Nonlinear Revision Control for Images

Hsiang-Ting Chen
National Tsing Hua University

Li-Yi Wei
Microsoft Research

Chun-Fa Chang
National Taiwan Normal University

Abstract

Revision control is a vital component of digital project management and has been widely deployed for text files. Binary files, on the other hand, have received relatively less attention. This can be inconvenient for graphics applications that use a significant amount of binary data, such as images, videos, meshes, and animations. Existing strategies such as storing whole files for individual revisions or simple binary deltas could consume significant storage and obscure vital semantic information. We present a nonlinear revision control system for images, designed with the common digital editing and sketching workflows in mind. We use DAG (directed acyclic graph) as the core structure, with DAG nodes representing editing operations and DAG edges the corresponding spatial, temporal and semantic relationships. We visualize our DAG in RevG (revision graph), which provides not only as a meaningful display of the revision history but also an intuitive interface for common revision control operations such as review, replay, diff, addition, branching, merging, and conflict resolving. Beyond revision control, our system also facilitates artistic creation processes in common image editing and digital painting workflows. We have built a prototype system upon GIMP, an open source image editor, and demonstrate its effectiveness through formative user study and comparisons with alternative revision control systems.

CR Categories: I.3.4 [Computer Graphics]: Graphics Utilities—Graphics editors;

Keywords: revision control, images, nonlinear editing, interaction

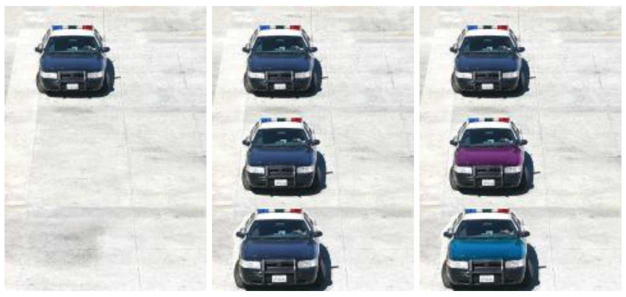
Links: DL PDF

1 Introduction

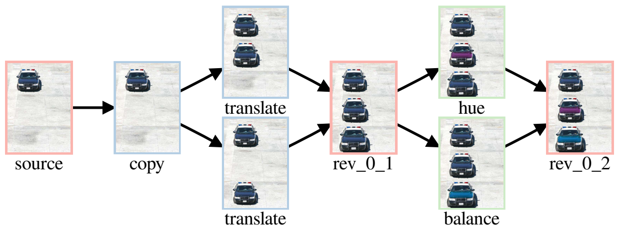
Revision control is an important component of digital content management [Estublier et al. 2005]. Popular revision control systems include CVS, Subversion, and Perforce, to name just a few. By storing file editing histories, revision control systems allow us to revert mistakes and review changes. Revision control systems also facilitate open-ended content creation [Shneiderman 2007] through mechanisms such as branching and merging.

So far, the development and deployment of revision control systems have been focused more on text than binary files. This is understandable, as text files tend to be more frequently used and revised, and it is easier to develop revision control mechanisms for them. (Simple line differencing already provides enough information for text files.) However, in many graphics projects, binary files, such as images, videos, meshes, and animations, can be frequently used and revised as well. Here the lack of revision control for binary files could cause several issues. Most existing general purpose revision systems employ a state-based model that stores the different revisions as individual files without any diff/delta information, thus bloating storage space and making it hard to deduce the changes between revisions. Even when deltas [Hunt et al. 1998] (or other low-level information like pixel-wise diff) are used, they usually lack sufficient high-level semantic information for reviewing, branching, merging, or visualization.

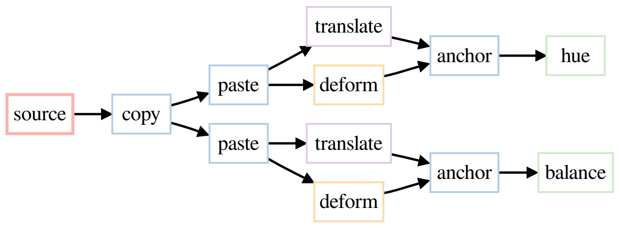
Such high level information can usually be recorded from live user actions with the relevant image editing software. The visualization and interaction design of such user action histories has long been a popular topic [Kurlander 1993; Klemmer et al. 2002; Heer et al. 2008; Su et al. 2009a; Grossman et al. 2010]. Nevertheless, the lack of a formal representation that depicts the comprehensive relationship (not only temporal but also spatial and semantic) between



(a) *source* **(b)** *rev_0_1* **(c)** *rev_0_2*



(d) *RevG*



(e) *DAG*

Figure 1: *Nonlinear revision control example. From the input image (a), we cloned the car twice with translation and perspective deformation (b) followed by modifying their colors (c). Our revision control system recorded and analyzed the actions into the DAG data structure as shown in (e). The DAG is our core representation for revision control but not directly visible to ordinary users due to potential complexity. Instead, we visualize the DAG through a graphical revision graph (RevG, shown in (d)) in our external UI. Users can interact with RevG and perform revision control functions. Node border colors denote the action types (Table 1) and paths delineate the action dependencies. In particular, parallel paths indicate operations that are semantically (e.g. translation and deformation) or spatially (e.g. coloring two individual cars) independent.*

image editing actions makes these approaches both inefficient and insufficient for revision control.

In this paper, we propose a nonlinear revision control system for images, designed with the common content creation workflows such as digital editing and sketching in mind. We maintain high-level and fine-granularity revision history by recording and consolidating user editing operations. The core idea of our system is a DAG (directed acyclic graph) data structure representing the nonlinear spatial, temporal, and semantic dependencies between these recorded image editing operations stored as DAG nodes. Such detailed information provides the necessary high level semantics for proper revision control compared to previous works where only low-level bitmap information is used. Built upon the DAG data structure, we provide the primary revision control commands including review, diff, addition, branch, merge, and conflict resolving. We present all these functionalities through a friendly user interface centered upon RevG (revision graph), a multi-resolution graphical revision history representation of the underlying DAG. In addition to core revision control, our system also facilitates open-ended content creation processes with non-linear editing and exploration [Kurlander 1993; Su et al. 2009a; Terry et al. 2004].

Based on our core DAG representation, we devise several algorithm innovations for both internal system implementation and external user interface design. We begin with on-the-fly action recording and DAG construction during user editing. With the recorded revision history, we are able to construct the DAG and filter it into a visually intuitive multi-resolution RevG representation. We devise mechanisms for automatic resolving and merging multiple revisions with potential conflicts, as well as a user interface that allows manual change and intervention of the automatically merged results. We also provide an image diff tool that can be particularly handy for visualization as prior schemes are primarily based on low level pixel information.

We have built a prototype system via GIMP, an open source image editor, through multiple iterations of user feedbacks during our design stages. We also demonstrate the effectiveness of our system through further user studies and comparisons with alternative revision control systems.

In summary, our paper has the following contributions:

- The idea of a nonlinear revision control system for images.
- The core DAG structure representing the revision history, upon which we build RevG, the multi-resolution revision graph, and various algorithm components for revision control.
- A prototype system built with GIMP for practical nonlinear revision control and an intuitive UI centered on RevG for common revision operations including addition, branching, merging, conflict resolving, diff, and non-linear replay.
- Additional applications of our system such as facilitating open-ended content creation and exploration.

