PRISAD: A Partitioned Rendering Infrastructure for Scalable Accordion Drawing

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.

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

1 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 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.

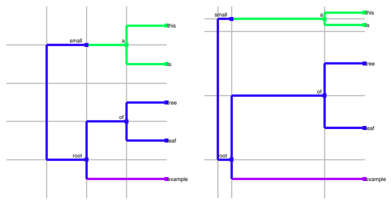


Figure 1: On the left, we show a tree dataset drawn with uniformly allocated space for each vertical node width and horizontal node height. When navigating by stretching a rubber-sheet surface, as in the right figure, the distortions allocate more screen-space to some regions of nodes and other regions are squished into less screen-space.

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 subtend a single screen-space region, yielding high depth complexity. To achieve scalable rendering performance for large datasets, we must efficiently reduce the amount of drawing 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; we must ensure both marked node visibility, and a proper representation of the dataset in every distorted region of screen space.

This paper presents our generic PRISAD infrastructure, and two applications built on it. PRITree implements the feature set of TreeJuxtaposer for visually comparing hierarchies [4], and PRISeq has the functionality of SequenceJuxtaposer for visualizing multiple aligned genomic sequences [13]. Our contributions include:

- PRISAD tightly bounds overdrawing with pixel-based rendering constraints, giving much more time-efficient rendering of dense, complex regions.
- 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.
- PRITree traversal algorithms for drawing and picking exploit the dataset topology, instead of adding a memory-expensive external data structure.
- PRISeq traversal algorithms efficiently aggregate columns to accurately reflect relative nucleotide proportionality.

In the next section, we give an overview of related work. In Section 3, we discuss our generic approach to scalable accordion drawing. We present PRITree in Section 4 and evaluate its performance in Section 5. PRISeq is covered in Section 6. We describe possible future work and conclusions in Section 7. Appendix A contains supplementary details of our PRITree rendering techniques.

*e-mail:{jslack,hilde,tmm}@cs.ubc.ca
[†]University of British Columbia
[‡]Bauhaus University Weimar

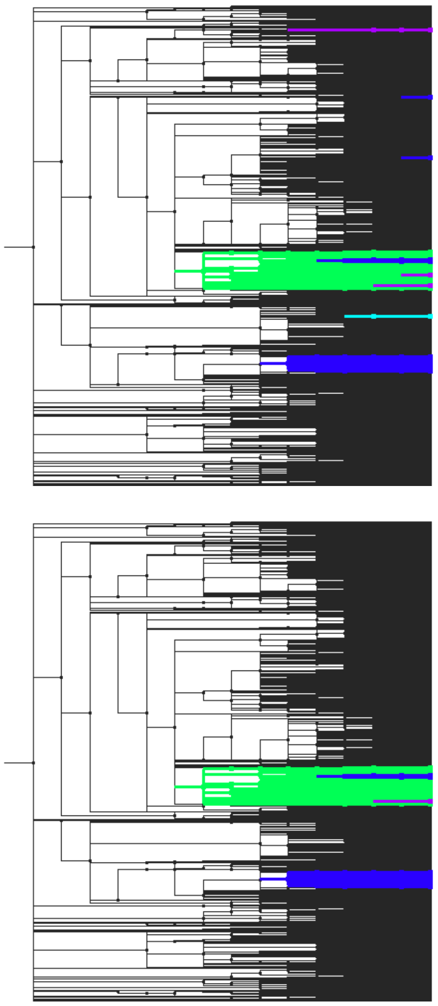


Figure 2: **Top:** For densely drawn regions of a dataset, we can mark several regions of interest with guaranteed visibility, and we always draw all marked regions that are smaller than a pixel. **Bottom:** In the identically marked tree without guaranteed visibility, these small regions may not be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

2 Related Work

The TreeJuxtaposer [4] 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 [4]. DOITrees [8], for example, have been used to explore the directory structure of the Open Directory Project website [10], 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 Section 5.

The TJC-Q and TJC applications [3] allow AD browsing of single trees, but do not support comparisons between multiple trees. PRISAD builds on the lightweight, grid-based AD infrastructure first proposed for these applications. Both perform culling when all leaves of a subtree subtend the same pixel, and are tuned for balanced binary datasets. Like TreeJuxtaposer, both share the limitations of being designed to handle only trees. The TJC application avoids the memory cost of quadtrees by supporting picking through the use of cutting-edge graphics hardware, and is capable of rendering a tree of 15 million nodes in one-third of a second. TJC-Q can run on commodity hardware, as can PRITree, and handles trees of 5 million nodes using lightweight quadtrees.

