Iterating Between Tools to Create and Edit Visualizations

Alex Bigelow, Steven Drucker, Danyel Fisher, and Miriah Meyer

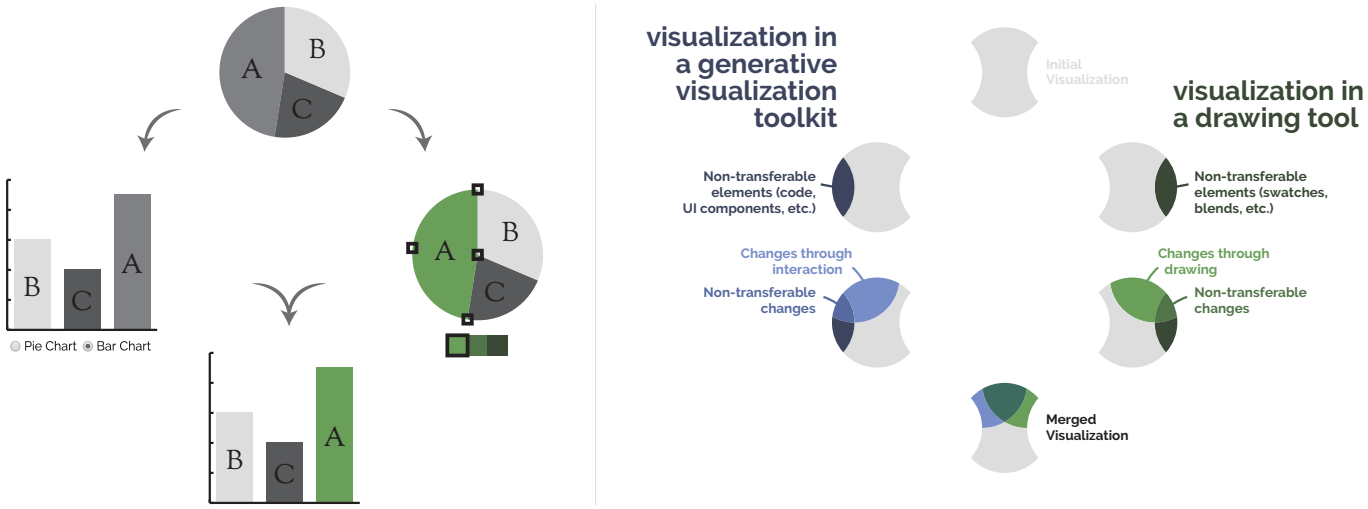

Fig. 1: An example combining edits to a visualization from two different tools. *(Left)* A visualization — initially a pie chart — is edited with a generative visualization toolkit on the left, such as D3, as well as with a drawing program on the right, such as Illustrator. Changes from both tools are merged into the final visualization. *(Right)* For such a process to be possible, we propose a *bridge model* that describes out edits from two tools can be combined. This model identifies edits that can be shared, as well as those that cannot, and merges them together with a careful consideration of potential conflicts. The resulting visualization can then reintegrated into the two tools, supporting further iterations.

Abstract—A common workflow for visualization designers begins with a generative tool, like D3 or Processing, to create the initial visualization; and proceeds to a drawing tool, like Adobe Illustrator or Inkscape, for editing and cleaning. Unfortunately, this is typically a one-way process: once a visualization is exported from the generative tool into a drawing tool, it is difficult to make further, data-driven changes. In this paper, we propose a bridge model to allow designers to bring their work back from the drawing tool to re-edit in the generative tool. Our key insight is to recast this iteration challenge as a merge problem - similar to when two people are editing a document and changes between them need to reconciled. We also present a specific instantiation of this model, a tool called Hanpuku, which bridges between D3 scripts and Illustrator. We show several examples of visualizations that are iteratively created using Hanpuku in order to illustrate the flexibility of the approach. We further describe several hypothetical tools that bridge between other visualization tools to emphasize the generality of the model.

Index Terms—Visualization, iteration, illustration

1 Introduction

Visualization designers use a variety of tools in the practice of their craft, particularly when creating infographics and telling stories with data. Designers will often transition between tools, first making use of tools like Tableau, ggplot, and D3 to automatically encode data into a chart. Then, they make stylistic changes and add embellishments in a tool like Illustrator [3]. This workflow, however, limits iteration: once a visualization is exported from the D3 script into Illustrator, the graphical elements are merely shapes, and are no longer linked to data; a designer cannot easily go back to modify the D3 script without losing their Illustrator work. The result of this disconnect between tools is that designers explore fewer design variations and have trouble handling changes to the underlying data [3].

We see an example of how this manifests in a visualization created by designer Craig Robinson and shown in Figure 2, which describes the actors employed by HBO and the TV shows in which they are cast[19]. The actors are sorted alphabetically based on their first name, with the exception of the last actor — the asterisk next to this actor's name is an apology for not placing him in the correct, sorted order. The likely scenario that resulted in this problem was that the designer performed significant manual work, placing nodes and connecting them with edges, in a drawing tool like Illustrator before realizing that one actor had been left out. He would need to do much more manual work to insert this name in the right place, moving many nodes and edges manually. In a generative tool like D3, however, adding the new name would be trivial — but it would come at the expense of losing the stylistic work done to the image in Illustrator such as color selection, font choices, and text layout.

This laborious *rework* [8] is slow and frustrating: rather then exploring a creative space, the designer must manually re-implement a design, or component of the design, that they have already created. Creative visualization designers encounter this form of repetition very frequently [3]. This rework can be alleviated by allowing designers to

- *Alex Bigelow and Miriah Meyer are with the University of Utah. E-mail: (abigelow,miriah)@cs.utah.edu.*
- *Steven Drucker and Danyel Fisher are with Microsoft Research. E-mail: (sdrucker,danyelf)@microsoft.com.*

Manuscript received xx xxx. 201x; accepted xx xxx. 201x. Date of Publication xx xxx. 201x; date of current version xx xxx. 201x.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org.
Digital Object Identifier: xx.xxxx/TVCG.201x.xxxxxxx/

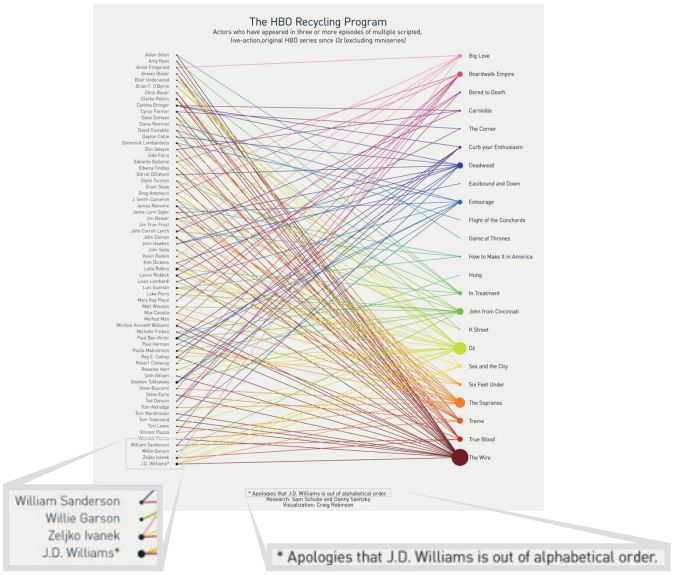

Fig. 2: The HBO Recycling Program infographic [19], showing actors alphabetized by first name, linked to shows that they performed in. The final actor, J. D. Williams, is listed out of order.

make changes in the tools that best support the types of changes they wish to make.

