Beamtrees : Compact Visualization of Large Hierarchies

Frank van Ham, Jarke J. van Wijk
Technische Universiteit Eindhoven
Dept. of Mathematics and Computer Science
P.O.Box 513, 5600 MB Eindhoven, The Netherlands
{fvham, vanwijk}@win.tue.nl

Abstract

Beamtrees are a new method for the visualization of large hierarchical data sets. Nodes are shown as stacked circular beams, such that both the hierarchical structure as well as the size of nodes are depicted. The dimensions of beams are calculated using a variation of the treemap algorithm. A small user study indicated that beamtrees are significantly more effective than nested treemaps and cushion treemaps for the extraction of global hierarchical information.

1. Introduction

The visualization of large hierarchical datasets is an important topic in the visualization community. The main problem is the limited amount of display space. Traditional node link diagrams lead to cluttered displays when used to visualize more than a few hundred nodes. They are therefore unsuitable to visualize, for instance, an average directory structure, containing tens of thousands of nodes. Possible solutions to this problem are threefold: We can increase available display space, by either using three dimensional and/or hyperbolic spaces; we can reduce the number of information elements by clustering or hiding nodes; or we can use the given visualization space more efficiently by using every available pixel. Of the latter category of solutions treemaps are the prime example.

Treemaps are compact displays, which are particularly effective to visualize the size of the leaves in a tree. However, one important drawback of treemaps is that the hierarchical structure is harder to discern than in conventional tree browsers: All space is used for the display of leaf nodes, and the structure is encoded implicitly. A number of attempts have been made to overcome this problem, the most notable being the use of nesting and shading.

Inspired by treemaps, we present a new algorithm that visualizes the structure more explicitly. The user is enabled to adjust the visualization to his or her preferences: Either all space can be used for the leafnodes, which results in a conventional treemap, or more space can be used for the display of the hierarchical structure, which results in a *beamtree*. More specifically: where nested treemaps use nesting to indicate a parent-child relationship, we use overlap (Fig 1). The hierarchical structure is visualized as a stack of rectangles and shading and 3D are used to strengthen the perceived depth in the structure.

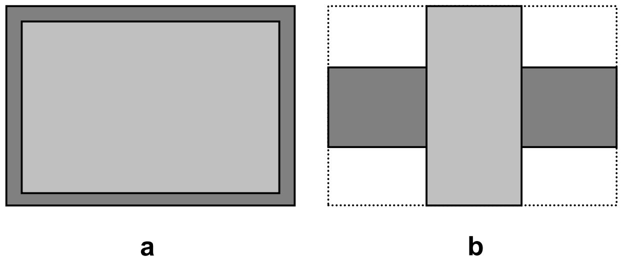

Figure 1. Displaying hierarchical data by (a) Nesting (b) Overlap

Section 2 discusses the basic treemap algorithm as well as a number of variations. The beamtree algorithm is presented in section 3 and section 4 discusses the results of a user test comparing treemaps to beamtrees. Finally, section 5 summarizes the results and discusses future work.

2. Related Work

Treemaps, invented by Johnson and Shneidermann in 1991 [6], are rectangular displays of trees that use the available visualization space very efficiently. The basic treemap algorithm is straightforward: An arbitrary rectangle representing the root node of the tree is horizontally subdivided into smaller rectangles, where each smaller rectangle represents a child of the root node, with the area of the rectangle being proportional to the size attribute of the node. This process is then repeated recursively, with the direction of the subdivision alternating between horizontal and vertical. Treemaps are especially useful for the display of large hierarchical datasets in which the size attribute of a node is important. As the size of each rectangle is proportional to the size of the node, one can easily spot the largest nodes in the tree.

However, standard treemap representations have two problems. Firstly, the method often leads to high aspect ratios (i.e. long thin rectangles). These make it more difficult to compare sizes and lead to interaction problems. Secondly, since the leaf nodes take up all of the visualization space, no space remains for the internal nodes of the tree. This makes it difficult to reconstruct the hierarchical information from the treemap, especially when dealing with large, deep trees. Recent research has tried to provide answers for both of these problems. New subdivision schemes have been developed by [2,7,9] amongst others, producing better aspect ratios but worsening layout stability. Methods to improve the display of hierarchical information include nesting [6,9], line width, color and, more recently, shading [10]. However, these methods still require a significant cognitive effort to extract the actual tree structure from the picture, especially when compared to the traditional node-link visualization. The use of nesting also influences the area of nodes deeper in the hierarchy since more space has to be reserved for the borders. We present a new approach that uses spatial ordering to display hierarchical ordering, while maintaining the proportionality property of treemaps.

