Improving Visualization Interactivity in Java

Christopher E. Weaver and Miron Livny

Computer Science Department
University of Wisconsin - Madison
1210 West Dayton Street
Madison, WI 53706 USA

ABSTRACT

An ongoing challenge for visualization researchers is to create tightly coupled dynamic query interfaces that respond quickly and smoothly to user interaction, regardless of the amount of data or the complexity of the display. Our solution is to follow a simple philosophy: during user activity, maintain the quality of visual *context* at the expense of visual *content* by reducing the impact of expensive visualization operations (query, render, blit) on interactivity. Our interactive visualization strategy splits finite computational resources by giving the user interface whatever resources it needs to maintain interactivity, and giving the data system whatever remains; user interface optimizations increase resources left for the data system, and data system optimizations make the best use of leftover resources. In this paper we describe three techniques that implement this strategy. First, we use a family of techniques called *throttling* to moderate the execution of queries and renders. Second, we improve refresh speed by eliminating several unnecessary, graphically-intensive operations. Third, we preprocess and render data into bitmaps using an asynchronous imaging engine. Using these techniques, we have increased the speed and smoothness of interface navigation in our visualization framework even during expensive query and render operations.

Keywords: data exploration, dynamic query, image rendering, interactivity, resource management, tight coupling, visualization

1. INTRODUCTION

An ongoing challenge for visualization researchers is to create tightly coupled dynamic query interfaces that respond quickly and smoothly to user interaction. Users rarely tolerate delays longer than 100ms between interaction and response.[1] In this short time, the interface must access, filter, transform, and render data into multiple views. Whether evaluating a hypothesis or simply exploring, a data analyst must be able to navigate efficiently between virtual locations in a visualization interface. Increased interface activity suggests an increased willingness to sacrifice data detail for speed of navigation.

Imagine driving over a scenic mountain pass. When the car is parked – perhaps at a scenic overlook – there is plenty of time to take in the vista in full detail. At full speed, however, the passing view is just a blur; enjoying the surroundings becomes subordinate to reaching the destination. Moderate speeds are a pleasant compromise, allowing some enjoyment of the scenery while making progress over the pass. The steering wheel, gas pedal, and brakes respond quickly and accurately regardless of speed. Despite the blur of the terrain at high speeds, the road and signs along it are always easy to follow.

Visualize the same drive in a different car. Pulling out of the overlook, the steering wheel, gas pedal, and brakes respond acceptably. As the car's speed increases, the wheel seems to lock up occasionally, the engine seems sluggish even with a lot of gas, and the brakes are soft. Through the windows, the passing terrain appears intermittantly, as if in a slide show. By the time the car reaches full speed, the wheel locks up for minutes at a time. The gas pedal sticks to the floor. The brakes engage hard by themselves, seemingly at random. The view out the windows, if any, rarely changes. Luckily, everything returns to normal when the car stops suddenly at the bottom of a cliff.

As unrealistic as the second scenario is for cars (except, perhaps, to secret agents and gangsters!), it is an accurate metaphor for what a user typically experiences while driving visualization software: dynamic queries consume enough resources to retard the processing of the very interactions that initiate them. Worse, the occasional completed query is of little help to a user trying to find a quick route from point A to point B. The user loses control.

Futher author information: (Send correspondence to C.E.W.) Email: {weaver,miron}@cs.wisc.edu

In contrast, the first scenario suggests that as environmental changes accelerate, users expect graceful degradation of fine detail without loss of control. We thus approach the problem of visualization interactivity with a simple philosophy: during user activity, always maintain the quality of visual *context* at the occasional expense of visual *content*.

Our core interactive visualization strategy is threefold. First, split finite computational resources between user interface and data system. The user interface receives whatever resources it needs to maintain interactivity. The data system receives any remaining resources. Second, optimize the user interface to leave as many resources as possible for the data system. Third, optimize the data system to best use whatever resources are left over.

Correspondingly, we have been able to increase the speed and smoothness of interface navigation in our Java-based visualization framework in three ways. First, we use a family of techniques called *throttling* to reduce the impact of expensive imaging operations on interactivity. Second, we speed up the built-in screen refresh mechanism by eliminating several unnecessary, graphically-intensive operations. Third, we preprocess and render data into bitmaps (instances of `java.awt.Image`) using an asynchronous imaging engine with tile caching.

This paper proceeds as follows. In Sect. 2 we summarize salient aspects of our visualization linking framework and implementation. Throttling is presented in Sect. 3. Section 4 describes improvements to baseline graphics performance. Section 5 describes the imaging engine and its integration into the framework. In Sect. 6 we detail the measurement of performance and interactivity, and present results. In Sect. 7 we suggest that the improvements to interactivity outlined here are widely applicable to visualization software, and even to interface development in general. We briefly describe several research questions for future exploration in Sect. 8, then conclude in Sect. 9.

