EUROGRAPHICS - IEEE VGTC Symposium on Visualization (2005), pp. 1–8
K. W. Brodlie, D. J. Duke, K. I. Joy (Editors)

Scalable, Robust Visualization of Very Large Trees

Dale Beermann [†] and Tamara Munzner[‡] and Greg Humphreys[†]

Abstract

*The TreeJuxtaposer system [MGT*03] allowed visual comparison of large trees with guaranteed visibility of landmarks and Focus+Context navigation. While that system allowed exploration and comparison of larger datasets than previous work, it was limited to a single tree of 775,000 nodes by a large memory footprint. In this paper, we describe the theoretical limitations to TreeJuxtaposer's architecture that severely restrict its scalability. We provide two scalable, robust solutions to these limitations: TJC and TJC-Q. TJC is a system that supports browsing trees up to 15 million nodes by exploiting leading-edge graphics hardware while TJC-Q allows browsing trees up to 5 million nodes on commodity platforms. Both of these systems use a fast new algorithm for drawing and culling and benefit from a complete redesign of all data structures for more efficient memory usage and reduced preprocessing time.*

Categories and Subject Descriptors (according to ACM CCS): I.3.6 [Computer Graphics]: Graphics data structures and data types

1. Introduction

Many domains require the manipulation and comprehension of complex hierarchical datasets. To address this need, many visualization systems have been developed that support interactive browsing of large trees [CN02, LRP95, Mun98, PGB02]. Despite the continuing progress in scalability, such as the recent TreeJuxtaposer system that handles up to 775,000 nodes [MGT*03], dataset size has grown as well, and many important datasets outstrip our ability to explore them interactively. For example, reconstructing the ancestral relationships between all known species on Earth, a tree estimated to contain at least 10 million leaves, is a current grand challenge for evolutionary biologists [Pen03]. Genealogical trees contain hundreds of millions of human names [Anc04]. The ability to explore the huge proof trees traversed by automated theorem provers could help people steer verification systems out of currently intractable computational bottlenecks [TBK92].

In this paper, we present two systems: TJC, which allows interactive browsing of trees of up to 15 million nodes on recent graphics hardware, and TJC-Q, for browsing trees up to 5 million nodes on commodity machines. Both support the **accordion drawing** technique, recently developed in the TreeJuxtaposer (TJ) application [MGT*03].

Accordion drawing has two key characteristics: a stretch-and-squish navigation metaphor, and guaranteed visibility. This requirement that marked objects be visible at all times, even when they fall into highly compressed screen-space regions smaller than one pixel, cannot be met by straightforward drawing, culling, and picking algorithms. The benefit of this guarantee is drastically reduced navigation time for tasks where it is possible to accurately mark regions of interest, as discussed previously [MGT*03].

TJ was originally designed to facilitate both browsing trees to build a mental model of their topological structure, and structural comparison between two or more trees. In this work, we focus only on the task of browsing. We do so for much larger datasets than could be handled previously by introducing more robust and scalable algorithms.

1.1. Challenges

We address four key challenges in the interactive exploration of very large hierarchical datasets:

Memory Footprint We limit ourselves to a process size of 2GB in order to use standard operating system memory management facilities. Fitting a tree of 15 million nodes into 2GB requires using an average of only 143 bytes per node.

[†] University of Virginia, {humper,beermann}@gmail.com
[‡] University of British Columbia, tmm@cs.ubc.ca

Of course, simply browsing the topological structure of a tree would be of little use without also drawing textual labels for a significant fraction of the nodes, so storage for those strings is included in our per-node memory budget.

Pre-processing Time The need for minimal startup time is an important aspect of software usability. At startup, we must not only parse the entire tree, but also carry out any pre-processing steps of spatial layout and data structure creation.

Drawing and Culling An ideal drawing algorithm would render only the visible parts of the scene even when faced with high depth complexity, while guaranteeing visibility of any marked areas even if they are shrunk to subpixel size. If the guaranteed visibility constraint is relaxed, then straightforward approaches to drawing and culling, as in [vWvdW99], would suffice and the amount of work to be done would be proportional to the size of the screen. However, ensuring mark visibility makes these problems more difficult. The quadtree-based TJ algorithm works for trees of a few hundred thousand nodes, but the culling is incorrect for trees of millions of nodes, and gaps may appear in some places where edges should be visible. Their algorithm also incurs large performance penalties by carrying out excessive overdrawing and maintaining a sorted priority queue of items to be drawn.

