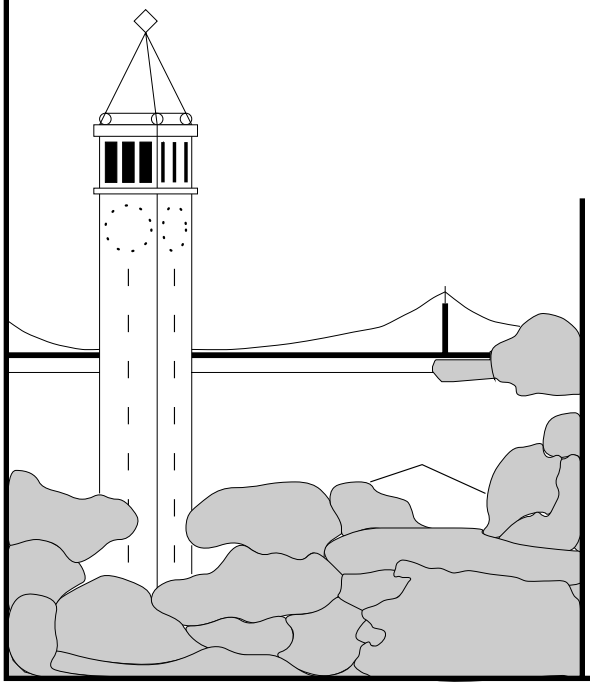
Interaction-based Rendering Optimization in Sketch-based User Interfaces

Li, Yang and Landay, James A.



Report No. UCB/CSD-3-1248

June 2003

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Interaction-based Rendering Optimization in Sketch-based User Interfaces[1]

Yang Li and James A. Landay
Group for User Interface Research, Computer Science Division
University of California, Berkeley
Berkeley, CA 94720-1776 USA
{yangli, landay}@cs.berkeley.edu

ABSTRACT
We describe two mechanisms, rendering task scheduling and the render cache, used to optimize graphics rendering in a scenegraph to provide continuous visual feedback and high interactivity in large-scale sketch-based user interfaces. We have implemented these mechanisms in SATIN, a toolkit to support development of sketch-based user interfaces. Our experiments with DENIM, an early-stage web site design tool built with SATIN, show that our changes significantly improve performance.

Keywords
Sketch-based user interface, scenegraph, visual feedback, render scheduling, render cache, SATIN, DENIM

INTRODUCTION
Sketch-based user interfaces allow natural interaction by freeform sketching. They typically involve the semantic processing and rendering of many graphical objects during interaction, e.g., freeform strokes. One distinct feature of sketch-based user interfaces is continuous interaction, which requires continuous visual feedback. This usually involves complicated transformation of graphical objects and heavy rendering, which impedes the responsiveness of interactive systems.

One useful way to organize graphical objects in a sketch-based user interface is to use a scenegraph [4]. SATIN [2] is a Java-based toolkit for developing sketch-based user interfaces, which employs a scenegraph to organize freeform graphical objects, such as strokes and patches. SATIN applications usually have large, complicated scenegraphs. For example, a typical web site design in DENIM [3], a web design application written with SATIN, usually has a scenegraph of

[1] This work was supported by the National Science Foundation under Grant No. IIS-0205644.

thousands of nodes (see Figure 1). An interaction action usually requires traversing the scenegraph and applying geometric transformations to many nodes. Moreover, a change to one node can trigger rendering requests throughout the scenegraph. All of this hinders prompt feedback and seriously affects the usability of the application.

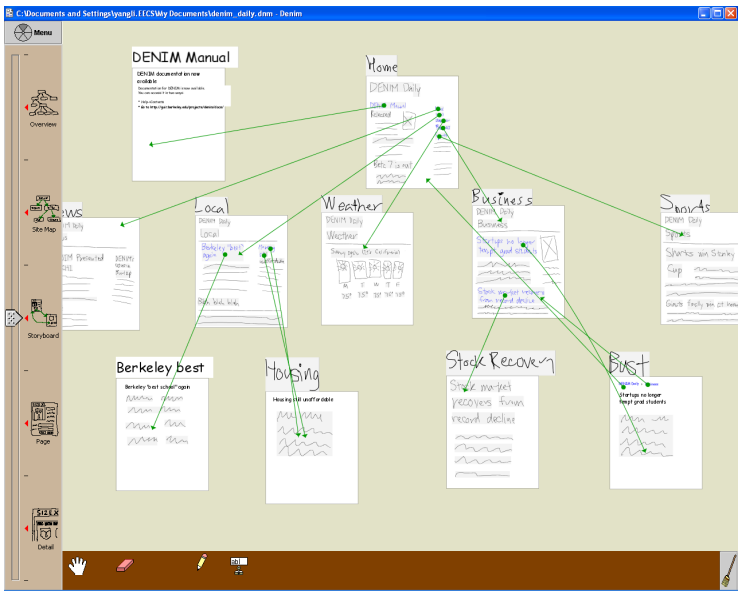


Figure 1: A typical web site design in DENIM, which has 1088 graphical objects in its scenegraph.

