

Gliimpse: Animating from Markup Code to Rendered Documents and Vice-Versa

Pierre Dragicevic, Stéphane Huot, Fanny Chevalier

▶ **To cite this version:**

Pierre Dragicevic, Stéphane Huot, Fanny Chevalier. Gliimpse: Animating from Markup Code to Rendered Documents and Vice-Versa. UIST 2011 - 24th ACM Symposium on User Interface Software and Technology, ACM SIGCHI, Oct 2011, Santa Barbara, CA, United States. inria-00626259

HAL Id: inria-00626259
https://inria.hal.science/inria-00626259

Submitted on 24 Sep 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Gliimpse: Animating from Markup Code to Rendered Documents and Vice Versa

Pierre Dragicevic[1]
[1]INRIA
F-91405 Orsay, France
dragice@lri.fr

Stéphane Huot[2,1]
[2]LRI - Univ. Paris-Sud & CNRS
F-91405 Orsay, France
huot@lri.fr

Fanny Chevalier
OCAD University
Toronto, Canada
fchevalier@ocad.ca



Figure 1: Detail of an animation between this article and its source code.

ABSTRACT

We present a quick preview technique that smoothly transitions between document markup code and its visual rendering. This technique allows users to regularly check the code they are editing in-place, without leaving the text editor. This method can complement classical preview windows by offering rapid overviews of code-to-document mappings and leaving more screen real-estate. We discuss the design and implementation of our technique.

ACM Classification: H5.2 [Information interfaces and presentation]: User Interfaces. - Graphical user interfaces.

General terms: Design, Human Factors.

Keywords: Document editing, Animation, Markup code.

INTRODUCTION

Despite the popularity of WYSIWYG editors, authoring and editing styled documents using markup languages such as LaTeX and HTML is a widespread practice among computer literates and has recently been democratized by Wikis. Markup languages are widely used and advocated because they provide a clean separation between content and form, with benefits in terms of maintenability, portability, visual consistency, predictability, expressive power and expert user performance [9]. Some users also find it easier to focus on the content without having to deal with the form — or even without having to *see* the form, as exemplified by the recent popularity of "dark room" word processors [4].

P. Dragicevic, S. Huot, F. Chevalier.
Gliimpse: Animating from Markup Code to Rendered Documents and Vice Versa.
In UIST '11: Proceedings of the 24th Symposium on User Interface Software and Technology, ACM, October 2011.
Authors Version

One difficulty with markup code is that, although it is human-readable, users cannot always predict the results with accuracy and need to regularly check if it is correct and produces expected results. This typically requires a series of actions to re-render the document and display it in a separate environment from the text editor. This causes disruptions of the editing flow that can be daunting to beginners who need to continuously check their code and sometimes search for commands by trial-and-error. Even the most experienced users occasionally need to check commands they are unsure about and having to switch from the domain object (e.g., an article being written) to the tools (a markup interpreter or document viewer) [3] may cause them to lose their train of thought.

A great deal of effort has been spent in text editing environments to address these issues. Markup editors help users write correct code from the start by providing tools such as syntax highlighting and auto-completion. Since these tools will never eliminate the need for checking the final document, many efforts have also been spent at improving the document preview workflow. Most editing environments now provide shortcuts for re-rendering and refreshing the document into the viewer. Some of them further support rendering on-the-fly, and sometimes WYSIWYG selection and/or editing. Although these tools dramatically improved the usability of document markup languages, they still require users to deal with two separate windows.

In this article, we present an alternative, in-place document preview approach called Gliimpse. This technique lets users quickly "glimpse" into the rendered document by having the markup code smoothly transition to the rendered document upon a hot key press. It is based on the observation that markup code bears visual similarities with the document it produces, making it a good candidate for animated transitions [16, 7, 8]. Animated transitions have unique features that can potentially make them a useful complement to existing document markup editing tools. They are:

- *Implicit*. Users do not have to explicitly select the pieces of code or regions in the document they are interested in as in synchronized views: they only have to follow them visually during animated transitions.
- *Contextual*. Animated transitions provide context on regions that are not of immediate interest to users, and thus can possibly help them by giving quick overviews of where things go from the code to the document and providing opportunities for incidental discoveries (e.g., spotting a mistake elsewhere). The explanatory power of animations might further help beginners learn a new markup language.
- *Concise*. When the animation is invoked as a quasi-mode like in Gliimpse (using a hot key), users can quickly check the results of a formatting command they are unsure about and immediately come back to the code (the only bottleneck being rendering time). Our animations are fast but smooth enough to let users follow objects of interest and/or build a quick mental map of where pieces of code end up in the final document.
- *In-place*. The document is shown within the text editor itself, which saves screen real-estate and makes it possible to display more content relevant to the writing task. Gliimpse additionally uses a visual stabilization algorithm that tries to have the region of code currently edited stay in place.

