

# Guided Navigation in Task-Oriented 3D Graph Visualizations

Guido Reina<sup>†</sup> Sven Lange-Last<sup>‡</sup> Klaus Engel<sup>†</sup> Thomas Ertl<sup>†</sup>

<sup>‡</sup>IBM Research Germany  
eServer Systems Management Technology

<sup>†</sup>Visualization and Interactive Systems Group  
University of Stuttgart, Germany

## Abstract

*In many application areas an optimal visualization of a complex graph depends on the specific task to be accomplished by the user. Therefore a means of locally customizable layouts is required to focus on different problem aspects. We propose a concept of user-driven hierarchization and layout optimization in the context of IBM zSeries I/O configurations. We combine the concepts of glyphs, local 3D layout managers and partitioning galaxies to create an optimal layout for a given task. Additionally, to simplify global navigation, structural details are abstracted and can be refined upon user request.*

*Since element context is crucial for understanding and consistently configuring an I/O configuration, guided navigation along the connections in this data structure is essential. To efficiently customize a graph layout, complex selection mechanisms are needed to quickly define the areas affected by a particular layout. Although we focus on IBM zSeries I/O configurations, the approach we present is quite generic and can be adopted for other fields of application.*

## 1. Introduction

Graphs are typical for many application areas, e.g. hypertext browsing, state-transition diagrams, computer networks, file-system structures etc. Often graphs are quite complex, i.e. the number of nodes and edges can grow very large. This makes visualization and navigation of such structures an extremely difficult task. Even though there already exist a number of highly optimized layout algorithms, the design of the optimal structure of 3D graphs may require additional knowledge that is often only available to the domain experts as well as task-specific information. For that reason, a hybrid, flexible mechanism for the layout of 3D graphs is required that provides domain experts with both powerful automatic layout algorithms and manual tools for creating optimal 3D visualizations.

This paper covers 3D visualization of IBM eServer zSeries I/O topologies and the associated navigation con-

cepts. Since we assume that the readers are not familiar with these I/O topologies we will give a short introduction into this topic and will outline the properties of I/O topology from an information visualization point of view. After that we will list requirements for the visualization.

These requirements lead to our approach for the visualization and navigation of complex I/O topologies, which is presented in section 5. Afterwards, the strength of the approach is demonstrated by evaluating the I/O topology visualization and navigation of a real-world enterprise configuration. Finally, section 6 sums up the paper.

## 2. IBM eServer zSeries

The IBM eServer zSeries is a multiprocessor mainframe class computer line. Such a machine can be logically partitioned (LPAR mode) and can as such run up to 15 totally separated operating systems at a time.

A particular strength of the IBM eServer zSeries is I/O: The IBM eServer zSeries architecture allows for 256 I/O channels connecting to I/O hardware like disks, tapes, and network. To avoid a single point of failure on the path from a zSeries mainframe to a device usually multiple paths are defined.

In the following paragraphs some rough zSeries properties are defined. The description is very terse and abstracts all concepts which do not directly influence the current visualization, probably sacrificing some coherence.

### 2.1. zSeries I/O Configuration

To relieve user programs and the operating system of performing I/O, a component called Channel Subsystem (CSS) is responsible for carrying out all I/O operations. In order to accomplish this task the CSS needs detailed information about the attached I/O topology (also known as *logical I/O configuration*) and its access rights, which determine connection permissions. The logical I/O configuration has to be specified manually by the system programmer.

The logical I/O configuration consists of the following elements:

- **PROCESSOR:** A processor corresponds to exactly one IBM eServer zSeries mainframe.
- **LPAR:** An LPAR is a logical partition running on one processor. Up to 15 LPARs are supported by one processor.
- **CHPID:** A channel path ID (CHPID) corresponds to an I/O channel. Up to 256 are supported by one processor.
- **CONTROL UNIT:** A control unit (CU) connects channels (CHPIDs) to devices. It provides capabilities like protocol translation, caching, and RAID.
- **DEVICE:** A device is the unit performing I/O and is connected to a CU. A device can be a disk drive, a tape recorder, a network interface, or even a simulated disk drive.
- **SWITCH:** A switch connects a CHPID to a CU or to another switch. A switch dynamically connects one switch port with another on the same switch. In this way, a CHPID can be connected to more than one CU directly. A port-to-port connection inside a switch can also be determined statically.