To improve the performance and usability of sketch-based user interfaces, we have developed techniques that optimize rendering in two ways. We first prioritize render requests to the system and delay render tasks that do not need to be done immediately. Then, we use the render cache to fulfill the render tasks. We have extended SATIN to take advantage of both of these techniques.

In the following sections, we first discuss related work. Then we analyze the interaction attributes of sketch-based user interfaces, which is followed by a discussion of render scheduling and the render cache. After this, we briefly describe some implementation details, usability feedback and regression tests. We finish with a discussion and conclusion.

RELATED WORK

We have employed the concepts of task scheduling and a render cache and applied them to sketch-based user interfaces. The render cache is an effective

solution to improving rendering performance of 3D graphics, which are computationally intensive [5], but it has not been widely adopted in 2D graphics.

Task scheduling and computational load analyses are often used to average out load and improve efficiency in real-time or multitasking systems [6]. However, no research has used these techniques to address the needs of interactive systems, either from the point of view of the computer or the user. It is necessary to optimize rendering by considering both machines and humans.

INTERACTION ANALYSIS OF SKETCH-BASED UIs

Sketch-based user interfaces based on the pen-and-paper metaphor allow people to express their ideas in a natural manner while providing electronic support. However, they require much higher levels of interactivity than traditional user interfaces with discrete interactions. At the same time, lots of computation and feedback frequently occur while the user is sketching. The feedback usually requires many freeform graphical objects to be rendered and simultaneously transformed, such as being moved or smoothed out.

Feedback rendering in a SATIN-based application requires frequent traversals of the scenegraph and rendering graphical objects in detail. This is because scenegraph-based systems collect rendering attributes and transformations by traversing the graph and then deliver them to graphical objects. Synchronizing the feedback with the interaction at all times takes up lots of processing time and decreases the responsiveness of the application. However, different types of feedback have different priorities. Some feedback must always be displayed right away, such as ink being rendered at the moment the user draws with the pen. However, it may be possible to delay other feedback that requires heavy semantic processing and rendering so that it does not interfere with the user's interaction, e.g., when interpreting and transforming some simple strokes into a DENIM web page. Nevertheless, it should be guaranteed that feedback is always synchronized with the associated interaction action when there is a semantic dependency. For example, users should get feedback immediately after they delete objects.

In sketch-based interactions, humans conduct their thinking and sketching in an interleaved manner with a very short "time slice." Some heavy rendering tasks that are delayable can be carried out during the "thinking" period.

INTERACTION-BASED RENDER SCHEDULING

In our system, a rendering request has one of two priorities, low and high. When an application issues a rendering request, it is first queued up in a FIFO buffer. Then, if a high-priority request is pushed into the buffer, all of the requests in the buffer are pushed out, merged together, and executed. By merging render requests

in a queue, we can reduce redundant rendering requests and, thus, scenegraph traversals. See Figure 2.

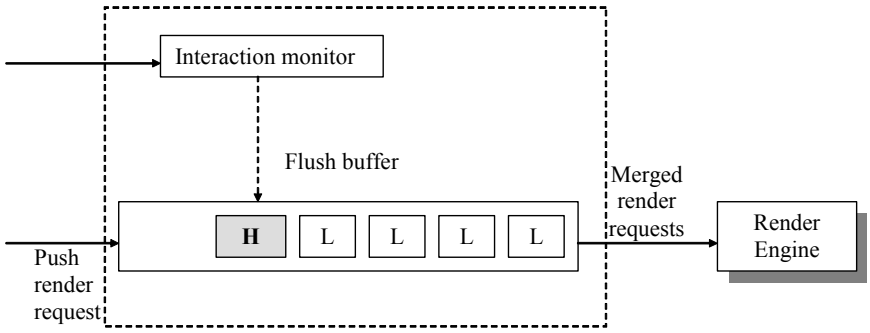


Figure 2: Render requests are buffered and merged before being executed. Low- and high-priority requests are labeled "L" and "H," respectively. The buffer is flushed in two ways: a high-priority request causes the buffer to flush all requests, or the interaction monitor flushes the buffer after it notices that there were no events during the event window.