2 Previous Work

Digital content management Digital content management refers to the general process of authoring, editing, collecting, publishing, and distributing digital content, for both text and binary data [Jacobsen et al. 2005]. Among the various components of digital content management, revision control remains one of the most important; see [Estublier et al. 2005] for a detailed survey.

Existing revision control mechanisms focus mainly on text rather than binary files as it is easier to deduce the changes via either low level (e.g. line diff [Hunt et al. 1998]) or high level (e.g. programming language syntax [Jackson and Ladd 1994]) information. However, for binary files, the prevailing methods store either the complete files for individual revisions or their crude binary differences [Hunt et al. 1998]. Both of these methods can consume significant storage, and more importantly lack relevant high level semantic information. These issues could hamper the adoption of revision control systems in managing graphics assets. Judging by the success of graphics content management systems (e.g. Alienbrain), such demands obviously exist, but to our knowledge no suitable revision control mechanism exists for binary graphics assets. The goal of our system is to fill this gap, allowing easy revision control for images, the most commonly used graphics data type.

Graphical history There exists a rich literature on graphical history visualization and interaction. A comprehensive survey can be found in [Heer et al. 2008]. Here we focus mainly on works that employ different kinds of temporal history models, as they are most relevant to revision control.

Prior graphical history methods can be classified into two major categories: linear [Kurlander 1993; Myers et al. 1997; Kurihara et al. 2005; Nakamura and Igarashi 2008] and nonlinear [Edwards and Mynatt 1997; Edwards et al. 2000; Klemmer et al. 2002; Su et al. 2009a] models. The linear history model, while sufficient for many visualization and interactive tasks, usually do not provide enough information for image revision control where predominant operations are nonlinear, including branching, editing, and replay. Such parallel information is representable via a nonlinear history model. But to our knowledge, none of the existing methods provide sufficient information for the comprehensive relationships between editing operations (not only temporal but also spatial and semantic dependency).

For example, in [Edwards and Mynatt 1997] the timeline is represented as a tree with nodes as states and edges as actions. Such a state-based model is not suitable for revision control due to potentially large storage size [Heer et al. 2008] and the loss of dependency information between operations. Edwards et al. [2000] deployed a multi-level history model in which many local linear histories are embedded within a global linear history. This allows only a single global timeline and thus cannot handle parallel revisions. Klemmer et al. [2002] also employed a state-based method and thus shared similar problems. An interesting feature of [Klemmer et al. 2002] is the representation of a non-linear history tree in a linear comic-strip fashion by shrinking the branches into a single node. However, this may be confusing as reported in the user study. Su et al. [2009a] proposed an inspiring methodology for representing revision history as an in-place graphic instead of abstract timelines. However, we could not identify a coherent data structure for practical revision control in their work.

Highly relevant to our work, Chronicle [Grossman et al. 2010] records user workflow histories and allows their local playbacks through videos. With a smart design of various filters and probes, users can review their works effectively. However, this system is not about revision control and cannot handle non-linear histories.

Graph structure for computational tasks Many computational tasks utilize a certain graph structure for modeling, e.g. visualization flows [Levoy 1994; Bavoil et al. 2005]. Our method is similar to these examples of prior art in that we also use DAG, a kind of graph structure, for workflow management. However, our system aims at automatic construction of DAG from user interactions whereas in these visualization systems the users are expected to directly construct the flow pipeline. In a sense, our goal for automatic construction is similar to the work on shading models [Cook 1984; Abram and Whitted 1990] even though we focus on a different domain of revision control for image editing. Graph structures have also been applied to solid modeling [Convard and Bourdot 2004], where the history graph allows the modification of editing param-

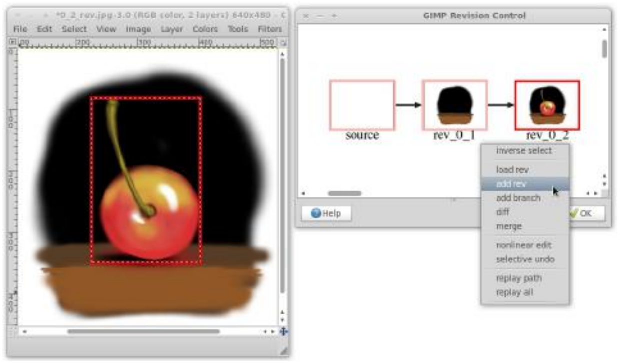


Figure 2: *Revision control user interface. Users can perform revision control functions via the right-click menu within the revision control window (right), which also visualizes the revision history through RevG. When a RevG node is selected, its corresponding spatial context will be highlighted via a bounding box in the main GIMP window (left).*

eters, such as the length of certain object components. However, the proposed technique is more for replaying graphical history than full-fledged revision control.

Highly related to our work, Generic Graphical Library (GEGL), the future core of GIMP, also used a DAG representation. Although GEGL shares a similar graph representation like ours, it is mainly designed as the internal infrastructure for non-destructive image editing on GIMP. Nodes in the DAG can be image editing operations or low-level data structure like image buffers, thus the generated DAG is typically not comprehensible to the users. Compared to our DAG representation, GEGL also does not consider the semantic relationship between operations.

3 Core Ideas

Here, we describe our two core data structures, DAG and RevG. Based on these we then describe our external user interface in Section 4 and internal system implementation in Section 5.

DAG This is our core data structure representing action-based revision history. A DAG is composed of nodes and directed edges. DAG nodes represent image editing operations with relevant information including types, parameters and applied regions. DAG edges represent the relationships between the operations. A (directed) sequential path between two nodes implies a spatial and/or semantic dependency and the path direction implies their temporal order. In contrast, multiple parallel paths between two nodes imply independent operation sequences, e.g. those that apply on disjoint regions or incur orthogonal semantics. The DAG faithfully records the user editing operations and gradually grows as more revisions are committed. Each revision in our system is a sub-graph of the DAG containing the root node which represents the act of initialization, i.e. opening an empty canvas or loading an existing image. The state of the revision is always equivalent to the result generated by traversing its corresponding sub-graph. Note that when dealing with image data, many modern state-based revision control systems (e.g. GIT, SVN and CVS) store separate images as revisions. While in our system, the DAG encodes only actions, not whole images.

RevG Due to the potentially high complexity and large size of DAG, we do not directly expose it to ordinary users. Instead, we visualize it via a state-based revision graph, which we call RevG. Each RevG is essentially a multi-resolution graph visualization of the underlying DAG which is the highest resolution of the revision history. Specifically, each RevG node is the aggregation of one or more DAG nodes and RevG edges are the corresponding DAG

rigid transformation	translation, rotation
deformation	scale, shear, perspective
color and filter	hue, saturation, color balance, birhgtness, contrast, gamma, color fill, blur, sharpen
edit	copy, paste, anchor, add layer, layer mask
brush	brush, pencil, eraser

Table 1: *Supported operations and their classes. Classes in the first three rows are semantic independent. Each font color above indicates border color of a RevG node with the corresponding operation class.*

edges after graph simplification. However, unlike the DAG, which is an abstract representation, we visualize each RevG node with a thumbnail according to the underlying DAG actions.

