This is the pre-print version
DOI: https://doi.org/10.1007/s11042-019-7735-9

Multimedia Tools and Applications manuscript No.
(will be inserted by the editor)

RECODE: Revision Control for Digital Images

Fabio Calefato · Giovanna Castellano* ·
Veronica Rossano

Received: date / Accepted: date

Abstract Revision control is a vital component in the collaborative development of artifacts such as software code and multimedia. While revision control has been widely deployed for text files, very few attempts to control the versioning of binary files can be found in the literature. This can be inconvenient for multimedia 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, respectively consume significant storage and complex semantic information. To overcome these limitations, in this paper we present RECODE, a revision control system for digital images. It stores revisions in the form of a DAG (directed acyclic graph) where nodes represent editing operations, and edges describe the spatial and temporal relationships between operations. Being integrated with GitHub, the largest project hosting platform, RECODE also facilitates the artistic creation process of distributed teams with different workflows that include image editing and digital painting. A preliminary user study was performed to assess the perceived usability of the proposed system.

Keywords: Multimedia design, image editing, revision control, collaborative design, digital painting.

1 Introduction

In multimedia design and development, there is a wide range of contents such as text, images, video, and audio that need to be created and edited. Recently,

F. Calefato
Computer Science Department, University of Bari, Via Orabona 4, 70125 Bari, Italy
E-mail: fabio.calefato@uniba.it

*G. Castellano (corresponding author)
Computer Science Department, University of Bari, Via Orabona 4, 70125 Bari, Italy
E-mail: giovanna.castellano@uniba.it

V. Rossano
Computer Science Department, University of Bari, Via Orabona 4, 70125 Bari, Italy
E-mail: veronica.rossano@uniba.it

collaborative forms of multimedia development have revealed useful for authoring, editing, collecting, and producing digital content [20, 7, 9].

When the development of multimedia is carried out in collaborative and integrated design environments, *revision control* (or version control) becomes essential to calculate, represent, and store differences between successive versions of the developed digital objects. Indeed, the development process can involve many authors with potentially different skills. The current paradigm of collaborative editing relies on sharing files between stakeholders upon each revision. This requires maintaining consistency of the versions and dealing with concurrent edits in the same part of a multimedia artifact. Hence, the need for revision control in digital artifacts comes in two different forms: ① maintain a versioned history of changes applied to artifacts and ② enable the creation of several concurrent variants of the same artifact as well as their consolidation.

However, so far revision control has been widely deployed for text files, while visual media such as diagrams and pictures have received considerably less attention [16]. Even popular, commercial tools for image editing such as *Adobe Photoshop* offer little or no support for managing variation in the produced artifacts, forcing users to employ basic techniques to track multiple versions of their work, such as merely creating multiple copies of the picture file. Also, very few methods are available for the efficient storage of modifications of visual artifacts and in general of any binary content. Existing strategies such as storing whole files for individual revisions or simple binary deltas could consume significant storage and obscure semantic information. This can be inconvenient for multimedia applications that use a significant amount of binary data, such as images, videos, and animations.

In this work, we present *RECODE* (REvision COntrol of Digital imagEs), a tool aimed to support image version control and collaborative creativity in digital multimedia projects. Inspired by version control systems for source code, such as *Git* and *Subversion*, where branching (i.e., creating a different version of a stored artifact) is a natural operation, we adopt the concept of paths and nodes in a graph to store persistent states over time owned by a particular image file. The core idea of the RECODE system is to store the history of editing operations applied to an image using a *Directed Acyclic Graph* (DAG) [2, 21], where each vertex contains the result of a performed editing operation. This representation is suitable for nonlinear revision control since the model has intrinsic support for collaborative editing, including branching and merging paths edited by multiple contributors. Also, the model does not require storing the whole modified image.

The remainder of the paper is organized as follows. In Section 2 we introduce the problem of revision control in visual artifacts as a background of our work. Section 3 overviews recent state-of-art works related to the proposed tool. In Section 4 we describe the proposed tool. Section 5 provides experimental results on usability tests. Concluding remarks are given in Section 6.

2 Background

Revision control systems are widely used tools in software development primarily aimed at managing versions of software source code during implementation [17]. In general, revision control tools assist the management of evolving digital artifacts, providing features that allow users to track intermediate revisions of artifacts and

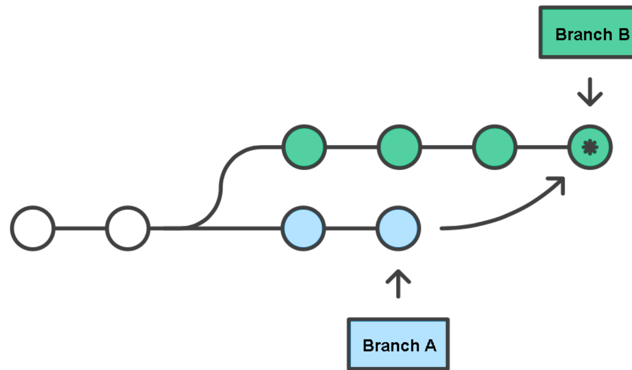

Fig. 1 An example of revision control with two parallel branches (A and B) created from a common ancestor and then merged.

their relations. Revision control systems also facilitate distributed and simultaneous content creation through four basic operations: diff, patch, branch, and merge. The `diff` operation is used to extract the differences, or deltas, between any two versions of a given artifact. The `patch` operation is used to generate a specific version of an artifact based on a delta. The `branch` operation is used to duplicate an artifact under revision so that modifications can happen in parallel and independently along two branches (Fig. 1). Finally, a `merge` is used to reconcile into one artifact two branches (i.e., parallel modifications) with a common ancestor (Fig. 1).