The buffer can also be cleared if there are only low-priority requests in the buffer. If there are no user events during an *event window* starting from the user's last event, the buffer will be cleared after the window closes.

The next two subsections describe the event window and merging the render requests in more detail.

Adjusting the Event Window Length

In sketch-based user interfaces, most interactions consist of pen down, drag, and up events. We wish to find intervals between these events that are long enough to complete a render task, yet short enough so that it will not interfere with the user's actions. Figure 3 shows interaction intervals for three sketching tasks in DENIM. While most intervals are short, there are quite a few long intervals that indicate pauses for thinking and other reasons.

To predicate whether an interval is long enough to flush the buffer and process requests, we set up an *event window* as shown in Figure 4. If a pen event occurs in that window, we do not flush the buffer. Otherwise, we flush the buffer and process the requests.

The length of the window is adjusted using an exponential backoff algorithm. If a pen event, e.g., Evt_2 in Figure 4, occurs after the window but while processing render requests, the decay of the exponential function is adjusted so that, after this event, the window is longer. If there are no pen events during the window and

while processing render requests (e.g., Evt_3 in Figure 4 comes after rendering), the decay is adjusted so that the window is shorter. If a pen event like Evt_1 happens during the window, the window is kept unchanged.

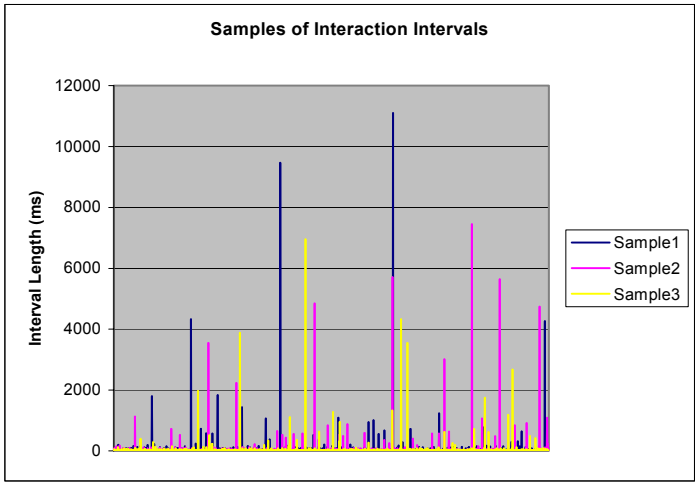


Figure 3. Analysis of intervals between pen events. The events can be pen down, drag, or up.

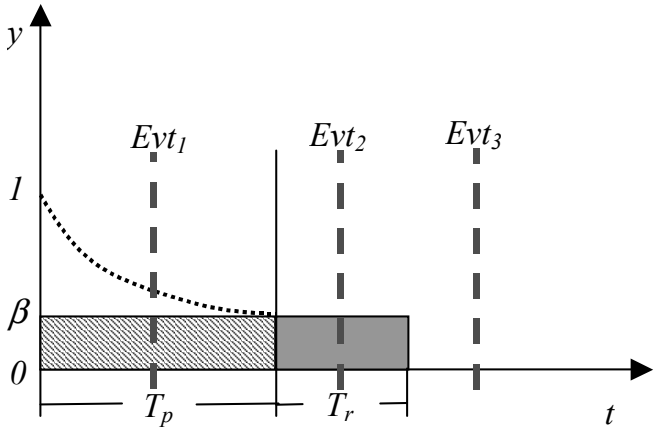


Figure 4: Adjustable event window. The length of the event window is T_p and T_r is the average render time. The dotted curve is the exponential decay function, where β is a constant and $0 < \beta < 1$.

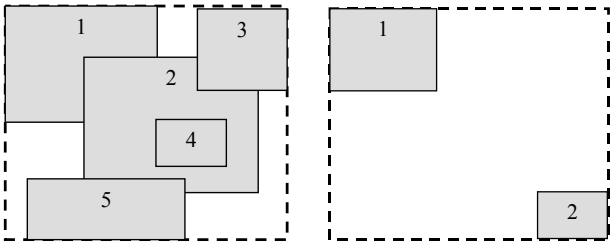
Merging Render Requests

To reduce the number of traversals and rendering over the scenegraph, all render requests are merged before being submitted to the render engine. To determine the effectiveness E_n of merging n requests, we sum up the areas of the rectangular

regions a_i that request i covers, then divide it by the area of the smallest rectangle A_n that overlaps all a_i. In other words:

$$E_n = \frac{\sum_{i=1}^{n} a_i}{A_n}$$

We consider merging n requests to be effective if $E_n \geq M$, where M is a constant that we have determined empirically as 0.3. Two examples of our effectiveness metric are shown in Figure 5.