Our RevG presents both revision branching and operation dependency in a unified representation. In particular, even for a linear revision history, the RevG can still exhibit non-linear parallel structures due to spatial or semantic independencies between operations, such as the editing example in Figure 1. Through RevG, users can easily navigate at different resolutions as well as perform all revision control functions such as addition, diff, branching, merging, and conflict resolving (Figure 2).

4 User Interface

Our UI is a single window displaying the RevG within which revision control commands are issued (Figure 2). Below we describe the main usage scenarios of our system.

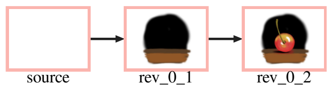
4.1 Revision Navigation

To accommodate potentially complex revision history, RevG provides a multi-resolution view. Users can continuously navigate between different levels of detail, from the coarse revision resolution (Figure 3a) to fine action resolution (Figure 3b), as well as use our constrained navigation mechanism to focus on specific subsets of actions (please refer to the supplementary video). Similar to prior thumbnail-based revision graph designs [Kurlander 1993; Klemmer et al. 2002], we directly embed the thumbnail image within each RevG node. We also designate each node with a descriptive text label and border color signaling its type (Table 1). To further facilitate interaction, we also provide a bi-directional selection mechanism between the main GIMP window and the RevG . By clicking on a RevG node, the bounding rectangle of the corresponding modified regions will be highlighted (Figure 2). Conversely, by selecting an image region, the corresponding RevG nodes will be highlighted.

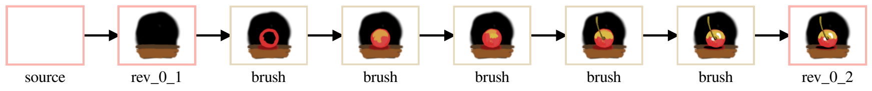
4.2 Revision Diff

To extract differences between text files, a common approach is the classic line-based diff [Hunt et al. 1998]. However, there is no such well-defined difference tool for images. Among all general image comparison visualization approaches, popular ones include side-by-side comparison (e.g. Adobe Bridge, Perforce), layer-based difference (e.g. PixelNovel), per-pixel difference, image overlay (e.g. Wet Paint [Bonanni et al. 2009]), and flickering difference regions (e.g. the compare utility of ImageMagick). These approaches are designed to handle only low level bitmap differences with little information about the editing semantics. In contrast, our system has all the relevant high level information recorded in the DAG for informative diff.

We have designed two revision diff visual mechanisms. The first one is the RevG itself (Figure 3), which users can directly interact with to obtain visual clues about the involved editing operations. The second is a standalone diff UI that can be triggered by selecting two RevG nodes (Figure 4). The diff UI provides a side-by-side parallel comparison between revisions as well as sequential replay



(a) *coarser resolution*



(b) *finer resolution*

Figure 3: *Multi resolution RevG. Our system automatically displays the RevG in proper resolution depending on the viewing conditions and user preferences.*

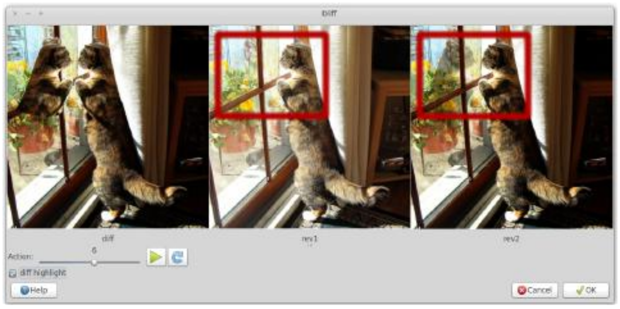


Figure 4: *Diff UI. The preview window (left) shows the editing process between two revisions (middle & right) . Users can manually drag the slider for a particular state or click on the play button for automatic playback. The refresh button would flicker between the two revisions for quick comparison. Users can also turn on the bounding box of difference regions (e.g. the cat reflection). Please refer to the supplementary video for detailed actions.*

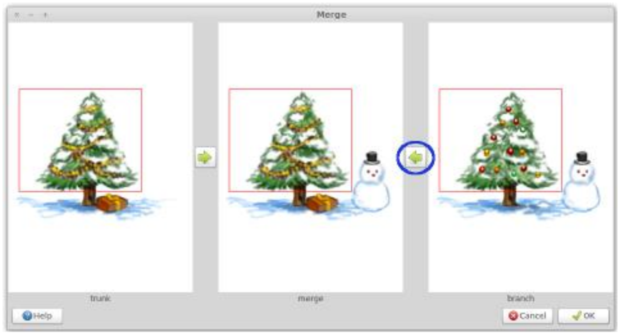
of the in-between actions. The replay speed depends on the underlying RevG resolution. Specifically, when the diff UI is invoked from RevG, we extract 15 steps (or all if the total number is under 15) according to their visual importance to avoid potentially lengthy animation. We also provide a refresh button to flicker between two revisions for in-place comparison as well as a checkbox to highlight the bounding box of difference regions. Note that our system can provide meaningful information to users by ignoring global edits (e.g. gamma correction in Figure 4) and emphasize only local modifications, a task which is difficult to achieve with only low-level bitmap information.

4.3 Add Revision

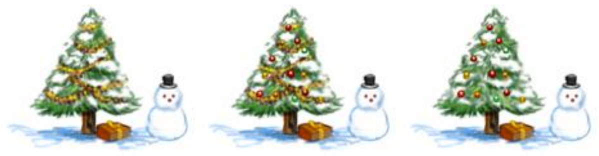
Adding revisions (a.k.a "checking-in") is one of the most frequently used revision control commands. In our system, to save the current work progress as a new revision, users can simply issue the command within the revision control window (Figure 2). Although users can check-in revisions whenever they like, it is generally unnecessary for them to do so in an action-wise fine-grained fashion, since our system can record all the actions and flexibly visualize them with RevG (Section 4.1). As a general guideline, we encourage users to check-in when one of the following two conditions is met: 1) some milestone of the work is achieved, or 2) when users would like to try out different variations. In the latter case, the committed revision can be used as a branch point for future reference or revision rollback, as detailed below.

4.4 Revision Branch and Merge

Branch In the process of software development, many programmers, especially those who work alone or with few other people, might not use the branch command that often. Nevertheless, in the context of open-ended creative content production (e.g. image editing and digital sketching), branching becomes essential even in the usage scenario of a single user. As described in [Terry and Mynatt 2002a; Hartmann et al. 2008], it is common for artists to perform trial-and-error experiments to reach the best outcomes. The rich history keeping of such trial-and-error process and the ability to keep multiple versions (branches) of the design could significantly support designers to achieve their creative goals [Shneiderman 2007]. Similar to the "add revision" command, branches can be easily created within the revision control window (Figure 2).



(a) *merge UI*



(b) *trunk only* **(c)** *branch after trunk* **(d)** *branch only*

Figure 5: *UI for revision merging. (a) shows the merge UI. Images at left and right are revisions to be merged while the center is the preview of the merge result. (b) is the automatic merge result. Non-conflict edits (gift and snowman) are automatically merged while for conflict ones (ornamental strips and balls) the trunk version is chosen by default. Our system also allows users to change the default behavior, including branch-after-trunk (c) by clicking the blue circled button once, branch-only (d) by clicking the blue circled button twice, and trunk-after-branch.*

Merge Merge can be performed between two revisions, either (1) both checked-in (trunk and branch) or (2) one trunk and one local (not yet checked in) copy. Our system first performs an automatic merge between non-conflicting edits (via our graph merge algorithm described in Section 5.4). We also provide a merge UI for manual resolving conflicts as well as for allowing users to change the default merge results. Since our UI and system implementations are very similar for scenarios (1) and (2), for clarity of presentation we will focus on scenario (1) in subsequent descriptions.

