SUPPORTING VISUAL TECHNIQUES FOR GRAPHS DATA ANALYSIS IN THREE-DIMENSIONAL SPACE

Vitaly Zabiniako, Pavel Rusakov

Riga Technical University, Department of Applied Computer Science, 1/3 Meza, Riga, Latvia, Vitalijs.Zabinako@rtu.lv, Pavels.Rusakovs@cs.rtu.lv

Abstract. In this article authors provide outline and formal description of visual techniques that support analysis and extraction of patterns from information encoded in spatial graphs. Both known methods (such as transparency, magnification, etc.) are being presented and offered by the authors for the enhanced analysis (such as projective shadows, illumination distance, gradient stenciling, etc.). The goal of this research is to assist in graph data visualization and mining tasks by providing a set of supplementary techniques for effective information comprehension and analysis. Components of computer graphics framework required for each type of technique (for example – support of shaders, presence of stencil and depth buffers, etc.) are being listed. The result of this analysis is presented in tabular form, comparing fitness of identified techniques against different three-dimensional graph layout algorithms, semantic data domains and desired analysis processes. This allows to identify main requirements for computer graphics framework being used for data visualization as a part of particular graph visualization software system technical specification. Conclusion about achieved results is made. Information about potential future researches in this field is presented.

Keywords: graph, three-dimensional, visual, analysis, techniques.

1 Introduction

Visual data mining and multidimensional exploratory data analysis and navigation for Business Intelligence and Science are relatively new concepts that emerged at the end of 20-th century after the evolution of modern computer graphics (examples of early works on this topic can be found in [1], [7], [2]). There are many fields in IT industry that could benefit from visual data representation.

For example, the traditional concept of system analysis includes routines that require extensive visual support for efficient understanding and deriving requirements for software system being implemented. The primary supplemented activities in this case are connected with modelling system functionality (for example, in form of UML use case diagrams), its environment (outer systems and interfaces) and design (inner architecture).

Another field that could benefit from visual representation of data is transportation network management (such as railroad or air traffic control systems). In this case properties like geometric / geographic relationships between primary network elements (vehicles, stations, routes, etc.) and object tracking activities are of great interest.

In many cases it is useful to represent models under inspection as graphs. Depending on the semantics, graph nodes may represent logical diagram elements / geographical objects, etc. Similarly, edges may represent relationships between entities / routes, etc.

Graph visualization usually produces two-dimensional drawings in a plane. Although this is most common output model, the authors of this paper argue that in case of drawing of complex graphs (in terms of elements number and density of their interconnections), three-dimensional graph layout is more favorable as it may benefit from properties of additional supplementary algorithms and visual techniques that become possible in three-dimensional space – examples of modern researches in this field can be found in [4], [6], [9]. Previous research of existing graph drawing solutions that was performed by the authors is presented in [13]. Many other literature sources exist with descriptions of individual approaches, although less is done to summarize and perform comparative analysis of main features and implementation of these techniques with “de-facto” graphical frameworks.

This article provides summary and description of identified visual techniques (both existing and those proposed by the authors of this article) that may aid in data analysis. Each description is supplemented with formal model (if appropriate), pseudo code, requirements for computer graphics framework and evaluation of its fitness for different tasks.

Taking into consideration the aforesaid, it is possible to identify the following goal: to assist in graph data visualization and mining tasks by providing a set of supplementary techniques for effective information comprehension and analysis (the term “supplementary” implies non-obligatory opportunities for data analysis, as opposite to such primary tasks as graph topology manipulation or data visualization itself). In order to reach this goal, there are four tasks being defined: 1) to provide a description of visualization techniques; 2) to analyze theoretical background, practical implementation and potential application aspects for different data domains of

each technique; 3) to make summary of this information in tabular form; 4) to make conclusion about achieved results and further work.

2 Visual Techniques for Graphs Data Analysis

The *OpenGL* (*Open Graphics Language*) framework has been chosen for demonstration of implementation of visual techniques (although many different solutions, such as *Direct3D* / *Java 3D* fit for this task, *OpenGL* will be used due to its procedural nature and implementation of state machine model [11] that reduces number of separate graphical operators required for the visualization process).

2.1 Transparency

Transparency is well-known technique that offers ability to implement partial visual occlusion of objects. In physics transparency is defined as property of allowing light to pass through the object. In computer graphics imitation of transparency can be achieved by using formula (1).

$$C_f = (C_s * S) + (C_d * D) \qquad (1)$$

where: C_f – final color of the pixel; C_s – color of the incoming pixel; C_d – color of the original pixel; S, D –pixel blending coefficients.

This model implies that drawing an object in the scene takes into the account existing content of the frame color buffer. Each pixel of the object being drawn is weighted by the coefficient that defines how much new pixel color will affect content of the buffer. Usually D is defined as $S – 1$. In this case $S=1$ means that all final pixels will be substituted by incoming pixels (new object is not transparent), $S=0$ means that the scene will remain unaffected (new object is fully transparent) and any other S that belongs to the interval (0,1) will result in partially occluded objects.

