

# VISUALIZING GRAPHS USING JAVA3D

Erik Hedenstrom  
December 6,2000

## **Abstract**

In this paper the design and implementation of an application to view graphs using Java 3D is provided. The application is designed as using three phases: information gathering, layout, and presentation.

The paper begins by introducing the reader to the three phases. After that it discusses how to implement a Java 3D application that displays a graph. First a section explaining the implementation of information gathering and layout is presented including a few examples of Java code.

Finally a section on the implementation of a Java 3D graph display is presented. The section begins by covering Java 3D basics. Next the techniques for setting up the 3D display are discussed. The section ends with an explanation of how to populate the 3D display with components of a graph, essentially drawing the vertices and edges in 3D.

Only a minimal amount of code is presented in this paper. A complete implementation of an advanced 3D graph viewer is provided as a complement for the reader to study once the basics presented in this paper have been understood.

**Keywords:** graph layout, Java 3D

# Table of Contents

Abstract .....	1
Introduction .....	3
Phases of Graph Visualization .....	4
Information gathering.....	4
Layout.....	4
Presentation .....	4
A Java 3D Application for Graph Presentation.....	6
Fig. 1 Simple application to perform information gathering and layout.....	7
Fig. 2 Main section of application.....	8
Implementation of Display3D.....	8
Java 3D Basics .....	8
Fig. 3 Conceptual Java 3D Renderer process.....	9
Fig. 4 Simple recipe for Java 3D programs.....	10
The main section of Display3D.....	10
Creating a scene from a graph.....	11
Fig. 5 Steps to create a scene from a graph.....	12
Fig. 6 Steps to cylinder between arbitrary points.....	13
Conclusion.....	14
Bibliography.....	15
Appendix A – Example of animation during layout .....	16
Appendix B – Sample graph data file .....	17

## Introduction

Programmers often need to design applications to visualize or present relational data. One way of dealing with the task is to treat the relational data as a graph. For example traffic on a website can easily be represented by a directed graph where the vertices represent web-pages, the edges represent transitions between pages (clicks), and edge weights represent the volume of transitions between two pages. Now the task has become the visualization of a graph.

The visualization of a graph can often be broken down into three phases, information gathering, layout, and presentation. This paper will attempt to provide an explanation of the phases and show the reader how it is possible to implement a graph viewing program using Java3D. To keep the page volume to a minimum the discussion of program implementation are maintained at a high level of abstraction. For low-level details regarding Java 3D programming the Sun Java site (<http://java.sun.com>) can provide all the information needed. Before we can begin looking at the program implementation we need to discuss the phases involved in graph visualization.

# Phases of Graph Visualization

## Information gathering

During this phase information is collected and distilled into a graph. The graph ( $G$ ) is simply a set of vertices ( $V$ ) and edges ( $E$ ). The information may be collected from a static or dynamic source. The output from this phase is a graph without any layout information.

## Layout

The layout phase receives a graph as an input. It applies some form of layout algorithm to the graph. When discussing the program implementation no specific layout algorithm is mentioned. However, it is important to note that the application is designed for an iterative layout algorithm. This means that the layout calculations may need to be performed many times before the layout is considered done [3]. The program implemented later in this paper is not dependent upon the layout algorithm. This is a design consideration based upon the fact that it may be desirable to switch layout algorithms depending upon how the data is to be visualized. When the layout phase is complete the graph has layout information attached to each vertex.

## Presentation

The final stage is the presentation stage. It receives a graph, in which each vertex has a 3-dimensional position, as its input. The exact details of the presentation stage are usually dependent upon many external factors. However, there are two things that most implementations have in common:

1. The position of each vertex in the graph must be calculated before it can be drawn. The position is calculated using three steps. The barycenter of the graph (average of all vertex positions) is subtracted from the vertex position, resulting in a vector relative to the origin. The next step is to scale the vector by dividing it by the length of the longest vector (the distance between the barycenter and the vertex that is farthest away from it). Finally the vector is scaled according to the size of the view area, resulting in a point that can easily be drawn within the view.
2. The position of each edge is calculated in the same manner as the position of a vertex. However, for each edge, two positions are calculated.

The calculation of position is usually the same regardless of whether a 2D or 3D drawing of the graph is being generated. In the 2D case, the Z-coordinate of each point can be ignored resulting in a parallel projection of the graph upon the X-Y plane.

## A Java 3D Application for Graph Presentation

The basic concepts of layout and presentation were discussed in the previous section. In this section, the design and implementation of a Java 3D application for graph presentation will be discussed. The application will be separated into three modules: information gathering, layout, and 3D display. Implementation of information gathering and layout will be briefly presented, while the main focus will be upon the 3D display.