2. LINKING

The research framework and optimizations described in this paper are offshoots of ongoing work on DEVise.[2] DEVise differentiates itself from other visualization software by the ability to display large datasets across multiple views. Views are rendered by transforming data records into shapes through a *mapping* projection. By linking several views using *visual links*, *record links*, and *cursors*, users can create the visual equivalent of database queries.

Although the linking primitives in DEVise are extremely powerful, they are often hard for users to understand, particularly in the context of a complicated visualization. In fact, every DEVise linking primitive can be described as one or more ranges shared between views to determine what subset of the rendered data space to display. Based on this observation we developed a simple visualization linking mechanism called Live Properties.

Live Properties is a value-sharing mechanism for property-oriented widgets. Each *control* (widget) defines *live properties* which bind to *variables*. A variable contains a single object *value*. A variable can bind to any number of live properties, but a live property binds to at most one variable. Controls access and modify values through their live properties. Two controls are linked when both have at least one live property bound to a common variable. Whenever a variable is modified, all controls having live properties bound to it are notified by callback. In this way, controls communicate indirectly through values shared by their live properties.

Live Properties is equivalent to a Model-View-Controller (MVC) architecture[3] with many small models, but avoids common difficulties with callbacks[4] that arise during visualization interface construction by using only a single kind of callback. Unfortunately, Live Properties is compatible with JavaBeans[5] only for sharing design-time properties. For runtime properties – precisely those we're most interested in linking – JavaBeans inherits Java's complicated variety of callbacks (methods of classes that implement `java.awt.event.EventListener`) and corresponding set of complex data models.

Our implementation takes the common approach of providing a small but extensible library of views and auxiliary widgets (such as value sliders) for use by designers. In particular, scatterplot views, color gradient views, axis range controls, and view "portal" selectors are common components of our own visualizations, such as the one shown in Fig. 1.

Although our view components are themselves not terribly noteworthy, using Live Properties to link them is. Building visualizations this way promises to be substantially easier than MVC-oriented and custom linking approaches, whether by type-and-compile programming, visual programming, or an integrated runtime layout mode (such as in DEVise). Visualization sliders,[6] "exploded cube" views, and other compound constructions are similarly easy to construct and reuse.

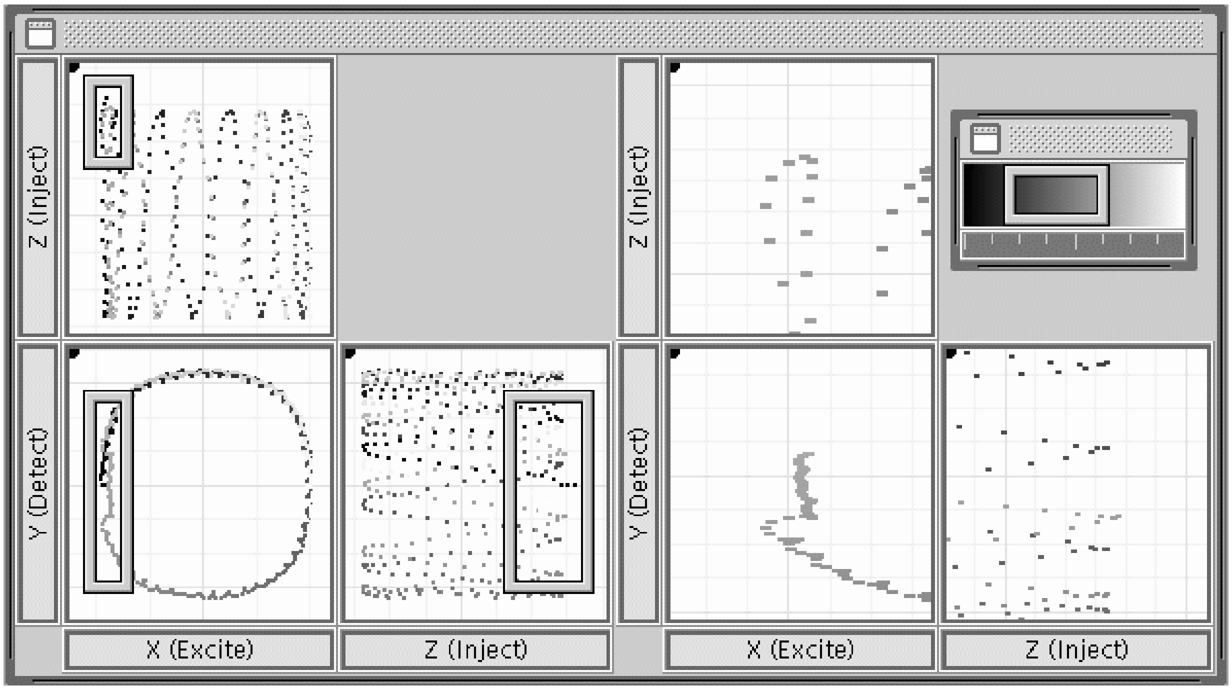

Figure 1. Screenshot of data visualized in four dimensions (a three-dimensional trajectory plus time). Portals in an "exploded cube" group of scatterplot views (left half) and in a color gradient view (top right) select the subset of data shown in a second "exploded cube" (right half). Axes label the spatial dimensions.

