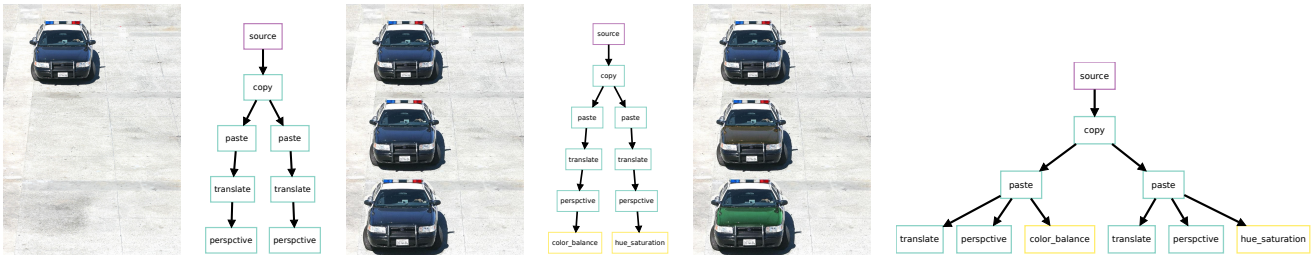
Nonlinear Revision Control for Images



(a) input (b) cloning (c) after cloning (d) color edit (e) output (f) parallel paths

Figure 1: Nonlinear revision control example. Given the input image (a) with a single police car, the user first clones the car twice with proper translation and perspective deformation (c) and then modify their colors (e). The corresponding DAGs are shown in (b) and (d). We use parallel paths to represent independent operations that could have either disjoint regions of interest (b) or orthogonal operations (f). These paths could be hierarchical, from high level (e.g. independent cars) to low level (e.g. translation and color editing for each car).

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 could be inconvenient for graphics applications, which might 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 parallel spatial, temporal and semantic relationships between the nodes. The DAG facilitates not only meaningful display of the revision history but also common revision control operations such as review, replay, branching, and merging. We have also proposed an UI that visualizes the DAG information in an intuitive and user friendly manner. We have built a prototype system upon GIMP, an open source image editor, and demonstrate the effectiveness of method through user study as well as storage advantage through comparisons with alternative revision control systems.

Keywords: revision control, images, nonlinear editing, interaction

1 Introduction

Revision control is an important component of digital content management [Estublier et al. 2005]. Many people have experienced certain revision control systems (e.g. CVS, Subversion, Perforce, to name just a few), especially when working on collaborative projects. By storing history of files, revision control systems allow us revert mistakes and review changes. Revision control systems also facilitate collaborations between multiple users through mechanisms such as merging, branching, and conflict resolving.

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

To obtain the relevant high level information, automatic extraction has proven to be difficult and time consuming even for images under constrained setting (see e.g. [Seitz and Baker 2009]). Fortunately, 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 (e.g. [Kurlander 1993; Meng et al. 1998; Klemmer et al. 2002; Heer et al. 2008; Su et al. 2009]. Nevertheless, the lack of a formal representation that depicts the comprehensive relationship (not only temporal but also spatial and semantic) between 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 ordinary digital editing and sketching workflows in mind. We achieve high-level and fine-granularity revision history by directly recording and consolidating user editing operations. The core idea of our system is a DAG (directed acyclic graph) representing the nonlinear spatial, temporal, and semantic dependencies between these recorded image editing operations (as DAG nodes). The fine operation granularity and dependency information represented in our DAG makes it possible to design a centralized and intuitive user interface for primary functions in revision control system including revision navigation, branch, merge and conflict resolution. Furthermore, our DAG representation also serves as a framework for several non-linear editing and exploration functions proposed in prior art [Kurlander 1993; Su et al. 2009; Myers and Kosbie 1996; Meng et al. 1998; Terry et al. 2004].

Based on our core DAG representation we have devised several algorithm innovations for both internal system implementation and external user interface design. For the former, we propose methods for on-the-fly DAG construction during the user editing/sketching process as well as semi-automatic node aggregation + annotation for grouping many low-level operations (e.g. repeated strokes) into a few high level ones. We also devise mechanisms for automatic resolving and merging multiple concurrent/conflict user revisions. These, along with other technical innovations, including script-based action logging and action state cache, have been designed to ensure easy usage, high computation speed, and low storage size. For the latter, we propose an unified UI design that seamlessly integrates traditional revision history representation and our proposed

underlying DAG structure, alongside the main user editing area. Within the unified UI, artists can navigate the revision history, explore the design space with tools like selective undo/redo [Meng et al. 1998], parallel preview [Terry et al. 2004], and nonlinear operation modification [Kurlander 1993; Su et al. 2009], while perform common revision control operations such as branching, merging, and conflict resolving. We have built a prototype system via GIMP [The GIMP Team 2009], an open source image editor, and demonstrate effectiveness of our system through user study and comparisons with alternative revision control systems.

2 Previous Work

Our nonlinear revision control system for images is closely related to and primarily inspired by two major groups of prior art: digital content management and graphical history. We also review the prior usage of DAG and other related graph structures.

Digital content management Digital content management refers to the general process of authoring, editing, collecting, publishing, and distributing digital content, including both textual data like documents and source codes or binary data like graphics assets [Jacobsen et al. 2005]. Among the various components of digital content management, revision control remains one of the most important part; see [Estublier et al. 2005] for a detailed history and survey of techniques in the context of software management. 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 and Szymanski 1977] or high level (e.g. programming language syntax) information [Jackson and Ladd 1994]. However, for binary files, the prevailing methods either store the complete files for each revision or use certain crude binary diff methods to store the differences [Hunt et al. 1998]. Both of these methods can consume significant storage, and more importantly lack relevant high level semantic information designating the nature of the changes. These issues could hamper the adoption of revision control systems in managing graphics assets. Judging by the success of content management systems for graphics assets (e.g. Alienbrain [AVID Techology 2009]), such demands obviously exist, but to our knowledge binary graphics assets have yet to enjoy as advanced revision control mechanisms as text files. The goal of our system is to fill this gap, allowing easy revision control for graphics assets. Within this paper we focus mainly on images as they are easier to visualize for illustration purposes and also tend to be more commonly used than other graphics data types.

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 model, as it is most relevant to revision control. Prior graphical history methods can be classified into two major categories: linear [Kurlander 1993; Berlage 1994; 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; Hartmann et al. 2008; Su 2007; Su et al. 2009] 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 that depicts the comprehensive relationship between image editing operations (not only temporal but also spatial and semantic dependency), making them either inefficient (in terms of speed or storage) or insufficient for revision control. For example, in [Edwards and Mynatt 1997; Hartmann et al. 2008] the timeline is represented as a tree with nodes as states and edges as actions. Such 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 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. [2007; 2009] proposed an inspiring methodology for representing revision history as in-place graphic instead of abstract timelines. However, we could not identify a coherent data structure for practical revision control in their works.

