

# Visualizing Code History for Rapid Exploration and Prototyping

William Choi, Abel Allison, Marcia Lee  
Stanford University HCI  
CS Department  
Stanford, CA 94305  
{wchoi, aalison, malee}@cs.stanford.edu

## ABSTRACT

In order to explore alternatives in code, programmers must be able to create many prototypes and to compare their code and resulting output effectively. Current systems and practices do not afford a lightweight and fluid comparison of versions. We present Rehearse, a system for Processing that visualizes code history to facilitate rapid exploration in code. Rehearse saves both the code and output for each version, and we present two complementary mechanisms to navigate between versions. In both mechanisms, the outputs from all versions are laid out side by side to facilitate recognition over recall of the program’s evolution, and the programmer can easily examine all executions from start to end with one mouse gesture. Although Rehearse is specific to Processing and visual programming, the concepts of 1) storing a version’s output in addition to the code, 2) presenting all outputs simultaneously to the user, and 3) affording a fluid navigation between versions to facilitate comparison are generalizable.

## Author Keywords

Programming Prototyping Histories Versions Visualization

## ACM Classification Keywords

H.5.2 Information Interfaces: User Interfaces

## INTRODUCTION

Prototyping in code enables programmers to rapidly explore different ideas, providing valuable opportunities to compare distinct designs and gain inspiration through the exploratory process before committing to a final design [1]. Writing code in this manner significantly differs from traditional code authoring processes, by prioritizing ease and speed over extensibility, modularity and maintainability of code [4].

While programmers often generate many alternatives that they would like to keep throughout their development workflow, current tools are not well-suited for interacting with

these alternatives to their benefit: comparing, gaining inspiration, identifying which alternatives work well and using them to guide the next iteration. There are several ways programmers currently manage alternatives. They may rely on heavyweight source control systems with high setup cost, save as a different project in the file system every time a new alternative is created, or simply resort to commenting in and out alternative code in the editor.

Surprisingly, none of these methods generate a representation of alternatives that allows for easy navigation and comparison across versions. For example, they do not help programmers identify versions that generate a particular output feature. Asking questions about how versions differ in code and output is difficult, and the potential benefits of exploratory prototyping are harder to realize.

Furthermore, all of these solutions require the programmer to make a proactive decision to identify when a version should be committed or saved and what piece of code should be commented out rather than deleted. Therefore, at every step of the exploratory process, the cognitive load on the programmer increases.

We hypothesize that minimizing this cognitive load would lower the risk of making significant changes to one’s code when prototyping and accelerate testing of more divergent ideas in code. In this paper, we describe a system we built that achieves this by automatically maintaining history of all versions and enabling a more fluid navigation of previous versions and their graphical output.

## RELATED WORK

Brandt et al. [7] examined how rapid prototyping in code occurs in practice, analyzing how this type of “opportunistic programming” process operates under unique goals and development styles. Opportunistic programmers tend to iterate rapidly and to treat code as impermanent and disposable. Brandt et al. observe that “participants often wished that they could revert to the code they had, for example, two tests ago, or quickly branch and explore two ideas in parallel” and suggest that “version control designed for a 10-minute scale” may be beneficial. This motivates our research hypothesis that such a history system would prove particularly beneficial for exploratory prototyping tasks where code is constantly in flux and big code changes occur more readily and rapidly.

Hartmann et al.’s Juxtapose system [1] is motivated by the same premise that exploration of several alternatives results in a better final prototype. The system allows for an easy way to tune variables and evaluate how output changes, and to define and manage several code alternatives in separate editor panels. While the system makes comparing alternatives and developing them easier, it still assumes and expects that programmers proactively define when alternatives should be created. The linked editing view with multiple editor windows may also grow too complex to manage as changes accumulate and code evolves in unexpected ways. In practice, alternatives develop more organically as the programmer experiments with different modifications to code, and managing alternatives reactively through the history trace may fit into the prototyping workflow better. Furthermore, Hartmann et al. discuss that experimenting with code using Juxtapose “explore[s] options within one particular solution strategy,” even though “different solution approaches may be based on distinct implementations.” A history-based approach would not have this limitation, and it would enable exploration both between solutions and within a solution.