Data files will be used as the source of graph data. Each file contains values describing the number of vertices and the number of edges. Each vertex name, shape, and color is then listed on a separate line. The edges are listed after all the vertices have been listed. Each edge is listed by specifying the name of the start vertex followed by the name of the end vertex. For a sample data file see appendix B. A utility class called `GraphFileParser` is used in the application in the following manner:

```
Graph graph = GraphFileParser.parse(filename);
```

The resulting graph object contains all the vertices and edges. Each vertex has an initial position of (0,0,0). The next step in the application is to apply the layout algorithm.

The layout algorithm has three methods that we are concerned with: *initialize*, *done*, and *update*. *Initialize* takes a graph object as the argument and performs any necessary initialization. *Done* returns a Boolean value representing whether the graph has reached a final layout or not. *Update* performs layout computation on each vertex in the graph. The graph application can now be written as (comments are within `/* */`):

```

/* Read the graph from a file */
Graph graph = GraphFileParser.parse(filename);

/* Create a new layout algorithm object */
Layout layout = new Layout();

/*Initialize the layout algorithm */
layout.initialize(graph);

while (!layout.done()) {
    /* Update the layout of the graph, until it is done */
    layout.update();
}

```

**Fig. 1 Simple application to perform information gathering and layout**

The code above will result in a graph with a final layout by reading a graph from a file, initializing the graph and the layout algorithm, and applying the layout until it is done. This is a non-interactive application. In the final version two display objects are added, Display2D, and Display3D.

The Display2D uses simple Java graphics to draw parallel projection of the graph object onto the XY plane. The Display2D object has two methods that are used in our application, *initialize*, and *update*. *Initialize* takes a graph as an argument and performs simple initialization. The *update* method performs the drawing of a parallel projection upon the XY plane. By calling the *update* method inside the while loop the layout phase becomes animated. Animation has a positive effect on the user since it becomes visibly apparent that the application is working. The layout phase (while loop) can take several seconds to complete for large graphs. If no output was provided the user might become impatient with the program. Animation is a simple way of keeping the users interest. Appendix A contains some images extracted from a Display2D object during animation.

The Display3D uses Java 3D graphics to create a visual representation of the graph.

Before we discuss the Display3D in depth presenting the entire code for the main application concludes this section.

```
Graph graph = GraphFileParser.parse(filename);
Display2D display2D = new Display2D();
Layout layout = new Layout();

layout.initialize(graph);
display2D.initialize(graph);
while (!layout.done()) { /* keep looping if NOT done */
    layout.update();
    display2D.update(); /* Perform animated display */
}
Display3D display3D = new Display3D();
display3D.initialize(graph);
display3D.update();
```

**Fig. 2 Main section of application**

## **Implementation of Display3D**

Before we can discuss the specifics of Display3D a basic understanding of Java 3D is required. The next section attempts to provide a condensed explanation of Java 3D basics.

### **Java 3D Basics**

“The java 3D API is a hierarchy of classes that serve as the interface to a sophisticated three-dimensional graphics and sound rendering system”. [4] At the core of this hierarchy is the scene graph. It is important to note that the scene graph is an acyclic directed graph, in which every node only has one parent [5]. It is essentially a tree representation of the

scene. A scene is a collection of objects that represent geometry (shapes), sound, lights, location, orientation, and appearance.

To place a shape in a scene it needs to be attached to the scene graph. An object is moved within the scene by attaching it to a *TransformGroup*, and then attaching the *TransformGroup* to the scene. The *TransformGroup* which an object is attached to can be static or dynamic. An example of a static transform would be a translation to a specific point. An example of a dynamic transform would be a rotation that changes value based upon the movements of a pointing device, or a rotation that changes with time. Object transforms are combined by attaching a *TransformGroup* to another *TransformGroup*.

When the construction of a scene graph is complete it is usually compiled. This step attempts to perform various optimizations of the scene graph such as combining static transforms or optimal path calculations.

Once the scene is created and compiled a simple Java 3D application will enter an infinite loop conceptually performing the following operations [4].

```
while(true) {  
    Process input  
    If (request to exit) break  
    Perform Behaviors  
    Traverse the scene graph and render visual objects  
}
```

**Fig. 3 Conceptual Java 3D Renderer process**

Java 3D provides many utility classes to simplify certain tasks. In the main application, several of these utility classes are used. Every simple Java 3D application involves 5 steps [4]; these steps are used in implementing Display3D in the next section.