SequenceJuxtaposer [13] 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 [5] systems show sequence data with jump cut transitions between different scales. In Section 6.2, we compare PRISeq, shown in Figure 3, with SequenceJuxtaposer.

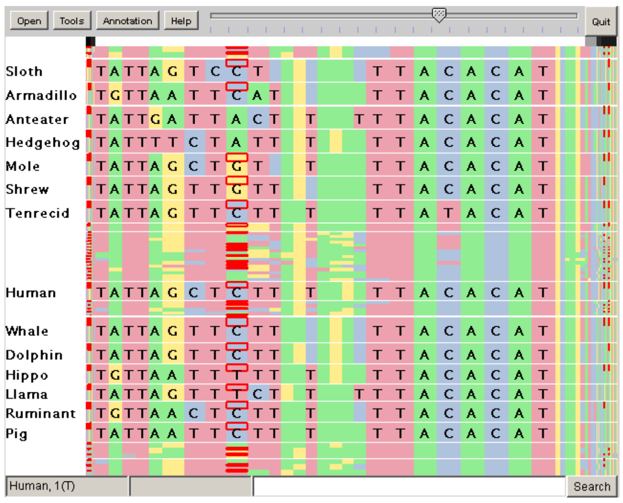


Figure 3: PRISeq is a genome sequence visualization application built on PRISAD with the feature set of SequenceJuxtaposer [13].

Slack discusses PRISAD and PRITree in detail in his thesis [12]. Few other information visualization systems can handle extremely large datasets. Fekete presents a system that can handle treemaps of one million nodes [7]. While AD could in theory be implemented within an existing toolkit such as the InfoVis Toolkit [6], its focus on generality rather than scalable accordion drawing precludes achieving the performance we describe here. The Tulip system for graph drawing [1] 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 [2] also provide support for multi-scale navigation through arbitrarily large 2D surfaces, but not guaranteed visibility of landmarks or rubber-sheet navigation. NicheWorks [14], 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.

3 PRISAD

Applications capable of interacting with PRISAD benefit from generic accordion drawing support for operations such as navigating, culling, and marking. We introduce our contribution of a generic infrastructure in Figure 4, by showing key components that PRISAD-enabled applications must provide to interact with PRISAD. We require the following algorithms for all PRISAD applications, and we discuss these algorithms for PRITree in Section 4:

- calculating the size of underlying PRISAD structures
- assigning dataset components to the PRISAD structures
- initiating a rendering action with two partitioning parameters
- ordering the drawing of geometric objects through seeding
- selecting or aggregating geometric objects for culling
- drawing individual geometric objects

Once an application meets these constraints, the PRISAD components are responsible for handling the following actions:

- initializing a generic 2D grid structure
- mapping geometric objects to world-space structures
- partitioning a binary tree data structure into adjacent ranges
- controlling drawing performance for progressive rendering

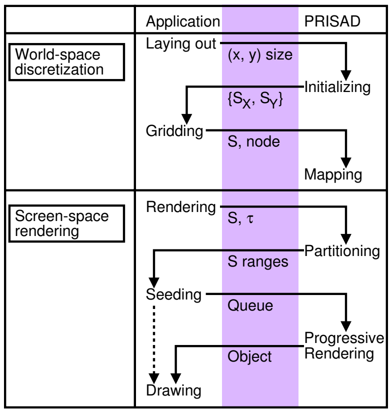


Figure 4: Initialization of a dataset in PRISAD applications requires a world-space discretization phase, which must generate several generic components from application-specific dataset structures. The rendering phase separates partitioning from drawing, which simplifies application drawing effort for faster pixel-based rendering performance. The blue column represents the communication required between separated generic and specific components. S, S_X, and S_Y refer to split line hierarchies, which we introduce in Section 3.1.

The initialization of PRISAD applications divide screen space into regions with a split line hierarchy, described in Section 3.1. These applications use a three-stage rendering pipeline, described in Section 3.2, which relies on application-specific components but provides several common AD algorithms for handling the flow of rendering between application and infrastructure.