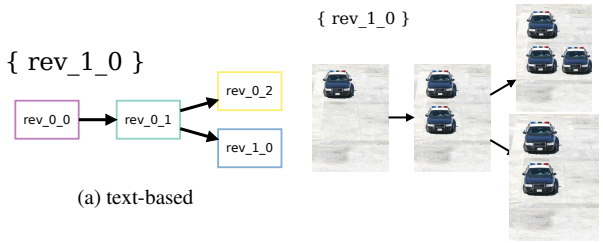
Graph structure for computational tasks Many computational tasks utilize a certain graph structure for modeling, e.g. visualization flows [Levoy 1994; Parker and Johnson 1995; Konstantinides and Rasure 1994; Bavoil et al. 2005; Schroeder et al. 1997]. Our method is similar to these prior art in that we also use the 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 parameters, such as the length of certain object components. However, the proposed technique is more for replaying graphical history (previous paragraph) than full fledged revision control.

Highly related to our work, Generic Graphical Library (GEGL), the future core of popular image editor GIMP, also used a DAG representation. Although GEGL shares similar graph representation like ours, it is mainly designed as the future internal infrastructure for non-destructive image editing of GIMP. Nodes in the DAG can be image editing operations or low-level data structure like image buffer, thus the generated DAG is typically not comprehensible to the users. Comparing to our DAG representation, GEGL also does not consider the semantic relationship between operations. Since GEGL is still under development and not yet fully integrated into GIMP, we could neither fully evaluate its functions and performance nor directly compare it against our method. However, we do plan to release source code and integrate our system with GEGL + GIMP upon the completion of this project.

3 Overview

Core representation We use DAG (directed acyclic graph) as the core representation of our action-based revision history. A DAG is composed of nodes and (directed) edges. Nodes in our DAG represent edition operations with relevant information including types, parameters and applied regions. DAG edges represents the relationships between the operations. A directed path exists between two nodes implies a spatial and semantic dependency and the path direction implies their temporal order. The DAG faithfully records the users' editing operations and grows as they commit more operations during their image editing sessions. And the image is always equivalent to the result generated by traversing the whole DAG.

When dealing with image data, many modern state-based revision control systems (e.g. GIT, SVN and CVS) store separate images as revisions. On the contrary, in our system, we store only one DAG



(a) text-based

(b) image-based

Figure 2: Basic UI that shows revision tree in two different styles. The top left corner shows the revision number of the node under mouse cursor.

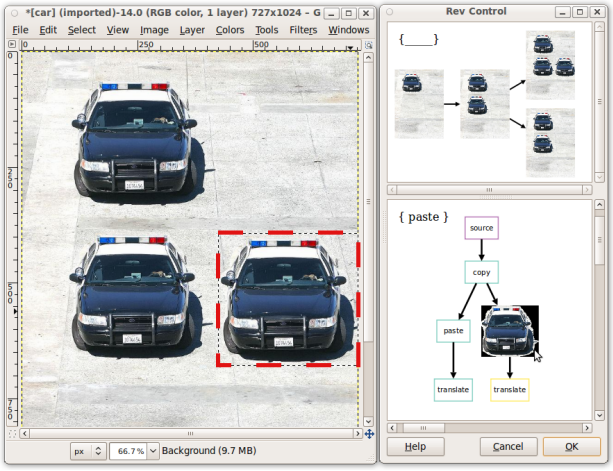


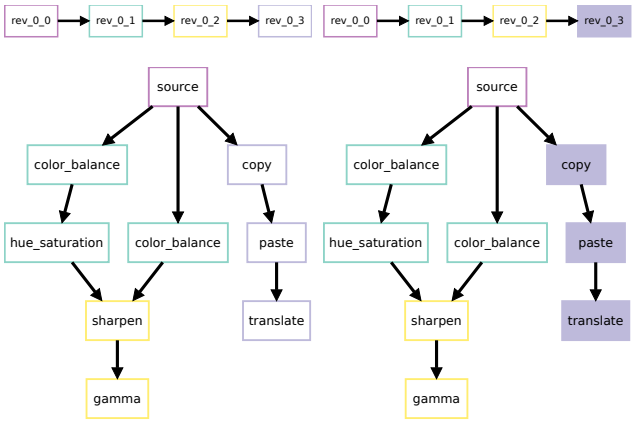
Figure 3: Main context of GIMP and our revision control UI for both the revision tree and the DAG. When a DAG node is single-clicked, its operation name will appear on the upper left window corner, and its spatial context will be highlighted via a rectangular box within the whole image. While double-clicked would switch the style of that particular node between text and image.

while each revision is a sub-graph of the DAG.

For representation and visualization purposes, we also allow a potentially hierarchical representation of DAG, where an **aggregate** node can represent a sub-graph.

External user interface Based on this core DAG representation, we design an user interface for revision control system that minimize the interruption of user's creative process while maximize the usability for various use case. An unified user interface is proposed to support the principle operations including branch, merge and revision history traversal. We design our UI so that it not only supports revision control but also facilitates experimental exploration of the design space by artists [Terry et al. 2004]. In this part, we focus on the questions of **what** features our system should support, and **why** we make these design decisions, as detailed in Section 4.

Internal system implementation Our core DAG representation is sufficiently flexible. However, to strike the right balance between computation speed, storage consumption, and easy usage, we have to carefully design our system. This part involves several algorithmic components for our system architecture and implementation, as detailed in Section 5. In this part, we focus on the question of **how** we implement the system.



(a) multi-color

(b) fill-color

Figure 4: Two different color representations in our advanced DAG UI. In multi-color mode, nodes belonging to different revisions are colored as the corresponding revision tree nodes. In fill-color mode, only nodes belonging to the selected revisions are filled with the corresponding color.

4 User Interface

For practical usage and evaluation, we have integrated our revision control system with GIMP [The GIMP Team 2009], an open source image editor, in the form of GIMP plug-in. Users can activate command logging function, save revisions and invoke our revision control plug-in from GIMP menu. Our revision control plug-in is directly routed to the main drawing window of GIMP and is capable of providing interactive user feedback; more details about GIMP integration can be found in Section 5.

The user interface of the original GIMP (as well as other commercial image sketching / editing software like Painter and Photoshop) already consists of many visible buttons and could be daunting to the users. Thus the main design principle of our revision control system is to provide an intuitive user interface that respects the logic of the original workflow of GIMP while avoiding unnecessary complexity to the users.

