D visual objects, add them to the main BranchGroup node created, and they will appear in our canvas. At this point we must also create picking methods, to enable 3D visual object selection, navigation behaviours and user interaction behaviours. I will further explain these methods in §4.6.

4.2 The Update Handler

As described in §4.1.2, our intermediary framework is designed to provide us with updates containing IDebugObject objects. We have seen the interface for the IDebugObject in listing 2, and we have seen in listing 1 that we instantiate an UpdateHandler object, and pass it to this intermediary framework. What we propose, is that this UpdateHandler will receive, and process all update commands. We expect all representations of the underlying memory graph nodes to pass through this update handler. Hence, the update method is shown in listing 4.

```
public void updateDebugModel(IDebugTarget debugTarget,
    Map<DebugChangeType, List<IDebugObject>> objectsChanged) {

  //Reset all objects state
  ...
  // Create iterator variable used to iterate through the objects.
  ...
  // Check for new objects in the system
  if (objectsChanged.containsKey(DebugChangeType.CREATED)) {

    // Set our iterator to the objects which are NEW
    iterator = objectsChanged.get(DebugChangeType.CREATED).iterator();

    // Iterate through, sending each IDebugObject to the View3D object
    while (iterator.hasNext()) {
      IDebugObject itemp = iterator.next();
      view3D.createNew(itemp);
      // Set the state of these Object3D objects to NEW
      View3D.idoToObject3D.get(itemp).state = "new";
    }
  }
  /*
   * Iterate through the changed objects, no need to send them through to
   * the View3D object however, just set their state as CHANGED
   */
  ...
  /*
   * Iterate through all the deleted IDebugObjects, notify view3D of their
   * removal
   */
  ...
  // Having processed all objects, finalise view
  // First extract all the Object3D objects still in our system
  // We do this by accessing our static mapping of IDebugObjects to Object3Ds
  ...
  // If positioning depends on rank, update the rank and positions to
  // accommodate these changes
  ...
  // We then perform an update on each object
  for (Object3D o3d : totalListOfObjects) {
    o3d.update();
  }
  ...
 }
}
```

Listing 4: The Update Handler

Essentially, we have notified the View3D object, our View maintainer, of all the changes to the underlying system. We set the current state of each Object3D (Our 3D object representation detailed in §4.3), and then we perform an update for each Object3D, creating the new layout in the virtual universe.

Initially, I had decided to update each object as it was sent to the View model. However, as we discuss later in §4.5.3, our layout of these objects depends on each other, hence, we shouldn't perform any changes to the Object3D's representation, until all the objects in our model are known. Once all the objects have been passed through to the View, we know the system is stable once again, and as such we can re-calculate our layout in the virtual world.

4.3 The Object3D Class

Having briefly seen in §4.2, we use an Object3D class to store details of our visual objects. Each Object3D instance in our system represents an underlying node in the memory graph. As such, it must deal with positioning of the visual object, appearance and size, and attaching itself correctly to the Java3D scene graph. We also integrate within this class methods for generating name labels for the object, and methods for generating directed lines to other Object3D instances in the virtual world.

As this class is somewhat large, I will simply highlight and explain certain interesting methods below.

Object Positioning

As we will discuss in §4.5.1, we will use a layout manager to calculate the actual positions for each object. However, placing this visual object correctly in the 3D space is the job of the Object3D class. Each Object3D has its own BranchGroup Node which is directly attached to the main BranchGroup. This means that we can apply a transform to the TransfromGroup governing this node, in the knowledge that all Object3D's will have the same reference point. In other words, because each Object3D is at the same level in the Java3D scene graph, they each are given the same default location. This default location is unimportant, as long as the visual objects are placed correctively *relative* to one another. In this way, we translate the TransfromGroup for this Object3D by the vector given by our layout manager.

In order to allow for the dynamic changing of positions we may require for certain layouts, we query the layout manager each time we update the object, and update our vector position. However, transforming the same Transform-Group will result in moving that direction, from our current one. Clearly this is not what we want. Instead we create a new Transform3D object each time the object is updated, thus resetting to our default location.

General Appearance summary

In order to make the visualisations somewhat attractive, I decided to represent each underlying object in the memory graph as a sphere. Each object is then given a colour, relative to its state. Namely, green for new objects, orange for changed objects, and white for unchanged objects. Each object is given material attributes which can be set in Java3D. These consist of how the object reacts to different lighting. As we saw in listing 3, our 3D universe has a light source, and direction. Thus, we set our objects to utilise this, providing a nice texture, and a simple way to differentiate between object states.

Creating Name Label

Each sphere represents an underlying node in the overall memory graph, however, in order to differentiate between them, we can apply a name label to each object. This name object is actually a three-dimensional object in its own right; we create it by using the font extrusion class available in the Java3D API. We then place it on the edge of the sphere by generating a new TransformGroup and BranchGroup for this object. In the scene graph, the name label's BranchGroup would be placed as a child of the associated Object3D's node. This is constructed much like the Object3D themselves, in that, each name label object is given a default location, this time at the centre of the sphere. A simple transform moves them to the edge of the sphere. In essence, we have put the objects themselves in charge of their name objects, making positioning straightforward, and providing the user with the ability to differentiate between, and focus upon, certain objects of interest.

Creating inter-Object3D lines

In order to allow for the smooth addition and removal of lines between objects, we also put the objects in charge of any links they may have to other objects in the memory graph. There are two aspects to this problem, the first is that we must create lines joining the two objects being linked and the second is that we must be able to visually determine the direction of this relationship. The IDebugObject, as seen in listiug 2, provides a list of all references made to any other objects. We utilise this list to discover the necessary links, and as each link is directed, we can place the Object3D class in charge of maintaining these links when and if they are necessary. Essentially, we can be sure that if each object displays links for each of its references, then all the references in the underlying system will be displayed visually in the virtual world.

In order to tackle the first problem, we utilise the Java3D LineArray class. Assume the source object has position v_{source}, represented by a three dimensional vector. This vector represents the position of the object in relation to the centroid of the overall design space. We can extract a similar vector for each object referenced by the source object. As each Object3D maintains a vector position for its object, we extract a vector for each of these target objects, as an example, let us call it v_{target_i}. Where each i represents the various objects our source object may reference. We now have a list of $(v_{source}, v_{target_i})$ pairings.

As we are maintaining the lines of this object within the Object3D class, and hence, in the Object3D's own sub-tree in the scene graph, we must make a few further calculations. When we create new child BranchGroup and TransformGroup pairs for our Object3D items, they are given the Object3D's vector position as a root position. Hence, we must calculate the vector $v_{source} \rightarrow v_{target_i}$ for each i. To find the target objects position relative to the source objects, we simply subtract v_{target_i} from v_{source}. We can then create a new BranchGroup, and place within it our lines generated from points $(0, 0, 0)$ and $v_{target_i} - v_{source}$. This constructs a line joining the two visual objects in the virtual world.

In order to represent the directed nature of these lines, we must construct arrow heads. Given the three-dimensional world we are in, we represent these arrowheads using cones. Java3D has an inbuilt class for Cone creation, however, this class simply creates a cone of given dimensions. Its placement is the job of this Object3D class. The Cone class takes as arguments a base radius, a length, and an Appearance parameter. In order to further distinguish line direction, we create the appearance of the cone to match that of the source object. This makes it easily recognisable when we look at the multi-colour nature of our scene graph, without being over bearing.

Calling the Cone class as described, generates a cone positioned at the Object3Ds relative root, and orientated along the y-axis of the 3D environment.

In other words, this Cone will appear at the centroid of the Object3D's sphere, pointing along the positive y-axis. In order to position it correctly, we must perform some vector manipulations. Firstly, we rotate the object, and then we translate the object.

The rotation involved is calculated using the knowledge of orthogonality in vector spaces, such that:

$$angle(x, y) = cos^{-1} \frac{< x, y >}{\|x\| \cdot \|y\|}$$

The inner product represented by $< x, y >$, is in fact calculated using the dot product in three-dimensional space. We can use this equation to calculate the angle between our current cone orientation, and our desired cone orientation. Notably, we consider our cone as pointing towards (0,1,0). And our target object as pointing towards $v_{direction_i}$, where $v_{direction_i} = v_{target_i} - v_{source}$. This allows us to calculate the following:

$$\theta = cos^{-1} \frac{(0, 1, 0).v_{direction_i}}{\|v_{direction_i}\|}$$

This provides us with θ, the angle between our two vectors. What we next need is the axis of this rotation, this is in order to actually rotate the cone. In order to calculate this, we normalise our target object direction vector, and calculate the cross-product of the two:

$$v_{direction_{norm_i}} = \frac{v_d irection_i}{\|v_{direction_i}\|}$$

$$v_{rotationalAxis} = (0, 1, 0) \times v_{direction_{norm_i}}$$

This provides us with a vector perpendicular to both vectors, perfect in providing the rotational axis for our transformation. In the event that θ is collinear, in other words, either 180 °C or 0 °C, the cross product will give us 0. Hence, in this case we check to see if θ is 180 °C, and if so, set the axis direction to (1,0,0). Otherwise, the angle is 0, and hence the axis of rotation is unimportant.

With the axis of rotation calculated, along with the angle of rotation, we simply perform this rotation on the code object. We then translate our now correctly pointing cone so it sits on the edge of the target object. This process is repeated for all i, such that each target object which our source object references has an associated line and arrow head.

4.4 Maintaining the Java3D scene graph - The View3D Class

As we have seen, we delegate much of the visual control to the Object3D and LayoutManager classes. However, what we must ensure is that we maintain a correctly formed Java3D scene graph, and keep track of all the Object3D instances in our current program. This job is performed by the View3D class. This class has the job of creating each Object3D instance, and placing it correctly in the scene graph. It provides a static mapping of IDebugObject objects to Object3D objects. This is done in the form of a HashMap.

The View3D class could be considered as the hub of this program. It keeps track of all the objects and the scene graph, communicates with the Eclipse framework (as seen in §4.1.1), and as we will see in §4.6, it handles all of the user interaction methods. The actual details of this class are somewhat trivial however, and as such, I will refrain from going into much detail. I will give more information about the user interaction aspect later in this report, and the full listing of this class is available in the appendix.

4.5 Managing Different Layouts

This section will deal with the positioning of our visual objects and how we can use a layout manager abstraction to help deal with this problem. We will then discuss how using an importance measure can allow for a more advanced layout system, which bases positioning on importance. In §4.5.4 we outline such a layout technique, and discuss its usability. In §4.5.6 we continue to look at how importance can affect us, but propose that items referencing each other should be positioned together. Hence, we outline a different algorithm, and draw comparisons about the two. Finally, in §4.5.7, we look at how to create forward and backward traces as discussed in §1.1.2.

4.5.1 Creating a Layout Manager

Each Object3D in our system looks to gain information about its position vector, from a dedicated layout manager. Seeing as one of the most interesting aspects of this project is the positioning of our visual objects in the 3D environment, it seemed only sensible to separate concerns, and create a layout manager interface for which any layout manager must extend. Our layout manager interface contains only two simple methods:

```
/**
 * This class serves as a controller for the positions of each Object3D in the
 * system
 *
 */
public interface LayoutManager3D {

   /**
    * @param o3d
    *              the Object3D we want the position of
    * @return A three dimensional vector representing it's position
    */
   public Vector3d getPosition(Object3D o3d);

   /**
    * This method tells the Layout Manager to reconsider its position values
    * We call this method when they underlying model changes
    */
   public void updateAllPositions();
}
```

Listing 5: The Layout Manager Interface

As we can see, we ouly expect our layont manager to respond to Object3D instances querying the layout manager for their position, as well as notifications that the underlying model has changed. However, as we will see, the more complex the layout gets, the more work it has to do behind the scenes. I will now discuss the layout managers implemented in the system, and the increasingly difficult challenges faced as more information regarding the underlying model is used.

4.5.2 Simple 3D Layout Designs

Initially, we consider layout managers for which a bare minimum of information from the model is extracted. Essentially, they just collate a list of Object3D instances in the system, and generate a position for each. Two such implemented layouts are called the GridLayout, and the StackLayout.

Grid Layout

The GridLayout manager simply creates a mapping of Object3D instances, to three-dimensional vectors. Each time an object asks for its position, if this object is in the mapping, we return its associated vector. Otherwise, we generate a new position in a grid-like fashion. We start out at (0,0,0), and each time increase the x position by the size of the object and some space. When the width of the view has been filled, we reset the x position to 0, decrease the y co-ordinate, and continue as such. An extremely simplistic method for filling the screen with objects.

This provides a very simplistic view of all the objects in the system. We essentially show the order in which objects are provided to the model, and not much else. It makes it very easy for a user to see how many objects are in the system, and their names, however, when we show the links between objects, this model does not fair so well. We also fail to utilise the third dimension available to us.

Stack Layout

In order to utilise the third dimension, we act as before, but increase the z co-ordinate each time the screen is filled. In other words, we create a stack of grid patterns. This again, is very simplistic, and simply provides the user with a time-line of objects. It makes it very difficult to do much else, especially when considering links between objects.

The two views discussed work as a general view layout. They are designed to be as simple as possible, and provide the user with a clear representation of the underlying system, even if such a representation rarely gives new insight into the program. However, they demonstrate the ability for the layout manager to abstract away from the intricacies involved in the Java3D model. We simply keep track of a set of three-dimensional vectors, nothing else.

These two layouts are also static, once an object has a position given to it, it is set. Hence, we don't need to make use of the update method, it simply has no effect. We will see as the views get more complex however, that such notifications become necessary. These two views are designed to give an extreme example of how simple the layout manager can be, but it also aims to show that the layout of the three-dimensional objects will determine the success or failnre of this three-dimensional view.

4.5.3 Ranking The Objects

In order to improve upon our simple layout designs, we must increase our knowledge of the underlying systems, and use that information in constructing our layout. One seemingly useful way to do this is to calculate a rank for each object. The underlying framework provides methods for us to do this, in fact, it utilises a system much like PageRank; an algorithm assigning rank based on the hyper-link structure of the web [5]. However, instead of links to other web pages, we consider links to other objects. As such, we can call the page rank method for each IDebugObject, and get an importance score for each visual object. With the ability to assign a score to each object, we can design our layouts based on this scoring system. We propose that the higher the objects importance, the more interest it poses to the end user.

4.5.4 Divide and Resize Algorithm

Our aim is a design which utilises both the three-dimensional properties of our system, as well as extracting information from the underlying rank of our objects. The algorithm I propose here makes use of the objects rank to determine position, and utilises the extra dimension available. The idea is that the most important object, should be at the centre of our focus. As the importance score drops, these objects should move away from our focus. In order to do this, I propose a system which utilises both the objects size, and its position to visualise its importance. The proposed algorithm does the following:

1. Rank all the objects, and place them into an ordered list.

2. Extract the first item, place it at the root, set its size parameter as the largest object you will want in the graph, and the bounding sphere the size of the view we are working with.

3. Create a set of 5 or 6 lists, depending on the root nodes origin. We transfer all the remaining items in the ordered list, incrementally, into the sub-lists. In other words, the first list gets the second ranked object, the third gets the third ranked, etc. Until the original list is empty.

4. We then go back to step 2 for each list. However, we move the root position out in all 6 directions, halfway to the edge of the bounding sphere, from the current root node respectively for each list. We also half the size of the bounding sphere we are allowed to work within, and we half the size of the object node. When we are past the first iteration, we only create 5 lists, as we don't send any objects back in the direction they came from.

This is performed by the following two methods:

```
private void createRankedListOfObjects() {

  // First extract all the Object3D objects still in our system
  Collection<Object3D> totalListOfObjects = View3D.idoToObject3D.values();

  totalRankedListOfObjects = new LinkedList<Object3D>(totalListOfObjects);

  // Sort the collection based on rank
  Collections.sort(totalRankedListOfObjects,
```

```
        new Comparator<Object3D>() {
          public int compare(Object3D arg0, Object3D arg1) {
            double diff = arg0.ido.getPageRank() - arg1.ido.getPageRank();
            if (diff > 0) {
              return -1;
            } else if (diff < 0) {
              return 1;
            } else {
              return 0;
            }
          }
        });
    // Save this total object ranking
    currentRanking = (LinkedList<Object3D>) totalRankedListOfObjects.clone();
  }
```

Listing 6: Ranking The Objects

```
private void createPositions(Vector3d root, double radius, int cameFrom,
      LinkedList<Object3D> rankedListOfObjects) {

    // Place root node in position
    idoVectorMap.put(rankedListOfObjects.removeFirst(), root);

    // Create sub-lists
    ...
    /* Divide List up into 5 or 6 depending on cameFrom location */
    int i = 0;
    while (!rankedListOfObjects.isEmpty()) {
      switch (i) {
      case 0:
        i++;
        if (cameFrom == 0) {
          break;
        }
        ll0.add(rankedListOfObjects.removeFirst());
        break;
      case 1:
        i++;
        if (cameFrom == 1) {
          break;
        }
        ll1.add(rankedListOfObjects.removeFirst());
        break;
        .
        .
        .
      }
    }

    // Create positions for the sub-lists.

    if (cameFrom != 0 && !ll0.isEmpty()) {
      createPositions(new Vector3d(root.getX() - radius, root.getY(),
          root.getZ()), radius / 2, 1, ll0);
    }
    .
    .
    .
  }
```

Listing 7: Method for Creating Positions

The main point to highlight here is that we can keep in view any number of objects, and yet maintain a constant sized space. We make sure our most important object is the focus of attention, and we ensure that focus draws away as the importance lessens. In constructing our sub-lists, and hence, direction of spread, we maintain the order inherent in the list, and hence, we do not need to worry about sorting for the sublists. This saves dramatically on the complexity of the algorithm.

This system provides a very usable overview of the underlying disconnected memory graph of our objects. Importantly, this algorithm maintains focus on the important objects, whilst removing clutter around them. It does this by not creating a sub-object space, where the object just came from. Hence, objects aren't placed crowding the important objects. This view seems to be an ideal way to represent the memory graph, whilst maintaining usability, and increasing the number of objects in the screen space compared to a 2D design. An example is shown in figure 3.

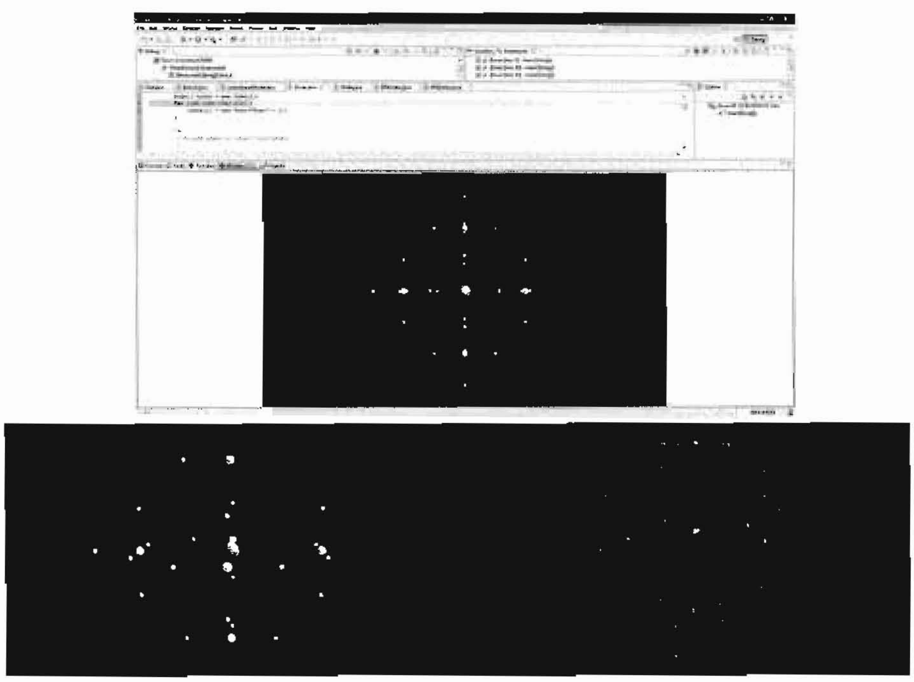

Figure 3: The Divide and Resize Algorithm in use.

4.5.5 A Different Approach to Determining Object Size

As we have seen in the Divide and Resize algorithm, the visual objects size can play a vital role in the usability of the general layout. The halving method employed in the divide and resize algorithm seems rather naive, even if it works well visually. Given that the model has access to an importance score for each object, it would seem nonsensical for two objects to be of the same size, when one is vastly more important than the other. Hence, I suggest a sizing algorithm based solely on the importance of the object. In the Divide and Resize algorithm of §4.5.4, the pattern of decreasing size will still exist by definition of the way the objects are positioned, however, the size may now be deemed as having more relevance.

This solution will also work with any layout manager pattern, regardless of whether it uses importance in positioning. For example, our naive grid and stack methods will instantly be more usefnl with such an implementation. Hence, I now model the objects size as a function of its importance. This is done by taking the highest scoring object, setting that at an acceptable size, and calcnlating every other object's size as a ratio of this size, corresponding to the ratio in importance scores between the two objects. We shift the available range such that every object in the system will be visible, generating a minimum value. Thus, every object's size lies between our minimum, and the size of the most important object, determined by its relative importance.