Depending on the adopted history model, that is, how the history of versions and changes to artifacts is stored, revision control systems can be classified as *state-based* or *change-based* systems [14]. Change-based revision control systems (also known as operation-based) store the operations actually performed between two succeeding versions in the revision control system. Instead, state-based systems store the history of changes as revisions of the versioned artifacts as they existed at various points in time. Most existing general-purpose revision systems such as Git[1] and Subversion (SVN)[2] employ a state-based model. These modern revision control tools save storage space by computing and persisting only the difference (i.e., delta) between succeeding revisions while preserving the full state of a few special versions – like initial or final (head) revisions [13].

Moreover, version control systems used for software development, such as Git and SVN are generally readily applicable to textual artifacts since source code is written in plain text format. However, these systems cannot handle the version control of binary files or offer limited built-in support for binary content [10]. When dealing with image data, in fact, they store separate images for each revision, thus wasting storage space. This issue alone hampers the adoption of revision control systems in managing digital images. Some extensions have been proposed (e.g.,

[1] https://git-scm.com
[2] http://subversion.tigris.org

Git LFS),[3] but they are meant to add support for storing large binary files rather than for versioning.

Besides, commonly used image-editing software tools offer minimal control for image versions. For example, *Adobe Photoshop* provides a version history that retains the undone operations on a stack. Hence, the user can quickly jump to any recent state of the image created during the current working session. However, high-level operations such as comparison (diff), branching, and merging of different versions are not available in Photoshop, nor other current drawing tools. A few other tools, such as *Adobe Drive* and *AutoDesk Vault*, offer version control for digital images, but they are proprietary and do not describe their specific model. Some more general version control tools, including *Perforce* and Git, offer support for image diff operations via services such as *GitHub*.[4] However, even when image deltas or low-level image information such as pixel-wise differences are used, they still lack sufficient high-level semantic information for reviewing, branching, merging, or visualization.

3 Related work

Version control systems manage content change in a digital artifact and maintain a history of its evolution due to successive editing operations. Recovering editing operations for binary data is more difficult than for text, posing a big challenge for collaborative editing of digital visual artifacts. To address this, different revision control systems for visual artifacts have been recently proposed in the literature. The common idea is to implement a nonlinear control of versions using a DAG. Using a directed acyclic graph, the focus is on paths of editing operations rather than on pixels or image objects, and challenges such as differencing (or *diff* in short) and merge are solved with graph operations. They also offer support for selective undo and nonlinear exploration, in which the user can adjust parameters to operations that have already been performed.

The first example of using a DAG for storing the history of operations (i.e., changes) and provide nonlinear revision control for binary image files is the seminal work by Chen et al. [8], who implemented a plug-in of the image editor GIMP,[5] which tracks user editing actions in form of graphs to visualize revisions and support branching and merging.

A similar work is the one by Zhao et al. [23], who developed *skWiki*, a wiki-based framework for collaborative creativity in digital multimedia projects, including different types of media (text, hand-drawn sketches, and photographs). The framework uses the concept of paths as trajectories of persistent state over time. A document or file is represented as a path hence it is stored as the ordered (and timestamped) sequence of document-specific operations that created and modified it. Paths are implemented using a database management system (DBMS) that stores individual paths as tuples in one table, and all of the operations in another using the path identifier as a primary key and including the revision serial number.

The process of image differencing and merging concerns not only 2D images, but also 3D digital assets. Albeit not directly comparable to RECODE, a couple of

[3] https://git-lfs.github.com
[4] https://github.com
[5] http://gimp.org

studies are worth mentioning because of the use of DAGs. Doboš and Steed [10, 11] proposed a general approach to serialize DAG structures representing 3D assets and store them in MongoDB, arguing that NoSQL databases are better suited for storing spatial information. Our implementation is similar in that we also serialize DAGs into a database, albeit we opted for MySQL since we deal with 2D (i.e., non-spatial) assets only. Wang et al. [22] proposed a 3D scene editor that builds upon the approach proposed by Doboš and Steed. Similarly to our work, they also extended an open source web-based editor (three.js),[6] store DAG structures into a database, and implement the typical features of a revision control system, including `commit`, `branch`, `diff`, and `merge`.

Instead of relying on DAG structures, da Silva et al. [19] implemented *IMUFF*, an image version control system that works at bit level – i.e., the delta between two images is computed as the set of bits that are changed across the two versions. Unlike DAG-based tools such as RECODE and skWiki, IMUFF has the advantage of being independent of the image editing tool. This, however, causes limitations too: ① a color image with a somewhat low resolution of 1024x1024 pixels requires over 4 million bitwise comparisons; hence, IMUFF has been designed to work only on architectures with high-end NVIDIA GPUs to take advantage of their massive parallel computational power; ② since there is no tracking of operations, as in RECODE and other DAG-based tools, IMUFF only supports the tracking of rigid transformations – i.e., differencing and merging works only for rotations, translations, and reflections – whereas tracking of filter effects such as blur and solarize is not supported.

Finally, besides academic prototypes, a few commercial solutions for image differencing and merging are also available. While primarily intended for source code revision, GitHub provides an image comparison tool that allows visualizing differences between stored images through a split pane view; the tool, however, does not support merging. *Araxis Merge*[7] is a commercial tool that supports both text and image differencing and merge. Several image formats are supported and differences are shown at a pixel level, as in the case of the *IMUFF* prototype by da Silva et al. [19]. *Abstract*[8] is a commercial tool that offers version control for files edited with Sketch,[9] a proprietary vector graphics editor for macOS. With respect to the collaboration workflow, Abstract is very similar to RECODE. Both tools allow creating projects in a repository (proprietary in their case, GitHub in ours), add images assets (plus any type of files in RECODE), storing only important revisions as entire binary files (instead of deltas), and branching/merging alternative versions of images, even those created by other team members who have access to the same project repository.