Edge Picking Using a mouse to select edges in a scene should incur as little computational overhead as possible. Mouse-based picking presents a sampling problem: the mouse location is given as integer pixel coordinates, but the true resolution of the structure being drawn can be much more fine-grained. This mismatch is particularly severe when considering our nonlinear distortion-based approach to navigation. A robust system should allow the user to pick any geometric item that is drawn.

1.2. Contributions

TJC and TJC-Q provide scalable accordion drawing with new data structures that use far less memory than those of TJ, improved algorithms to drastically increase drawing speed, and a completely new approach to picking that uses cutting-edge graphics hardware to provide pixel-accurate picking while saving both time and memory.

Redesigned Architecture We have created a scalable system through careful redesign. Every data structure has been carefully considered and either eliminated or redesigned to reduce both memory footprint and startup cost. Our architecture can handle trees of 5 million nodes on commodity platforms, and 15 million nodes on systems with leading-edge graphics hardware.

Edge Drawing We present a new unified algorithm for drawing and culling edges. This method ensures that no visible gaps appear, avoids most overdrawing, guarantees the visibility of marked areas, and is fast enough to draw the entire scene in less than a second for any tree size on a workstation-resolution display.

Edge Picking We present a new robust picking algorithm that exploits modern graphics hardware, allowing us to replace large data structures with lightweight ones that can be created quickly. We solve the aforementioned sampling mismatch by using multiple rendering targets to determine the edge that is the current focus of user interaction.

2. Previous Work

TJC and TJC-Q were inspired by the TreeJuxtaposer (TJ) system and its accordion drawing technique [MGT*03]. TJ was written in Java, whereas we have chosen to work in C++. Although the language switch does reduce our memory requirements somewhat, the primary reductions come from our new data structures and architecture. The switch does affect processing time, especially the time to parse a file.

Many tree drawing systems have been presented in a variety of application domains. The recent survey of Herman *et al.* discusses over one hundred systems for interacting with visual representations of trees and graphs [HMM00], but few are scalable. The TreeMap approach to visualizing large hierarchies has recently been scaled up to trees of one million nodes [FP02], but their space-filling approach is suitable for exploring attribute values of the nodes rather than topological structure.

Many application-specific tree drawing systems are useful for only small datasets because the only navigational controls are panning and rigid zooming. PhyloDraw is one of many examples in the application domain of phylogeny [CJKC00]. The TreeWiz system for exploring phylogenetic trees can accomodate trees of 50,000 nodes [RBB02] by aggregating subtrees into supernodes to avoid visual clutter, but the exploration method is extremely disorienting: clicking on an aggregated node spawns a new window showing its subtree.

In these approaches the screen area required to lay out a tree grows exponentially as its depth increases, so details are too small to comprehend when looking at an overview of the entire tree, and panning around the tree after zooming in to see details can be extremely disorienting. The idea of using surrounding context to help people stay oriented when investigating the details of large datasets has been extensively explored in the information visualization literature under the name Focus+Context [LRP95]. Among the most scalable were systems that used the mathematics of hyperbolic geometry for a fisheye-like effect, with trees of around 10,000 nodes in the 2D case and 100,000 nodes in the 3D case [Mun98]. Although these systems were effective for browsing large local regions of trees, users still lost track of their global location in very large trees.

TJ addresses this problem with the global Focus+Context

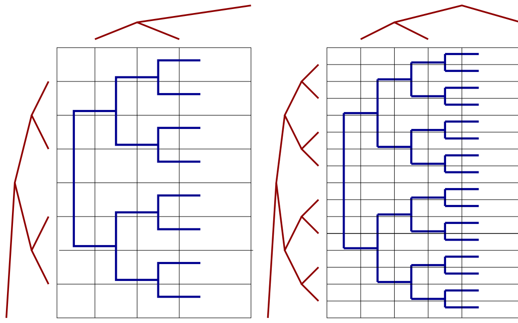

Figure 1: *TreeJuxtaposer Quadtree Structure. The bottom level of the quadtree defines a grid with vertical lines between each level of the topological tree and horizontal lines between each leaf. As the size of the tree grows, the cells become much wider than they are tall. For roughly balanced trees, increasing the size of the tree by one level halves the height of a cell while the width is made only slightly smaller.*

navigation metaphor of a stretchable rubber sheet with its borders tacked down, as named by Sarkar *et al.* [SSTR93]. The accordion drawing technique proposed by TJ combines this rubber sheet metaphor with guaranteed visibility of landmark regions. SpaceTree [PGB02] and Degree-Of-Interest Trees [CN02] are Focus+Context tree visualization systems that use aggregate glyphs to show surrounding context for the data, whereas the opposite accordion drawing approach strives for the highest possible information density. This choice imposes significant challenges, requiring aggressive culling to achieve realtime rendering. However, the lightweight interaction of simply moving the mouse inside the window allows users to quickly explore a large fraction of their datasets without the overhead of explicit navigation. The stretch-and-squish approach allows for extremely fluid navigation when the user does choose to do so.