Several previous works propose solutions for visualizing design histories. Kurlander’s Chimera system suggests a comic-strip metaphor, with each thumbnail image representing and summarizing an operation in the graphical editor [3]. By visualizing the change in output that resulted from each operation, the system makes it easy to quickly navigate around the history and go back to a point that had a certain graphical state. This is directly applicable to how best we should visualize each version in history. Klemmer et al. have also examined how to capture and interact with design histories. [2] While the work focuses on recording a collaborative design session in the physical world, it provides useful insight on how snapshots in history may be presented on a timeline and on interaction issues that arise when visualizing branching histories. Heer et al.’s analysis of graphical history for information visualization systems such as Tableau [5] also provide some guiding motivations for our work, specifically that highly iterative tasks that produce widely ranging output can benefit greatly from having a well-designed interface that shows salient points in history by automatically chunking smaller edits together. While these designs for working with histories provide a useful basis for our work, the design of our history system needs to address the unique challenges we face in dealing with changes in *code*.

## METHODS

Rehearse’s goals are to encourage the creation of many prototypes and to support a fluid comparison of these different prototypes.

In order to achieve the first goal, Rehearse decides when a version is saved, not the programmer. As opposed to traditional source control systems, Rehearse automatically commits a version after each program execution. Since programmers no longer need to worry about the mechanics of creating and saving versions, they can focus more attention on exploring the design space. This system behavior also encourages more divergent exploration because the program-

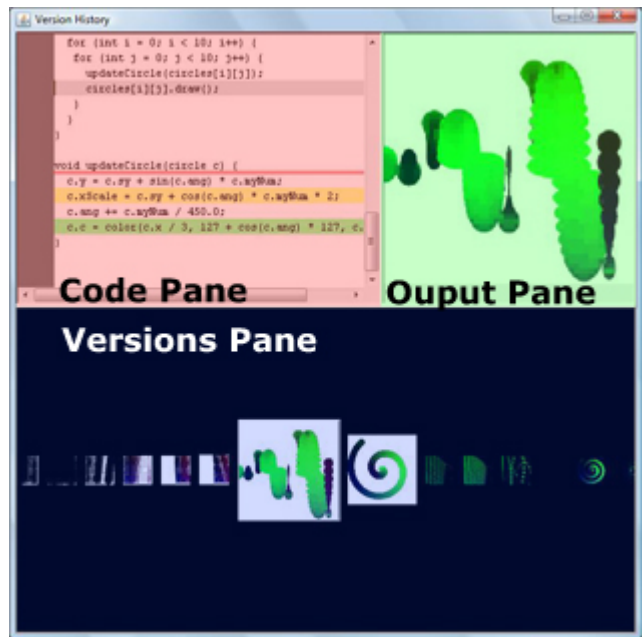


Figure 1. The Rehearse Version History View shows all versions of a project. Both playback of the execution and code diffs are provided for each version.

mer can easily start from a clean slate without fear of accidentally losing prior work.

Simply storing additional data does not necessarily assist the programmer, and the main advantage of Rehearse is the presentation of the version history. Rather than store only the code as in traditional systems, Rehearse stores both the code and the corresponding visual output. Rehearse presents the different outputs side by side to facilitate lightweight comparison, which differs greatly from the laborious process of running different versions and examining the output sequentially. The system also enables rapid viewing of execution playbacks.

## RESULTS

The Rehearse system presents the version history in a window consisting of three panes (see figure 1). The two panes on the top display the code and visual output for a specific version. The bottom pane presents all versions’ outputs in chronological order. Rehearse adopts this particular layout because it allows the user to focus on the specific details of a version in the temporal context of the other versions. The bottom pane affords recognition over recall by displaying all the outputs side by side. The user can easily compare many versions at a glance. The user specifies which version to display in the top panes by mousing over the desired version in the bottom pane. In one gesture, the user can easily view the evolution of the program from version to version, both in terms of code and in terms of visual output. There is no need to repeatedly revert and run code sequentially.

### Code Pane

```

void draw() {
  background(250);
}

void draw() {
  background(250);
  rect(0, 0, 10, 10);
}

```

Figure 2. Rehearse highlights the differences between the currently selected and previously selected versions.