4.5.6 Clustering Method

As we have discussed thronghout this section, onr aim is to draw upon information in the graph strncture, and present this information as well as possible in the layout of our 3D environment. Following on from the Divide and Resize algorithm in §4.5.4, we build an extra layer of information. What we now utilise is the fact that in order to keep the 3D graph as 'tidy' as possible, it would be preferable to keep all similar items close together. Drawing upon knowledge from computational linguistics, we can apply the 'Distributional Hypothesis' [10]. In linguistics, this refers to gaining knowledge regarding a single word from the company it keeps. We apply this to the object model by drawing upon the knowledge of referenced nodes, in order to define the positioning of a single node. We essentially cluster groups of objects together. Thus, our disconnected graph is divided np into its connected sections.

In order to achieve this, we use the framework of the Divide and Resize algorithm, however, when producing our sub-lists, instead of distributing on importance alone, we distribute on the context of the objects. In other words, we put all objects in the same connected graph, into the same list. We can perform this creation of groups of connected objects, by iterating through all the node points in our current subgraph, iteratively calculating the references it contains as we go. We ensure that each sub-group still maintains its importance order however, an important feature of this algorithm.

The other main difference between this algorithm and the divide and resize algorithm is that we remove the space requirement. We no longer keep all the objects within a predetermined sphere of 3D space. In essence, we allow the graph to grow outwards in all directions. In order to do this effectively we always allow our objects to move away from the centroid, once we reach a point where the subgraph is fully connected, we apply the divide and resize algorithm as before.

What this provides us with is a simple solution to the problem of overcrowding and crossing of links between different parts of the program. We now separate out the different memory graphs, and provide an extremely user-friendly approach to dealing with the disconnected nature of the overall graph. In essence, we maintain the most important object as the centroid, and cluster the graph based on our reference context measure. We then apply this iteratively to each of the subgraph's most important objects.

Figure 4 shows us the view this algorithm achieves. Figure 5 shows a side by side comparison of the two layout managers, showing the added detail brought in by the clustering model. It also shows a midway step, whereby we have clustered the initial group, before finally showing the result of doing this iteratively for each sub-group. It must be noted however that due to the added complexity of this algorithm, our code is no longer quite as efficient. A more detailed explanation and the effects of this are detailed in §5.2.

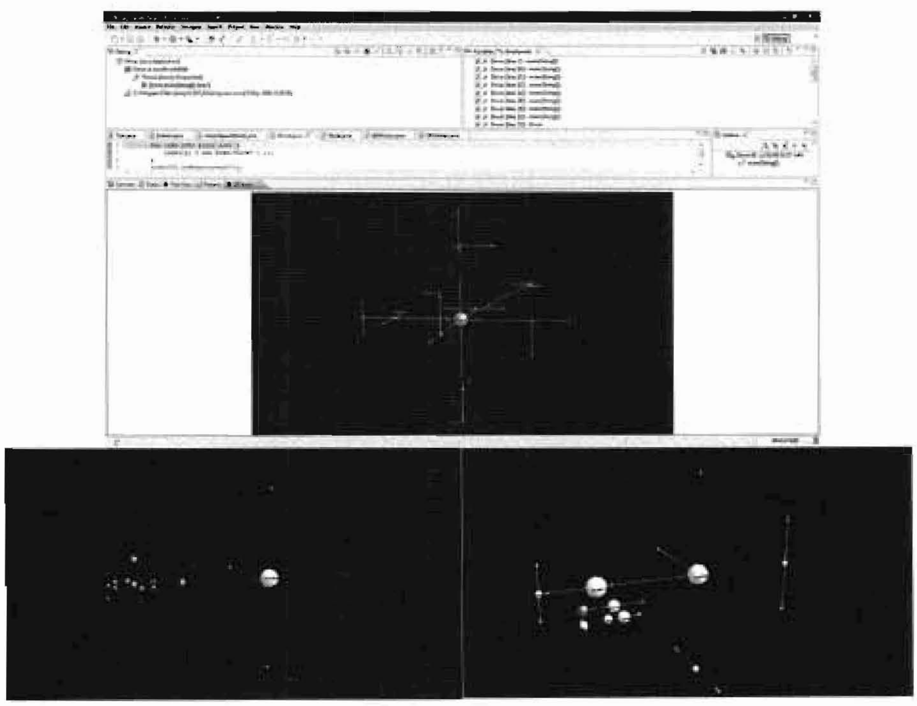

Figure 4: The Clnstering based layout manager in use.

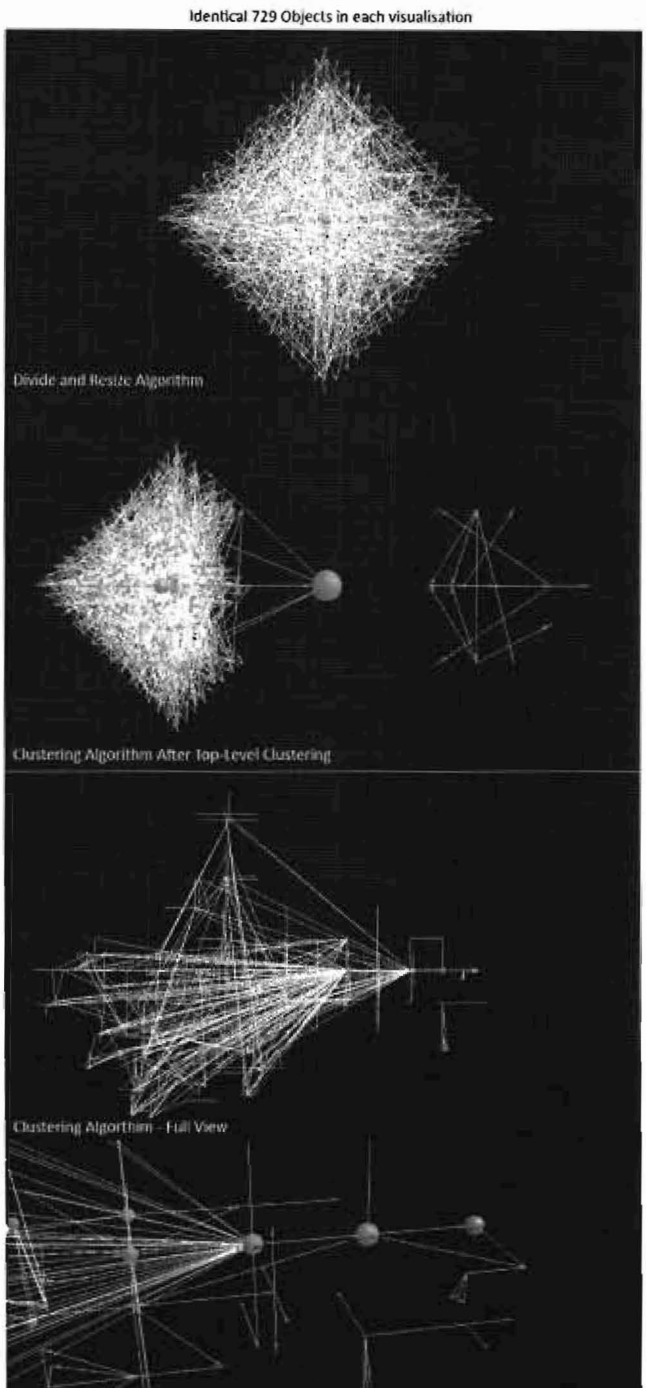

Figure 5: A side by side comparison of the two algorithms.

4.5.7 Generating Forward and Backward Traces

In §1.1.2, we discussed the notion of forward and backward traces. In our model this is essentially the equivalent of following all the forward or backward links from an object, and drawing the tree representing that link structure. We give the user the option of selecting the root node from our generalised view, and as we have constructed our generalised view based on importance, we know that the most links will be found using the most important node.

Firstly, the user selects the node they would like to act as the root node. All the objects in the scene graph are then removed, and we call a tree generation method in the associated Object3D instance of the selected root node. This is performed by the method shown in listing 8, located in the View3D class.

```
public void createTrace() {
  Object3D tempo = currentRightClickedNode;

  Collection<Object3D> c = idoToObject3D.values();

  // Clear the scene graph
  for (Object3D o3d : c) {
    mainTransformGroup.removeChild(o3d.getBranchGroup());
  }

  /*
   * Signify which object is the root. We need to know this for further
   * right click events.
   */
  currentRootNode = tempo;

  // create tree layout for objects.
  if (traceDirection == 0) {
    // Create forward trace
    tempo.createCurrentTree();
    tempo.displayObjectLinks();
  } else if (traceDirection == 1) {
    // Create backward trace
    tempo.createCurrentBackLinkTree();
    tempo.displayObjectBackLinks();
  }

  // We only want to highlight the root node!
  tempo.highlightCurrentObject();
}
```

Listing 8: Creating the Trace - View3D's role

The Object3D instance for the root node then begins to construct the tree, it simply iterates through all the objects it has links to, and the objects those objects link to, and re-creates them in the scene graph. The positions of the visual objects are calculated by an associated tree layout manager, which each Object3D instance accesses. The Object3D instance then draws the directed lines connecting the graph, including backlinks. As backlinks are possible, we have to keep track of the objects we have seen, this makes sure we don't attempt to create an already visible object. For a forward trace, the associated method is shown in listing 9.

```
private void createSubObjects() {

  // Remove any lines if they are currently on display
```

```
  if (linesVisible) {
    removeLines();
  }
  // Remove the TransformGroup for this Object3D.
  bg.removeChild(tg);

  // Get position from the tree layout manager and set
  // that position for this Object3D
  Vector3d pos = treeLayout.getPosition(this);
  ...
  // Create object, now based on its new trace position
  createObject();

  // Add newly updated BranchGroup to the scene graph
  view3D.mainTransformGroup.addChild(bg);
  // Restore details if they were visible
  ...
  // Create local map for this Object3D's links
  Map<IDebugObject, IVariable> linklist = linklist = ido.objectLinks();
  ...
  // We add this ido to our seen list, ensuring we don't try to create it
  // again
  seenObjectList.add(ido);
  ...
  // Iterate through object links, creating each object
  for (Entry<IDebugObject, IVariable> variableLink : linklist.entrySet()) {
    IDebugObject i = variableLink.getKey();
    if (i != null && !seenObjectList.contains(i)) {
      // If we haven't seen this object yet, search it.
      View3D.idoToObject3D.get(i).createSubObjects();
      seenObjectList.add(ido);
    }
    // Draw lines from this Object3D to each of it's children
    createLines(this, i);
  }
}
```

Listing 9: Creating a forward Trace - Object3D's role

We now just need to discuss the layout manager's construction of the traces. This is a standard tree drawing problem. What we perform is a Breadth-First search of the tree, starting at the root node, calculating the required space of each sub-tree. This is a single pass of the tree structure where we remember seen nodes in order to handle backlinks and self-referential objects. This creates a mapping of Object3D nodes, to their associated subtree size. We then begin once again at the root, and knowing the size required for each sub-tree, allocate the space accordingly on a level by level basis. In other words, we create a list for each level of the tree, and then draw each level at a time. In order to deal with forward traces, we look at the references from the respective object, and generate the tree in the negative y-axis. In contrast, for the backlinks structure, we look at objects that point to the respective objects, and create the tree in the positive y-axis. The result of this trace drawing algorithm can be seen in figure 6.

What this algorithm provides is a guarantee that all the objects will be drawn correctly, and no overlapping, or ill-placement will occur. We know through our subtree size calculations how much space each subtree reqnires, and it is the use of this fact which allows us to draw our graph in a beautified and clear manner. We are able to position each node, with the advanced knowledge of the number of nodes we need to place below it.

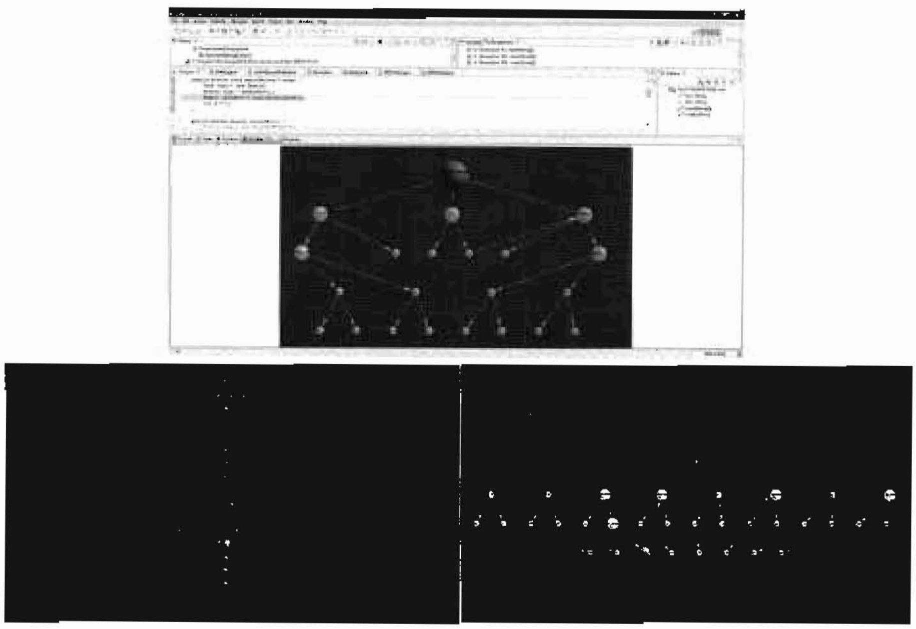

Figure 6: A look at forward and backward traces.

4.6 User Interaction

In order to make use of the various optional views our plug-in provides, we want to provide a simple way for the user to interact with the 3D environment. As we have seen in §4.1.3, we instantiate Java3D mouse rotate, mouse translate, mouse wheel zoom and keyboard navigation behaviours. This allows our users to move around the virtual world with ease. It should be noted that the rotation and translation methods have been created on the Object3D branch group node, and hence, physically move the objects in the 3D space as one whole. On the other hand, the zoom and keyboard navigation behaviours have been created on the view side of the scene graph, and hence, move the perspective of the user. It is this solution that best suits the needs of the user, providing a very intuitive way to move around the 3D universe.

Having generated ways to manipulate the view of the virtual world, we must look at a way in which we can directly affect the underlying structure. The methods I provide are detailed in figure 7. However, in order to provide these methods we must discuss a few more Java3D requirements. Firstly, we have available to us a Java3D picking class. Essentially, we subscribe and implement the View3D class as a mouse listener for the Java3D canvas. Then, when mouse events arrive, we can qnery the picking class to find out which 3D object lies at the current monse point on the canvas. We then create our menus accordingly.

As we can see in our Eclipse, and further two close-up 3D environment screenshots in figure 7, menu creation depends on the state of the view. In other words, we separate the user from the idea that a forward or backward

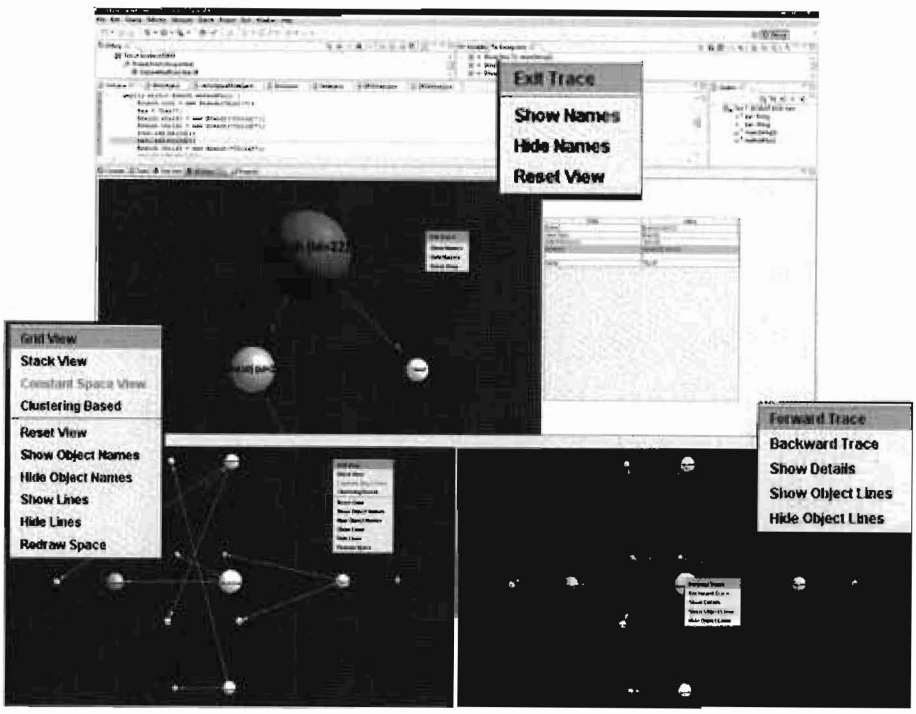

Figure 7: The menu options available

trace is simply another layout; the user can switch between the main overall view, and any trace, seamlessly.

What I would also like to highlight from figure 7 is the small table to the right of the 3D view, in the Eclipse window. If a user selects to "show details", then this table emerges, showing all the information we have about the object. These include its name, its type, any variables it has, and any objects it references. These are all extracted from the IDebugObject getValue method. This extra information can provide the user with added debugging opportunity, as well as extensibility in the project as a whole. The JDT debugger offers the changing of live values, if our underlying framework can cope with this, then our view can provide a simple way to change the values of variables in an object, on-the-fly. In fact, our table implementation is capable of exactly this, however, the underlying model currently in use doesn't allow for that to occur.

4.7 Overall Design View

Having discussed the working, and some of the interactions of the classes involved, along with the Java3D scene graph, it is sensible to provide graphical illustrations for both. The UML diagram represents the Java classes I have implemented, however, it simplifies the intermediary debug model framework, which I haven't created. This is shown in figure 8. The Java3D scene graph represents the graph structure I have generated, which is renderable by the Java3D renderer. It follows the rules and conventions outlined in §2, and is shown in figure 9.