3. Quadtree Limitations in TreeJuxtaposer

TJ relies heavily on a quadtree to support accordion drawing. The quadtree serves four main purposes: placing nodes in screen space, culling to terminate recursive drawing, drawing edges in sorted order, and spatial subdivision for picking. The use of a quadtree has several drawbacks when trying to work with trees containing several million nodes. In this section, we discuss the quadtree used by TJ and its limitations, motivating the improvements we present in later sections. In overcoming these limitations, we were able to remove the quadtree entirely, allowing us to meet our memory requirements.

3.1. Placing Nodes

We use the term `GridCell` for nodes in the quadtree, to distinguish them from the `TreeNodes` in the topological tree. In TJ, the `GridCells` at the bottom level of the quadtree form a grid in which to place `TreeNodes`. Munzner *et al.* [MGT*03] describe how to extend a standard quadtree to support distortion-based navigation by encoding the position of lines in the grid relative to each other in a hierarchical structure. Both lookup and update of the absolute position of a line in space can be done in logarithmic time relative to the number of leaves in the topological tree, due to the hierarchy shown in Figure 1. The user can stretch or shrink the quadtree by dragging lines of the grid closer together or further apart. Any interaction therefore creates a global deformation of the tree.

3.2. Culling

After TJ's quadtree is created, `TreeNodes` are attached to `GridCells` to create a relationship between the `TreeNode` and its screen-space extent. The bounding box for that node is defined by its horizontal edge (connecting the node to its parent), and its vertical edge (connecting the node to its children). A node is attached to the lowest `GridCell` in the quadtree that completely encloses it. The bounding box of a `GridCell` thus provides an upper bound on the screen-space extent of all edges attached to it and to its children. TJ uses the `GridCell` size as a heuristic to determine whether or not edges attached to a `GridCell` should be visible and should therefore be drawn.

When rendering very large trees, basing the culling decision on `GridCell` properties is not a scalable solution for two reasons. First, the correspondence between the spatial extents of a `GridCell` and its attached edges becomes more distorted as the size of the tree increases. As the number of leaves in the tree increases and the tree stays roughly balanced, the `GridCells` become much wider than they are tall (Figure 1). When this happens, the `GridCell` is a poor approximation of the true bounding box, and testing whether a `GridCell`'s area or height is less than a pixel is an incorrect termination condition for the recursive drawing procedure, resulting in visible gaps in the rendering.

Even if we compute the exact bounding box of the edges attached to a `GridCell`, a second problem remains. The spatial extent of those edges is different than the spatial extent of the subtrees formed by their descendants in the topological tree. The latter extent is needed for a correct termination test because the subtree can fall outside the spatial region represented by the `GridCell`. In this case, culling the edges attached to a `GridCell` would also result in incorrectly culling edges outside that `GridCell`, due to the drawing algorithm discussed in the next section.

3.3. Drawing

Drawing in TJ is handled with a progressive rendering approach. The core assumption is that the number of geometric items to draw in the scene will often outstrip the capa-

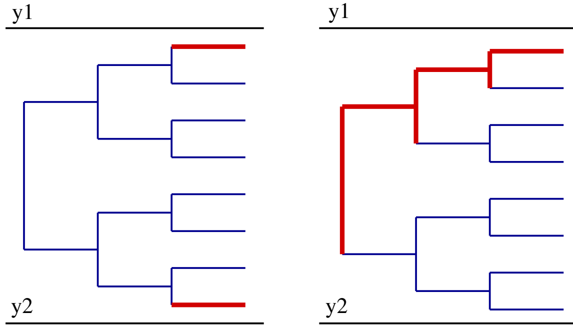

Figure 2: *Subtree Culling.* ***Left:*** *For any given subtree, we check the top-most and bottom-most leaves (marked in red) to determine if they map to the same pixel.* ***Right:*** *If so, then the entire subtree maps to the same row of pixels, with y-coordinate y2. We can then draw only the edges between the root of the subtree and the top-most leaf, marked in red. This flattened path maps to a single straight line on the screen.*