After providing a brief overview of related work, we describe the basic features of Gliimpse. We then go into more details on the design and implementation issues behind our technique and finally discuss possible future work.

RELATED WORK

Gliimpse is related to two bodies of work: document editing and animated transitions. We briefly review them here.

Document Editing

Document markup languages and WYSIWYG systems are two approaches for editing documents, each with their own advantages and drawbacks. There has been a lot of effort in developing tools that combine the benefits of the two.

Examples of such efforts are integrated editing environments that combine a markup language editor with a WYSIWYG view. This idea was first introduced with the Lilac document editor [5]. Current examples include Dreamweaver and Firebug for HTML, and Instant Preview and Whizzy-TEX for TEX. These tools typically support synchronized highlighting (hovering an element in one view highlights the corresponding element in the other) and in some cases synchronized editing (changes in one view are reflected in the other). Such tools have been proved very valuable but since the use of two windows might sometimes come with disadvantages — for one thing, two windows take more screen real-estate than one — single-view approaches have also been explored.

Single-view preview approaches are either markup-oriented or WYSIWYG-oriented. Markup-oriented ones include text editors with WYSIWYG display features like WikEd, LyX and X-Symbol [12]. These pre-render symbol commands (e.g., displaying `\int` as $\int$) or use rich font attributes in their syntax highlighting that resemble the final document. Preview-latex [12] pushes the concept further by rendering code regions such as math formulaes in-place. As for WYSIWYG input features, toolbars and menus for inserting markup commands into the code are common in advanced code editors like emacs. Conversely, WYSIWYG-oriented editors exist that let users type markup code that is either inserted in the document and interpreted later like in Wikispaces or interpreted on-the-fly like in TeXmacs [17]. To further explore this rich design space, we propose an alternative in-place preview approach that uses animated transitions.



Figure 2: Animation of an HTML form.

Animated Transitions

Animated transitions are a type of animation that consist in showing a visual change in a smooth rather than an abrupt way. Their use in user interfaces has been advocated [7, 16] and studies have shown that they can help understand the spatial relationship between views and help users track changes in a variety of tasks (for a brief review, see [8]).

Animated transitions can however be hard to design and to implement. Animated text, in particular, has been used for various purposes such as expressing ideas and emotions [14] but there has been little work on when and how to support animated text transitions. Two exceptions are Chang et al's system for showing and hiding annotations in documents [6] and more recently Diffamation, a system for showing document edits over time [8]. The latter work has shown that animating text between revisions rather than abruptly flipping pages helps users navigate in edit histories.

Gliimpse targets a different application domain and also differs from the design and implementation standpoints. Chang et al's system merely animates the scale and position of text paragraphs. The Diffamation system introduces a richer animation language involving text insertions and deletions and paragraph reflow. Gliimpse goes a step further by supporting animation between documents having a different layout, different fonts, and possibly involving complex transitions such as a markup command changing into an image.

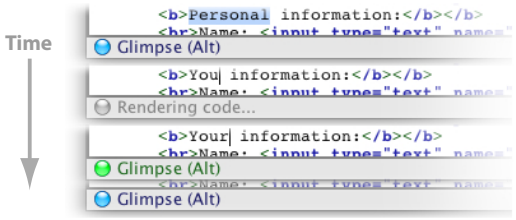


Figure 4: Gliimpse's status bar.

GLIIMPSE OVERVIEW

Gliimpse has been designed to be a quick, in-place preview tool for document markup languages. With Gliimpse, users stay in their text editing environment and can focus on their code, but are still able to check its rendering whenever they need. Whenever the user presses a hot key (we use Alt), the text editor switches to a document view with a fluid 1-second animation (Figures 1, 2). When she releases the hot key, the reverse animation is performed and she is back in her code.

The use of smooth animated transitions rather than abrupt view switches is motivated by previous work showing their benefits for navigating across text revisions [8]. Although fast and visually rich animated transitions can initially intimidate users, specific portions of text are easy to follow [8].