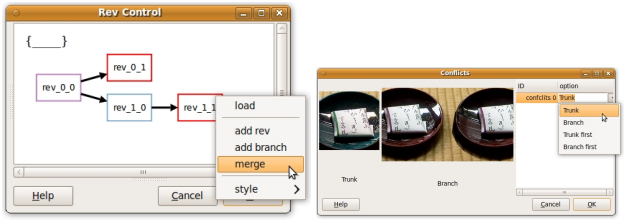
The default user interface of our revision control system is simply a revision tree as shown in Figure 2. Users can switch between text-based mode where each node shows the revision number and image-based mode where each node shows the thumbnail of that revision. In both modes, when the user moves the mouse cursor over a particular node, the corresponding revision name will appear on the top-left corner.

For advanced users who would like to identify the fine-grained operations between revisions for the purpose of reviewing or design space exploration, she can switch on the underlying detailed DAG representation (Figure 3). Through DAG, users can appreciate the detailed dependency between operations. Similarly, users can switch between text-based and image-based modes for more visual clues.

4.1 Image, Revision Tree, and DAG

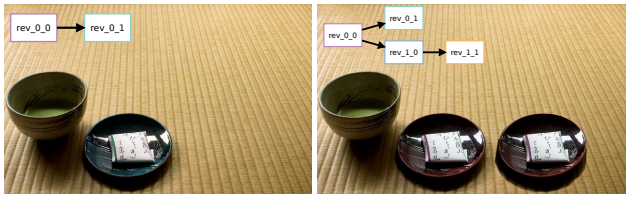
Here, we describe the high level relationships between the image and the corresponding DAG and revision tree. In a sense, the DAG is a finer resolution representation of the revision tree. And the final image can be reproduced via the sequence of operations recorded in the DAG.

The three way relationships between images, DAGs and revision trees are linked through both static thumbnail images embedded in the latter two, as well as dynamic user interactions: when selecting certain image regions, our system highlights the corresponding

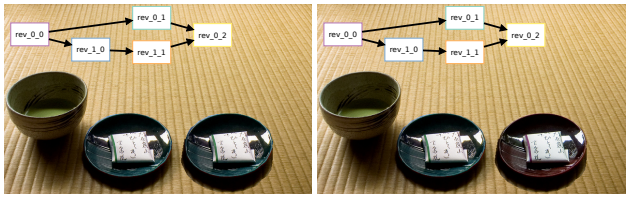


(a) merge from revision tree (b) conflict solver

Figure 5: UI for merging. Users can easily merge two revisions from the revision tree (a). When conflicts are detected, our system launches a conflict solver UI, showing the conflicting image regions (b). User can select one of four merge options from the drop-down menu in the list.



(a) modification in trunk (b) parallel branch editing



(c) merge result 1 (trunk first) (d) merge result 2 (branch first)

(e) source

Figure 6: A conflict merge example. (a) and (b) show two parallel editing. A conflict happens at the dish region. (a) to (d) show four possible merge results. (Note that we embed the revision tree inside each image for illustration purpose.)

revision tree and DAG nodes; conversely, when the mouse cursor is moved over a revision-tree/DAG node, the corresponding region will be highlighted within the whole image (Figure 2, 3, 4).

Note that in most use cases, we do not directly expose the DAG to the users, as they might get distracted from their original work flows due to the potential visual complexity of the DAG. Fortunately, principle revision control functions such as history navigation, branching and merging can be achieved directly from the revision tree.

For advanced tasks, like nonlinear play-backs or DAG annotation, where direct DAG interaction is necessary, we also carefully design the user interface following the convention of the original GIMP interface (e.g. selection highlight on image, using default GIMP widget interface, etc).

In the following two sections, we describe more detailed functionality for both the revision tree and DAG through concrete examples.

4.2 Revision Tree Visualization and Interaction

Navigation We provide two display modes for our revision tree user interface (Figure 2): the *text* mode, where the node label represents the branch number (first number) and revision number (second number), and the *image* mode, where small thumbnails of the corresponding revision nodes are shown in a comic strip fashion. Similar user interface designs that embed the snapshots of the image states into the graphical history can be found in many previous works (e.g. [Hartmann et al. 2008; Kurlander 1993]). Primary revision control operations such as rollback, branch, or merge can be performed directly on the revision tree via the right click pop-up menu (more details in Section 4.2).

Branching and merging Branching and merging are two important operations for nonlinear revision control in the context of software development. It is even more so when it comes to the open-ended creative content production, where it is common for artists to perform trial-and-error experiments to obtain the best outcome of design. As discussed in [Terry and Mynatt 2002; Hartmann et al. 2008], the history of trial-and-error process itself and the ability to keep multiple versions (branching) of the design provide valuable information for designers to achieve their goals. Considering its potential high frequency of use, we try to simplify our user interface for branching and merging as much as possible.

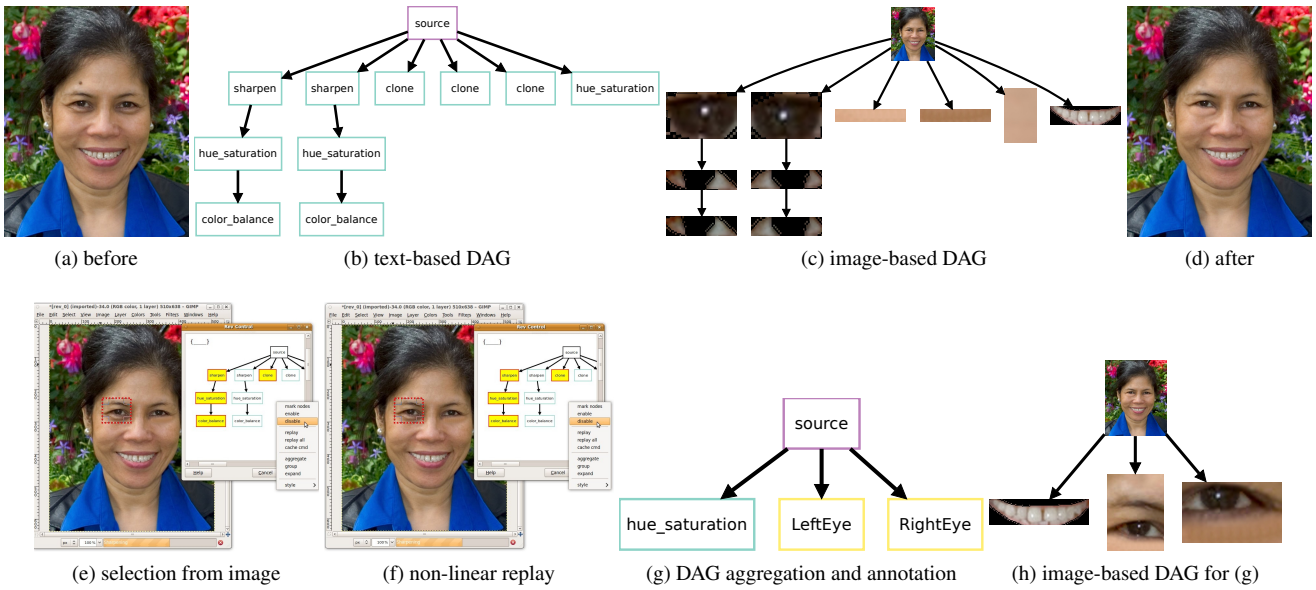
For branching, the user first navigates to the relevant revision, right clicks on the revision node, and selects the "create branch" option from pop-up menu. Then she can continue her editing from there. Merging can be performed between either (1) two branch (already checked-in) revisions or (2) one local (not-yet-checked-in) and one trunk (already checked-in) revisions. For scenario (1), the user simply selects the two relevant revisions to be merged, and our revision control systems will first try to merge them automatically according to their underlying DAGs (details in Section 5.2). When the two revisions cannot be automatically merged (due to conflicts that require user inputs to revolve), our system invokes a conflict resolution dialog asking for manual intervention (Figure 5b). For scenario (2), a similar process happens without requiring the user to select the two revisions to be merged as that information can be automatically detected by the latest trunk version and the current local version. Since our UI and algorithms can support both scenarios in a very similar fashion, for clarity of presentation we will focus on scenario (1) in subsequent descriptions.