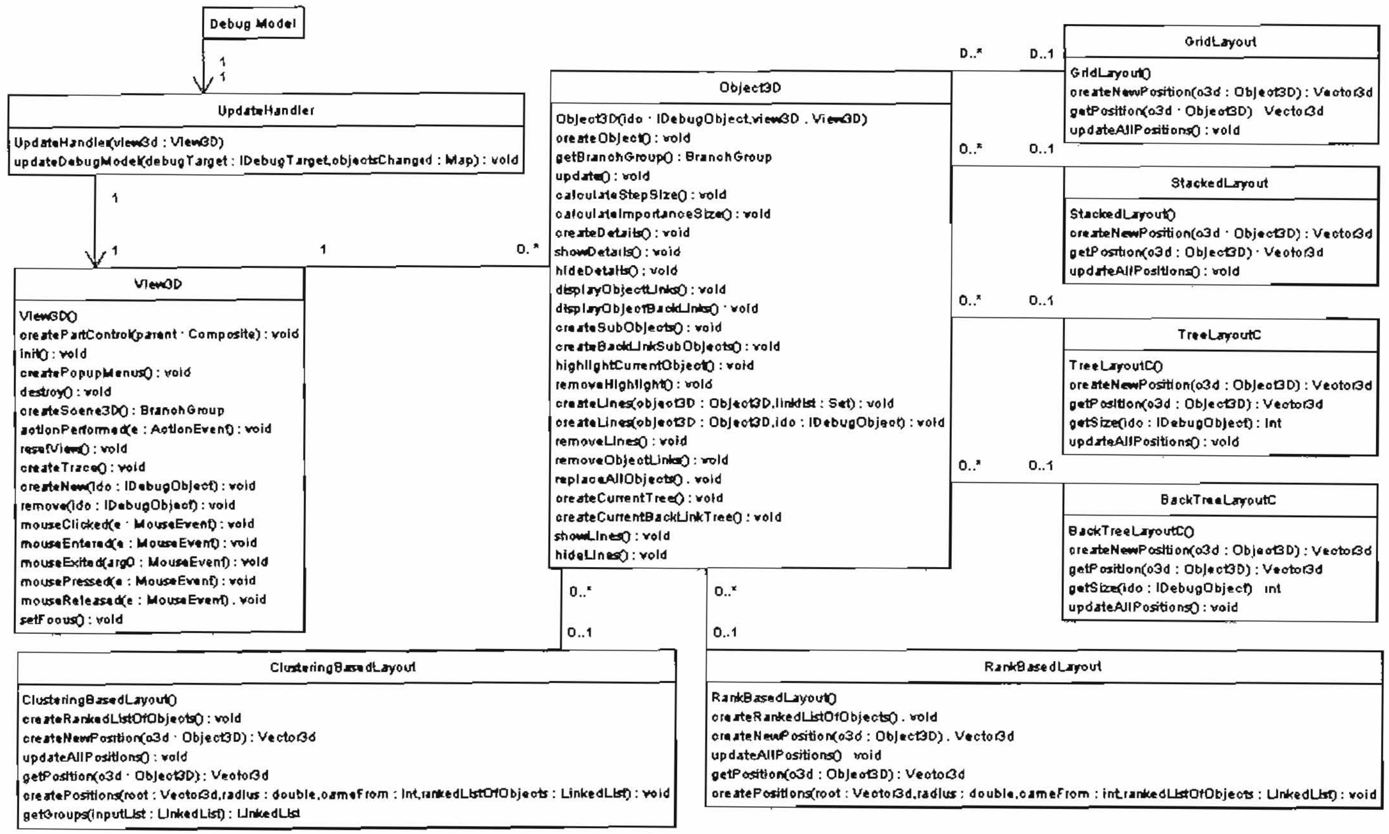

Figure 8: A UML representation of my design

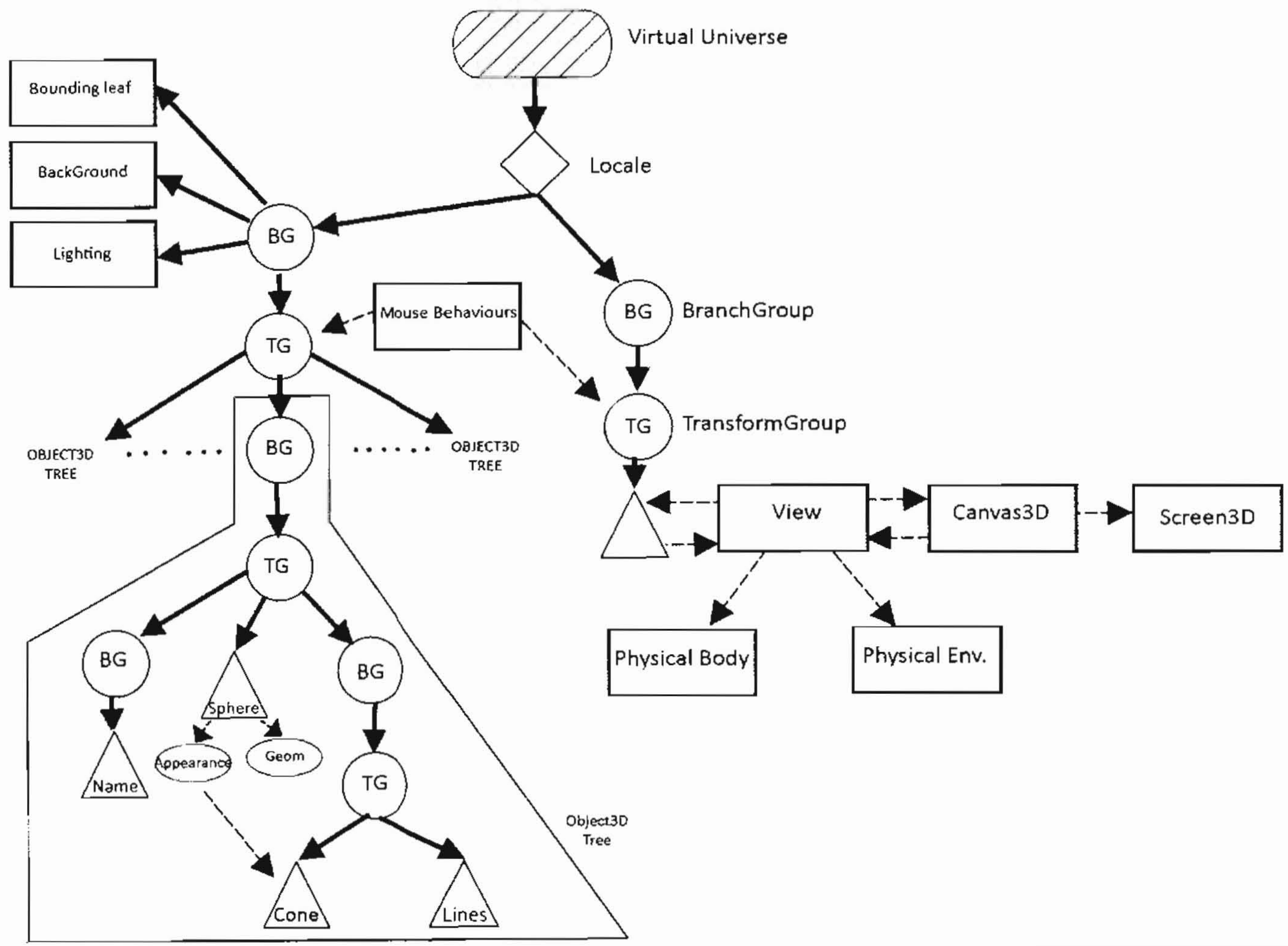

Figure 9: The Java3D scene graph structure of the final code

5 Testing

We have already seen some screen shots of the working program, however, we provide two stringent tests for our program to ensure it works as intended, along with a test rig to fully analyse the program. In both test programs, I will run through the whole series of options available to the user, and ensure its correctness. However, I will also demonstrate its ability to visualise code, and hopefully provide valuable insights whilst debugging.

5.1 Simple Program - BFS and DFS using the Visitor Pattern

This test program begins by creating an underlying tree structure. This tree structure is represented by Node objects, and the links between them. We construct a tree which has the following representation:

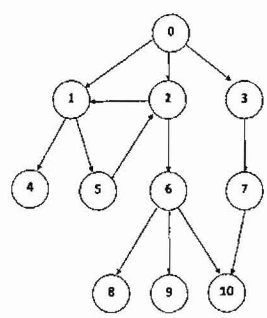

Figure 10: The tree we want to perform BFS and DFS on

Our test program begins by creating this structure, and then performs a DFS, followed by a BFS, both using the visitor design pattern. Within Eclipse, we set a break point after the last node has been generated, this provides us with an object state as in figure 11:

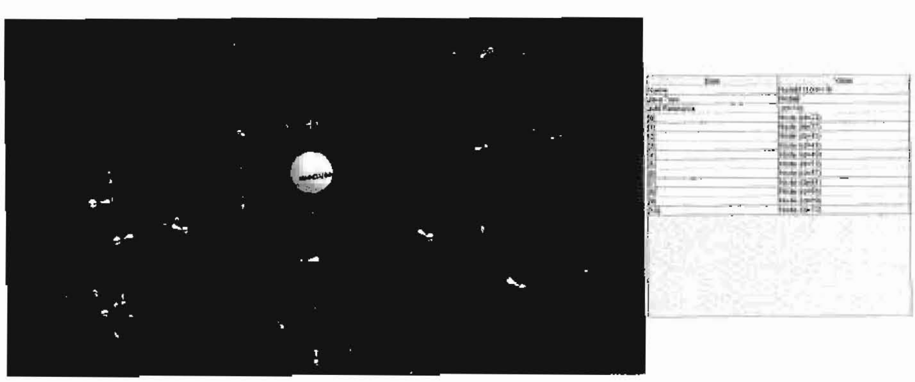

Figure 11: The main view

This displays the most important object, as the array holding all the Node instances. In order to see the representation of our underlying model, we simply request a forward trace on 'Node0'. This trace is shown in figure 12.

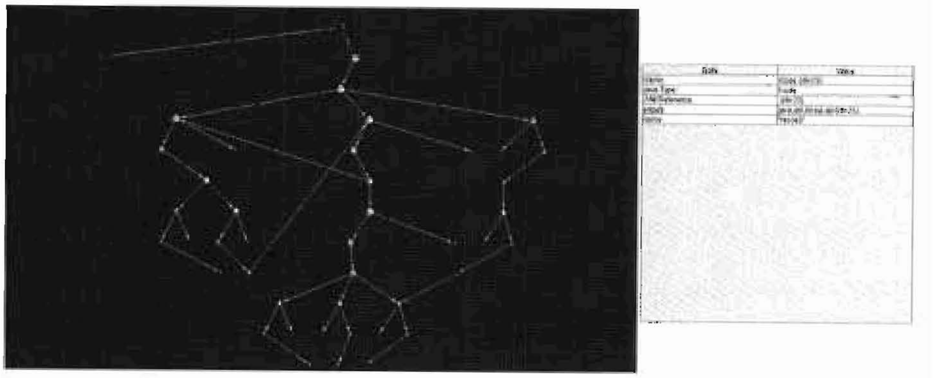

Figure 12: Forward trace of the Node0 object

Looking more closely at figure 12 uncovers a few interesting facts. Namely, we see the representation of the actual node is a Node object, pointing to a String object, the variable name, and the list containing its pointers. Hence, figure 12, is a direct representation of the underlying tree from figure 10. Looking at the backwards trace of Node10 also provides us with what we would expect. This is shown in figure 13, and shows us that the array storing all Node instances points at Node10, and drawn the expected tree resulting from that.

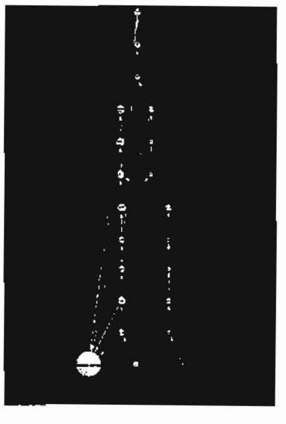

Figure 13: Backward trace of the Node10 object

Now let us imagine there is a bug in the code, and we can't understand why the output from the DFS and BFS is incorrect. Given figure 14, and the representation shown, it's clear that our intended tree isn't being created. We can see that one of the nodes isn't attached properly in the tree construction method, namely because there are two nodes with no *incoming* links. Further investigation shows us that this is Node5. Low and behold, Node5 was never added to the edges of Node1 in this run.

This kind of debugging is intuitive, and simple to do within this framework. If you have an intuitive understanding of what the underlying model in your program should look like, it is fairly straight forward to spot bugs like this in

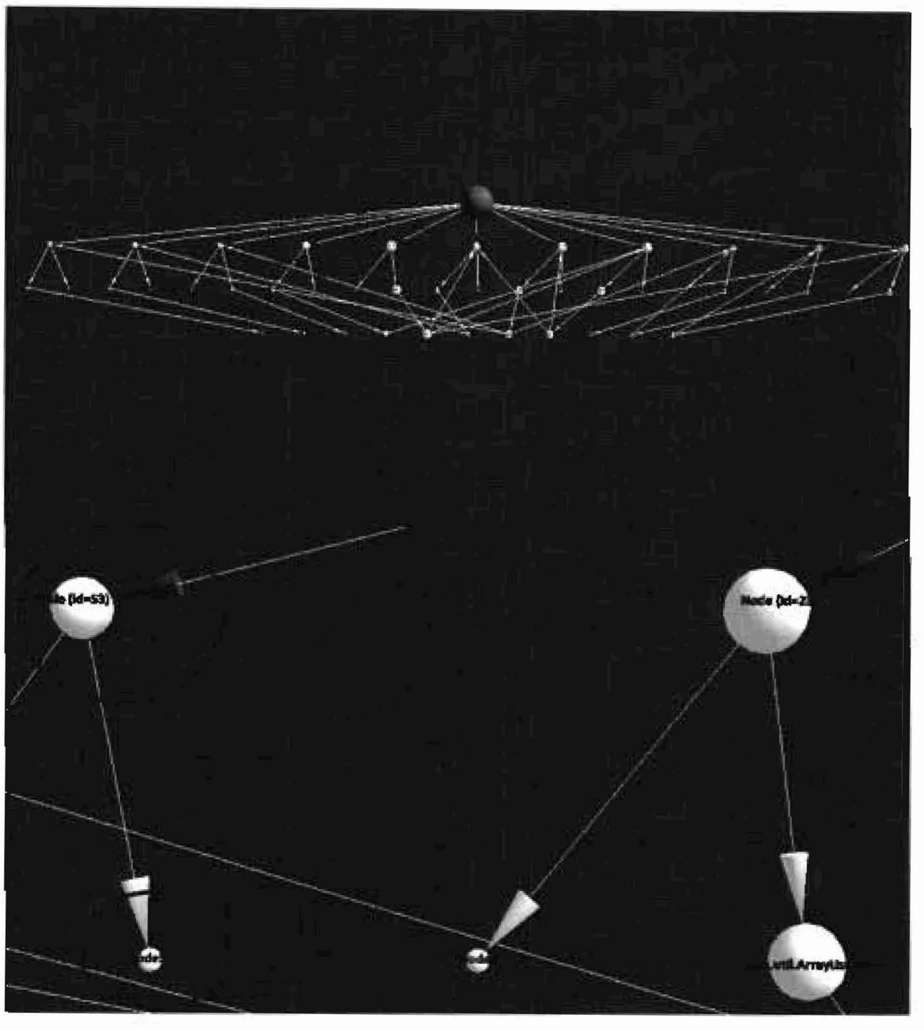

Figure 14: A bug in the code

small code samples. Assuming a larger program is in use, the user must delve a little deeper into the part of the graph which they suspect the bug to exist in. This is obviously heavily aided by the JDT debugger itself. However, this test still shows the usability of the code in a small program, and shows that the code can cope with the different types of back links and cross links that can occur in a memory graph.

5.2 Complex Program - Vector Space Document Retrieval Model

In order to test the usability of our code on a more realistic example, we evaluate its ability to cope with a much larger program. Namely, a document retrieval system which is based on the vector space model. This system generates hundreds of objects, makes use of large data structures, and is generally quite computationally expensive.

The program looks to analyse an inverted file index for a set of 2,631 documents. This index consists of each word, its document frequency, and a list of document, term frequency pairs for each document which this term appears in. This information is then used to retrieve relevant documents, given a query. In order to do this, the program makes use of various data structures. It uses mappings of terms onto document frequencies, terms onto lists of (document,frequency) pairs, and documents onto their document lengths. It also creates a sorted set of document scores to provide a ranked set of results. These mappings are created from the inverted file index, and then used to create the scores for each document given a query.

Given the program structure provided, let us see how our program deals with its visualisation. Firstly, we look at the initial creation of the maps, and how they are presented in the 3D space. This is shown in figure 15. Figure 16, shows the state of our program once the data structures have been filled. Unfortunately, in filling these data structures, the underlying system seems to become overloaded. So much so, that it stops communicating with the update handler (outlined in §4.2). As such, it isn't possible to push the 3D world to its limit in this system. This is unfortunate, but we will see in §5.3 that our program can in fact cope with many more objects.

The visualisations we can achieve initially show us our document retrieval system generating 17 objects (Figure 15). These objects consist of the instantiation of the vector space model itself, the maps we discussed and their components, and a set. Initially, all of these sets are empty. What we see is the minimal number of objects required in setting up these structures.

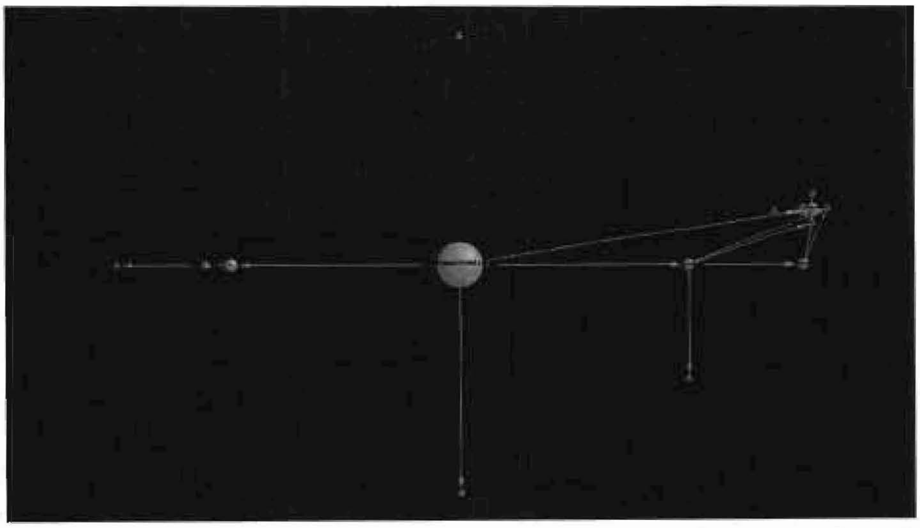

Figure 15: Initialising the data structures.

Figure 16 then shows us the structures as they begin to fill, including the relationship the vector space model component has with them. What we notice is that the InputStreamReader object used to read the inverted file index has its own space in the universe, concerned with reading the file. The data structures in our vector space model object then grow as more words are read and processed from the inverted file index.

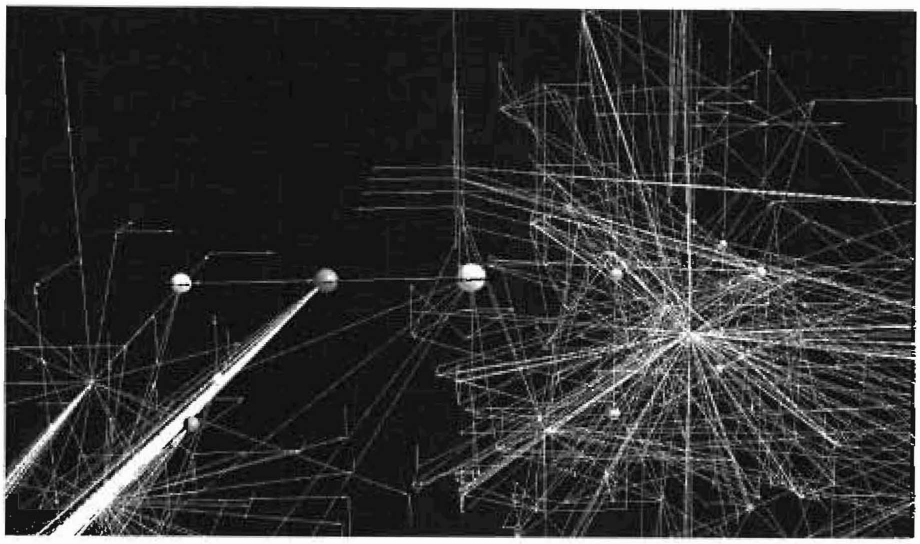

Figure 16: Filling the structures with the data. (3375 objects)

No. Of Objects	Time to Update View	Lines Creation	$s(object)^{-1}$
17	0.051s	0.018s	0.003s
730	0.28s	0.357s	0.000489s
1066	0.255s	0.208s	0.000239s
1975	0.868s	0.519s	0.000439s
2026	0.94s	0.298s	0.00046s
3051	2.117s	0.398s	0.00069s
4250	4.634s	1.09s	0.00109s
5527	8.798s	1.507s	0.00159s

Table 1: Analysis of growth

As discussed, this system can visualise around 5,000 objects before memory issues in the underlying system pose a problem. Timings for the growths can be seen in table 1. What we see is an expected growth regarding creating the visualisations, namely, that our system is not linear.

Empirically, we have seen that our system is not linear, in fact, results would lead us to believe that the program is $O(n^2)$. Doubling the number of objects, roughly quadruples the time taken. Looking at our system, we see that each iteration results in a sort of the entire collection, namely at a cost of $O(nlogn)$. This however is not our biggest computational task. In fact, our clustering algorithm, whilst iteratively removing a node from a cluster, and re-clustering, performs in the worst case n(n-1)/2 iterations. This in fact involves n^2 comparisons, as at each iteration we must look to see if the node has been seen before. Thus, if at each step we cluster into only two groups (the worst case), we only reduce the size of our search space by one at each step, this costs $O(n(n-1)/2)$, which is equivalent to $O(n^2)$ and is the most complex algorithm in use. Hence, the main contributor is the clustering algorithm, but as we see from the timings, our system is still extremely usable.

We now look at figures 17 and 18 to see how useful our program can be in potential debugging, and aiding understanding of such a large program. As we have seen, figure 16, shows us how the system is separated into two overall sections. One for our vector space model, the other for the file reader. Delving deeper into the vector space model, we see in figure 17 that our vector space model object has references to six other objects. This includes, four maps, one list, and one set. Exactly what we would expect here, given our program construction. As we can see, verification of this is extremely straight forward.

From the same view, namely figure 17 we can also see that two of the maps seems to have a much higher importance that the rest. Figure 18 looks in more detail, telling us that the most predominant mapping is the mapping of documents to strings. Second to that is the mapping of terms onto lists of (document,frequency) pairs. These two mappings are what make up the majority of our 3D space, and as such, take up a lot of the memory in the running of the program. This information could be crucial to a designer, showing how the program operates, and visualising the problem of potentially repeated data. In this case, it might be possible to combine information, and provide a more efficient map structure.

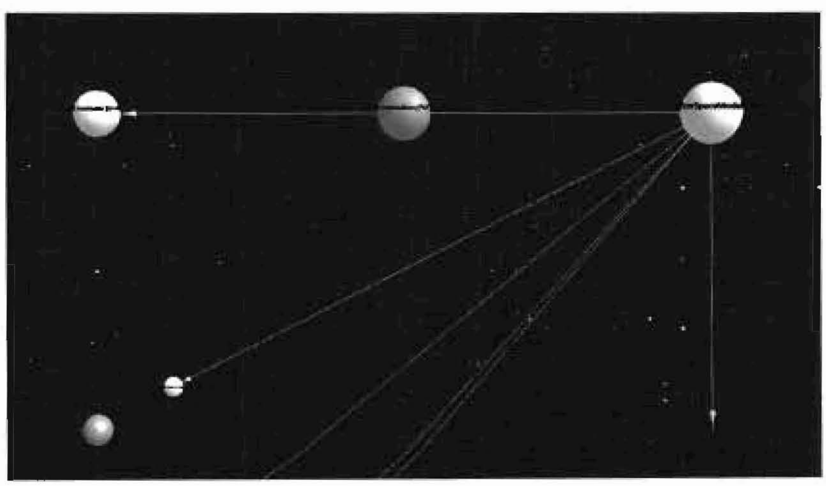

Figure 17: The VSM Object

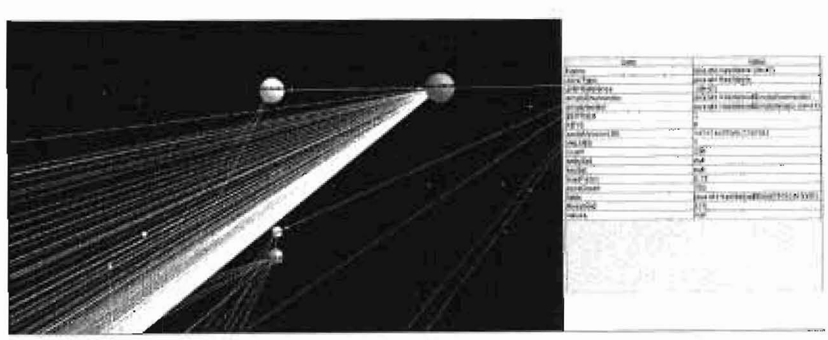

Figure 18: Detailing the important maps

5.3 Test Rig

In order to have a sustained and thorough testing available through the creation of the program, we make use of a test rig. When the testing mode is switched on, through a flag in the source code, at each step of the update our system provides timings, and performs every operation available to the user. This means activating every menu option, and hence, testing each method in the program. The program is fairly straight forward, and timings can be seen in the previous section. Suffice to say that upon completion, the test rig runs through cleanly, and with no errors on all of our test programs.