Gliimpse always transitions to the document region corresponding to the code currently in focus. The code focus is the area immediately surrounding the caret when it is visible, or the whole viewport content otherwise. The result is that the piece of code being edited will not move while the animation is performed, unless the document viewer has reached its scrolling limits. When the user moves the caret outside the viewport by scrolling elsewhere in the text editor and uses Gliimpse, the overall motion of the viewport is stabilized.

Since an actual document viewer is displayed upon completion of the animation, the user can scroll the rendered document while holding the hotkey. The viewport motion is then stabilized in the opposite direction at key release and Gliimpse animates back to a possibly new region of the code. Gliimpse can therefore be used as a navigation aid: the user just has to zoom the document out then gliimpse whenever she wants to jump to a different part of the document.

Because the layout of documents and markup code do not always match (e.g., with table cells and floating figures), visual objects can cross each other. To address this, Gliimpse has an option where objects that move with respect to the current focus follow curved paths. Figure 3a on the next page shows an example where a table (green square) is being edited. While gliimpsing, a larger table defined below in the code jumps above but goes around the focus (purple arrow). The user sees that the large table went to the wrong place and edits its placement options, which changes the focus to the large table and stabilizes it during the next gliimpse (Figure 3b).

This scenario illustrates how the overview and context provided by animations can help users make incidental discoveries and quickly build a mental map of where things go from the code to the document and vice versa. This didactic aspect of code / document animation can be exploited to help users learn a markup language, for example in Web tutorials.

Finally, when the code is edited, a background process re-renders the document and the animation. In our current prototype, this can last from 1/10 sec to more than a minute depending on document size (see the implementation section). A gray light in the Gliimpse status bar indicates that the process is working (Figure 4). After the user stops editing and once the animation is ready, the light switches to green, meaning that Gliimpse can be used. After a few seconds the light then switches to blue, meaning that a higher-quality, visually smooth version of the animation is ready to play.

DESIGN AND IMPLEMENTATION

We implemented the Gliimpse prototype in Java, with basic support for LaTeX, HTML, MediaWiki and RTF documents. We describe how we animate between these markup languages and the rendered documents.

Mapping the Code View with the Document View

To be able to compute animations, one needs to first generate a *code view* (i.e., a visual representation of the code as shown by the text editor), a *document view*, and retrieve a precise (character-level) mapping between the two views.

Figure 5 illustrates the problem and introduces notations that will be used in the rest of this section. It shows relationships between the markup text (T_0, bottom left of the Figure), the code view (V_0, top left), the document view (V_1, top right) and the document in raw text format (T_1, bottom right). We use the notation $X_A Y_B$ to denote a function that maps subsets of X_A to subsets of Y_B (thick black lines in the Figure):

- V_0V_1 (top) is the mapping between the two views. We assume V_0 and V_1 to be collections of glyphs and other graphical objects (possibly structured as a scene graph) with all information needed to render them individually (bounds, font, etc.). The mapping function V_0V_1 is the information we need in order to compute animations, which virtually no programming library or API directly provides.
- T_0V_0 (bottom left) is the mapping between the source code and its rendering in the text editor, which we assume to be a function that maps subsets of T_0 (i.e., collections of character indices) to subsets of V_0. This information is typically provided by the text editor's inspection methods or accessibility API.

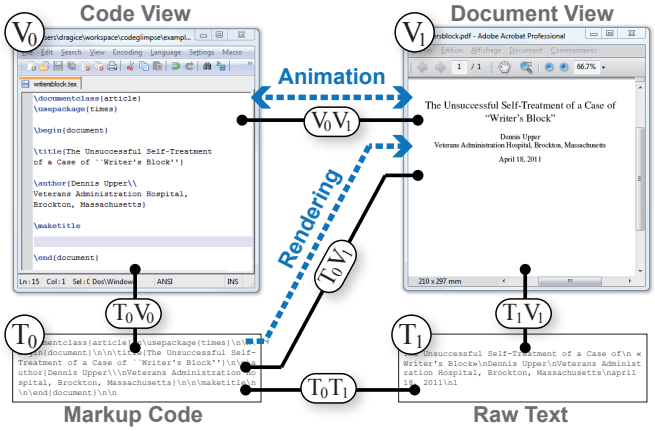


Figure 5: Mappings between the code (T_0), its view (V_0), the document view (V_1) and its text version (T_1).