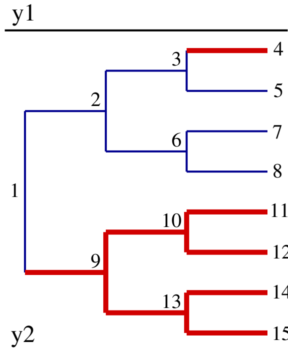

Figure 3: *Guaranteeing the visibility of marked edges in a small subtree. For the subtree starting at node 1, we must continue descending into the tree until we find a subtree that is either homogeneous, or either the top or bottom flattened path of all edges from the root to the top-most leaf is completely marked. For the subtree starting at node 9, we see that all edges along the line from the root to the top-most leaf are marked, and we only need to draw the flattened path of edges for nodes 9, 10, and 11.*

bilities of the graphics pipeline. Therefore, drawing items in order of "importance" would allow both interactivity and a good view as the scene is gradually filled in. Items are drawn by popping a `GridCell` off a queue and drawing all of its attached edges. Those edges' parents' and children's enclosing `GridCells` are then enqueued. In order to avoid confusing intermediate views, they sort the queue according to local topological coherence in order to maximize the context around the starting points. The queue is seeded with the `GridCell` in which interaction last occurred, and one `GridCell` from each active set of landmarks (which are guaranteed to be drawn).

The queue of `GridCells` is implemented using the Java `TreeSet` data structure, a sorted, balanced, binary tree that supports logarithmic insertion and deletion. The number of nodes in the queue is frequently a major fraction of all visible nodes, so the computational overhead of this approach is considerable. This overhead causes performance to be limited by the bookkeeping task of maintaining the queue, and yields graphics performance two orders of magnitude less than the raw capabilities of the graphics accelerator.

3.4. Picking

Quadtrees provide support for picking objects in the scene by using a logarithmic-time recursive traversal from the node to any leaf. Although the resolution of the quadtree subdivision is bounded only by floating-point precision, mouse events are reported in integer screen coordinates. This mismatch causes a serious problem for sufficiently large trees, where a `GridCell` can only be reached through a mouse event if the `GridCell` is larger than a pixel in both dimensions, or it straddles the border of two pixels.

In dense areas of the tree, there is a pickable edge at every mouse position, so the fact that some edges are unpickable is not usually a problem. However, in sparse areas of the tree, a single edge might be plainly visible but unpickable. Although expanding that region is one approach to making the desired edge pickable, requiring such navigation could interfere with the user's exploratory intent.

4. Combining Drawing and Culling

We present a new drawing and culling algorithm that eliminates all visible gaps, avoids most overdrawing, guarantees the visibility of marked areas, and is fast enough to draw the visible parts of a 15-million-node dataset in less than one second.

Our method avoids the use of heuristics by exploiting the relationship between a subtree's structure and its pixel coordinates. For any subtree, we find the current absolute location of its two boundary leaves (Figure 2). If these two leaves map to the same pixel in screen space, then the entire subtree will map to a horizontal row of pixels. We can then draw only the single path from the subtree's root to its topmost leaf.

This approach still supports TreeJuxtaposer's notion of guaranteed visibility. We can check for marks in a subtree by comparing the range of nodes in the subtree to a list of marked ranges in the entire tree (Figure 3). If we use node indices from either a preorder or a postorder traversal of the tree, then subtree ranges are exact [MGT*03]. When we find a flattened path from the root to either the top or bottom leaf

we can safely cull the rest of the subtree if it is entirely either marked or unmarked. If there is a mix of marked and unmarked nodes in the subtree we must continue our recursive descent into the subtree. The pathological worst case of a tree where all the leaves (but none of the interior nodes) are marked does not arise in practice because users typically mark entire subtrees.

Both drawing and culling are now based entirely on tree topology, and do not depend on any properties of `GridCells`. In addition, there is now little benefit to drawing items in order of "importance", because our drawing algorithm requires a nearly constant amount of work after the dataset size surpass a particular point (Table 1). We eliminate the extreme performance penalty of updating a sorted queue of items to be drawn, instead using a simple FIFO. The time to draw large trees is now determined entirely by the size of the screen.

5. Robust Picking in Hardware

As we discussed in section 3.4, TreeJuxtaposer's quadtree-based picking method makes some visible objects unpickable. We instead handle picking by exploiting leading-edge graphics hardware capabilities, eliminating the need for quadtree traversal. Recent graphics hardware supports an extension to OpenGL allowing multiple render targets [Arc02]. This extension gives us the ability to draw encoded edge identifier information into an offscreen buffer at the same time that we draw color into the draw buffer.