1. Create a Canvas3D Object
2. Create a SimpleUniverse object which references the Canvas 3D object
  - a. Customize the SimpleUniverse object
3. Construct content branch
4. Compile content branch graph
5. Insert the content branch graph into the SimpleUniverse

**Fig. 4 Simple recipe for Java 3D programs.**

## **The main section of Display3D**

When the Display3D object is created in our application (see fig. 2) the steps for simple Java 3D applications are executed using the following code:

```
Canvas3D canvas = new Canvas3D(config);

SimpleUniverse universe = new SimpleUniverse(canvas);
universe.getViewingPlatform().setNominalViewingTransform();

BranchGroup scene = createSceneGraph(universe);
scene.compile();

universe.addBranchGraph(scene);
```

As mentioned earlier these steps are practically identical for all simple Java 3D applications. Most of the work is performed inside the *createSceneGraph* method. It is important to note is that the method *setNominalViewingTransform* places the eye approximately 2.41 meters along the Z-axis looking at the origin. At this distance, with

the default field of view, objects that are max 2 meters in height or width will fit on the viewing plate. The knowledge of these dimensions is crucial when scaling the graph of data (not the scene graph) in the *createSceneGraph* method.

## **Creating a scene from a graph**

The application is interactive by providing the ability to rotate rendered graphs, zoom in, zoom out, and move the eye up/down or left/right. The interactivity is implemented in the application by creating transforms for the various mouse behaviors. The graph is attached to a mouse rotation transform, mouse zoom and translate are attached to the viewing platform transform which controls the position of the eye.

Creating a scene graph from a graph, which has been passed through the layout algorithm, involves the following steps, which are essentially an abstraction of what takes place in the *createSceneGraph* method:

1. Create a scene graph (scene).
2. Initialize a TransformGroup for rotation and retrieve the TransformGroup for eye positioning.
3. Attach the rotation TransformGroup to scene.
4. Create and attach a Mouse Rotation to the rotation TransformGroup.
5. Create and attach a Mouse Translation to the eye TransformGroup.
6. Create and attach a Mouse Zoom to the eye TransformGroup.
7. For every vertex in the graph:
  - a. Convert the position of the vertex in graph space to view space.
  - b. Create a translation TransformGroup using the converted position.
  - c. Create a sphere with vertex color.
  - d. Attach the sphere to the translation TransformGroup.
  - e. Attach the translation TransformGroup to the rotation TransformGroup.
  - f. For every edge of the vertex, if the edge is not already in the scene:
    - i. Convert the end position of the edge.
    - ii. Create a cylinder between the start and end position.

- iii. Attach the cylinder to the rotation TransformGroup.
8. Create directional lighting, and attach it to scene.
9. Create fog, and attach it to scene.
10. Return the scene graph.

**Fig. 5 Steps to create a scene from a graph.**

Steps 8 and 9 are performed primarily to add visual cues that enhance depth perception. The directional lighting results in diffuse reflections from the spheres and cylinders. The fog makes objects further away from the eye fade [5].

Conversions of positions from graph space are computed using simple vector arithmetic. Subtracting the barycenter of the graph from the vertex position creates a vector. The vector is then divided by the largest distance between any vertex and the barycenter. The scaled vector is then multiplied by a constant that governs how much of the view space is to be used in displaying graphs. As mentioned earlier, in the *SimpleUniverse* using the nominal viewing transform objects of height and width equal 2 meters fit exactly into the display. By using a constant of 0.8 meters the graphs will have a maximum dimension of 1.6 meters making them fit neatly into the viewing window.

The most complex task, creation of a cylinder, in the previous steps was abstracted by step 7.f.ii. Java 3D provides a cylinder class that creates cylinders of a given radius and height, with the Y-axis passing through the center. The middle of the cylinder coincides with the origin. A utility class was implemented so that cylinders could be created by providing the start position, end position, and radius. The utility has the following steps:

1. Calculate the center of the cylinder by averaging the start and end position.
2. Calculate the height and unit vector along the cylinder axis.
3. Calculate a matrix for the rotation of the unit vector onto the Y axis.
4. Invert the rotation matrix.
5. Create a rotation TransformGroup using the rotation matrix.
6. Create a translation TransformGroup using the center.
7. Create a Java 3D Cylinder (oriented along Y-axis).
8. Attach the cylinder to the rotation TransformGroup.
9. Attach the rotation TransformGroup to the translation TransformGroup.
10. Return the translation TransformGroup.