Generative tools, such as D3, Processing, Tableau, Microsoft Excel, and VTK are those in which a visualization is created, based on a dataset, through a series of computational steps; they support placing data marks, sorting data items, and exploring layout strategies. Grammel et al. [11], articulate a taxonomy of generative tools, which all create visual objects based on data and an underlying generative model.

In *drawing tools*, like Photoshop, Illustrator, and Inkscape, a user creates and manipulates graphics; they support tasks like modifying color palettes, select fonts, and lay out text and annotations. Figure 3 shows a cross-tool design iteration of the HBO infographic where each part of the design process is conducted in a tool most appropriate for the task, specifically using D3 for generative, computational tasks, and Illustrator for manual drawing ones. To support workflows like this we propose building *bridges* between tools. A bridge is software that allows visualizations made in one tool to migrate and coexist in another; and to bring edits and updates back and forth. In this paper we focus on articulating the design space, requirements, and considerations for building bridges between generative and drawing tools.

This paper contributes, first, a model that supports bridging between generative and drawing tools for visualization design. The model supports design iteration and reduces rework by recasting the iteration problem as one of merging. This *bridge model* is general, and works across many tools; it describes a large space of design possibilities, trade-offs, and considerations. The second contribution is a single instance of the bridge model, an open-source tool called Hanpuku (Japanese: *iterate*). Hanpuku allows a designer to programmatically generate visualizations by writing or reusing D3 code, to then edit those visualizations in Illustrator, and to bring the edited visualization back into D3 for further generative modifications. This editing cycle can be repeated over and over, supporting iteration on the visual representation, the style, and the data itself. We illustrate the richness that Hanpuku affords to the design process in several examples. Hanpuku, along with high-level descriptions of several other possible bridges, validate the efficacy of the underlying bridge model for supporting visualization design iteration across a range of existing tools.

2 Related Work

Increasingly, software systems are beginning to support designers in the endeavor of bridging generative and drawing tools using one of two dominant strategies. The first strategy is to create all-in-one tools that support both data and drawing operations in the same interface. There is a second, growing strategy which involves creating one-way bridges between generative and drawing tools.

The first approach is to create a single tool that accommodates a broad spectrum of needs, combining the efficiency and accuracy of generative generation of charts with the richness and flexibility of drawing tools. To some extent, contemporary generative tools attempt to provide some of these features: Tableau and Microsoft Excel allow users to make a variety of design choices within the tool such as font choice and mark color. The expressiveness of these tools, however, is limited: common functions in drawing tools, such as controlling paper size, precise alignment, and adding textual annotations, are only weakly supported in these generative tools. Conversely, while Adobe Illustrator allows users to create basic chart types within the drawing, editing the chart in Illustrator breaks the connection to the underlying data, and so updating the visualization still requires rework.

Building a tool to unify these approaches has a long history, from early efforts like RBE [14] and SageBrush [20] to more recent approaches like iVisDesigner [18]. Some systems, such as Lyra [22], go further and support interactive visualization creation in the context of a larger, more modular software stack [4, 23, 32]. Such tools, however, must address the underlying question of how to merge the stylistic changes and embellishments into the results of generative execution, as well as how to recreate the extensive features and techniques already present in industry-hardened software.

The second approach represents a different kind of modularity in that it bridges existing, disparate tools, easing the transition between generative tools and drawing programs. Tools such as D3 Deconstructor [12] and SVG Crowbar [6] help smooth the transition from generativity-generated visualizations to drawing environments. Some generative visualization systems, such as Raw [7], are designed entirely with this end in mind. These programs, however, can only operate one way. There is no way to iterate on the generative aspects of a visualization after drawing aspects have been considered without engaging in rework. The bridge model that we present enables designers to operate in either direction.

The bridge model depends on the idea of bringing together distinct sets of modifications from multiple sources, making it similar to both revision control systems [28] and collaborative editing systems. Each of these has mechanisms for merging changes disparate authors. Revision control concepts have been applied to design contexts, such as CAD tools [10] and collaborative editing of 3D models [9, 26]. One important difference is that the bridge model focuses on iteration between different classes of tools, rather than different people using the same tool.

Designers have created visualizations with a variety of tools, from paper [29, 30] to tangible blocks [13]. Our bridge tool approaches visualizations that are natively digital: created in software. In contrast, digital sketches that are linked to data [15] do fit comfortably within the bridge model.

3 The Challenge of Iteration

Design iteration refers to the creative exploration of the many aspects of a design. Designers are more successful when they can explore the parts of a design in any order [27]; conversely, serial design workflows present fewer opportunities to weed out poor designs and discover creative opportunities [16]. The ability to go back and make changes at any level of the design is the meaning of *iteration* that we use throughout this work.

Visualization designers use a variety of tools because the creation of compelling, engaging, and accurate infographics, based on increasingly large and complex data, implies two sets of software requirements. Many of the requirements are addressed by traditional visualization tools: handling large amounts of data, updating a visualization based on changes to the data, changing the visual representation, and algorithmic generation of certain graphical representations such as the placement of marks and creation of complex layouts. Tools like Excel, Tableau, and ggplot, as well as programming languages and environ-