Conflict solver Similar to merge options provided in modern revision control systems, we provide four possible merge options: **trunk, branch, trunk first, branch first**. For **trunk** option, the conflict content from the branch is completely discarded. For **branch** option, the conflicted content in the trunk is discarded. For **trunk first** option, contents from **branch** are appended after trunk ones. Finally, for **branch first** option, it is the other way around.

Here we illustrate different outcomes of these four options using the example in Figure 6. Figure 6e is the original source image. Figure 6a represents the main trunk revision and Figure 6b the revision in branch. When we tried to merge the branch back into trunk, the conflict happens at the dish region. If **trunk** option is taken, the conflict in the branch is discarded, and the merge result is identical to Figure 6a. Similarly, for **branch** option, the merge result is Figure 6b. For **trunk first** option, the color modification in trunk took place prior to the clone operation in branch, thus the modified color is propagated (Figure 6c). For **branch first** option, clone operation has the priority and thus only one dish is affected by the color modification (Figure 6d).

4.3 DAG Visualization and Interaction

As mentioned in Section 3, DAG is the core internal representation of our revision control system. In most cases, the DAG is hidden from users to reduce the visual clutter and the complexity of user interface. Nevertheless, there are still some situations where expos-



(a) before (b) text-based DAG (c) image-based DAG (d) after

(e) selection from image (f) non-linear replay (g) DAG aggregation and annotation (h) image-based DAG for (g)

Figure 7: Advanced DAG user interface. (b) shows the detailed DAG describing the digital retouching process. To find out the operation of interest, one can either switch to image-based mode (c) or directly select on the image (e). After that, one can perform various interaction such as non-linear exploration (f) or annotation (g).

ing DAG to the user might become useful. For example, one might be interested in reviewing the fine-grained operations of her own or others work flow to better appreciate the logic and process of the creativity production. Through DAG, users can clearly grasp the spatial, temporal and semantic dependencies between operations including parallel/independent operation sequences.

In the following paragraphs, we first explain the visualization and interaction of DAG. We then further elaborate other use cases where DAG would be useful including non-linear design space exploration, merge and process annotation.

Example Here we illustrate our system via a portrait retouching example (Figure 7). The retouching process includes eyes sharpening, teeth whitening and eye-bag/mole removal. These are all useful and popular retouching techniques commonly seem in photography retouching process. We follow the process in [Kelby 2005], which is written for Photoshop user, but the idea is basically the same.

DAG representation The DAG nodes are colored according to their corresponding revision tree nodes so that the user can easily figure out their correspondences (Figure 4a). For sufficiently complex revision history that may contain many colored nodes, we also provide another mode that highlights the relevant DAG nodes belongs to the selected revision (Figure 4b). Similar to the revision tree nodes, users can browse the DAG in text-based (Figure 7b) or image-based (Figure 7c) style. In both representations, when the mouse cursor is over the node or thumbnail, the corresponding operation name will appear on the top-left window corner.

Non-linear exploration Based on the selection scheme described above, users can easily achieve two important interactions in creative content production: non-linear design space exploration and history annotation.

The open-ended design is typically a continuous trial-and-error process [Terry and Mynatt 2002; Hartmann et al. 2008]. Designers tend to try out all kinds of design possibilities, perform many side-by-side comparisons and try to figure out the best outcome through this experimental process.

As discussed in Section 4.2, artists can already preserve and compare parallel working copies via revision tree user interface. Via DAG, artists can further access more detailed operation information for the design space exploration. For example, as shown in Figure 7f, to clarify the outcome of the image editing operations related to character's eyes, the user can selectively turn off the operations related to left eye and perform a side-by-side comparison with right eye. The whole process is fairly simple; the user first circles the left eye on the image, our system then automatically highlights the related nodes. The user can then disable the nodes or adjust their parameters.

DAG aggregation and annotation To further enhance readability, one can aggregate multiple nodes and add text annotation to provide better semantic description. This is similar to the idea of hierarchical command objects and command chunking proposed in [Kurlander and Feiner 1991; Myers and Kosbie 1996]. The user could select a collection of nodes and our system will fold the nodes into a single aggregate node. An example is shown in (Figure 7g), where we aggregate and annotate operations applied to the left and right eyes into separate aggregation nodes.