(a) E_5 = 0.9326 (b) E_2 = 0.1856

Figure 5: Two examples of the merging effectiveness metric.

Here is our algorithm, which uses the effectiveness metric, for determining which requests to merge and send to the render engine. Note that E_0 is always effective by itself.

Given:

- q: the FIFO buffer containing the render requests
- q_i: the i^{th} item in the buffer, where $0 \leq i <$ length(q)
- r: the merged render request so far
- $E(i)$: the merging effectiveness metric for items 0 to i

```
while length(q) > 0:
    r := q0
    n := length(q)
    for i := 1 to n − 1:
        if E(i) < M:
                remove q0..qi-1 from q
                exit for
        merge qi into r
    submit r to render engine
```

RENDER CACHE

To process the merged render requests, parts of the scenegraph are rendered. We have augmented the scenegraph with the concept of a *render cache*. A render cache stores a pre-rendered bitmap image of the node to which it is attached and all of the node's children. SATIN simply draws this image instead of rendering the node and its descendants directly. This obviates the need to traverse the node's descendants, greatly improving performance.

When a node's render cache becomes invalid, all of the render caches of the node's ancestors also become invalid, as shown in Figure 6. Also, it is possible for a node not to have a render cache, in which case it will share the cache with the closest ancestor that has one.

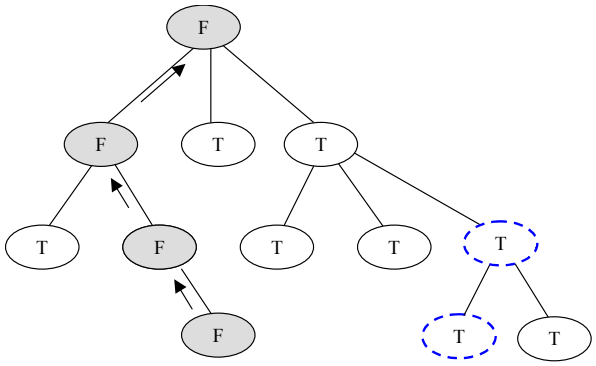


Figure 6: Cache tree and invalidity propagation. The two circles with dashed borders represent nodes without render caches.

SATIN also has support for building zoomable user interfaces [1]. Using render caches can greatly improve performance while zooming. However, creating the cache image for a node at the node's current zoom level will result in the node looking blurry while zooming in, since that is basically taking a low-resolution bitmap of the node and zooming in on it. Therefore, before zooming in, the render cache for the node is populated with the node rendered at a higher resolution to reduce the blurriness.

IMPLEMENTATION

We have implemented a render scheduler and merger in SATIN, and we have augmented graphical objects (which are nodes in SATIN's scenegraph) so that render caches can be attached to them.

Any graphical object can issue a render request by calling its method `damage(int flag, Rectangle2D area)`. The flag indicates scheduling

priority of the render request, either `DAMAGE_IDLE` for low-priority requests or `DAMAGE_NOW` for high-priority requests. The area specifies the region to be repainted, which is the bounding box of the graphical object by default. The render cache stores the bitmap image using Java's BufferedImage.

PERFORMANCE TESTS

We conducted regression performance tests on DENIM, the most sophisticated application built on SATIN. We tested four of the most common interaction tasks in DENIM: sketching, dragging, zooming, and panning. For each task, we created 12 different samples. Each sample was run 6 times in both DENIM 1.1, which uses the rendering scheduler and render cache, and DENIM 1.0, which does not include the new optimizations. Tests were performed by a combination of manual control and automatic mouse event replay via Java's Robot class. All tests were performed on an IBM ThinkPad laptop (700 MHz Pentium III) running Windows 2000 and Java 2 SDK 1.4.1, with 128 MB memory and an S3 display adapter with 8 MB memory.

As shown in Figure 7, we found that sketching with DENIM 1.1 is consistently faster than DENIM 1.0, an average of 2.26 times. However, the performance speedup of animated zooming, panning and dragging varies greatly from sample to sample. The more complicated the scenegraph or the dragged object is, the larger the performance speedup. Speedup of animated zooming varies from 1.06 to 2.23. Speedup of panning varies from 1.17 to 2.77. Speedup of dragging ranges from 2.41 to 18.62.