The flexibility of linking in our framework frequently encourages extensive, and even surprising, tight coupling between components. Our initial naive attempts to activate dynamic queries resulted in an unacceptable decrease in interactivity, and provided the motivation for the approaches we describe in the next three sections.

3. THROTTLING

We define *throttling* to be retardation of low-priority processes to assure timely completion of high-priority processes. In accord with our interaction philosophy, event handling and basic redraws are high-priority; blits, renders, and queries are of successively lower priority.

Throttling can be user-triggered or heuristically activated. Triggered throttling lets the user indicate processing preferences. During ongoing user activity – such as a mouse drag – useful throttles include:

- Elide blits. Views draw shaded rectangles in lieu of images.
- Delay renders. Views draw a stale image instead of rerendering.
- Simplify images. Rendered shapes are omitted or greeked (e.g. drawn as rectangles).

In our current implementation, the user can set the first two throttles globally or on each view independently. Each view has a popup menu to let the user toggle blit elision and delayed renders during active editing of view ranges.

Several characteristics of Java limited the possible designs of our heuristic throttling mechanism:

1. Invisible runtime differences between thread models (green and native)
2. A weak thread priority model

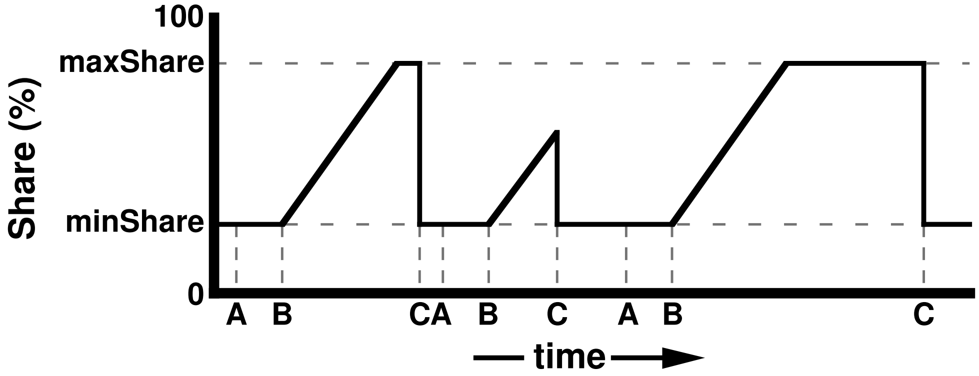

Figure 2. The effect of interactive events on the overall share of time available to all throttled threads. Each **A** marks a time when the throttler detects that the user event queue has emptied, followed after `eventDelay` by a time marked **B**. Each **C** marks the first detection time of a newly non-empty user event queue.

3. No per-thread virtual time
4. System time limited to one millisecond resolution

Moreover, Java lacks an API for customized thread scheduling beyond a rudimentary priority scheme. For these reasons, we designed a throttler based on cooperative suspension. Using tasks per second as a dynamic measure of work, throttled threads sleep voluntarily as needed to loosely guarantee some minimum interface responsiveness. The throttler suspension algorithm is encapsulated in a single `Throttler` class; in general the only alteration needed to throttle code is to call the `Throttler.throttle()` method whenever the code has completed its allotted number of tasks.

The `Throttler.throttle()` method returns a new allotted number of tasks based on the share of time available to the current thread, past work completed, and expected average task duration. The throttling algorithm maintains three data structures: the duration since the most recent interaction event (global), a windowed queue of past throttling events (global), and a work descriptor (per thread).

The share of time available for all throttled code is a linear function of the most recent interaction event time and the `minShare`, `maxShare`, and `eventDelay` constants (see Fig. 2):

$$share(t) = minShare + (maxShare - minShare) \times (t - eventDelay - lastEventTime) \tag{1}$$

The three constants correspond roughly to the minimum share of time per second available when the user is active, the maximum share available when the user is not, and the amount of time to wait in anticipation of possible future user activity. A simple additive priority scheme splits `share` between threads.