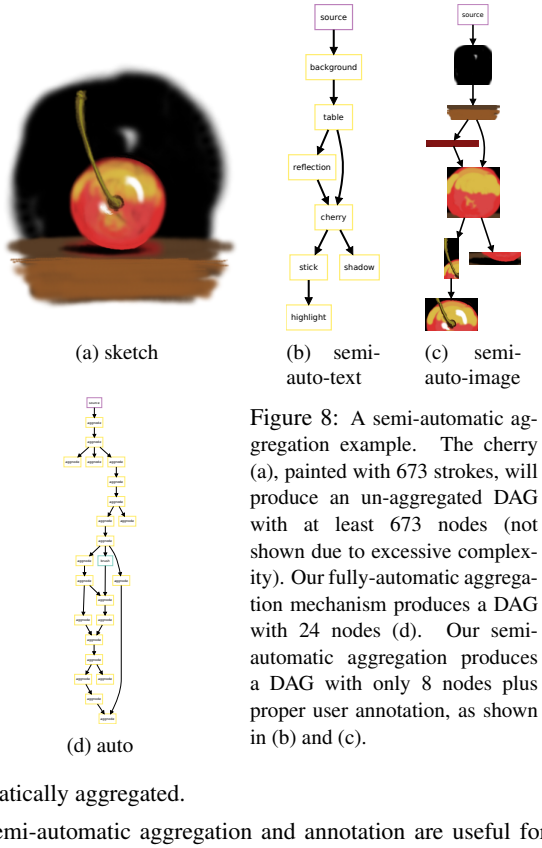
Beside such manual selection, we also provide a semi-automatic mechanisms for DAG aggregation. The semi-automatic aggregation is an interactive process where the system first automatic aggregates the operations using principles detailed below, followed by further user manual aggregation and annotation as described above.

The principles for auto-aggregation are:

Layer basis, for which operations applied on same layer are aggregated into a single node. In such cases, the DAG becomes similar to a layer list commonly seem in popular image manipulation software such as GIMP or Photoshop.

Spatial proximity, for which nodes applied on nearby regions are automatically aggregated.

Operation similarity, for which operations with similar properties or parameters, e.g. similarly colored brushes, are auto-



(a) sketch (b) semi-auto-text (c) semi-auto-image

(d) auto

Figure 8: A semi-automatic aggregation example. The cherry (a), painted with 673 strokes, will produce an un-aggregated DAG with at least 673 nodes (not shown due to excessive complexity). Our fully-automatic aggregation mechanism produces a DAG with 24 nodes (d). Our semi-automatic aggregation produces a DAG with only 8 nodes plus proper user annotation, as shown in (b) and (c).

matically aggregated.

Such semi-automatic aggregation and annotation are useful for a variety of scenarios, e.g. digital sketching which could easily contain hundreds or thousands of strokes. Without any aggregation, a DAG of such complexity could be difficult to comprehend. On the other hand, fully manual aggregation might be too tedious for the users, while fully automatic aggregation might not work well as there might not be well defined rules for aggregating strokes. Our semi-automatic mechanism helps users guide our system to aggregate DAG nodes as well as add meaningful annotation, which could be useful for later replay, review, or exploration.

An example is shown in Figure 8, where the fully automatically aggregated DAG based on the principles listed above is shown in Figure 8d, which aggregate 673 nodes into 22. Although the automatic aggregated DAG shows dependency between strokes, without proper annotation, it did not provide too much meaningful information to the users. Our semi-automatic mechanism allows the user to intervene with the automatic aggregation process. Such intervention could take place on the fly, or later during the replay stage if the user prefers to focus on the drawing first. Figure 8b shows the DAG with user annotation. See Section 6 and the accompany video for more examples.

5 System Implementation

Besides external user interface, there are two main internal components in our system:**operation recorder/replayer** and **DAG repository**. During image manipulation process, **operation recorder** quietly records all operations committed by the user. The operation logs are parsed, transformed into DAG and stored in the **DAG repository**. **Operation replayer** can then replay the operations according to the DAG topology for various tasks such as command cache and non-linear play-back.

Algorithm 1 Node Insertion

// $\mathbf{G}$: DAG, initialized with a single source node
// $\mathbf{V}(\mathbf{G})$: nodes in DAG
// $\mathbf{E}(\mathbf{G})$: edges in DAG
// c : operation node to be inserted into DAG
// s : the source image (root) of the DAG
search for nodes $\mathbf{v} \in \mathbf{V}(\mathbf{G})$ overlapped with c
if $\|\mathbf{v}\| = 0$ **then**
  $\mathbf{E} = \mathbf{E} \cup \{(s, c)\}$ //link to root node
else
  while $\|\mathbf{v}\| \neq 0$ **do**
    $v \leftarrow \mathbf{v}(0)$
    calculate shortest path $\mathbf{P}$ from v to the last entry of the path
    In reverse order of $\mathbf{P}$, find first node p overlapped with c
    $\mathbf{E} = \mathbf{E} \cup \{(p, c)\}$
    $\mathbf{v} = \mathbf{v} - \mathbf{P}$
  end while
end if
$\mathbf{V} = \mathbf{V} \cup \{c\}$

Algorithm 2 Parallel Path Building

// $\mathbf{V}$: nodes in DAG
// $\mathbf{C}$: clusters containing paths to be parallelized
// *class* means operation class of the node
// search for path that can be parallelized
$\mathbf{S} \leftarrow \mathbf{V}$
while $\|\mathbf{S}\| \neq 0$ **do**
  $\mathbf{P} \leftarrow \mathbf{S}(\mathbf{0})$
  while $\exists p \in adj(\mathbf{P})$, whose class $\neq$ class of all nodes in $\mathbf{P}$ **do**
    $\mathbf{P} = \mathbf{P} \cup p$
  end while
  $\mathbf{C} = \mathbf{C} \cup \mathbf{P}$
  $\mathbf{S} = \mathbf{S} - \mathbf{P}$
end while
//parallelize the path
for all paths $\mathbf{P}$ in $\mathbf{C}$ **do**
  find common ancestor $p0$ and children $p1$ of the path
  remove all connected edges in $\mathbf{P}$
  create edges from $p0$ to $p \in \mathbf{P}$
  create edges from $p \in \mathbf{P}$ to $p1$
end for

Algorithm 3 Merge

// r_i : revision number
// G_i : DAG of revision i
// $T_i(v)$: sub-tree with root v in DAG G_i
// $V(G)$: nodes in DAG
// $\mathbf{C}$: conflict list
// assume users are merging branch revision r_j into trunk revision r_i
search for r_0, common ancestor of r_i, r_j, and its DAG, G_0
$G_d = G_j - G_0$
//handle different structure in graph
for all $v_j \in V(G_d)$ (in BFS order) **do**
  check for dependency between v_j and v_i, where $v_i \in V(G_i) - V(G_0)$
  if dependency exists, push sub-tree $(T_i(v_i), T_j(v_j))$ into $\mathbf{C}$
  else, insert v_j into G_i using Algorithm 1
end for
invoke the conflict solver interface and obtain the merge options for the users
for all sub-tree pair in $\mathbf{C}$ **do**
  push the sub-tree pair in the order specified by the user using Algorithm 1
end for

5.1 Operation Recorder/Replayer

Our prototype revision control system is built upon GIMP. At its current version 2.7, GIMP has not yet provided any API for command logging, as in its current architecture all commands are tightly bound with and invoked directly from user interface. As a result, to obtain fine-grained command log, we manually hard wired commands to our operation recorder and record useful information of the command including its name, parameter and working region. Hopefully, operation recording and replaying would be much easier after GIMP developers implement the long desired feature of macro recording. In our prototype system, important GIMP operations listed in Table 1 are supported.