In the ESCON architecture, at most two switches can be located on the path from a CHPID to a CU. If two switches are located on the path from a CHPID to a CU, at most one of two switches may establish the path dynamically. Keeping in mind that, using switches, up to 4,048 CUs and 1,036,288 devices can be connected to a single CHPID, the complexity of an I/O configuration becomes clear.

In addition to the logical I/O configuration (which is needed by the mainframe to perform I/O), the zSeries customer needs information about the *physical* I/O configuration.

The physical I/O configuration reflects which items really are located in the data center, e.g. cables connecting channel card jacks (CHPIDS) and switch ports, hundreds of cables used to connect these ends with short patch cables. Nowadays, devices and control units are not physically separated or even don't exist any more – although the Channel Subsystem (CSS) still uses this model. The physical I/O configuration describes these 'storage servers' and how they are mapped to the logical I/O configuration.

## 2.2. Properties of Logical and Physical I/O Configuration

Roughly, logical I/O configurations form a hierarchy if the access lists are not observed: At the top are processors which contain the CHPIDs. The CHPIDs connect to switches or directly to CUs which, in turn, connect to devices. Only switches may connect to elements – i.e.

switches – on the same hierarchy level (Note: With ESCON, at most two switches may be chained). This implicit order in a configuration enables the user to differentiate between upstream and downstream connections and make some assumptions for a creating a better layout (e.g. the user knows that below control units there can only be devices).

Things look different if access lists are considered. With access lists, LPARs have a direct connection to CHPIDs and devices.

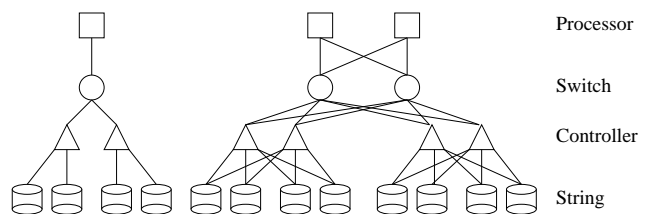
Logical I/O configurations of a whole data center contain more than one processor. In these configurations, a control unit can be connected to several CHPIDs belonging to different processors.

All this means that logical I/O configurations are not tree structures but directed graphs. They are directed because there is always a path from the processor down to the devices. For this reason, most parts of the logical I/O configuration contains no cycles. Only two switches may be part of a cycle: Consider two CHPIDs connecting to two different switches and then, connect the output port of each switch to the other.

In physical I/O configurations, the situation is more difficult. Patch cabinets are used to connect cables and therefore can be situated anywhere in the hierarchy. Any patch cabinet can be located at multiple levels of the hierarchy, introducing cycles.

## 2.3. Typical I/O configuration

A typical I/O configuration (either logical or physical) supports redundancy. If a functional unit needed to perform I/O is only available once on the path from the processor to the device, the whole I/O fails if this functional unit fails. Such a *single point of failure* should be avoided in production configurations. This means duplicate processors, CHPIDS, switches, CUs, and devices to have data always available. This means that the basic graph structure resulting from such a configuration – which could ideally be a tree (see figure 1) – contains many parallel links as well as crosslinks to avoid single points of failure.



**Figure 1. general configuration structure (left) compared to a schematic one (right)**

### 3. Requirements

The person who deals with a zSeries I/O configuration (called system programmer) must have a tool to visualize the I/O topology at his or her hands. This topology is either the logical or the physical I/O configuration. The visualization and the associated navigation has to be appropriate to accomplish typical tasks:

- View the logical and physical I/O configuration to determine where new I/O hardware can be integrated into the existing topology.
- Add new objects to the I/O configuration and connect them to existing objects.
- Modify or remove existing objects.
- Locate specific objects in the I/O configuration to perform problem determination. Find out which other objects the object in question is connected to and how.