Whenever a thread throttles itself, a record of the time and tasks performed since its previous throttle is pushed onto the throttle event queue. Old events are popped off the queue until the queue fits into its time window. For each popped event, the descriptor of the thread that created it is adjusted by recalculating the number of tasks (`descriptor.tasks`) and window size (`descriptor.window`) specific to the thread. Three successive calculations determine the number of tasks allotted to the thread for its next work cycle (`window` and `factor` are constant):

$$slice = (share \times window - descriptor.window) \times factor \tag{2}$$

$$taskRate = \frac{descriptor.tasks}{descriptor.window} \tag{3}$$

$$tasks = slice \times taskRate \tag{4}$$

If `tasks` is negative, the thread sleeps for a period equal to `slice` before reducing the queue for another try.

We currently throttle only threads that perform rendering. The imaging engine currently uses in-memory tables as its data source, so no query throttling is necessary. Because throttling the Java event and painting threads would directly delay interaction, throttling of blits during paints would be counterproductive, and is not done. Section 4 describes ways to improve interactivity when the cost of blitting images is unavoidable.

4. DRAWING

The inherent slowness of Java as a language is exacerbated by poor graphics performance. In particular, the way the Java windowing library handles redrawing of visible components suffers from several problems:

- Double-buffering in Java can be toggled on and off as needed. However, Java provides no management of bitmaps, so it allocates and garbage-collects the global (screen) double-buffer repeatedly.
- By enabling double-buffering by default, Java redraws appear seamless but are slow. Moreover, bitmap-oriented interfaces end up performing redundant copies from bitmap to buffer to screen.
- Unlike most windowing systems, Java uses a single rectangle to represent the region of screen needing to be redrawn. Invalidating two distant, small areas – a common scenario in tightly-coupled dynamic query interfaces – can cause a large region to be repainted.

All three problems arise in the `javax.swing.RepaintManager` class that comes with Java (as a part of Swing with Java 1.1 or built into Java 1.2). Luckily, `RepaintManager` exposes an API designed explicitly for replacing it in a portable and transparent manner with a specialized or optimized implementation. At application startup, we replace the default `RepaintManager` with our own `BetterRepaintManager` class using this API. In `BetterRepaintManager` we have implemented successful solutions to the image allocation and double-buffering problems. Our attempts to address the single-rectangle invalidation problem met little success. The three efforts are described in the following sections.

4.1. Image Buffer Management

The `ImageManager` class implements a size-indexed, memory-bounded pool of bitmaps (instances of `java.awt.Image`). The global double-buffer and bitmaps for the imaging engine come from this pool. If the `ImageManager` does not contain a reference to a bitmap of the requested size, it allocates one. It is also possible to request images at least as large as a given size. The `BetterRepaintManager` uses such a request when the global double-buffer changes size.

Image management eliminates nearly all image reallocations, and thus avoids the 10-20% performance penalty due to allocation and garbage collection we experienced prior to its use. It should be noted that some operating systems manage image buffers in native interface code used by Java, thereby negating most of the advantage of the `ImageManager`. Linux is one example.

4.2. Aggressive Single-Buffering

Aggressive single-buffering is a form of the direct drawing optimization common in multimedia players and games. Like these applications, image-oriented visualization fails to receive the primary benefit of double-buffering – flicker-free display over multiple drawing operations – because it already performs its own buffering of multiple drawing operations by rendering into bitmaps. Simply turning off double-buffering is possible in Java on a global or per-component basis, but nearly always results in unacceptable flicker.

Aggressive single-buffering works as follows. Components turn double-buffering off during extended manipulations (such as mouse drags) that involve repeated bitmap blits. Flicker due to fills of overlapping areas within a component is avoided by careful design. Flicker due to background fills is avoided by making a component's ancestors transparent during single-buffering. There is no way to completely eliminate flicker; our own visualization components flicker terribly when forced to overlap. The only solution is to eliminate any overlap. Nevertheless, aggressive single-buffering is not as fragile as it sounds; the vast majority of visualization layouts contain no overlap, and infrequent visual garbage is cleaned up when double-buffering is reinstated.

Repaints of components that fill themselves with a single image are theoretically at least twice as fast when aggressively single-buffered as when double-buffered. Simple components in which double-buffering dominates the cost of painting see a much larger speedup. Examples of these two situations are the scatterplot view and the axis range control, respectively.

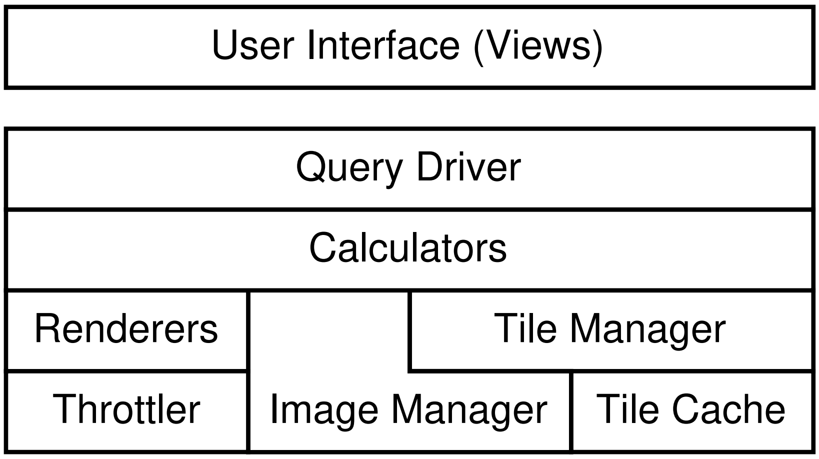

Figure 3. Software layers in the imaging engine.