3. Beamtrees

Since treemaps already use two dimensions to display size, we use a third dimension, i.e. display depth, to indicate depth in the tree. However, all of the available display space is already used up by the leaves, so we have to reduce the size of the leaves. As a first step we adapt the standard treemap algorithm: Instead of displaying the leaves space-filling we render all rectangles representing nodes with a scaled width. In other words, we reduce their dimension perpendicular to the direction of subdivision. Figure 2 shows that, as the scaling factor decreases, the partial occlusion creates a stronger depth effect.

A disadvantage is that leaf nodes tend to become very thin, even stronger than with standard treemaps. To reduce this problem and to decrease the number of separate elements in the display, we therefore treat leaf nodes differently. Instead of scaling their width, we use a treemap style subdivision algorithm: We assign to each leaf node a section of its parent node that is proportional to its size (Fig 2e). To provide for a more pleasant display we sort the children of a node, such that all leaf nodes are aggregated at the top or left side of their parent rectangle (Fig 2f). Since all rectangles are scaled with the same scale factor their size is still proportional to the size of the node they represent.

This display still suffers from problems however. Firstly, many non-leaf nodes have touching edges, making it more difficult to perceive them as separate visual entities. This can be resolved by also applying a scale factor to their length, using constraints to ensure the structure doesn't break up in separate pieces. We elaborate on this in section 3.1. Secondly, the large number of overlapping rectangles makes the resulting display abstract and hard to interpret. This can be significantly improved by the use of shading (section 3.2)

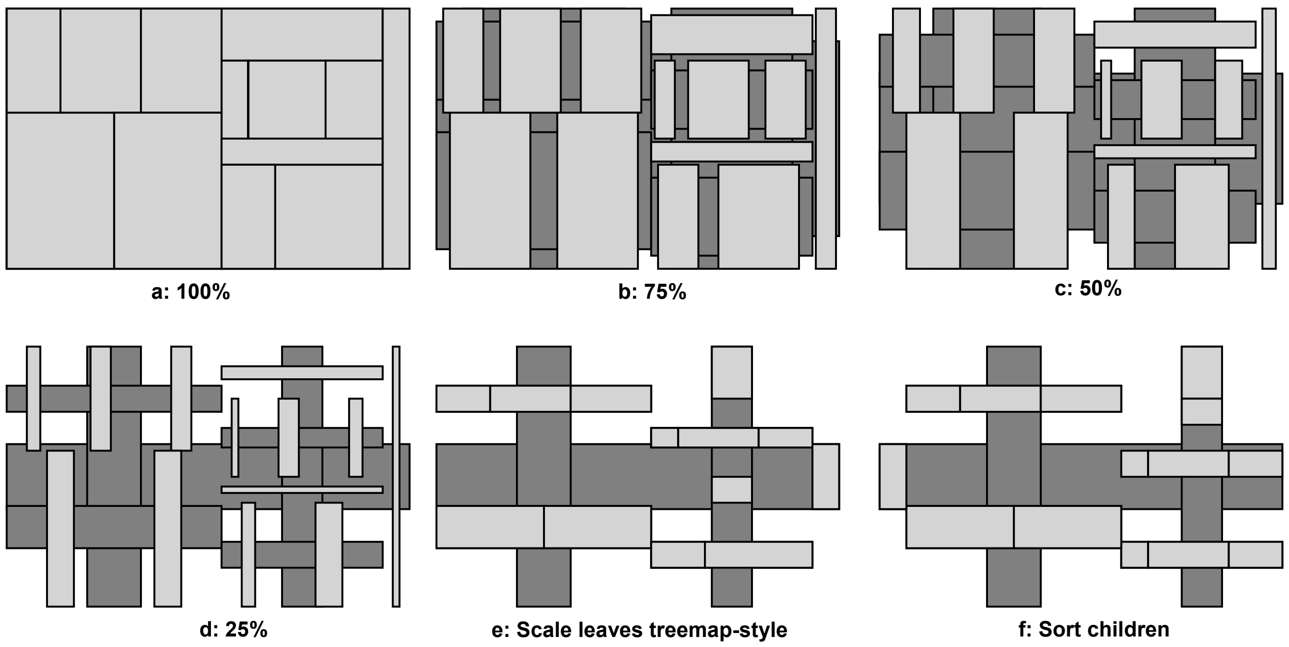

Figure 2. Scaling a treemap to a beamtree.

3.1 Constrained scaling

Since we did not scale the length of non-leaf rectangles, many of them still have touching edges (Fig 3a). As this makes it more difficult to perceive them as different visual entities we therefore also scale the length of each rectangle. We obviously run into problems when applying unconstrained scaling to internal nodes, since the structure might fall apart into separate pieces when doing so. Another problem is that due to scaling, leaf nodes may overlap, which is also not desirable (Fig 3b). We therefore need constraints that limit the scaling of internal nodes. Leaf nodes do not pose a problem in this case, since their size only depends on the scaled size of their parent node.