Depending on the task to be performed, different aspects of the I/O topology are of interest for the system programmer. For some tasks, it might be helpful to see which devices are reachable from a specific processor. For other tasks, it might be helpful to see which processors can reach a specific processor. (In general, reachability information is very helpful for the system programmers.) For this reason, the appropriate visualization depends on the task to be accomplished by the system programmer. There won't be a single generic layout which satisfies all needs.

### 4. Related Work

Many highly optimized algorithms for automatic graph layout have been developed in the past years. The classical cone tree[10] approach has been enhanced into Reconfigurable Disc Trees[2], reducing occlusion and enhancing flexibility. Another solution uses hemispheres instead of discs for denser element distribution and lays them out in hyperbolic space [8]. Since distances in hyperbolic space are non-linear, elements which are farther away from the camera are subpixel-sized. Such an approach is therefore difficult to use for a general overview of a whole graph. A 3D spring modeling algorithm, as presented in [4], could be used for a purely connection-based layout.

For our problem it would be best, however, if the user could locally change layout strategies to better suit his needs for that context. A good start has been presented in [5]. This article introduces a hierarchically nested graph structure which allows layout customization on each level of the hierarchy. For zSeries I/O configurations it does not make sense, however, to enforce a hierarchy (e.g. by use of a spanning tree), since we do not want to obfuscate the fact that some graph elements have multiple parents (that information is vital for redundancy comprehension) and we do not

want to imply an order among elements of the same type. In the same way we do not want a thoroughly hierarchy-defeating layout like a spring embedder, for example, since the user may want to retain a hierarchical display for some regions of the graph, like controllers and their connected devices. Therefore we need an approach which is flexible for every element to maximize layout optimization possibilities for the user.

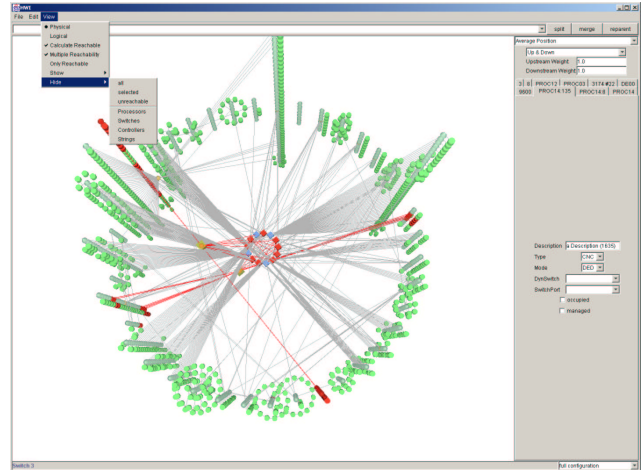


Figure 2. Prototype GUI

### 5. Visualization and Navigation of I/O Topologies

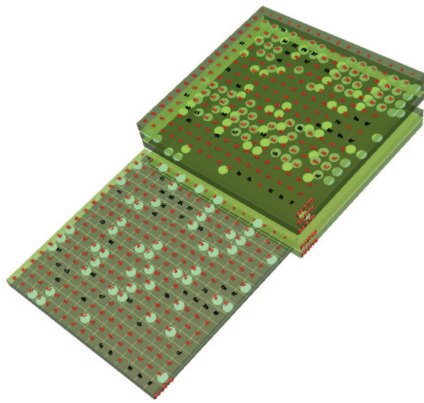
The prototype implemented to demonstrate the concepts presented in this paper has a simple GUI (see figure 2). The main part is occupied by a 3D view of the zSeries I/O configuration, on top there is a toolbar for rough configuration partitioning, and on the right side there is a panel where the user can change the layout and edit the properties of any selected configuration element on the fly (called *attribute pane*). The visualization and interaction with the prototype will be presented in the following sections.

A 3D View of the I/O configuration graph has been chosen in order to improve the user's grasp of connectivity. [11] demonstrates that for arbitrary graphs, and as such for our overlapping trees, a 3D view with manipulable view point is preferable over a 2D layout. A simplified approach is used employing a mouse instead of tracking mechanisms, since we wanted the program to run on any common PC without special peripherals.