4.3. Multi-Rectangle Invalidation

Although many native graphics libraries that run on the major operating systems support invalidation of non-rectangular screen regions, portability in Java compels a least-common-denominator approach. While an invalidation algorithm written around a simple rectangular representation is easy to write and has minimal overhead, it can cause unnecessary drawing. For example, the invalidation of two distant, small rectangles invalidates the large rectangle containing both. Furthermore, double-buffering exacerbates the cost of unnecessary paints of falsely invalidated regions.

We added to `BetterRepaintManager` an algorithm that maintains the invalid screen region as a list of rectangles. Only components which intersect one or more rectangles are repainted, with clipping performed as needed. From the layout of our example visualizations, we expected the improved algorithm to eliminate 20-50% of all repaints. Real improvement was less than 10%. The discrepancy is that the single-rectangle algorithm works well when distant regions are contained in unshared, opaque parents, a frequent occurrence in our visualizations. Even in the absence of this special case, Java's repaint thread almost always consumes repaint requests fast enough to clear the first small invalidation before the second one is added. Coupled with the fact that clipping is also limited to a single rectangle in Java – regardless of the invalidation algorithm – these observations lead us to conclude that our multi-rectangle algorithm offers no significant benefit over the single rectangle technique.

5. IMAGING

Storing precomputed data in ready-to-draw form within view code is a common strategy for coaxing interactive response from tightly coupled dynamic query interfaces. Unfortunately, the price for performance is loss of the numerous benefits of an architecture that cleanly segregates data model from interface.

Instead of taking the data to the view, why not take the view to the data? In our visualization framework, data is not drawn directly into views, but into bitmaps which are in turn copied into the view. We have designed and implemented an interface-independent, asynchronous imaging engine for high-performance rendering into bitmaps. Prefetching and caching of rendered images are supported by employing a bitmap tile manager and tile pool, much like demand-paged memory in an operating system.

The structure of the imaging engine is shown in Fig. 3. Whenever a view detects some change that affects its appearance – whether due to a window layout operation or tight coupling through links – it builds a query from its relevant live property values and issues the query to the imaging engine. The engine associates each view with a calculator, and places the query on the calculator's work queue for eventual consumption. The results of the completed query are returned to the view by callback.

Each query contains the size of the view, a pair of ranges for data-to-screen coordinate transformation, an instance of the `Renderer` interface, and additional values specific to the renderer. (Views can share renderers and

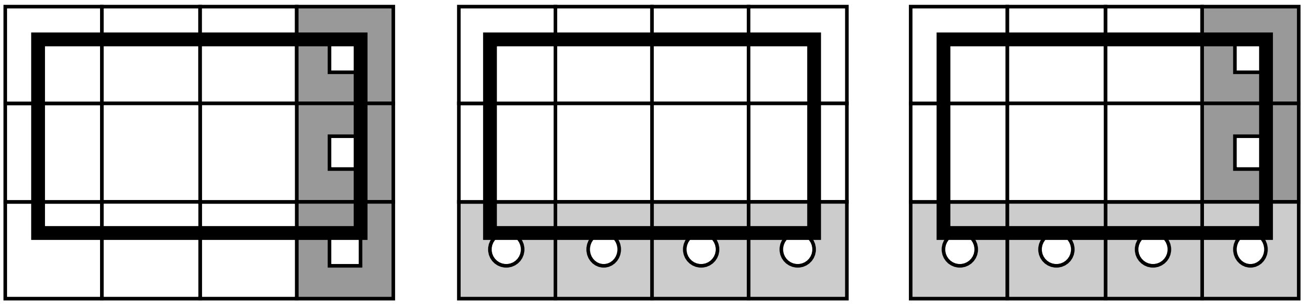

Figure 4. Response of the slide rendering algorithm to three common user manipulation patterns: column(s) for horizontal movements (left), row(s) for vertical movements (middle), and L-shaped regions split into two rectangles for oblique movements (right). In this example, each view (thick rectangle) overlaps twelve tiles.

switch between them at runtime.) The calculator allocates one or more images of appropriate size(s) then calls the renderer to draw into the graphics context of each one.