For completion and accuracy, table 2 shows us our object *creation* timings from the test rig. In order to do this we create dummy IDebugObject objects, each with a pseudorandom importance value, and they are all sent through to the system as new, at once. Therefore, this demonstrates the time taken to create the objects, calculate their position, and to display them. Figure 19 shows us the view having created 20,000 objects in the 3D space.

No. Of Objects	Time to Create Objects	$s(object)^{-1}$
1	0.057s	0.057s
10	0.063s	0.0063s
100	0.094s	0.00094s
1000	0.77s	0.00077s
10000	19.216s	0.0019s
20000	85.152s	0.0043s

Table 2: Test Rig Timings

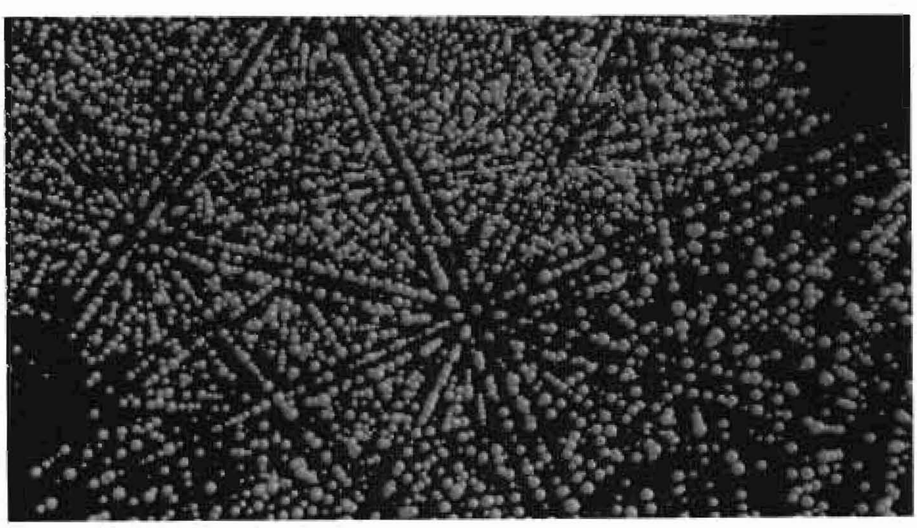

Figure 19: Displaying 20,000 unrelated objects.

6 Conclusions

Having thoroughly tested my code I feel I can now draw some conclusions about the project as a whole. In order to do this effectively, I will look back to the requirements outlined in §3, and judge the success of my project on how well these criterion were met.

Accuracy
With the careful construction of the Java3D scene graph, and the separation of concerns used throughout the project, I feel accuracy should occur as a by-product. This project does well at allowing each class to be concentrated upon regardless of other implementation concerns, solely aiming to fulfil its task. Hence, any inaccuracies are likely to be picked up at the point in which they may occur. Our interaction with the underlying model is somewhat seamless, and along with the results of the testing phase, I feel we can be assured as to the accuracy achieved in reflecting the run-time disconnected memory graph we set out to create.

Efficiency
As we have seen in our larger test runs, the response of the initial generation of the scene-graph, no longer remains instant. However, this delay only occurs upon this initial generation, and comes partly down to the Java3D renderer having to render so many 3D objects, but mainly due to our layout algorithm. Notably, if speed becomes a concern, we can switch to one of the other layout managers outlined in §4.5, and dramatically reduce complexity. Once rendered, the model behaves exquisitely however, and as such, I feel the efficiency concerns which were raised have been overcome. In most cases the program is instantaneous, and as programs become larger we only see a small increase in delay.

Making use of efficient data structures within the program, and ensuring that nodes aren't revisited and recalculated unless necessary, I feel this solution fairs extremely well in keeping track of the underlying memory graph.

Usability
As our efficiency requirement explains, this program is very responsive, but we also make sure it is intuitive for the end-user. We provide a mouse-based input, and a menu structure which adapts to the state of the program. Alongside the JDT debugger itself, this solution provides ease of use for both a debugger, and someone looking to visualise their program. Overall, I feel this program is a valuable addition to the already extremely user friendly Eclipse debugging perspective.

Extensibility
One of the best aspects of this program is its extensibility. It provides the framework for visualising any underlying object-orientated system, given a model to draw the information from. This is an important factor in making this project portable to other systems.

In addition to this, we create a layout manager abstraction which allows for added layout managers to be created, whilst abstracting away from the intricacies of the underlying scene graph creation. This provides us with an easy way to implement new three-dimensional layout techniques if they become available. This method of coding is a valuable asset to any system.

Integration
Not much needs to be said here as the program sits perfectly inside of Eclipse. When debugging, users have the option of opening a '3D View', which results in our model being created and executed inside the Eclipse window. This provides a method of using this 3D debugger and visualiser, side-by-side with the JDT debugger itself.

Having seen that the code does in fact meet the original requirements laid out, it can be said with confidence that the project has achieved what was intended. However, this is not to say that improvements cannot be made, and in §6.1, these changes will be discussed. Given the time allotted for this project, I am happy to say we have achieved something new. No program has ever set out to visualise and debug programs in this way, and I feel the end product is an extremely usable one. I do feel there is room for improvement when compared to advanced 2D debuggers and visualisers, however integration of these algorithms has been made simple by careful thought of our design. We provide a usable platform from the beginning, but also allow for future development of an exciting new aspect to program analysis and visualisation.

6.1 Further Work

Having completed this project, and received good results, it is still felt that there are areas in which more work can be done. Time constraints have not allowed this work to be carried out as yet, however it would be recommended that the maximal improvements would be achieved in the following areas.

6.1.1 Calculating Differences Between Program States

Zimmerman and Zeller discuss in their paper the idea of the greatest common subgraph [17]. This idea comes from the fact that a debugger may want to compare two program states, or runs, to see the differences. One such method for doing this, is in the construction of the greatest common subgraph. This gives us the opportunity to discover bugs, given a run that works correctly, and one which does not, the difference between the two program states would reveal the cause of the failure. Greatest common subgraph creation would be a solution to this problem, and a worthy addition to the framework we have already created.

6.1.2 On-the-fly Updating/Editing of Variables

The JDT debugger offers the ability to change values of variables in a live system [4]. As we saw in §4.6, we provide a table showing the values of a given object. Hence, it would be interesting to be able to update the values of a live system via changes here. This would simply move some of the JDT optionality, into the 3D universe view. Currently, our table implementation is extensible in this respect, however, the requirements of the underlying model must be updated to include this extra functionality.

6.1.3 Animating Program Runs

Another interesting aspect of Java3D is the ability to add animation [3]. It would be interesting to have an automated visualiser, possibly more so for teaching purposes, which would step through a program, and animate its construction. This would simply involve line creation animation, and object creatiou, modification and deletion animation effects. Overall, I think this would provide a more interesting way to display visualisations in a step-by-step manner, not generating new insight into the program, but increasing its accessibility and potential usability.

7 Acknowledgements

I would like to use this section to thank Professor Oege de Moor for his help in guiding me throughout this project, as well as Dr. Gavin Lowe for ensuring such good progress was made. I would also like to thank Luke Cartey for allowing use of his underlying debug model implementation in this project.

References

[1] Chris Aniszczyk and Pawel Leszek. Debugging with the eclipse platform. IBM Developer Networks Online - http://www.ibm.com/developerworks/java/library/os-ecbug/, 2007.

[2] Thomas Ball and Stephen G. Eick. Visualizing program slices. In *Visual Languages*, pages 288–295, 1994.

[3] Dennis J Bouvier. Getting started with the java3d api, 2002. http://java.sun.com/developer/onlineTraining/java3d/.

[4] David Boxer, Ashutosh Galande, and Thuc Si Mau Ho. The architecture of the eclipse jdt. https://netfiles.uiuc.edu/dboxer2/shared/cs527/JDT%20Architecture.pdf, 2004.

[5] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.

[6] Michael Callaghan and Heiko Hirschmüller. 3-D visualisation of design patterns and java programs in computer science education. *SIGCSE Bull.*, 30(3):37–40, 1998.

[7] Stephan Diehl, editor. *Software Visualization, International Seminar Dagstuhl Castle, Germany, May 20-25, 2001, Revised Lectures*, volume 2269 of *Lecture Notes in Computer Science*. Springer, 2002.

[8] Larry J. French. An interactive graphical debugging system. In *DAC '70: Proceedings of the 7th workshop on Design automation*, pages 271–273, New York, NY, USA, 1970. ACM.

[9] David R. Hanson and Jeffrey L. Korn. A simple and extensible graphical debugger. In *Winter 1997 USENIX Conference*, pages 173–184, 1997.

[10] Zellig Harris. Distributional structure. *Word*, 10(2/3):146–162, 1954.

[11] Claire Knight and Malcolm Munroe. Visualizing software - a key research area. In *ICSM '99: Proceedings of the IEEE International Conference on Software Maintenance*, page 437, Washington, DC, USA, 1999. IEEE Computer Society.

[12] J. Maletic, J. Leigh, A. Marcus, and G. Dunlap. Visualizing object oriented software in virtual reality. In *Proceedings of International Workshop on Program Comprehension (IWPC01)*, pages 26–35, 2001.

[13] Dave Springgay. Creating an eclipse view. http://www.eclipse.org/articles/viewArticle/ViewArticle2.html, 2001.

[14] R Stallman and R Pesch. Debugging with GDB, the GNU source-level debugger. *The Free Software Fondation, Inc*, (4), 1993.

[15] Frank Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[16] Andreas Zeller and Dorothea Lutkehaus. DDD - a free graphical front-end for UNIX debuggers. *SIGPLAN Notices*, 31(1):22–27, 1996.

[17] Thomas Zimmermann and Andreas Zeller. Visualizing memory graphs. In *Software Visualization*, pages 191–204, 2001.

8 Appendix

The following pages will contain the code for the majority of the methods in my program, restricted only by the page limit imposed.

```java
/**
 * The UpdateHandler Class:
 * This class will be passed on to the underlying debug interface, and used in
 * order to inform our model of underlying changes. We essentially expect this
 * method to be notified of all new, changed and deleted objects. The job of
 * this class is to then pass this information to the View3D object provided.
 *
 * @author Darius Bradbury.
 */

public class UpdateHandler implements DebugModelContainerListener {

  View3D view3D;

  /**
   * Assigns the local view3D object, and instantiates the UpdateHandler.
   *
   * @param view3d -
   *        the associated View3D object.
   */
  public UpdateHandler(View3D view3d) {
    view3D = view3d;
  }

  /**
   * This method is called by the intermediary framework. It is used to update
   * the 3D universe, passing on and new/changed/deleted objects to the View3D
   * object.
   *
   * A call to this method signifies that the underlying state of the program
   * has changed.
   *
   * @param debugTarget -
   *        the underlying debug target.
   * @param objectsChanged -
   *        A mapping of change type (new/changed/deleted) to
   *        IDebugObjects.
   */
  public void updateDebugModel(IDebugTarget debugTarget,
      Map<DebugChangeType, List<IDebugObject>> objectsChanged) {

    /*
     * Reset all objects state.
     *
     * We must ensure that each object is considered unchanged, unless told
     * otherwise. This updates the fact that we have entered a new system
     * state.
     */
    for (Object3D o3d : View3D.idoToObject3D.values()) {
      o3d.state = "unchanged";
    }

    // Create iterator variable used to iterate through the objects.
    Iterator<IDebugObject> iterator;

    // Check for new objects in the system.
    if (objectsChanged.containsKey(DebugChangeType.CREATED)) {

      // Set our iterator to the object which are NEW.
      iterator = objectsChanged.get(DebugChangeType.CREATED).iterator();

      // Iterate through, sending each IDebugObject to the view3D object.
      while (iterator.hasNext()) {
        IDebugObject itemp = iterator.next();
        view3D.createNew(itemp);
        // Set the state of these Object3D objects to NEW.
        View3D.idoToObject3D.get(itemp).state = "new";
      }
    }

    /*
     * Iterate through the changed objects, no need to send them through to
     * the View3D object however, just set their state as CHANGED.
     *
     */
    if (objectsChanged.containsKey(DebugChangeType.CHANGED)) {
      iterator = objectsChanged.get(DebugChangeType.CHANGED).iterator();

      while (iterator.hasNext()) {
        IDebugObject itemp = iterator.next();
        View3D.idoToObject3D.get(itemp).state = "changed";
      }
```

```java
        }

        /*
         * Iterate through all the deleted IDebugObjects, notify view3D of their
         * removal.
         */
        if (objectsChanged.containsKey(DebugChangeType.DELETED)) {

            iterator = objectsChanged.get(DebugChangeType.DELETED).iterator();
            while (iterator.hasNext()) {
                IDebugObject itemp = iterator.next();
                view3D.remove(itemp);
            }
        }

        // Having processed all objects, finalise view:

        // First extract all the Object3D objects still in our system.
        // We do this by accessing our static mapping of IDebugObjects to
        // Object3Ds.
        Collection<Object3D> totalListOfObjects = View3D.idoToObject3D.values();

        // If positioning depends on rank, update the rank and positions to
        // accommodate these changes.
        Object3D.layoutManager.updateAllPositions();

        /*
         * We then perform an update on each object. We do this at such at the
         * end in case the position of the objects depends on other objects. As
         * such, we must wait until all the objects have been sent through to
         * the view. Note - that the object isn't created until this step.
         *
         */
        System.out.println("[View] Updating objects.");
        for (Object3D o3d : totalListOfObjects) {
            o3d.update();
        }
        // Recreate lines if necessary.
        for (Object3D o3d : totalListOfObjects) {
            if (o3d.linesVisible) {
                o3d.hideLines();
                o3d.showLines();
            } else if (View3D.allLinesVisible) {
                o3d.showLines();
            }
        }
        // If we were in a sub-view, we re-create that same view.
        if (view3D.justSubObjects) {

            // We check that the root node of this sub-view hasn't been deleted.
            if (View3D.idoToObject3D
                    .containsValue(view3D.currentRightClickedNode)) {
                view3D.createTrace();
            } else {
                // Object we were tracing no longer EXISTS.
                // return to full graph.
                view3D.justSubObjects = false;
            }
        }
        // Print to console number of objects generated.
        System.out.println("[VIEW] There exists " + View3D.idoToObject3D.size()
                + " visual objects in the overall graph.");
    }
}
```

```
/**
 * The Object3D Class:
 * This class will hold objects definitions, with their 3D
 * representations. It acts as a wrapper for the IDebugObjects from the
 * underlying model, and provides methods for maintaining its 3D representation.
 *
 * @author Darius Bradbury.
 */

public class Object3D {

  public static LayoutManager3D layoutManager; // Layout manager in use.
  public IDebugObject ido; // The underlying IDebugObject.
  private TransformGroup tg; // This objects TransformGroup
  private BranchGroup bg; // This objects BranchGroup.
  // The sub-BranchGroup for the visual object representing the name of the
  // object.
  private BranchGroup bgName;
  private View3D view3D; // The view3D object.
  public Vector3d v3d; // Vector representing object's position.
  public String name; // Object's name.
  public boolean detailsVisible = false; // name on or off flag.
  public boolean linesVisible = false; // trace lines on or off flag.
  // Collection of created lines.
  private LinkedList<BranchGroup> linesList = new LinkedList<BranchGroup>();
  // Appearance NodeComponent for this visual object.
  private Appearance appearance;
  // Tree layout manager for forward traces.
  public static TreeLayoutC treeLayout;
  // Tree layout for backward traces.
  public static BackTreeLayoutC backTreeLayout;
  public float objectSize; // Size of the object.
  // Definitions for the general view layout manager type.
  static final int gridtype = 0;
  static final int stacktype = 1;
  static final int rankbased = 2;
  static final int clusterbased = 3;
  // Setting the layout manager type.
  static int layoutManagerType = clusterbased;
  // Seen list for BFS tree generation.
  public static LinkedList<IDebugObject> seenObjectList =
    new LinkedList<IDebugObject>();
  public String state = "unchanged"; // Current state.

  /**
   * Instantiates an Object3D object, attaches the associated IDebugObject,
   * and links to the View3D controller class.
   *
   * @param ido -
   *        The IDebugObject this class provides a wrapper for.
   * @param view3D -
   *        The associated View3D controller class.
   */
  public Object3D(IDebugObject ido, View3D view3D) {
    this.ido = ido;
    this.view3D = view3D;
    // Create new TransformGroup node for this Object.
    tg = new TransformGroup();
    // Add it to the view3D map to allow picking.
    view3D.tgToObject3D.put(tg, this);

    // Create the BranchGroup node, and attach the TransformGroup node.
    bg = new BranchGroup();
    bg.setCapability(BranchGroup.ALLOW_DETACH);
    bg.addChild(tg);
    // Create a new Vector3D to hold this objects position.
    v3d = new Vector3d();
    // Set the name
    name = ido.getValue().toString();
    // Create the associated layoutManager on first run.
    if (layoutManager == null) {
      switch (layoutManagerType) {
      case gridtype:
        layoutManager = new GridLayout();
        break;
      case stacktype:
        layoutManager = new StackedLayout();
        break;
      case rankbased:
        layoutManager = new RankBasedLayout();
        break;
      case clusterbased:
```

```
          layoutManager = new ClusteringBasedLayout();
          break;
        }
      }
    }

    /**
     * This method creates the 3D sphere. It uses the current state of the
     * global Object3D parameters for position, size, etc.. to do this.
     */
    private void createObject() {

      // Create the Sphere and set associated capabilities.
      Sphere newObj = new Sphere(objectSize);
      newObj.setPickable(true);
      newObj.setName(name);
      newObj.setCapability(Shape3D.ALLOW_APPEARANCE_WRITE);
      newObj.setCapability(Shape3D.ALLOW_APPEARANCE_READ);
      newObj.setCapability(Group.ALLOW_CHILDREN_WRITE);
      newObj.setCapability(Primitive.ENABLE_APPEARANCE_MODIFY);

      // Remove the current TransformGroup, create a new one, and attach.
      view3D.tgToObject3D.remove(tg);
      tg = new TransformGroup();
      view3D.tgToObject3D.put(tg, this);

      // Get appearance and set capabilities.
      appearance = newObj.getAppearance();
      appearance.setCapability(Appearance.ALLOW_MATERIAL_WRITE);

      tg.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
      tg.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
      tg.setCapability(Node.ENABLE_PICK_REPORTING);
      tg.setCapability(BranchGroup.ALLOW_DETACH);
      tg.setCapability(Group.ALLOW_CHILDREN_EXTEND);
      tg.setCapability(Group.ALLOW_CHILDREN_WRITE);
      tg.setCapability(Shape3D.ALLOW_APPEARANCE_WRITE);

      // Create new Transform.
      Transform3D transform = new Transform3D();
      // Set the Transform to move to the Objects current position.
      transform.set(v3d);
      // Perform this translation.
      tg.setTransform(transform);
      // Place the sphere in place.
      tg.addChild(newObj);

      // Create new BranchGroup.
      bg = new BranchGroup();
      bg.setCapability(BranchGroup.ALLOW_DETACH);
      // Add newly created TranformGroup.
      bg.addChild(tg);
      bg.setName(name);

      // Set Colour depending on objects state.
      if (state.equals("new")) {
        Color3f ambientColor = new Color3f(0.0f, 0.6f, 0.0f);
        Color3f emissiveColor = new Color3f(0f, 0f, 0f);
        Color3f diffuseColor = new Color3f(0.5f, 0.5f, 0.5f);
        Color3f specularColor = new Color3f(0.7f, 0.7f, 0.7f);
        float shininess = 64;
        Material mat = new Material(ambientColor, emissiveColor,
            diffuseColor, specularColor, shininess);
        // Set material, sets colour and lighting attributes.
        appearance.setMaterial(mat);
      } else if (state.equals("changed")) {
        Color3f ambientColor = new Color3f(1f, 0.4f, 0f);
        Color3f emissiveColor = new Color3f(0f, 0f, 0f);
        Color3f diffuseColor = new Color3f(0.5f, 0.5f, 0.5f);
        Color3f specularColor = new Color3f(0.7f, 0.7f, 0.7f);
        float shininess = 64;
        Material mat = new Material(ambientColor, emissiveColor,
            diffuseColor, specularColor, shininess);
        appearance.setMaterial(mat);
      } else {
        Color3f ambientColor = new Color3f(0.7f, 0.7f, 0.7f);
        Color3f emissiveColor = new Color3f(0.0f, 0.0f, 0.0f);
        Color3f diffuseColor = new Color3f(0.7f, 0.7f, 0.7f);
        Color3f specularColor = new Color3f(0.9f, 0.9f, 0.9f);
        Material mat = new Material(ambientColor, emissiveColor,
            diffuseColor, specularColor, 64.f);
        mat.setColorTarget(Material.SPECULAR);
```

```java
      appearance.setMaterial(mat);
    }
  }

  /**
   * This method returns this Object3Ds root node, in the Java3D scene graph.
   *
   * @return The root BranchGroup.
   */
  public BranchGroup getBranchGroup() {
    return bg;
  }

  /**
   * This method is called once the system is stable, and is expected to
   * update the state variables of the Object3D instance, thus allowing the
   * createObject() method to create a correctly positioned and coloured
   * object. We also maintain the Object3D's name object.
   */
  public void update() {
    // Remove currently stored BranchGroup from overall map.
    view3D.mainTransformGroup.removeChild(bg);
    // Remove current TransformGroup node from our BranchGroup.
    bg.removeChild(tg);
    // Get current Object position from the Layout Manager.
    Vector3d pos = layoutManager.getPosition(this);
    // Set our stored position to match this.
    v3d.setX(pos.getX());
    v3d.setY(pos.getY());
    v3d.setZ(pos.getZ());

    // Calculate size.
    // calculateStepSize();
    calculateImportanceSize();

    // Now all the variables have been set, create the visual object.
    createObject();
    // Add newly created BranchGroup to the overall mapping.
    view3D.mainTransformGroup.addChild(bg);
    // If details had been created for this object, create them again.
    if (detailsVisible) {
      tg.removeChild(bgName);
      createDetails();
    }
  }

  /**
   * Method used to calculate pure step size. Whereby, as objects get less and
   * less important, their size is halved. This was originally used in the
   * divide and resize algorithm. However, we now make use of the importance
   * size metric.
   */
  public void calculateStepSize() {

    double rank = 1;
    LinkedList<Object3D> ll = LayoutManager3D.currentRanking;
    boolean stillSearching = true;

    for (Object3D rankedObject : ll) {
      if (rankedObject.equals(this)) {
        stillSearching = false;
      } else if (stillSearching) {
        rank++;
      }
    }
    objectSize = 10;
    while (rank > 1) {
      objectSize = objectSize / 2;
      if (rank <= 7) {
        rank = 0;
      } else {
        rank = rank / 5;
      }
    }
  }

  /**
   * This method calculates and sets the object size based on its importance
   * relative to the importance of all the other objects in the system.
   */
  public void calculateImportanceSize() {