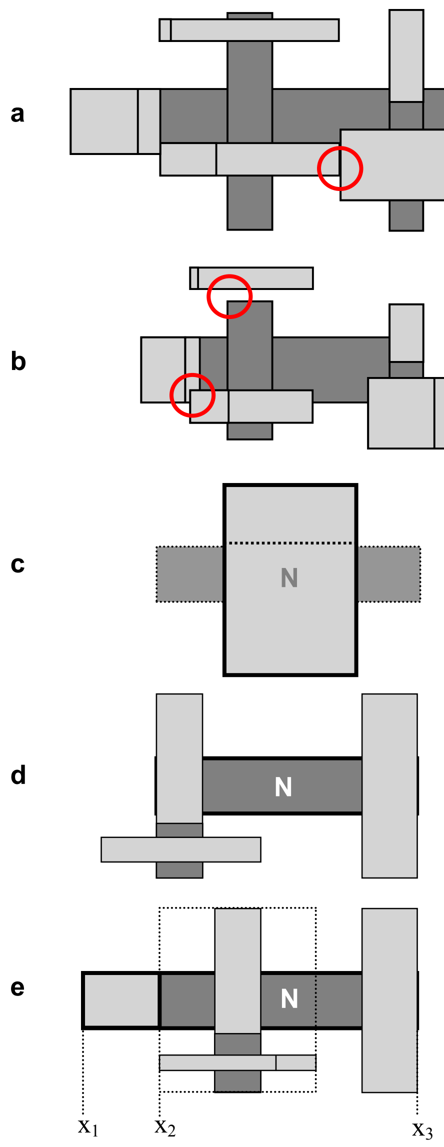

Figure 3. Scaling lengths

We calculate constraints bottom up in the tree, starting at the deepest internal nodes. We discern three cases for an internal node *N*:

1. **N has only leaf nodes as children:** In this case we don't need to compute constraints, since leaf nodes get scaled with their parent by definition (Fig 3c);
2. **N has only non-leaf nodes as children:** To avoid the structure from breaking up into separate pieces we use the front edge of the rectangle representing the first child of *N* as a lower bound and the back edge of the rectangle representing the last child of N as an upper bound (Fig 3d);
3. **N has both leaf and non-leaf nodes as children:** We can use the back edge of the last child node of *N* (i.e. x_3) in the same manner as we did in the previous case. Since *N* also has leaf nodes as children we have to take space for them into account to avoid overlapping structures. Bound x_2 is equal to the front edge of the bounding box encasing the first non-leaf child of *N*. Since we also know the total size of all leafnodes $S(L)$ and the total size of all child nodes $S(N)$, we can derive x_1 by solving (Fig 3e):

$$\frac{x_2 - x_1}{x_3 - x_1} = \frac{S(L)}{S(N)}$$

Because the length of each rectangle might get scaled differently due to the above constraints, the area of a rectangle may no longer be proportional to the size of the node it represents. We solve this problem by scaling down the width for these rectangles appropriately, so the resulting area remains proportional to the node size.

3.2 Algorithm

To show that the above-mentioned cases can easily be integrated in the conventional treemap algorithm, we present the entire beamtree algorithm here in detail. We use the following data types:

```
Dir = (X,Y);
Bound = (Min, Max);
Rectangle = **array**[Dir, Bound] **of** real;
Node = **class**
        *sizeN* : real;
        *sizeL*: real;
        *child* : **array**[1..*nchild*] **of** Node;
        *nleaf*: 1..*nchild;*
        *rt* : Rectangle;
        *rs* : Rectangle;
        **function** *rbound*: Rectangle;
    **end;**
```

Node attributes *sizeN* and *sizeL* store *S*(*N*) and *S*(*L*) respectively. Array *child* contains *nchild* childnodes, sorted such that the first *nleaf* nodes in this array are leafnodes. The dimensions of the rectangle representing the node in a regular treemap are stored in *rt*, while *rs* stores the scaled version of the same rectangle. Finally, function *rbound* returns the bounding rectangle for a scaled treemap rectangle and its children.

To facilitate notation we use three auxiliary functions: Function *Alt* alternates the direction of subdivision (i.e. *Alt*(X) = Y and *Alt*(Y) = X). Functions *SetSize* and *GetSize* respectively assign and return the length or width of a rectangle:

```
procedure SetSize(var r: Rectangle; d: Dir; s: real);
var c: real;
begin
   c := (r[d, Min] + r[d, Max])/2;
   r[d, Min]  := c – s/2;
   r[d, Max] := c + s/2
end;

function GetSize(r: Rectangle; d: Dir): real;
begin
  GetSize := r[d, Max] – r[d, Min]
end;
```