3.1 Split Line Hierarchy

A **split line** is the dividing line of the 2D grid structure of an AD rendering surface; split lines partition screen space and are used to map world-space regions onto screen regions. TreeJuxtaposer uses a two-dimensional quadtree to support AD functionality, and the memory required for that additional data structure is the primary limitation of its scalability. We use less memory by decoupling the horizontal and vertical split line hierarchies, as proposed by TJC [3].

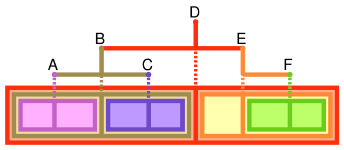


Figure 5: A split line hierarchy is a binary tree structure that provides a linear ordering and a hierarchical subdivision of areas. For instance, the region for split line B is bounded by its parent region D, and B separates its bounded descendants A and C.

Figure 5 shows that a split line hierarchy provides a linear ordering of the lines, and a recursive subdivision of regions. The initialization of a split line hierarchy for both horizontal and vertical components of our grid surface begins after the application determines the required grid size. 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 $O(p)$.

After initialization of our split line hierarchies, the application determines world-space positions for each node in both hierarchies. Furthermore, an application-specific object is attached to each split line in the hierarchy. In the case of trees, a leaf is assigned to each split line of the vertical hierarchy, whereas sequences store an aggregated column of aligned nucleotides in the horizontal hierarchy. We use this generic mapping step, as shown in Figure 4, to create a bidirectional mapping between split lines and their associated geometric objects that supports constant-time lookup.

3.2 Rendering pipeline

The rendering pipeline of PRISAD defines a generic structure for all applications that use our infrastructure. Our pipeline has three stages:

1. partitioning a split line hierarchy into small regions
2. seeding the partitioned split line regions and marked groups
3. drawing each seeded region as a geometric object

All previous AD infrastructures, which are tightly coupled to application-specific algorithms, perform partitioning during drawing. The use of embedded, drawing-time partitioning restricts those drawing methods to application-specific dataset domains. Since we separate partitioning from drawing, our application-specific drawing algorithms are simple, are bounded by the number of partitions, and do not require computation of screen-space positions to guarantee coverage of specific pixels. The next section describes partitioning in more detail.

3.2.1 Partitioning

In PRISAD a partition of the dataset into roughly equal sized screen-space regions must be recomputed every time navigation occurs. After partitioning, these screen-space regions are either smaller than one pixel, or contain only one geometric object to draw. Each region is bounded by split lines, so partitioning returns a list of split line ranges.

S_X and S_Y are the horizontal and vertical split lines hierarchies that constitute our AD grid in PRISAD. An application developer must determine which of the two hierarchies to partition for the rendering phase; we refer to the hierarchy used to render a scene as S. 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 S_Y so that the primary rendering direction is horizontal.

In contrast, for PRISeq the primary rendering direction is vertical, so we partition S_X. Vertical nucleotide columns are expected to be similar, because the rows of multiple gene sequences are aligned. We exploit this property to save time and space by run-length encoding. Rendering a single tall rectangle for a strip of identical nucleotides is faster than drawing many squares of the same color, and keeping a list of the rows where nucleotide changes occur is concise because of these similarities. For example, the left image in Figure 12 illustrates how we can draw the entire $k+2$ column for all sequences using a single vertical rectangle, and store only one row in the list.

An application developer must also determine the optimal value of the partitioning stopping criteria, τ, which we compute to be one-quarter pixel for PRITree and one pixel for PRISeq. The PRISeq case is straightforward, because the geometric object is a filled rectangle adjacent to its neighbors. The derivation of this optimal one-quarter pixel τ value for PRITree is given in Appendix A; the general discussion of choosing the best τ for all PRISAD applications is beyond the scope of this paper.

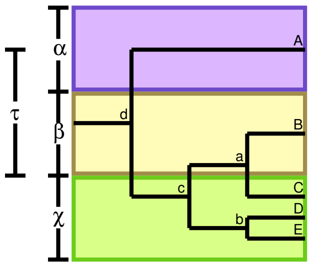


Figure 6: The partitioning phase of the rendering pipeline subdivides a split line hierarchy according to the sub-pixel stopping criteria τ, in this case subdividing horizontally. Widths of each horizontal partitioned stripe, α, β, and χ, are narrower than τ, and partitioning does not follow the topology of the dataset.

