

PRISAD: A Partitioned Rendering Infrastructure for Scalable Accordion Drawing (Extended Version)

James Slack^{*†}

Kristian Hildebrand^{*†‡}

Tamara Munzner^{*†}

ABSTRACT

We present PRISAD, the first generic rendering infrastructure for information visualization applications that use the accordion drawing technique: rubber-sheet navigation with guaranteed visibility for marked areas of interest. Our new rendering algorithms are based on the partitioning of screen-space, which allows us to handle dense dataset regions correctly. The algorithms in previous work led to incorrect visual representations because of overculling, and to inefficiencies due to overdrawing multiple items in the same region. Our pixel-based drawing infrastructure guarantees correctness by eliminating overculling, and improves rendering performance with tight bounds on overdrawing.

PRITree and PRISeq are applications built on PRISAD, with the feature sets of TreeJuxtaposer and SequenceJuxtaposer, respectively. We describe our PRITree and PRISeq dataset traversal algorithms, which are used for efficient rendering, culling, and layout of datasets within the PRISAD framework. We also discuss PRITree node marking techniques, which offer order-of-magnitude improvements to both memory and time performance versus previous range storage and retrieval techniques. Our PRITree implementation features a five-fold increase in rendering speed for non-trivial tree structures, and also reduces memory requirements in some real-world datasets by up to eight times, so we are able to handle trees of several million nodes. PRISeq renders fifteen times faster and handles datasets twenty times larger than previous work. The software is available as open source from <http://olduvai.sourceforge.net>.

CR Categories: I.3.6 [Computer Graphics]: Methodology and Techniques—Graphics data structures and data types;

Keywords: Focus+Context, Information Visualization, Real Time Rendering, Progressive Rendering

INTRODUCTION

PRISAD, our Partitioned Rendering Infrastructure for Scalable Accordion Drawing, is a generic Accordion Drawing (AD) infrastructure for rendering and navigating large datasets. AD is a visualization technique that features rubber-sheet navigation and guaranteed visibility of selected nodes. Rubber-sheet navigation involves the user-guided action of stretching on-screen regions of interest; a stretched region has more screen real estate in which to draw more unoccluded geometric items from the same world-space region. When a region is stretched, the nailed-down borders of the window prevent data from being pushed off-screen and AD squishes data in appropriate regions, as shown in Figure 1.

Guaranteed visibility of data, represented by geometric objects on screen, is trivial with small datasets. The topological structure



Figure 1: **Left:** A tree dataset drawn with uniformly allocated space for each vertical node width and horizontal node height. **Right:** When navigating by stretching a rubber-sheet surface, the distortions allocate more screen-space to some regions of nodes and other regions are squished into less screen-space.

of the tree shown in Figure 1, and colors for each node, are visible without navigation. However, when the size of the dataset becomes large, as in Figure 2, AD must guarantee the visibility of all marked regions. A brute-force drawing algorithm, which would render every node in the dataset, does not offer sufficient rendering performance for animating such large datasets, especially with our guaranteed visibility requirements.

As data is never pushed off-screen with AD navigation, we can always map data from its infinite-precision world-space position to our finite-precision dataset representation in screen space. AD navigation leads to compressing regions of many data items to subdivide a small screen-space region, yielding high depth complexity. To achieve scalable rendering performance for large datasets, we must efficiently reduce the amount of overdrawing in dense screen-space regions where drawing a subset of geometric data objects is sufficient to represent the entire region. Culling the correct data in dense regions is particularly difficult when we must guarantee the visibility of important features at all times. A correct drawing with no overculling is visually indistinguishable from the brute-force rendering where every item is drawn. We need both marked node visibility, and a proper representation of the dataset in every distorted region of screen space.

We present our generic PRISAD infrastructure, and two applications built on it, both of which use Java with the GL4Java graphics library. PRITree implements the feature set of TreeJuxtaposer for visually comparing hierarchies [11], and PRISeq has the functionality of SequenceJuxtaposer for visualizing multiple aligned genomic sequences [16]. Our contributions include:

- **Time-Efficient Generic Rendering:** PRISAD tightly bounds overdrawing in dense, complex regions by separating pixel-based partitioning from application-specific rendering actions.
- **Correct Generic Rendering:** PRISAD eliminates overculling, so no misleading gaps appear in the dataset picture.
- **Space-Efficient Marking:** PRITree computes and stores marked regions of trees in structures capable of determining marking characteristics quickly, eliminating the need for caching marking properties for each node.
- **Space-Efficient Traversal:** PRITree traversal algorithms for drawing and picking exploit the dataset topology, instead of adding a memory-expensive external data structure.

^{*}e-mail:{jslack,hilde,tmm}@cs.ubc.ca

[†]University of British Columbia

[‡]Bauhaus University Weimar

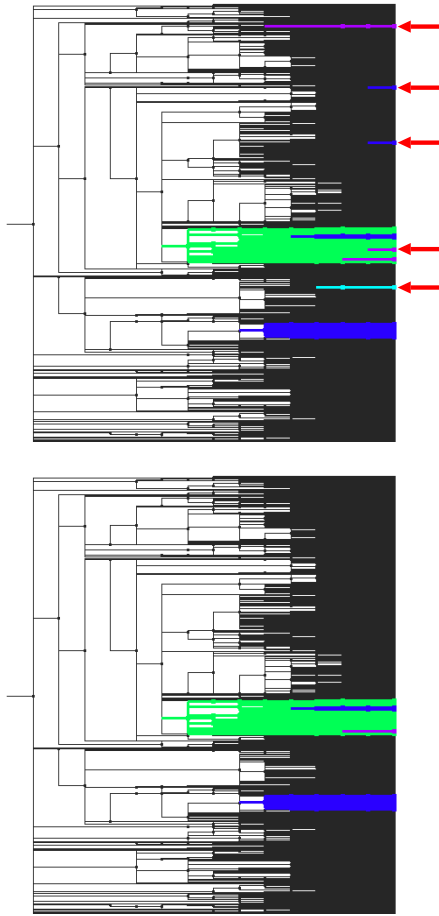


Figure 2: **Top:** For densely drawn regions of a dataset, we can mark several regions of interest with guaranteed visibility, and we always draw all marked regions that are smaller than a pixel. **Bottom:** In the identically marked tree without guaranteed visibility, these small regions, marked with red arrows above, may not be drawn.

- **Correct and Efficient Sequence Rendering:** PRISeq traversal algorithms efficiently aggregate columns to accurately reflect relative nucleotide proportionality.

This journal paper is an extended version of an InfoVis conference paper [15]. In addition to adding new material on picking algorithms, we have completely reworked and expanded the exposition of the sections on PRISAD, PRITree, PRISeq, and PRISAD performance.

In the next section, we give an overview of related work. In our PRISAD section, we present our generic approach to scalable accordion drawing. We cover PRITree and PRISeq separately in the next sections, and then evaluate their performance. Finally, we describe possible future work and conclusions. We also include an appendix, which contains supplementary details of our PRITree rendering techniques.

RELATED WORK

The TreeJuxtaposer [11] application introduced AD navigation with tree topologies and performed structural comparisons among a small set of tree datasets. TreeJuxtaposer includes fast tree comparison algorithms, which provide the primary bidirectional mapping between common tree structures. The mapping allows users to visually determine structure, and the application uses the mapping

results to highlight regions of structural difference. Since TreeJuxtaposer scales to tree datasets with many more nodes than the number of available on-screen pixels, highlighted regions would not necessarily be visible without adhering to our requirements for guaranteed visibility.

The AD infrastructure used by TreeJuxtaposer is optimized for rectilinear trees and is not capable of displaying datasets from other application domains. Also, the scalability of TreeJuxtaposer limits the maximum size of single tree datasets to 550,000 tree nodes, or comparisons of two 150,000 node trees [11]. DOITrees [8], for example, have been used to explore the directory structure of the Open Directory Project website [1], which contains more than 600,000 nodes. The rendering performance of large datasets becomes an issue with non-trivial topological structures; the TreeJuxtaposer results that benchmark performance with only balanced binary trees do not capture performance results with real-world datasets with high-degree nodes. We compare the performance of TreeJuxtaposer with PRITree in our performance section.