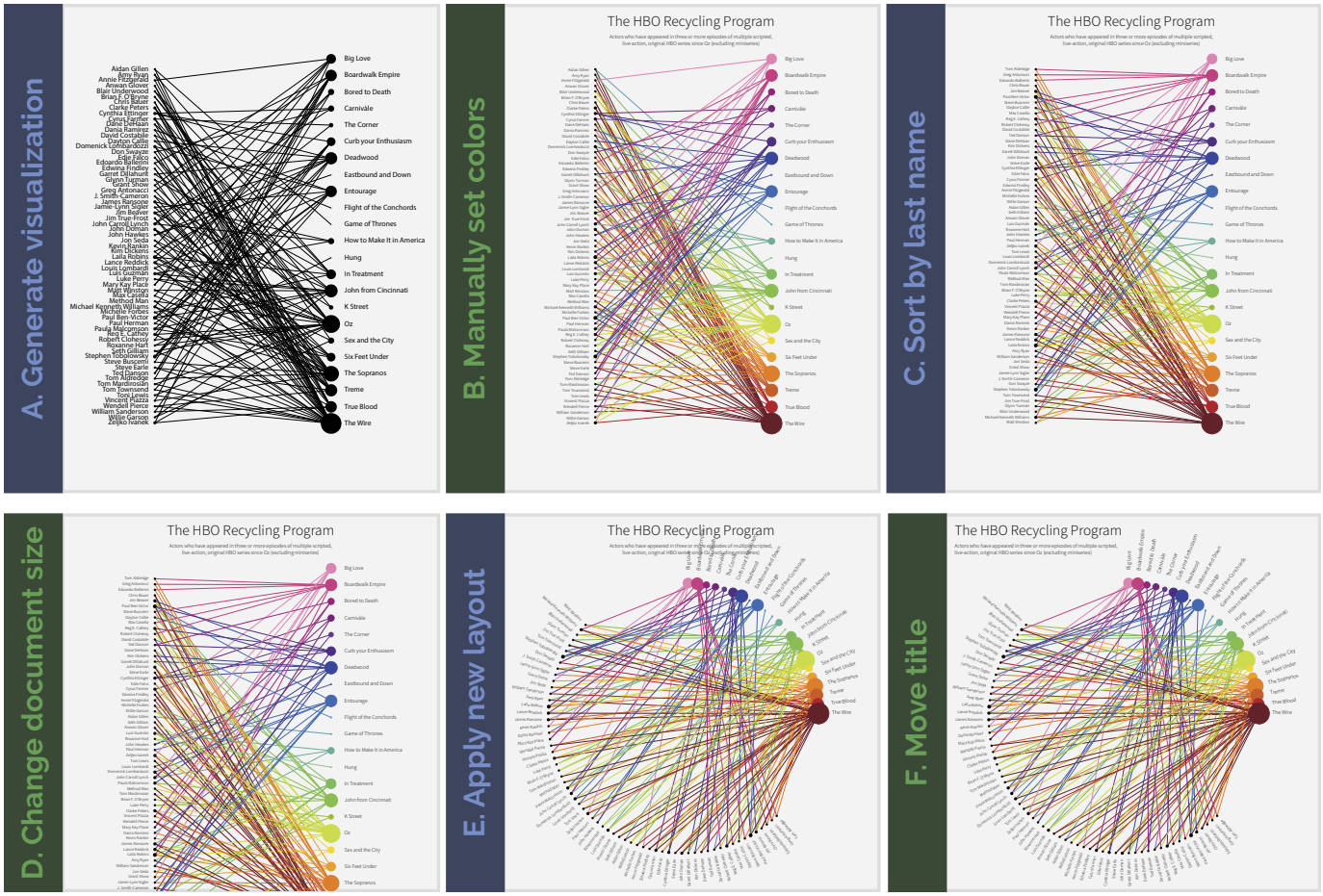

Fig. 3: Extended recreation of Figure 2. Blue steps are done in a generative tool; green steps in a drawing tool. (A) An initial graphic is produced using a D3 script. (B) Illustrator's tools are used to adjust typefaces and font sizes, add a title, a subtitle, and color to each TV show, with its corresponding edges. (C) A change is added to the D3 script to reorder the actors by last name—then the script is re-run on the existing document. (D) The aspect ratio of the page is changed with Illustrator's tools. (E) The layout algorithm in the D3 script is modified to accommodate the new aspect ratio. (F) The title is adjusted using Illustrator's tools.

ments like D3, Processing, and VTK are proficient at these sorts of generative tasks that a visualization creator would want to define abstractly and repeat automatically.

Conversely, designers also make use of flexible, manual controls for quick, visual iteration on stylistic elements and visual embellishments, such as color palettes, fonts, annotations, overall sizing, and placement of text. Exploring these detailed aesthetic choices can be tedious with generative controls — a designer might need to restart execution to test a different shade of blue — and instead benefit from the immediate visual feedback available in tools like Illustrator and Inkscape. Furthermore, drawing tools focus on supporting a very rich set of manual controls as well as intuitive ways of showing design options and saving iteration provenance. The flexibility of drawing tools is important for allowing a designer to style and individualize a visualization [3].

Designers often work their way from one tool to another. An example is a visualization that designer Bang Wong created for a scientific publication ([21], Figure 2). He is quoted as saying about his design process: *"I created a plot in Tableau and then exported it to Illustrator to put the labels on. I had to add the specific tick-marks by hand. [For the labels], Tableau does labels, but it isn't very smart... I had to manually pull them apart when they overlapped. I did the donut chart in Excel because Tableau doesn't have them. I then changed the colors [in Illustrator] to greyscale to get rid of the Microsoft colors."*[3] This workflow, which begins in generative tools to create a first cut on a visualization and then moves to a drawing tool for refinements, is common.

Unfortunately, this process can severely limit flexibility. Once a visualization has been brought over from a generative tool to a drawing one, it loses its connection to data: exporting is a one-way process. If a designer wants to modify the visualization later — perhaps to accommodate new data, or to correct algorithmic errors — he or she needs to make a difficult decision. The changes made in drawing tools cannot be transferred backed into the generative tool, and so changing a visualization entails redoing any graphical revisions.

The challenge with iteration is that the original image generated in generative visualization software embodies a mapping from underlying data into an image. The core algorithm behind many visualization programs is a simple loop: for each data item, compute the location and rendering for the corresponding mark and plot the mark. Once the marks are plotted as shapes, and the resulting visualization is produced and exported, however, the underlying mapping to the data is lost. When the designer manipulates the marks in a drawing tool there is no longer a connection between the data and the visualization.

4 The Bridge Model

Our solution to this problem is the *bridge model*, so called because it maintains a bridge between the generative tool and the drawing tool. The bridge model is general across multiple platforms, generative tools, and drawing tools. In Section 5 we present a specific implementation of the bridge model, a tool called Hanpuku, which bridges D3 code and Adobe Illustrator. The design decisions we made

in Hanpuku help illustrate the virtues and challenges of this model. We discuss high-level design decisions for several other possible bridges in Section 6.

The key insight in the model is to recast the iteration problem into a merge problem. We formulate the problem in terms of isolating the changes made in each tool—splitting the set of serial edits into parallel sets of edits—and then merging these changes. Under this approach, iteration becomes a process of repeated merges. This merging can be viewed as a kind of visual *join*, in the sense of the word used both by the database community and the D3 tool: each shape is associated with an identity, known as a *key*; the merger then decides how to integrate the changes to the shapes on each key, made by each side. The bridge model articulates the considerations necessary for merging changes from two different tools.