Once PRISAD is given both a split line hierarchy S and a sub-pixel stopping criterion τ, the partitioning process descends S until the minimum set of split line regions smaller than τ is found. The descent terminates when the process determines the first split line region smaller than τ, and we add that region to the partitioned queue, P. If the hierarchy cannot find a split line region smaller than τ for a descent, then the smallest possible region is added to P. When the partitioning process is complete, P will represent the entire hierarchy of S, where split line regions are grouped into either single geometric dataset items or ranges of items. For example, in Figure 6, A and B are single items in split line ranges α and β, and split line region χ stopped descent when it was smaller than τ, with the range of items $[C,E]$. We use the results of this partitioning to initialize the application specific seeding algorithm.

3.2.2 Seeding

Progressive rendering of large datasets requires a seeding stage, where applications can impose an ordering for drawing. For example, to ensure visibility of landmarks during animated transitions of datasets too large to completely render in one frame, we render the marked regions of PRITree first. The region that we enqueue for seeding is the partitioned split line hierarchy, described in Section 3.2.1. If we can draw the entire scene sufficiently fast such that progressive rendering is unnecessary, the seeding stage can pass the drawing order computed during the partitioning stage directly to the drawing stage, as shown by the dotted line in Figure 4.

4 PRITree

PRITree is our tree-based PRISAD application, which we use in this paper to benchmark performance differences between AD implemented in PRISAD applications, versus TreeJuxtaposer [4], the original AD application for visualization of trees. PRITree and TreeJuxtaposer are functionally equivalent, so we claim the performance advantages of PRITree manifest with our improvements in both AD infrastructure and tree-specific components.

PRITree requires new algorithms for translating tree structures for use in our generic infrastructure, as shown in Figure 6. To use our PRISAD infrastructure, PRITree performs laying out and **gridding** operations to align tree nodes to the smallest possible grid structure. In PRISAD, gridding is the positioning of a world-space object in a discrete, screen-space region, which is formed by the construction of a grid between horizontal and vertical split line hierarchies.

After the width and height of a tree are determined from parsing the tree dataset, PRITree sends the two grid dimensions to PRISAD, which returns a pair of split line hierarchies, S_X and S_Y. PRITree is then responsible for positioning each node in its rectilinear world-space layout position into the PRISAD grid by assigning a bounding rectangle in S_X and S_Y. Even though no rectangles overlap in PRITree, preventing the overlapping of geometric objects is not a generic restriction of PRISAD. When a leaf node is positioned in our gridding process, PRITree sends that leaf and its position in S_Y back to PRISAD for mapping. This mapping allows for constant-time bidirectional lookup for leaves near a given screen-space position and for an on-screen position given a leaf object from the topology.

We discuss traversing the topological tree in Section 4.1, and creation and traversal of data structures for guaranteed visibility in Section 4.2.

4.1 Tree Traversal

Tree traversal is the process of following a path from a starting node to an ancestor, or descendant, node. As discussed in Section 3.2.1, the partitioning process creates several adjoining ranges of leaves that represent the entire set of leaves in the tree. Each adjoining range subtends less screen space than the constant τ, which we derive as one quarter the size of a pixel in Appendix A. For each

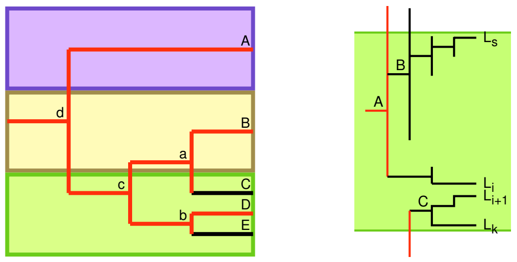


Figure 7: **Left:** Each partitioned range of leaves renders one path to the root from some leaf in its range; we only draw tree edges marked in red. The top two partitions contain single leaves, A and B, so they are the only choices. When deciding between C, D, and E, we must choose either D or E or else b will not be rendered, which would be an incorrect rendering gap. **Right:** Our selection traversal processes paths from the green leaf range to all subtrees with leaves in that range larger than τ. The black edges represent traversal paths and red edges stop the traversal from processing subtrees larger than τ.

partitioned range, we draw a leaf path, as shown by the red tree edges in the left image of Figure 7. Each path follows the shortest route from a selected leaf in the range to either the root or a previously drawn path. The selection of the leaf for each range is the most important run-time decision for our drawing algorithm, since poor leaf choice leads to an incomplete scene rendering. We now describe how our leaf selection process works, using the right image of Figure 7.