Inspired by the work proposed in [8], in [6] we introduced the first prototype of a revision control system for digital images that implements a hybrid approach combining the use of graphs with both state-based and change-based revision control. Like in Chen et al. [8] and other change-based systems [14], we also expose the history as a DAG to represent spatial, temporal, and semantic dependencies between successive recorded image-editing operations that are stored as graph

---
[6] https://threejs.org
[7] https://www.araxis.com/merge
[8] https://www.goabstract.com
[9] https://www.sketchapp.com

nodes. However, like state-based revision control systems, we also allow users to store select, important revisions as binary files, thanks to the integration with Git, so that users do not need to constantly reconstruct them by reapplying the entire sequence of operations that ultimately lead to their creation. The prototype presented in [6] was intended as a proof of concept for our hybrid model, hence it offered only a minimal set of graphical operations. In order to make our solution more robust, we developed a new version of our system that was obtained by integrating our hybrid model within a more sophisticated open-source image editor. As a result of this integration, in this paper, we present RECODE,[10] a tool for nonlinear versioning control to be used in collaborative works involving digital image editing.

Although inspired by the work proposed in [8], RECODE differs from it in the following aspects:

- Chen et al. [8] developed their prototype on the GIMP image editing software, which, however, does not provide an API to access the history of operations; therefore, they had to fork the code and develop a custom version that is now obsolete – i.e., it has never been updated to be on par with the official version. To avoid a similar issue, we developed RECODE by extending *miniPaint*,[11] an open-source, online image editor that, while being not as sophisticated as GIMP and Photoshop, provides native support for extensions as well as access to the entire history of edit operations. As such, RECODE promises to be more future-proof.
- RECODE is integrated with Git and GitHub, thus it enables teams to collaborate in the creation of both visual and text artifacts. Besides, our system works well with the Git LFS extension, thus allowing users to check out only the entire image files needed for the current task at hand, and just symbolic references for other images that are not needed.
- RECODE can be seamlessly used online and offline. In fact, despite being a web application developed in JavaScript, we provide a Docker image that allows to execute the tool locally and to synchronize the local Git clone with the remote origin repository as soon as an Internet connection is again available.
- Finally, instead of using custom formats as in [8], RECODE stores meta-data and project-related information using files in standard JSON objects, which are serialized into a MySQL database.

Finally, the RECODE tool satisfies the following fundamental properties required to software tools supporting collaboration around digital artifacts [23]:

1. ***Mobility*** – *platforms should be accessible from everywhere*; RECODE leverages a responsive user interface that adapts to the screen size of mobile devices such as tablets.
2. ***Collaboration*** – *platforms should allow collaboration between geographically and temporally distributed participants*; RECODE workflow can support collaboration between teams of distributed people who can work remotely and also online/offline by running the tool locally via a Docker container.
3. ***Revision history*** – *platforms should enable the tracking and versioning of digital assets*; RECODE supports the most important features available in revision control systems thanks to its tight integration with Git and GitHub.

[10] https://github.com/RECODE2/recode

[11] https://github.com/viliusle/miniPaint

4. ***Transparency*** – *platforms should hide revision systems complexity from users*; RECODE keeps complexity to a minimum by providing an intuitive UI to execute revision system commands such as `commit`, `branch`, and `merge`.
5. ***Rich media*** – *platforms should allow for the presence of different media types*; RECODE allows teammates with different interests (e.g., developers, illustrators, writers) to collaborate using the same Git repository, which can archive all types of artifact, whether textual or binary.
6. ***Divergent creativity*** – *platforms should enable the creation and tracking of different versions of the same artifacts*; RECODE supports divergent creativity by enabling branching in non-linear revision – i.e., representing alternative versions of an image as alternative paths in a DAG – as well as the differencing and merging of such 'divergent' artifacts.

4 The revision control system

The core data structure of RECODE is a Direct Acyclic Graph (DAG) that is used to store the revisions as deltas (Fig. 2). DAG nodes represent image editing operations with relevant information such as the type of operation and its parameters, the author who applied the operation, the time of the application and eventual notes. DAG edges represent the relationships between the operations.

A (directed) sequential path between two nodes implies a spatial and/or semantic dependency between operations and the path direction gives information about their temporal order. Spatial dependency considers the spatial relationships between operations. Two operations are spatially independent if they are applied to non-overlapping regions. For example, *drawing a shape* and *coloring it* are spatially dependent operations. Conversely, *drawing a shape* and *coloring another existing shape* are independent operations. Semantically independent operations are rigid transformations (e.g., translation, rotation), deformation (e.g., scale, shear, perspective), color adjustment (e.g., hue, saturation, brightness, contrast, gamma) and filter (e.g., blur, sharpen).

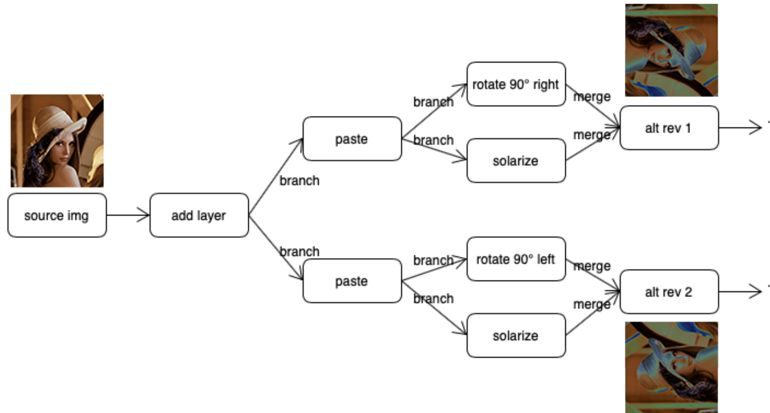

Fig. 2 An example of recorded editing operations stored as a DAG.