The bridge model starts from a shared representation of a visualization. Specifically, the visualization must have a representation both in the generative tool and the drawing tool. The model also requires a way to uniquely identify each graphical element within the visualization and to maintain those identities across the tools.

Each tool may imbue the visualization with *non-transferable elements*: parts of the visualization that do not translate to the other tool. For example, in a generative tool the non-transferable elements might include the source code, interactive slider bars and filter buttons, or interactive components like roll-over highlighting or user-influenced force-directed layout. Conversely, in a drawing tool there might be color palettes, layering effects, or text layout features such as kerning and drop caps that do not easily translate to standard generative tool features.

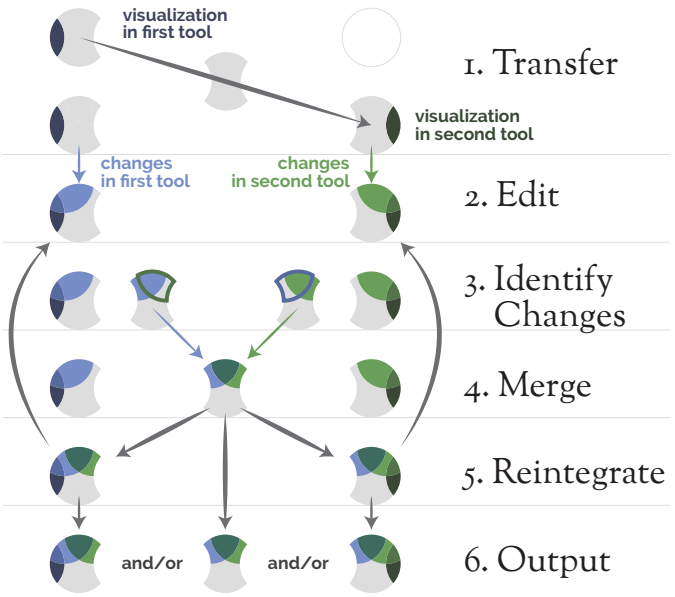

Fig. 4: An abstract representation of each stage of the bridge model, showing how changes to the visualization in one tool can propagate to the other. Note that, while it is impossible for tool-specific, non-transferable elements to exist as part of the visualization at the same time, they can still continue to contribute to the visualization's design throughout the workflow.

The full model is illustrated in Figure 4. In this figure the left side represents work done in a generative tool and the right side represents work done in a drawing tool. (1) A visualization is generated in a generative tool consisting of shared elements in grey and non-transferable elements in dark blue. The shared portion of the visualization is *transferred* to the drawing tool. (2) *Edits* to the visualization are made in either tool, shown as an intersecting light blue region for changes made in the generative tool and an intersecting light green region in the drawing tool. These changes may include non-transferable elements. (3) Changes to the visualization are *identified* by the bridge, along with an identification of changes that may be in conflict. (4) The changes are *merged* and the conflicts resolved, shown as a teal region. (5) The merged changes are *reintegrated* into the representations of the visualization in both the generative and drawing tools. At this point the process repeats as the designer iterates on the visualization. (6) The final visualization is *output*, either directly by one of the tools or as some other representation.

We believe there is no single right way to build an effective bridge; instead, bridge creators must carefully consider the design space and choose a set of strategies that balance flexibility, richness, and implementation difficulty. This design space is large. The goal of the remainder of this section is to lay out the strategies for implementing each component in order to allow bridge creators to carefully consider the multitude of options in a structured way. In Section 5 we will discuss our specific decisions with respect to our D3-Illustrator bridge, Hanpuku. We note that this design space is not only relevant for creating bridges between tools, but also for building internal bridges within a single tool that supports both generative and manual changes to a visualization, such as IVisDesigner [18].

Within this paper, such as in the sections on Hanpuku and Figures 5-6, we show two different paths being carried out simultaneously. In this paper changes are seen not as simultaneous, but happening in parallel on two different tools. Even in cases where the designer simply merges the newer changes in one tool into the other tool, the edits must be consolidated together — and in many scenarios, the changes made in one tool are not reflected in the other. For example, in Hanpuku changes that the designer makes to the visualization in Illustrator do not alter the D3 script, and so the changed D3 script's output must be merged with the Illustrator changes.

4.1 Identifying Changes

In the bridge model, an image consists of three types of content: data-bound graphical elements, non-bound graphical elements, and non-transferable elements. Data-bound graphical elements are the visual marks that have a clear association with specific data items. The non-bound graphical elements are those elements of the image that do not correspond directly to individual data points, such as legends in a generative tool, or titles and background images in a drawing tool. These elements need to be transferred between the tools as well, so the bridge must also provide them with unique identities. Finally, there are non-transferable elements which have no meaning on the other side, such as the source code in a generative tool, or the paper size in a drawing tool.

An implementation of the bridge must have both some mechanism to share the original visualization between tools, and to communicate what changes each tool made to the visualization. Communicating the changes requires that the bridge maintain a notion of the identity of graphic elements within the visualization. All shared elements, including data-bound and non-bound elements, must be identifiable on either side of the bridge. Additionally, data-bound elements must be linked back to the original data points to which they correspond. Non-transferable elements, however, are not shared, and therefore do not need to be identified.

A bridge must maintain a link between the data items, and the representation in each tool — we call these links *data bindings*. These data bindings can exist in application-specific forms. In the simplest case many data formats such as the HTML DOM (Document Object Model) and PDF allow arbitrary user tags to be attached to graphical elements. Those tags can be used to hold data binding identities. Tagging data items as they are transformed into shapes for a visualization is a common approach; the technique is used in VisTrails [25, 5] to track provenance, and in Weave [2] and Improvise [31] to enable brushing and linking. In D3 Deconstructor [12], data tags are used to reconstruct the original data mappings from the visualization. We take advantage of that same approach.

In Figure 5 we illustrate the role of the data bindings. A pie chart is created in a generative tool; each element includes an ID tag. This visualization is then modified twice: in a generative tool, the designer changes from drawing a pie chart to a bar chart; in a drawing tool, the

designer highlights wedge "A" in a unique color. The merger is able to recognize the changes made on each side: because it can detect that the new bar #2 corresponds to the wedge #2, it can apply the fill of the wedge to the bar. Maintaining data bindings in both tools allows a merger to appropriately track the changes made to the same elements.

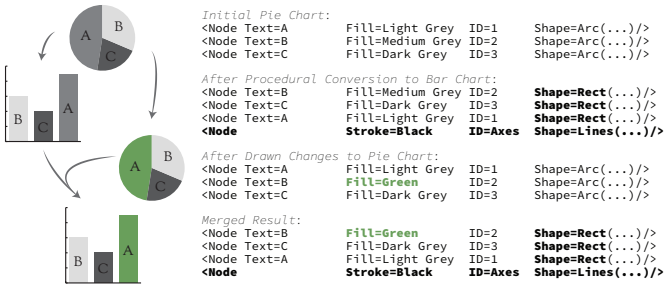

Fig. 5: Merging changes together. On the left, the visual effect: two different changes are merged. On the right, the element IDs render the merge unambiguous.