Our selection traversal starts at the first leaf node in the range L_s. 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 under 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. 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 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 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; in fact, any leaf in [L_s, L_i] is a good choice.

By incorporating τ as our ascent termination criteria as well as our partitioned leaf range maximum size, we limit the number of necessary ascents to, at most, two per leaf range. This limit is the direct result of paths ascending to either side of a leaf range, because a subtree larger than τ must exit the leaf range on at least one of the two possible sides of the range. This hard limit on the number of ascents per leaf range tightly bounds the amount of traversal necessary to render an entire scene. We defer further discussion on leaf choice to the Appendix A.2 derivation of the optimal value for τ.

4.2 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. For each marked group we store the ranges in a binary tree structure, which allows us to search the list of all marks for any node in $O(log\ r)$ time, for r marked ranges. This look-up is much more efficient than the $O(rn)$ cost of TreeJuxtaposer, where n is the number nodes of the dataset. Although TreeJuxtaposer cached the last computed group after each marking action, Figure 11 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 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 8, draws enough of each range to quickly portray a useful skeleton of marks at low cost. The time to render a skeletal path is $O(h)$ for a subtree of height h, 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.

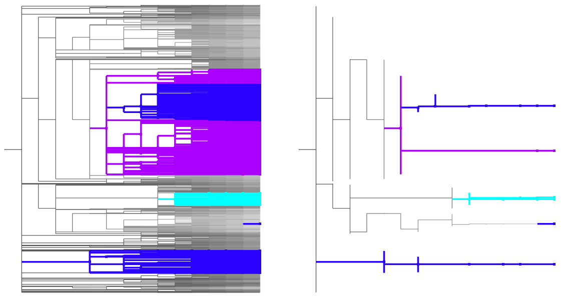


Figure 8: **Left:** A fully rendered tree scene with several colored marks. **Right:** The skeleton view of the same tree, with each marked group represented as a path from node to root.

5 Evaluating PRITree

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.

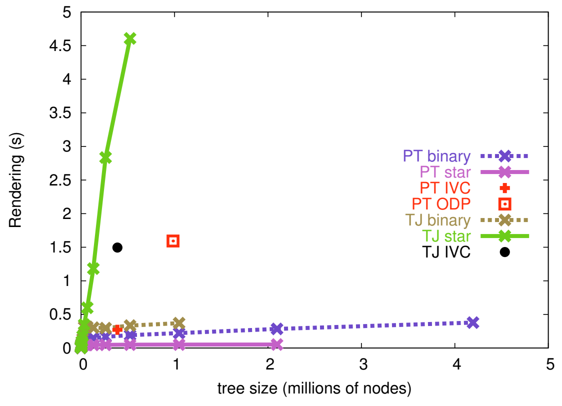
First, we compare the performance of both applications with respect to rendering a series of synthetic and large, real-world datasets. Our analysis of both total scene rendering time and memory consumption shows that we do not lose performance by switching from application-specific algorithms to the 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) [11], each with over 190,000 nodes; and two Open Directory Project categorization trees (ODP) [10], from March and June 2004, each with over 480,000 nodes.

5.1 Results

The top of Figure 9 shows that both TJ and PT achieve near-constant rendering performance, except for the linear cost of star tree rendering with TJ. TJ performs poorly with bushy trees, since when the root node is larger than one pixel, TJ will draw all of its children. The star tree is the worst possible case for bushiness, but the IVC comparison in TJ is considerably more expensive than the binary tree curve for the same reason. In PT, ODP requires four times longer to render than a similarly sized binary tree, because the pair of ODP trees has over 30,000 nodes marked as different. Therefore, PT must render many more nodes to provide guaranteed visibility of many marks. Since there are relatively few local differences, marked group look-up and rendering is not a huge cost for IVC, when compared to ODP.

The bottom of Figure 9 is a detail view showing the faster, sub-second rendering times for the rest of the datasets. 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.