In order to support transparency, graphics framework must allow reading individual pixel colors from color buffer. *OpenGL* supports this technique directly by providing a set of appropriate commands (here and further, only those commands that are essential for achievement of desired effects are included):

```
/*1*/ ... //draw scene
/*2*/ glEnable(GL_BLEND);
/*3*/ glBlendFunc(GL_SRC_ALPHA,
               GL_ONE_MINUS_SRC_ALPHA);
/*4*/ ... //draw transparent object
/*5*/ glDisable(GL_BLEND);
```

Code lines 2 and 5 ensure switching blending mode on/off, code line 3 allows setting coefficients to desired mode, code lines 1 and 4 assumes calling routines for drawing the scene and transparent object.

The result of implementation of this technique in custom graph visualization environment (refer to [12], [8], [5] for description of such systems) is shown in Fig. 1. (each technique in according implementation utilizes real-time dynamic data visualization).

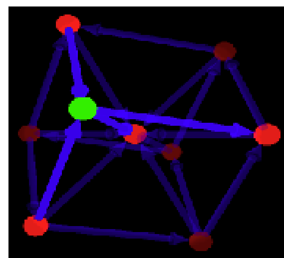

Figure 1. Transparent graph elements

Transparency is useful in case if certain object or a set of objects is chosen for further analysis (a node with its direct children and parents in Fig. 1.) – it helps to abstract from the other data by visually "weakening" it while still allowing to perceive the whole structure.

This technique is equally useful both for analysis that puts priority on topological characteristics of data (for example, in field of system analysis, visualization of WWW structure / "callgraphs", etc.) and the one that is concerned with spatial relationships of objects (for example, in field of logistics or real-world objects tracking).

2.2 Magnification

Magnification is another technique that allows to visually distinguish objects under inspection. In this case it is done by directly altering geometric properties of the objects. Magnification is related to scaling up visuals to be able to see more detail by increasing its resolution. Magnification must be interactive – as the user moves the reference pointer (usually – cursor) the system should track with it and show the new enlarged portion, similar to real world magnifying glass.

OpenGL supports magnification either through standard model-view matrix manipulation command or image post-processing. The first case can be implemented as follows:

```
/*1*/ glMatrixMode(GL_MODELVIEW);
/*2*/ glScalef(Xm,Ym,Zm);
/*3*/ ... //draw enlarged object
```

Code line 1 sets the required matrix mode, code line 2 modifies model-view matrix so that it applies scaling to object drawn with code line 3. Usually *Xm*, *Ym* and *Zm* are equal and correspond to desired magnification rate (setting these variables to "1" preserve original object size; values less than "1" will result in decrease of size).

In second case the content of color frame buffer after complete scene visualization is read, scaled and outputted back to buffer via appropriate raster image manipulation commands. The implementation of this approach is as follows:

```
/*1*/ ... //draw object to enlarge
/*2*/ glRasterPos2i(Mx-100,My-100);
/*3*/ glPixelZoom(X,Y);
/*4*/ glCopyPixels(Mx-50,My-50,
                   100,100,GL_COLOR);
```

Mx and *My* correspond to the reference pointer coordinates. Withdraw and height of the zooming region in this case is equal to 100 pixels. Code line 1 ensures that the object to enlarge already exists in the color buffer. Code line 2 specifies screen coordinates that correspond to the reference pointer. Code line 3 sets magnification rate. Code line 4 outputs the stretched image to the color buffer.

In order to support first magnification model, graphics framework must provide native matrix manipulation solution. The second approach doesn't require such property and may be used for any raster input. In both cases presence of the depth buffer is necessary for correct visualization.

The result of implementation is shown in Fig. 2.

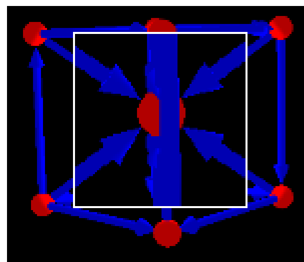

Figure 2. Visually magnified graph elements

Like with transparency, magnification allows to enhance perception capabilities of information under inspection. The main difference is that magnification doesn't affect appearance of other data.

This technique is useful virtually in any data analysis field. The main concern is which of the methods is applied – scaling with model view matrix doesn't affect resolution of the enlarged object (it is increased equally with the object size). Scaling with raster manipulation decreases image quality (resolution stays the same while size is increased).

2.3 Illumination distance

Illumination distance is proposed by the authors of this article and serves for the focusing on data. The idea is to visually "weaken" graph elements (by reducing color intensity) while being guided with certain pre-set function which takes under inspection geometrical distance between the region and current region. The distance is calculated according to the formula (2).

$$d = \sqrt{(x_i - x_a)^2 + (y_i - y_a)^2 + (z_i - z_a)^2} \quad (2)$$

where: d – distance between regions; x_i, y_i, z_i – coordinates of the centre of the region under inspection; x_a, y_a, z_a – coordinates of the centre of arbitrary region.

Calculated distance value serves as input to color intensity function according to the formula (3).

$$I = F(d) \tag{3}$$

where: I –object color intensity rate; F – user-chosen function.