4.2 Merging Changes

The goal of the merge process is to bring the edits made in each tool into a single representation of the visualization. In revision control merge algorithms, the merge attempts to choose an output that incorporates non-conflicting changes to source code. In the bridge model, a merge similarly starts with a representation of the edits made in each tool, and attempts to reconcile them into one version, possibly with human assistance.

When two modifications of the visualization have no conflicts, it is easy to merge them together. For example, if there were no graphical elements changed by both tools, merging is straightforward process of taking the newest version from each. The process becomes more challenging when there are *conflicts*. Conflicts are changes that cannot be implemented at the same time; conflict can occur when both tools make changes to the same graphical element that effects the same, or similar, encoding channels, such as both tools modifying an element's color.

The easiest way to deal with conflicts is to avoid them. One way to do this is to track and identify changes in as fine-grained a way as possible. For example, consider the refinements made to a pie chart in Figure 5. In this example each tool carries out a set of changes: in the generative tool we change the shape of the graphical elements from pie-slices to bars and move their positions; in the drawing tool we recolor one graphical element. Because we track changes at the fine-grained level of color, shape, and position the changes are straightforward to merge as they operate on different parts of the encoding. If we looked at the graphical elements in a more coarse way — treating elements as atomic, for example — then this situation would turn out to be a conflict.

The refinements shown in Figure 6 are an example of conflicting changes. Starting with a bar chart, the two different tools each refine the visualization: in the drawing tool, a designer highlights one mark of interest; in the generative tool, he or she adds a texture to all of the marks. In this situation, it is not obvious what the bridge model should do; the result is ill-defined. Valid mergers might choose from a variety of strategies: one refinement might trump another; the system might compute a combination of the refinements; or the system might ask for human input to resolve the difficulty.

Another strategy for minimizing conflict is to provide as much context as possible for each change. In the fine-grained example above, the bridge was able to look inside the graphical element to merge on its attributes. One way to help ensure that these attributes are available is to place related graphical elements into groups, and label the group with the data key. This ensures that groups will move together. This is useful when the system renders the label, the fill area, and the outline for a bar as three separate shapes. This allows great flexibility: the

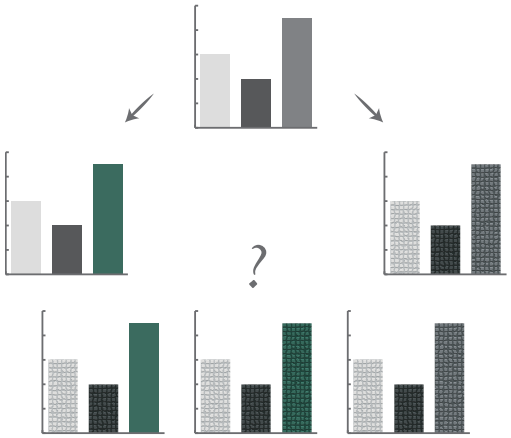

Fig. 6: Three possible ways to handle a conflict.

user can use the illustration tool to remove the label, or replace the bar with an image; the existence of the group helps point to what changes were made, and the shapes will be moved relative to the group.

4.3 Capturing Intent

So far we have talked about well-defined changes that a designer makes to a visualization. In many types of refinements, however, the designer has a broader intent for the way the visualization as a whole is being changed. These types of ill-defined changes are difficult, if not impossible, to accurately capture. In this section we discuss two types of ill-defined, but common, changes — annotations and manually defined algorithmic rules — and explain why they pose a challenge for the bridge model.

To illustrate the challenges around intent, we can begin by looking at one problematic case, annotation. One convenience of drawing tools is that it is easy to annotate objects to call out specific aspects of the data. For example, a designer might place a textual annotation over the teal bar in Figure 7. In merging this change, the bridge implementation must decide what the location of this new annotation means. Is its position meant to be absolute on the page, or relative to the bar? Is it meant to be three pixels higher than the teal bar, or the highest bar? If the highest bar changes, or the order of the bars change, or the designer changes back to a pie chart, where should the annotation go?

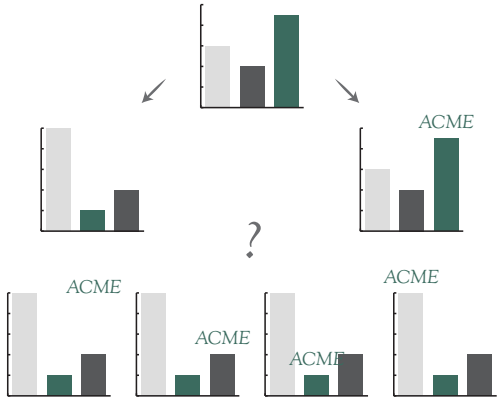

Fig. 7: It can be difficult to automatically infer a designer's intent in placing a label.

Some of the mechanisms we outlined above can help here: putting the annotation into a group signals that if the bar is moved, the annotation moves with it; keeping the annotation disconnected from data signals that the annotation should sit at an absolute position.

Alternately, adding a constraint system into drawing tools could help disambiguate user intent here. The constraints would help show

which items are meant to correspond to each other. This need not be a heavyweight process. In Microsoft's PowerPoint, for example, when a user places an object, the system shows guidelines that help align it with existing objects on the slide; it is not hard to imagine using this mechanism to store constraints.

Another form of intent appears when the designer has a sense of an algorithmic rule, but implements the rule manually in the drawing tool and not in the generative representation. For example, consider a visualization with a color map that runs from red to blue. If the designer uses a drawing tool to change one end of the color map to green, it would be difficult under the bridge model to automatically infer their intent, and recolor the rest of the color map to fit.

We encounter this uncaptured intent in the example shown in Figure 3. The designer manually defined a color map by explicitly setting the colors of each TV show node in an ordered fashion. These colors are then applied to edges that emanate from each show. When an additional actor node is added using a generative tool, shown in Figure 8, the nodes are moved appropriately, but the color of the edges is not encoded in the generative representation. These colors must therefore be manually edited.

Fig. 8: An additional actor is added to the example in Figure 3. Note that the text and lines do not reflect the designer's manual color and typography assignments.

A viewer of this visualization can identify the color algorithm: edge colors should be the same as the node they are attached to. Maintaining that sort of constraint — as well as constraints like color maps — is easy to do generatively, but challenging to articulate within a drawing tool.