#### 5.1. Element Grouping

A severe visualization problem is represented by the connectivity of zSeries configuration graphs. Redundancy and bandwidth necessities make for multiple connections between different ports of the same physical configuration el-

ements. To avoid visual clutter, we address this problem by displaying only abstract connections which represent an arbitrary number of connections between two physical elements. This could be referred to as an extremely simple structural clustering [7]. This way, 100.000 logical connections in an existing configuration have been reduced to less than 700. The user can still access the actual connections by displaying an abstract connection's detail view (see below). A similar reduction is possible for the configuration elements: if only physical elements are displayed, the graph of the aforementioned configuration consists of 27.000 elements. By only displaying physically discernible elements like Processors and Controllers, we reduce the number of elements to about 500.



**Figure 3. Processor detail view with selected partition**

## 5.2. Dynamic Details

Since the elements of an I/O configuration have an internal structure and wrap a number of subelements, a mechanism for displaying them is needed. A possible solution would be a distortion viewing technique as in [1, 3]. This approach is most advantageous if the magnified data fits in a relatively small area of the screen as to allow displaying of the context of the magnified region. In our case, however, the detail information is so complex that it may even need more than a whole screen to be presented. In our approach the user can have the view zoom in on the elements he wants to see the structure of while that element is replaced by a 3D detail view of itself and its subelements. Because of performance issues, the context of that element is hidden for now. For processors this approach is very compact and intuitive to use: The processor is displayed as a stack of transparent partitions, so the user can grasp the overall partition-CHPID access permissions by using a top view. If the user wants to view and edit a particular partition, he simply selects it (see figure 3). The prototype then pulls out that partition like a drawer and places a grid on it to help the user with CHPID

selection. Configuration is performed via the attribute pane and the context menu. The desktop metaphor behind this view makes the concept easy to grasp even for less experienced users.

This solution works since we know that the user wants to see the whole internal structure, so we can just zoom that area, even expending the whole display area, until the items can be conveniently inspected and edited and thus saving the user any manual zoom operation. This mechanism is also well-suited to collapse parts of the graph into single elements and to dive into those collapsed graphs.

## 5.3. Display Modes

To enable the user to view the physical configuration with its cables as well as the access rights of the logical configuration, the display can be toggled between these two modes. Since the logical connections tend to form the transitive hull of the physical connections, the display clarity is heavily compromised. To address this issue, we give the user the possibility to only display the connections reachable from the current selection.

## 5.4. Element Visualization

To visualize whole ESCON configurations, different metaphors are used for each level of the data structure. The elements are shown as glyphs [9], with the element type encoded by geometry and color. Abstract geometries were chosen to maximize differentiation between the few element types. Processors are shown as blue boxes, switches as yellow spheres, controllers as gray cones and strings as green cylinders. Upon user request it is possible to color in red all elements which are transitively connected to the currently selected element (visible in figure 2). By this means elements can be related to their context: the user can assess the repercussions of configuration changes to the currently selected element so he knows which elements he needs to update next.

For this kind of reachability highlighting the geometry coding maintains element type discernibility. When the reachability display is deactivated, the color coding ensures that the types of elements observed at great distances are still distinguishable, so this is useful for configuration overviews. User control on elements visualization is currently limited to visibility modification, i.e. the user can hide the current selection of elements.

On the element group level we use layout managers. This concept has been adopted from 2D GUI toolkits, like Java's Swing or gtk/Qt. Unlike [5], which proposes independent layout algorithms for the different hierarchy levels of a graph, we do not premise a hierarchy in the graph, but allow the user to change the layout strategy for every single element of the I/O configuration. In our case a layout man-

ager is an object which distributes its managed elements (called *children* from now on) in an arbitrary way (depending on which layout algorithm we want to use). Each layout manager can provide a panel which will be displayed for the user to edit parameters for layout fine-tuning. Layout managers can only be instantiated by a `getInstance()` method. That way the programmer can ensure there are no two layout managers who do the same work if one would suffice.

The next level in the data structure is formed by elements and their layout managers grouped together as galaxies. Such galaxies provide support for partitioning a graph at a high level, so the user can divide a configuration by locations, for example, and then locally assign layout managers to the contained elements. Since galaxies have a defined center, which can be arbitrarily positioned by the user, the galaxy-local layout managers can distribute their children in relation to that center.