```

```
        // Get ordered list of ranked objects from Layout Manager.
        LinkedList<Object3D> ll;
        ll = LayoutManager3D.currentRanking;

        // Take top scoring object score.
        double upperBound = ll.getFirst().ido.getPageRank();
        // Get score for this object.
        double importanceScore = ido.getPageRank();
        // Calculate ratio
        double ratio = importanceScore / upperBound;
        // Biggest object = size 10, every other object a ratio of that.
        // The range of object sizes is 0.5-10.
        objectSize = (float) ((9.5 * ratio) + 0.5);
    }

    /**
     * This method generates a BranchGroup node containing the visual object
     * representing the name of this Object3D. It then adds it to the Object3D's
     * own BranchGroup node, and hence, the virtual world.
     */
    public void createDetails() {

        // Create Font object
        // Size of font based on size of object.
        Font f = new Font("calibri", Font.BOLD,
                (int) ((objectSize * 2) / 10) + 1);
        // Create Font Extrusion, used to turn 2D font, into 3D object.
        FontExtrusion fe = new FontExtrusion();

        // Create 3D Font object.
        Font3D f3d = new Font3D(f, fe);

        // Set position at edge of visual object.
        Point3f point3f = new Point3f(0, 0, objectSize);
        // Generate 3D Text object.
        Text3D text = new Text3D(f3d, name, point3f);
        text.setAlignment(Text3D.ALIGN_CENTER);
        Shape3D textShape = new Shape3D();
        textShape.setGeometry(text);

        // Set colour and response to light of the Text Object.
        Color3f ambientColor = new Color3f(0f, 0f, 0f);
        Color3f emissiveColor = new Color3f(0.0f, 0.0f, 0.0f);
        Color3f diffuseColor = new Color3f(0.0f, 0.0f, 0.0f);
        Color3f specularColor = new Color3f(0.0f, 0.0f, 0.0f);
        Material mat = new Material(ambientColor, emissiveColor, diffuseColor,
                specularColor, 64.f);
        Appearance textAppearance = new Appearance();
        textAppearance.setMaterial(mat);
        textShape.setAppearance(textAppearance);

        // Create BranchGroup to govern this text object.
        bgName = new BranchGroup();
        bgName.setCapability(BranchGroup.ALLOW_DETACH);
        bgName.addChild(textShape);

        // Add our Text Object to this Object3D's TranformGroup.
        tg.addChild(bgName);
    }

    /**
     * Method to show details of this Object3D, namely, show its name.
     */
    public void showDetails() {
        if (!detailsVisible) {
            createDetails();
            detailsVisible = true;
        }
    }

    /**
     * Method to hide the details of this Object3D.
     */
    public void hideDetails() {
        if (detailsVisible) {
            tg.removeChild(bgName);
            detailsVisible = false;
        }
    }

    /**
     * This method performs a depth-first-iteration through the tree of forward
```

```
     * links, creating each object on the way.
     */
    public void displayObjectLinks() {

      // Clear current seen object list, used to avoid repeat visiting nodes.
      seenObjectList.clear();
      createSubObjects();
      view3D.justSubObjects = true;
      view3D.traceDirection = 0;
    }

    /**
     * This method performs a Depth-First iteration of the backward links of
     * this node, generating each object as it goes.
     */
    public void displayObjectBackLinks() {

      // Clear current seen object list, used to avoid repeat visiting nodes.
      seenObjectList.clear();
      createBackLinkSubObjects();
      view3D.justSubObjects = true;
      view3D.traceDirection = 1;
    }

    /**
     * This method generates this Object3D instance, then calls the relevant
     * creation method in each of the Object3D's it has forward links to. In
     * other words, it generates a forward trace.
     */
    private void createSubObjects() {

      // Every child removed when object right-clicked.
      // We just add correct objects.

      // Remove any lines if they are currently on display.
      if (linesVisible) {
        removeLines();
      }
      // Remove the TransformGroup for this Object3D.
      bg.removeChild(tg);

      // Get position from the tree layout manager.
      Vector3d pos = treeLayout.getPosition(this);
      v3d.setX(pos.getX());
      v3d.setY(pos.getY());
      v3d.setZ(pos.getZ());

      // Create object, now based on its trace position.
      createObject();

      // Add newly updated BranchGroup to the mapping.
      view3D.mainTransformGroup.addChild(bg);
      // Restore details if they were visible.
      if (detailsVisible) {
        tg.removeChild(bgName);
        createDetails();
      }

      // Create local map variable for this Object3D.
      Map<IDebugObject, IVariable> linklist = null;

      // We ensure the ido is not null, however, if it is we throw an
      // exception.
      try {
        linklist = ido.objectLinks();
      } catch (NullLinkException e) {
        e.printStackTrace();
      }

      // We add this ido to our seen list, ensuring we don't try to create it
      // again.
      seenObjectList.add(ido);

      // Iterate through object links, creating each object.
      for (Entry<IDebugObject, IVariable> variableLink : linklist.entrySet()) {
        IDebugObject i = variableLink.getKey();
        if (i != null && !seenObjectList.contains(i)) {
          // If we haven't seen this object yet, search it.
          View3D.idoToObject3D.get(i).createSubObjects();
          seenObjectList.add(ido);
        }
        // Draw lines from this Object3D to each of it's children.
```

```
        createLines(this, i);
      }
    }

    /**
     * This method creates a backward trace, performing a depth-first iteration
     * of the backward links of this object and it's associated IDebugObject.
     */
    private void createBackLinkSubObjects() {
      try {
        // Every child removed when object right-clicked.
        // Don't need to worry about that, just add correct objects.

        if (linesVisible) {
          removeLines();
        }

        bg.removeChild(tg);

        Vector3d pos = backTreeLayout.getPosition(this);
        v3d.setX(pos.getX());
        v3d.setY(pos.getY());
        v3d.setZ(pos.getZ());

        createObject();
        view3D.mainTransformGroup.addChild(bg);
        if (detailsVisible) {
          tg.removeChild(bgName);
          createDetails();
        }

        Map<IDebugObject, IVariable> linklist;

        linklist = ido.backLinks();
        seenObjectList.add(ido);

        // Iterate through object links, creating each object.
        for (Entry<IDebugObject, IVariable> variableLink : linklist
            .entrySet()) {
          IDebugObject i = variableLink.getKey();
          if (i != null && !seenObjectList.contains(i)) {
            View3D.idoToObject3D.get(i).createBackLinkSubObjects();
            seenObjectList.add(i);
          }
          // Create the lines from this object to each of its children.
          createLines(this, i);
        }

      } catch (NullLinkException e) {
        throw new RuntimeException(e);
      }
    }

    /**
     * This method changes the colour of the Object3D's sphere to be red. We use
     * this method to highlight the root node in a trace.
     */
    public void highlightCurrentObject() {
      Color3f ambientColor = new Color3f(0.33f, 0, 0);
      Color3f emissiveColor = new Color3f(0, 0, 0);
      Color3f diffuseColor = new Color3f(0.5f, 0.5f, 0.5f);
      Color3f specularColor = new Color3f(0.7f, 0.7f, 0.7f);
      float shininess = 64;
      Material mat = new Material(ambientColor, emissiveColor, diffuseColor,
          specularColor, shininess);
      appearance.setMaterial(mat);
    }

    /**
     * This method removes any highlight that had been imposed.
     */
    public void removeHighlight() {

      Color3f emissiveColor = new Color3f(0.0f, 0.0f, 0.0f);
      Color3f ambientColor = new Color3f(0.1f, 0.1f, 0.1f);
      Color3f diffuseColor = new Color3f(0.7f, 0.7f, 0.7f);
      Color3f specularColor = new Color3f(0.9f, 0.9f, 0.9f);
      Material mat = new Material(ambientColor, emissiveColor, diffuseColor,
          specularColor, 64.f);
      mat.setColorTarget(Material.SPECULAR);
      appearance.setMaterial(mat);
    }
```

```java

    /**
     * This method generates lines between the given Object3D and list of
     * IDebugObjects, in the 3D world.
     *
     * @param object3D -
     *         The source object. Where the lines must come from.
     * @param linklist -
     *         The target objects. Where the lines are going to.
     */
    private void createLines(Object3D object3D, Set<IDebugObject> linklist) {

      for (IDebugObject ido : linklist) {
        // We must check objects don't have links to themselves.
        // Otherwise we try to create lines with null transforms.
        if (!object3D.equals(View3D.idoToObject3D.get(ido))) {
          createLines(object3D, ido);
        }
      }
    }

    /**
     * This method generates a single line, from the given Object3D to the
     * IDebugObject given. This involves finding the position of both in the 3D
     * universe, and then generating a directed line representing the link
     * between them.
     *
     * @param object3D -
     *         The source node.
     * @param ido -
     *         The target node.
     */
    private void createLines(Object3D object3D, IDebugObject ido) {

      // Create appearance for the lines.
      Appearance app = new Appearance();
      ColoringAttributes ca = new ColoringAttributes(
          new Color3f(43, 173, 43), ColoringAttributes.SHADE_FLAT);

      // Create point array to contain start and end position of line.
      Point3f[] linePoints = new Point3f[2];

      /*
       * Object clicked on Position, relative to this object ie. Current
       * Position!
       */
      linePoints[0] = new Point3f(0, 0, 0);
      /*
       * Object linked to Position, relative to this object ie. Linked to
       * Object vector, minus current object vector.
       */
      float destx = ((float) View3D.idoToObject3D.get(ido).v3d.getX())
          - (float) v3d.getX();
      float desty = ((float) View3D.idoToObject3D.get(ido).v3d.getY())
          - (float) v3d.getY();
      float destz = ((float) View3D.idoToObject3D.get(ido).v3d.getZ())
          - (float) v3d.getZ();
      linePoints[1] = new Point3f(destx, desty, destz);

      // Create line based on positions.
      LineArray lineArray = new LineArray(2, GeometryArray.COORDINATES);
      lineArray.setCoordinates(0, linePoints);

      app.setColoringAttributes(ca);

      Shape3D lines = new Shape3D(lineArray, app);

      // Create BranchGroup to Govern this line.
      BranchGroup bgtemp = new BranchGroup();
      bgtemp.setCapability(BranchGroup.ALLOW_DETACH);
      bgtemp.addChild(lines);

      /*
       * As line is directed, we must create an arrowhead. To do this we use a
       * Cone object.
       */

      // Cone size must be proportional to distance apart of objects.
      Vector3f lineVector = new Vector3f(destx, desty, destz);
      float length = lineVector.length();
      float coneLength = length / 10;
      // Set a cone size limit!
```

```
    if (coneLength > 5) {
      coneLength = 5;
    }
    // Create cone.
    Cone arrow = new Cone(coneLength / 5, coneLength, object3D.appearance);

    // Create TransformGroup to correctly position cone.
    TransformGroup tgArrow = new TransformGroup();
    Transform3D t3dArrow = new Transform3D();

    // Vector we need to translate cone by.
    Vector3f vArrow = new Vector3f(destx, desty, destz);

    // Make sure the two objects are in fact not in the same place!
    if (vArrow.length() == 0) {
      System.out.println("[VIEW] vArrow.length() == 0!");
    } else {
      // Scale such that translation moves to edge of object,
      // not centroid.
      vArrow.scale(((vArrow.length()
          - View3D.idoToObject3D.get(ido).objectSize)
          - coneLength / 2)
              / vArrow.length());

      // Calculating rotational properties.

      Vector3f objectTo = new Vector3f(destx, desty, destz);
      // Angle of rotation
      float angle = (new Vector3f(0, 1, 0)).dot(objectTo);
      angle = angle / objectTo.length();
      angle = (float) java.lang.Math.acos(angle);

      // Axis of rotation
      Vector3f direction = new Vector3f();
      Vector3f yAxis = new Vector3f(0, 1, 0);
      objectTo.normalize();
      direction.cross(yAxis, objectTo);
      if ((int) java.lang.Math.toDegrees(angle) == 180) {
        // Dealing with perpendicular issue with
        // (0,1,0) and (0,1,0)!
        direction = new Vector3f(1, 0, 0);
      }

      // Set rotation first.
      t3dArrow.setRotation(new AxisAngle4d(direction.getX(), direction
          .getY(), direction.getZ(), angle));
      // Then set translation.
      t3dArrow.setTranslation(vArrow);
      // Then perform transform as a whole.
      tgArrow.setTransform(t3dArrow);

      // Create BranchGroup for the cone, and add just created
      // TransformGroup
      // to it.
      BranchGroup bgArrow = new BranchGroup();
      tgArrow.addChild(arrow);
      bgArrow.addChild(tgArrow);

      // Add the cone to the BranchGroup governing the lines.
      bgtemp.addChild(bgArrow);

      // Add this whole BranchGroup to the Object3D's TransformGroup node.
      tg.addChild(bgtemp);
      // Maintain a list of all the line BranchGroups for easy removal.
      linesList.add(bgtemp);
    }
  }

  /**
   * Making use of the maintained visible lines list, this method removes all
   * lines in the system from the main TranformGroup node.
   */
  public void removeLines() {

    for (BranchGroup b : linesList) {
      tg.removeChild(b);
    }
    linesList.clear();
  }

  /**
   * This method removes the lines for all the Object3D's in a trace.
```

```
 */
public void removeObjectLinks() {
  for (IDebugObject i : seenObjectList) {
    if (View3D.idoToObject3D.containsKey(i)) {
      View3D.idoToObject3D.get(i).removeLines();
    } else {
      // We know this object has already been removed since the
      // seen list creation.
    }
  }
}

/**
 * This method restores the general view, replacing all the objects in it.
 */
public void replaceAllObjects() {

  Collection<Object3D> c = View3D.idoToObject3D.values();

  for (Object3D o3d : c) {
    try {
      o3d.update();
    } catch (Exception e) {
      System.out.println("[VIEW - ERROR]" + e.getMessage());
    }
  }
  // Recreate the lines.
  for (Object3D o3d : c) {
    if (o3d.linesVisible) {
      // We know the lines exist, thus we have to hide, and recreate.
      o3d.hideLines();
      o3d.showLines();
    } else if (View3D.allLinesVisible) {
      // We know the lines weren't visible, but the user wishes ALL
      // lines to be visible, so we show them.
      o3d.showLines();
    }
  }
  // No longer in trace view.
  view3D.justSubObjects = false;
}

/**
 * This method creates an up-to-date TreeLayout Object for forward traces.
 */
public void createCurrentTree() {
  treeLayout = new TreeLayoutC();
}

/**
 * This method creates an up-to-date TreeLayout Object for backward traces.
 */
public void createCurrentBackLinkTree() {
  backTreeLayout = new BackTreeLayoutC();
  backTreeLayout.getPosition(this);
}

/**
 * This method shows the forward links for this Object3D.
 */
public void showLines() {
  try {
    createLines(this, ido.objectLinks().keySet());
  } catch (NullLinkException e) {
    throw new RuntimeException(e);
  }
  linesVisible = true;
}

/**
 * This method hides the forward links for this Object3D.
 */
public void hideLines() {
  removeLines();
  linesVisible = false;
}
}
```

```
/**
 * The View3D Class:
 * This class aims to maintain communication between the update handler, the
 * Object3D instances, and interaction with the user.
 *
 * @author Darius Bradbury.
 */
public class View3D extends ViewPart implements ActionListener, MouseListener {
  private boolean TESTING = false; // Testing mode flag to run test rig.
  public static java.awt.Frame f; // The frame for our 3D Canvas.
  public static int width;// Initial width of graphics window.
  public static int height; // Initial height of graphics window.
  // Ratio of width compared to height in widescreen window.
  public static final double wideScreenRatio = 1.77;
  // Size of bounding sphere.
  public static double boundingSphereSize = Double.MAX_VALUE;
  Canvas3D canvas3D; // 3D rendering canvas
  Panel b_container; // Container to hold the buttons
  Panel c_container; // Container to hold the canvas
  Panel l_container; // Container to hold the labels
  Panel instruct_panel; // Panel to hold instructions
  Button instruct_button; // Instructions button
  Button new_object_button; // New Object Button
  Button create_3DS_object;
  Button clear_screen;
  Button remove_last;
  TextArea instruct_text; // TextArea object that holds instructions
  Button instruct_return_button; // Return button for instruction panel
  String textString; // Storage area for instructions
  private SimpleUniverse universe = null;
  Transform3D transform;
  int count; // current number of objects
  BranchGroup scene3D; // scene branchgroup
  TransformGroup mainTransformGroup; // main transform group!
  BranchGroup mainBranchGroup; // main Branch Group!
  // Main HashMap for mapping IDebugObjects to thier Object3D containers.
  public static HashMap<IDebugObject, Object3D> idoToObject3D =
    new HashMap<IDebugObject, Object3D>();
  // Keeping track of which transformGroup owns what. Used to enable picking.
  HashMap<TransformGroup, Object3D> tgToObject3D =
    new HashMap<TransformGroup, Object3D>();
  boolean justSubObjects = false; // Current state flag
  Object3D currentRootNode; // Used for keeping track of trace root.
  public Object3D currentRightClickedNode; // For passing of currently
  // selected node.
  private JScrollPane objectDetails; // Object details Table.
  private PickCanvas pickCanvas; // The PickCanvas used.
  public int traceDirection; // Passing of current trace direction.
  public static boolean allLinesVisible = false; // ALL lines flag.

  // Main Menu
  JPopupMenu mainMenu;
  JMenuItem gridView;
  JMenuItem stackView;
  JMenuItem divideResize;
  JMenuItem clusteringBased;
  JMenuItem resetView1;
  JMenuItem showObjectNames1;
  JMenuItem hideObjectNames1;
  JMenuItem showLines;
  JMenuItem hideLines;
  JMenuItem redrawSpace;
  // Object menu
  JPopupMenu objectMenu;
  JMenuItem forwardTrace;
  JMenuItem backwardTrace;
  JMenuItem showObjectDetails;
  JMenuItem showObjectLines;
  JMenuItem hideObjectLines;
  // Sub Objects main menu
  JPopupMenu subObjectsMenu;
  JMenuItem exitTrace;
  JMenuItem showObjectNames2;
  JMenuItem hideObjectNames2;
  JMenuItem resetView2;

  public View3D() {
    // Calls the ViewPart class.
    super();
  }

  /**
```

```
 * This is a callback that will allow us to create the view perspective, and
 * initialise it.
 *
 * What is expected is that we create a frame based on the input Composite
 * object, which will contain our view.
 */
public void createPartControl(Composite parent) {

  // Create new Composite object given parent node.
  Composite composite = new Composite(parent, SWT.EMBEDDED);
  // Set the 2D layout manager as a FillLayout.
  composite.setLayout(new FillLayout());
  // Create a frame to add our canvas into, along with any
  // other components we wish to display.
  f = SWT_AWT.new_Frame(composite);
  // Set the internal frame layout to a FlowLayout.
  f.setLayout(new FlowLayout());

  /*
   * Create an Update handler object to deal with all underlying change
   * notifications. Subscribe the update handler to our intermediary
   * debugging framework.
   */
  UpdateHandler uh = new UpdateHandler(this);
  DebugModelContainer.INSTANCE.addListener(uh);

  // Initialise the view (Create a virtual 3D universe and a physical
  // canvas)
  init();

  // pack the resulting frame.
  f.pack();

  // Deal with maintaining the correct aspect ration during resizing.
  composite.addControlListener(new ControlAdapter() {
    public void controlResized(ControlEvent e) {
      canvas3D.setSize((int) (f.getBounds().height * wideScreenRatio), f
          .getBounds().height);
    }
  });
  // Set the initial size.
  canvas3D.setSize((int) (f.getBounds().height * wideScreenRatio),
      f.getBounds().height);

  // Commence testing if mode selected.
  if(TESTING)
  {
    new TestRig(uh);
  }
}

/**
 * Initialisation of the Java3D minimal scene graph.
 *
 * This function aims to initialise the parameters required in setting up
 * the Java3D scene graph. It also configures the user input methods,
 * allowing interaction with the environment.
 */
public void init() {

  // Create a 3D graphics canvas.
  canvas3D = new Canvas3D(SimpleUniverse.getPreferredConfiguration());

  // Create the scene BranchGroup.
  scene3D = createScene3D();

  // Pick enabling
  pickCanvas = new PickCanvas(canvas3D, scene3D);
  pickCanvas.setMode(PickTool.GEOMETRY);
  pickCanvas.setTolerance(0);

  // Add mouse Listener
  canvas3D.addMouseListener(this);

  // Create a universe with the Java3D universe utility.
  universe = new SimpleUniverse(canvas3D);
  BoundingSphere bounds = new BoundingSphere(new Point3d(0.0, 0.0, 0.0),
      boundingSphereSize);

  // Create a method for rotating the whole 3D environment.