Our original implementation (called *basic*) rendered once per query into a single view-sized image. Despite modest gains in interactivity due to asynchronicity alone, single-image rendering fails to exploit substantial overlap of successive queries during common patterns of user manipulation. The tile manager exploits overlap by combining a tile cache (using an LRU replacement policy) with a hashtable for associative tile lookup. A tile query and data space region together define a unique association key.

Three improved calculator algorithms, called *parts*, *whole*, and *slide*, use the tile manager to improve query speed. All three build the minimal 2-D array "quilt" of tiles that covers a view, then fill it with references to cached tiles. The algorithms differ in how they complete uncached areas of the quilt.

The *parts* algorithm treats each tile independently. For each uncached section of the quilt, a new tile is allocated, rendered, and cached for future use. *Parts* is simple to implement and requires no image allocations or copies beyond the tiles themselves. However, the overhead of multiple calls to the renderer can be prohibitive, as it is for our current scatterplot renderer which scans an unindexed table of data.

The problem is that the renderer is not *tile separable*. That is, it lacks the ability to efficiently exclude data or work irrelevant to the appearance of the tile being rendered. Tile separability is a special case of the principle of *data separability*[7] exploited by data flow architectures to combat memory and latency problems[8] that arise when visualizing large amounts of data. Although we do not address here how large amounts of data might efficiently flow into the imaging engine, we have found that clipping in Java's graphics system automatically provides a limited degree of tile separability.

The *whole* algorithm is a tile-oriented version of the basic algorithm. It renders once into a single image, then creates each uncached tile by copying the appropriate section of the image. Although *whole* must allocate a temporary image, rows and columns of cached tiles at the edges of the query reduce the size of the temporary image compared to the *basic* algorithm. However, the extra image copies can be costly.

The *slide* algorithm is a heuristic combination of the *parts* and *whole* algorithm that uses common patterns of user manipulation to balance the cost of image allocation, image copying, and multiple renders. Keyboard and mouse operations in views often lead to groups of uncached tiles in rows, columns, or L-shapes (see Fig. 4). The algorithm works as follows: identify the shape of the minimal sub-quilt of uncached tiles; if L-shaped, break into two rectangular shapes and recurse at most once; otherwise, run the *whole* algorithm on the sub-quilt. Although *slide* is theoretically susceptible to scattered uncached tiles lost to cache replacement, we have not experienced this in practice. The savings due to increased clipping and fewer image copies dominates the cost of an additional image allocation and render, as we show in Sect. 6.

6. PERFORMANCE

The essence of interactivity is the overall subjective experience of real users, doing real work, on whatever system is available. However, user adaptiveness means that differences in interactivity across testing conditions frequently result in different patterns of behavior, not just differences in the rate of work. For these reasons we adopted two goals for evaluating the performance of the techniques described in the previous three sections:

1. Determine the overall performance boost during operation as similar as possible to real interaction by users.
2. Measure interactivity under various visualization conditions, such as different interface complexities and different amounts of data.

For the same reasons we avoided several otherwise valid experimental approaches:

1. Modeled sequences of actions
2. Subsets of interactions (e.g. keyboard only)
3. Absolute performance across operating systems, virtual machines, or versions of Java

We measured interactivity using a "player piano" technique. During a real session, the visualization application logs user interactions. The session can be replayed from its log using an instrumented version of the application. During replay, a trace of the events of interest – paints, queries, calculations, and renders – are recorded to a second log. A second Java application creates a summary. Finally, the rendering, throttling, and drawing algorithms and algorithm parameters are varied over many replays. At several minutes per replay, we were limited to independent linear probes of the algorithmic parameters.

We applied the record/replay technique to four different visualizations of the same data. Two of the visualizations were a complex combination of views, data projections, and links. The other two consisted of a single view linked only to two axis range controls. One of each pair visualized a small dataset (512 records), while the other visualized a synthetic dataset (32K records) derived from the small one. Qualitative improvements for our techniques across the four sessions were substantially similar. Unless stated otherwise, the results that follow are for the complex, small data visualization, identical to the one shown in Fig. 1. Measurements were taken on a machine with a 233MHz processor and 128MB of memory, using Java 1.1.8 with a just-in-time compiler.

Although our performance comparison concentrated on the relative performance between improved and unimproved replay of the visualizations, we measured the baseline performance of several image operations using tight-loop benchmarks. Image allocation time was 1.0-1.1ms for a 128x128 pixel image, asymptotically linear with area at about 20-30 megapixels per second and with an overhead of 0.4-0.6ms per image. Image copies took 0.8-1.0ms at 128x128 pixels, again asymptotically linear with area at about 17-21 megapixels per second with an overhead of 0.3-0.7ms per image. Time to render shapes into image tiles was independent of both image size and shape type (rectangle or oval), taking 0.06-0.07ms per shape for a rate of 14-16 kiloshapes per second. Shapes not actually drawn due to clipping were rendered at 70-80 kiloshapes per second.