The highest level of metaphors consists of user-defined views. A view stores visibility, layout and galaxy partitioning information for all I/O configuration elements. This concept enables the user to focus on different aspects of an I/O configuration by defining views that highlight the particular information he needs for the task at hand.

## 5.5. General Layout Algorithm

The layout mechanism of our prototype works by flagging the validity of the layout by element. When the user changes a layout parameter, the layout is invalidated for each element. Then all the layout managers present in a configuration are gathered in a queue. This queue is then iterated, removing every layout manager if it reports a successful layout, until it is empty.

The problem in this phase is the fact that many layout managers use a pivot element, which, in turn, is laid out by another layout manager. So a layout manager can only complete its work when the layout manager responsible for the pivot has successfully completed the pivot's layout. Partially successful layouts can be detected at element level, alleviating the problem somewhat. An order in which the inter-depending layout managers have to recalculate their children's positions is too costly to be calculated ad hoc, so all the layout managers with at least one invalidated child are just called on each iteration until all the prerequisites are met.

### Caveats

A carelessly programmed layout manager could introduce circular dependencies between layout managers as well as the user could. The former problem is addressed by limiting the maximum number of iterations the queue can undergo, the latter can be solved by checking the user's input when layout managers are assigned.

The current layout mechanism suffers from collisions, too. For now, layout managers have no possibility to find out if the space they want to distribute their children in is already occupied by another layout manager, so if the user manipulates the layout to make several elements overlap, the layout engine cannot correct this problem as of yet.

To investigate the proposed concepts, some simple layout managers have been implemented in the prototype. These layout managers can be divided into two groups: *active* layout managers and *passive* layout managers. The former distribute their children on a defined shape or surface relatively to one other element (which serves as pivot). The latter calculate the layout depending on the elements connected to each of their children.

### Active Layoutmanagers

- **LM3DAbsolute**  
This layout manager allows the user to input numeric coordinates or clone coordinates from another element. It is best used for fixating positions.
- **LM3DCircle**  
This layout manager distributes its children on a circle around a definable center, i.e. any displayable element of the configuration (*pivot element*), not necessarily the end point of all upstream connections of the children. The user can also set the axis around which the circle is wrapped. This layout works best for small numbers of elements.

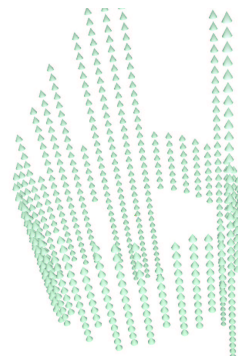


Figure 4. LM3DRods clusters

- **LM3DRods**  
This layout manager makes use of clustering to divide its children into groups based on their upstream connections. Each of these clusters forms a rod by stacking the contained elements. These rods are distributed on a circle of user-definable radius around a pivot element. The user can customize the axis around which that circle is wrapped as well as the maximum number of clusters that are created and the minimum number of elements that must be contained inside a clus-

ter. The strength of this layout is the bundling of upstream connections, which makes a global view more concise. Another advantage is the structure of the clusters in combination with 3D viewing: the user can observe the elements along the axis around which they are distributed to figure out the overall cluster connectivity before tilting the view to study the properties of single elements.

- LM3DHemisphere

This layout manager is based on the algorithm described in [8]. It clusters its children in the same way as LM3DRods, but lays out each cluster on the surface of a hemisphere. This layout is very compact, but does not give the incoming and outgoing connections a particular structure.