The main algorithm is similar to the original treemap algorithm from [6], using two extra procedures to scale down the rectangles. Note that non-leaf nodes have to be scaled before leaf nodes, because for scaling the latter we need information on the (possibly constrained) scaled length of their parentnode.

```
procedure Node.BeamTreeLayout(r: Rectangle; d: Dir);
var i : integer; f: real;
begin
  rt := r;
  if sizeN > 0 then f := GetSize(rt, d) / sizeN else f :=0;
  for i := 1 to nchild do
  begin
    r[d, Max]:= r[d, Min] + f * child[i].sizeN;
    child[i].BeamTreeLayout(r, Alt(d));
    r[d, Min]:= r[d, Max]
   end;
  if nchild > 0 then ResizeNode(d);
  if nleaf  > 0 then ResizeLeavesOfNode(d);
end;
```

Scaling of internal nodes is done in method *ResizeNode*. Global parameters *LengthScale* and *WidthScale* contain the scale factors for the length and width respectively. In practice, values of 0.95 for *LengthScale* and 0.35 for *WidthScale* give good results. The scale factor is applied to the original treemap rectangle *rt* in the second line. If we are dealing with a case 2 or case 3 internal node (see paragraph 3.1), we have to apply constraints to the scaled rectangle. Finally, the difference between the unconstrained and constrained scaled lengths is taken into account when scaling the width, to maintain size proportionality.

```
procedure Node.ResizeNode(d: Dir);
var L,W, x1, x2, x3: real;
begin
   rs := rt;
   L  := LengthScale* GetSize(rt, d);
   SetSize(rs, d, L);

   if nleaf = 0 then /* Case 2 */
   begin
      rs[d, Min]:= min(rs[d, Min],   child[1].rs[d, Min]);
      rs[d, Max]:= max(rs[d, Max], child[nchild].rs[d, Max])
   end else
   if nleaf  < nchild then /* Case 3 */
   begin
     x2 := child[nleaf + 1].rbound[d, Min];
     x3 := max(rs[d, Max], child[nchild].rs[d, Max]);
     x1 := x3 – (x3–x2)/( 1– sizeL / sizeN);
     rs[d, Min]:= Min(rs[d, Min], x1);
     rs[d, Max]:= x3
   end;

   W := WidthScale*GetSize(rt, Alt(d))*L/GetSize(rs, d);
   SetSize(rs, Alt(d), W);
end;
```

In a last step we resize the leaves of a node in a treemap style fashion, using the scaled size of their parent as the initial rectangle.

```
procedure Node.ResizeLeafsOfNode(d: Dir);
var i : integer; f : real; r : Rectangle;
begin
   r := rs;
  if sizeN > 0 then f := GetSize(r, d) / sizeN else f := 0;
   for i := 1 to nleaf do
   begin
     r[d, Max] := r[d, Min] + f * child[i].sizeN;
     child[i].rs := r;
     r[d, Min] := r[d, Max]
   end
end;
```

3.3 Beamtree Visualisation

Though occlusion is a strong depth cue, it also presents some new problems, the most notable being the fact that overlapping rectangles tend to break up into visually separate pieces. A solution to this problem is the use of shading [5] to indicate the direction of subdivision. As such, nodes are no longer visualized as two-dimensional rectangles but tend to resemble three-dimensional round beams, further strengthening the perceived depth in the picture. Additional depth cues can be provided by atmospheric attenuation and cast shadows (Fig. 4). Figure 5a shows a file system rendered in 2D using the above-mentioned cues.