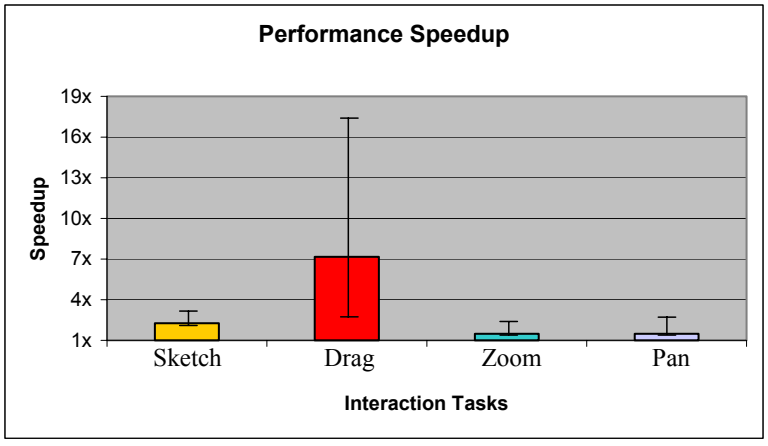


Figure 7: Performance speedup for 4 typical DENIM tasks.

These performance improvements enable DENIM to more effectively handle large-scale web site design and make an important step towards making it a practical tool.

DISCUSSION

Our own experience with DENIM 1.1 indicates that our algorithms do a good job in rendering high-priority requests immediately, and that low-priority requests are usually rendered when the user is not interacting with DENIM. DENIM 1.1 also seems more responsive, because heavyweight feedback no longer interferes with sketching.

We found that our interactions with DENIM 1.1 interfered with rendering tasks less and less often, the more we used it. One reason is that the length of the event window adapts over time. Perhaps another reason is that we adapted our behavior subconsciously.

In using a scenegraph-based toolkit like SATIN, it is hard for application developers to completely avoid issuing redundant render requests and they often try to merge render requests by themselves. By using the DAMAGE_IDLE flag, they can submit requests wherever they think it is required without considering redundancy.

To simplify the implementation we have used an empirically obtained average time T_r to estimate the load of render requests. In the future, we intend to refine our measurement of render loads. For example, we can use the render area of a request as a rough evaluation of task load.

CONCLUSION

We have implemented two mechanisms, render scheduling and the render cache, to improve the performance of SATIN, a toolkit for creating sketch-based, zoomable user interfaces. We have demonstrated that these methods have significantly improved the performance of DENIM, a web site design tool that is the most sophisticated application written with SATIN.

DENIM 1.1 along with a new version of SATIN, which includes the performance improvements described in this paper, are available at http://guir.berkeley.edu/denim.

ACKNOWLEDGEMENTS

We would like to thank James Lin for improving this paper. Thanks to James Lin and Jason Hong for helping us with SATIN and DENIM. Also thanks to James Lin, Marc Ringuette, Anoop Sinha and Zhiwei Guan for helpful discussions on

this topic. This work was supported by the National Science Foundation under Grant No. IIS-0205644.

REFERENCES

1. Bederson, B.B., J. Meyer, & L. Good. (2000). Jazz: An Extensible Zoomable User Interface Graphics Toolkit in Java. In Proceedings of the *ACM Symposium on User Interface Software and Technology: UIST 2000*. San Diego, CA. pp. 171–180, Nov. 5–8, 2000.
2. Hong, J.I. and J.A. Landay. SATIN: A Toolkit for Informal Ink-based Applications. In Proceedings of the *ACM Symposium on User Interface Software and Technology: UIST 2000*. San Diego, CA. pp. 63–72.
3. Lin, J., M.W. Newman, J.I. Hong, and J.A. Landay. DENIM: Finding a Tighter Fit Between Tools and Practice for Web Site Design. In *CHI Letters: Human Factors in Computing Systems, CHI 2000*, 2000. 2(1): pp. 510–517.
4. Strauss, P.S. An Object-Oriented 3D Graphics Toolkit. *ACM Computer Graphics*, 26(2). July 1992. pp. 341–349.
5. Walter, B., G. Drettakis, and S. Parker. Interactive Rendering Using the Render Cache. In the Proceedings of the *10th EG Workshop on Rendering: Rendering Techniques '99*. June 1999. Granada, Spain.
6. Wolski, R., Spring, N. and Hayes, J., Predicting the CPU Availability of Time-shared Unix Systems on the Computational Grid. In Proceedings of the 8th High Performance Distributed Computing Conference, August, 1999.