  MouseRotate behavior = new MouseRotate();
  behavior.setTransformGroup(mainTransformGroup);
```

```java
    mainBranchGroup.addChild(behavior);
    behavior.setSchedulingBounds(bounds);

    // Create a method for translating the whole 3D environment.
    MouseTranslate behavior1 = new MouseTranslate();
    behavior1.setTransformGroup(mainTransformGroup);
    behavior1.setFactor(0.5);
    mainBranchGroup.addChild(behavior1);
    behavior1.setSchedulingBounds(bounds);

    // Create a method for zooming the users viewpoint.
    MouseWheelZoom behavior2 = new MouseWheelZoom();
    // Note, the transform group relies on the viewPlatformTransform, not
    // the MainTransformGroup.
    behavior2.setTransformGroup(universe.getViewer().getViewingPlatform()
        .getViewPlatformTransform());
    behavior2.setFactor(20);
    mainBranchGroup.addChild(behavior2);
    behavior2.setSchedulingBounds(bounds);

    // Create a method for moving around the view point with the arrow keys.
    KeyNavigatorBehavior keyNavBeh = new KeyNavigatorBehavior(universe
        .getViewer().getViewingPlatform().getViewPlatformTransform());
    keyNavBeh.setSchedulingBounds(bounds);
    mainBranchGroup.addChild(keyNavBeh);

    // Add our scene3D branch, to the universe.
    universe.addBranchGraph(scene3D);

    // Move the initial view back slightly, so that all the objects can be
    // seen.
    TransformGroup tg = universe.getViewingPlatform()
        .getViewPlatformTransform();
    transform = new Transform3D();
    transform.set(65.f, new Vector3f(0.0f, 0.0f, 600.0f));
    tg.setTransform(transform);

    // Add the 3D canvas created by Java3D to our Eclipse frame.
    f.add(canvas3D);

    // Create the pop-up menus to allow extra interactions with the 3D
    // environment.
    createPopupMenus();
  }

  /**
   * This method sets the global pop-up menu parameters. It sets their names,
   * and their ordering.
   */
  private void createPopupMenus() {

    // This line allows heavyweight creation of Swing objects.
    JPopupMenu.setDefaultLightWeightPopupEnabled(false);

    // Create the main pop-up menu.
    mainMenu = new JPopupMenu();

    gridView = new JMenuItem("Grid View");
    stackView = new JMenuItem("Stack View");
    divideResize = new JMenuItem("Constant Space View");
    clusteringBased = new JMenuItem("Clustering Based");
    gridView.addActionListener(this);
    stackView.addActionListener(this);
    divideResize.addActionListener(this);
    clusteringBased.addActionListener(this);
    mainMenu.add(gridView);
    mainMenu.add(stackView);
    mainMenu.add(divideResize);
    mainMenu.add(clusteringBased);
    clusteringBased.setEnabled(false);
    mainMenu.addSeparator();

    resetView1 = new JMenuItem("Reset View");
    resetView1.addActionListener(this);
    mainMenu.add(resetView1);
    showObjectNames1 = new JMenuItem("Show Object Names");
    showObjectNames1.addActionListener(this);
    mainMenu.add(showObjectNames1);
    hideObjectNames1 = new JMenuItem("Hide Object Names");
    hideObjectNames1.addActionListener(this);
    mainMenu.add(hideObjectNames1);
    showLines = new JMenuItem("Show Lines");
```

```java
    showLines.addActionListener(this);
    mainMenu.add(showLines);
    hideLines = new JMenuItem("Hide Lines");
    hideLines.addActionListener(this);
    mainMenu.add(hideLines);
    redrawSpace = new JMenuItem("Redraw Space");
    redrawSpace.addActionListener(this);
    mainMenu.add(redrawSpace);

    objectMenu = new JPopupMenu();

    forwardTrace = new JMenuItem("Forward Trace");
    forwardTrace.addActionListener(this);
    objectMenu.add(forwardTrace);
    backwardTrace = new JMenuItem("Backward Trace");
    backwardTrace.addActionListener(this);
    objectMenu.add(backwardTrace);
    showObjectDetails = new JMenuItem("Show Details");
    showObjectDetails.addActionListener(this);
    objectMenu.add(showObjectDetails);
    showObjectLines = new JMenuItem("Show Object Lines");
    showObjectLines.addActionListener(this);
    objectMenu.add(showObjectLines);
    hideObjectLines = new JMenuItem("Hide Object Lines");
    hideObjectLines.addActionListener(this);
    objectMenu.add(hideObjectLines);

    subObjectsMenu = new JPopupMenu();

    exitTrace = new JMenuItem("Exit Trace");
    exitTrace.addActionListener(this);
    subObjectsMenu.add(exitTrace);
    subObjectsMenu.addSeparator();
    showObjectNames2 = new JMenuItem("Show Names");
    showObjectNames2.addActionListener(this);
    subObjectsMenu.add(showObjectNames2);
    hideObjectNames2 = new JMenuItem("Hide Names");
    hideObjectNames2.addActionListener(this);
    subObjectsMenu.add(hideObjectNames2);
    resetView2 = new JMenuItem("Reset View");
    resetView2.addActionListener(this);
    subObjectsMenu.add(resetView2);
  }

  public void destroy() {
    universe.cleanup();
  }

  /**
   * This method sets up the main BranchGroup parameters. This is the Branch
   * of the Java3D scene graph which will contain all of our run-time objects.
   * We set parameters including lighting, background colour, boundingSphere,
   * and capabilities of the main BranchGroup node. We also assign this
   * BranchGroup an associated TransformGroup which will deal with the
   * Transforms made upon the whole universe.
   *
   * @return The Main BranchGroup node. ie. A node to add all the visual 3D
   *         objects to.
   */
  public BranchGroup createScene3D() {

    // Define colours
    Color3f white = new Color3f(1.0f, 1.0f, 1.0f);
    Color3f bgColor = new Color3f(0.25f, 0.25f, 0.25f);

    // Create the Main BranchGroup
    mainBranchGroup = new BranchGroup();

    // Create the bounding leaf node
    // This specifies the size of the rendering space.
    BoundingSphere bounds = new BoundingSphere(new Point3d(0.0, 0.0, 0.0),
        boundingSphereSize);
    BoundingLeaf boundingLeaf = new BoundingLeaf(bounds);
    mainBranchGroup.addChild(boundingLeaf);

    // Create the background
    Background bg = new Background(bgColor);
    bg.setApplicationBounds(bounds);
    mainBranchGroup.addChild(bg);

    // Create the ambient light
    AmbientLight ambLight = new AmbientLight(white);
```

```java
    ambLight.setInfluencingBounds(bounds);
    mainBranchGroup.addChild(ambLight);

    // Create the directional light
    Vector3f dir = new Vector3f(-1.0f, -1.0f, -1.0f);
    DirectionalLight dirLight = new DirectionalLight(white, dir);
    dirLight.setInfluencingBounds(bounds);
    mainBranchGroup.addChild(dirLight);

    // Create the transform group node
    mainTransformGroup = new TransformGroup();
    // Set the appropriate capabilities for the TranformGroup node.

  mainTransformGroup.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);

  mainTransformGroup.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
    mainTransformGroup.setCapability(Node.ENABLE_PICK_REPORTING);
    mainTransformGroup.setCapability(BranchGroup.ALLOW_DETACH);
    mainTransformGroup.setCapability(Group.ALLOW_CHILDREN_EXTEND);
    mainTransformGroup.setCapability(Group.ALLOW_CHILDREN_WRITE);

    mainBranchGroup.setCapability(BranchGroup.ALLOW_DETACH);
    mainBranchGroup.setCapability(Group.ALLOW_CHILDREN_EXTEND);
    mainBranchGroup.setCapability(Group.ALLOW_CHILDREN_WRITE);
    // Add the main TransformGroup node to the main TransformGroup.
    // This means the main transform group will be in charge of all the
    // transformations of the universe as a whole.
    mainBranchGroup.addChild(mainTransformGroup);

    return mainBranchGroup;
  }

  /**
   * This method is called when a menu item is selected, and allows for the
   * relevant task to be carried out.
   *
   * @param e -
   *        provides the menu item which was selected.
   */
  public void actionPerformed(ActionEvent e) {

    // Get the menu item, to be compared to the known items.
    Object target = e.getSource();

    if (target == forwardTrace) {
      // Set global trace direction parameter to forwards.
      traceDirection = 0;
      // Call the trace generating method.
      createTrace();
      // Reset the users perspective.
      resetView();
    }
    if (target == backwardTrace) {
      // Set global trace direction to backwards.
      traceDirection = 1;
      // Call trace creation method.
      createTrace();
      // Reset users perspective.
      resetView();
    }
    if (target == resetView1 || target == resetView2) {
      // Allows the user to reset the view.
      resetView();
    }
    if (target == exitTrace) {
      // If in a trace view, can exit to the main view.
      currentRootNode.removeHighlight();
      currentRootNode.removeObjectLinks();
      currentRootNode.replaceAllObjects();

      resetView();
    }
    if (target == showObjectNames1 || target == showObjectNames2) {
      // Allows the showing of object names.
      if (justSubObjects) {
        // If in the trace view, only create the names of the items in
        // the trace.
        for (IDebugObject i : Object3D.seenObjectList) {
          idoToObject3D.get(i).showDetails();
        }
      } else {
        // Else, create all names for ALL the objects in the collection.
```

```
    for (Object3D o : idoToObject3D.values()) {
      o.showDetails();
    }
  }
}
if (target == hideObjectNames1 || target == hideObjectNames2) {
  // Inverse of above, hiding the object names.
  if (justSubObjects) {
    for (IDebugObject i : Object3D.seenObjectList) {
      idoToObject3D.get(i).hideDetails();
    }
  } else {
    for (Object3D o : idoToObject3D.values()) {
      o.hideDetails();
    }
  }
}
if (target == showLines) {

  // Create the directed lines of the graph.
  for (Object3D o : idoToObject3D.values()) {
    o.showLines();
  }
  // Set global all lines visible to true.
  // If user continues debugging, all new objects will have lines
  // created.
  allLinesVisible = true;
}
if (target == hideLines) {
  // Remove directed lines of the graph.
  for (Object3D o : idoToObject3D.values()) {
    o.hideLines();
  }
  allLinesVisible = false;
}
if (target == showObjectLines) {
  // Create lines for this object only
  currentRightClickedNode.showLines();
}
if (target == hideObjectLines) {
  // Hide lines for this object only
  currentRightClickedNode.hideLines();
}
if (target == redrawSpace) {
  // Allows the space to be redrawn.
  for (Object3D o : idoToObject3D.values()) {
    o.update();
    if (o.linesVisible || allLinesVisible) {
      o.showLines();
    }
  }
  resetView();
}
if (target == showObjectDetails) {
  // Allows the user to view extended details of an object.
  if (objectDetails != null) {
    // If object table already exists, remove it.
    f.remove(objectDetails);
  }
  Vector<String> columnNames = new Vector<String>();
  columnNames.add("Data");
  columnNames.add("Value");

  // Create data vector, add all the information to it.
  Vector<Vector<String>> data = new Vector<Vector<String>>();
  Vector<String> name = new Vector<String>();
  name.add("Name:");
  name.add(currentRightClickedNode.name);
  data.add(name);

  Vector<String> javaType = new Vector<String>();
  javaType.add("Java Type:");
  try {
    javaType.add(currentRightClickedNode.ido.getValue()
        .getJavaType().toString());
  } catch (DebugException e2) {
    e2.printStackTrace();
  }
  data.add(javaType);

  Vector<String> ref = new Vector<String>();
  ref.add("JVM Reference");
```

```
      try {
        ref
            .add(currentRightClickedNode.ido.getValue()
                .getValueString());
      } catch (DebugException e2) {
        e2.printStackTrace();
      }
      data.add(ref);

      try {
        for (IVariable var : currentRightClickedNode.ido.getValue()
            .getVariables()) {
          Vector<String> variable = new Vector<String>();
          variable.add(var.getName());
          variable.add(var.getValue().toString());

          data.add(variable);
        }
      } catch (DebugException e1) {
        e1.printStackTrace();
      }

      // Create table from collected data
      JTable objectDetailsTable = new JTable(data, columnNames);
      objectDetails = new JScrollPane(objectDetailsTable);
      // Add table to frame.
      f.add(objectDetails);
      // pack the frame.
      f.pack();
    }
    if (target == gridView) {
      // Change general layout manager.
      Object3D.layoutManagerType = Object3D.gridtype;
      Object3D.layoutManager = new GridLayout();
      // Update objects to update positions from new layout manager.
      for (Object3D o : idoToObject3D.values()) {
        o.update();
      }
      // All objects created, create lines.
      for (Object3D o : idoToObject3D.values()) {
        if (o.linesVisible) {
          o.hideLines();
          o.showLines();
        } else if (allLinesVisible) {
          o.showLines();
        }
      }
      resetView();
      // Disable relevant menu option.
      gridView.setEnabled(false);
      stackView.setEnabled(true);
      divideResize.setEnabled(true);
      clusteringBased.setEnabled(true);
    }
    if (target == stackView) {
      // Change general layout manager.
      Object3D.layoutManagerType = Object3D.stacktype;
      Object3D.layoutManager = new StackedLayout();
      // Update objects to update positions from new layout manager.
      for (Object3D o : idoToObject3D.values()) {
        o.update();
      }
      // All objects created, create lines.
      for (Object3D o : idoToObject3D.values()) {
        if (o.linesVisible) {
          o.hideLines();
          o.showLines();
        } else if (allLinesVisible) {
          o.showLines();
        }
      }
      resetView();
      // Disable relevant menu option.
      gridView.setEnabled(true);
      stackView.setEnabled(false);
      divideResize.setEnabled(true);
      clusteringBased.setEnabled(true);
    }
    if (target == divideResize) {
      // Change general layout manager.
      Object3D.layoutManagerType = Object3D.rankbased;
      Object3D.layoutManager = new RankBasedLayout();
```

```
      // Update objects to update positions from new layout manager.
      for (Object3D o : idoToObject3D.values()) {
        o.update();
      }
      // All objects created, create lines.
      for (Object3D o : idoToObject3D.values()) {
        if (o.linesVisible) {
          o.hideLines();
          o.showLines();
        } else if (allLinesVisible) {
          o.showLines();
        }
      }
      resetView();
      // Disable relevant menu option.
      gridView.setEnabled(true);
      stackView.setEnabled(true);
      divideResize.setEnabled(false);
      clusteringBased.setEnabled(true);
    }
    if (target == clusteringBased) {
      // Change general layout manager.
      Object3D.layoutManagerType = Object3D.clusterbased;
      Object3D.layoutManager = new ClusteringBasedLayout();
      // Update objects to update positions from new layout manager.
      for (Object3D o : idoToObject3D.values()) {
        o.update();
      }
      // All objects created, create lines.
      for (Object3D o : idoToObject3D.values()) {
        if (o.linesVisible) {
          o.hideLines();
          o.showLines();
        } else if (allLinesVisible) {
          o.showLines();
        }
      }
      resetView();
      // Disable relevant menu option.
      gridView.setEnabled(true);
      stackView.setEnabled(true);
      divideResize.setEnabled(true);
      clusteringBased.setEnabled(false);
    }
  }

  /**
   * This method aims to reset the View, in case the user wishes to return to
   * the default view position.
   */
  private void resetView() {
    mainTransformGroup.setTransform(new Transform3D());
    TransformGroup tg = universe.getViewingPlatform()
        .getViewPlatformTransform();
    transform = new Transform3D();
    transform.set(65.f, new Vector3f(0.0f, 0.0f, 600.0f));
    tg.setTransform(transform);
  }

  /**
   * This method generates a trace based on the current node which has been
   * selected, and a pre-set int representing the direction of the trace.
   */
  public void createTrace() {
    Object3D tempo = currentRightClickedNode;

    Collection<Object3D> c = idoToObject3D.values();

    // Clear the scene graph
    for (Object3D o3d : c) {
      mainTransformGroup.removeChild(o3d.getBranchGroup());
    }

    /*
     * Signify which object is the root. We need to know this for further
     * right click events.
     */
    currentRootNode = tempo;

    // create tree layout for objects.
    if (traceDirection == 0) {
      // Create forward trace.
```

```java
            tempo.createCurrentTree();
            tempo.displayObjectLinks();
        } else if (traceDirection == 1) {
            // Create backward trace
            tempo.createCurrentBackLinkTree();
            tempo.displayObjectBackLinks();
        }
        // We only want to highlight the root node.
        tempo.highlightCurrentObject();
    }

    /**
     * This method generates new Object3D instances. We expect it to be called
     * from the UpdateHandler class when new objects have been generated.
     *
     * @param ido -
     *        the IDebugObject we would like to make an Object3D wrapper
     *        for.
     */
    public void createNew(IDebugObject ido) {

        if (idoToObject3D.isEmpty() || !idoToObject3D.containsKey(ido)) {
            Object3D newObj = new Object3D(ido, this);
            mainTransformGroup.addChild(newObj.getBranchGroup());
            idoToObject3D.put(ido, newObj);
        }
    }

    /**
     * If the underlying system removes an object, we must remove it from our 3D
     * graph.
     *
     * @param ido -
     *        The IDebugObject which has been removed.
     */
    public void remove(IDebugObject ido) {

        mainTransformGroup.removeChild(idoToObject3D.get(ido).getBranchGroup());
        idoToObject3D.remove(ido);
    }

    /**
     * Dealing with user interaction.
     *
     * Alt+Left-Click = Reset View.
     * Left-Click on Object = Generate name for that object.
     * In general view:
     * right-click in `space' = Create main menu.
     * right-click on object = Create Object menu.
     * In Trace view:
     * right-click in `space' = Create trace menu.
     * right-click on root = go back to general view.
     * right-click on child node = create selected objects trace.
     * @param e -
     *        The MouseEvent received from which we can decipher what action
     *        must be taken.
     */
    public void mouseClicked(MouseEvent e) {

        // Alt+ Left-Click resets the view.
        if (e.isAltDown() && e.getButton() == MouseEvent.BUTTON1) {
            resetView();
        }
        // Left-Click generates name of object selected.
        else if (e.getButton() == MouseEvent.BUTTON1) {

            pickCanvas.setShapeLocation(e);
            // Pick object in that position.
            PickResult result = pickCanvas.pickClosest();
            if (result == null) {
                // Nothing Picked, do nothing.
            } else {

                // Get Object selected.
                Primitive p = (Primitive) result.getNode(PickResult.PRIMITIVE);

                if (p != null) {
                    // Get Object3D wrapper for selected object.
                    Object3D tempo = tgToObject3D.get(p.getParent());
                    // Show/hide name object for picked node.
                    if (!tempo.detailsVisible) {
                        tempo.showDetails();
```

```
          } else {
            tempo.hideDetails();
          }
        }
      }
    }
    // In general view, right-click generates main menu, or object menu
    // dependent on whether object selected or not.
    else if (!justSubObjects && e.getButton() == MouseEvent.BUTTON3) {

      pickCanvas.setShapeLocation(e);
      PickResult result = pickCanvas.pickClosest();

      if (result == null) {
        // Nothing picked, show main menu.
        mainMenu.show(e.getComponent(), e.getX(), e.getY());
      } else {
        // Create object menu, for this object, setting current right
        // clicked node parameter.
        objectMenu.show(e.getComponent(), e.getX(), e.getY());
        Primitive p = (Primitive) result.getNode(PickResult.PRIMITIVE);
        Object3D tempo = tgToObject3D.get(p.getParent());
        currentRightClickedNode = tempo;
      }
    }
    // In Trace view, right-click generates trace menu if `space' clicked,
    // if root node picked, we return to general view, else we create trace
    // for selected object.
    else if (justSubObjects && e.getButton() == MouseEvent.BUTTON3) {

      pickCanvas.setShapeLocation(e);
      PickResult result = pickCanvas.pickClosest();

      if (result == null) {
        // Nothing picked, show trace menu.
        subObjectsMenu.show(e.getComponent(), e.getX(), e.getY());
      } else {

        Primitive p = (Primitive) result.getNode(PickResult.PRIMITIVE);

        if (p != null) {

          Object3D tempo = tgToObject3D.get(p.getParent());

          if (tempo == currentRootNode) {
            // Go back to general view.
            tempo.removeHighlight();
            tempo.removeObjectLinks();
            tempo.replaceAllObjects();
            resetView();
          } else {
            /*
             * In this situation, the user probably want to pick the
             * tree corresponding to the clicked on object. We must
             * therefore reset the view, and perform the operation
             * for the new object.