The TJC-Q and TJC applications [4] for AD tree browsing are a considerable improvement on the original TreeJuxtaposer system. The TJC-Q system, which runs on commodity hardware, handles up to 5 million nodes, commensurate with PRITree. The TJC system uses advanced graphics card features to handle up to 15 million nodes, which is three times the limit of PRITree. However, these systems do not support comparison between multiple trees, and their algorithms are hardwired to work only with trees and could not be easily adapted to a generic AD framework. They both use the same top-down rendering algorithm, where subtrees that subtend more than one pixel draw all of their children. This approach leads to poor performance because of overdrawing for datasets with high-degree nodes, and we note that it was only benchmarked for balanced binary trees.

SequenceJuxtaposer [16] is an AD application for the visualization of genomic sequences of up to 1.7 million nucleotides, using a quadtree-based AD infrastructure built on the algorithms used by TreeJuxtaposer. In contrast, standard Web-based genome browsers such as the Ensembl [9] and UCSC [10] systems show sequence data with jump cut transitions between different scales. In our performance section, we compare PRISeq, shown in Figure 3, with SequenceJuxtaposer.

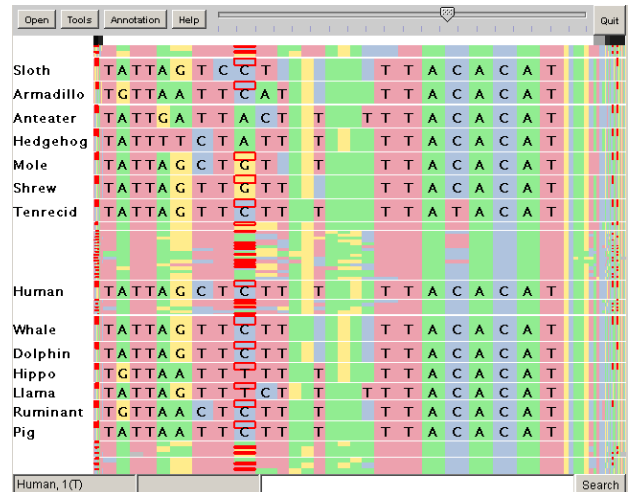


Figure 3: PRISeq is a genome sequence visualization application built on PRISAD with the feature set of SequenceJuxtaposer [16].

Slack discusses PRISAD and PRITree in detail in his thesis [14]. Few other information visualization systems can handle extremely large datasets. Fekete presents a system that can handle treemaps of

one million nodes [6]. While AD could in theory be implemented within an existing toolkit such as the InfoVis Toolkit [5], its focus on generality rather than scalable accordion drawing precludes achieving the performance we describe here. The Tulip system for graph drawing [2] is quite general and its data structures were carefully designed for scalability. However, it would be very difficult to adapt Tulip for general accordion drawing, especially due to our guaranteed visibility requirements for rendering. The Jazz and Piccolo zoomable user interface toolkits [3] also provide support for multi-scale navigation through arbitrarily large 2D surfaces, but not guaranteed visibility of landmarks or rubber-sheet navigation. NicheWorks [17], a graph visualization application that lays out nodes radially, is capable of displaying graphs of up to 50,000 nodes with real time manipulation, and its performance decreases linearly with dataset size. In contrast, PRISAD provides constant rendering performance for datasets.

PRISAD

PRISAD provides support for navigation, culling, drawing, picking, and marking. Applications must be designed to interact with the generic PRISAD infrastructure to benefit from its capabilities. The interplay of control flow between PRISAD-enabled applications and the components provided by this infrastructure is shown in Figure 4. We distinguish between a pre-processing discretization stage that operates entirely in world-space, and the rendering step that runs for each drawn frame where computations are handled in screen-space coordinates. PRISAD-enabled applications must support the following functions:

- laying out the dataset as a collection of geometric objects in world space
- gridding each geometric object between its four enclosing grid lines
- seeding the partitioned ranges for drawing in priority order
- drawing representative geometric object for each range, through selection or aggregation

The generic PRISAD components handle the remaining actions:

- initializing binary trees holding horizontal and vertical grid lines
- mapping between geometric objects and grid lines
- partitioning grid lines into adjacent ranges based on screen-space positions
- progressively controlling rendering for realtime performance

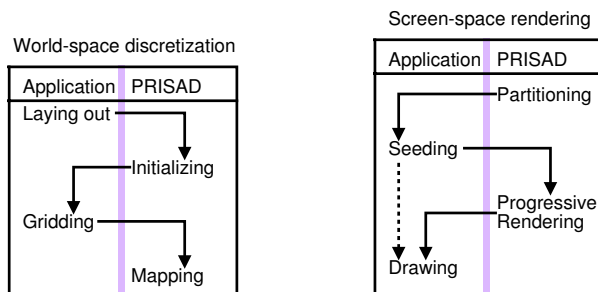


Figure 4: **Left:** Initialization of a dataset in PRISAD applications requires a world-space discretization phase, which must generate several generic components from application-specific dataset structures. **Right:** The rendering phase separates partitioning from drawing, which simplifies application drawing effort for faster pixel-based rendering performance.

Split Line Hierarchy

The link between discretization in world space and rendering in screen space is the grid of lines that keeps track of the stretching and squishing of navigation actions. Figure 1 shows the deformation of a small tree, with this malleable two-dimensional grid structure explicitly indicated as an overlay on the rendered picture of the tree. A **split line** is a dividing line of that 2D grid structure; split lines partition the space in which the geometric objects are drawn and are used to map world-space regions onto screen regions.

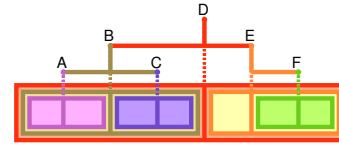


Figure 5: A split line hierarchy is both a binary tree structure that provides a linear ordering and a hierarchical subdivision of areas. For instance, the region for split line *B* is bounded by its parent region *D*, and *B* separates its bounded descendants *A* and *C*.

Figure 5 shows that a split line hierarchy provides both a linear ordering of the lines, and a recursive subdivision of spatial regions. Each split line may be moved independently in its region, and we use a relative offset for the position of a split line in its bounded region. Moving a split line affects the absolute, screen-space position of both the moving split line and all of its split line hierarchy descendants. All AD implementations achieve $O(\log n)$ performance for computing the absolute positions of split lines using similar hierarchies, when any position is required by the rendering algorithm. However, since we cache absolute positions of nodes, and only require absolute positions for $O(p)$ split lines, for p pixels on screen, the amortized per-frame cost of world-to-screen computation is also $O(p)$.

We use minimal memory overhead by decoupling the grid into separate horizontal and vertical split line hierarchies, as proposed by TJC [4]. In contrast, the original TreeJuxtaposer system uses a quadtree data structure for partitioning in both directions simultaneously, and the memory required to maintain that data structure is the primary limitation of its scalability.

World-space discretization

The pre-processing phase of discretization occurs in world space. The application lays out the dataset as a collection of geometric objects, and passes information about the size of split line hierarchy needed to contain it to PRISAD for grid initialization. The gridding phase finds the four bounding split lines that enclose each geometric object, and if needed PRISAD will record the bidirectional mapping between these split lines and geometric object in a lookup table. For each of the four world-space discretization steps, we refer to Figures 6 and 7 for illustrative examples in PRITree and PRISeq with a small dataset.

Laying out The spatial layout of a dataset; that is, the world-space position of the geometric objects that comprise the whole, is determined by the application. For instance, layout in PRITree, shown in Figure 6a, uses a standard horizontal rectilinear tree layout method. Edges are drawn with T-shaped lines and nodes are drawn as points at the junction of the T, with leaves right-aligned on the side of the screen. PRISeq positions pre-aligned genomic sequences in the vertical direction, shown in Figure 7a, displaying the nucleotides from left to right as color-coded boxes that represent the bases A, C, G, or T. The two applications presented here have non-overlapping layouts for geometric objects. Our generic PRISAD

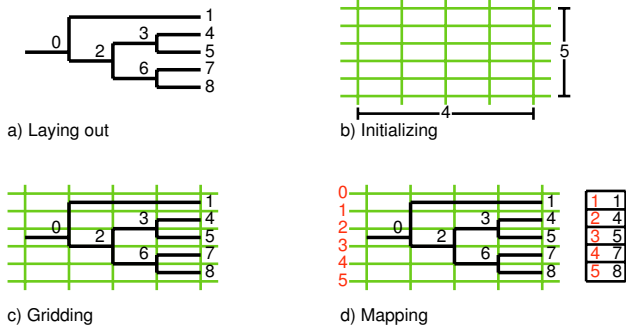


Figure 6: World-space discretization for trees. The map on the right of **d**) shows the association between split lines on the left and leaves on the right.