### Passive Layoutmanagers

- LM3DAveragePosition

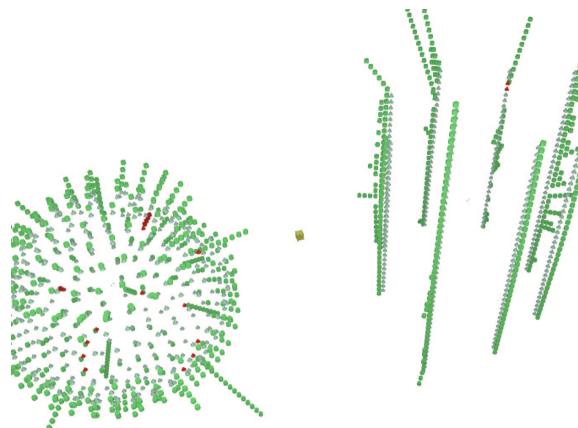
This layout manager positions its children by averaging the positions of connected elements. The user can adjust if the algorithm considers upstream or downstream connections (or both). Layout of an element is defined as successful if a user-definable fraction of the connected elements have a valid position (defaults to 0.7). This step is taken to avoid problems when elements using this layout manager are interconnected. This layout manager can be used only for a very small number of elements since the probability of success decreases if many elements depend on a completed layout of others. However, this layout manager is primarily intended for Switch layout which are the most unfrequent elements in an I/O configuration.

- LM3DRays

This layout manager places its children on an extension of the layout radius of their upstream parents. This layout works best on strings, if the corresponding controllers are laid out around some center, and allows to clearly display elements with multiple parents. Figure 5 shows controllers with the corresponding strings laid out with LM3DRays. On the left side, the controllers are laid out by using LM3DHemisphere, while on the right LM3DRods is used.

The default layout provided by the prototype consists of a mixture of the above (see figure 2).

These layout managers are just provided as examples: further layout managers will be integrated, like a classical cone tree and a spring embedder. For simpler data structures these could yield results that need only be adjusted in very few regions. It would also be possible to integrate layout managers specialized for specific configurations or application domains, so that these could provide a suitable default layout.



**Figure 5. Strings positioned relative to Controllers with LM3DRays**

## 5.6. Navigation and Interaction

The 3D view of the configuration cannot be altered freely. To avoid that the user loses his bearings, the focus of the view is always constrained to an element of the configuration. The camera can only be orbited around and dollyed relative to this element. By this means the user cannot look into empty portions of 3D space. If the user wants to change focus, he can simply [Alt]-Click onto another element to make it the camera pivot. The transition from one pivot to another is animated so the user can reconstruct the display change and does not lose orientation (see [6]). The navigation is denoted as guided since the user must only specify destinations, and does not need to move the camera himself.

### Selection

Selection has been made as comfortable as possible. After examining some of the possibilities [12], we decided to implement several ways for the user to select a particular element: Simple clicking in the 3D view selects the nearest element under the cursor. Since this method can be tedious to the average user because of occlusion and perspective, an assisted "interactive selection" is possible by middle-clicking. This pops up a dialog which shows all elements which have been intersected by the selection ray sorted by type. We can make the user save time that way when elements are tightly packed or small. Another option is dragging a box around the wanted elements. More advanced methods are *selection by context*, where the user can select the up-/downstream-reachable elements from the context menu of an already selected element, or selection from the search dialog.

The search dialog lets the user choose the element type he wants to search for and then shows him a table containing

all of the corresponding elements alongside their alphanumeric attributes. The user can then specify a substring filter for each column/attribute in a second table located below until the selection is accurately specified and finally add selected table elements to the current selection (see 6). All selection operations can be performed while holding [Shift], toggling the elements instead of replacing the selection. Any selection can be further modified from the menu, where the user can filter any element type or restrict the selection to a single type of choice.

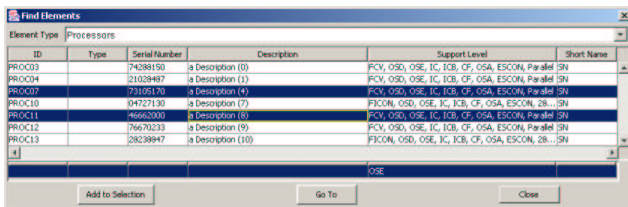


Figure 6. Search Dialog

### Element Manipulation

The GUI has been designed to allow fast and simple interaction. Taking advantage of detail views, a user can quickly access all subelements of a given physical element. Since the attribute pane is always available, the user can edit layout and element attributes without accessing further dialogs or menus. To reconstruct or backtrack changes, the element attribute panels are collected in a tabbed pane as a history of the last 10 edited elements. These tabs can also be used to reselect or set the focus on the respective elements.