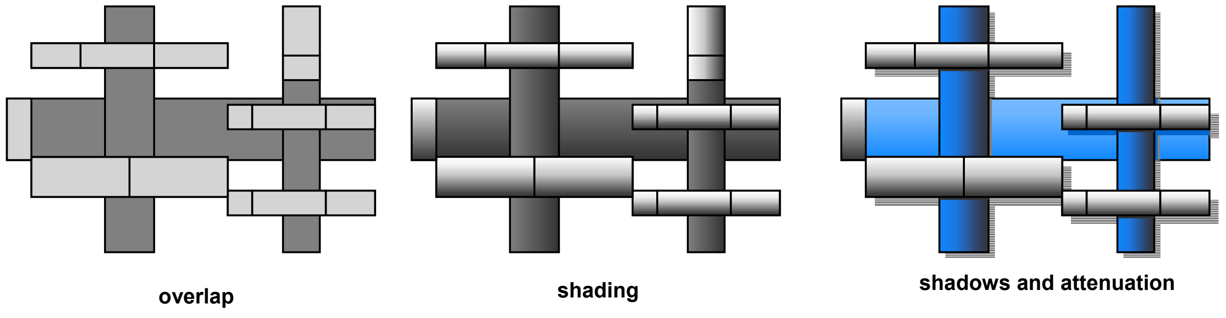

Figure 4. Adding visual cues

Another strong depth cue is motion parallax. We implemented a prototype using OpenGL, in which the user can rotate the beamtree structure and return to the top-down view at the press of a button. An extra advantage of a three dimensional view is that the user can move his or her viewpoint to a point parallel to a beam direction and view the entire structure in a more conventional way with one display axis indicating node depth (Fig 5c). As height for each beam we choose the minimum of width and length, as this provides the most aesthetically pleasing picture. Disadvantage is that, with height varying over each beam, beams at the same level in the tree do not necessarily get laid out at the same depth. We therefore provide an option to *layer* the stack of beams, that is, display each beam at a depth proportional to its depth in the tree (Fig. 5d). Connections between beams are indicated by lines when beams are displayed in layered mode. To make it easier for the user to maintain context, transitions to and from layered mode are animated.

Instead of explicitly modeling every leaf node in 3D, we used procedurally generated one-dimensional texture maps to represent leaves. Depending on the number of leafnodes of the node under consideration, resolution of the texture can be increased. Using this approach we were able to render more than 25,000 nodes in real-time on a PC with a GeForce2 graphics card (Fig 5b).

4. User test

A small pilot study was conducted to test if a combination between explicit hierarchical information and treemap-style size information is an improvement over existing methods. Although a recent study [1] suggests treemaps are not the most ideal visualization tool for small hierarchies, we feel there are not many alternatives for larger (say over 200 nodes) hierarchical structures. We therefore conducted a similar user test, comparing both the 2D and 3D beamtrees to cushion treemaps, as implemented in SequoiaView [8], and nested treemaps, as implemented in HCIL's Treemap 3.2 [4]. Twelve coworkers participated in the experiment, none of which were familiar with the concept of beamtrees, although seven were familiar with general treemap concepts.

4.1 Setup

The experiment was set up as a 4 x 5 factorial design, with 5 tasks for each of the 4 visualization methods (2D Beamtrees, 3D Beamtrees, Nested Treemaps and Cushion Treemaps). The tasks focused on:

- **File size:** Users had to select the three largest leaf nodes. Any node could be selected, and a response was judged correct only if all three largest leaves were selected.
- **Tree topology**: Users had to indicate level of a predetermined node. To avoid potential confusion the level of the root node was defined as 1.
- **Tree topology**: Users had to indicate the total number of levels in the tree.
- **Tree topology**: Users indicated the deepest common ancestor of two predetermined nodes.
- **Node memory**: In order to test how fast user could achieve a mental map of the tree structure, they were asked to memorize the positions of two predetermined nodes. If the user believed he or she could remember the location, the view was closed and reshown at a different size, to prevent users from remembering the screen location instead of node location. Users then had to indicate the positions of both nodes.

4.2 Procedure

We created 4 smaller randomized trees consisting of approximately 200 nodes and 4 randomized larger ones of approximately 1000 nodes. Node sizes were distributed using a log-normal distribution.

For each visualization method participants had to perform all 5 tasks for both a smaller and a larger tree, for a total of 40 tasks per participant. Trees and methods were matched up randomly and the order of visualization methods was also randomized. The order of tasks to be performed for each tree remained the same throughout the experiment.