Table 1 Some of the most common image-edition operations leading to the creation of a new node in the DAG.

Type	Operation
Rigid Transformation	Mirror, Flip, Transpose
Deformation	Scale
Color and Filter	Histogram, Brightness, B&W, Sepia, Invert, Solarize, Posterize
Edit	Crop, Text, Reset
Brush	Brush
Load image	New, Import

Multiple parallel paths between two nodes imply independent operation sequences, namely those that apply on disjoint regions of the image. The DAG records the user editing operations and dynamically grows as more revisions are committed. Each revision in our system is a sub-graph of the DAG originating from the first 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. It should be noted that in our system, the DAG encodes only actions, not whole images. Table 1 lists the editing operations that lead to the creation of a new node in the DAG.

In the following, we briefly describe the main revision control commands implemented in RECODE.

4.1 Revision control commands

Based on the DAG data structure, RECODE provides the primary mechanisms for automatic resolving and merging multiple revisions with potential conflicts, as well as a user interface that allows manual change and intervention on automatically merged images (see Sect. 4.2).

The implemented revision control commands include review, addition, branch, merge, difference, and conflict resolution. All these commands are offered through a friendly user interface.

4.1.1 Diff

While the classic line-based `diff` command [13] is commonly used to extract differences between text files, there is no such well-defined difference tool for images. Among 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* [4]), 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, following the idea of Chen et al. [8], RECODE realizes an informative `diff` by leveraging all the relevant high-level information recorded in a DAG. As further illustrated in Sect. 4.2.1, all data structures are stored as JSON objects,

Fig. 3 An example of merging two versions of the same image.

including DAGs and image metadata. Therefore, to implement the `diff` operation between two image revisions, RECODE relies on the `fast-deep-equal` package[12] of Node.js to identify changes in all the fields of a couple of JSON files and stores the differences (i.e., the delta) in a new JSON file. This delta is designed not only to reduce storage space but also to allow for the fast identification of changes and conflict resolution during the integration of changes – i.e., the `merge` operation described next.

4.1.2 Merge

The `merge` operation is performed in version control systems to consolidate two artifact revisions created in parallel branches. Unlike text, merging two image versions (Fig. 3) requires complex procedures in order to identify the difference between them. In text files, changes are identified through line-by-line comparison. Instead, with binary files, it is difficult to define which parts of an image have been changed.

To implement the merge operation in RECODE, we relied on the `merge-json` package[13] for Node.js. The package is used to merge the editing history of two different images recorded into their corresponding JSON metadata (Fig. 4). Thus, in the resulting merged JSON file, all the operations are sequentially ordered in one branch. This is possible because all operations applied to an image are recorded using a growing integer as unique `id` and an attribute `order` to define the priority when changes involve an overlapping area of the image. Further information on the structure of JSON files are given in Sect. 4.2.1.

4.1.3 Git commands

The RECODE tool is developed to use Git repositories shared online on GitHub. When a new project is started, the URL of the associated GitHub repository must be provided, along with the path of a local folder where the content of the remote repository will be cloned locally. Then, it is possible to execute all the Git commands to control the image revisions as binary files both locally and remotely. The Git commands reviewed next are made available from the UI (see Sect. 4.2 for more).

Since Git is a distributed revision system, the storing and sharing of artifacts are performed in separate steps; specifically, the `git commit` command is used to store files locally, whereas the `git push` command is used to send local changes to the shared remote repository, so that collaborators can retrieve them. Because this

[12] https://www.npmjs.com/package/fast-deep-equal
[13] https://www.npmjs.com/package/merge-json

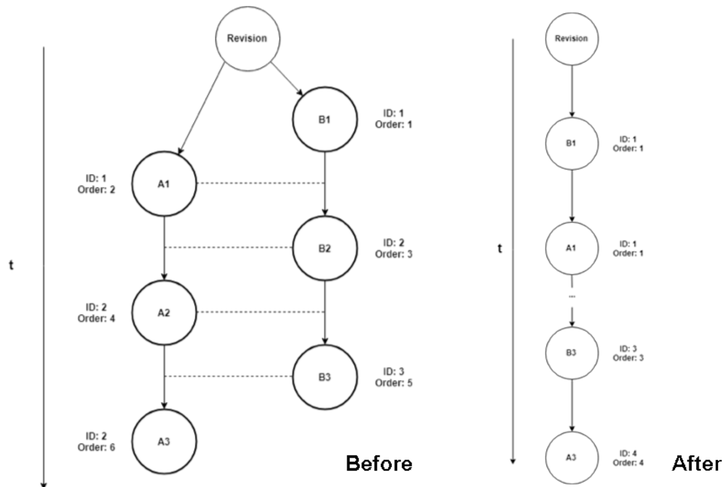

Fig. 4 RECODE: An example of a merge between image A and B (before), with operations and resulting revisions ordered sequentially in a single branch (after).

distinction may increase the cognitive load for the non-developer users, RECODE has been designed to simplify the workflow and hide away such complexity. Initially, when the repository is empty, the user adds the initial image and commits it as revision 0. To this end, RECODE provides a high-level 'Add revision' command from the revision control window, which locally stores the first image revision (i.e., the initial node of the DAG) as a binary file; the file is committed to the local Git repository by issuing under the hood a `git commit` command.

Committing revisions is one of the most frequently used revision control commands. To save the current work progress as further binary revisions, users can select the high-level command 'Commit' from the revision control window, which will save changes as a binary file both locally and remotely – i.e., by issuing first a `git commit` command, followed by a `git push`. Although users can save revisions whenever they like, it is generally unnecessary for them to do so in an action-wise fine-grained fashion, since RECODE can record all the actions and flexibly visualize them as a DAG. As a general guideline, users proceed to commit revisions when one of the following two conditions is met: 1) some milestone of the work is achieved or 2) 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.