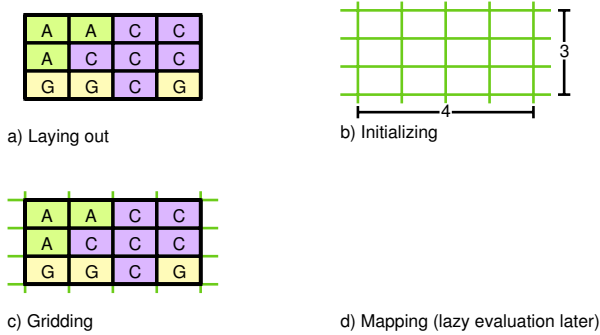


Figure 7: World-space discretization for sequences. Mapping is performed lazily as needed later, during rendering, to reduce the application startup time.

infrastructure could in theory handle object overlap, but that would add complexity to the application-specific drawing phase.

Initializing After layout, the application calculates how many split lines are required in each direction, to allow navigation at the appropriate granularity for the geometric structure. This calculation is straightforward for both PRITree and PRISeq, as shown in Figures 6b and 7b. For trees, we want one split line between each leaf in the vertical direction, and one between each layer from the root to the leaves horizontally. For sequences, we need one split line between each nucleotide box. The PRISAD infrastructure can create and initialize the two split line hierarchies after being supplied with the required sizes by the application.

Gridding In PRISAD, gridding is the specification of which four split lines enclose a world-space geometric object on the top, bottom, left, and right sides. In PRITree, the space required by each leaf edge is uniform in one direction, with one edge between each pair of equally-spaced split lines. However, horizontal edge lengths, and the vertical extent of the interior node edges, depend on the tree topology, as shown in Figure 6c. For PRISeq, in Figure 7c, the nucleotide boxes are all of uniform size in world space, so gridding is straightforward.

Mapping The final discretization step provides a constant-time bidirectional lookup function to map between the enclosing split-lines and the geometric objects. In PRISeq, there is no need to explicitly calculate or store extra information because of the regular grid structure inherent in the horizontal sequence rows and vertical columns of aligned nucleotides. However, we make use of the PRISAD mapping infrastructure when we perform the first scene rendering. Instead of mapping at initialization, which becomes slow for large datasets, we map as part of the application-specific draw-

ing stage. This means, as shown in Figure 7d, that no mapping is done in the PRISeq world-space discretization stages. Details about how we map aggregated columns of nucleotides are in the PRISeq section.

In PRITree, the layout is more complex, so the relationship between tree nodes and split lines must be explicitly recorded before the first rendering. Figure 6d shows the table stored by PRISAD that provides $O(1)$ access from a leaf node to the split line assigned to it, and vice versa from a split line to its attached leaf node. Interior nodes are not mapped to split lines, since screen-space rendering operations that require the bidirectional mapping operate only on the leaves of the dataset. This mapping allows for constant-time bidirectional lookup: leaves can be found near a given screen-space position, and likewise on-screen positions can be found given a leaf object from the topology.

Screen-space rendering

PRISAD rendering occurs in screen space, and again the control flow bounces between the infrastructure and the application. First, partitioning an entire split line hierarchy creates a list of ranges that cover small screen-space areas of roughly equal size. Seeding then allows the application to impose an order of drawing by turning the range list into a priority queue. The infrastructure has optional support for progressively rendering the prioritized queue, checking for interaction or animation events at regular intervals. Finally, the application is responsible for determining a single geometric object to draw for each range in the queue. Figure 8 shows a small PRITree example.

All previous AD infrastructures, which are tightly coupled to application-specific algorithms, perform partitioning during drawing using a top-down approach. They begin ordered rendering by enqueueing a single root object, and recursively enqueue its descendants until some stopping criteria are satisfied. Determining whether it is safe to terminate the recursion requires complex application-specific calculations, in particular because of the guaranteed visibility requirement. Generalizing this top-down hierarchical approach at the infrastructure level would be difficult, even for applications with highly regular structure such as the grid-based layout of aligned sequence data. The key innovation of PRISAD is separating screen-space partitioning, which can be handled generically, from the drawing that must be done by the application. Our application-specific drawing algorithms are simple, are executed a bounded number of times linear in the number of partitions, and do not require computation of screen-space positions to guarantee coverage of specific pixels.

We note that because accordion drawing is explicitly based on discretization, the classical topological definition of rubber-sheet geometry as a homeomorphic transformation does not hold [13]. A homeomorphism is a bijective, continuous function with a continuous inverse, whereas the discretization that we carry out in order to efficiently handle large datasets is of course not continuous. We nevertheless use the term rubber-sheet navigation in describing AD because it captures the feel of the interface.

Partitioning The main idea of PRISAD is to partition the dataset into screen-space regions of roughly equal size before drawing any geometric objects. This partition computation involves a binary search traversal of the split line hierarchy. It relies on the screen-space positions of the grid lines, and thus must be recomputed each time that any navigation action occurs. After partitioning, the resulting screen-space regions are either smaller than a target size, or contain only one geometric object to draw. Each region is bounded by split lines, so partitioning returns a list of split line ranges. Figure 8a shows the PRITree partitioning of the leaf set $\{1, 2, 3, 4, 5\}$ into the queue of ranges, $\{[1, 2], [3, 4], [5]\}$. The segmentation is based only on the position of the tree leaves with respect to vertical

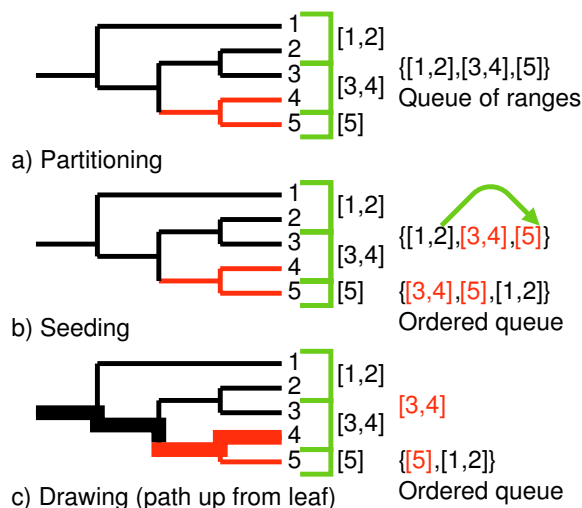


Figure 8: Screen-space rendering for trees. The lines to the right of the tree demarcate screen-space regions, and navigation will change which objects fall into them.

screen-space regions. This example illustrates that the partitioning does not need to match the hierarchical structure of the topological tree: the subtree with leaves 4 and 5 is split across multiple ranges, and the $[3, 4]$ split line range contains leaves from multiple topological subtrees.

An application developer must determine which of the two hierarchy directions to partition for the rendering phase. With PRITree, we observe that the dense structure of topological leaves in the vertical direction is ideal for culling, whereas the horizontal direction lacks uniform, traversable structure; thus, we partition so that the primary rendering direction is horizontal. In contrast, for PRISeq the primary rendering direction is vertical. Many nucleotides in a column are expected to be identical, because the rows of multiple gene sequences are aligned. We exploit this property to save time and space by run-length encoding in the vertical direction, as described in the PRISeq section.

Seeding and Progressive Rendering The output from the partitioning stage is a list of ranges. The seeding stage allows applications to transform that list into a queue, specifying the order in which to draw items when progressive rendering support is enabled. With datasets small enough to render quickly, the entire scene can be drawn in a single frame and drawing order is irrelevant, applications can disable progressive rendering. The dotted line in Figure 4 represents this pass-through case. However, the PRISAD infrastructure offers support for guaranteed frame rate rendering to ensure that each frame finishes within a bounded amount of time, with rendering spread across multiple frames. In this case, drawing order is visible to the user and the application can impose its own semantics. For example, to ensure visibility of landmarks during animated transitions of datasets, we render a representative object for each marked region first in PRITree and PRISeq, and we also move objects selected by the user for resizing to the front of the render queue. For example, if the subtree containing the leaves $[4, 5]$ is marked as in Figure 8b, we would reorder the partition queue P as $\{[3, 4], [5], [1, 2]\}$ since the marked leaves of 4 and 5 should be drawn before the other unmarked leaves.

Even if progressive rendering is unnecessary, seeding is still required to ensure that drawings are correct to avoid overculling marked regions. Seeding prevents the rendering errors shown in Figure 2. Our sections on marking for PRITree and PRISeq describe how marked areas are enqueued in the PRITree and PRISeq

seeding phases, respectively, to ensure guaranteed visibility.