The code pane displays the code that was executed in a particular version. As the user navigates between versions, Rehearse keeps the same region of code centered in the code pane, and this behavior allows the user to examine the region’s evolution over time. Rehearse achieves this behavior by counting the number of additions and deletions that occur above and below the region, and then the scroll position is adjusted accordingly.

The code pane also highlights lines using the standard diff convention of green for additions, red for deletions, and yellow for modifications. Traditional diff viewers display diffs in chronological order, whereas Rehearse highlights the lines based on the direction of the user’s mouse movement. In Figure 2 above, the user added one line in the first version, resulting in the second version. Mousing from the first to the second version produces the second image, where the green line reflects that line’s addition. Mousing from the second version back to the first version gives the first image, which reflects the line’s deletion.

### Output Pane

The output pane displays the visual output, scaled to fit. The visual output can be thought of as a movie recording, and the user can scrub through the movie by mousing over the pane. In other words, the user can view the movie from the first to the last frame by mousing over the left edge to the right edge of the pane. Scrubbing over a particular version in the bottom pane will also cause the same scrubbing behavior in output pane.

### Versions Pane

Rehearse offers two views for the bottom pane: a clip-based view and a fish-eye view. In both views, the user mouses over a particular version in this pane to cause the above code and output panes to update. Again, scrubbing within a version in this pane will cause the upper right output pane to update accordingly as well.

#### Clip-based View

In the clip-based view (see figures 3 & 4), each version is a row of clips, similar to the clips of a movie reel. Across

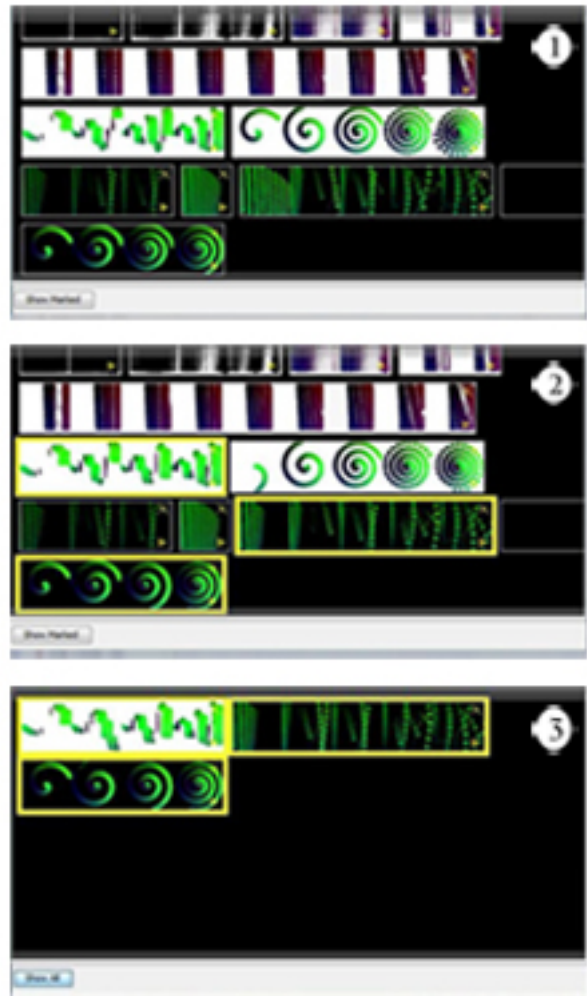


Figure 3. This sequence presents Rehearse’s interactions for reducing clutter in the clip-based view (1). The user can highlight important versions (2) and choose to display only those (3). The user can toggle between showing all versions and showing only highlighted versions.

all versions, each clip corresponds to a second of execution time, thus longer executions are displayed with more clips. The benefit of this approach is that the user can infer an output’s approximate progression without the need to use the mouse to scrub through the entire output. Also, since each clip corresponds to the same amount of time, the scrubbing rate through the different versions will be consistent. The width of each version can also serve as a secondary cue for programmers to help them in navigating around different versions. If the recorded output is insufficient, the user can press the play icon to re-run the version without any changes to the Processing editor that may contain different code.