Furthermore, our system is compatible with the Git LFS (Large File Storage) extension that allows, when enabled, the storage of large files to a separate repository. The original repository will only contain pointers to the actual large files. Thus, a user can decide to download these files only if need be. As such, the use of Git LFS is recommended for speeding up the access to project repositories that host many very large files as in the case of images.

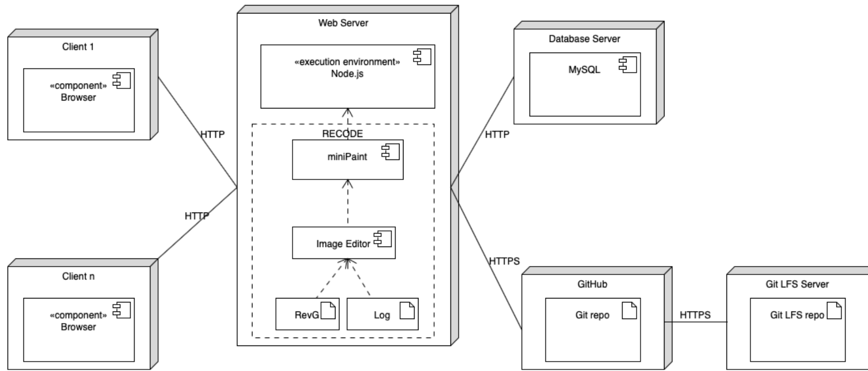

Fig. 5 Deployment diagram showing RECODE architecture and the distribution of its component to deployment targets.

4.2 Architecture

RECODE is based on a client/server architecture that includes the following components (Fig. 5):

- the *Web server*, which executes the Node.js JavaScript runtime environment, where the miniPaint image editor is executed.
- the *Image Editor*, which extends miniPaint adding the components that enable the revision control workflow, and in particular:
  - the *Logger*, which silently records user-editing actions in the background in the form of JSON log files.
  - the *Replayer*, which replays actions in the image editor starting from the deltas stored in the log files;
- the *Database*, which stores all the information needed to create the revision graph and implement the revision control functions;
- the *Git and Git LFS Servers*, which physically store images files as binary artifacts and enable a distributed revision control.
- *Client browser*, which renders the tool UI and triggers the revision control commands.

In the following, we describe the custom components of RECODE in further details.

4.2.1 Image editor

To develop the image editor, we chose to extend an existing open source tool, *miniPaint*, which provides for an easy GUI and a wide set of editing tools, filters, and rigid transformations, which can be applied to images. Image editing is also based on layers and images can be saved in several common formats, including PNG, JPEG, BMP, and TIFF. miniPaint is a single page application that runs in Node.js, which is an I/O event-driven framework based on the Chrome JavaScript V8 engine. Despite Node.js is a server-side framework, miniPaint uses a specific module to compile the code in JavaScript and use it in a Web browser.

```
{
        "info": {
                "width": 800,
                "height": 720,
                "layer_active": 2
        },
        "layers": [
                {
                        "id": 1,
                        "type": "brush",
                        "opacity": 100,
                        "order": 1,
                        "data": [
                                [
                                        99,
                                        101
                                ]
                        ],
                        "color": "#5cabfb",
                },
                {
                        "id": 2,
                        "parent_id": 0,
                        "name": "bicchiere.jpeg",
                        "type": "image",
                        "opacity": 100,
                        "order": 2,
                }
        ],
        "data": [
                {
                        "id": 2,
                        "data": "data:image/png;base64"
        ]
}
```

Fig. 6 An excerpt of the JSON file generated by miniPaint.

A distinguishing feature of miniPaint is that it can natively export all the sequence of actions made on an image into a JSON file structured as follows (Fig. 6):

1. *info* contains information such as the file `name`, the software release version, and the `id` of the last active layer.
2. *layers* is a list of all the layers existing in an image; for each layer the following setting info are recorded: `id`, the `type` of applied operation, the `order` (priority) of the layer with respect to other layers, a `data` field containing all points where the editing operation was applied, and the `type` field, which can be `image` when a new image is loaded or the type of the specific editing operation applied (e.g., `brush`).
3. *data* contains the numeric `id` of the image, as well as its MIME type and encoding in the `data` field (in the example, the image is in PNG format with a base64 encoding).

4.2.2 Logger and Replayer

Two of the main components of RECODE developed to extend miniPaint are the `logger`, which records in text mode in the background all the actions that the user performed by a user while editing an image, and the `replayer`, which allows the

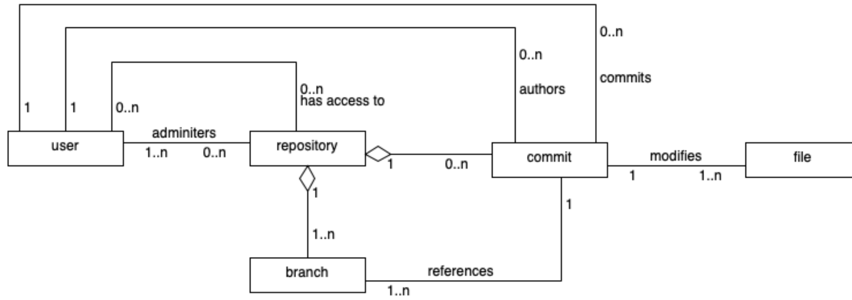

Fig. 7 A portion of RECODE conceptual schema to store the revision graph in the database.

'replay' of the actions stored in the DAG to visualize the final image in the editor. miniPaint easily support these two features since it can natively export into JSON format the sequence of actions applied to an image and is also able to portray an image in the editor by reading from its JSON file the actions performed by the user.