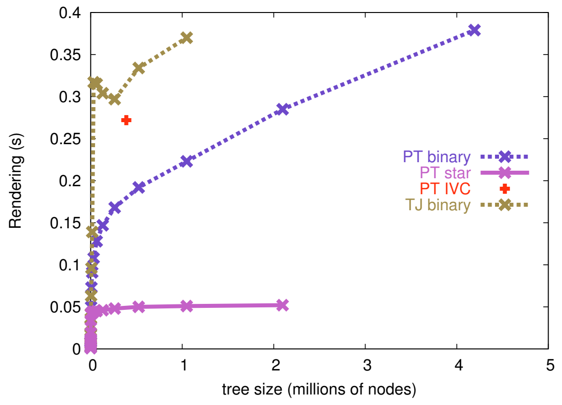


Figure 9: **Top:** Performance time for PRITree (PT) and TreeJuxtaposer (TJ) with several datasets. **Bottom:** detail of lower left corner.

The performance of binary trees in PT also becomes sub-linear after a threshold number of nodes. Again, when we render datasets with four times the number of leaves as vertical pixels, we only render that many more nodes for every doubling in size of our 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.

We note the inconsistency in the graph for binary trees in TJ: the rendering time for a binary tree of 262,143 nodes is faster than a tree of less than half its size, illustrating the overculling problem in TJ where large binary trees are incorrectly rendered with gaps.

Figure 9 also shows that the rendering time for IVC with PT is more than five times faster than TJ. 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, With PT, we again see the contest comparison closer to the binary tree curve, simply because it has much more internal structure than the star tree.

In Figure 10, 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 trees four times larger than TJ. For the contest comparison, PT is more than three times as efficient as TJ.

Finally, in Figure 11, 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

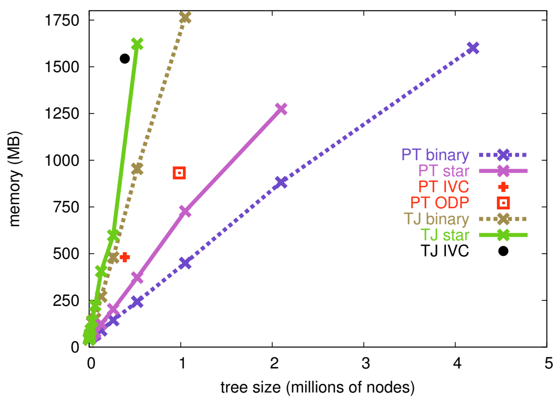


Figure 10: Memory performance for PRITree (PT) and TreeJuxtaposer (TJ) with several datasets.

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 11: The marking time performance, in seconds, for a classification tree from the InfoVis 2003 contest [11].

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 our memory footprint, leading to better scalability.

6 PRISeq

PRISeq, our genomic sequence-based PRISAD application, uses a layout identical to the layout proposed by SequenceJuxtaposer [13]. We position pre-aligned genomic sequences in the vertical direction, while we display the sequence data of nucleotides, which consists of A, C, G, and T, from left to right.

As discussed in Section 3.2.1, the PRISeq partitioning exploits the probability of vertical coherence in a column of nucleotides. To constrain the drawing time so it depends on the number of pixels and not on the dimensions of the dataset, we must, for the grid-based layout of PRISeq, cull in both directions. While the PRITree drawing strategy hinges on culling by careful selection along a leaf path, the PRISeq culling strategy is to 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.

6.1 Aggregating Columns

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. Our 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, the run-length encoding strategy described in Section 3.2.1 is used to minimize rendering time and save storage space.

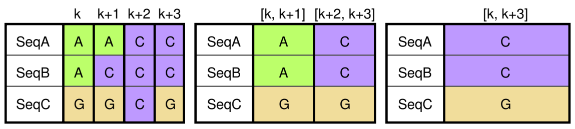


Figure 12: PRISeq recursively aggregates information for columns encompassed by split lines to determine which nucleotide color should be used for the representative object. **Left:** No aggregation is performed at the highest magnification since every nucleotide is visible. Rendering column $k+2$ requires 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, but we instead save time and space by lazy evaluation that fills a 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 $p = h*v$.

6.2 Performance

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 [13]. PS can render the same dataset in less than one half-second.

7 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 overdrawing 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`.

8 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] David Auber. Tulip - a huge graph visualization framework. In Petra Mutzel and Michael Jünger, editors, *Graph Drawing Software*, Mathematics and Visualization series, pages 105–126. Springer-Verlag, 2003.

[2] B. B. Bederson, J. Grosjean, and J. Meyer. Toolkit design for interactive structured graphics. *IEEE Trans. Software Engineering*, 30(8):535–546, 2004.