The edge identifier is specified using OpenGL's secondary color state, and a simple fragment program directs the edge color into the framebuffer for display, and the identifier into the offscreen picking buffer. When we need to know what edge a mouse event refers to, we simply read back a pixel from this buffer and use the results to find the chosen edge. Because we only read a single pixel, this picking method is extremely fast. It also eliminates the need for the quadtree, giving dramatic memory savings. Currently this technique supports trees of up to 16 million nodes because of the 24-bit precision of OpenGL's secondary color. However, it would be trivial to use the 32-bit primary color to encode the edge identifier, and use the 24-bit secondary color to draw the edges in the framebuffer.

6. Replacing Quadtrees with Grids

The approaches described above for fast and accurate culling, drawing, and picking eliminate the need for a quadtree. However, we still need to support the stretchable navigation of accordion drawing. We employ a lightweight grid data structure where horizontal and vertical lines are decoupled from each other and stored separately. Our grid structure is very similar to the O-buffer, extended to support our navigational needs [QK04]. Figure 4 shows that we

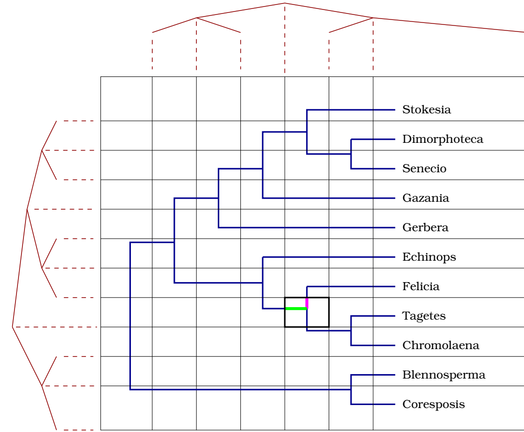

Figure 4: *Grid Structure. We replace the quadtree with a lightweight hierarchical grid that still allows stretchable navigation, but with decoupled horizontal and vertical lines. Each line is represented by a split value with respect to a set of two parent lines, shown by maroon annotation lines. Each node has a row index for a horizontal reference line plus a vertical offset from it (magenta), plus a column index for a vertical reference line plus a horizontal offset (green).*

still use a hierarchical approach of storing the relative distance where a child line forms a split between two parent lines. Every TreeNode in the topological tree stores a row index and vertical offset from the split line for that row, and similarly a column index and horizontal offset. The separate arrays of p vertical lines and q horizontal lines leads to an $O(p+q)$ memory cost for our grid, which is far cheaper than the $O(p*q*log(p*q))$ upper bound for a quadtree. Moreover, building this data structure is fast enough that the time it takes to build it is greatly outweighed by the time to parse the file, as shown in Figure 6.

7. Low-Memory Quadtrees

In Section 5 we presented an algorithm that supports picking through graphics hardware rather than quadtree traversal, but that approach depends on cutting-edge graphics hardware. Many potential users of our tree visualization system will be using commodity platforms. We present TJC-Q, a second system that runs on commodity hardware and uses quadtrees with a drastically reduced memory footprint. TJC-Q can handle trees of 5 million nodes, a factor of three less than the 15 million nodes of the quadtree-free TJC.

The critical factor in reducing memory consumption is reducing the per-object memory cost as our data structures contain several million objects. We use inheritance as one method to decrease memory usage, creating separate leaf cell and interior cell classes derived from a common parent. The three major types of data that GridCells contain are

structural pointers, edge attachments, and positional splits. All GridCells need a structural pointer to the parent GridCell. Interior cells require four structural pointers to child GridCells, but leaf cells do not.

Leaf cells also do not require any edge pointers: edges can only be attached in the interior because they always span at least two leaf GridCells. For interior cells, we store edge pointers in a resizeable vector because the number of attached edges cannot be know *a priori*. The STL [Sil94] vector class has a slightly higher static overhead than we would like, and their policy of growing vectors by doubling in size greatly increases the required memory. We use a simple vector class, requiring 8 bytes to store the index and array size, and a growth policy that states that array size is incremented by a constant value of 4 items. This policy guarantees a maximum of 12 unused bytes per vector in the worst case of 3 unused pointers. The number 4 was selected after experiments showed it to be a good compromise between increased running time from memory copying and increased memory size from unused items.