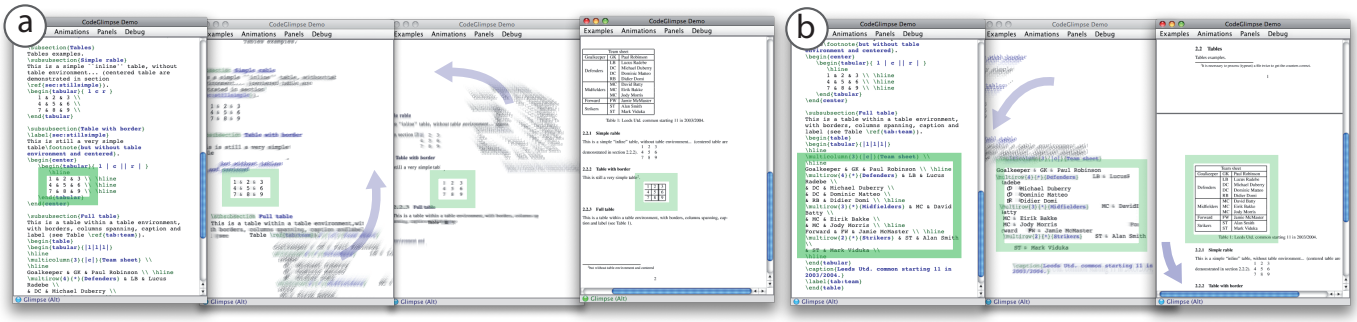


Figure 3: a) Motion of a large table (arrows) while another table is in focus (rectangle) ; b) The large table is now the focus.

- T_0V_1 (diagonal line) is the mapping between the source code and the document view. Markup language interpreters and renderers rarely maintain this information and when they do, it is often at a too coarse level of granularity to allow for precise animations. For example, TEX Source Specials and SyncTEX only provide mappings at the line level due to limitations of the TEX engine [13].
- T_1V_1 (bottom line) is the mapping between a raw text version of the document and the document view. This information is typically maintainted by document viewers to allow for text selection or to provide accessibility support.

The best way to reconstruct V_0V_1 is by computing $V_0V_1 = T_0V_0^{-1} \circ T_0V_1$. However, since getting T_0V_1 at the character level is hard in practice and because we initially just wanted to build a prototype, we chose to reconstruct V_0V_1 by computing $V_0V_1 = T_0V_0^{-1} \circ T_0T_1 \circ T_1V_1$. This approach is less robust but more flexible and makes it easier to plug Gliimpse into any interpreter and renderer available in Java.

In our current prototype, we obtain V_0 and T_0V_0 from the Java's text component (`JEditorPane`) inspection methods. The same Java component is used to render HTML, MediaWiki and RTF documents, and also provides V_1, T_1 and T_1V_1. LATEX source code is interpreted through a native call to `pdflatex` and the generated PDF file is rendered with a custom Java component based on the Sun PDF Renderer. We use the fonts from this renderer and the Apache PDFBox library to extract V_1, T_1 and T_1V_1. In all formats, non-character elements are mapped to white spaces in T_1.

The mapping T_0T_1 is built by cleaning up T_0 and T_1 and computing a diff between the two strings. We use Myer's diff algorithm [15], which supports deletions, insertions and moves. The cleaning up of T_0 essentially consists in stripping out markup elements and replacing non-character elements such as `<img*/>` with custom tags. The white spaces in T_1 that are mapped to non-character glyphs are replaced with the same tags. All string operations maintain a mapping with the original character indices. T_0T_1 can therefore be obtained by computing $T_0T_1 = T_0T_0^\star \circ T_0^\star T_1^\star \circ T_1 T_1^{\star -1}$, with $T_0^\star$ and $T_1^\star$ being the processed texts and $T_0^\star T_1^\star$ their diff.

For the formats we support, this method accurately rebuilds character mappings between markup code and the raw text version of simple documents. It is however not robust enough for a final product: duplicate text regions and complex mappings can defeat the diff algorithm and cause document regions to be either wrongly animated or not animated at all. We implemented a T_0T_1 mapping editor and used it to author the LATEX math tutorial scenario shown in the accompanying video. Fully automated approaches are arguably preferable and we hope that in-place preview techniques like Gliimpse will inspire the implementation of accurate, robust and usable APIs for mapping code views with rendered views.