[3] Dale Beermann, Tamara Munzner, and Greg Humphreys. Scalable, robust visualization of very large trees. In *Proc. EuroVis 2005*, pages 37–44, 2005.

[4] Tamara Munzner *et al.* TreeJuxtaposer: Scalable tree comparison using Focus+Context with guaranteed visibility. *ACM Trans. on Graphics (Proc. SIGGRAPH 2003)*, 22(3):453–462, 2003.

[5] W.J. Kent *et al.* The human genome browser at UCSC. *Genome Research*, 12:996–1006, 2002. `genome.ucsc.edu/`.

[6] Jean-Daniel Fekete. The InfoVis Toolkit. In *Proc. IEEE Symposium on Information Visualization*, pages 167–174, 2004.

[7] Jean-Daniel Fekete and Catherine Plaisant. Interactive information visualization of a million items. In *Proc. IEEE Symposium on Information Visualization*, pages 117–124, 2002.

[8] Jeffrey Heer and Stuart K. Card. DOITrees revisited: scalable, space-constrained visualization of hiearchical data. In *Proc. Advanced Visual Interfaces (AVI '04)*, pages 421–424, 2004.

[9] T Hubbard *et al.* The Ensembl genome database project. *Nucleic Acids Research*, 30(1):38–41, 2002. `www.ensembl.org/`.

[10] OpenDirectoryProject, 2005. `dmoz.org/` .

[11] Catherine Plaisant and Jean-Daniel Fekete. InfoVis 2003 contest, 2003. `www.cs.umd.edu/hcil/iv03contest/`.

[12] James Slack. A partitioned rendering infrastructure for stable accordion navigation. Master's thesis, Univ. British Columbia, 2005.

[13] James Slack, Kristian Hildebrand, Tamara Munzner, and Katherine St. John. SequenceJuxtaposer: Fluid navigation for large-scale sequence comparison in context. *Proc. German Conference on Bioinformatics*, pages 37–42, 2004.

[14] Graham J. Wills. NicheWorks: Interactive Visualization of Very Large Graphs. *Graphical and Computational Statistics*, 8(2):190–212, 1999.

Appendix A PRITree traversal details

A.1 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 5. 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.

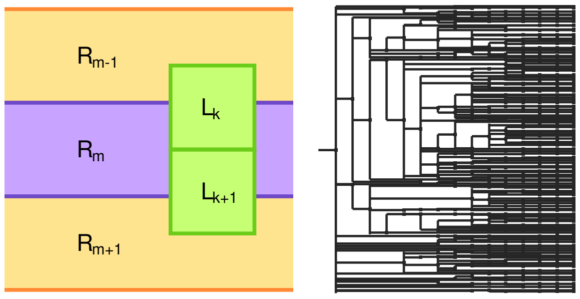


Figure 13: If τ is too large, then rendering gaps are visible throughout the tree topology. The adjacent leaf ranges L_k and L_{k+1} render a single leaf, which may be in pixels adjacent to pixel row R_m, rather than in row R_n itself which would be left blank.

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 13, 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 14, 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.

A.2 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

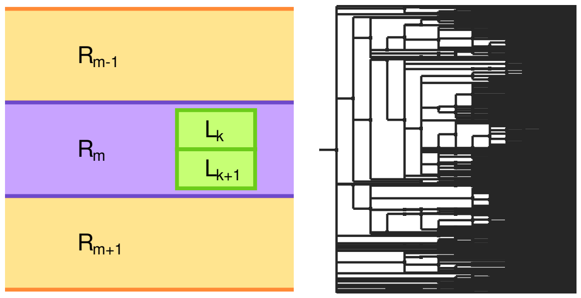


Figure 14: Restricting τ to less than one-half pixel prevents gaps in rendering the set of leaves at the expense of overdrawing. Other gaps in rendering are also prevented by our tree traversal.

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 [3].

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 4.1, 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 with of a pixel:

$$\psi \geq \tau \quad \rightarrow \quad \psi - \tau \geq 0 \tag{1}$$
$$\tau + \psi < p/2 \quad \rightarrow \quad p/2 - \tau - \psi > 0 \tag{2}$$
$$p/2 - 2\tau > 0 \quad \rightarrow \quad \tau < p/4 \tag{3}$$
$$\textit{maximize } \tau \quad \rightarrow \quad \tau = p/4 \rightarrow \psi > p/4 \tag{4}$$

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.