On the other hand, operation replay can be easily achieved, thanks to the procedure database (PDB) architecture in GIMP. Most useful high-level image editing functions are registered to the PDB and can be invoked by the plug-in. However, with the rapid evolution of GIMP, not all new functions have been added into PDB properly. For example, functions related to brushes are out-dated in PDB and one will have to register such functions manually if she would like to achieve operation replay.

5.2 DAG Repository

Three main components, **log cleaner**, **graph builder** and **graph merger** are responsible for maintaining the DAG repository.

Log cleaner The log cleaner removes redundant and unnecessary commands and group identical operations together, similar to the one proposed in [Grabler et al. 2009]. Redundant command logs often occur while the user is exploring the design space in a trial and error manner such as positioning an object in different places or filling a region with different colors. The log cleaning algorithm is quite simple; we linearly scan whole command log, locate chunk of identical commands and keep only the latest one. Note that we deploy a relatively conservative policy here to preserve the whole image editing process as much as possible.

Graph builder The graph builder is responsible of building the DAG from the parsed log. The topology of the DAG conveys the spatial, temporal and semantic dependency between operations. Spatially independent operations, such as those applied on different layers or regions, are located in independent paths. Operations applied on the same object/region but are semantic independent, such as translation and deformation, are put into parallel paths with the same ancestor and children nodes. The directional edges show the temporal order of the commands. Here we first introduce a node insertion routine considering spatial and temporal relationship followed by an algorithm to separate the semantic independent operations into parallel paths.

First, the DAG is initialized with a single source node, which could be a blank canvas or an existing image to be modified. Then for each operation node c to be inserted into the DAG, we search each independent path of DAG for the possible parent node p that is spatially overlapped with c and is the latest entries for that path. If p exists in the path, an edge $p \rightarrow c$ is inserted. Note that there is always a root node, which could be the source image or the latest revision of the image, whose region of interest covers whole image and thus there is at least one such node p in the DAG. Pseudo code for the node insertion routine is provided in Algorithm 1.

Figure 1d is an example DAG built from Algorithm 1. As shown, operations applied to different regions are clustered into two independent sub-graphs. At this point, only spatial and temporal, but not semantic, dependencies between nodes are considered.

To consider semantic dependency, we categorize operations into five different classes as shown in Table 1 for which the first three rows are semantically independent (see also [Su 2007]). Semantically independent operations applied on same object or region are represented as parallel paths in a DAG with the same ancestor and children nodes to convey more precise dependencies as well as to provide more flexibility in non-linear exploration and navigation.

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/remove/duplicate layer, layer mask
brush	brush, pencil, eraser

Table 1: Supported operation classes.

Figure 1f is the resulting DAG considering both spatial-temporal relationship and semantic dependency. The problem here can be viewed as trying to find out as many clusters containing nodes with semantic independent classes as possible. Here we use a straight forward greedy algorithm that starts from the DAG root and greedily looks for its neighbors for the operations that are applied on the same area and belong to independent operation class. The pseudo code is listed in Algorithm 2. Note that this is just a fast heuristic algorithm and might not be the best solution (find out largest number of clusters). But the result is guaranteed to be correct (parallel paths contain only operation nodes within independent classes). To avoid distracting topology changes and save computation time, Algorithm 2 is applied at the end of user session or as user demands.

DAG merger The DAG merger is responsible for maintaining the consistency between DAGs belong to different branches and revisions. The possible usage scenarios have been discussed in Section 4.2, and here we describe the merge algorithm.

With the underlying DAG, the merge algorithm is quite straight forward. We first calculate the difference graph between two DAGs. Then we tried to insert the nodes in the difference graph into the trunk (assume we are merging branch into trunk) with Algorithm 1. If conflicts are detected, the nodes and their sub-trees are pushed into the conflict list and wait for user's decision on the insertion order. The detailed pseudo code is provided in Algorithm 3.

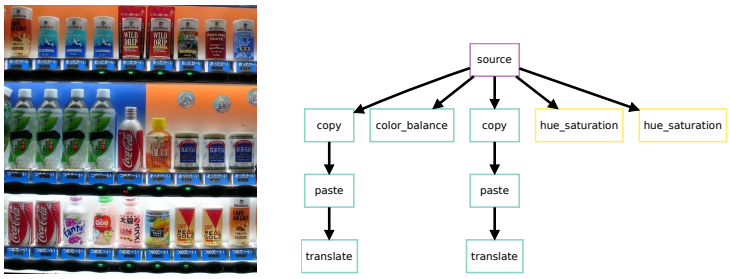
Note that for two graphs without one-to-one correspondence, finding graph difference is equivalent to subgraph isomorphism and is proven to be NP-complete [Cook 1971]. While in our case, nodes share consistent labels (operation ID provided by the operation recorder), the algorithm thus becomes straightforward.

Node cache Each DAG node stores relevant information for the corresponding operation, including its parameters and region mask. To facilitate non-linear exploration, we allow the storage of multiple sets of parameters (e.g. different brush colors) with respect to each DAG node. Our system also allows partial or complete storage of image data at user selected revision-tree or DAG nodes to accelerate history navigation.

6 Results and Evaluations

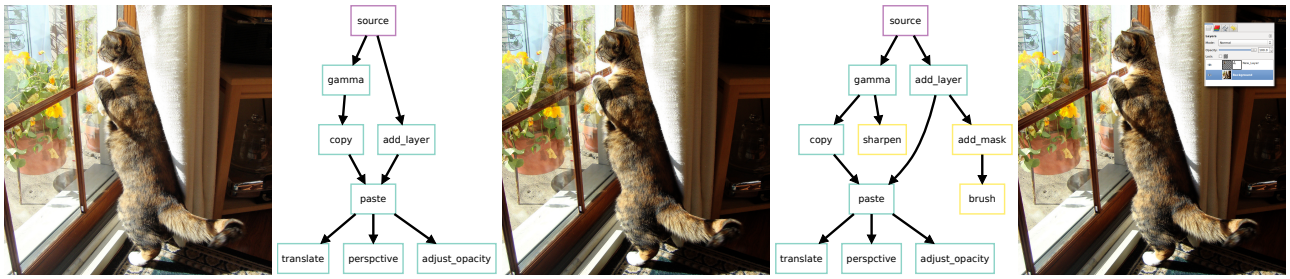
In this section, we demonstrate more examples and evaluations of our system, for both editing an existing image or digital painting/sketching from scratch. In addition to the basic revision control, we also demonstrate potential usage of our system for nonlinear replay and exploration. Finally, we compare the speed/storage performance of our system against other existing mechanisms, and perform a user study with the collaborating artist and users.