**Fig. 6 Steps to cylinder between arbitrary points.**

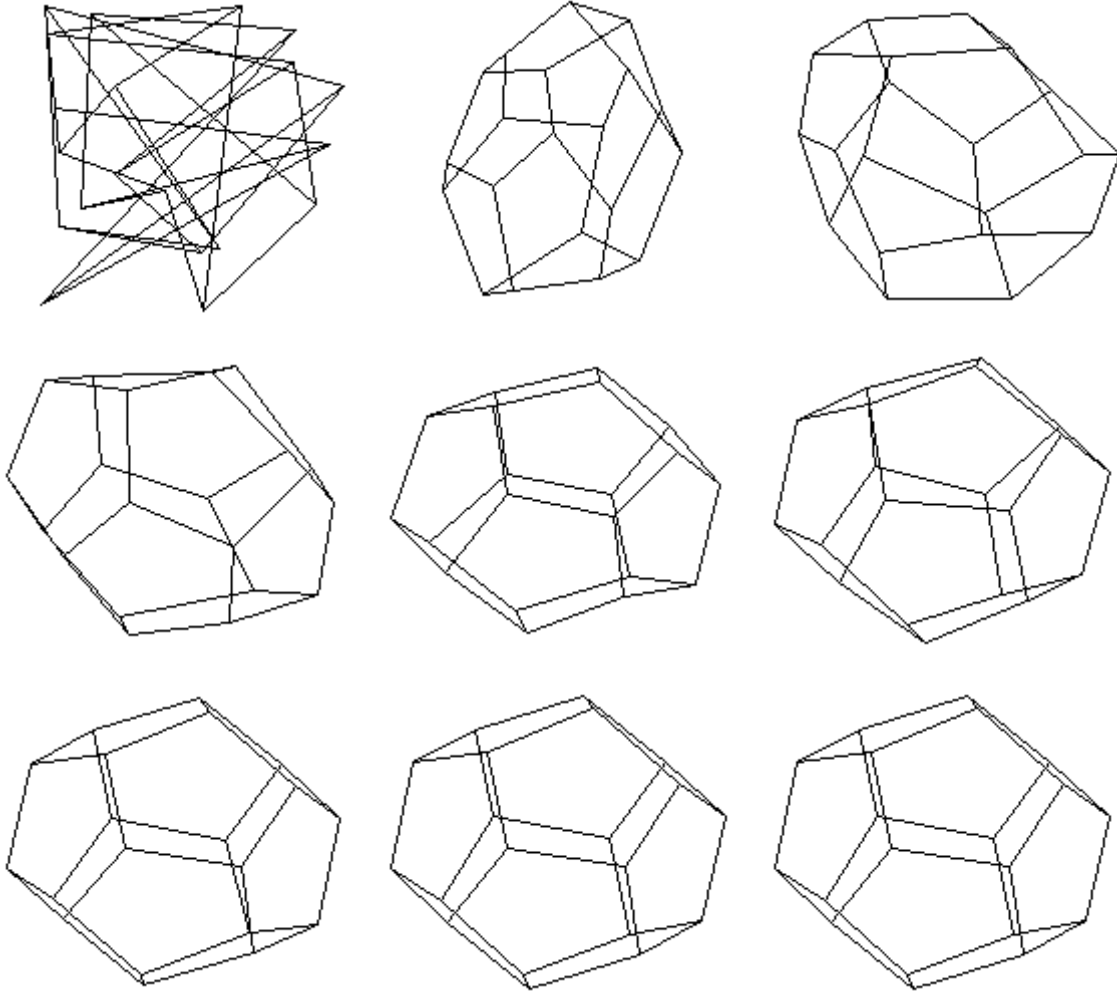
## **Conclusion**

By using a simple graph layout algorithm, in combination with Java 3D, graph visualization in 3 dimensions can easily be implemented. The primary concern when visualizing graphs is the interactivity of the application. By providing an animated display of the layout process, which can take several seconds, the users attention is maintained. Once layout is completed Java 3D is used to create final view of the graph. The final view provides user interaction by responding to mouse clicks and movements.

## Bibliography

- [1] Frick, A., Ludwig, A., Mehldau, H.: *A Fast Adaptive Layout Algorithm for Undirected Graphs. Proc. Workshop on Graph Drawing 94*. LNCS 894 (1994) 389-403
- [2] Frick, A., Bruss, I.: *Fast Interactive 3-D Graph Visualization. Lecture Notes in Computer Science (Proc. GD '95)*, 1027 (Springer Verlag): 99-110, 1996.
- [3] Battista, G. D., Eades, P., Tamassia, R., Tollis, I. G.: *Graph Drawing: Algorithms for the Visualization of Graphs*, Prentice Hall: 303-325, 1999.
- [4] Bouvier, J. D.: *Getting Started with the Java 3D API v. 1.1.2*, Sun Microsystems Inc.: 1.1 – 2.44, 1999.
- [5] *The Java 3D API Specification v. 1.2*, Sun Microsystems Inc.: 1–601 2000.

**Appendix A – Example of animation during layout  
using the GEM 3D algorithm [2]**



## Appendix B – Sample graph data file

# A simple cube

8 12 n

# Vertices

1 quader red

2 quader red

3 quader red

4 quader red

5 quader red

6 quader red

7 quader red

8 quader red

# Edges

1 2

1 4

1 5

2 3

2 6

3 4

3 7

4 8

5 6

5 8

6 7

7 8