Differences across tile methods were substantial. The *basic* method had the fastest rendering time but much slower overall query time due to lack of caching. Even with caching, the *whole* method suffered slow queries by ignoring the cache during diagonal moves. As expected, the *parts* method was very fast on individual renders thanks to clipping, but very slow on queries because of the expensive overhead of rendering each tile separately. The *slide* method was four to eight times faster for queries overall despite slightly increased paint time over the *basic* method. For the simple, large data visualization, the *slide* method displayed stale content less frequently by completing a query for 33% of all requests, whereas the other methods topped out at 10%. The average query time was 980ms for the *slide* method, and 5000ms or more for the others (just under 27000ms for the *parts* method). Paint time varied from 60ms for the *basic* method to 76ms for the *slide* method. The average render time of the *parts* method (2240ms) was the lowest of the render times, but it performed many more renders.

Differences in tile size showed a large per-image overhead below 96x96 pixels, with a paint time of 60-70ms at larger sizes. The 128x128 size seems to be a good balance between poor prefetching at smaller tile sizes and overaggressize prefetching at very large tile sizes, particularly when the tile cache is small. At cache sizes below about 1.5MB (a size we suspect depends on the visualization layout and user interaction patterns), the time per query increased rapidly due to cache misses. Moreover, very small caches forced the *slide* tiling method to worst case performance due to scattering of cached tiles. Average query times slowly decreased from about 2000ms for a 1.5MB cache to about 1000ms for an 8MB cache. We found the latter size to be a lot to ask of a typical system already burdened with a Java virtual machine.

The primary difference in throttling effectiveness was simply whether the throttler was on or off. While paint time was the same between these two cases, rendering (and hence query) time increased by about 50% when the throttler was on, allowing more paints in exchange for fewer completed queries over the course of the replay. The result was that replaying the visualization took 5030s without throttling, but only 4300s with. This difference translates to increased interaction rate during real operation.

Changes to the `eventDelay`, `minShare`, and `maxSlice` throttler parameters had substantial performance implications at their extreme values. Throttling effectiveness dropped off for `minShare` above 35% but had diminishing returns below 15%; between these values the throttled threads can make progress with minimal impact on interactive threads. For `eventDelay` less than 20ms, throttling effectiveness decreased by increasing allowed work too soon after the most recent interaction (thus before possible following ones). Throttling effectiveness was best for very short `maxSlice` of 10-20ms and dropped off above about 60ms. These results suggest setting `eventDelay` and `maxSlice` near the average delay between interaction events. The insignificance of many parameters – at least for our linear probes – indicated that simplification of the throttling algorithm is warranted.

It is our general experience that user interface optimizations are hard to evaluate in Java, even with statistical averaging. Our replay measurements contained large variations across runs even on quiet systems. First, thread context switches caused large positive errors in measured durations. Without access to per-thread virtual time within Java, it is extremely difficult to measure accurate time differences between events in one thread. A second source of variation is the inherently poor resolution of system time. Measuring the time of operations that take only a few milliseconds is hard when the absolute time resolution is one millisecond at best. As an example, two replays, different only by whether the double-buffering optimization was off or on, had average view painting times of 76ms and 56ms. An expected duration of about 15ms for the tile image blits which make up the bulk of the paint leaves a lot of time unexplained.

If nothing else, our experience confirmed the wisdom that good software design choices take precedence over localized optimization. Particularly for languages like Java that have heavily-threaded runtimes, local optimizations should be targeted at frequent, heavy operations. For our visualization implementation this has guided us to prefer design improvements for querying and rendering, but local optimization whenever image copying is involved.

Overall, we found that rendering had the largest impact on transient interactivity changes, and double-buffering had the largest impact on continuous interactivity. With double-buffering off, the average paint time dropped from 76ms to 56ms, the number of paints increased by 10% between replays, and the interactions per second increased from about 9 to over 12. In addition, the savings for each paint resulted in more completed queries (696 versus 599) for the unbuffered replay. For the best set of parameters we tried, even complex visualizations that display large amounts of data and cover a 1000x1000 pixel area of screen real estate allow interaction at 8-12 times per second with minimal reduction of processing power for non-interactive purposes, compared with 2-4 interactions per second for the same visualizations without our improvements.

7. INTERACTIVITY

With digital video, virtual reality, and games, visualization belongs to a software category in which continuous data transformation threatens to consume all system resources to the detriment of interaction. It seems likely that hardware advances will continue long after the human sensory limitations bound the resources needed by many of these applications. However, visualization and other applications may tax increasing hardware resources indefinitely as data-processing needs escalate. Three common strategies address poor interactivity when resources are scarce:

1. Disallow interaction during updates
2. Impose bounds on data volume or render complexity
3. Postpone updates during interaction pending implicit or explicit triggers

None of these solutions is entirely satisfactory. Even if display updates are asynchronous (and hence do not directly delay further interaction), interactive threads must compete for computational resources.