The disadvantage of the clip-based approach is that very long executions will be cropped in this pane. If the user wanted to scrub through from start to end of many long versions, he would have to repeatedly select the version in the bottom pane and then jump to the upper right output pane to scrub through completely. Although not extremely unwieldy, this approach does require several mouse detours to view all ver-



Figure 4. The user can also minimize versions by clicking the red 'x' at the upper right corner (4), and the versions can be expanded by clicking anywhere in the red rectangle (5).

sions' outputs from start to end. In navigating through the version history, some of the versions may be irrelevant to the programmer's current task. The clip-based view allows the user to minimize recordings and to mark recordings of interest. The user then has the option to display only marked outputs, which reduces clutter and may eliminate the need to scroll.

#### *Fish-eye View*

In the fish-eye view, each version is represented by one frame, and each frame's size is dependent upon its distance to the mouse (see figure 5). The main advantage of the fish-eye view is that the user can scrub through every version in each version's entirety in one smooth gesture, whereas the clip-based view did not afford this task. The user can perform this task easily because scrubbing from edge to edge of one frame plays through the entire version, and the need to scroll has been eliminated. Because every version is laid out side by side, seeing the changes made from one version to the next is simple. Conceptually, the versions can be thought of as lying on a conveyor belt, and the conveyor belt moves left or right appropriately when navigating between versions. In other words, a cursor on the far left of the fish-eye view allows the user to examine the very first version, and a cursor on the far right allows the user to examine the most recent version.

The disadvantage of the fish-eye view is that the user must scrub through versions in order to discern the progression of the execution, as opposed to the clip-based view that gives a rough approximation with the clips at different points in time. Also, the scrubbing rate is variable between versions

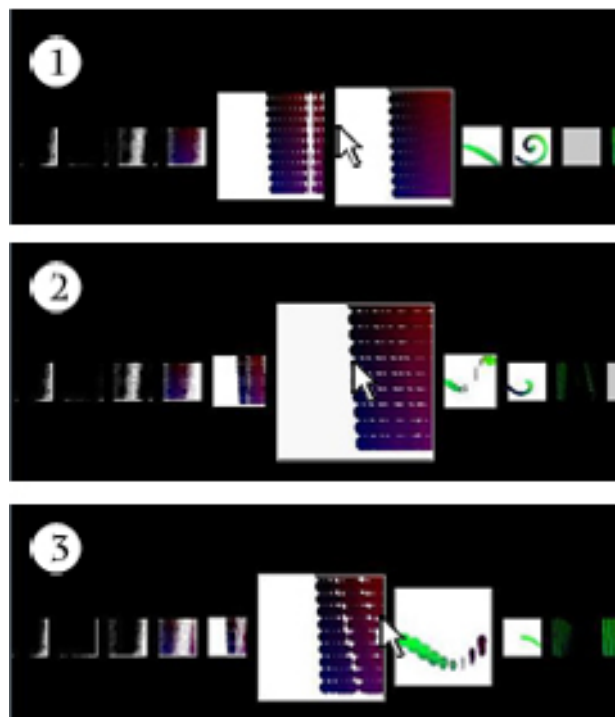


Figure 5. Fish-eye View. Versions closer to the mouse cursor appear bigger, which can be seen in this sequence as the mouse cursor moves left to right.

as well within a specific version. This behavior results because executions of different length are all compressed into one frame and because frames change size based on the distance to the mouse cursor. Although this is not a great disadvantage, it may result in an unintuitive experience if the programmer has prototyped a program that differs in the rate of an object changing over time. An example of such a program could be a simulation of objects moving around and accelerating at different rates.

The fish-eye view has an additional feature that highlights frames that differ from the version in the Processing editor in the lines that the user has selected (see figure 6). For example, when the programmer puts the cursor on a specific line in the editor, all versions that either inserted that line or

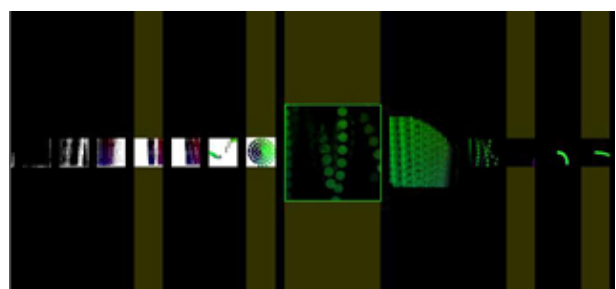


Figure 6. The vertical bars indicate the versions which differ in the lines that the user has currently selected in the Processing editor.

made modifications to that line are highlighted. If the programmer selects multiple lines, versions that made changes to all those lines are highlighted. This allows the programmer to index into the history using sections of code, in addition to output.