In the simplest case function output might be inversely proportional to the argument. This yields gradual attenuation of graph elements colors – the selected object and its closest neighbor regions will stay bright while those situated further away will be darkened. More complex function will result in different behavior – for example piecewise defined function as in formula (4).

$$F(d) = \begin{cases} \frac{1}{d}, d < 10 \\ \frac{1}{10}, 10 \le d \le 20 \\ \frac{1}{d^2}, d > 20 \end{cases} \tag{4}$$

In this case there will be three regions – the closest with linear attenuation, followed by the one with constant mid-range intensity and the last, with quadratic attenuation rate. In order to support this technique, ability to define color of object being drawn is enough:

```
/*1*/ ... //calculate I=F(d)
/*2*/ glColor3f(R*I,G*I,B*I);
/*3*/ ... //draw illuminated object
```

R, G and B values correspond to original color of the graph element according to “Red-Green-Blue” model. The result of implementation of this technique is shown in Fig. 3.

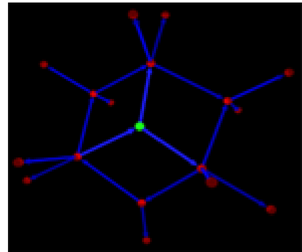

Figure 3. Distance-dependent data illumination

The effect of application of this technique is similar to the effect of transparency, although it gives additional control on data attenuation rate. In fact, transparency might be combined with this technique by substituting parameter S in formula (1) with parameter I in formula (3). This will result in user-defined gradual attenuation of transparency and/or color.

An example of usage of this technique is visual exploration of relationships between well-known “Twitter” members and their mutual following of each other tweets as presented in Fig. 4 – A. In this case, by illuminating one of the members in the whole graph and using distance-depended attenuation of other elements, it is possible to distinguish on the spot immediate followers of the selected member as it is shown in Fig. 4 – B.

Figure 4. Distance-dependent illumination of Twitter followers

2.4 Gradient stenciling

Gradient stenciling is proposed by the authors of this article and as it may be perceived from the name of this technique, uses both gradient images and stenciling concept. This technique if useful in case if layout of

graph spatial model places nodes in different layers (for example, orthogonal or hierarchical layout – refer to [11]). Considering that individual layers might be of particular interest for analysis, it is possible to define general gradient direction in each case (for example "from top to bottom of the screen" or "from left to right"), generate according image and apply it to the color of elements of the graph. The problem is that generated gradient usually is rectangular opaque image. Even by increasing its transparency it will affect the entire scene, not the individual graph elements, as it is desired. Possible solution of this problem is in "stenciling" technique. The stencil buffer, like color buffer in its essence is an array of values associated with each screen pixel. The difference is that it may contain user-defined values, allowing to make an image "mask". The idea is as follows: the scene (graph body) will be drawn first. When drawing each individual pixel color into the frame color buffer, stencil buffer is updated as well. After first step of data visualization we have a mask that corresponds to the graph body. This mask will be used in the second pass when gradient image will be drawn on top of the graph body. Only masked pixels will be affected in this step. That is why only graph body itself will be affected by color change. According schematic model is demonstrated in Fig. 5.

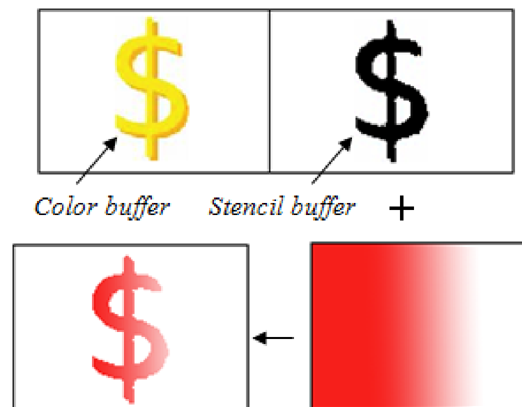

Figure 5. Gradient drawing with stencil mask

In order to support this technique, access to stencil buffer must be provided by the graphical framework together with its control commands. *OpenGL* framework allows to implement gradient stenciling with the code as follows:

```
/*1*/ glEnable(GL_STENCIL_TEST);
/*2*/ glStencilFunc(GL_ALWAYS,1,1);
/*3*/ glStencilOp(GL_ZERO,GL_ZERO,
                           GL_REPLACE);
/*4*/ ... //draw object
/*5*/ glStencilFunc(GL_EQUAL,1,1);
/*6*/ glStencilOp(GL_KEEP,GL_KEEP,
                              GL_KEEP);
/*7*/ ... //draw gradient
```

Code line 1 enables stencil test. Code line 2 sets the mode in which stencil test will always pass. Code line 3 forces for each pixel of the object being drawn to set corresponding stencil array element to value "1". Then graph body is drawn in code line 4. After that, in code line 5, the stencil test is set so that it will pass only if the stencil array element is equal to "1" and code line 6 specifies that the mask won't be changed anymore. Finally, the gradient image is drawn, altering only masked pixels.

The result of implementation of this technique to graph nodes is shown in Fig. 6.