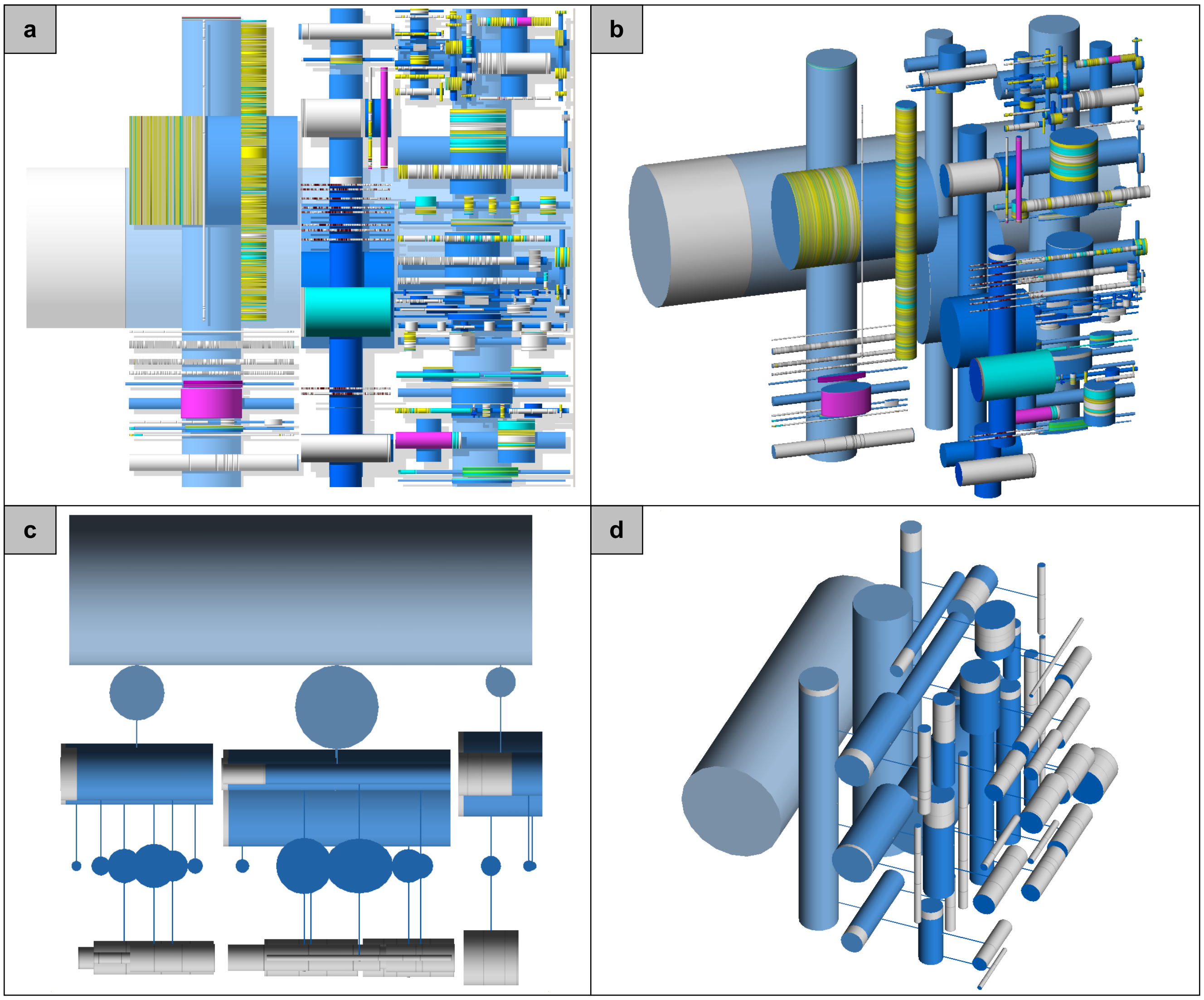

Figure 5. **a) Filesystem rendered in 2D, b) Same filesystem in 3D, c) Layered orthogonal view, d) Layered isometric view**

All visualizations used straightforward slice-and-dice partitioning and were displayed at a resolution of 1280x1024 pixels. Nested treemaps used a 3-pixel border. Beamtrees used scale factors of 0.35 for width and 0.95 for length. Node labels were not displayed, so users had to rely entirely on the visualization instead of textually encoded information. Detail size information about each node was still available by means of a tooltip window that popped up when a user moved the mouse over a node.

If users indicated they were not familiar with treemaps they were given a short explanation on the general principle of treemaps, followed by a small test to see if they had understood the concept. The tests for each visualization method were preceded by an explanation on how to interpret the visualization method and what tasks had to be performed. Subsequently, one timed trial run was performed to make the participant more comfortable with the task. During this explanation and trial run the participant was free to ask questions. If the participant was convinced he or she understood the visualization method and the tasks to be performed, actual testing began. Participants were not allowed to ask questions and no feedback on response time and accuracy was provided during the timed run.

Timing was self-paced and started immediately after the user was asked a question. Timing ended when the user verbally indicated he or she had found the answer, or, in the case of the memory task, thought he or she could remember the location of the nodes. At the end of all tests users rated the methods based on their subjective preference.