The third kind of GridCell data is positional information pertaining to the split lines that form cell boundaries. In TreeJuxtaposer, each GridCell stores an index to the arrays holding the relative split values in both the vertical and horizontal directions, and the computed absolute position of the lines that form its four boundaries and two splits in both world and screen coordinates. First, leaf cells do not need to store split information. More importantly, we observe that the same position information is redundantly stored when we attach it to a GridCell rather than the line in question. Although the combination of these twelve absolute positions is unique for each GridCell, any one of these values is shared by multiple GridCells and thus computed multiple times and cached in multiple places. For instance, the same line would correspond to the bottom of every GridCell in a row, the top of every GridCell in the next row, and also the split or top or bottom of many other GridCells at different quadtree levels.

The lightweight line-based grid described in the previous section is exactly the right data structure to store this absolute position information. In TJC-Q each GridCell only stores a pointer to the vertical and horizontal grid lines that form the two splits. This change brings three benefits; first, the obvious per-object memory reduction. Second, updating the node position between frames is much faster because updated grid line values are shared between multiple GridCells rather than being updated for each one. Finally, these indices can also be initialized dynamically as we create the quadtree, removing the need for a post-process traversal of the quadtree data structure.

8. Results

We have addressed the four main challenges in drawing extremely large trees: minimizing memory use, minimizing startup time, correct and efficient drawing, and correct picking. We present two results: the TJC system that exploits leading-edge graphics hardware to handle trees of 15 million nodes, and the TJC-Q system that handles trees of 5 million nodes on commodity hardware. These systems allow users to browse trees much larger than the previous TJ limit of 775,000 nodes, giving dataset size improvements of up to 1.2 orders of magnitude. We also note that the numbers published in the original TJ paper for a single tree used a Java heap of 1100MB, allowing for a maximum tree size of 550,000 nodes. Increasing the heap size allowed us to view a slightly larger tree. All results in this paper are from an Athlon 1800 with 2GB of memory and an ATI Radeon 9800.

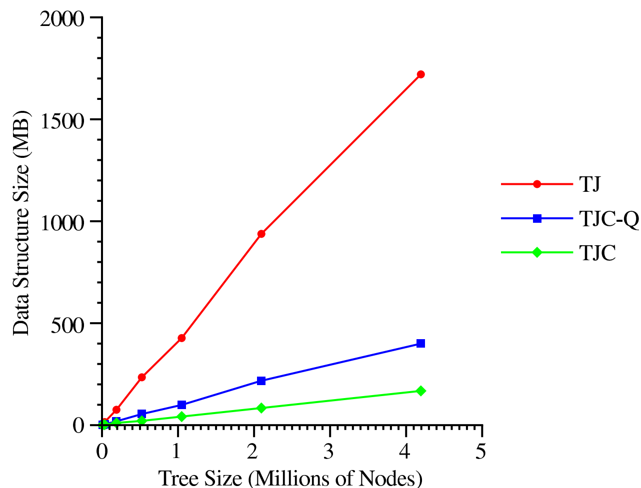

Figure 5: *Data Structure Sizes. We compare the theoretical memory requirements of the quadtree-based TreeJuxtaposer (TJ), the optimized quadtree of TJC-Q, and the lightweight grid of TJC. These theoretical numbers were generated by inspecting class fields rather than instrumenting a running executable, so as not to include overhead due to language differences. As expected, the required memory for each data structure grows linearly, but the constant factor is smaller with TJC-Q and much smaller with TJC. Note that these numbers do not include edge attachment information for reasons described in Section 8.*

Figure 5 compares the memory required for the three data structures in the quadtree-based TJ, the quadtree-lite TJC-Q, and the quadtree-free TJC. Both TJC and TJC-Q use a grid data structure whose size depends on the width p (number of leaves) plus the height q (number of levels) of the grid, rather than on n, the total number of nodes in the tree. Topological trees with n TreeNodes require a quadtree with approximately $2*n$ GridCells. TJ stores this grid information redundantly, requiring the larger memory cost. The TJC-Q quadtree avoids the extra cost for the grid and gains a large benefit from the architectural redesign stated in Section 7.