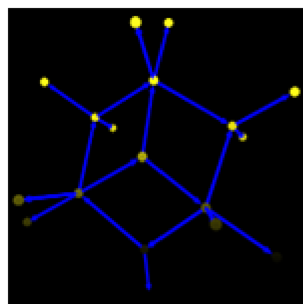

Figure 6. Gradient stencilling of graph nodes

In general, by using stenciling it is possible to apply any kind of raster image to the body of the graph. Gradient is particularly useful solution that allows to alter perception of hierarchical data nature and different space layers (due to explicit correlation between these concepts). Considering that there are different types of gradients (for example – linear, radial), it is possible to apply different shading strategies to the data in order to get different analysis patterns.

Gradient stenciling is useful when there is a need to explore highly organized networks, for example – neural *GMDH* (*Group Method of Data Handling*) networks that consist of multiple levels of interconnected neurons [14]. Analysis of topological properties of input layer, hidden layers and output layer (Fig. 7 – A) requires different strategies for distinguishing of elements. Applying of different gradients is necessary in each individual case – gradual intensity decrease (from left to right) for input layer (Fig. 7 – B), centered radial decrease for hidden layers (Fig. 7 – C) and left-to-right increase for output layer (Fig. 7 – D).

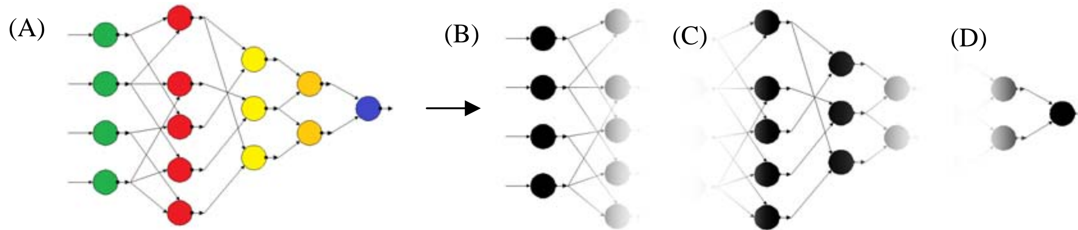

Figure 7. Gradient stenciling of GMDH neural network

2.5 Projective shadows

"Projective shadows" is the technique proposed by the authors of this article that allows to represent graph data in single instance of three-dimensional space and multiple instances of two-dimensional spaces simultaneously, using the concept of a shadow.

The idea is to draw the three-dimensional data model in iterative steps. The first step captures the model from the current position of the virtual camera like in all previously mentioned techniques. During next steps camera is placed so that it faces model orthogonally – directly from the top, front, left etc. Each rendering result is placed into separate texture. When all steps are complete, textures are placed on corresponding faces of the rectangular parallelepiped (or cube). The parallelepiped is drawn in the scene so that three-dimensional data model is situated at its centre, according to "room" concept. In this case each texture represents a projection or essentially a "shadow" of original data structure. This information visualization model is demonstrated in Fig. 8.

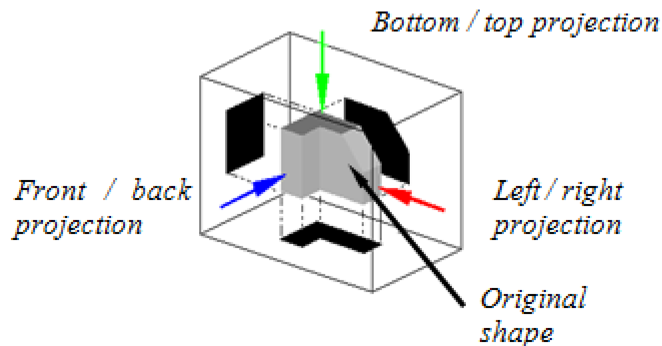

Figure 8. Data projection model

As the result, user perceives not only the graph itself, but he can also evaluate and choose one two-dimensional instance which suits for outputting it in a plane surface (for example – for printing the graph on a paper). In case if two-dimensional instances are updated each frame, modifications that user performs with the spatial graph will be reflected in all projections in real-time which makes this technique particularly useful.

In order to support this technique, graphical framework must provide access for rendering to texture which can be applied to the scene later. *OpenGL* framework allows to implement projective shadows with the code as follows:

```
/*1*/ ... //set matrix for camera
/*2*/ ... //draw object
/*3*/ for (x=1; x<=DimNum; x++)
/*4*/ {
/*5*/ ... //set matrix for dimension
/*6*/ ... //draw object
/*7*/ glCopyTexImage2D(GL_TEXTURE_2D,
            0,GL_RGBA,0,0,512,512,0);
/*8*/ }
/*9*/ //draw cube with textured faces
```

Code line 1 and 2 draw graph body as normal. Code lines 3 to 8 draw graph body as multiple projections (number of projection defined by variable *DimNum*) and save each generated image in corresponding texture. Code line 9 binds captured projections to the faces of the cube and draws this cube around the graph body.

The result of implementation of this technique is shown in Fig. 9.