The clip-based and fish-eye views complement each others' strengths and weaknesses, and thus Rehearse presents both options to users to choose between.

## IMPLEMENTATION

We chose to work on top of Processing [8], an open-source programming language, editor, and environment that supports creating visual and interactive sketches. Because the environment centers around creating a visual output, it is well-suited for our examination of how the history of code and output can be represented and visualized. We discuss how our work might generalize beyond programs that generate visual output in the Future Work section.

We extended Processing so that each output that it produces when run is saved along with the code that produced that output. All of these are saved locally as text and movie files in the "history" subdirectory of the project folder. Thus, the history is persistent across separate runs and portable along with the project code files.

## DISCUSSION

The Rehearse system for code history visualization provides a quick and easy way to search for and locate a version of a Processing sketch. Testing with users and careful reflection yielded a number of observations and directions for improvement. Our motivating goal is to support exploratory programming, where the programmer may not have a well-defined path of development in mind.

Currently, our system aids in deciding which version to revert to; however, we chose not to implement the ability to branch from an old version in our system. The visualizations for a code's history are designed for single linear histories. A variation of our current prototypes must be designed to accommodate code histories with a tree-structure form. Additionally, better support needs to be built into the editor to support reverts to, branches from, and integrations with other versions.

For the purposes of this paper, we restricted our domain to Processing sketches with visual output. Processing sketches produce a video or image output, which makes it easy to adopt typical interactions appropriate for visual media. Scrubbing through an execution recording is a natural way to parse an execution quickly and efficiently. If we were to consider building a system for all software, our interaction techniques break down. Many programs do not produce a visual artifact. The majority of cues for version retrieval are based upon the interaction with the visual medium. What cues can be derived from just code? Proposed solutions to this problem are detailed later.

The important aspects of a diff system are deciding which

two versions will be diffed together, as well as how the differences between versions are visualized. Currently, our system only shows the focused version's code. Changes are shown with respect to the previously examined version. There are benefits to both the single view and the side-by-side dual view. A single window is useful as you scrub through your history, as it shows the code evolving over time. Similar to a movie, the previous version is replaced by the new one in-place, and there is no need to redirect one's eyes to see the changes. A dual window setup is useful to compare two distinct versions. The side-by-side view enables more complex analysis across versions, giving the user a macro view of changes from version to version.

Regarding usability, the Rehearse system currently lacks sufficient annotations to clarify which two versions are being diff'ed at any given time. While there are benefits to simply comparing the previously examined version to the current one, the user must be able to compare arbitrary versions of their code. Consequently, it must be made clear what two versions are being compared at any given time.

The diff view also lacks good macro-views for the entire code source. The pane to view source code is currently too small. As the user scrubs through versions many changes may be made to the code that are not visible in the code pane. Some compact view of changes to the entire source would notify a user when changes exist that are not currently visible. A simpler "above/below" annotation could also help show when changes (and how many) have been made above or below the currently scroll position.

## FUTURE WORK

Several problems need to be addressed in the Rehearse version history system.

### Smarter Aids to Version Browsing & Search

While it is useful for the application to automatically save versions, our heuristic for when to save is naive and generates a large amount of version "clutter". To further aid search of your history, the system could intelligently cluster versions into reasonable divisions. It would be ideal for the system to identify when you have completed a conceptual "chunk" of code, and to snap a version off at that point.

Another problem we identified is that the change between versions may not be very apparent in the video. Most of the examples we made have clear, noticeable changes in the video, but this may not always be the case. A user might be tweaking subtle effects on an image, or changing something shown in a very small portion of the screen. A hard problem to tackle is modifying the video output to emphasize and point to these changes between videos. A simple video diff could be effective until randomness (or user interaction) is introduced into an execution. Can changes in code be linked to a change in the video? Or do we have to rely upon user annotation to point out salient differences?