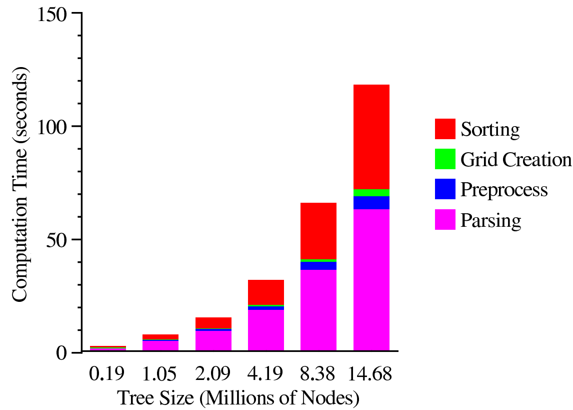

Figure 6: *TJC Startup Times. The bulk of startup time is spent on the unavoidable task of parsing the file. Preprocessing time refers to a tree traversal which initializes certain data members such as the row and column for any node. The sorting time is an optional addition.*

Nodes in Tree	w/ Cull	w/o Cull	Nodes Drawn
7161	0.030	0.037	5846
35346	0.106	0.145	23648
190265	0.216	0.746	51255
524287	0.092	1.566	40940
1048575	0.113	3.042	31522
2097151	0.140	6.077	37293
4194303	0.192	12.60	43538
8388607	0.239	24.66	50356
14680061	0.286	——	56137

Table 1: *Drawing Performance. We show two performance measures for our drawing and culling algorithm: the time to draw the scene with vs. without culling, and the number of nodes drawn (with culling). Our new method achieves a near-constant drawing time for any size tree. Note that for the largest tree, we are not able to enqueue all nodes in the tree as the size of the vector grows too large for the application to fit in main memory.*

Replacing the TJC-Q quadtree with the TJC grid allows us to handle datasets three times larger.

The quadtree numbers shown in Figure 5 are also quite conservative. We do not include edge attachment information, as the TJC-Q vectors for edge attachment are slightly different than TJ's, and TJ can only handle trees up to 775,000 nodes so this information would be an estimate anyway. Nonetheless, the figure still shows that we have succeeded in drastically reducing the per-object memory cost. TJ required 1.4GB of memory to handle 786,000 nodes, for a total per-node cost of 1912 bytes, whereas TJC can handle 14.68 million nodes with 1.7GB of memory, for a total per-node cost of 124 bytes.

Our second challenge was reducing the startup time required to view a dataset. For a tree of 512K nodes, TJ required a total of 14.1 seconds for parsing, and 63.1 seconds of preprocessing time. TJC offers an order of magnitude speedup, requiring 2.2 seconds for parsing and 1.4 seconds of preprocessing time. A large part of our preprocessing time is taken up by an optional sort, as shown in Figure 6, which allows the user to search for a particular node by name. If this capability is not required, we can build our data structures in a fraction of the parsing time. We use the standard engineering solution of Flex and Bison for parsing in order to concentrate on the research pieces of the problem.

One way to evaluate the success of our new drawing and culling algorithm is to consider the number of nodes drawn compared to the total number of nodes in the tree. As we discuss in Section 3.2, the scalability limits in the old method meant that to avoid spurious gaps in the image, every node in the tree had to be drawn. The effectiveness of our new algorithm in reducing that number is demonstrated in Table 1. We achieve a near-constant drawing time even as tree size increases because the number of nodes drawn grows very slowly compared to the size of the tree itself. The slow increase in the drawing occurs because some new nodes can be drawn in formerly sparse regions of the screen. The greatly improved drawing speed allows users to see much more global context while interacting with the tree.

9. Future Work

If we could improve rendering speed slightly, we could completely eliminate the rest of the progressive rendering overhead. Flushing the graphics pipeline is expensive, and its elimination would result in an even larger increase in render speed. We are still limited by the time it takes to update the grid hierarchy rather the time it takes to draw the tree.

We would also like to explore better methods for guaranteed visibility. Our current method does not perform well when many leaves deep in the tree are marked, as we need to continue to descend into the tree until we find them. This case does not seem to appear in practice, but nonetheless could be a potential problem. Although our current method is fast enough that even if the entire tree needs to be drawn the user will only have to wait for a short period of time; a method that eliminates rather than avoids overdrawing would be appealing.