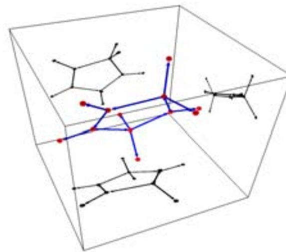

Figure 9. Projecting graph body onto the cube

Projective shadowing allows to get more detailed comprehension about data structure – when perceiving spatial model from particular point of view it is sometimes hard enough to judge how complex the graph topology in individual dimensions is. Multiple projections allow to get rid of this problem. This concept is similar to orthographic projection views in *CAD* (*Computer-Aided Design*) systems. The difference is that in this case shadows allow to explore topological relationships of multiple elements rather than purely geometrical properties of single object.

It is also suitable for any kind of tasks where the result of visualization must be presented or printed out as planar image.

The fact that the topological structure of the graph is simultaneously visualized in multiple projections allows for convenient identification of patterns. An example would be the exploration of mutual relationships between inherited classes of object-oriented software. As it is well known, one of the conceptual problems in this case is so called "diamond problem" of multiple inheritance which can be distinguished by the presence of cycles. Projective shadows allows for advanced exploration of such graphs (Fig. 10 – A) in three-dimensional space, while maintaining ability to conveniently identify such unwanted properties (Fig. 10 – B).

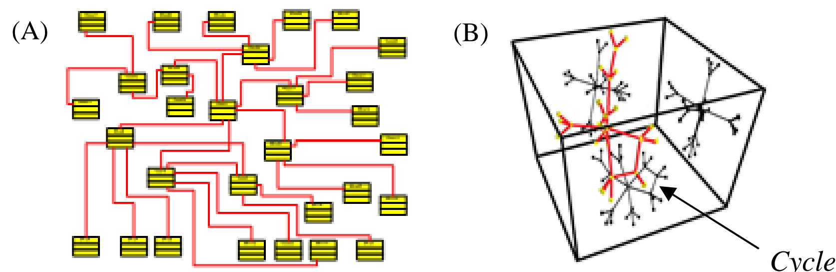

Figure 10. Projection shadows of a class diagram

2.6 Benedictine space

The concept of Benedictine space unites all attempts to artificially increase number of dimensions for representation of the data. The step from planar to spatial data drawing adds one spare dimension (depth), making it possible to map data contained in the element to more informative cortege $\{x,y,z\}$ instead of $\{x,y\}$.

The problem is that representing information in three dimensions "directly" is current limit both for computer graphics and human perception capabilities. The possible solution is to express additional dimensions

via auxiliary object properties. Common solutions include usage of auxiliary properties such as size, color, shape, transparency, etc. This concept applied to several data units is demonstrated in Fig.11.

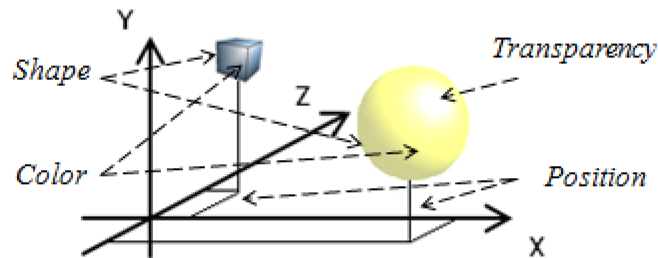

Figure 11. Data representation in multiple dimensions

This technique allows to encode multiple data characteristics into single spatial model. For example, structure of the company can be presented as graph in which location of nodes that represent branch offices correspond to their geographical coordinates, number of workers in particular office is represented with node size, while its specialization is marked by certain node shape.

In order to support custom user-defined objects graphical framework must provide not only the ability to influence size / color / transparency properties but it also must allow to import custom shapes (in case of spatial graph model – via standard 3D object file formats like ".obj" or ".x").

OpenGL framework allows to implement Benedictine space in its general form with code as follows:

```
/*1*/ glMatrixMode(GL_MODELVIEW);
/*2*/ glScalef(X,Y,Z);
/*3*/ glColor4f(R,G,B,A);
/*4*/ ... //load 3D shape
/*5*/ ... //draw loaded shape object
```

Code lines 1 and 2 ensure required change of the size by scaling the node. Code line 3 sets desired RGB color and alpha transparency value via single command. Code line 4 ensures loading of custom shape (either the one fetched earlier and stored on demand or directly – at the command execution time) and line 5 presents it on the screen taking into consideration all previously stated properties.

The result of implementation of this technique is shown in Fig. 12.

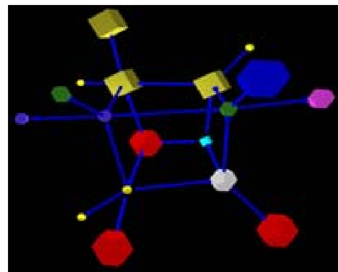

Figure 12. Graph data in Benedictine space

This shape of the node might be substituted with two-dimensional raster image. This is useful for graph visualization systems that deal with management of real-world objects (for example – in logistics). In this case particular object type is directly represented by its visual image which allows for user of the system to distinguish and filter out irrelevant objects as soon as possible.

Using images for data marking is presented in Fig. 13.

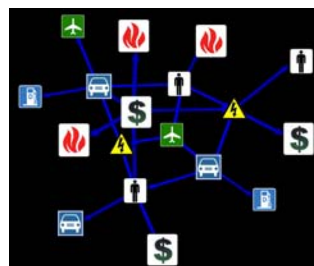

Figure 13. Benedictine space with images

Benedectine space is useful solution for representation of multidimensional data. Although its image-driven version is best suited for visualization of real world objects, its general form can be successfully adopted for any kind of analytical data, where analysis of topology of data structure is primary.