Drawing In the drawing phase, one geometric object from each enqueued object range is drawn. For trees, one leaf node is selected from each range, and the full or partial path from the leaf up towards the root is drawn. Figure 8c shows the effect of drawing from leaf 4 to the root as a thick line along the path. For sequences, aggregation in the horizontal directions occurs as needed to create a representative box for a range, and the entire column is drawn with the minimal number of boxes using run-length encoding. The following sections describe application-specific drawing approaches in more detail.

PRITREE

In the previous section, we discussed the generic infrastructure for world-space discretization and screen-space rendering, including examples of PRITree. In this section, we discuss more details of tree traversal for rendering, creation and traversal of data structures for guaranteed visibility of marked groups, and support for efficiently picking geometric objects near the cursor.

Rendering: Leaf Selection

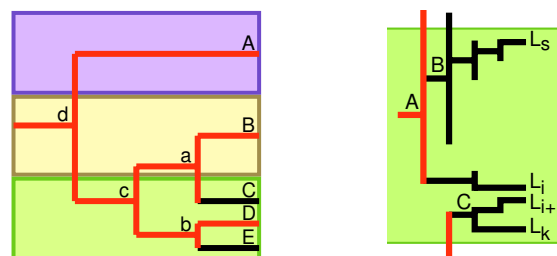


Figure 9: **Left:** Each partitioned range of leaves in $P = \{[A], [B], [C, E]\}$ must render only one path from some leaf in its range to the root; we only draw tree edges marked in dark grey and always render the leaf paths in ranges with a single leaf. Sub-pixel partitions are shown as alternating colored regions. When deciding on a leaf in $[C, E]$, we must choose either D or E , or else the interior node b would not be rendered. **Right:** Our selection traversal processes paths from the shaded partition to all subtrees with leaves in that range larger than τ . The black edges represent traversal paths and dark grey edges stop the traversal from processing subtrees of extent larger than τ .

In the drawing phase, the application is given a split line range, and must determine which geometric object to draw from the set of objects attached to those grid lines. For trees, the choice is which leaf to select, and the path from that leaf up to the root is drawn. The path drawing can safely terminate early when a path segment that has already been drawn is encountered. The selection of the leaf for each range is the most important run-time decision for our drawing algorithm. A poor leaf choice would lead to incorrect overculling, where a misleading gap appears in the drawing. Figure 9 Left shows three ranges, with the selected leaves and their upward paths drawn in dark grey and culled objects in black. In this example, note that selecting leaf C could lead to a visible gap because the interior edge b would not be drawn.

Selecting a safe leaf from a range requires traversing the topological tree dataset and using the split lines associated with topological tree edges to quickly determine screen-space distances. Our leaf selection algorithm terminates after at most two partial upward traversals from a leaf toward the root. We ascend from the first leaf in the range until we find an internal node whose vertical edge is larger than the screen-space extent of the partition. We then jump to the first leaf in the next subtree over, and if we are still within

the partition we again ascend until we find an edge larger than the partition size. We choose the leaf belonging to the leftmost of these two candidate edges.

Working through the example shown in Figure 9 Right for the range $[L_s, L_k]$ illustrates why this algorithm works. We denote the maximum vertical screen-space extent of a partition as τ , shown as the green filled-in area; Appendix A presents a detailed justification for setting τ to one-quarter of a pixel. Our selection traversal starts at the first leaf node L_s in the range. We ascend to the ancestors of L_s until we find the first internal node larger than τ , which is A ; the size of A is the sum of the sizes of leaves under A . It follows that the size of B , the child of A on the path to L_s , is not as large as τ , so we know that we can draw the subtree under B as line of a single pixel. One point of caution about rendering B as a single pixel width line: if the path under B to L_s crosses between two pixels, a jagged line will be displayed. In Appendix , we show that these jagged lines do not matter with τ smaller than one-quarter pixel, and how such paths cause gaps in TreeJuxtaposer rendering. We will draw the leaf path from the starting node L_s if no other subtree that is larger than τ can be drawn by drawing a path from L_s to B .

We locate the next leaf to ascend, L_{i+1} , by finding the node adjacent to L_i , the maximum leaf under A ; our mapping process gives us $O(1)$ lookup time for L_{i+1} following the ordering of leaves mapped to split lines. Lookup of L_i from A is also $O(1)$ since subtrees keep references to their minimum and maximum leaves. Our algorithm continues by ascending from L_{i+1} because this leaf is still in the range $[L_s, L_k]$. Similar to finding B , the ascent finds C to be the uppermost node not as large as τ . However, the pixel-high path from L_{i+1} to C would be shorter than the path from L_s to B , so we keep L_s as the representative leaf rather than switching to L_{i+1} . Finally, the maximum leaf under the parent of C is outside the range $[L_s, L_k]$, so our algorithm terminates, choosing to draw the path from L_s to the root; in fact, any leaf in $[L_s, L_i]$ is a good choice.

By using τ in our ascent termination criterion, we limit the number of necessary ascents to at most two per leaf range. A subtree larger than τ would exit the leaf range on at least one of the two possible sides of the range. Leaf selection, and thus drawing, is linear in the number of partitions; that is, in the number of vertical pixels.

Our leaf selection algorithm has many possible safe choices for representative leaves, so we have no guarantees that leaves corresponding to a marked group will be chosen. We explicitly seed the queue with ranges of marked objects to ensure guaranteed visibility, as we describe next.

Marked Groups

In PRISAD, **marked groups** are sets of geometric items that should be drawn in a specified color. These groups might contain computed differences, or user selections. Each tree node has a unique key in our topological structure. Keys are assigned by a pre-order traversal, so every complete subtree of the topology is a single, continuous range of keys, with the root node key smaller than all other keys, as shown on the tree layout in Figure 6a. For each marked group, we store the ranges in a binary search tree structure, which allows us to search the list of all marks for any node in $O(\log r)$ time, for r marked ranges. Instead of storing every individual node in the search tree, we store ranges of marked nodes, and concatenate any adjacent marked node ranges if possible. This look-up is much more efficient than the $O(m)$ cost of TreeJuxtaposer, where n is the number nodes of the dataset. Although TreeJuxtaposer cached the last computed group after each marking action, Figure 16 shows that the cost of color look-up before caching is very slow in a worst-case marking situation.

To provide visual landmarks during animated transitions, our progressive rendering algorithm draws a skeleton representing

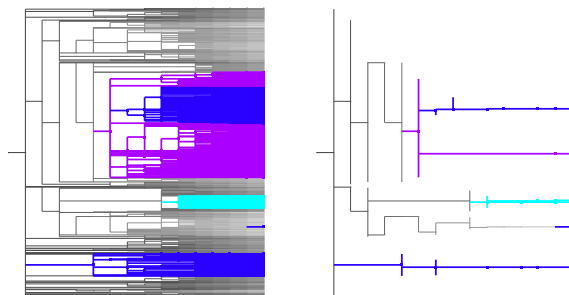


Figure 10: **Left:** A fully rendered tree scene with several colored marks. **Right:** The skeleton view of the same tree, with each marked group represented as a path from node to root.

marked groups before drawing the rest of the scene. TreeJuxtaposer also renders marked groups before unmarked objects, but there is no guarantee of finishing in one frame if the marked regions contain large ranges. Unlike TreeJuxtaposer, PRITree progressive rendering only draws a single leaf path from any leaf in the marked range to the root, for each marked range. This sparse marking, as shown in Figure 10, draws enough of each range to quickly portray a useful skeleton of marks at low cost, and also guarantees that sub-pixel width subtrees are not culled out of the scene. The time to render a skeletal path is $O(h)$ for a subtree of height h , which is usually at most $O(\log(n))$, versus $O(n)$ for a subtree containing n nodes. With this improvement, we also render skeletal paths for all marked groups in the first frame.