The automatic merge can already provide satisfactory results under usual circumstances and requires no further manual intervention from the user. As shown in Figure 5b, the non-conflict contents are automatically merged while, for conflicted ones, the content from the trunk is chosen by default. When automatic merged results are not satisfactory, or there are conflicted scenarios that cannot be automatically resolved, users can invoke our merge UI (Figure 5a). The merge UI contains two revision images and the preview result. The user can interactively drag, zoom, and select the region of interest (the red rectangle in the figure). User interactions are synchronized among all three images for easy comparison.

Once a region is selected, there are four possible merge combinations: **trunk only**, **branch only**, **trunk after branch**, or **branch after trunk**. All these can be achieved via simple button clicks.

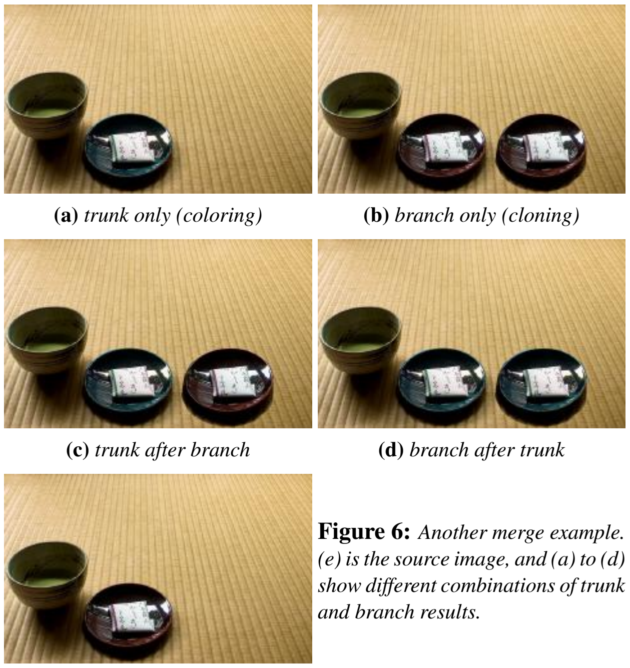

(a) *trunk only (coloring)* (b) *branch only (cloning)* (c) *trunk after branch* (d) *branch after trunk* (e) *source*

Figure 6: *Another merge example. (e) is the source image, and (a) to (d) show different combinations of trunk and branch results.*

Take Figure 5 for example. If we would like to put the ornamental balls (branch) over the strips (trunk), we could simply select the tree (red rectangle) then single click on the right button (marked with a blue circle) and the result is shown in Figure 5c. Or, if we want to completely replace the strip with the ornamental balls (branch only), it can be done by clicking on the button twice (Figure 5d). Figure 6 is another interesting example that clearly demonstrates the effects created via the four combination modes.

5 System Implementation

For practical usage and evaluation, we have fully integrated our revision control system with GIMP. Referencing the architecture of open source visualization systems, including Prefuse [Heer et al. 2005] and the Kitware visualization toolkit, we have built a flexible revision control framework by orthogonalizing the main modules: GIMP core, UI/renderer frontend, and revision control backend (Figure 7). This design provides flexibility for easy integration with different systems.

More specifically, we have modified the GIMP core and added two main components: **logger** for recording user actions and **replayer** for replaying actions. The revision control core communicates with GIMP via its official gimplibrary interface. Recorded logs and other revision control information are stored in the **repository**. The logs are analyzed and transformed into **DAG**, which are further simplified through various **filters** into **RevG**. Finally, RevG is rendered in the frontend **renderer** based on GTK+.

5.1 Logger and Replayer

The **logger** silently records user editing actions in the background in the form of text logs and these can be replayed in the GIMP via the **replayer**. Consecutive identical logs are consolidated similar to [Grabler et al. 2009]. At its latest version 2.8, GIMP has not yet provided any API for command logging. Consequently, to obtain a fine-grained command log, we have to manually hard-wire commands to the logger. On the other hand, operation replay can be easily achieved through the procedure database (PDB) architecture in GIMP, where most image editing functions are registered and can

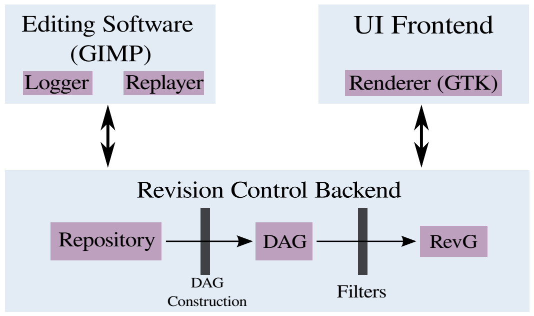


Figure 7: *Major components of our system. With clean separation between GIMP, core revision control backend, and frontend renderer, our system can be easily integrated with other image editing software or operating systems.*

be invoked via libgimp. Nevertheless, with the rapid evolution of GIMP, not all new functions have been added into the PDB properly. For example, functions related to brushes are out-dated and require manual registration. Hopefully, operation recording and replaying would be easier after GIMP developers implement the long desired feature of macro recording.

Log An action log consists of its action name, action parameters, layer ID and a selection mask. For brush and sketching actions, their mouse/stylus motions and pressures are also stored. In our prototype, logs are stored in an ascii format.

5.2 Repository and DAG

When a new revision is committed, the corresponding action logs are transferred from GIMP into the repository and constructed into DAG. For orthogonal implementation, we store the logs in a linear data structure with DAG nodes containing pointers to the corresponding logs.

Operation dependency Our DAG records two kinds of dependencies: spatial and semantic. Spatial dependency considers the spatial relationships between operations; they are spatially independent if their regions of operations do not overlap. As for semantic dependency, we categorize operations into five different classes as shown in Table 1 in which the first three rows are semantically independent (see also [Su et al. 2009a]). Semantically independent operations applied on the same object or region are put into parallel paths (Figure 1).

DAG construction We build the DAG by sequentially inserting each action into the graph with one DAG node corresponding to one action. When inserting a node, the insertion algorithm would search for dependent nodes already in the DAG. If some dependent nodes are connected (i.e. there are paths between them), the latest one among them is picked. The procedure can be done efficiently by a post-order DFS and is detailed in Algorithm 1.

5.3 Filters

Our system creates RevG from DAG interactive by applying a list of "filters", including a *viewport filter* for culling RevG nodes/edges outside the current viewport, a *layout filter* for determining the position, path, shape and color of RevG nodes (via the classical hierarchical layout algorithm [Gansner et al. 1993]), and a *visual importance filter* as detailed in the following paragraph. Note that we provide a generic filter interface so that users with sufficient programming skills can create customized filters.