4.2.3 Database

A relational MySQL database (Fig. 7) has been adopted to ensure the persistence of metadata, thus maintaining the semantic, spatial, and temporal mutual dependencies between artifacts, namely to store all the information needed to create the revision graph and to implement the revision control functions. Using the database, we track information about the logged users (via GitHub, see Fig. 8), the list of repositories created by them, the commits made by different users sharing the same repository, and the branches generated from each revision. As described earlier, RECODE stores meta-data and project-related information using files in standard JSON objects, which are then are serialized into the database.

4.2.4 Git server

RECODE implements a hybrid approach combining the use of graphs with both state-based and change-based revision control. The DAG is used to represent spatial, temporal, and semantic dependencies between successive recorded image-editing operations that are stored as graph nodes. However, like state-based revision control systems, RECODE also allow users to store important revisions as binary files, thanks to the integration with Git, so that users do not need to continually reconstruct them by reapplying the entire sequence of operations that ultimately lead to their creation. This has been achieved by integrating RECODE with GitHub, a Git hosting provider, which allows sharing Git repositories. Thanks to the `git clone` operation, each user can download a copy of the repository from GitHub (via its API v3) and work locally and then commit back the changes.

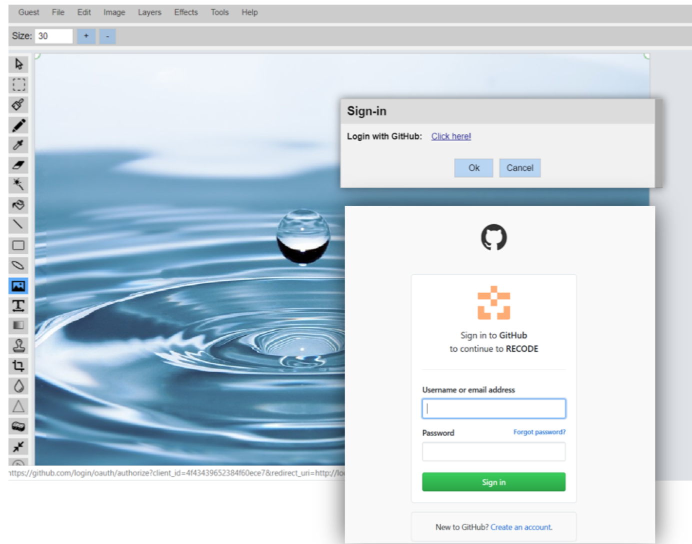

Fig. 8 Login window via GitHub with OAuth protocol.

4.3 Graphical User Interface

The GUI of RECODE is based on the GUI of miniPaint. Position and layout of the menus are the same, with a few changes applied to accommodate the implementation of the extra features. Since miniPaint is not integrated with GitHub, we added support for user login and repository management. After authentication, users can access the revision control functions using the 'Git' item in the menu bar. The submenu includes 'VCS' and 'Repository' items.

The 'VCS' (Version Control System) item groups the following commands:

- *Revision Graph*, to visualize the DAG of a specific image (Fig. 9). The browser sends the JSON to the server for processing JSON in order to reduce the impact in terms of memory on the client side. As explained earlier, only the differences between two consecutive revisions are stored in a JSON file. Accordingly, when the Revision Graph of an image is loaded, all the differences along the path to the initial revision (i.e., the yellow node in Fig. 9) are concatenated to create the resulting JSON representation of the image to be displayed. The DAG viewer is a key feature of RECODE, which offers a visual management interface for navigating the evolving graph structure, with alternative paths generated during the editing of different image revisions.
- *Add Revision*, to commit an initial image revision and start a new workflow (Fig. 10);
- *Commit*, to store the binary version of an image and share it with the other members of the team via GitHub. Each commit requires a mandatory textual description to allow collaborators to understand what changes has been made to the image.
- *Merge*, to combine different image revisions. This function is used to combine changes from either the same user or other teammates who have modified the

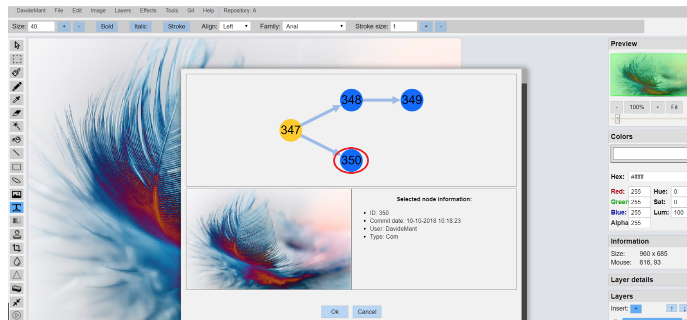

Fig. 9 GUI for the DAG viewer opened after a *Revision Graph* command.

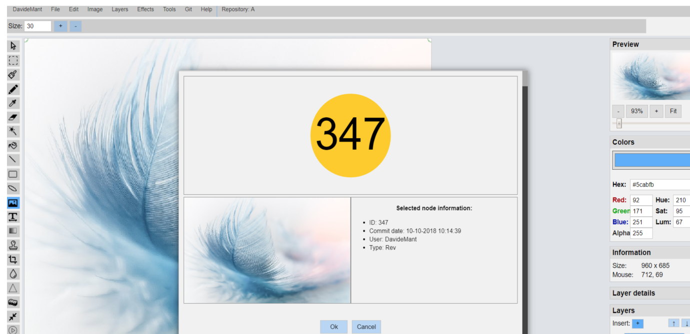

Fig. 10 GUI for the *Add Revision* command.