Picking

Picking is the inverse problem from rendering, namely going from a screen-space region representing a cursor picking area to a geometric object. Just as with rendering, providing realtime responsiveness becomes more difficult as dataset size grows. Many of the tree edges in PRITree are skinny, so the the well-known Fitts' Law [7] effect holds that small targets can be irritatingly difficult to select. The obvious solution is allowing a small **picking fuzz** region around objects to be considered as a hit. However, fuzz alone is not sufficient: backtracking is a robust solution to the problem. The quadtree-based picking of TreeJuxtaposer and TJC-Q did not support backtracking, and picking could fail in sparse regions when a quadtree cell was empty, even though an adjacent cell within the picking fuzz region had a pickable tree node

PRITree picking is a descent-based technique with backtracking, described in pseudocode in Figure 11. We find a child node, N_k , of some tree node N , that is enough within the picking fuzz of the cursor, M , and store adjacent sibling tree nodes, N_{k-1} and N_{k+1} , for later backtracking if we choose the wrong child node. When we do not find a tree node with our choice in the path, we begin using the contents of the stack. Since our PRITree layout technique fills the entire grid, such that subtree roots cover the extent of their leaves in the direction of S_Y , our style of picking does not rely on descending the exact subtree that is above M . We may be "off-by-one" in either direction safely, because if a backtrack is necessary, the only possible subtree to examine would be the one geometrically adjacent subtree in the S_Y direction.

This method works just as well as other descent methods when regions are dense, because we can guarantee to find a close enough node anywhere within the picking fuzz range in a dense region, and we approach M the further we descend in the hierarchy. An example of a sparse case where backtracking is necessary is when a very narrow subtree is adjacent to a very wide subtree. In this case, the narrow subtree is hard to pick, so most often a picking

Picking Function

input: mouse screen position $M = (X, Y)$
 root $\text{TreeNode } T = (\text{kids}, \text{cell})$ where
 $\text{kids} = \{T_0, T_1, \dots, T_{n-1}\}$
 $\text{cell} = (X_{\min}, X_{\max}, Y_{\min}, Y_{\max})$
output: picked $\text{TreeNode } T_{(X,Y)}$, a node close to (X, Y)

```

stack  $S \leftarrow \emptyset$ 
 $S.\text{push } T$ 
while  $S \neq \emptyset$ 
   $N \leftarrow S.\text{pop}$ 
  if  $(X, Y)$  over edge of  $N$ 
    return  $N$  // return  $N$ 
   $xMin \leftarrow N.\text{cell}.X_{\min}$ 
  if  $N.\text{isLeaf}()$  or  $N.\text{cell}.bounds(Y)$  or  $xMin > X$ 
    continue // throw away  $N$ 
   $k \leftarrow \text{BinarySearch}(N.\text{kids}, Y)$ 
  if  $k > 0$ 
     $S.\text{push } N_{k-1}$ 
  if  $k < n - 1$ 
     $S.\text{push } N_{k+1}$ 
   $S.\text{push } N_k$ 
end while
return  $\emptyset$  // no node picked
  
```

Figure 11: PRITree *Picking*: descend tree under node T until a tree node close to mouse coordinates (X, Y) is found. Stack S is used for backtracking if a descent is unable to find a tree edge; at each step of the descent, two siblings of N_k , the next node to be checked, are pushed onto S . All screen-space distance functions: BinarySearch ; $N.\text{cell}.bounds(Y)$; M over edge of N , apply a picking fuzz.

algorithm may select the wide subtree and give up even when the cursor is very close to the narrow subtree.

When the cursor is in a wide subtree, but is too far to pick any node in that subtree, we know that the cursor must be vertically between some node in the subtree and one of the siblings of the subtree. The sibling of the subtree that is vertically opposite the cursor is not pickable, so we know that for any backtracked descent, at most one sibling that we cache in the stack is useful. Also, when backtracking, we know that we will backtrack to exactly the appropriate sibling we need to find the cursor. This is true because a backtrack means that the cursor is vertically further than the picking fuzz away from the edges of the subtree that define the bounds of its drawn tree edges. Only ascending to the node that bounds the cursor on the opposite side will continue the picking descent. Therefore, although our algorithm always caches both siblings, when possible, we follow at most two paths in the entire tree if the cursor is in regions where nodes are sparse; it is cheaper to cache first, and determine if the siblings are appropriate later.

Our overall complexity depends on the branching factors of the nodes involved, since a binary search is required at a cost of $O(\log c)$, where c is the maximum branching factor. For paths that descend into dense regions, we incur the costs of traversing the height of our tree, which is $O(H)$, for a tree of maximum height H . Therefore, our overall picking complexity is $O(H * \log c)$.

PRISEQ

The PRISEQ partitioning exploits the probability of vertical coherence in a column of nucleotides, as discussed in the section on PRISAD screen space rendering. Our goal is a rendering algorithm with complexity that depends on the number of pixels as opposed to the dataset size. In PRITree, culling occurs in only one direc-

tion: leaves are culled, and the drawing strategy hinges on culling by careful selection along a leaf path. In PRISEQ, we need to cull in both directions. We aggregate information about the entire region encompassed by a split line to draw a representative object for it. These representatives are computed at most once, by caching the results of lazy evaluation.

Rendering: Column Aggregation

We aggregate across multiple columns according to the split line hierarchy. Recall that split lines encompass regions of space, with lines higher in the hierarchy subtending larger regions, and that the partitioning respects this hierarchical structure. SequenceJuxtaposer selects a nucleotide in a region at random for every frame, giving a misleading visual indicator of nucleotide density and causing flicker during transitions due to the lack of frame-to-frame coherence. Since the sequence layout introduced by SequenceJuxtaposer uses filled rectangles in a packed grid, our partitioning stopping criterion τ for PRISEQ can be set to terminate partitioning columns at one pixel resolution.

Our PRISEQ representative object reflects the density of nucleotides in the region in question; specifically, we find the most frequently occurring nucleotide in the region and use its color. Representatives are recursively computed and cached, so finding a higher-level split line automatically populates the cache with its descendants. We break ties with random selection from the candidate colors, but the true nucleotide counts are propagated upwards so that the selection does not bias its ancestors, and so that the selection persists across frames due to the caching. Figure 12 shows a small example. After the representative objects are computed for each row of an aggregate column, our previously described run-length encoding strategy is used to minimize rendering time and save storage space.

	k	k+1	k+2	k+3	[k, k+1]	[k+2, k+3]	[k, k+3]
SeqA	A	A	C	C	A	C	C
SeqB	A	C	C	C	A	C	C
SeqC	G	G	C	G	G	G	G

Figure 12: PRISEQ recursively aggregates information for columns encompassed by split lines to determine which nucleotide color should be used for the representative object. **Left:** No aggregation is performed at the highest magnification since every nucleotide is visible. Rendering column $k+2$ requires drawing only a single vertical rectangle since C is in every sequence for that column. **Center:** For column range $[k, k+1]$, SeqB has a tie, so A is randomly chosen but the true counts are propagated upwards. **Right:** When aggregating all four columns, C is found to occur most frequently for SeqB .

Aggregating a single region encompassed by a split line has a one time cost of $O(r)$, where r is the number of nucleotides in the range. We could precompute the aggregation for the entire split line hierarchy in the mapping stage of world-space discretization, in the empty step in Figure 7d, but we instead save time and space by lazy evaluation that fills the mapping cache. The runtime cost for drawing a frame where all aggregated columns are found in the cache is $O(h * v)$ where h is the number of horizontal pixels and v is the number of vertical pixels, because there are at most h columns, drawing a column requires at most $O(v)$ work, and cache lookup time is constant. The number of sequences or nucleotides may far exceed the number of vertical or horizontal pixels, but our aggregation method for PRISEQ renders only $O(p)$ geometric objects in $O(p)$ time, where p is the number of on-screen pixels and $p = h * v$.

Marked Groups

Similar to PRITree, marked groups in PRISeq are given seeding priority over the rest of the dataset. Each partitioned region in PRISeq that contains a marked item is drawn with an additional colored rectangle across some vertical range of the culled area. Regions that are horizontally adjacent are rendered with a continuous marked area. We store each marked region type in PRITree as a separate marking tree. The marking trees use a standard binary tree library, and store continuous ranges of nucleotides that are marked for that region type.

We enqueue marks in our rendering queue by processing our marking trees in nucleotide, then sequence order. For each marked nucleotide range, we find the culling regions that a mark belongs to, which may be several nucleotides long, and several sequences high. After we find the horizontal ranges for each mark, we determine if any mark is adjacent to the last culling region, and append adjacent regions until we find a marking discontinuity.

Our marked region drawing is much faster than rendering the marks with the rest of the dataset, since we draw marks as continuous rectangles of the same marking color. Since we cull marks in both directions, we achieve $O(h*v)$ rendering time for h horizontal and v vertical screen pixels. The search for marks itself depends on the current marking state. Each marked range search takes $O(\log k)$ time for k marked ranges, which indicates that total marked range rendering is $O(h*v + k*\log k)$, since we search each marked range for a culled region. In practice, this upper bound is quite liberal: we typically perform fewer than k searches, and would only draw at most k marked ranges with a brute force approach. We chose our method because it renders fewer geometric objects, and is capable of rendering $O(h*v)$ marks on with interactive frame rates.