Algorithm 1 Node Insertion

```
// $c$ : node to be inserted into DAG
// $\mathbf{P}$ : parents of $c$, initialized as empty
// $\mathbf{L}$ : nodes in DAG in the order of post-odering DFS
while $L \neq \emptyset$ do
    $v = pop\_front(L)$
    if $v$ and $c$ are dependent then
        $P = P \cup v$
        remove parents of $v$ from $L$
    end if
end while
insert directed edges from nodes in $P$ to c.
```

Visual importance filter The purpose of the visual importance filter is to dynamically simplify a DAG into a RevG for proper display. The visual importance filter has two stages. First, it assigns a visual importance value to each DAG node. Second, it filters the DAG into RevG according to the threshold assigned to the current RevG window resolution.

The visual importance value v of a DAG node n, which contains one action $act(n)$, is determined by two major factors: image content difference $I(n, m)$ and action context $A(n, m)$. It can be expressed as follows:

$$v(n) = \frac{1}{w} \sum_{m \in N(n)} I(n, m) A(n, m) \qquad (1)$$

where m is a neighboring node of n in DAG within distance w, for which we set to be 2 in our current implementation.

Here, $I(n, m)$ is simply the low-level per-pixel difference between the images after applying $act(n)$ and $act(m)$. On the other hand, $A(n, m)$ takes advantage of the high-level information recorded in the DAG. It assigns higher visual importance to actions with different types or parameters among its neighbors while penalizing those with similar or identical actions, such as repetitive strokes commonly seen in digital sketching. More specifically, if $act(n)$ and $act(m)$ are of different types or parameters, then $A(n, m) = 10^d$, where d is the distance between n and m in the DAG; otherwise, $A(n, m) = 1$. $A(n, m)$ is actually a very flexible term and can be fine-tuned to fit user requirements. For example, we give the operation "add layer" a higher value (100 in our implementation), as our user study indicated that it is very useful to see the operations history on a layer basis (please see Section 7 for more detail).

For $I(n, m)$, it is possible to deploy more sophisticated metrics that consider visual attention and perceptual image analysis such as [Itti et al. 1998; Yee et al. 2001]. However, they are generally computationally expensive and restricted to certain scenarios. The advantage of our system is that, in addition to the low-level bitmap information, we also have the high-level information about the editing operations. By combining the low-level $I(n, m)$ with the high-level $A(n, m)$, our system already provides satisfactory results.

After assigning visual importance values to all nodes, we traverse the DAG in the DFS order and accumulate the visual importance values. Once the accumulated value is higher than the threshold value of current resolution, the corresponding nodes in the DAG are clustered into a single RevG node. After the node clustering, we add back the graph edges. The procedure is similar to classical multi-resolution mesh simplification [Hoppe 1996] and graph visualization [Fairchild et al. 1999; Heer and Card 2004].

5.4 Revision Control Commands

Revision diff In our system, the problem of extracting differences between two revisions can be posed as a graph difference problem of the underlying DAGs. For graph difference, computing one-to-one correspondence between the nodes from scratch is equivalent to isomorphism, a NP-complete problem [Cook 1971]. Fortunately, in our case, we can extract common nodes between two DAGs by matching their labels (as recorded from the original action logs). The rest would be the difference nodes, from which we could implement various diff visualization mechanisms as described in Section 4.2. Note that when two revisions lie on different branches, it is possible to have DAG nodes with different labels that actually represent identical editing operations. Here we simply treat operations with different labels as real differences and have found such approximation satisfactory for diff visualization.

Algorithm 2 Automatic Merge

```
// $G_i, G_j$, DAG of trunk and branch revision, $G_m$, DAG after merge
// $L$ : nodes in $G_j$ in the order of BFS
// $CL$ : conflict list
$G_m \leftarrow G_i$
while $L \neq \emptyset$ do
    $v = pop\_front(L)$
    if $v$ conflicts with nodes in $G_m$ then
        remove children of $v$ from $L$
        push $v$ and its children into $CL$
    else
        insert $v$ into $G_m$ with Algorithm 1
    end if
end while
```

Revision merge Similar to revision diff, revision merge also works on the DAG level. The automatic merge algorithm searches for conflicts between two graphs, merges the non-conflict nodes together and outputs a conflict list, which users could manually resolve via our merge UI. As described in Section 4.4, by default we use the **trunk only** option for conflict resolution. The pseudo-code of the merge algorithm is provided in Algorithm 2. Our merge UI also provides users with the flexibility to merge arbitrary regions (Figure 5). This is achieved by applying Algorithm 2 to DAG nodes whose actions correspond to the selected image regions.

6 Examples

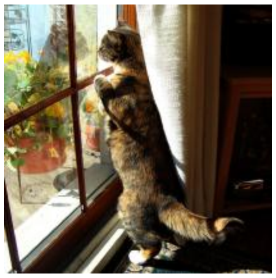
In this section, we demonstrate several usage examples of our system, for both image editing and digital sketching applications. In addition to basic revision control, we also demonstrate potential usage of our system as a creativity support tool.

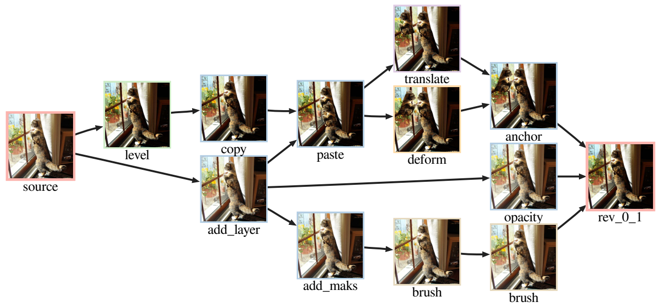
Image editing Figure 8 and 9 are two practical image editing examples where many popular image compositing and photo retouching techniques were used. From the embedded thumbnail images and underlying text labels in the RevG nodes, users can clearly identify the involved operations and their dependencies. With RevG, users can easily review their own work or learn from others. We show the RevGs of these two figures at different levels of detail to demonstrate the multi-resolution feature of our representation.

Digital sketching Figure 10 and Figure 11 are two digital sketching examples produced by our collaborating professional artist. Each example consists of about one thousand actions, which exceeds the maximum size of state-based undo history stack of GIMP. Our system faithfully recorded all actions and can reproduce the whole digital sketching sessions. As shown in the figures, RevG provides a compact and informative visualization of the sketching history by laying out independent operations in parallel paths and clustering together actions with similar parameters.

Creativity support tool In addition to revision control, our system can also facilitate design space exploration, an essential task

(a) *before*


(b) *after*



(c) *RevG at the finest resolution*

Figure 8: *An image compositing example. A fake reflection of the cat is created to enhance the richness of the photograph. Gamma adjustment is first applied followed by the copy and paste procedures. An additional layer mask is also used to correct the occlusion between reflection and window.*

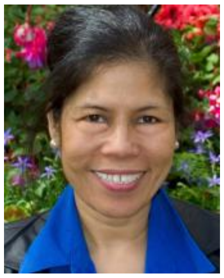
for creativity support tools [Shneiderman 2007]. With our system, it is possible to nonlinearly adjust the parameters of previous operations as shown in Figure 12. Given a user selected image region, our system would search for the related editing operations and prompt the dialog for parameter tuning (Figure 12a). It also provides an intuitive nonlinear selective undo function where users can select any region on the image and undo the associated operations, similar to [Su et al. 2009a; Kurlander 1993]. In Figure 12d, we undo the operations applied on one eye without affecting other regions, which is difficult to achieve with a linear history stack provided by most image editing systems. Finally, our system also provides selective replay as proposed in [Grossman et al. 2010] where users can nonlinearly replay operations corresponding to specific regions of interest.