4.4 Reintegration and Output

After the merge stage of the bridge model, the process is not yet complete; the reintegration step brings the merged representation back into each tool. The previous stages of the merge process have stripped the common elements in both tools away from any non-transferable elements that may be a part of the visualization in each environment, such as scripts or interactive UI elements like buttons and menus in the generative environment, or color swatch and layer compositing settings in the drawing environment. Reintegration brings the merged result back into the tools at each side, allowing the user to make further edits that build on the merged results without losing their previous work. Reintegration, therefore, enables the iterative loop.

Reintegration need not be a two-way process. While in Hanpuku changes can be propagated from the design back into the generative tool, that is not strictly necessary. A one-way merger would mean that integration steps would only be pushed forward into the drawing tool. The merger would still need to make sure that changes created on both sides be brought together. In an extreme design, it might be possible that *neither* tool sees the merged results; instead, the merged output might be in a form that neither tool can view.

One result of reintegration is output: if only a static image is the intended result, it may be appropriate to only support one-way reintegration into the drawing tool. Or, if an interactive visualization is the intended result, it may be appropriate to only support one-way reintegration into the generative environment. In some cases it may be sufficient to allow the user to make a series of changes in each tool, followed by one large merge at the end, skipping reintegration altogether. This final approach can still support an iterative workflow—changes can still be made in each respective environment, and the merged output can be regenerated.

Though non-transferable elements do not necessarily need to be identified in the bridge model, identifying and considering such elements can still be useful. As we saw in Figure 4, non-transferable elements from two different tools can never exist together in the visualization at any point. A bridge, however, can employ various strategies regarding non-transferable elements to facilitate seamless reintegration.

The simplest strategy for handling non-transferable elements is to simply ignore any aspects of the representation that cannot be shared, leaving them in their source environments until the merge is reintegrated. The drawback to this approach is that any refinements made to these aspects of the representation will not be mergeable. This solution may be appropriate if very little of the visualization is affected, or if we expect the output to be in the tool that contains this non-transferable information.

The other strategy is to build in translations and approximations as proxies for non-transferable elements. These proxies become shared elements that can participate in the merge. For example, the merger could decide that when it sees a gradient shade applied in a drawing tool it will fall back to a solid color instead if the generative tool does not have an equivalent feature. In more sophisticated implementations the merger might include more sophisticated computational approximations: Adobe Illustrator cannot represent a circle directly, so any SVG elements with circles become four bezier curves instead. The reintegration step must then decide how to apply changes that affect the proxy element to the original non-transferable element in its native environment.

5 Hanpuku: An Example of a Bridge

We have built an example of a bridge between a generative tool and a drawing tool in the form of Hanpuku, an Adobe Illustrator extension that merges changes made by a D3 script into an Illustrator document. Here we discuss Hanpuku, highlighting the various strategies from the bridge model.

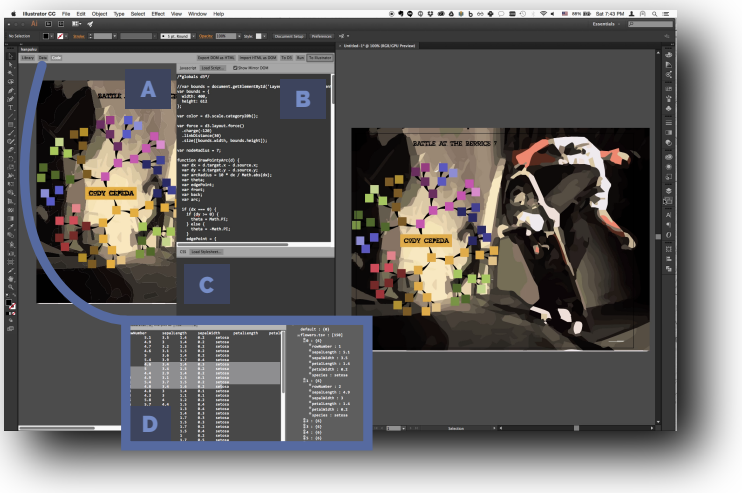

Fig. 9: Hanpuku includes (A) a web view that mirrors the illustrator document, (B) javascript and (C) CSS code editors, and (D) a raw text data editor with an interactive preview of how D3 will parse the raw data.

As shown in Figure 9, Hanpuku provides a programming environment for editing and running D3 code. Fundamentally, Hanpuku incorporates two different kinds of merges, each triggered by a button click. Clicking the "To D3" button merges changes from the Illustrator

document into the programming environment, and clicking the "To Illustrator" button merges changes from the programming environment into the Illustrator document.

5.1 Identify Strategies

To use Hanpuku, a designer creates an empty document in Illustrator. They drop javascript code into Hanpuku's D3 view (Figure 9B), and possibly a CSS declaration (9C).

To match visual elements across tools and identify the corresponding changes, Hanpuku takes advantage of D3's existing patterns: D3 scripts output an HTML DOM hierarchy; every item in the DOM that was created from data also has a data binding associated with it [4]. Illustrator uses an in-memory document hierarchy that is similar to the HTML DOM. The similarities in document structure make establishing identities between graphical elements straightforward.

D3 manages bindings between visual elements and data. D3 uses a concept of a *data join* to manage changes in data. When the main loop is executed, D3 calls a user-specified "Add" function on new data points, and an "Update" function on existing data points. D3 uses a user-defined key function to identify each object to be updated, or created.

When a user runs the D3 code for the first time by pressing "To Illustrator," the D3 code calls these "Add" functions to create a DOM hierarchy; Hanpuku then translates that into an Illustrator in-memory model, and integrates the new elements into the Illustrator document. Like the D3 Deconstructor [12], Hanpuku takes advantage of the data bindings in the DOM; it translates those bindings into metadata in the Illustrator model.

After a designer has modified the visualization in Illustrator, a "To D3" merge translates the Illustrator document into an HTML DOM for D3. The data bindings from the Illustrator side are conserved back through the merge process. With the merged DOM now live in D3, the designer can modify the D3 script. When the D3 code is executed, the system executes a data join against the current dataset and the visualization. If the data has changed, new objects can be created with Add functions; Update methods are called for pre-existing element. Good D3 coding style calls for Update methods not to modify anything that does not need to be edited. As such, any changes made in Illustrator are preserved, unless the coding has changed, and the Update function must overwrite them.

5.2 Merge and Reintegrate Strategies

Merge conflicts occur in Hanpuku when a user's D3 code overwrites the same fields as were edited in Illustrator. At this time, Hanpuku employs the simplest strategy outlined in Section 4.2 — the "To Illustrator" merge always incorporates the differences from the D3 side, and the "To D3" merge always incorporates the differences from the Illustrator side. If the Update function in the D3 code overwrites a field that was edited in Illustrator, it will be lost. In practice, this simple strategy works well. For example, a user can interact with a node-link diagram layout in the "Update" loop, and radical representational changes such as that shown in Figure 5 can be maintained.