Reserving resources for interactive threads is a fourth general strategy for (loosely) guaranteeing interactivity while ensuring progress of other threads. Luckily, the resource needs of basic interaction (event handling and structural redraws) are bounded by human sensorimotor limitations, and are easily exceeded in most current systems. This is true even for our Java-based visualization implementation despite the inherent slowness and large memory footprint of Java.

We have found Java to be a delightful visualization development environment because of its convenient user interface and thread APIs. The rapid design-code-debug cycle is particularly welcome. However, Java has no inherent functional advantage over other environments (such as C++) for implementing the improvements described in this paper. In particular, we plan to incorporate Live Properties and the imaging engine into DEVise, which is written in C++ with a Tcl/Tk interface. (We do expect that thread package portability issues in C++ may necessitate substantial changes from the Java version.)

8. FUTURE WORK

Image-oriented visualization raises numerous research questions:

- What image/data caching policies can be developed to exploit user interaction patterns?
- What image/data prefetching policies can be developed to exploit user interaction patterns?
- How can images be fruitfully cached on continuous, non-spatial dimensions?
- How can large shapes be rendered efficiently over many tiles?
- How can users interact with the rendered data (as opposed to the image containing it)?
- What impact does asynchronous tile rendering have on the implementation and optimization of data queries?

In particular, the speed of data query operations is critical to rendering performance. Other project members are currently developing an efficient database engine to work with the imaging engine. A primary contribution of the database engine will be adaptive multiple query optimization which builds indexes on the spatial dimensions of viewed data, making per-tile rendering practical.

With query threads as well as rendering threads, assuring that low-priority threads make useful progress will be a challenge. User actions often invalidate incomplete work performed in response to prior actions. This situation offers two choices: complete the work or abandon it. The interface either receives stale content, or risks starvation as needed renders are repeatedly cancelled. However, some invalidation can be avoided in advance by coalescing enqueued work requests. Java automatically eliminates redundant input events and repaint requests. Similarly, the imaging engine filters out obsolete rendering requests.

9. CONCLUSION

Maintaining visualization interactivity boils down to the ubiquitous problem of managing finite computational resources. Although many visualization systems as a whole readily exceed the capabilities of typical current hardware – and appear likely to do so for the foreseeable future – their interactive components often have tractable requirements. Our interactive visualization strategy splits finite computational resources by giving the user interface whatever resources it needs to maintain interactivity, and giving the data system whatever remains. Optimizations to the user interface increase resources left for the data system. Optimizations to the data system make the best use of whatever resources are left over. In this paper we have described several techniques which when combined help maintain a minimum level of interface responsiveness under a wide variety of visualization workloads. By incorporating them into our own visualization interface framework, we have been able to create tightly coupled dynamic query visualization interfaces that respond quickly and smoothly to user interaction.

ACKNOWLEDGMENTS

This work was supported by National Science Foundation award IIS 9802882. Vuk Ercegovac developed and benchmarked an early prototype of the imaging engine.

REFERENCES

1. B. Shneiderman, "Dynamic queries for visual information seeking," *IEEE Software 11* **6** , pp. 70–77, 1994.
2. M. Livny, R. Ramakrishnan, K. Beyer, G. Chen, D. Donjerkovic, S. Lawande, J. Myllymaki, and K. Wenger, "DEVise: Integrated querying and visualization of large datasets," in *Proceedings of SIGMOD'97*, pp. 301–312, ACM, New York, 1997.
3. G. E. Krasner and S. T. Pope, "A cookbook for using the Model-View-Controller interface paradigm in Smalltalk-80," *Journal of Object-Oriented Programming 1* **3**, pp. 26–49, 1988.
4. B. A. Myers, "Separating application code from toolkits: Eliminating the spaghetti of call-backs," in *Proceedings of UIST'91*, pp. 211–220, ACM, New York, 1991.
5. *JavaBeans*, G. Hamilton, ed., Sun Microsystems, Inc., 1997.
6. S. Eick, "Data visualization sliders," in *Proceedings of UIST'94*, pp. 119–120, ACM, New York, 1994.
7. C. C. Law, W. J. Schroeder, K. M. Martin, and J. Tempkin, "A multi-threaded streaming pipeline architecture for large structured data sets," in *Proceedings of IEEE Vis'99*, 1999.
8. M. Cox and D. Ellsworth, "Managing big data for scientific visualization," in ACM Siggraph '97 Course #4 *Exploring gigabyte datasets in real-time: Algorithms, data management, and time-critical design*, 1997.