Animating Between Fonts

In contrast with text animation techniques described in previous work [6, 14, 8], we need to animate between different fonts. Parametric typefaces [1] can linearly interpolate between glyphs but only if they share the exact same structure by design. Morphing between arbitrary shapes is a hard problem for which methods have been proposed [2] but they are computationally expensive.

For the purposes of animated transitions, we found that simply using alpha-blending to produce a dissolve effect yields excellent visual results. However, for this effect to work glyphs need to be properly aligned. Glyphs from different fonts can significantly differ in size even when the same font point size is used (Figure 6a). Aligning those glyphs based on their logical or geometrical bounds (Figure 6b,c) does not fully solve the problem. We therefore chose to refine glyph alignment using a pixel-based approach (Figure 6d).

We first collect all pairs of glyphs that need to be animated. For each pair, we draw one of the two glyphs off-screen with a fixed size (we use a point size of 30) and draw the other one on top using the method from Figure 6c. We compute the pixel color difference then vary the geometry of the second glyph (x, y, $width$, $height$ and $x\text{-}shear$ for italics) using a gradient descent scheme until a local minimum is reached. The result is cached and generalized to fonts of different sizes using linear interpolation. This method works best for within-character animations but can also polish animations such as changes in title capitalization.

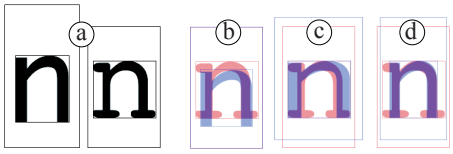


Figure 6: Alignment of two glyphs with the same font size (a), based on logical bounds (b), geometrical bounds (c) and pixels (d).

Animating the Document

Gliimpse's animation scheme is similar to the Diffamation system [8], with a few notable differences we outline here.

Object Interpolation. Like Diffamation, Gliimpse builds a parametric scenegraph that displays the initial document when rendered at $t = 0$ and the final document when rendered at $t = 1$. For $t \in]0, 1[$, the scenegraph essentially performs a linear interpolation of its nodes' bounding boxes.

We support 3 basic node types: *glyph transitions*, *non-glyph transitions* and *paragraph transitions*. Non-glyph transitions include transitions from a glyph to an image and between glyph groups. They are rendered by linearly interpolating the bounding boxes of the initial and final objects and alpha-blending them. Glyph transitions are computed the same way in addition to the alignment transformation previously described. Non-glyph objects are currently rendered as rectangles (see Figure 2). Paragraphs will be described later on.

When an object in V_0 or V_1 maps to nothing, we rapidly fade it in or out, in-place (as opposed to [8] where objects grow or shrink). Furthermore, we found that animating between groups of glyphs produces visual clutter or very wide characters (e.g., when animating from `\ref{fig:teaser}` to `1`), both of which are visually unpleasant. We therefore occlude those transitions at the middle of the animation by overlaying a rectangle with opacity $1 - 2 \cdot |t - 1/2|$ (see Figure 2). Finally, the scenegraph's root is animated by interpolating its background color and its position in the two viewports.

Paragraph Extraction. Like in [8], we group scenegraph nodes in paragraphs to be able to animate text reflow. However, we cannot rely on well-formed paragraph structures because some formats such as PDF produce V_1 content that merely consists in flat collections of glyphs. We therefore extract paragraphs from individual glyphs as follows:

We iterate over characters t^i of T_0 and at each step we compute the bounding box B^i of the object $T_0V_1(t^i)$ in V_1 and append it to the bounding box L^i of the current text line in V_1. At each step we test the following cases (for all our formats we use $vspace_{T0} = 2$ and $vspace_{V1} = 2$):

- If $\{t^{i-k}, \ldots, t^i\}$ contains only whitespaces among which $vspace_{T0}$ carriage returns, a new paragraph is created,
- if $T_0V_1(t^i) = \varnothing$ we proceed to the next index $i + 1$,
- if $B^i_{y0} > L^{i-1}_{y0} + vspace_{V1} \cdot L^{i-1}_{height}$ a new paragraph is created,
- if $B^i_{x0} < B^{i-1}_{x0}$ and $B^i_{y0} > L^{i-1}_{y0}$ a new line is started, in which case L^i is initialized to B^i,
- if $B^i_{x0} > B^{i-1}_{x0}$ and $B^i_{y1} > L^{i-1}_{y0}$ and $B^i_{y0} < L^{i-1}_{y1}$ the line continues and L^i is updated to $L^{i-1} \cup B^i$.
- otherwise, a new paragraph is created.