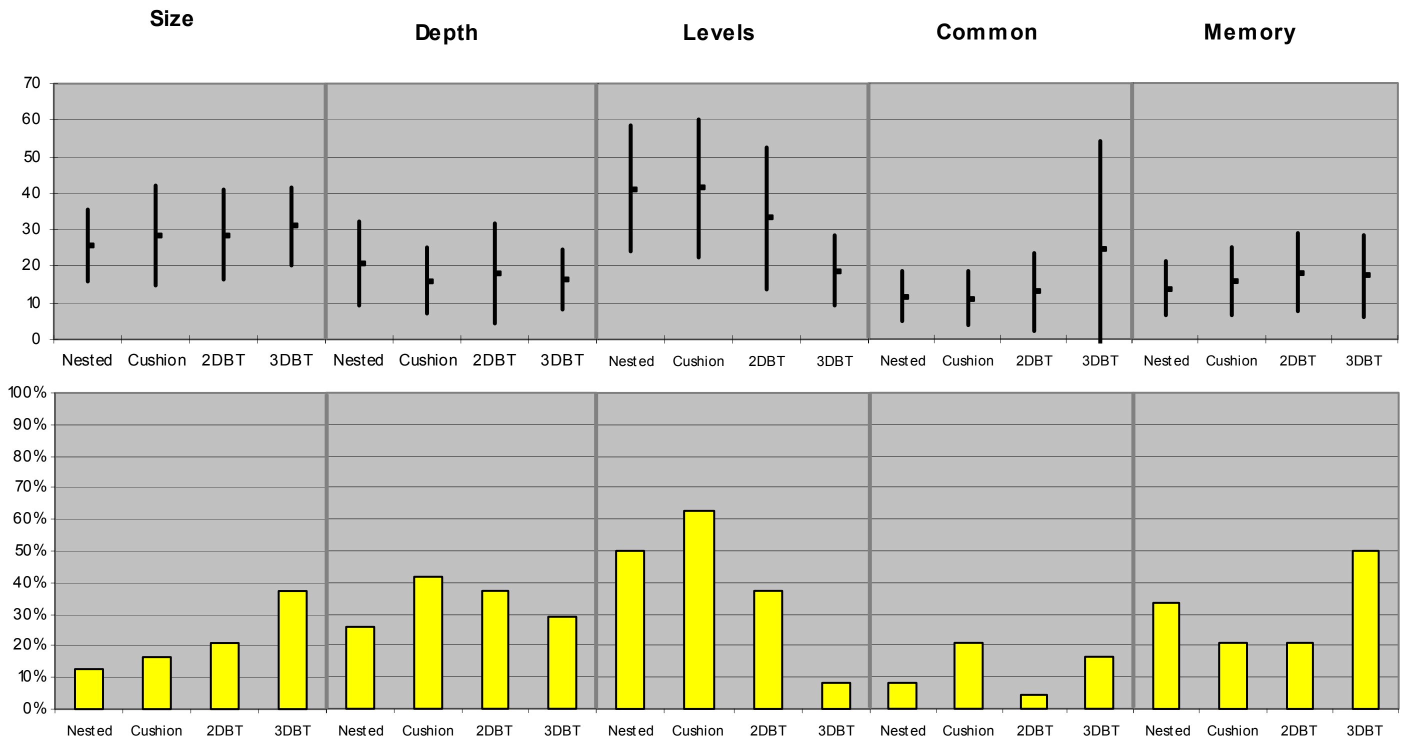

Figure 6. Average response time ± σ (top) and error rate (bottom) for each of the five tasks.

4.3 Results

Figure 6 shows the average response times in seconds and average error rate per visualization method for each of the five tasks. We observed the following results for each task:

Size: Determining the three largest nodes took slightly more time using beamtrees, which is understandable since they allocate less of the screen space to leaves, making small differences in size less obvious. The error rate for this task was generally (but not significantly) higher for beamtrees, supporting Cleveland [3], who stated that interpretation speed is negatively related to interpretation accuracy.

Depth: Interpreting the depth of a node did not show significant difference in response times. Error rates seem to indicate that nested treemaps and 3D beamtrees perform slightly better in this respect, since both of them dually encode depth information. Nested treemaps use both an alternating subdivision scheme and nesting, while 3D beamtrees make use of overlap and a third dimension to indicate depth. Cushion treemaps use both an alternating subdivision scheme and shading, but the latter seems to be too subtle to provide accurate depth information.

Levels: For global hierarchical information, such as the total number of levels, 3D beamtrees perform significantly ($p < 0.01$) better than both of the treemap methods in both time and accuracy. Most users simply rotated their view to the one depicted in figure 7d and counted the number of levels. 2D beamtrees performed somewhere in between, both in response time and accuracy. Most users got frustrated using the treemap methods to find the maximum depth, which accounts for the reasonable response times but low accuracy.

Common parent: Finding the common parent did not prove too difficult for three of the methods, although 3D beamtrees stand out with a higher average and a rather large standard deviation. This is due to the fact that some users took the wrong approach and started rotating the structure, looking at it from all sides and trying to discern where and if beams were connected. In fact, using the default top view proves much simpler, considering the response times for 2D beamtrees. We expect a definite learning factor here, so response times should improve as users become more familiar with beamtrees.

Memory: The results from the memory task were comparable, though there is a definite (but not statistically significant) difference between the accuracies for 2D and 3D beamtrees. This might be due to the difference between the flat shading and aliasing effects in the OpenGL rendering and the crisper 2D representation, but we are not sure.