             */
            currentRootNode.removeHighlight();
            currentRootNode.removeObjectLinks();
            currentRootNode.replaceAllObjects();

            currentRightClickedNode = tempo;
            // Create trace takes into account the trace direction.
            createTrace();
            // Centre the root node.
            resetView();
          }
        }
      }
    }
  }
}
```

```java
package view.interfaces;

import java.util.LinkedList;
import javax.vecmath.Vector3d;
import view.views.Object3D;

/**
 * This class serves as a controller for the positions of each Object3D in the
 * system.
 *
 * @author Darius Bradbury
 */
public interface LayoutManager3D {

  /**
   * Maintained current ranking list, updated each time updateAllPositions is
   * called.
   */
  public static LinkedList<Object3D> currentRanking
    = new LinkedList<Object3D>();

  /**
   * @param o3d -
   *        the Object3D we want the position of.
   * @return A three-dimensional vector representing it's position.
   */
  public Vector3d getPosition(Object3D o3d);

  /**
   * This method tells the Layout Manager to reconsider its position values.
   * We call this method when the underlying model changes.
   */
  public void updateAllPositions();
}
```

```
/**
 * The GridLayout Class:
 * This class aims to maintain a grid of 3D vector
 * positions. New positions are created as new objects are passed into the
 * model.
 *
 * @author Darius Bradbury.
 */
public class GridLayout implements LayoutManager3D {

  // Storage of Object3D to position vectors.
  private HashMap<Object3D, Vector3d> o3dVectorMap;
  // Current position in the grid.
  private Vector3d curPos;

  /**
   * Instantiate object, and set initial grid position.
   */
  public GridLayout() {
    o3dVectorMap = new HashMap<Object3D, Vector3d>();
    curPos = new Vector3d(-50, 30, 0);
  }

  /**
   * This method creates a new 3D vector for the given Object3D object.
   *
   * @param o3d -
   *        Object wanting new grid position.
   * @return - Vector corresponding to that Object3D's position.
   */
  private Vector3d createNewPosition(Object3D o3d) {

    // Make sure we haven't gone past the screens width.
    if (curPos.getX() > (View3D.f.getBounds().height *
        View3D.wideScreenRatio) / 10) {
      // If gone past screen width, drop down a line, and go back to
      // initial X-axis position.
      curPos.setX(-50);
      curPos.setY(curPos.getY() - 25);
    }
    // Generate a new vector for current position.
    Vector3d thisVec = new Vector3d(curPos);
    // Move vector along.
    curPos.setX(curPos.getX() + 25);
    // Place this vector into the map.
    o3dVectorMap.put(o3d, thisVec);
    // Return newly generated vector.
    return thisVec;
  }

  /**
   * This is a public method designed to return the position of the given
   * Object3D. If it's never been seen, create a new one, else pass on old
   * position.
   *
   * @param o3d -
   *        Object querying for it's position vector.
   * @return - 3D Vector representing its position.
   */
  public Vector3d getPosition(Object3D o3d) {

    if (o3dVectorMap.isEmpty() || !o3dVectorMap.containsKey(o3d)) {
      // Never seen this Object3D, thus create new position.
      return new Vector3d(createNewPosition(o3d));
    } else {
      // Seen this Object3D before, return it's position vector.
      Vector3d pos = o3dVectorMap.get(o3d);
      return new Vector3d(pos);
    }
  }

  /**
   * This is a required method for all subclasses of the LayoutManager class.
   * We require it to maintain the ranking of the objects when called, this
   * allows for proper resizing of the objects when the underlying state
   * changes.
   *
   */
  public void updateAllPositions() {

    // First extract all the Object3D objects still in our system.
    Collection<Object3D> totalListOfObjects = View3D.idoToObject3D.values();
```

```
        LinkedList<Object3D> totalRankedListOfObjects = new LinkedList<Object3D>(
                totalListOfObjects);

        // Sort the collection based on rank
        Collections.sort(totalRankedListOfObjects, new Comparator<Object3D>() {
            public int compare(Object3D arg0, Object3D arg1) {
                double diff = arg0.ido.getPageRank() - arg1.ido.getPageRank();
                if (diff > 0) {
                    return -1;
                } else if (diff < 0) {
                    return 1;
                } else {
                    return 0;
                }
            }
        });
        // Clear current ranking.
        currentRonking.clear();
        // Save this total object ranking.
        currentRonking.addAll(totalRankedListOfObjects);
    }
}
```

```java
/**
 * The RankBasedLayout Class:
 * This class performs the Divide and Resize process to distribute the objects,
 * providing a layout manager to access the positions for each Object3D object.
 *
 * @author Darius Bradbury
 */
public class RankBasedLayout implements LayoutManager3D {
  // Storage of Object3D to position vectors.
  private HashMap<Object3D, Vector3d> o3dVectorMap =
    new HashMap<Object3D, Vector3d>();
  // Locally stored ranked list of objects, used to generate positions.
  private LinkedList<Object3D> totalRankedListOfObjects;
  // The radius of the 3D sphere we are to contain our objects within.
  private double totalRadius;

  /**
   * We instantiate a new RankBasedLayout manager, update the
   * current list of objects, and define the size of the 3D space we are to
   * contain our objects within.
   */
  public RankBasedLayout() {
    createRankedListOfObjects();
    totalRadius = 100;
  }

  /**
   * This method creates, or updates, our ranked list of objects. It is called
   * each time the underlying state changes, and is used in generating the
   * layout.
   */
  private void createRankedListOfObjects() {

    // First extract all the Object3D objects still in our system.
    Collection<Object3D> totalListOfObjects = View3D.idoToObject3D.values();

    totalRankedListOfObjects = new LinkedList<Object3D>(totalListOfObjects);

    // Sort the collection based on rank
    Collections.sort(totalRankedListOfObjects, new Comparator<Object3D>() {
      public int compare(Object3D arg0, Object3D arg1) {
        double diff = arg0.ido.getPageRank() - arg1.ido.getPageRank();
        if (diff > 0) {
          return -1;
        } else if (diff < 0) {
          return 1;
        } else {
          return 0;
        }
      }
    });
    // Clear current ranking
    currentRanking.clear();
    // Save this total object ranking.
    currentRanking.addAll(totalRankedListOfObjects);
  }

  /**
   * Performs a BFS to create all nodes in order of rank. This method is only
   * called once, and creates positions for all the objects when called.
   *
   * @param o3d -
   *         The Object3D wishing to get it's position vector.
   */
  private Vector3d createNewPosition(Object3D o3d) {

    // Create all positions.
    createPositions(new Vector3d(0, 0, 0), totalRadius, 6,
        totalRankedListOfObjects);
    // Return position for given Object3D.
    return o3dVectorMap.get(o3d);
  }

  /**
   * This method updates our ranked list of objects, and then creates the
   * positions for all Object3D objects based on our new ranking.
   */
  public void updateAllPositions() {
    // Create Ranking.
    createRankedListOfObjects();
    // Create Positions.
    createPositions(new Vector3d(0, 0, 0), totalRadius, 6,
```

```java
}

/*
 * (non-Javadoc)
 *
 * @see view.interfaces.LayoutManager3D#getPosition(view.views.Object3D)
 */
public Vector3d getPosition(Object3D o3d) {

  if (o3dVectorMap.isEmpty()) {
    // If map empty, create ranked list, and all positions.
    createRankedListOfObjects();
    Vector3d v3d = createNewPosition(o3d);
    return v3d;
  } else if (!o3dVectorMap.containsKey(o3d)) {
    // If map non-empty, but doesn't contain given Object3D, clear the
    // mapping, recreate our ranked list, and recreate alll positions.
    o3dVectorMap.clear();
    createRankedListOfObjects();
    return createNewPosition(o3d);
  } else {
    // Position in map, just return it.
    return o3dVectorMap.get(o3d);
  }
}

/**
 * This method takes the root position for this rankBased Layout, the root
 * object, the radius of the sphere within it must work, and the direction
 * from which it was generated.
 *
 * 0 means it cam from -inf(x)
 * 1 means it came from +inf(x)
 * 2 means it came from -inf(y)
 * 3 means it came from +inf(y)
 * 4 means it came from -inf(z)
 * 5 means it came from +inf(z)
 * 6 means it's the root, and can go out in all directions.
 *
 * It then creates positions for each of the positions in the given list.
 *
 * @param root -
 *        Our root position, the starting point for space generation.
 * @param radius -
 *        Radius of the sphere of 3D Space allotted for our objects.
 * @param cameFrom -
 *        Direction came from relative to parent Object3D.
 * @param rankedListOfObjects -
 *        The Object3Ds to distribute in this space.
 */
private void createPositions(Vector3d root, double radius, int cameFrom,
    LinkedList<Object3D> rankedListOfObjects) {

  // Ascertain the number of objects we must distribute.
  int numberOfOBjects = rankedListOfObjects.size();

  // Place root node in position.
  Object3D rootNode = rankedListOfObjects.removeFirst();
  o3dVectorMap.put(rootNode, root);

  // Once placed, add to totalSeen set, so it is no longer considered by
  // sub-groups.
  totalSeen.add(rootNode);

  // Create list of lists representing groups of objects.
  // Do NOT destroy rankedListOfObjects.
  LinkedList<LinkedList<Object3D>> groups = getGroups(rankedListOfObjects);

  // Create sub-lists - we want to keep similar objects together.
  LinkedList<Object3D> ll0 = new LinkedList<Object3D>();
  LinkedList<Object3D> ll1 = new LinkedList<Object3D>();
  LinkedList<Object3D> ll2 = new LinkedList<Object3D>();
  LinkedList<Object3D> ll3 = new LinkedList<Object3D>();
  LinkedList<Object3D> ll4 = new LinkedList<Object3D>();
  LinkedList<Object3D> ll5 = new LinkedList<Object3D>();

  // Start from last direction used. This means we get a more even
  // distribution of directions within our space.
  // We could use a random number for even distribution, but we want our
  // visualisations to be the same each time.
  int i = directioni;
  // Add groups of nodes at a time, as each group represents similar
```

```java
        ll1.add(rankedListOfObjects.removeFirst());
        break;
      case 2:
        i++;
        if (cameFrom == 2) {
          break;
        }
        ll2.add(rankedListOfObjects.removeFirst());
        break;
      case 3:
        i++;
        if (cameFrom == 3) {
          break;
        }
        ll3.add(rankedListOfObjects.removeFirst());
        break;
      case 4:
        i++;
        if (cameFrom == 4) {
          break;
        }
        ll4.add(rankedListOfObjects.removeFirst());
        break;
      case 5:
        i = 0;
        if (cameFrom == 5) {
          break;
        }
        ll5.add(rankedListOfObjects.removeFirst());
        break;
      }
    }

    /*
     * Create positions for the sub-lists, each time halving their space,
     * and repositioning their root. We do this in order to ensure that each
     * sub-space doesn't "grow" towards it's parent node.
     */

    if (cameFrom != 0 && !ll0.isEmpty()) {
      createPositions(new Vector3d(root.getX() - radius, root.getY(),
          root.getZ()), radius / 2, 1, ll0);
    }
    if (cameFrom != 1 && !ll1.isEmpty()) {
      createPositions(new Vector3d(root.getX() + radius, root.getY(),
          root.getZ()), radius / 2, 0, ll1);
    }
    if (cameFrom != 2 && !ll2.isEmpty()) {
      createPositions(new Vector3d(root.getX(), root.getY() - radius,
          root.getZ()), radius / 2, 3, ll2);
    }
    if (cameFrom != 3 && !ll3.isEmpty()) {
      createPositions(new Vector3d(root.getX(), root.getY() + radius,
          root.getZ()), radius / 2, 2, ll3);
    }
    if (cameFrom != 4 && !ll4.isEmpty()) {
      createPositions(new Vector3d(root.getX(), root.getY(), root.getZ()
          - radius), radius / 2, 5, ll4);
    }
    if (cameFrom != 5 && !ll5.isEmpty()) {
      createPositions(new Vector3d(root.getX(), root.getY(), root.getZ()
          + radius), radius / 2, 4, ll5);
    }
  }
}
```

```java
/**
 * The ClusteringBasedLayout Class:
 * This class performs a clustering algorithm
 * to distribute the objects, providing a layout manager to access the positions
 * for each Object3D object.
 *
 * @author Darius Bradbury
 */
public class ClusteringBasedLayout implements LayoutManager3D {

  // Storage of Object3D to position vectors.
  private HashMap<Object3D, Vector3d> o3dVectorMap =
    new HashMap<Object3D, Vector3d>();
  // Our local ranked list of objects, used in creating positions.
  public LinkedList<Object3D> totalRankedListOfObjects;
  private HashSet<Object3D> totalSeen; // Maintains placed objects.
  private double totalRadius; // Size of space we initially work with.
  private int directioni = 0; // Direction we grow into.

  /**
   * We instantiate a new ClusteringBasedLayout manager, update the current
   * list of objects, and define the size of the 3D space we are to contain
   * our objects within.
   */
  public ClusteringBasedLayout() {
    createRankedListOfObjects();
    totalRadius = 100;
    // Create seen object list.
    totalSeen = new HashSet<Object3D>();
  }

  private void createRankedListOfObjects() {

    // First extract all the Object3D objects still in our system.
    Collection<Object3D> totalListOfObjects = View3D.idoToObject3D.values();

    totalRankedListOfObjects = new LinkedList<Object3D>(totalListOfObjects);

    // Sort the collection based on rank
    Collections.sort(totalRankedListOfObjects, new Comparator<Object3D>() {
      public int compare(Object3D arg0, Object3D arg1) {
        double diff = arg0.ido.getPageRank() - arg1.ido.getPageRank();
        if (diff > 0) {
          return -1;
        } else if (diff < 0) {
          return 1;
        } else {
          return 0;
        }
      }
    });
    // Clear current ranking
    currentRanking.clear();
    // Save this total object ranking.
    currentRanking.addAll(totalRankedListOfObjects);
  }

  /**
   * Performs a BFS to create all nodes in order of RANK
   */
  private Vector3d createNewPosition(Object3D o3d) {

    createPositions(new Vector3d(0, 0, 0), totalRadius, 6,
        totalRankedListOfObjects);
    return o3dVectorMap.get(o3d);
  }

  /*
   * (non-Javadoc)
   *
   * @see view.interfaces.LayoutManager3D#updateAllPositions()
   */
  public void updateAllPositions() {
    // Reset parameters.
    directioni = 0;
    totalSeen.clear();
    o3dVectorMap.clear();
    // Recreate local ranked list.
    createRankedListOfObjects();
    // Create new positions.
    createPositions(new Vector3d(0, 0, 0), totalRadius, 6,
        totalRankedListOfObjects);
```

```
        totalRankedListOfObjects);
}

/*
 * (non-Javadoc)
 * @see view.interfaces.LayoutManager3D#getPosition(view.views.Object3D)
 */
public Vector3d getPosition(Object3D o3d) {

  if (o3dVectorMap.isEmpty()) {
    // If map empty, create ranked list, and all positions.
    createRankedListOfObjects();
    Vector3d v3d = createNewPosition(o3d);
    return v3d;
  } else if (!o3dVectorMap.containsKey(o3d)) {
    // If map non-empty, but doesn't contain given Object3D, clear the
    // mapping, recreate our ranked list, and recreate alll positions.
    o3dVectorMap.clear();
    createRankedListOfObjects();
    return createNewPosition(o3d);
  } else {
    // Position in map, just return it.
    return o3dVectorMap.get(o3d);
  }
}

/**
 * This method creates the vector positions for the given Object3D's.
 * The came from location tells us the location of this sub-space, relative
 * to its parent's space:
 *
 * 0 means it came from -inf(x)
 * 1 means it came from +inf(x)
 * 2 means it came from -inf(y)
 * 3 means it came from +inf(y)
 * 4 means it came from -inf(z)
 * 5 means it came from +inf(z)
 * 6 means it's the root, and can go out in all directions.
 *
 * @param root -
 *        Our root position, and centroid of space for given Object3D's.
 * @param radius -
 *        Radius of the sphere of 3D Space allotted for our objects.
 * @param cameFrom -
 *        Direction came from relative to parent Object3D.
 * @param rankedListOfObjects -
 *        The Object3Ds to distribute in this space.
 */
private void createPositions(Vector3d root, double radius, int cameFrom,
    LinkedList<Object3D> rankedListOfObjects) {

  // Place root node in position.
  o3dVectorMap.put(rankedListOfObjects.removeFirst(), root);

  // Create sub-lists.
  LinkedList<Object3D> ll0 = new LinkedList<Object3D>();
  LinkedList<Object3D> ll1 = new LinkedList<Object3D>();
  LinkedList<Object3D> ll2 = new LinkedList<Object3D>();
  LinkedList<Object3D> ll3 = new LinkedList<Object3D>();
  LinkedList<Object3D> ll4 = new LinkedList<Object3D>();
  LinkedList<Object3D> ll5 = new LinkedList<Object3D>();

  /*
   * Divide List up into 5 or 6 depending on cameFrom location We evenly
   * distribute our Object3D's over the lists and ensure that each list
   * preserves its rank order.
   */
  int i = 0;
  while (!rankedListOfObjects.isEmpty()) {
    switch (i) {
    case 0:
      i++;
      if (cameFrom == 0) {
        break;
      }
      ll0.add(rankedListOfObjects.removeFirst());
      break;
    case 1:
      i++;
      if (cameFrom == 1) {
        break;
      }
```

```
// objects.
while (!groups.isEmpty()) {
  switch (i) {
  case 0:
    i++;
    if (cameFrom == 0) {
      break;
    }
    ll0.addAll(groups.removeFirst());
    break;
  case 1:
    i++;
    if (cameFrom == 1) {
      break;
    }
    ll1.addAll(groups.removeFirst());
    break;
  case 2:
    i++;
    if (cameFrom == 2) {
      break;
    }
    ll2.addAll(groups.removeFirst());
    break;
  case 3:
    i++;
    if (cameFrom == 3) {
      break;
    }
    ll3.addAll(groups.removeFirst());
    break;
  case 4:
    i++;
    if (cameFrom == 4) {
      break;
    }
    ll4.addAll(groups.removeFirst());
    break;
  case 5:
    i = 0;
    if (cameFrom == 5) {
      break;
    }
    ll5.addAll(groups.removeFirst());
    break;
  }
}
directioni = i;

/*
 * Here we distribute the objects based on how many we are dealing with.
 * If we have over 50, we "grow" our graph, such that, we move outside
 * of our given bounds, however, we only grow "outwards", not towards
 * our parent node. Otherwise, we stick to the space we have, and
 * generate this space as in the Divide and Resize algorithm.
 */
if (numberOfOBjects > 50) {
  // Check positions are free, if not, put into guaranteed free
  // direction

  // Set toPosition to represent moving in the negative X-axis
  // direction.
  Vector3d toPosition = new Vector3d(root.getX() - radius, root
      .getY(), root.getZ());
  // Check no node already exists there.
  if (o3dVectorMap.containsValue(toPosition)) {
    // If node exists, pass these elements to a different direction
    // list.
    ll1.addAll(ll0);
    ll0.clear();
  }
  // Set toPosition to represent moving in the positive X-axis
  // direction.
  toPosition = new Vector3d(root.getX() + radius, root.getY(), root
      .getZ());
  if (o3dVectorMap.containsValue(toPosition)) {
    ll2.addAll(ll1);
    ll1.clear();
  }
  // Set toPosition to represent moving in the negative Y-axis
  // direction.
  toPosition = new Vector3d(root.getX(), root.getY() - radius, root
```

```
            .getZ());
        if (o3dVectorMap.containsValue(toPosition)) {
          ll3.addAll(ll2);
          ll2.clear();
        }
        // Set toPosition to represent moving in the positive Y-axis
        // direction.
        toPosition = new Vector3d(root.getX(), root.getY() + radius, root
            .getZ());
        if (o3dVectorMap.containsValue(toPosition)) {
          ll4.addAll(ll3);
          ll3.clear();
        }
        // Set toPosition to represent moving in the negative Z-axis
        // direction.
        toPosition = new Vector3d(root.getX(), root.getY(), root.getZ()
            - radius);
        if (o3dVectorMap.containsValue(toPosition)) {
          ll5.addAll(ll4);
          ll4.clear();
        }
        // Set toPosition to represent moving in the positive Z-axis
        // direction.
        toPosition = new Vector3d(root.getX(), root.getY(), root.getZ()
            + radius);
        if (o3dVectorMap.containsValue(toPosition)) {
          // If we find positive Z-axis contains a node, we put nodes into
          // guaranteed free direction.
          // Namely, away from our cameFrom location!
          if (cameFrom == 0) {
            ll1.addAll(ll5);
          }
          if (cameFrom == 1) {
            ll0.addAll(ll5);
          }
          if (cameFrom == 2) {
            ll3.addAll(ll5);
          }
          if (cameFrom == 3) {
            ll2.addAll(ll5);
          }
          if (cameFrom == 5) {
            ll4.addAll(ll5);
          }
        }

        /*
         * We now create the positions by iteratively calling this method
         * again. However, not that we don't change the radius size, and we
         * move along by the whole radius size.
         */
        if (cameFrom != 0 && !ll0.isEmpty()) {
          toPosition = new Vector3d(root.getX() - radius, root.getY(),
              root.getZ());