Hanpuku is designed to support a workflow that produces an Illustrator document, and thus the tool is particularly careful to leave non-transferable elements in the Illustrator document intact unless the D3 script has directly changed them. A "To Illustrator" merge updates elements in place in the document hierarchy, leaving Illustrator-specific non-transferable elements untouched. In contrast, much of the non-transferable information in a D3 visualization is usually encapsulated in the D3 script itself. To avoid complications with interactive callbacks that may already exist, Hanpuku creates a fresh DOM with each "To D3" merge, only containing elements from the Illustrator document, with any data bindings attached. This way, the script can cleanly reattach its non-transferable interactive callback events.

5.3 Examples

In this section we demonstrate how Hanpuku supports visualization iteration through several examples based on real-world, published infographics. In Figure 3, we recreate the infographic shown in Figure 2. Notably, the original infographic contains a name out of alphabetical order — without a bridge between a generative and a drawing tool it is difficult to fix this problem. With Hanpuku as a bridge, however, it is straight-forward to re-order the names, or to even experiment with different layouts, without discarding manual effort.

It is important to note in this example the difference between generative encodings and manual encodings. In the workflow in Figure 3, colors and typefaces are applied manually in Illustrator, rather than generativity using D3. Therefore, while additional data can be placed appropriately in the middle of the design process — as the layout is defined generativity — any new elements will have the default color and typeface from the D3 script. While other bridging tools, employing some of the strategies outlined above, might be able to infer the designer's intent, Hanpuku's simple, proof-of-concept design does not have this capability. Instead, the designer can choose to continue to manually enforce color and typeface patterns for any new elements, or they can formalize those decisions in the D3 script. In either case, previous work is preserved.

As a second example we recreated part of a diagram from the ACM SIGGRAPH 2010 Conference [17], designed by Isabel Meirelles. In this case we were given permission to review the designer's original process and design artifacts. Notably, she implemented much of the design using Processing code, creating new PDF files each time a generative change was made. Each generative change consequently lost any previous design work in Illustrator, requiring significant amounts of rework to re-apply the manual aspects of the diagram.

We have attempted to follow her process as closely as possible in Figure 10 using Hanpuku (and D3 instead of Processing). Hanpuku made the iterative process far less difficult by removing the rework incurred in the original design process — generative adjustments, such as tweaking the scale in Figure 10(e), were straight-forward to perform without having to repeat previous manual effort. Consequently, our version took very little time to create, and we were able to skip many of the steps that she was forced to take.

6 Other Uses of the Model

The bridge model is meant to generalize beyond the specific technical considerations of bridging D3 and Illustrator. We made some strategic decisions in implementing the bridge model in Hanpuku; bridges between other environments will similarly need other considerations. In this section we briefly outline a hypothetical Processing-Illustrator bridge and a Processing-Photoshop bridge to discuss different instantiations of the bridge model, as well as to emphasize its generality.

6.1 Bridging Processing and Illustrator

Processing is a programming language that has gained currency among designers to create visualizations. Some designers employ a one-way workflow between Processing and Illustrator, such as the original workflow used to create the SIGGRAPH infographic in Figure 10. Here we describe a possible bridge between the two tools, illustrated in Figure 11.

The first issue to be resolved is maintaining data bindings. While Processing can emit PDFs, it does not ordinarily maintain ID tags with the graphical objects. One workaround would be to insert ID tags into the metadata fields in the PDF output through an extension of Processing's PDF export library. Illustrator would then edit the PDF.

For Hanpuku, we were able to use D3's update function to use parts of existing graphics. Processing's drawing commands do not provide this functionality. Therefore, we would use one-way integration: when a designer modified the Processing code, the bridge would propagate the changes forward to Illustrator, merging their changes on each side.

The bridge software for this scenario would then merge based on the following files: 1) an initial visualization in PDF format produced by Processing; 2) a modified PDF of the original visualization, edited in Illustrator; and 3) a modified PDF produced by Processing with additional generative edits. The bridge would then identify differences between the PDFs in 1 and 2, and merge those differences on 3, producing a PDF containing changes from both the drawing and generative tools. Rather than comparing artifact differences directly, this

Fig. 10: Partial recreation of an ACM SIGGRAPH 2010 Conference Diagram [17]. Blue steps are done in the generative tool; green steps in the drawing tool. (A) A graphic is produced using a D3 script, encoding conference participant numbers for each group (student volunteers, conference chair, etc) with bar width, and months of participant involvement as the curved bar length. (B) Illustrator's tools are used to adjust the colors. (C) The direction of the bars is reversed, and the width of each bar is changed to encode total hours spent by the group. (D) Colors are again adjusted in Illustrator, this time identifying appropriate PANTONE colors, and labels are added. (E) As most of the bars happen to nearly achieve right angles, the scale is increased in D3 to achieve that effect. (F) Scale labels are drawn using Illustrator's tools.

approach attempts to reconstruct and replay the user's actions, similar to the way provenance mechanisms work in VisTrails [25, 5].

6.2 Bridging Processing and Photoshop

Another potential instantiation of the bridge model is between Processing and Photoshop, shown in Figure 12. A challenge with this bridge is that the common representation of the visualization between the two programs is a bitmap — there is no way to attach metadata directly to graphical elements as there is with PDF. A basic approach takes two files from Processing: a bitmap file of the visualization, and an additional text, log file containing a many-to-many mapping between graphical element identities and related pixels. The bridge also requires a Photoshop extension that augments the tool's existing ability to export a text history log with a one-to-many mapping between Photoshop actions and the affected pixels in a given bitmap image. Given both log files and both images, the merge program — as either a Photoshop extension or an independent program — would extend or adjust the relevant Photoshop actions so that the effects are still applied to data-bound pixels, even if those locations have changed.

Bridging tools through a bitmap presents a different set of challenges similar to those that Savva et al address in the Revision system [24]. While unambiguous operations could be supported, such as a drawing operation affecting only pixels associated with data, operations on pixels that correspond to multiple occluding marks present identification and conflict resolution problems, as shown in Figure 12. The bridge model makes the problem in this particular design clear — this bridge does not adequately provide for maintaining graphical element identities, or, more specifically, data bindings.

This is not to say that bridges cannot be constructed between bitmap-based tools, or that such bridges are not useful. For example, the bridge model describes how it may be possible to perform data-aware merges between 3D volume renderings and post-processing effects performed on 2D rasterized graphics. In the case of this basic model, improvements might include more intelligent drawing tools that use the associated data IDs to constrain painting effects, limiting their influence to only the relevant pixels. A different approach would be to utilize layers instead of pixel maps to keep element identities distinct.