Once a paragraph is created for the indices i to j, we iterate over all subsets of $\{t^i, \ldots t^j\}$ of cardinality > 1 that map to content in V_1 and add them to the paragraph if their content lies within the paragraph bounds in V_1.

Text Reflow. Gliimpse animates text reflow within paragraphs in a way similar to [8]: depending on which path is shorter, a character can either follow a direct path or be "modulo-animated", i.e., move along its line and reach the edge to re-appear on the other side. In our algorithm, each character is animated independently and modulo-interpolations are limited to a full paragraph width. The visual effect on a paragraph with growing width would be that words on the second line move slowly to the left, those on the third line move similarly but faster, and so on until a line breaks in two pieces, one going quickly to the left and the other one slowly going up. Finally, our modulo metrics accounts for the fact that our paragraphs can have lines of different heights and these heights are linearly interpolated during the animation.

Scrolling stabilization. We stabilize horizontal and vertical scrolling between the code and the document viewports by minimizing the average motion of objects that are visible in the source viewport, as described in [8]. In addition, when the caret is visible, we use its position as the region of interest and stabilize the nearest object rather than the entire viewport. Stabilization is recomputed every time the user scrolls into the code, moves the caret, or scrolls into the document.

Curved Trajectories. In contrast with [8], we support diff move operations and chunks of text can therefore cross on the screen. To make these motions easier to understand and limit occlusions of the region of interest, an option allows objects to follow curved trajectories (see Figure 3). Our method, inspired from link drawing techniques in graphs [11], consists in having objects follow an arc and those moving in the opposite direction follow an arc oriented to the opposite side. More specifically, after scrolling stabilization we add to the absolute trajectory $P^{(t)}$ of each object (paragraph or isolated node) the vector $sin(\pi t)^k \cdot [r_x(P^{(1)}_y - P^{(0)}_y), r_y(P^{(1)}_x - P^{(0)}_x)]$. We use $k = 0.5$, $r_x = -0.5$ and $r_y = 0.05$.

Since arc radii are proportional to object motion, animations with no crossings will have close-to-straight trajectories (since they have been stabilized) but crossing objects will deviate from their path as if they tried to avoid each other. Even if this method does not guarantee the absence of overlaps, it presents the advantage of being context-free (every object ignores the position of others), which makes it simple and guarantees motion coherence (objects which are normally close will remain close). The asymmetry between the values we chose for r_x and r_y stems from fact that documents are structured in lines.

Playing Back and Recomputing Animations

The animated scenegraph is parented to a Java layered container that also contains the document viewer and the text editor. When the hot key is pressed, the scenegraph is set to $t = 0$, brought to the top and the animation starts. When the scenegraph shows the final document at $t = 1$ the actual document viewer is brought to the top. This transition is shown with a quick dissolve effect because the document view occasionally shows decorations that are not inspectable and hence not visible during the animation. The reverse sequence of operations is performed when the hot key is released.

We animate between $t = 0$ and 1 with a duration of 1 second, which has been shown to be appropriate for reasonably complex visual transitions and with a slow-in slow-out pacing, which has been shown to facilitate object tracking [10].

		Task a (s)	b (s)	c (s)	d (s)	e (s)	FPS (Hz)
HTML	1^{st}	0.21	0.12	0.04	0.03	0.29	75
	2^{nd}	0.21	0.07	0.02	0.01	0.01	89
LaTeX 1	1^{st}	1.90	0.28	0.41	0.06	0.64	36
	2^{nd}	0.97	0.26	0.28	0.04	0.02	40
LaTeX 2	1^{st}	3.72	37.0	8.68	10.5	19.5	12
	2^{nd}	2.68	41.3	8.53	10.5	0.62	13

Table 1: Task execution times for three documents.

Animations are updated on-the-fly by a scheduler that runs in a separate thread and skips unnecessary computations. For example, modifying the source code requires a) re-rendering the document, b) computing the T_0T_1 mapping, c) inspecting the views for T_0V_0 and T_1V_1, d) building the scenegraph, e) aligning new glyphs and f) stabilizing the views. However, when a view is resized only tasks c) to f) are performed, and when it is scrolled only task f) is done. After the animation is ready the scheduler runs an off-screen rendering task after which complex animations play back more fluidly with an optional motion blur effect (visible in the Figures).