10. Conclusion

We have identified the theoretical and practical limitations of using quadtree-based methods to draw large trees, and surmounted these restrictions with new algorithms. We present two new systems: TJC exploits leading-edge graphics hardware for picking, allowing us to completely eliminate the

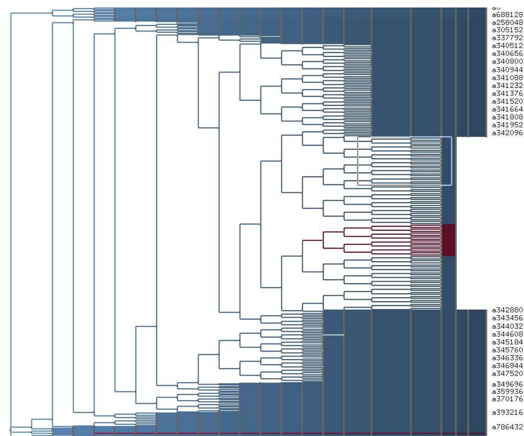

Figure 7: *Screenshot ofTJC during interaction, running on a tree of 14.6 million nodes with a window size of 1024x768.*

quadtree and scale to datasets of 15 million nodes, a size improvement of more than an order of magnitude beyond previous work. The TJC-Q system provides an order of magnitude increase on commodity platforms, because of our drastic reduction in the quadtree memory footprint. For trees of 15 million nodes, the TJC system can carry out all preprocessing within two minutes, and display the entire tree in less than one second.

Acknowledgements

We would like to thank Mark Segal at ATI for his extremely patient help in debugging our countless GPU problems and François Guimbretière for his early input and support.

References

[Anc04] ANCESTRY WORLD TREE:. `http://www.ancestry.com/trees`, cited April 9 2004. 1

[Arc02] ARCHITECTURAL REVIEW BOARD: *ARB ATI_draw_buffers Specification*, Dec 2002. 5

[CJKC00] CHOI J.-H., JUNG H.-Y., KIM H.-S., CHO H.-G.: PhyloDraw: A phylogenetic tree drawing system. *Bioinformatics 16*, 11 (2000), 1056–1058. 2

[CN02] CARD S. K., NATION D.: Degree-of-interest trees: A component of an attention-reactive user interface. In *Proc. Advanced Visual Interfaces (AVI)* (2002), pp. 231–245. 1, 3

[FP02] FEKETE J.-D., PLAISANT C.: Interactive information visualization of a million items. In *Proc. InfoVis* (2002), pp. 117–124. 2

[HMM00] HERMAN, MELANÇON G., MARSHALL M. S.: Graph visualization and navigation in information visualization: A survey. *IEEE Trans. Visualization and Computer Graphics 6*, 1 (2000), 24–43. 2

[LRP95] LAMPING J., RAO R., PIROLLI P.: A Focus+Content Technique Based on Hyperbolic Geometry for Viewing Large Hierarchies. In *Proc. CHI '95* (1995), pp. 401–408. 1, 2

[MGT*03] MUNZNER T., GUIMBRETIÈRE F., TASIRAN S., ZHANG L., ZHOU Y.: TreeJuxtaposer: Scalable tree comparison using Focus+Context with guaranteed visibility. *ACM Trans. Graph. (SIGGRAPH) 22*, 3 (2003), 453–462. 1, 2, 3, 4

[Mun98] MUNZNER T.: Drawing Large Graphs with H3Viewer and Site Manager. In *Proc. Graph Drawing, LNCS 1547* (1998), Springer-Verlag, pp. 384–393. 1, 2

[Pen03] PENNISI E.: Modernizing the tree of life. *Science 300*, 5626 (13 June 2003), 1692–1697. 1

[PGB02] PLAISANT C., GROSJEAN J., BEDERSON B.: SpaceTree: Design evolution of a node link tree browser. In *Proc. InfoVis* (2002), pp. 57–64. 1, 3

[QK04] QU H., KAUFMAN A. E.: O-buffer: A framework for sample-based graphics. *IEEE Trans. Vis. Comput. Graph. 10*, 4 (2004), 410–421. 5

[RBB02] ROST U., BORNBERG-BAUER E.: Treewiz: interactive exploration of huge trees. *Bioinformatics 18*, 1 (2002), 109–114. 2

[Sil94] SILICON GRAPHICS INC.: *Standard Template Library Programmer's Guide*, 1994. 6

[SSTR93] SARKAR M., SNIBBE S. S., TVERSKY O. J., REISS S. P.: Stretching the Rubber Sheet: A Metaphor for Viewing Large Layouts on Small Screens. In *Proc. User Interface Software and Technologies (UIST)* (1993), pp. 81–91. 3

[TBK92] THÉRY L., BERTOT Y., KAHN G.: Real theorem provers deserve real user-interfaces. In *Proc. 5th SIGSOFT Symp. on Software Development Environments* (1992). 1

[vWvdW99] VAN WIJK J. J., VAN DE WETERING H.: Cushion treemaps. In *Proc. InfoVis* (1999), pp. 73–78. 2