Picking

The partitioning of nucleotides produces a screen-space division of culled geometric objects in the horizontal direction. To assist in picking visible on-screen objects, we also partition the vertical split line component of PRISeq into ranges of sequences that subtend the same pixel. By having a partitioned split line hierarchy in both directions, PRISeq picking becomes a binary search for a small picking region around the cursor position in the horizontal and vertical directions. Since our rendering process uses the same partitions to determine what to draw for each rectangular culled range, we can use the same descent operation for picking to ensure that what we see is what we pick.

PRISAD PERFORMANCE

This section shows how our applications, PRITree and PRISeq which use our generic AD infrastructure, compare with functionally similar applications. Therefore, although we would like to compare PRITree with the performance of TJC, which can render datasets with three times the number of tree nodes as PRITree, we can only assert how TJC would fare using our test datasets. It is worth mentioning that the performance of TJC on binary trees [4] is available, but as we show in the following section on PRITree, using datasets with different characteristics, especially real-world examples, may not be as fast or as memory efficient.

PRITree vs. TreeJuxtaposer

In this section, we evaluate the performance of PRITree (PT) using TreeJuxtaposer (TJ) performance for identical actions as our benchmark. All performance tests were performed using a 3.0 GHz Pentium IV processor, Java 1.4.2.04-b05 HotSpot runtime environment with a maximum heap of 1.8 gigabytes, GL4Java v1.4 graphics libraries, and an nVidia Quadro FX 3000 video chipset, running

twm in XFree86 version 4.3.99.902. The window size was set to 640 by 480 pixels, and timing results were output by millisecond-accurate Java system functions, and averaged from several manually prompted redrawings of each tested dataset.

First, we compare the performance of both applications with respect to rendering a series of synthetic and large, real-world datasets. Our analysis of both total scene rendering time and memory consumption shows that we do not lose performance by switching from application-specific algorithms to the generic infrastructure of PRISAD; on the contrary, we achieve a speed-up. We then investigate the worst-case marking performance on the comparison of large datasets.

The space of all possible trees is vast and hard to classify. We use two sequences of synthetic data that bound the degree of nodes: balanced binary trees, and **star trees**: the bushiest possible trees where all nodes but one are leaves, attached to a single root node. For real-world datasets we chose two pairs of large comparable trees: the InfoVis 2003 contest classification trees (IVC) [12], each with over 190,000 nodes; and two Open Directory Project categorization trees (ODP) [1], from March and June 2004, each with over 480,000 nodes.

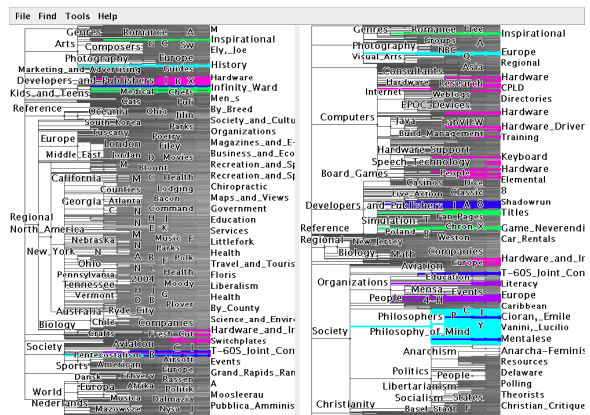
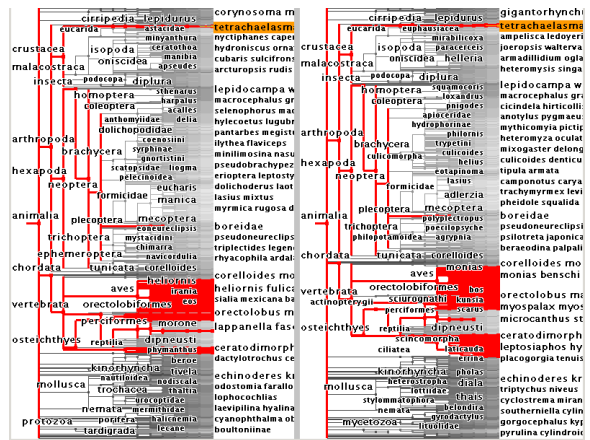


Figure 13: Our real-world application performance testing datasets. **Top:** Comparison of two InfoVis 2003 contest classification trees, each with over 190,000 nodes. **Bottom:** Two OpenDirectory Project categorization trees, each with over 480,000 nodes, with some marked subtrees. There are over 30,000 differences between these two trees, which makes this rendering very slow with differences enabled.

Results

The top of Figure 14 shows that both TJ and PT rendering time performance on star trees is comparable with a small number of nodes. Once the star trees include more leaves than available vertical screen pixels, PT culls efficiency while TJ continues to render the entire dataset. Star trees of 130,000 nodes take one second for TJ to render, while any star tree larger than 4000 nodes renders in 50 ms. TJ performs poorly with bushy trees, since when the root node is larger than one pixel, TJ will draw all of its children. PT quickly reaches a constant-time plateau with star trees, showing that PRISAD has succeeded in setting strict limits in the number of leaves to draw through partitioning: the number of leaves rendered is at most four times the number of vertical pixels on screen. Although the TJC system was not tested on this type of graph, the rendering algorithm used means that it is likely to exhibit similar poor behavior for the star tree case as TJ.

The center of Figure 14 shows again that rendering time performance is identical between PT and TJ until graphs of larger than 4000 are rendered. The performance of binary trees in PT becomes sub-linear after a threshold number of nodes. Again, when PT renders datasets with four times the number of leaves as vertical pixels, it only renders that many more nodes for every doubling in size of the balanced binary trees. This progression of drawing a constant number of leaves more for every doubling in dataset size is exactly the graph of $O(\log n)$, for trees with n nodes. The graph does not remain linear in the semi-log plot due to the increased height of traversal necessary to find the first large subtree from each leaf during the rendering traversal. A more favorable dataset for PT would have a larger branching factor to reduce that depth of traversal. Also, note that the TJ trendline has some peculiar features: the rendering time for a binary tree of 262,143 nodes is faster than a tree of less than half its size. This downturn illustrates the over-culling problem in TJ, where large binary trees are incorrectly rendered with gaps.

The bar graph on the bottom of Figure 14 shows how PT and TJ react with real-world datasets, after the first scene has been drawn. IVC includes many high-degree internal nodes, and the slow performance of TJ during the contest comparison is primarily related to the overdrawing of dense regions. PT is capable of rendering a single IVC tree over five times faster than TJ. For IVC on both applications, the rendering time appears to be approximately double for tree comparisons. PT renders two ODP trees much slower under comparison due to the very large number of differences between trees; there are 30,000 differences due a great number of sparse leaf changes between the datasets, all of which are rendered for guaranteed visibility. Since there are relatively few local differences, marked group look-up and rendering is not a huge cost for IVC, when compared to ODP. It is also interesting to note that the rendering time of TJ for IVC comparison is less than twice the rendering time for a single IVC tree, which may be related to how TJ caches marked nodes.

In Figure 15 top, we see that the binary and star trees series both consume linear amounts of memory, but with different constants. The PT memory performance comparison reveals that PT is easily capable of loading star trees four times larger, or binary trees three times larger, than TJ. In the bottom of Figure 15, we note that for the contest comparison, PT is about five times as efficient as TJ. Also, the footprint of PT from single trees to comparisons is very close to double, while TJ consumes about four times the memory for comparisons of the IVC dataset. This means that since our methods of storing marked nodes is the major difference between these two applications once the grid layout methods are normalized, that our PT marked node storage is far more efficient than methods used by TJ. Considering the number of marked nodes in the ODP dataset comparison that causes it to render very slowly, this is a surprising result of the memory efficiency of PRITree.