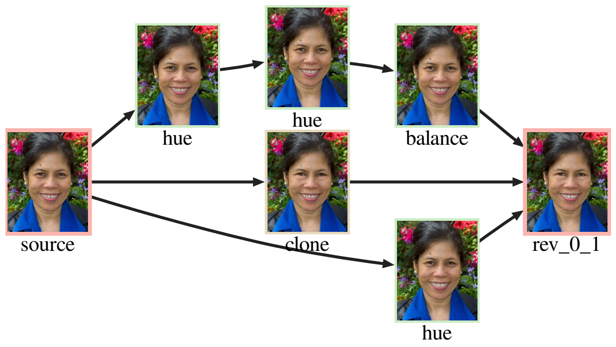
Our merge UI provides an additional tool for creating different variations. An example is shown in Figure 13. It is common for artists to first finish up the main components (character in this example) and then to try out all kinds of decorations by using different layers. However, this can require a careful separation of items into layers while ensuring that the number of layers is manageable. With our system, users could simply draw the alternatives on one layer, save the revision, and create the variations with our merge UI.

7 Evaluation

Since our system is the first for comprehensive image revision control, it is difficult to perform a summative usability test to compare it with other systems [Lau 2010] or to devise general performance metrics on these creative production tasks. Instead, we provide two evaluations for our system: (1) a formative user study conducted during and after our development cycles to shed light on the rationales behind our UI design and the efficacy of our system, and (2) an objective performance comparison on storage size with other revision control systems.

(a) *before*


(b) *after*



(c) *RevG*

Figure 9: *An image retouching example. Several popular image retouching techniques are applied in this example, including clone brush and hue/balance adjustment for eye sharpening, eye whitening, eye bag removal, and teeth whitening.*

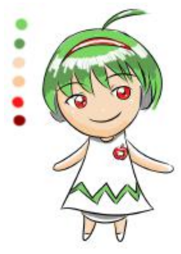
7.1 Formative Usability Study

In our early development cycle, we collaborated with one professional illustrator and two CS graduate students with significant experience of photo shoots and retouching to help our UI design process. At a later stage, we recruited two more professional industry designers, whose daily work involved the use of image editing software such as Adobe Photoshop and Painter, and two more CS graduate students.

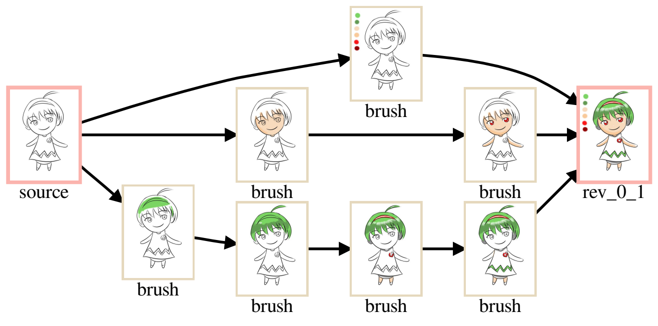
Prior practice Among the participants, only the CS students have experience with revision control systems. Nevertheless, all participants have their own practice for data management and version control for either solo or team projects. One common ad-hoc approach deployed by all participants is to categorize projects into different folders and save the data with informative filenames, usually containing the creation date and simple comments. One participant also mentioned that his company required him to compress and archive files on a weekly basis for potential future reference. The major downside of this approach is that as the project expands the users could easily lose track of the revisions with numerous files scattered in different folders.

Some manual version control approaches are based on functions provided by specific tools. For example, artists commonly use the layer function in image editing software (e.g. GIMP and Adobe Photoshop) for saving different design variations. By toggling the visibility of layers, they can explore different design combinations and alternatives. However, as the number of layers increases (our collaborating artist showed us a case with 40+ layers), keeping variations as layers could make the already difficult layer management even harder. Furthermore, there is also no easy way to browse through these variations stored as layers.

(a) *sketch*



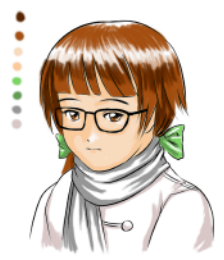
(b) *final image*



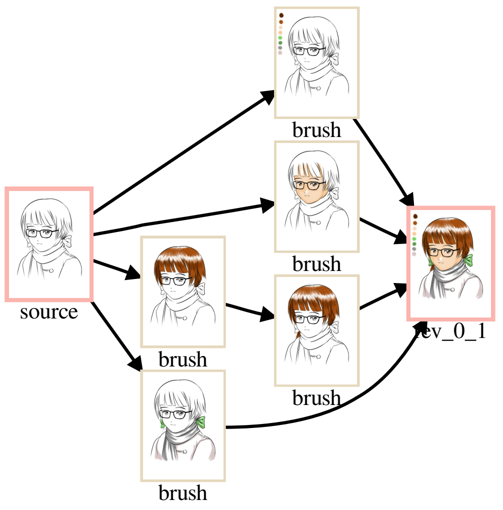
(c) *RevG*

Figure 10: *A digital sketching example. The artist performed 649 brush strokes and employed three layers for the color palette, face, hair, and body. Note that our visual importance filter clusters operations with similar parameters (e.g. brush color) together.*

(a) *sketch*



(b) *final image*



(c) *RevG*

Figure 11: *Another digital sketching example. The artist employed 1391 operations and four layers for the palette, face, hair, and clothing.*

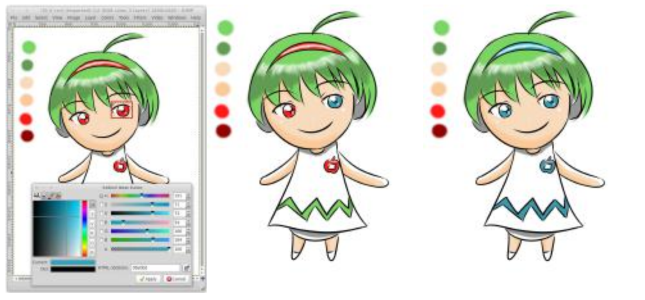
Early design To address these issues discussed above, a natural solution is to use DAG as the core of our system as any revision history can be represented as a DAG. During the early phase of our project, one major design decision is about how to present the revision information encoded in the underlying DAG to the users. In our early prototype, we attempted to expose the DAG directly with relevant tools such as semi-auto node aggregation and annotation as well as the ability for direct DAG manipulation. The CS students and the authors have found such tools complicated but powerful and interesting. On the other hand, the collaborating artist was not interested in such fine-grained operations; rather than dealing with an abstract DAG, the artist preferred to directly interact with images.

Current design Based on the feedback, we have made two key design decisions: 1) hide DAG from ordinary users and visualize it through RevG, and 2) direct all revision control mechanisms through RevG and the revision window. These keep our revision control system easy to use and comprehend, and combinable with the main image editing window (GIMP in our particular implementation). Specifically, users can still perform the ordinary image editing operations and our nonlinear exploration (Figure 12) with the main GIMP window, and interact with our RevG only for revision control commands. The two are naturally connected via the bidirectional mechanisms where selections made on image regions will automatically lead to the corresponding RevG nodes and vice versa (see Figure 2 and 12).