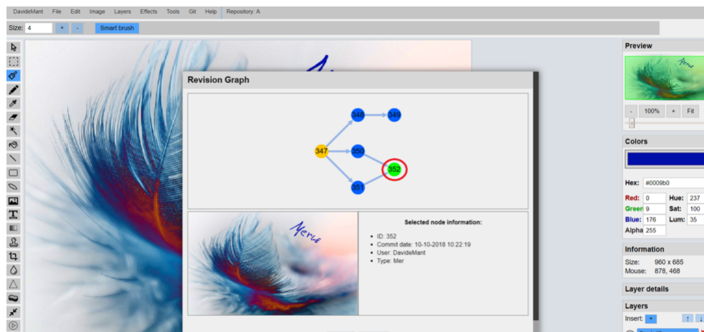

Fig. 11 Visualization of the DAG after the *Merge* command.

same image. This is graphically represented as merging two branches of the same graph by creating a new node that contains the sum of all contributions, as depicted in Fig. 11). Conflicts are solved by assigning a priority to each node in the DAG.

The 'Repository' menu item groups the commands related to the management of the GitHub repository in which all the images will be stored. The available functionalities are:

- *Create*, adds a new repository in GitHub;
- *List/Choose*, shows the list of user's repositories to choose which to work on;
- *Clone Git repo*, creates a local copy of the working Git repository;
- *Participating users*, lists the users that can contribute to the image editing process;
- *Invite user*, allows the owner of the working Git repository to invite other collaborators;
- *Delete user*, allows the owner of the repository to delete one or more participants.

5 User Test

In order to evaluate the system, a usability test has been conducted. The adopted procedure is compliant with the eGLU usability protocol used by the Italian Public Administration [1]. This protocol, which is one of the most used for simplified usability tests, has been enriched by the eGLU-M for mobile systems and by the Usability Glossary of WikiPA project [1]. The protocol has been defined in order to be adaptable to different kinds of software. In this case, the following documents related to eGLU 2.1 protocol have been used:

- data of participants;
- description of the task;
- questionnaire for computing the Net Promoter Score (NPS);
- questionnaire for computing the System Usability Scale (SUS);
- table of results.

5.1 Study participants

The participants involved in the study were 6 subjects interested in image editing. In particular, the participants were 2 graphic designers who usually use professionals tools for image editing, 2 computer scientists who are familiar with revision control systems, and 2 users interested in using image editors but without any specific background. As argued by Nielsen [15], 5 users are expected to find 85% of the usability problems of a system.

5.2 Assigned tasks

Each user was asked to complete the following tasks:

1. Create a project and a repository;
2. Open an image and commit the first Add Revision.
3. Apply three changes to the image and commit the result;
4. Starting from an image revision (a node in the DAG) create a new branch, apply some changes, and commit the result;
5. Visualize the information about a commit and load an image in the editor;
6. Apply the `merge` operator to unify the changes between two branches;
7. Invite a new participant to the repository;

8. Load an image, apply a change, and create a new revision.

We observed that the interaction with the system was not easy for all users. Some of them encountered difficulties using both the image editor and the version control commands. In particular, all users took a lot of time to carry out Task#2. As for Task#4, none of the users completed the task, and only 3 users completed Task#5. The other tasks were completed without difficulties. However, this was somewhat expected due to the different background of the participants. Indeed not all of the participants were familiar with both version control systems and image editors.

5.3 Net Promoter Score

After carrying out the tasks, the Net Promoter Score (NPS) was used to measure the users overall usefulness perception of the system. The NPS is calculated by collecting the answers to the question: "*How likely is it that you would recommend RECODE system to a friend or colleague?*" The answers were calculated using a 0-10 scale. Respondents can be *promoters* (score 9-10), i.e., enthusiasts of the system, *passives* (score 7-8), i.e., satisfied but unenthusiastic users, *detractors* (score 0-6), i.e., unhappy users.

The results show that 33.33% of the participants belong to promoters, 50% are passives and the 16.67% are detractors. Subtracting the percentage of *detractors* from the percentage of *promoters* yields the NPS, which can range from a low of -100 (if every customer is a *detractor*) to a high of 100 (if every customer is a *promoter*). In our case, the resulting NPS is 17, which is rated good according to existing literature [12].

5.4 System Usability Scale

The System Usability Scale (SUS) is a quick and reliable questionnaire for measuring the usability of an artifact (software code or multimedia) [5] and is currently used to evaluate the usability of different kinds of products and services. The questionnaire consists of 10 standard items,[14] using a 5-point Likert scale.

Scores given by the participants are processed in order to derive a single value using the following procedure. Let x and y denote the score given by the user for odd items (1, 3, 5, 7, 9) and even items (2, 4, 6, 8, 10) respectively. The SUS score z is computed using the following formula:

$$z = 2.5 \cdot [(x - 1) + (5 - y)]$$

The resulting value z ranges in $[0, 100]$ and represents the average satisfaction level of the user.

Based on existing literature [3], a SUS score above 68 can be considered above the average. In our case, an average SUS score of 55 was achieved (Fig. 12). This means that the RECODE tool needs to be improved in terms of usability. In particular, a very low score was obtained for the question "*I needed to learn a lot*

[14] https://www.measuringux.com/SUS.pdf

of things before I could get going with this system." Arguably, one of the possible explanation for this result is that none of the subjects in the experimental sample had previous experience with the use of collaborative tools and, therefore, they might have found cognitively complex the execution of the collaborative workflow required by the experimental tasks. Indeed, the terms used to indicate commands in the current version of the interface are very specific and collaborative domain-dependent.

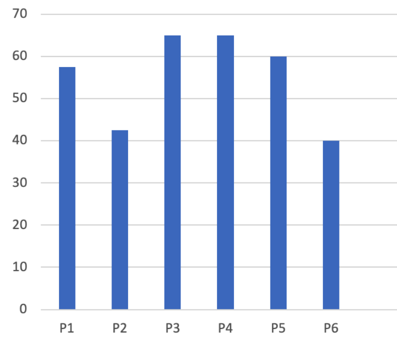

Fig. 12 Results of the usability test on 6 participants (P1,P2,...,P6).