### Context Navigation

The user can let the program guide him through the configuration. The context menu of any selection contains commands to change the view focus to a particular up/downstream reachable element. This focus transition is animated in the same way as the transition triggered by [Alt]-Clicking into the 3D view to ensure the user keeps track of the view changes.

If desired by the user, the coloring of reachable elements affects only the most recently selected element instead of the whole selection. The user can then cycle the reachability display through all selected elements using a hotkey (the focus is cycled accordingly).

### Galaxies Operations

The toolbar contains buttons which allow the user to partition the graph into large functional/geographical units, for example. The 'split' operation creates a new galaxy from the current selection and creates copies for any layout managers (those which are constrained to the old galaxy center (origin) are constrained to the new origin). These origins can then be positioned by the user by accessing their attribute

pane. Other galaxies operations are 'merge', which causes all galaxies in the current selection to be merged inside the galaxy of the first selected element, and 'reparent', which inserts the selection into the galaxy whom the first element of the selection belongs to.

### View Interaction

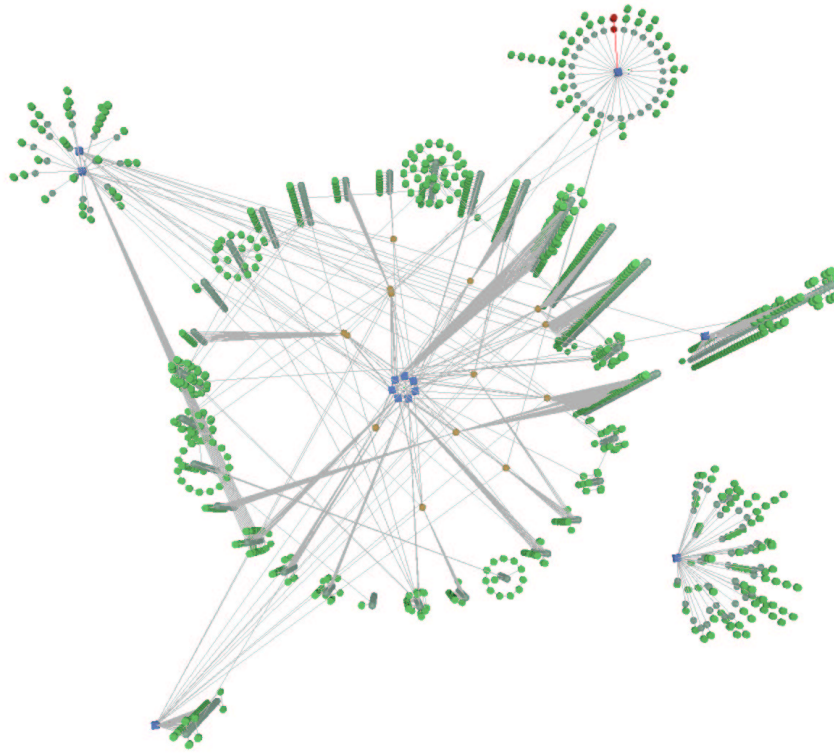
The user can simply name a view at some point. This will create a new view with the current layout/visibility/partitioning state. Any further layout modifications will be stored only in the new view, allowing the user to switch back and forth between different task-oriented views.

## 6. Results and Conclusions

The System we have implemented is programmed in Java3D making use of the Java3D Scenegraph API. We have been able to achieve interactive frame rates on a mid-range PC (800 Mhz CPU, 256MB RAM and GeForce2MX graphics). This demonstrates that Java3D can deliver a fast enough solution even with high polygon counts while offering a powerful scenegraph which saves the programmer work when transparency and interaction is required, for example.

Figure 7 shows an existing configuration after some rough partitioning and few layout changes. The benefits of abstract, banded connections can clearly be seen in the central galaxy. The position of switches is not constrained into the hierarchy since they would belong to more than one of the overlapping trees this configuration contains, so their position is determined by their context using LM3DAverage. The graph has been partitioned into six galaxies using reachability information; in fact inter-galaxy connections are relatively sparse.