Table 1 shows computation times on a PC with a 2.40GHz Intel Xeon processor for the tasks mentioned above (task f is negligible), as well as the animation frame rate. Figures are given for the first run (1^{st}) and after inserting a character in the code (2^{nd}). The first two examples are a short HTML file (700 characters) and LaTeX file (1900 characters, about 1 page). These computation times are adequate for interactive use but the third example, a 5-page LaTeX draft of this article (30,000 characters), shows that our current prototype does not scale up. The most expensive task is the code/document mapping task, which uses heavy regexp searches and hashtables. This operation can be optimized or avoided altogether with an API that provides T_0V_0 as previously discussed.

CONCLUSION AND FUTURE WORK

We presented Gliimpse, a quick in-place preview technique that smoothly transitions between markup code and its visual rendering. This technique is an alternative to classical preview tools that takes less screen real-estate and offers rapid overviews of code-to-document mappings.

More work is needed to identify the actual benefits of animations over well-established approaches such as synchronized views. We hypothesize that animations can save users time and effort because they involve *attentional* rather than *explicit* selection of regions of interest. This is however only a conjecture that needs to be put to the test. And even in case animation helps, it is likely that synchronized views are more suited for some tasks and that both need to be supported.

Possible future extensions include local animated previews, integration with synchronized highlighting and editing, animation of series of code transformations (e.g., XSLT) and animation of non-textual markup documents like music sheets or vector graphics. Furthermore, Gliimpse is currently only a prototype and a more robust version is necessary for users to be able to try it on real writing tasks.

ACKNOWLEDGEMENTS

We thank Jean-Daniel Fekete for insightful discussions.

REFERENCES

1. Adobe. Designing multiple master typefaces, 1995. http://partners.adobe.com/public/developer/en/font/5091.Design_MM_Fonts.pdf.
2. M. Alexa, D. Cohen-or, and D. Levin. As-rigid-as-possible shape interpolation. In *Annual Conference on Computer Graphics*, 157–164, 2000.
3. M. Beaudouin-Lafon. Instrumental interaction: an interaction model for designing post-wimp user interfaces. In *Proc. CHI '00*, 446–453. ACM.
4. S. bin Ali. 20 fantastic full screen text editor for distraction free writing, 2009. http://www.techmalaya.com/2009/02/07/full-screen-text-editor-blogger/.
5. K. Brooks. Lilac: a two-view document editor. *Computer*, 24(6):7 –19, jun 1991.
6. B.-W. Chang, J. D. Mackinlay, P. T. Zellweger, and T. Igarashi. A negotiation architecture for fluid documents. In *Proc. UIST '98*, 123–132. ACM.
7. B.-W. Chang and D. Ungar. Animation: from cartoons to the user interface. In *Proc. UIST '93*, 45–55. ACM.
8. F. Chevalier, P. Dragicevic, A. Bezerianos, and J.-D. Fekete. Using text animated transitions to support navigation in document histories. In *Proc. CHI '10*, 683–692. ACM.
9. J. H. Coombs, A. H. Renear, and S. J. DeRose. Markup systems and the future of scholarly text processing. *Commun. ACM*, 30:933–947, Nov. 1987.
10. P. Dragicevic, A. Bezerianos, W. Javed, N. Elmqvist, and J.-D. Fekete. Temporal distortion for animated transitions. In *Proc. CHI '11*, 2009–2018. ACM.
11. J.-D. Fekete, D. Wang, N. Dang, and C. Plaisant. Overlaying graph links on treemaps. In *Proc. Infovis '03 (demo)*, 2003.
12. D. Kastrup. Revisiting WYSIWYG paradigms for authoring latex, 2002. http://www.tug.org/TUGboat/tb23-1/kastrup.pdf.
13. J. Laurens. Direct and reverse synchronization with synctex. *TUGboat*, 29(3), 2008.
14. J. C. Lee, J. Forlizzi, and S. E. Hudson. The kinetic typography engine: an extensible system for animating expressive text. In *Proc. UIST '02*, 81–90. ACM.
15. E. Myers. An $o(nd)$ difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986.
16. G. G. Robertson, J. D. Mackinlay, and S. K. Card. Cone trees: animated 3d visualizations of hierarchical information. In *Proc. CHI '91*, 189–194. ACM.
17. J. van der Hoeven. Gnu texmacs: A free, structured, wysiwyg and technical text editor. In *Actes du Congres GUTenberg*, volume 39-40, 39–50, 2001.