6 Conclusions

In this work, we have presented RECODE, a tool for the revision control of digital images. Unlike other existing systems for image revision control, RECODE adopts a hybrid approach that saves user editing actions as direct acyclic graphs (to save storage space) but also allows users to save important milestones revisions as binary files. Thanks to the integration with the popular revision control system Git, our system is capable of supporting both collaboration and revision control. Hence, we argue that the tool can be adopted in distributed collaborative team works by virtually any audience involved in digital content creation, from web designers to engineers, researchers, and creative artists.

The usability study conducted with a few subjects provided initial evidence that our revision system has some usability problems. A possible cause can be found in the composition of the sample, since all the participants were not expert in collaborative activities. Moreover, we underline that the aim of this work was to develop a novel revision control system for multimedia, to fill a gap in the state of the practice. Hence, our work has focused so far more on the development of the tool that fulfilled its purpose, devoting less attention to other aspects such as the usability. Further work is currently in progress to improve the usability of the system, and in particular to adapt the interface terminology, which was one of the main difficulties reported by the users during the test session. To address this issue, we intend to apply the Participatory Design Approach [18], so as to involve the final users, namely graphical designers and multimedia developers. In this way,

we expect to make more intuitive the usage of the tool, thus increasing the user satisfaction. Moreover, further tests have been planned to evaluate our system by involving users in real collaborative scenarios of multimedia development.

Acknowledgement

This work is partially funded by the project "Creative Cultural Collaboration" (C3) under the Apulian INNONETWORK programme, Italy.

References

1. Gruppo di lavoro per l'usabilitá (glu), linee guida di design per i servizi web della pa (2018). URL http://www.funzionepubblica.gov.it/glu
2. Bang-Jensen, J., Gutin, G.Z.: Digraphs: theory, algorithms and applications. Springer Science & Business Media (2008)
3. Bangor, A., Kortum, P.T., Miller, J.T.: An empirical evaluation of the system usability scale. Intl. Journal of Human–Computer Interaction **24**(6), 574–594 (2008)
4. Bonanni, L., Xiao, X., Hockenberry, M., Subramani, P., Ishii, H., Seracini, M., Schulze, J.: Wetpaint: scraping through multi-layered images. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pp. 571–574. ACM (2009)
5. Brooke, J., et al.: Sus-a quick and dirty usability scale. Usability evaluation in industry **189**(194), 4–7 (1996)
6. Calefato, F., Castellano, G., Rossano, V.: A revision control system for image editing in collaborative multimedia design. In: Proceedings of the 22nd International Conference on Information Visualisation (IV 2018), pp. 512–517 (2018)
7. Chen, C.W., Peng, J.W., Kuo, C.M., Hu, M.C., Tseng, Y.C.: Ontlus: 3d content collaborative creation via virtual reality. In: International Conference on Multimedia Modeling, pp. 386–389. Springer (2018)
8. Chen, H.T., Wei, L.Y., Chang, C.F.: Nonlinear revision control for images. In: ACM Transactions on Graphics (TOG), vol. 30:4, p. 105. ACM (2011)
9. Claman, T.H., Coniglio, S.J., Daigle, S., Gonsalves, R.A., Wallace, R.C.: Methods and systems for collaborative media creation, us patent no. 9864973 (2018)
10. Doboš, J., Steed, A.: 3d revision control framework. In: Proceedings of the 17th International Conference on 3D Web Technology, pp. 121–129. ACM (2012)
11. Doboš, J., Steed, A.: Revision control framework for 3d assets. Eurographics (2012)
12. Grisaffe, D.B.: Questions about the ultimate question: conceptual considerations in evaluating reichheld's net promoter score (nps). Journal of Consumer Satisfaction, Dissatisfaction and Complaining Behavior **20**, 36 (2007)
13. Hunt, J.J., Tichy, W.F.: Addendum to delta algorithms: an empirical analysis. ACM Transactions on Software Engineering and Methodology (TOSEM) **7**(4), 449 (1998)
14. Kleine, M., Hirschfeld, R., Bracha, G.: An abstraction for version control systems. Universitätsverlag Potsdam (2012)
15. Nielsen, J.: How many test users in a usability study. Nielsen Norman Group **4**(06) (2012)
16. O'Sullivan, B.: Making sense of revision-control systems. Queue **7**(7), 30 (2009)
17. Ruparelia, N.B.: The history of version control. ACM SIGSOFT Software Engineering Notes **35**(1), 5–9 (2010)
18. Schuler, D., Namioka, A.: Participatory design: Principles and practices. CRC Press (1993)
19. da Silva Junior, J.R., Clua, E., Murta, L.: Efficient image-aware version control systems using gpu. Software: Practice and Experience **46**(8), 1011–1033 (2016)
20. Slaughter, D.S., Murtaugh, M.C.: Collaborative management of the elearning design and development process. In: Leading and Managing e-Learning, pp. 253–269. Springer (2018)
21. Thulasiraman, K., Swamy, M.: Acyclic directed graphs. Graphs: Theory and Algorithms **118** (1992)
22. Wang, Z., Cai, H., Bu, F.: Nonlinear revision control for web-based 3d scene editor. In: 2014 International Conference on Virtual Reality and Visualization (ICVRV), pp. 73–80. IEEE (2014)

23. Zhao, Z., Badam, S.K., Chandrasegaran, S., Park, D.G., Elmqvist, N.L., Kisselburgh, L., Ramani, K.: skwiki: a multimedia sketching system for collaborative creativity. In: Proceedings of the 32nd annual ACM conference on Human factors in computing systems, pp. 1235–1244. ACM (2014)