The benefits of the approach presented in this paper can be summed up as follows: On the visualization side, the user has complete control over the layout of all elements of the I/O configuration, being therefore able to task-specifically highlight all the information he needs in such a graph. Since the customizable layout is user/domain expert-driven, any application area can benefit from the concepts after importing the data. Visual clutter has also been significantly reduced by only displaying abstract connections and physical elements of the I/O configuration. The architecture has also been developed to be easily extensible; any previous and future work could be implemented as a separate layout manager. Such a layout manager can be integrated into the prototype by writing 4 lines of code. The navigation concepts prevent that the user loses his bearings by attaching the focus to configuration elements. By animating any focus transitions, the user is supported in keeping track of his location relative to the whole configuration. Furthermore the



**Figure 7. Example configuration overview**

user is guided through a configuration following the existing connections between elements, allowing him to explore the context of a selected element.

To further evaluate the presented concepts, usability testing with domain experts (i.e. data center system programmers) should be performed.

This article focuses on the zSeries field of application to point out that a globally uniform layout is not sufficient to highlight any task-specific properties of a graph. The concepts presented could also be applied to web site structures, corporate networks or call graphs used for code reverse engineering. To enhance the graph structure, a domain expert should be able to customize the layout himself on a per-element basis, if necessary, to comply with the requirements of the current task.

Currently, the prototype is being ported to C++/OpenGL for performance improvement. Additionally, it is being fit with a second layout pass for collision detection and simple repulsion to avoid element overlapping. The new implementation will read generic graph files in GXL<sup>1</sup> format with arbitrary attributes to be suitable e.g. for testing with real world call graph data for software reverse engineering. Another objective is to visualize Annotation Graphs for linguistic analysis of language corpora, which also do not have tree-like structure and can benefit from locally optimized layout and partitioning.

<sup>1</sup>Graph eXchange Language, see <http://www.gupro.de/GXL/>

## References

- [1] M. S. T. Carpendale, D. J. Cowperthwaite, and F. D. Fracchia. 3-dimensional pliable surfaces: For the effective presentation of visual information. In *ACM Symposium on User Interface Software and Technology*, pages 217–226, 1995.
- [2] C.-S. Jeong and A. Pang. Reconfigurable disc trees for visualizing large hierarchical information space. In *IEEE Symposium on Information Visualization*, pages 19–25. IEEE Visualization, 1998.
- [3] T. A. Keahey. Getting along: Composition of visualization paradigms. In *Proceedings of the IEEE Symposium on Information Visualization*, pages 37–40. IEEE Visualization, 2001.
- [4] A. Kumar and R. H. Fowler. A spring modeling algorithm to position nodes of an undirected graph in three dimensions. Technical report, 1996.
- [5] W. Lai and P. Eades. A graph model which supports flexible layout function. Technical Report 96-15, Callaghan 2308, Australia, 1996.
- [6] P. Lüders and R. Ernst. Research report - improving browsing in information by the automatic display layout. In *IEEE Symposium on Information Visualization*, pages 26–33. IEEE Visualization, 1995.
- [7] B. Mirkin. *Mathematical Classification and Clustering*. Kluwer Academic Publishers, 1996.
- [8] T. Munzner. H3: Laying out large directed graphs in 3d hyperbolic space. In *Proceedings of the IEEE Symposium on Information Visualization*, pages 2–10. IEEE Visualization, 1997.
- [9] R. Pickett and G. Grinstein. Iconographic displays for visualizing multidimensional data. In *Proceedings of the 1988 IEEE Conference on Systems, Man, and Cybernetics*, pages 514–519, 1988.
- [10] G. Robertson, J. Mackinlay, and S. Card. Cone trees: Animated 3d visualizations of hierarchical information. In *CHI '91 Conference Proceedings on Human Factors in Computing Systems*, pages 189–194. ACM Press, 1991.
- [11] C. Ware and G. Franck. Evaluating stereo and motion cues for visualizing information nets in three dimensions. *ACM Transactions on Graphics*, 15(2):121–140, 1996.
- [12] G. Wills. Selection: 524,288 ways to say 'this is interesting'. In *Proceedings of the IEEE Symposium on Information Visualization*, pages 54–60. IEEE Visualization, 1996.