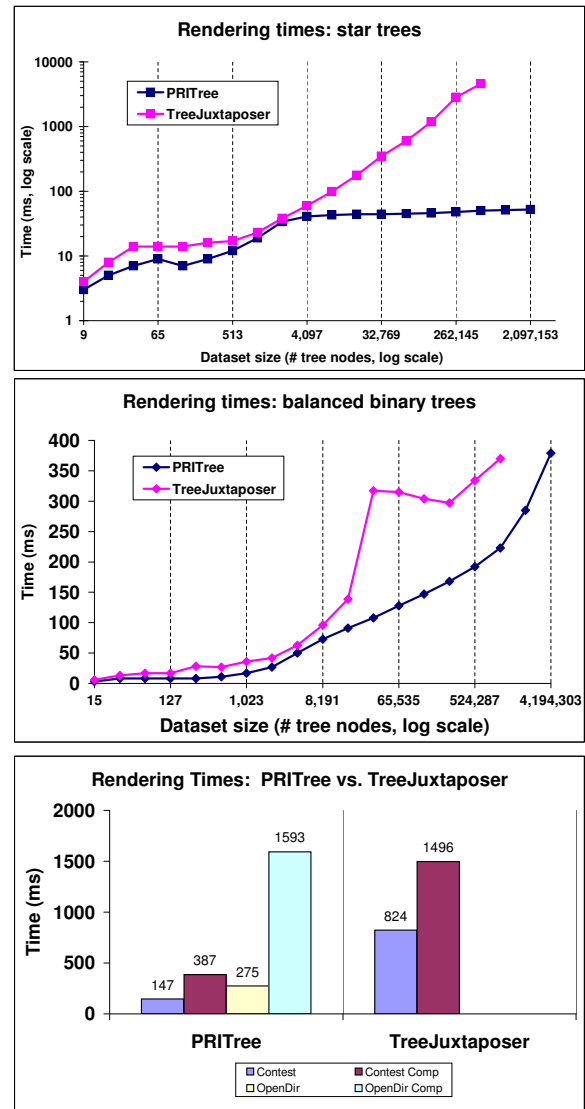


Figure 14: **Top:** Rendering times for PRITree and TreeJuxtaposer with star trees. Both applications render efficiently until 4000 nodes, after which PRITree remains constant and TreeJuxtaposer remains linear. **Center:** Rendering times for PRITree and TreeJuxtaposer with balanced binary trees. Applications diverge at 8000 nodes, where PRITree becomes nearly logarithmic until 0.5 million nodes but is still sublinear afterward, and TreeJuxtaposer begins to have rendering errors and therefore inconsistent performance times. **Bottom:** Rendering times for PRITree and TreeJuxtaposer with real-world datasets. PRITree is much faster even with the OpenDir dataset that has more than double the nodes of the Contest dataset, but becomes very slow with the OpenDir comparison with over 30,000 guaranteed visibility marks. The rendering time for the first frame of a comparison is ignored to allow TreeJuxtaposer to cache marks.

Finally, in Figure 16, we see that the performance of PRITree is orders of magnitude faster than TreeJuxtaposer immediately after marking. The first scene drawn after marking with TreeJuxtaposer must recompute colors for each node in the topology, which requires linear traversal through a list of all marked nodes. PRITree does not cache marks for nodes, which gives slower post-marking performance, but only a small one-time cost for computing the colors for all nodes. By not caching the marks in PRITree, we decrease

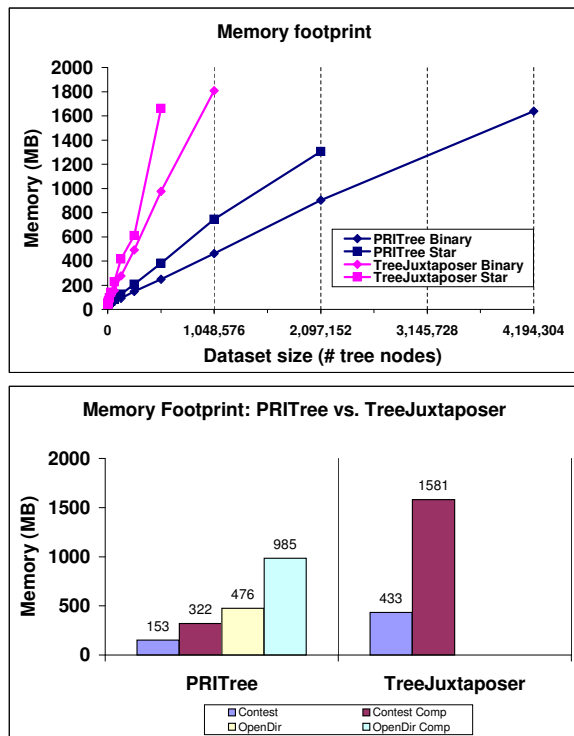


Figure 15: **Top:** The memory footprints of binary and star synthetic datasets with PRITree and TreeJuxtaposer. PRITree is four times more efficient with star trees and three times more efficient with binary trees. **Bottom:** Memory footprints of real world datasets, InfoVis Contest and OpenDirectory, with PRITree and TreeJuxtaposer. PRITree is three times more efficient with single trees and five times more efficient with comparisons. TreeJuxtaposer could not load the OpenDirectory dataset, so this efficiency analysis is conservative.

our memory footprint, leading to better scalability.

PRISeq vs. SequenceJuxtaposer

The result of using the PRISAD framework is order-of-magnitude improvements in both time and space for PRISeq (PS) compared to SequenceJuxtaposer (SJ). PS can handle datasets of 6400 sequences of 6400 nucleotides each, for a total of 40 million nucleotides, which is a twenty-fold improvement over the 1.7 million nucleotide limit of SJ. Rendering a dataset of 44 species with 17,000 nucleotides, for a total of 740,000 nucleotides, takes 7 seconds with SJ [16]. PS can render the same dataset in less than one half-second.

Action / Application	TreeJuxtaposer	PRITree
First Scene Unmarked	115	0.27
Subsequent Scenes Unmarked	1.5	0.27
First Scene Marked	130	2.5
Subsequent Scenes Marked	1.5	0.55

Figure 16: The marking time performance, in seconds, for a classification tree from the InfoVis 2003 contest [12].

FUTURE WORK AND CONCLUSIONS

Many users have requested editing functionality for trees, which would require modifying PRISAD to support dynamic rather than static data. Adding internal logging capabilities to PRISAD would also benefit users who wish to undo actions, replay their activities, or load a previously saved navigation state. Finally, we would like to combine PRITree and PRISeq to allow biologists to explore the interplay between genomic data and hypothesized evolutionary trees.

We have presented PRISAD, a partitioned rendering infrastructure for scalable accordion drawing. Our infrastructure is the first to provide a generic interface to the accordion drawing features of rubber-sheet navigation and guaranteed visibility of marked nodes. Additionally, PRISAD tightly bounds overdraw with pixel-based rendering constraints; all partitioning terminates at a known pixel-based value and the application-specific algorithms are prohibited from further partitioning. These constraints yield bounded rendering time performance for several tree sizes and topologies evaluated in comparison to TreeJuxtaposer performance. PRITree and PRISeq are applications built on PRISAD that duplicate the feature sets of TreeJuxtaposer and SequenceJuxtaposer, respectively. A detailed comparison of PRITree and TreeJuxtaposer, using the IVC dataset, shows an improvement of three to four times more efficient memory usage, and five times faster rendering. Our new data structures and algorithms for marking groups in PRITree yield an order of magnitude speed increase. PRISeq provides order-of-magnitude improvements for both rendering speed and memory usage. PRITree and PRISeq are open source and available for source or binary download at <http://olduvai.sf.net>.

ACKNOWLEDGEMENTS

Funding was provided by NSF/DEB-0121651/0121682, NSERC/RGPIN 262047-03, and Hildebrand was supported by the German Academic Exchange Service. We thank Ciarán Llachlan Leavitt for comments on paper drafts.

REFERENCES

- [1] Open Directory Project [WWW document] <http://www.dmoz.org/> (accessed 21 October 2005).
- [2] Auber D. Tulip - a huge graph visualization framework. In: Mutzel P, Jünger M (Eds). *Graph Drawing Software*, Mathematics and Visualization series. Springer-Verlag: London, 2003; 105–126.
- [3] Bederson B, Grosjean J, Meyer J. Toolkit design for interactive structured graphics. *IEEE Transactions on Software Engineering* 2004; **30(8)**: 535–546.
- [4] Beermann D, Munzner T, Humphreys G. Scalable, robust visualization of very large trees. *Seventh Eurographics / IEEE Visualization and Graphics Technical Committee Symposium on Visualization (EuroVis) 2005* (Leeds, UK), IEEE Computer Society Press, 2005; 37–44.
- [5] Fekete JD. The InfoVis Toolkit. *Tenth IEEE Symposium on Information Visualization 2004* (Austin, Texas), IEEE Computer Society Press, 2004; 167–174.
- [6] Fekete JD, Plaisant C. Interactive information visualization of a million items. *Eighth IEEE Symposium on Information Visualization 2002* (Boston, Massachusetts), IEEE Computer Society Press, 2002; 117–124.
- [7] Fitts P. The information capacity of the human motor system in controlling the amplitude of movement. *Journal of Experimental Psychology* 1954; **47**: 381–391.
- [8] Heer J, Card S. DOITrees revisited: scalable, space-constrained visualization of hierarchical data. *Advanced Visual Interfaces 2004* (Galipoli, Italy), ACM Press, 2004; 421–424.
- [9] Hubbard T et al. The Ensembl genome database project. *Nucleic Acids Research* 2002; **30(1)**: 38–41.