User study In the post design user study, we brought in two more industrial designers and two more CS graduate students as test subjects. After a brief tutorial and demonstration of our system, we assigned them two kinds of tasks: (1) **execution**, where subjects perform their familiar image editing projects (digital sketching or photo retouching) while issuing revision controls through our system and (2) **cognition**, where subjects are given some pre-recorded examples with RevG and asked to deduce the involved editing operations.

Overall, subjects expressed no difficulty in performing basic revision control functions such as checking-in, branching, merging, and revision history navigation. Some subjects even commented that they were happy enough even with a subset of our system to save and roll back revisions. For both tasks, we have observed that subjects tend to stay longer on the coarser RevG resolutions while they are more tentative on venturing into finer resolutions. For task 1, it is understandable since subjects should have innate knowledge of the editing process and thus do not need to consult the finer RevG resolutions. As for task 2, the collected comments suggest that subjects did find the finer-resolution helpful especially when some subtle or unfamiliar effects are involved. However, finer-resolution RevG also means more thumbnail images to analyze and memorize, so subjects tried to stay on coarser resolutions as long as they can comprehend the operations. Some participants also invoked the revision diff tool and manually adjusted the slider back and forth for task 2. Their comments for such a decision were that they preferred seeing the progress of modification in a single preview window in the speed they want for better temporal context and easier identification of modified contents.

For digital sketching, the subjects commented that they care more about the temporal order of strokes than the semantic dependency in RevG. However, they also commented that it is very helpful for our RevG to display brushes applied on different layers in different paths (e.g. Figure 10), which is similar to a layer-based history list, a long desired function for artists. This indicated a practical benefit of our non-linear revision structure. Notably, our collaborating artist is especially interested in the stroke-by-stroke replay.


(a) *rough selection* **(b)** *nonlinear exploration* **(c)** *nonlinear exploration*

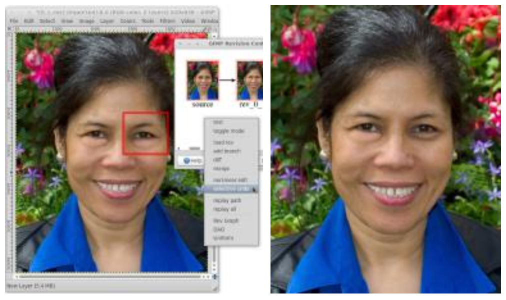

(d) *rough selection* **(e)** *selective undo*

Figure 12: *Nonlinear exploration. Given a rough selection region (a), our system would search for the relevant operations and adjust their parameters as intended (b). (c) shows a possible variation created via this function. Similarly, selective undo can be easily achieved for side-by-side comparison or non-linear editing (d) (e).*

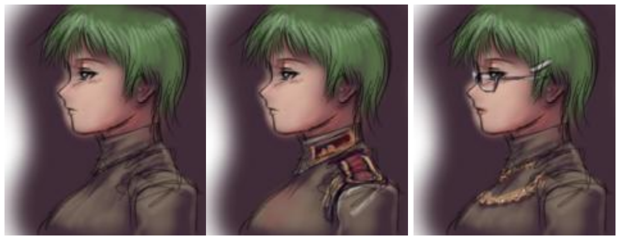
He commented that it is generally difficult to deduce the original painting operations merely from the final flattened image; with our system, one can truly appreciate others' technique and more importantly one's own drawing logic.

Most participants thought that revision merge is very interesting and has a lot of potential but might take some time and practice to completely immerse into their daily work flows. Nevertheless, they like the possibility of creating multiple variations with our UI without worrying about the underlying layers (in contrast to their prior practice) while having fun creating variations as in Figure 13.

In summary, our user study indicates that our system is indeed easy (and sometimes fun) to use, and helpful for revision control image editing projects, even though it might take some time and practice to sink in. For execution tasks (e.g. revision control during image editing) the participants mostly stick to a single RevG resolution, while for cognition tasks (e.g. reviewing or exploring) they find the multi-resolution RevG informative. They also comment that our nonlinear revision history is very helpful in depicting editing information, especially for potentially complex layer structures. In addition to core revision control, participants also find our system conducive for their creative processes.

7.2 Performance

The storage consumption of our system is particularly small compared to other image editing and revision control systems, such as GIMP (.xcf file), GIT, and SVN, as shown in Table 2. For all figures, we divide the whole editing process into four revisions, and commit them to the revision repository. The storage overhead of our system mainly comes from the cached thumbnail images. The overhead of internal data structures for GIT and SVN are not precisely calculated here, but our advantage on storage size is clear. Regarding computation speed, our system runs at interactive speed, and users of our system have not found any slowdown compared to


(a) *branch point* **(b)** *branch1* **(c)** *branch2*

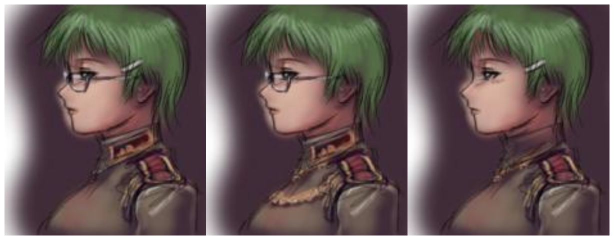

(d) *variation1* **(e)** *variation2* **(f)** *variation3*

Figure 13: *Revision merge as a creativity support tool. From the input (a), the artist created two different branches in (b) and (c). Based on these three images, the artist then created several design variations using our merge UI. Note that only one layer is used for all operations.*

the original GIMP or other revision control systems.

	input	# op	GIMP	SVN	GIT	our
Figure 1	502	11	2.7K	2.1K	2.0K	640
Figure 3	1.6	672	267	224	180	73
Figure 8	945	11	3.5K	3.7K	3.6K	1.3K
Figure 9	276	10	972	1.2K	1.2K	420
Figure 10	377	649	2.3K	2.4K	2.5K	652
Figure 11	425	1391	2.5K	2.7K	2.7K	775

Table 2: *Storage size compaison. All sizes are expressed in K-bytes.*

8 Limitations and Future Work

The main limitation of our current implementation is that it is integrated into a single tool (GIMP) instead of a general mechanism that can work with arbitrary image editing softwares. Popular image editing tools, e.g. Photoshop and Painter, have with their own built-in history mechanisms. We would like to have a universal revision control interface to facilitate the automatic integration of our revision control system in a heterogeneous multi-tool environment.

Our current prototype system is implemented primarily for a single user as it is the predominant usage scenario for current artistic workflows. However, proposed algorithms and system framework directly support multiple users, and the UI can be easily extended for that too. In particular, all one needs to do is to adopt a client-server model, implementing our revision control backend on the server side with the UI frontend and image editor on the client side.

In our current design phase, we have only conducted the usability study with a few subjects (one professional artist, two industrial designers and four CS graduate students). The ideal test bed is to integrate our system into a commercial studio's pipeline. However, it is rather difficult to find a studio using GIMP as their main tool. Nevertheless, we do hope the release of our source code in the public domain can help us gather more user feedback.