2.7 Visual clustering

Clustering in its essence is concerned with grouping objects in a way that allows distinguishing individual classes that include objects with similar properties. Visual clustering allows to decrease number of elements in the scene by substituting a set of geometrically close (using formula 2) or adjacent nodes (or edges – refer to [3]) with artificially constructed new single parent node that holds same relations to remaining graph topology which was previously true for all its children.

Visual model of such relationship is provided in Fig. 14.

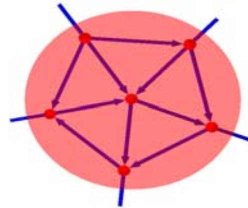

Figure 14. Node with its clustered children

It is important to signal in a certain way to the user of graph visualization system that particular node is a cluster that can be converted back to original set of nodes. Usually this can be done by increasing the size of a node and changing its color. More complex solution involves usage of transparency to hide children of a cluster partially. In this case number of graph elements stays the same – these are just placed within “fake” semi-transparent node and the clustering effect is purely visual.

Clustering allows to choose abstraction level at which system user will examine the data. Some systems provide recursive clustering capabilities – multiple clustered nodes may become children of higher-degree cluster, making it possible to collapse all data to the single root node at extreme case.

In order to support this technique, graphical framework must provide access for selection of individual nodes (both non-partitioned and partitioned) via reference pointer in order to expand / collapse clusters.

OpenGL framework allows to implement selection of elements via special mode that allows to define unique identifiers for each object and then returns an array of those identifiers that correspond to user-selected object. The code is as follows:

```
/*1*/ glSelectBuffer(64,SelArray);
/*2*/ glRenderMode(GL_SELECT);
/*3*/ ... //drawing of objects
/*4*/ hits = glRenderMode(GL_RENDER);
/*5*/ if (hits>0) nodeid=SelArray[0];
/*6*/ ... //changing graph structure
/*7*/ ... //drawing of objects
```

Code lines 1 and 2 initialize selection buffer and enable selection mode. Code line 3 draws the scene for the first time for selection mode. Code line 4 and 5 determine number of elements that were selected (clicked on) by user and if the selection buffer is not empty, identifier of the first selected element is retrieved. This identifier is used to decide how the graph structure will be affected by collapsing / expanding of the selected cluster (code line 6). Code line 7 outputs modified graph body to the frame buffer.

The result of implementation is shown in Fig. 15.

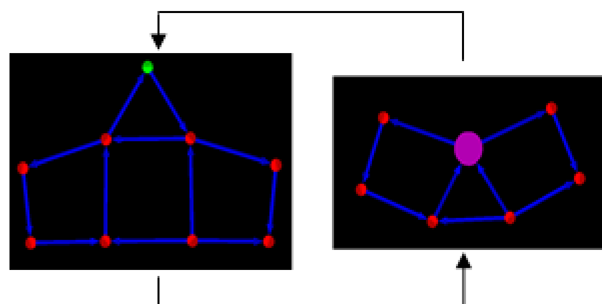

Figure 15. Clustered graph node

Clustering is common activity in the field of analysis where abstraction is required to filter out irrelevant details of the information. Other fields that could benefit from cluster analysis are as follows: web mining, image processing, machine learning, artificial intelligence, pattern recognition, social network analysis, bioinformatics, geography, geology, biology, psychology, sociology, customers' behavior analysis, marketing to e-business, etc.

It is less common in the field of management / tracking real world objects, because those are relatively independent unique units that must be viewed and processed by the operator separately.

2.8 Auxiliary navigation

In case if information volume that must be perceived or analyzed is big enough, the user must be provided with an informative tool that help him to navigate through the data.

This task is especially important in management of real world objects where the concept of data navigation corresponds to the general concept of geographical navigation. This, in turn, requires implementation of "virtual compass".

The implementation of this tool is so that the pointer of the compass always points to the pre-defined direction (in case of geographical navigation – north). This enables quick navigation capabilities regardless of current position of virtual camera.

Like in the case of Benedictine space, graphical framework must provide a way to load visual image of the compass from the file or construct it from built-in set of shapes.

OpenGL framework allows to implement auxiliary navigation with the code as follows:

```
/*1*/ glMatrixMode(GL_MODELVIEW);
/*2*/ glRotatef(Angle,Xr,Yr,Zr);
/*3*/ glTranslatef(Xt,Yt,Zt);
/*4*/ ... //draw objects in the scene
/*5*/ glPushMatrix();
/*6*/ glLoadIdentity();
/*7*/ glRotatef(Angle,Xr,Yr,Zr);
/*8*/ ... //draw navigation tool
/*9*/ glPopMatrix();
```

The general idea is to apply both rotation and translation modifications to the scene being drawn (code lines 1-4) but only rotation component of the camera must be applied to the model of navigation tool (code lines 7-8). To achieve this effect, the state of the model-view matrix must be saved (code line 5) reset to initial identity matrix (code line 6) and finally – restored after drawing of the navigation tool (code line 9).