- [10] Kent WJ, Sugnet CW, Furey TS, Roskin KM, Pringle TH, Zahler AM, Haussler D. The human genome browser at UCSC. *Genome Research* 2002; **12**: 996–1006 .
- [11] Munzner T, Guimbrètière F, Tasiran S, Zhang L, Zhou Y. TreeJuxtaposer: Scalable tree comparison using Focus+Context with guaranteed visibility. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2003)* 2003; **22**(3): 453–462 .
- [12] Plaisant C, Fekete JD. InfoVis 2003 contest [WWW document] <http://www.cs.umd.edu/hcil/iv03contest/> (accessed 8 July 2003).
- [13] Sarkar M, Reiss S. Manipulating screen space with *StretchTools*: Visualizing large structures on small screens. Technical Report CS-92-42, Department of Computer Science, Brown University, 1992.
- [14] Slack J. A partitioned rendering infrastructure for stable accordion navigation. Master’s thesis, University of British Columbia, 2005.
- [15] Slack J, Hildebrand K, Munzner T. PRISAD: A Partitioned Rendering Infrastructure for Scalable Accordion Drawing. *Eleventh IEEE Symposium on Information Visualization 2005* (Minneapolis, Minnesota), IEEE Computer Society Press, 2005; 41–48.
- [16] Slack J, Hildebrand K, Munzner T, St. John K. SequenceJuxtaposer: Fluid navigation for large-scale sequence comparison in context. *German Conference on Bioinformatics 2004* (Bielefeld, Germany), Gesellschaft für Informatik, 2004; 37–42.
- [17] Wills GJ. NicheWorks: Interactive Visualization of Very Large Graphs. *Computational and Graphical Statistics* 1999; **8**(2): 190–212 .

APPENDIX PRITREE TRAVERSAL DETAILS

Leaf overculling

The primary focus of previous tree rendering applications, such as TreeJuxtaposer and TJC, is to minimize the number of branches drawn for a subtree beneath a node, rather than minimizing the global number of nodes drawn. Attempts by these applications to prevent overdrawing fail for some complex topologies, as demonstrated by the evaluation of TreeJuxtaposer in Section . Overdrawing *between* topologically partitioned components is the major inefficiency of top-down partitioning and rendering. Top-down approaches do not consider overlaps of adjacent topologies, which in some datasets renders ten times the number of leaves than there are vertical screen pixels.

Our PRITree rendering begins by drawing tree scenes starting from the set of all leaf nodes, and then proceeding bottom-up, or toward the root node. The leaf nodes are partitioned in a separate process from the drawing algorithm, which simplifies the entire rendering algorithm. We can partition and draw simple paths from the leaves to the root provided that it is still possible to correctly render the entire scene, which means no visible differences from the brute-force drawing of every node. In this section, we show that the maximum size for partitioning leaf ranges, to prevent overculling at the leaves and without exact pixel arithmetic, is half the width of a pixel.

If τ , the maximum partition size of leaf ranges, is set to one pixel, then we may underdraw nodes at the leaf level, which then propagates rendering errors to nodes higher in the topology. When both adjacent leaf ranges draw outside of a shared pixel, as shown in Figure 17, gaps may appear in many places throughout the topology. One solution to this problem would be to perform exact pixel arithmetic to ensure each dense leaf region is subdivided until every leaf range is contained within some pixel.

Our solution, which does not use exact pixel arithmetic, guarantees rendering in every pixel for leaf ranges by using τ of smaller than one-half pixel. As shown in Figure 18, a smaller τ guarantees rendering into each pixel in the set of all leaves. However, this is only a solution for complete rendering of dense regions of leaf nodes; the complexities of bottom-up rendering are discussed next.

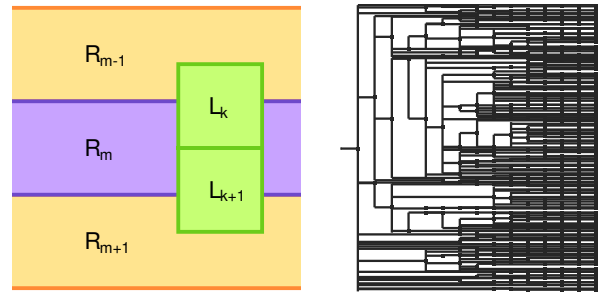


Figure 17: If τ is too large, then rendering gaps are visible throughout the tree topology. The adjacent leaf ranges L_k and L_{k+1} render a single leaf, which may be in pixels adjacent to pixel row R_m , rather than in row R_n itself which would be left blank.

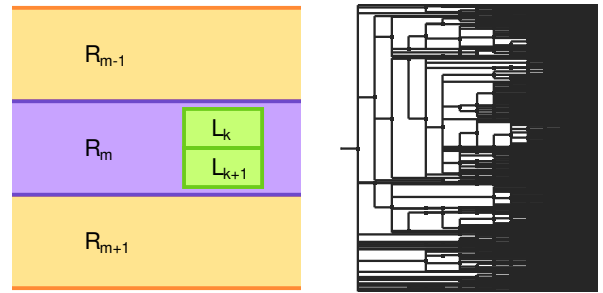


Figure 18: Restricting τ to less than one-half pixel prevents gaps in rendering the set of leaves at the expense of overdrawing. Other gaps in rendering are also prevented by our tree traversal.

Hierarchical overculling

After AD partitions the split line hierarchy to form a set of consecutive, non-overlapping leaf ranges, PRITree rendering draws one leaf path per leaf range. The leaf path consists of every ancestor, along the path to the root, of one carefully selected leaf in each range. Selecting the wrong leaf will result in drawing errors, which we refer to as hierarchical overculling. Unlike leaf overculling, we may notice these drawing errors in sparsely populated regions of leaf nodes.

Consider a path of tree nodes, P , drawn from a leaf toward the root, which is entirely contained in a given pixel row. P may be culled and not drawn if another path of nodes, Q , from the same leaf range, may be drawn over the entire length of P . If both P and Q terminate at a common node, R , in the topology, then the subtree of nodes under R between P and Q can be culled to the same path on-screen path; this logic is similar to the subtree culling arguments used in TJC [4].

The more difficult case occurs when P and Q do not terminate at the same node. To determine which of P or Q is the better for rendering, we must traverse, as described in Section , to find the longest of these two paths. The termination criteria of the subtree width for P and Q , which we call ψ , is at least as large as τ in order to guarantee a strict bound of two ascents per leaf range. However, if we also apply the restriction that the sum of τ and ψ is less than one-half pixel, then we may use a similar argument from the previous section that filled all rendering gaps in the range of all leaves. Consider the following equations, where p is the width of a pixel:

$$\psi \geq \tau \rightarrow \psi - \tau \geq 0 \quad (1)$$

$$\tau + \psi < p/2 \rightarrow p/2 - \tau - \psi > 0 \quad (2)$$

$$p/2 - 2\tau > 0 \rightarrow \tau < p/4 \quad (3)$$

$$\text{maximize } \tau \rightarrow \tau = p/4 \rightarrow \psi > p/4 \quad (4)$$

where (3) is the addition of our restrictions, (1) and (2). Since we also want to minimize the number of partitions, we maximize the size of τ to give us (4). This final solution tells us that with our restrictions, we have optimal solutions of τ and ψ , which means that we render up to four times the number of leaves as there are vertical pixels on-screen and each leaf range tree ascent requires at most two traversals. The advantage of this result is that we do not have to perform exact pixel arithmetic on adjacent subtrees, which would become costly for complicated tree datasets. Instead, we have a rendering result that depends only on the number of on-screen pixels, which reduces the cost of rendering complex and dense datasets.