### Branching History

Furthermore, the system does not fully aid the user in branching from old versions of code. The history view aids in exploring and finding a particular version, but further action is not well supported. To be a useful tool for exploratory programming, it should aid in both moving between versions as well as resuming work from an old version. This has two requirements: allowing users to branch from an old version, and visualizing a branching structure in the code history. Currently, our history views are linear and have no marks to annotate branching structure. Klemmer [2] proposes a linear history view for web pages that incorporates semantics about branching structure. Tree-shaped views make the search problem more difficult because a 1-dimensional linear search space becomes a 2-dimensional plane. Keeping the history in a linear view simplifies the search.

### Generalizability to Non-visual Programs

As mentioned before, the interactions are designed for a specific domain: programming visual software. In order to be a useful tool, it must be redesigned to work for software whose output is not necessarily visual. Can a similar history view be implemented that uses just code as its search material? We propose a similar linear view of code versions that uses code as a version visualization instead of output. A project's code files are concatenated together to form a rectangle. Changes in the code (insertions, changes, deletions) are shown as colored lines inside the rectangle corresponding to where the changes occur. Between versions, trapezoids show movement and changes in size of various code regions in order to help show progression. Even if the text is too small to read, showing zoomed-out code in the rectangles could aid recognition. We hypothesize that code shape serves as a useful cue for version recognition.

Further information can be added to visualizations of semantic blocks of a project. For example, a function both references other functions and is referenced by other functions. The history of a function's context can be visualized by showing the callers and callees of a function. The space for visualizing code history for quick retrieval is not well-explored. These proposed solutions are only a start to finding the optimal way to search through one's code history.

### CONCLUSIONS

We have examined an important disconnect between fundamental prototyping ideals and current practices in prototyping in code. Although rapid iteration and exploration of multiple alternatives that span the design space are cornerstones of the prototyping process, programmers face difficulties managing versions as they grow in number and complexity. Providing a design history of ideas tested in code allows programmers to take control of their constantly evolving code. Making the act of saving a version reactive rather than proactive lets the programmer focus on exploration of ideas rather than worrying about the risk of making big modifications.

We present a system that provides lightweight navigation through a programmer's entire version history. We have presented the clip-based and fish-eye views, and we have de-

tailed their complementary advantages and disadvantages. Combining these contextual views with the detail of the code and output panes, Rehearse presents more information in a more accessible way than do traditional methods of prototyping in code. Although several issues remain to be resolved to improve Rehearse, we believe that the system addresses the problem of prototyping in code in a new and compelling way.

### REFERENCES

1. Hartmann, B., Yu, L., Allison, A., Yang, Y., and Klemmer, S. R. Design as exploration: creating interface alternatives through parallel authoring and runtime tuning. *Proceedings of UIST 2008*. ACM, New York, NY, 91-100.
2. Klemmer, S. R., M. Thomsen, E. Phelps-Goodman, R. Lee, and J. A. Landay. Where Do Web Sites Come From? Capturing and Interacting with Design History. *Proceedings of CHI 2002*. pp. 18, 2002.
3. Kurlander, D. and S. Feiner, A Visual Language for Browsing, Undoing, and Redoing Graphical Interface Commands, in *Visual Languages and Visual Programming*, S.K. Chang, Editor. Plenum Press: New York, NY. pp. 257-75, 1990.
4. Brandt, J., Guo, P. J., Lewenstein, J., and Klemmer, S. R. 2008. Opportunistic programming: how rapid ideation and prototyping occur in practice. *Proceedings WEUSE '08*. ACM, New York, NY, 1-5.
5. Heer, J., Mackinlay, J.D., Stolte, C., Agrawala, M. *IEEE Information Visualization (InfoVis) 2008*. [http://vis.berkeley.edu/papers/graphical\\_histories/2008-GraphicalHistories-InfoVis.pdf](http://vis.berkeley.edu/papers/graphical_histories/2008-GraphicalHistories-InfoVis.pdf)
6. Hartmann, B., Klemmer, S.R., Bernstein, M., Abdulla, L., Burr, B., Robinson-Mosher, A., Gee, J. Reflective physical prototyping through integrated design, test, and analysis. *Proceedings of UIST 2006, October 2006*.
7. Brandt, J., Guo, P. J., Lewenstein, J., Dontcheva, M., and Klemmer, S. R. 2009. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. *Proceedings of CHI '09*. ACM, New York, NY, 1589-1598.
8. Fry, B. *Processing Software*. MIT. <http://processing.org>.