The result of implementation of this technique is shown in Fig. 16.

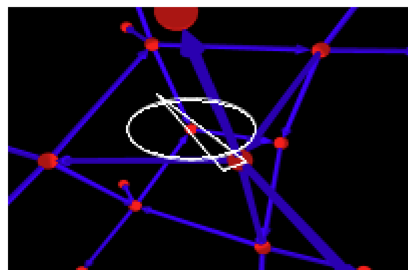

Figure 16. Navigation with auxiliary tool

Auxiliary navigation might be also adapted to the field of analysis. In this case there is no meaning in trying to define a "global direction", because all data is essentially consistent. But still, the navigation tool might be used in case if there is a need to track relationship between selected nodes. For example, the "compass" may always point to the parent of selected node – this allows to improve navigation capabilities between data hierarchies.

2.9 Data blurring / sharpening

Another effective visual data filtering mechanism is connected with blurring and sharpening. Three different models might be used – blurring of irrelevant data units; sharpening of data units under inspection; combination of both mentioned modes.

Both blurring and sharpening require per-pixel access to the scene visualization result. The mathematical model of changing blur level is based on a gradual modification of each pixel by taking into

consideration current color values of its neighbor pixels and averaging them according to a set of coefficients that form blurring / sharpening matrixes (5).

$$Color = \sum_{i=0}^{8} sample[i] * M[i] \quad (5)$$

Blurring (B) and sharpening (S) matrices are presented in formula (6).

$$B = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 1 & 2 \\ 1 & 2 & 1 \end{bmatrix} / 13 \quad S = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix} \quad (6)$$

In order to perform changing of blur level according to this model, graphics framework must provide access to shaders (detailed information about the purpose of vertex and pixels shaders can be found in [10]). In this case pixel shader is of particular interest, as it can perform requested operation on per-pixel basis. *OpenGL* framework allows to implement change of level of blurring with code as follows:

```
/*1*/ for (i=0; i<9; i++)
/*2*/ sample[i]=texture2d(sampler,
        gl_TexCoords[0].st+offset[i]);

/*3*/gl_FragColor=(sample[0]+
          (2*sample[1])+sample[2]+
          (2*sample[3])+sample[4]+
          (2*sample[5])+sample[6]+
          (2*sample[7])+sample[8])/13;
```

Code line 1 and 2 performs saving of pixels of scene texture into the array *sample*. This array is then used for calculating final color value *gl_FragColor* for current pixel at code line 3. It is possible to use different matrixes (for example 5x5 instead of 3x3) for calculation of image color. The choice of the matrix size affects blurring strength – the bigger is the matrix, the stronger is blurring effect.

The result of implementation is shown in Fig. 17.

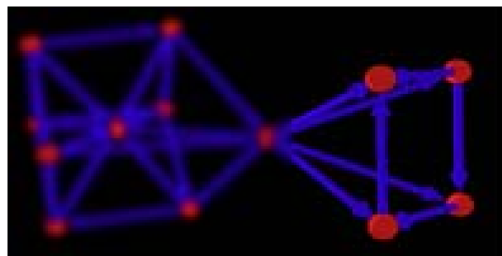

Figure 17. Filtering information with blurring

Data blurring is similar to image post-processing version of magnification in the sense that it allows to focus on certain information regions, although its logic is inverse – visual weakening of non-selected elements. Additional aspect is that the level of blurring can be dynamically changed according to user commands or based on the pre-defined formula, like with illumination distance technique.

3 Conclusions

In order to summarize all previously stated characteristics of different supplementary techniques, the comparison table is constructed (Table 1). It contains information about most important aspects of each technique – desired result of its application, implementation requirements, compatible graph layouts (refer to [11]) and possible application domains (based on description provided in chapter 2). It is possible to conclude that visual scene management can be useful for different kind of tasks – starting with helping to focus on data portion being analyzed and ending with general data navigation and representation routines.

General techniques exist that might be applied regardless data semantics, but it is possible to define two sets of application domains that require unique types of techniques – the domain where topological properties of data is of major importance and the one with primary geometrical characteristics. As it was stated previously, visual clustering is more appropriate for the first type of tasks, while auxiliary navigation plays important role in second case.

Potential of the graphical framework is crucial in all cases – even basic visual techniques require such features as matrix manipulation or depth buffer. This means that only technologies with advanced architecture, such as *OpenGL*, *Direct3D*, *Java3D* and similar may satisfy this demand (also, taken into consideration driver-level optimized performance which is crucial in case of visualization of graphs containing thousands or even more elements).

Table 1. Comparison of visual techniques.