(a) source (b) trunk DAG (c) trunk revision (d) branch DAG (e) branch revision



(f) merged result (g) merged DAG

Figure 9: A merge example. (c) and (e) are two parallel revisions. In (c), colors and arrangements of cans on second and third rows are modified. In (e), the label colors and cans arrangement on first and second row are modified. (f) shows the merged result with **trunk** option. Note the independent paths between DAGs are merged automatically.



(a) input (b) create reflection (c) intermediate (d) layer mask & sharpen (e) final

Figure 10: An example of creative art with layer composition and masking. The achieved effects include the creation of cat shadow on the window (c) with desired occlusion order (e). Note that *paste* node is connected to both *copy* node and *add layer* node because it is copied from original source and pasted into the newly created layer.

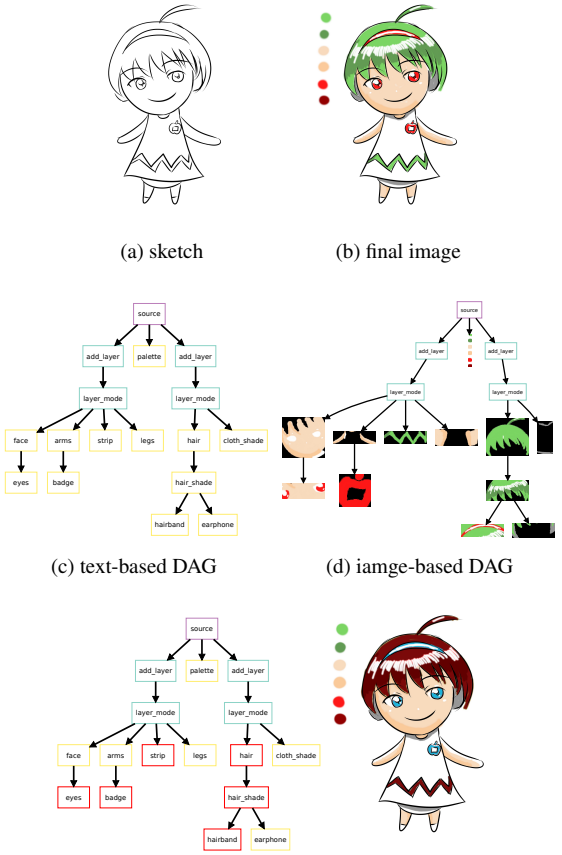
Image editing Here we show a parallel editing example and its merge result (Figure 9). Starting from source image (Figure 9a), we have two parallel editing revisions. In Figure 9c, the user perform two clone operations (coke on third row and tea cans on second row) and one color adjustment operation (central milk can on third row). The corresponding DAG is shown in Figure 9b. In branch Figure 9e, the user modify the label color from red to blue on first and second rows, perform a clone operation on the tea can on second row. In this example, we choose the **trunk** option for merge and obtain the final merge result (Figure 9f). With our merge algorithm, the independent operations are automatically merged. Note that we show DAG here for illustration purpose, users can perform the merge operation without the knowledge of underlying DAG.

Figure 10 is an image composition example. We created a fake reflection of cat on the window to enhance the richness of the photograph. First, gamma adjustment is applied, then we copy the cat into another layer with proper transformation and opacity adjustment (Figure 10c). Finally, we use a layer mask to mark the correct occlusion relationship between reflection and window then apply a sharpen mask on the cat (Figure 10e).

Digital sketching Figure 11 & 12 are digital sketching examples recorded from professional artists using our system. In Figure 11, the artist first sketched the figure (Figure 11a). He then prepared a color palette (those colorful dots beside the images) to paint the figure in the order of face, eyes, arms, hair, legs, and cloth. The final result is Figure 11b with the corresponding annotated DAG in Figure 11c and Figure 11d. Figure 12 is another example of the digital sketching and non-linear play-back.

Nonlinear replay and exploration An additional bonus feature of our system beyond the basic revision control is non-linear replay and exploration. With our system, it is possible to review or change the parameters of the operations and replay them non-linearly to explore the design space as shown in Figure 11f and Figure 12f. Explored paths are visualized as nodes with red color in the DAGs. Unlike normal image processing techniques that directly modify the final image, with recorded user operations, our system allows a richer space for experimentation and exploration and received some positive comments from our initial user study.

Performance The storage consumption of our system is particularly small compared to other image editing and revision control systems, such as GIMP (native .xcf file), GIT and SVN (Subversion), as shown in Table 2. For all figures, we divide the whole editing process into four revisions, and commit them to the revision control server. The storage overhead of our system mainly comes from the cached thumbnail image. The overhead of internal data



(a) sketch (b) final image

(c) text-based DAG (d) iamge-based DAG

(e) explored nodes (f) nonlinear replay

Figure 11: A digital sketching example. With semi-automatic aggregation method and artist's annotation, the final DAGs in (c) and (d) provide clear and meaningful information about the structure of the image and the sequence of operations. The result in (f) is produced by non-linear exploration on nodes visualized with red border color in (e).

structures for GIT and SVN are not precisely calculated here, but our advantage on storage size is clear.