Based on the results of this pilot study we expect 3D beamtrees to perform significantly better than regular treemaps when it comes to global structural information such as maximum depth or balance. Not only does the 3D view provide stronger depth cues, but when viewed from the side it provides views that are similar to a

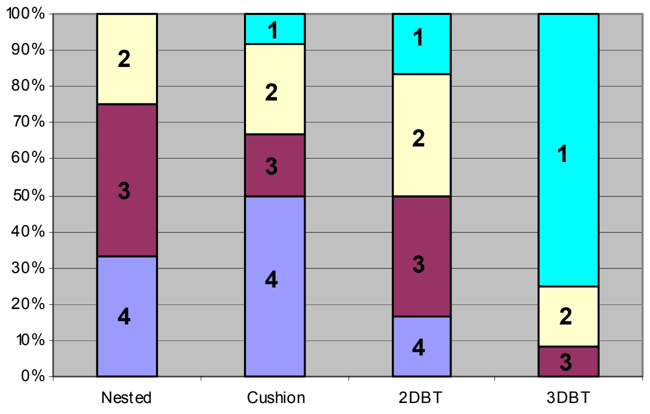

Figure 7. User preference rankings

conventional layered tree layout as well. Users also had a strong preference for 3D beamtrees, as is indicated by figure 7. This might be due to the fact that they were better able to answer depth related questions with the 3D version or the fact that 3D visualizations are generally found more interesting than 2D versions. Most users also felt they could perform better using beamtrees if they had more experience using the visualization.

5. Conclusions

We have presented a generalisation of the treemap algorithm. By scaling down individual nodes in a treemap we introduce occlusion as an extra depth cue, while retaining the size proportionality of treemaps. This allows us to display both hierarchical and size information in a single display. Both 2D and 3D variants were implemented and tested, with the 3D variant achieving significantly better results in determining the number of levels in the tree. Compared to other tree visualization methods that display size and structure simultaneously, like Icicle plots and tree rings [1], beamtrees offer higher scalability, up to thousands of nodes.

Although 3D visualizations generally introduce new problems such as occlusion and potentially difficult interaction, we don't expect these to be too much of a problem. In the default top-down view none of the leafnodes are overlapping, while interaction can be facilitated by providing meaningful preset viewpoints (i.e. top down and side-views). Adding more sophisticated interaction methods like highlighting the path to the root will also improve insight.

An advantage is that, contrary to regular treemaps, beamtrees actually display almost all internal nodes. One could argue that this is also the case with nested treemaps, but for interaction to be efficient here, nesting offsets would have to be around 5-10 pixels, taking up too much valuable screen space. Although we have only experimented with inspection, there might be some interesting research opportunities in interaction with beamtrees.

Beamtrees only work with slice and dice partitioning. Long thin rectangles already prevalent in this type of partitioning, suffer even more when scaled in their width. We partially avoided this problem by applying a treemap style subdivision to all leafnodes and simply discarding any remaining beams thinner than 2 pixels, but this is clearly not the most elegant solution. It is an open question whether work that has been done in the area of aspect ratio improvement is applicable to beamtrees.

In summary, we think the concept of beamtrees presents a valuable improvement over treemaps, displaying depth in a more natural way without compromising any of the advantages of conventional treemaps.

References

[1] T. Barlow and P. Neville, "A Comparison of 2-D Visualizations of Hierarchies", Proceedings of IEEE Symposium on Information Visualization 2001, *IEEE CS Press*, pp 131-138, 2001.

[2] D. M. Bruls, C. Huizing, J.J. van Wijk, "Squarified Treemaps", Proceedings of the joint Eurographics and IEEE TVCG Symposium on Visualization, 2000, *Springer*, pp 33-42.

[3] W. Cleveland, "Elements of Graphing Data", *Kluwer Academic Publishing*, New York, NY, 1994.

[4] Human Computer Interaction Lab, "Treemap 3.2", available from http://www.cs.umd.edu/hcil/treemap3.

[5] P. Irani, M. Tingley and C. Ware, "Using Perceptual Syntax to Enhance Semantic Content in Diagrams", IEEE Computer Graphics and Applications, September/October 2001, *IEEE CS Press*, pp 76-85.

[6] B. Johnson and B. Shneiderman, "Tree-Maps: A Space-Filling Approach to the Visualization of Hierarchical Information Structures", Proceedings of IEEE Visualization '91 Conference, *IEEE CS Press*, pp 284-291, 1991.

[7] B. Shneiderman and M. Wattenberg, "Ordered Treemap Layouts", Proceedings of IEEE Symposium on Information Visualization 2001, *IEEE CS Press*, pp 73-78.

[8] Technische Universiteit Eindhoven, "SequoiaView", available from http://www.win.tue.nl/sequoiaview.

[9] F.Vernier and L. Nigay, "Modifiable Treemaps Containing Variable Shaped Units", IEEE Information Visualization 2000 Extended Abstracts, available from http://iihm.imag.fr/publs/2000/Visu2K_Vernier.pdf.

[10] J.J. van Wijk and H.M.M. van de Wetering, Cushion Treemaps: "Visualization of Hierarchical Information", Proceedings of IEEE Symposium on Information Visualization 1999, *IEEE CS Press*, pp 73-78, 1999.