Nr.	Name of the technique	Desired result	Requirements to graphical framework	Compatible graph layouts	Application domains
1	Transparency	Focusing on data under inspection; decreasing influence of other data.	Reading of individual pixel colors from color buffer.	Force-based; hierarchical; tree; orthogonal.	Requirements analysis; management / tracking of real world objects; web mining; callgraphs.
2	Magnification	Focusing on data under inspection.	Matrix manipulation; depth buffer.	Force-based; hierarchical; tree; orthogonal.	General data analysis.
3	Illumination distance	Focusing on data under inspection; defining custom influence rate of other data.	Ability to define color of object being drawn; depth buffer.	Force-based; hierarchical; tree; orthogonal.	Requirements analysis; management / tracking of real objects / networks. Refer to section 2.3 for an example.
4	Gradient stenciling	Focusing on data under inspection; hiding irrelevant information.	Access to stencil buffer.	Hierarchical; orthogonal.	Data with distinctive hierarchical nature. Refer to section 2.4 for an example.
5	Projective shadows	Evaluating data complexity; finding best layout for data representation in a plane.	Access for rendering to texture; depth buffer.	Force-based; hierarchical; tree.	Requirements analysis; management / tracking of real objects; tasks where the result must be presented in a plane; pattern regognition. Refer to section 2.5 for an example.
6	Benedictine space	Representing multiple data attributes in limited space dimensions.	Ability to set size / color / transparency / import custom shapes; depth buffer.	Force-based; hierarchical; tree; orthogonal.	Requirements analysis; management / tracking of real objects; representation of multidimensional data.
7	Visual clustering	Finding similar data units and treating them as single object.	Access for selection of nodes via reference pointer; depth buffer.	Force-based; hierarchical; tree.	Requirements analysis; web mining; artificial intelligence; social network analysis; bioinformatics, etc.
8	Auxiliary navigation	Maintaining local/global references while navigating through data.	Ability to load visual image of the navigation tool from the file / construct it from built-in primitives.	Force-based; hierarchical; tree; orthogonal.	Tracking of real objects.
9	Data blurring / sharpening	Focusing on data under inspection; dynamically decreasing rate of irrelevant information.	Access to pixel shaders; depth buffer.	Force-based; hierarchical; tree; orthogonal.	General data analysis.

A number of mentioned techniques have been integrated in custom graph visualization system "*3DIIVE*". Those are implemented as a part of inner repository of visualization techniques, according to general object-oriented approach (for additional information on implementation refer to [12]).

This research not only summarizes and provides theoretical description of already well-known visual techniques, but also demonstrates how certain proposed techniques can be used to gain benefit and enhance perception capabilities of the user in different problem domains, such as exploration of social networks, neuron networks and object-oriented software architecture.

While continuing this research, it is possible to seek for opportunities to combine useful properties of aforementioned techniques, like it was proposed for using of illumination intensity definition formulas (4) for transparency and magnification. This may result in development of new visual approaches that will increase usability and application of data visualization.

References

[1] **Asimov D.** The grand tour: A tool for viewing multidimensional data. *SIAM Journal of Science & Stat. Comp.* – 1985, vol. 6., pp. 128–143.

[2] **Chambers J. et. al.** Graphical Methods for Data Analysis. – *Chapman & Hall*, 1983, p. 395.

[3] **Cui W. et. al.** Geometry-Based Edge Clustering for Graph Visualization. *Visualization and Computer Graphics, IEEE Transactions*, 2008, vol. 14., pp. 1277-1284.

[4] **Keim D.** Information Visualization and Visual Data Mining, *IEEE Transactions on Visualization and Computer Graphics*, 2002, pp. 1–8.

[5] **Lecerf L., Chidlovskii B.** Visalix: A Web Application for Visual Data Analysis and Clustering. *ACM KDD 2009 (International Conference on Knowledge Discovery and Data Mining)*, Paris, France, June 28-31, 2009.

[6] **Lee B. et. al.** Task Taxonomy for Graph Visualization. *Proceedings of the AVI Workshop on BEyond time and errors novel evaluation methods for information visualization*, 2006, pp. 81-85.

[7] **Pickett R., Grinstein G**. Iconographic displays for visualizing multidimensional data. *In Proc. of IEEE Conf. On Systems, Man and Cybernetics, IEEE Press, Piscataway – NJ*, 1988, pp. 514–519.

[8] **Schulz H. et. al.** A Framework for Visual Data Mining of Structures. *Conferences in Research and Practice in Information Technology*, Vol. 48., 2006, pp. 157-166.

[9] **Wong P. et. al.** Dynamic Visualization of Graphs with Extended Labels. *IEEE Symposium on Information Visualization*, 2005, pp. 73-80.

[10] **Wright R., Lipchak B.** OpenGL SuperBible (third edition). *USA – Sams*, 2004, p. 1200.

[11] **Zabiniako V., Rusakov P.** Analysis of Visualization Problems of Graphs and Models of Graphs. *The 47th Scientific Conference of Riga Technical University*, 2007, pp. 138-148.

[12] **Zabiniako V., Rusakov P.** Development and Implementation of Partial Hybrid Algorithm for Graphs Visualization. *The 48th Scientific Conference of Riga Technical University*, 2008, pp. 192-203.

[13] **Zabiniako V., Rusakov P.** Definition of General Requirements for Graph Visualization Software. *The 49th Scientific Conference of Riga Technical University*, 2009, pp. 168-179.

[14] **Zjavka L.** Generalization of Patterns by Identification with Polynomial Neural Network. *Journal of Electrical Engineering*, Vol. 61, No. 2, 2010, pp. 120–124.