In this paper we focused mainly on images, but we believe similar principles are applicable to other binary graphics assets, such as videos, meshes, or animation data. Extending revision control to these data types could be another potential future work direction.

Acknowledgements We would like to thank Eugene Lei for digital sketching examples and valuable comments, Sin-Jhen Chiu at DigiMax and Chuan-Chang Wang at Next Media Animation for their insights on the commercial animation production pipeline, Eric Stollnitz for video dubbing, Duncan Stevenson and the SIGGRAPH English Review Service for proofreading, as well as Ke-Sen Huang and the anonymous reviewers for their valuable suggestions. This work is supported in part by National Science Council (Taiwan) under grant NSC 98-2220-E-003-001.

References

ABRAM, G. D., AND WHITTED, T. 1990. Building block shaders. In *SIGGRAPH '90*, 283–288.

BAVOIL, L., CALLAHAN, S. P., SCHEIDEGGER, C. E., VO, H. T., CROSSNO, P. J., SILVA, C. T., AND FREIRE, J. 2005. Vistrails: Enabling interactive multiple-view visualizations. In *Visualization 2005*, 135–142.

BONANNI, L., XIAO, X., HOCKENBERRY, M., SUBRAMANI, P., ISHII, H., SERACINI, M., AND SCHULZE, J. 2009. Wetpaint: scraping through multi-layered images. In *CHI '09*, 571–574.

CONVARD, T., AND BOURDOT, P. 2004. History based reactive objects for immersive cad. In *SM '04: Symposium on Solid modeling and applications*, 291–296.

COOK, S. A. 1971. The complexity of theorem-proving procedures. In *STOC '71: Symposium on Theory of computing*, 151–158.

COOK, R. L. 1984. Shade trees. In *SIGGRAPH '84*, 223–231.

EDWARDS, W. K., AND MYNATT, E. D. 1997. Timewarp: techniques for autonomous collaboration. In *CHI '97*, 218–225.

EDWARDS, W. K., IGARASHI, T., LAMARCA, A., AND MYNATT, E. D. 2000. A temporal model for multi-level undo and redo. In *UIST '00*, 31–40.

ESTUBLIER, J., LEBLANG, D., HOEK, A. V. D., CONRADI, R., CLEMM, G., TICHY, W., AND WIBORG-WEBER, D. 2005. Impact of software engineering research on the practice of software configuration management. *ACM Trans. Softw. Eng. Methodol. 14*, 4, 383–430.

FAIRCHILD, K. M., POLTROCK, S. E., AND FURNAS, G. W. 1999. Readings in information visualization. ch. SemNet: three-dimensional graphic representations of large knowledge bases, 190–206.

GANSNER, E. R., KOUTSOFIOS, E., NORTH, S. C., AND VO, K.-P. 1993. A technique for drawing directed graphs. *IEEE Trans. Softw. Eng. 19*, 3, 214–230.

GRABLER, F., AGRAWALA, M., LI, W., DONTCHEVA, M., AND IGARASHI, T. 2009. Generating photo manipulation tutorials by demonstration. In *SIGGRAPH '09*, 66:1–9.

GROSSMAN, T., MATEJKA, J., AND FITZMAURICE, G. 2010. Chronicle: capture, exploration, and playback of document workflow histories. In *UIST '10*, 143–152.

HARTMANN, B., YU, L., ALLISON, A., YANG, Y., AND KLEMMER, S. R. 2008. Design as exploration: creating interface alternatives through parallel authoring and runtime tuning. In *UIST '08*, 91–100.

HEER, J., AND CARD, S. K. 2004. Doitrees revisited: scalable, space-constrained visualization of hierarchical data. In *AVI '04: Proceedings of the working conference on Advanced visual interfaces*, 421–424.

HEER, J., CARD, S. K., AND LANDAY, J. A. 2005. prefuse: a toolkit for interactive information visualization. In *CHI '05*, 421–430.

HEER, J., MACKINLAY, J., STOLTE, C., AND AGRAWALA, M. 2008. Graphical histories for visualization: Supporting analysis, communication, and evaluation. *IEEE Transactions on Visualization and Computer Graphics 14*, 6, 1189–1196.

HOPPE, H. 1996. Progressive meshes. In *SIGGRAPH '96*, 99–108.

HUNT, J. J., VO, K.-P., AND TICHY, W. F. 1998. Delta algorithms: an empirical analysis. *ACM Trans. Softw. Eng. Methodol. 7*, 2, 192–214.

ITTI, L., KOCH, C., AND NIEBUR, E. 1998. A model of saliency-based visual attention for rapid scene analysis. *IEEE Trans. Pattern Anal. Mach. Intell. 20* (November), 1254–1259.

JACKSON, D., AND LADD, D. A. 1994. Semantic diff: A tool for summarizing the effects of modifications. In *ICSM '94: Proceedings of the International Conference on Software Maintenance*, 243–252.

JACOBSEN, J., SCHLENKER, T., AND EDWARDS, L. 2005. *Implementing a Digital Asset Management System: For Animation, Computer Games, and Web Development*. Focal Press.

KLEMMER, S. R., THOMSEN, M., PHELPS-GOODMAN, E., LEE, R., AND LANDAY, J. A. 2002. Where do web sites come from?: capturing and interacting with design history. In *CHI '02*, 1–8.

KURIHARA, K., VRONAY, D., AND IGARASHI, T. 2005. Flexible timeline user interface using constraints. In *CHI '05*, 1581–1584.

KURLANDER, D. 1993. Chimera: example-based graphical editing. In *Watch what I do: programming by demonstration*, 271–290.

LAU, T. 2010. Rethinking the systems review process. *Commun. ACM 53* (November), 10–11.

LEVOY, M. 1994. Spreadsheets for images. In *SIGGRAPH '94*, 139–146.

MYERS, B. A., MCDANIEL, R. G., MILLER, R. C., FERRENCY, A. S., FAULRING, A., KYLE, B. D., MICKISH, A., KLIMOVITSKI, A., AND DOANE, P. 1997. The amulet environment: New models for effective user interface software development. *IEEE Transactions on Software Engineering 23*, 347–365.

NAKAMURA, T., AND IGARASHI, T. 2008. An application-independent system for visualizing user operation history. In *UIST '08*, 23–32.

SHNEIDERMAN, B. 2007. Creativity support tools: accelerating discovery and innovation. *Commun. ACM 50* (December), 20–32.

SU, S. L., PARIS, S., ALIAGA, F., SCULL, C., JOHNSON, S., AND DURAND, F. 2009. Interactive visual histories for vector graphics. Tech. Rep. MIT-CSAIL-TR-2009-031, Massachusetts Institute of Technology, Computer Science and Artificial Intelligence Laboratory, June.

TERRY, M., AND MYNATT, E. D. 2002. Recognizing creative needs in user interface design. In *C&C '02: Proceedings of the 4th conference on Creativity & cognition*, 38–44.

TERRY, M., MYNATT, E. D., NAKAKOJI, K., AND YAMAMOTO, Y. 2004. Variation in element and action: supporting simultaneous development of alternative solutions. In *CHI '04*, 711–718.

YEE, H., PATTANAIK, S., AND GREENBERG, D. P. 2001. Spatiotemporal sensitivity and visual attention for efficient rendering of dynamic environments. *ACM Trans. Graph. 20*, 1, 39–65.