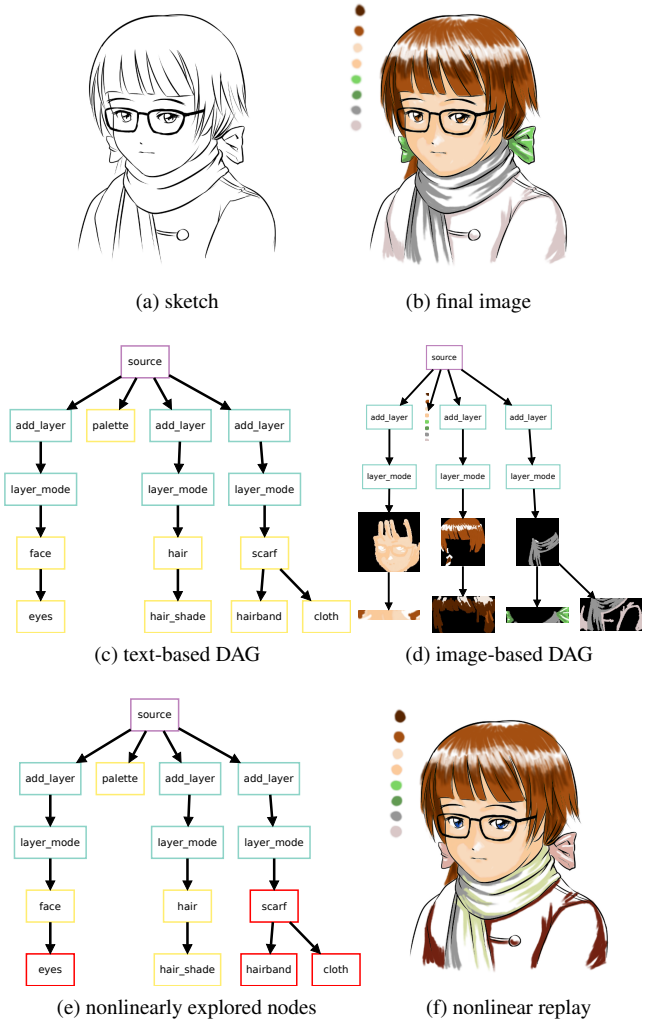
Regarding computation speed, our system is very efficient and runs at interactive speed, and users of our system have not found any slow down compared to the original GIMP or other revision control systems.

	input	# op	GIMP	SVN	GIT	our
Figure 1	502	11	2.7K	2.1K	2.0K	640
Figure 6	246	4	818	588	632	360
Figure 7	276	10	972	1.2k	1.2k	420
Figure 8	1.6	672	267	224	180	73
Figure 9	238	12	900	1K	1K	415
Figure 10	945	11	3.5K	3.7K	3.6K	1.3k
Figure 11	377	649	2.3K	2.4K	2.5K	652
Figure 12	425	1391	2.5K	2.7	2.7K	775

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

User study We perform some initial user study with one professional illustrator and two CS graduate students, who possessed less experience on photo retouching but are both familiar with software revision control system (SVN and CVS).

Unlike the CS major students, we found that the participating artist had a difficult time with the concept of nonlinear revision history.



(a) sketch (b) final image

(c) text-based DAG (d) image-based DAG

(e) nonlinearly explored nodes (f) nonlinear replay

Figure 12: A more complex digital sketching example. The artist first performs a digital sketch (a), and paints colors through a sequence of operations (c) with the final result in (b). After that, the artist could nonlinearly explore the design space. As marked in (e), the user lightens the hair shading a little bit, and changes the color of hair band, scarf and cloth. The nonlinearly explored result is shown in (f).

For he rarely performs parallel editing with others on one painting and is used to save the versions as files with different names.

In our early design, we directly expose the recorded command logs and the generated text-based DAG to the users. However, the participants commented that it is difficult for them to establish the connection between the DAG and the image intuitively. And it leads to our current design of image-based DAG representation and the design principle to minimize the user's direct interaction with the fine-grained DAG, e.g by showing only the high-level reversion tree for primary revision control tasks and DAG node selection from the image.

Our collaborating artist is especially interested in the stroke-by-stroke replay and non-linear playback functions. For the former, he commented that it is generally difficult to deduce the correct color overlay from the final flattened image, and with our system, one can truly appreciate other artists' technique and more importantly one's own drawing logic. For the later, he commented that it can save him lots of time to explore the possible color style candidates in his mind. Typically, he would paint some rough color blocks for

experimentation and look for possible color combinations while in the end find out that the details are not satisfying. With in our system, he could more boldly paint without fear since he knows that he can easily switch to different color scheme and look for the best outcome afterward.

7 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 an arbitrary collection of image editing software. It is possible to extend our system for supporting multiple tool sets by incorporating the tool/software id/version into each operation node. Beyond this, however, we might still need to manually integrate our system with each tool, a potentially tedious process. A potential future work is to design a universal revision control interface (or hookup) to facilitate the automatic integration of our system with different tools.

Due to resource constraints we have recruited only one professional artist and two CS grad students for our user study. We plan to put our source code in the public domain after the publication of our paper. Doing so would allow us to gather more feedbacks from the community to further refine our UI and system designs.

Within the scope of this paper we have focused mainly on images, but we believe similar principles might be 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.

References

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

AVID TECHOLOGY, 2009. Alienbrain. `http://www.alienbrain.com/`.

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. *Visualization Conference, IEEE 0*, 18.

BERLAGE, T. 1994. A selective undo mechanism for graphical user interfaces based on command objects. *ACM Trans. Comput.-Hum. Interact. 1*, 3, 269–294.

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 Papers*, 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.

GRABLER, F., AGRAWALA, M., LI, W., DONTCHEVA, M., AND IGARASHI, T. 2009. Generating photo manipulation tutorials by demonstration. *ACM Trans. Graph. 28*, 3, 1–9.

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., 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.

HUNT, J. W., AND SZYMANSKI, T. G. 1977. A fast algorithm for computing longest common subsequences. *Commun. ACM 20*, 5, 350–353.

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.

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.

KELBY, S. 2005. *The Photoshop CS2 Book for Digital Photographers*. New Riders 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.

KONSTANTINIDES, K., AND RASURE, J. 1994. The Khoros software development environment for image and signal processing. *IEEE Transactions on Image Processing 3*, 3, 243–252.

KURIHARA, K., VRONAY, D., AND IGARASHI, T. 2005. Flexible timeline user interface using constraints. In *CHI '05: CHI '05 extended abstracts on Human factors in computing systems*, 1581–1584.

KURLANDER, D., AND FEINER, S. 1991. Editable graphical histories: the video. In *CHI '91*, 451–452.

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

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

MENG, C., YASUE, M., IMAMIYA, A., AND MAO, X. 1998. Visualizing histories for selective undo and redo. In *APCHI '98: Proceedings of the Third Asian Pacific Computer and Human Interaction*, 459.

MYERS, B. A., AND KOSBIE, D. S. 1996. Reusable hierarchical command objects. In *CHI '96*, 260–267.

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.

PARKER, S. G., AND JOHNSON, C. R. 1995. Scirun: a scientific programming environment for computational steering. In *Supercomputing '95*, 52.

SCHROEDER, W., MARTIN, K., AND LORENSEN, B. 1997. *The Visualization Toolkit, Third Edition*. Kitware Inc.

SEITZ, S. M., AND BAKER, S. 2009. Filter flow. In *ICCV '09*.

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.

SU, S. L. 2007. Visualizing, editing, and inferring structure in 2d graphics. In *Adjunct Proceedings of the 20th ACM Symposium on User Interface Software and Technology*, 29–32.

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.

THE GIMP TEAM, 2009. Gimp: Gnu image manipulation program. `http://www.gimp.org/`.