          createPositions(toPosition, radius, 1, ll0);
        }
        if (cameFrom != 1 && !ll1.isEmpty()) {
          toPosition = new Vector3d(root.getX() + radius, root.getY(),
              root.getZ());
          createPositions(toPosition, radius, 0, ll1);
        }
        if (cameFrom != 2 && !ll2.isEmpty()) {
          toPosition = new Vector3d(root.getX(), root.getY() - radius,
              root.getZ());
          createPositions(toPosition, radius, 3, ll2);
        }
        if (cameFrom != 3 && !ll3.isEmpty()) {
          toPosition = new Vector3d(root.getX(), root.getY() + radius,
              root.getZ());
          createPositions(toPosition, radius, 2, ll3);
        }
        if (cameFrom != 4 && !ll4.isEmpty()) {
          toPosition = new Vector3d(root.getX(), root.getY(), root.getZ()
              - radius);
          createPositions(toPosition, radius, 5, ll4);
        }
        if (cameFrom != 5 && !ll5.isEmpty()) {
          toPosition = new Vector3d(root.getX(), root.getY(), root.getZ()
              + radius);
          createPositions(toPosition, radius, 4, ll5);
        }
      }
```

```
        /*
         * If we have under 50 objects to place in our given space then we
         * perform the normal Divide and Resize algorithm.
         */
        else {
          // Direction to move root for current sub-object list.
          Vector3d toPosition;

          // Note that we half the radius given to our sub objects list in
          // this instance.
          if (cameFrom != 0 && !ll0.isEmpty()) {
            toPosition = new Vector3d(root.getX() - radius, root.getY(),
                root.getZ());
            createPositions(toPosition, radius / 2, 1, ll0);
          }
          if (cameFrom != 1 && !ll1.isEmpty()) {
            toPosition = new Vector3d(root.getX() + radius, root.getY(),
                root.getZ());
            createPositions(toPosition, radius / 2, 0, ll1);
          }
          if (cameFrom != 2 && !ll2.isEmpty()) {
            toPosition = new Vector3d(root.getX(), root.getY() - radius,
                root.getZ());
            createPositions(toPosition, radius / 2, 3, ll2);
          }
          if (cameFrom != 3 && !ll3.isEmpty()) {
            toPosition = new Vector3d(root.getX(), root.getY() + radius,
                root.getZ());
            createPositions(toPosition, radius / 2, 2, ll3);
          }
          if (cameFrom != 4 && !ll4.isEmpty()) {
            toPosition = new Vector3d(root.getX(), root.getY(), root.getZ()
                - radius);
            createPositions(toPosition, radius / 2, 5, ll4);
          }
          if (cameFrom != 5 && !ll5.isEmpty()) {
            toPosition = new Vector3d(root.getX(), root.getY(), root.getZ()
                + radius);
            createPositions(toPosition, radius / 2, 4, ll5);
          }
        }
      }

      /**
       * We calculate groups based on contexts. We remove all nodes already placed
       * in graph from context, and thus group or cluster these elements based on
       * links without the parent node, and hence, all links reachable from it,
       * but not from within the group members directly. In other words, the
       * context of a node is all the nodes it can reach, without going through
       * the objects already placed in the graph.
       *
       * In this way, we split the graph into it's sub-graphs.
       *
       * @param rankedListOfObjects -
       *          objects in this part of the 3D graph.
       * @return List of related groups.
       */
      private LinkedList<LinkedList<Object3D>> getGroups(
          LinkedList<Object3D> inputList) {

        // Set our seen set, to all the objects PLACED in the map.
        HashSet<Object3D> seen = new HashSet<Object3D>(totalSeen);

        // Create ranked list of objects based on input set (which is already in
        // order.)
        LinkedList<Object3D> rankedListOfObjects = new LinkedList<Object3D>(
            inputList);
        // Create list of lists.
        LinkedList<LinkedList<Object3D>> groups = new
LinkedList<LinkedList<Object3D>>();

        for (Object3D o3d : rankedListOfObjects) {
          // We only want to create new groups for UNSEEN objects.
          if (!seen.contains(o3d)) {
            LinkedList<Object3D> group = new LinkedList<Object3D>();
            // Add to seen list, as we don't want to pass through this node
            // again.
            seen.add(o3d);
            // Add to current group.
            group.add(o3d);

            // We then find related items to this o3d, and place into this
```

```
// list.
LinkedList<Object3D> contextList = new LinkedList<Object3D>();

try {
  // Look at forward links.
  for (Entry<IDebugObject, IVariable> variableLink : o3d.ido
      .objectLinks().entrySet()) {
    Object3D forwardLinkObject = View3D.idoToObject3D
        .get(variableLink.getKey());
    if (!seen.contains(forwardLinkObject)) {
      // Unseen node, so add to current context,
      // overall group, and seen list.
      contextList.add(forwardLinkObject);
      seen.add(forwardLinkObject);
      group.add(forwardLinkObject);
    }
  }
  // Look at backward links.
  for (Entry<IDebugObject, IVariable> variableLink : o3d.ido
      .backLinks().entrySet()) {
    Object3D backwardLinkObject = View3D.idoToObject3D
        .get(variableLink.getKey());
    if (!seen.contains(backwardLinkObject)) {
      // Unseen node, so add to current context,
      // overall group, and seen list.
      contextList.add(backwardLinkObject);
      seen.add(backwardLinkObject);
      group.add(backwardLinkObject);
    }
  }
} catch (NullLinkException e) {
  e.printStackTrace();
}

// Now iterate through this objects context nodes.
while (!contextList.isEmpty()) {
  Object3D newContextObject = contextList.remove();
  try {
    // look at forward links.
    for (Entry<IDebugObject, IVariable> variableLink :
        newContextObject.ido.objectLinks().entrySet()) {
      Object3D forwardLinkObject = View3D.idoToObject3D
          .get(variableLink.getKey());
      if (!seen.contains(forwardLinkObject)) {
        // Add to current context.
        contextList.add(forwardLinkObject);
        // Add to seen nodes.
        seen.add(forwardLinkObject);
        // Add to current group.
        group.add(forwardLinkObject);
      }
    }
    // look at backward links.
    for (Entry<IDebugObject, IVariable> variableLink :
        newContextObject.ido.backLinks().entrySet()) {
      Object3D backwardLinkObject = View3D.idoToObject3D
          .get(variableLink.getKey());
      if (!seen.contains(backwardLinkObject)) {
        // Add to current context.
        contextList.add(backwardLinkObject);
        // Add to seen nodes.
        seen.add(backwardLinkObject);
        // Add to current group.
        group.add(backwardLinkObject);
      }
    }
  } catch (NullLinkException e) {
    e.printStackTrace();
  }
}

// Sort our new group list based on importance.

Collections.sort(group, new Comparator<Object3D>() {
  public int compare(Object3D arg0, Object3D arg1) {
    double diff = arg0.ido.getPageRank()
        - arg1.ido.getPageRank();
    if (diff > 0) {
      return -1;
    } else if (diff < 0) {
      return 1;
    } else {
```

```
                return 0;
            }
        }
    });
    // Add this group to our overall set of groups.
    groups.add(group);
  }
}
return groups;
}

}
```

```java
/**
 * The TreeLayout Class:
 * This class controls the positioning of the all the
 * objects in a forward trace, given a root node. It performs a Breadth-First
 * search to do this.
 *
 * @author Darius Bradbury
 */
public class TreeLayoutC implements LayoutManager3D {

  // IDebugObject to Position Vector mapping.
  private HashMap<IDebugObject, Vector3d> idoVectorMap;
  // Position of the root.
  public Vector3d rootPos;
  // Current position in the tree.
  public Vector3d curPos;
  // List of seen objects, to cope with loops.
  LinkedList<IDebugObject> seenList;
  // Map of IDebugObjects to their sub-tree size.
  private HashMap<IDebugObject, Integer> sizeMap;

  /**
   * Creates a new Tree Layout Manager, resetting the root and current
   * position vectors.
   */
  public TreeLayoutC() {
    idoVectorMap = new HashMap<IDebugObject, Vector3d>();
    rootPos = new Vector3d(0, 30, 0);
    curPos = new Vector3d(0, 30, 0);
  }

  /**
   * This method creates a new position for the given Object3D object, in
   * doing so, it creates positions for all Object3D's in its forward trace
   * subtree, and sets the given node as the root.
   *
   * @param o3d -
   *        Object3D not in map, thus needing its position.
   * @return 3D Vector representing Object3D's position.
   * @throws NullLinkException
   *         due to the IDebugObject's link extraction method.
   */
  public Vector3d createNewPosition(Object3D o3d) throws NullLinkException {

    // Create fresh list of seen nodes.
    seenList = new LinkedList<IDebugObject>();
    // Create fresh map of sub-tree sizes;
    sizeMap = new HashMap<IDebugObject, Integer>();
    // Calculate the size of this IDebugObjects sub-tree, and all the
    // IDebugObjects within that sub-tree.
    getSize(o3d.ido);
    // Create root position, place given node in root position.
    Vector3d thisVec = new Vector3d(rootPos);
    idoVectorMap.put(o3d.ido, thisVec);

    // Create all the nodes, and leaves.

    // Forward links container mapping.
    Map<IDebugObject, IVariable> linklist;
    // Put objects AND primitives.
    linklist = o3d.ido.objectLinks();

    // List of link entries.
    LinkedList<Entry<IDebugObject, IVariable>> children =
      new LinkedList<Entry<IDebugObject, IVariable>>();
    // Seen list for this pass.
    LinkedList<IDebugObject> seen = new LinkedList<IDebugObject>();
    // Add root node to seen list.
    seen.add(o3d.ido);

    // Iterate through each IDebugObject our root points to.
    for (Entry<IDebugObject, IVariable> ido : linklist.entrySet()) {
      if (!seen.contains(ido.getKey())) {
        // If not in seen list, add to seen list, add to children.
        children.add(ido);
        seen.add(ido.getKey());
      }
    }

    // Create List of lists representing levels of the tree.
    LinkedList<LinkedList<IDebugObject>> levelsOfChildren = new
LinkedList<LinkedList<IDebugObject>>();
```

```java
    // Create temporary list containing working level.
    LinkedList<IDebugObject> thisLevel = new LinkedList<IDebugObject>();
    // indicator for level change.
    int levelIndicator = children.size();

    /*
     * Here we create a list of lists containing the objects of each level.
     * In other words, we are creating a list of the levels by performing a
     * BFS, once all nodes in a level have been consumed, we generate a new
     * list.
     */
    while (!children.isEmpty()) {
      if (levelIndicator == 0) {
        // We know we have come to the end of this level, must create a
        // new one.
        levelIndicator = children.size();
        levelsOfChildren.add(thisLevel);
        // Create new working list for new level.
        thisLevel = new LinkedList<IDebugObject>();
      }

      // look at current child from list, ie. BFS.
      IDebugObject curChild = children.removeFirst().getKey();

      // Iterate through child's forward links.
      for (Entry<IDebugObject, IVariable> variableLink : curChild
          .objectLinks().entrySet()) {
        if (!seen.contains(variableLink.getKey())) {
          children.add(variableLink);
          seen.add(variableLink.getKey());
        }
      }

      // Add current child node to level list.
      thisLevel.add(curChild);
      // Decrement level counter, so we know when level has finished.
      levelIndicator--;
    }

    // Add final level to overall levels.
    levelsOfChildren.add(thisLevel);

    // Now create positions from this list of levels.
    // Iterating through each level, until leaves have been reached.
    // In other words, we place nodes, on a level-at-a-time basis.
    while (!levelsOfChildren.isEmpty()) {

      // New level, so we drop down a level in our 3D space.
      curPos.setY(curPos.getY() - 25);

      // Take current level from list of levels.
      LinkedList<IDebugObject> currentLevel = levelsOfChildren.remove();

      /* Calculate space required */
      int currentLevelSize = 0;
      // Level size determined by the sum of the size of each child.
      for (IDebugObject i : currentLevel) {
        currentLevelSize += getSize(i);
      }

      // Move horizontal position all the way to the left.
      curPos.setX(rootPos.getX() - ((currentLevelSize * 25) / 2));

      // Iterate through children, and place them.
      while (!currentLevel.isEmpty()) {
        // Get child.
        IDebugObject currentObj = currentLevel.removeFirst();
        // get child's size.
        int currentObjSize = getSize(currentObj);
        // Move position in relation to the child's size, such that
        // all its children will fit underneath it.
        curPos.setX((currentObjSize * 25) / 2 + curPos.getX());
        // Place object.
        idoVectorMap.put(currentObj, new Vector3d(curPos));
        // As object placed in the middle of this space, move over to
        // the edge, such that a new object can be placed.
        curPos.setX((currentObjSize * 25) / 2 + curPos.getX());
      }
    }
    // Return root position.
    return idoVectorMap.get(o3d.ido);
  }
```

```java

/*
 * (non-Javadoc)
 *
 * @see view.interfaces.LayoutManager3D#getPosition(view.views.Object3D)
 */
public Vector3d getPosition(Object3D o3d) {

  if (idoVectorMap.isEmpty() || !idoVectorMap.containsKey(o3d.ido)) {
    // Need to create the tree with this Object3D as the root.
    try {
      // Thus create tree with o3d as root.
      return new Vector3d(createNewPosition(o3d));
    } catch (NullLinkException e) {
      throw new RuntimeException(e);
    }
  } else {
    // Object3D already in tree, thus, just return its position.
    Vector3d pos = idoVectorMap.get(o3d.ido);
    return new Vector3d(pos);
  }
}

/*
 * This method returns the number of leaves in the object links tree. The
 * Size Map should be cleared at each iteration of the program, this is so
 * that new sizes can be updated when they change.
 */
public int getSize(IDebugObject ido) throws NullLinkException {
  // If size already calculated, thus in size mapping, return entry.
  if (!sizeMap.isEmpty()) {
    if (sizeMap.containsKey(ido)) {
      return sizeMap.get(ido);
    } else {
      throw new RuntimeException(
          "SYSTEM CALLED FOR GETSIZE ON AN UNKNOWN ELEMENT.");
    }
  }
  // If size mapping yet to be created, and given IDebugObject has some
  // forward links, perform single pass through objects, calculating
  // sizes as we go.
  else if (ido.objectLinks().size() > 0) {

    // Create list of IDebugObject and their ancestors in the tree.
    LinkedList<idoAncestorsListPair> list =
      new LinkedList<idoAncestorsListPair>();
    // Add root node to the seenList.
    seenList.add(ido);

    // Iterate through the forward links of this IDebugObject, creating
    // IDebugObject, ancestor pairings as we go.
    for (Entry<IDebugObject, IVariable> variableLink : ido
        .objectLinks().entrySet()) {
      if (!seenList.contains(variableLink.getKey())) {
        idoAncestorsListPair idoAncestorsPair = new idoAncestorsListPair(
            variableLink.getKey());
        // root is parent, so add to ancestor list.
        idoAncestorsPair.addAncestor(ido);
        // Add to overall BFS search list.
        list.add(idoAncestorsPair);
        // Add to list of seen nodes, maintaining BFS search pattern.
        seenList.add(variableLink.getKey());
      }
    }

    // Calculate how many elements are below root.

    // Iterate through list of children.
    while (!list.isEmpty()) {
      // Create clone of our list, to allow it to be destroyed.
      LinkedList<idoAncestorsListPair> tempList =
        (LinkedList<idoAncestorsListPair>) list.clone();
      // clear current list.
      list.clear();
      // Iterate through each child node in the original list.
      for (idoAncestorsListPair i : tempList) {
        int childrenCount = 0;
        // Iterate through that child's forward links.
        for (IDebugObject newi : i.getIDO().objectLinks().keySet()) {
          // List added with new objects
          // While loop continues until all objects
          // iterated through.
```

```
                if (!seenList.contains(newi)) {
                    // increment number of children counter.
                    childrenCount++;
                    idoAncestorsListPair idoAncestorsPair = new idoAncestorsListPair(
                            newi);
                    // Add all current ancestors.
                    idoAncestorsPair.addAncestors(i.getAncestors());
                    // Add current parent.
                    idoAncestorsPair.addAncestor(i.getIDO());
                    // Add this node to the "to-be iterated" list.
                    list.add(idoAncestorsPair);
                    // Add to list of seen nodes. (Dealing with
                    // backlinks.)
                    seenList.add(newi);
                }
            }

            /*
             * If node has no children, we know it's a leaf! Crucially,
             * we can now look at all its ancestors, and increase their
             * size. As we do this for all leaves, we know each node in
             * the tree will have a size depending on the number of LEAF
             * nodes in its sub-tree.
             */
            if (childrenCount == 0) {
                // New node, so put straight into map.
                sizeMap.put(i.getIDO(), 1);
                // We then increment the size of EVERY ancestor.
                for (IDebugObject ancestor : i.getAncestors()) {
                    if (sizeMap.containsKey(ancestor)) {
                        int curSize = sizeMap.get(ancestor);
                        sizeMap.put(ancestor, curSize + 1);
                    } else {
                        sizeMap.put(ancestor, 1);
                    }
                }
            }
        }
    }
    // Return size of original IDebugObject.
    return sizeMap.get(ido);
        } else {
            // If original IDebugObject is a leaf, size is simply 1.
            sizeMap.put(ido, 1);
            return 1;
        }
    }

    /*
     * (non-Javadoc)
     * @see view.interfaces.LayoutManager3D#updateAllPositions()
     */
    public void updateAllPositions() {
        // Tree recreated at each step, this is not a general view layout
        // manager, so we don't need to implement this method.
        // This is a special case for Layout Managers.
    }

}
```

```java
/**
 * The idoAncestorListPair class:
 * This class allows for the size of a tree to be
 * calculated efficiently. It provides a way of storing each node, alongside all
 * of its ancestors.
 *
 * @author Darius Bradbury.
 *
 */
public class idoAncestorsListPair {
  // The IDebugObject node.
  IDebugObject ido;
  // The IDebugObject's ancestors in the tree.
  LinkedList<IDebugObject> ancestors;

  /**
   * Instantiates the object, setting the node to the given IDebugObject.
   *
   * @param ido -
   *        The node we want to maintain a list of ancestors for.
   */
  public idoAncestorsListPair(IDebugObject ido) {
    this.ido = ido;
    ancestors = new LinkedList<IDebugObject>();
  }

  /**
   * Add an ancestor to the list.
   *
   * @param o3d -
   *        One of the nodes ancestors.
   */
  public void addAncestor(IDebugObject o3d) {
    ancestors.add(o3d);
  }

  /**
   * Add a list of ancestors to the list.
   *
   * @param ancestorList -
   *        list of ancestors to be added.
   */
  public void addAncestors(LinkedList<IDebugObject> ancestorList) {
    ancestors.addAll(ancestorList);
  }

  /**
   * Enable IDebugObject to be retrieved.
   *
   * @return the IDebugObject node.
   */
  public IDebugObject getIDO() {
    return ido;
  }

  /**
   * Returns a list of all the ancestors of this IDebugObject.
   *
   * @return List of ancestors.
   */
  public LinkedList<IDebugObject> getAncestors() {
    return ancestors;
  }
}
```

```java
/**
 * The activator class controls the plug-in life cycle
 */
public class Activator extends AbstractUIPlugin {

  // The plug-in ID
  public static final String PLUGIN_ID = "View";

  // The shared instance
  private static Activator plugin;

  /**
   * The constructor
   */
  public Activator() {
    plugin = this;
  }

  /*
   * (non-Javadoc)
   * @see
org.eclipse.ui.plugin.AbstractUIPlugin#start(org.osgi.framework.BundleContext)
   */
  public void start(BundleContext context) throws Exception {
    super.start(context);
  }

  /*
   * (non-Javadoc)
   * @see
org.eclipse.ui.plugin.AbstractUIPlugin#stop(org.osgi.framework.BundleContext)
   */
  public void stop(BundleContext context) throws Exception {
    plugin = null;
    super.stop(context);
  }

  /**
   * Returns the shared instance
   *
   * @return the shared instance
   */
  public static Activator getDefault() {
    return plugin;
  }

  /**
   * Returns an image descriptor for the image file at the given
   * plug-in relative path
   *
   * @param path the path
   * @return the image descriptor
   */
  public static ImageDescriptor getImageDescriptor(String path) {
    return imageDescriptorFromPlugin(PLUGIN_ID, path);
  }
}
```