7 Discussion and Future Work

The bridge model describes how different modes of work can be used in the same workflow, fitting the best tool to the best task. For example, it is completely possible to perform typography and label placement adjustments with D3, or to manually draw data-driven marks in a chart with Illustrator — we commonly see examples of each of these in practice. Generative visualization tools and drawing tools overlap in some of their technical capabilities, but, as we have shown, using both modes of work in the same workflow can constrain the ability to iterate. The bridge model helps identify what is common between each way of working, what is distinct, and how these differences can

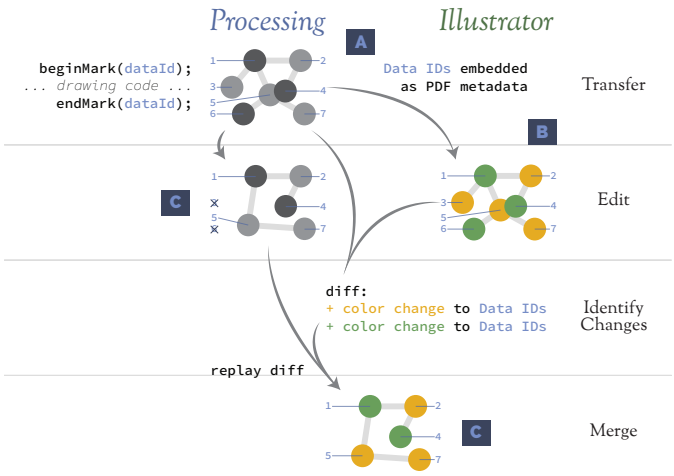

Fig. 11: A hypothetical bridge between Processing and Illustrator. Additional syntax is added to Processing's language to associate drawing commands with data IDs (1), that are preserved in an exported PDF file. The exported file is edited in a drawing program (2). Two nodes are then deleted in the Processing sketch (3). An external merge program compares (1) and (2), identifying any changes made by a drawing program, and replays these changes on (3), producing a PDF file with both the drawing and generative changes.

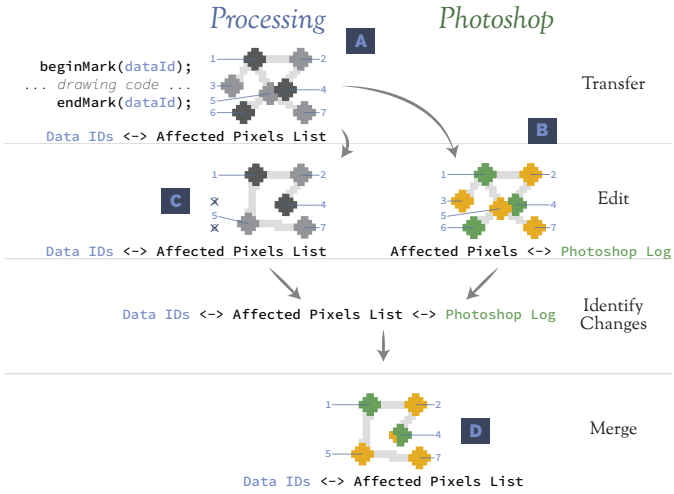

Fig. 12: A hypothetical bridge between Processing and Photoshop. As before, additional syntax is added to Processing's language to associate drawing commands with data IDs (1), that are preserved in an external file that maps data IDs to their affected pixels. The exported bitmap is edited in a drawing program (2), and a log is created, matching the drawing program's actions to the affected pixels. Two nodes are then deleted in the Processing sketch (3). A merge program replays the drawing program's log on the updated bitmap, adjusting the effects where data pixels are missing or have moved (4). Note that, in this naive bridge, an artifact has been left on node D because a painting action meant for node E also affected it.

be managed through the design workflow. The model is not a panacea — it does not make all tools come together. Rather, we propose the bridge model as a different way to think about how we create visualization tools that are rich, flexible, and efficient across the broad range of tasks performed throughout the design process.

Though we have mainly discussed bridges between distinct software tools, the concepts are still important for all-in-one tools that attempt to encompass both generative and drawing modes of working such as iVisDesigner [18]. Internally, such tools must still consider the effects of generative operations on existing drawings, as well as the effects of drawing operations on existing generative specifications.

The model does not, however, encompass approaches like CSS that seek to separate out design tasks, albeit it in a generative way. While stylesheets separate the stylistic and functional aspects of a visualization, a stylesheet does not contain an independent representation — there is nothing to merge. The model instead describes workflows that include a variety of tools, each of which operates independently on the visualization.

With any bridge, some sacrifices are likely unavoidable. As we discussed in Section 6.2, a hypothetical Processing-Photoshop bridge is likely to be constrained by the technical limitations of each tool, including what aspects of the visualization can be shared and what kinds of changes can be merged. Both hypothetical bridges, as well as Hanpuku itself, each enable a more iterative workflow, but limitations remain. In all the bridges we have discussed, users likely need to understand each tools' underlying document structure in order to effectively prepare for merging.

The bridge model exposes ways that visualization toolkits and drawing programs can make themselves more interoperable. Our choice of specific tools is not an accident: we repeatedly use Illustrator as an example throughout this paper because it is a widely-used drawing program that preserves arbitrary metadata attached to elements in its documents — data bindings can be preserved when visualizations are edited. We also focus heavily on D3 because it can perform data joins with existing graphical elements such that manual changes done in a drawing program can survive generative updates. For now, the pass-through relationship between these two tools makes interoperability straight-forward.

There are, however, additional benefits that can come from their flexibility. For example, Illustrator currently allows users to select elements by color and other visual properties. Because our system now adds metadata and identify information to visual objects, it would be straightforward to support user selections based on data, similar to the GUESS system [1].

Going forward we are interested in instantiating other bridges, such as those we describe in Section 6. We believe these, and other variations, will become easier to design and implement as the inputs and outputs of proprietary tools become more easily accessible. Another interesting line of future work is to consider a format standard for visualizations that explicitly supports metadata attached to graphical elements. This standard would support writing these files, reading these files, editing the files, as well as explicitly making use of the metadata in visualization tools.

References

[1] E. Adar. Guess: A language and interface for graph exploration. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '06, pages 791–800, New York, NY, USA, 2006. ACM.

[2] A. Baumann. *The Design and Implementation of Weave: A Session State Driven, Web-based Visualization Framework*. PhD thesis, 2011. AAI3459174.

[3] A. Bigelow, S. Drucker, D. Fisher, and M. Meyer. Reflections on how designers design with data. In *Proceedings of the 2014 International Working Conference on Advanced Visual Interfaces*, AVI '14, pages 17–24, New York, NY, USA, 2014. ACM.

[4] M. Bostock, V. Ogievetsky, and J. Heer. D3; data-driven documents. *Visualization and Computer Graphics, IEEE Transactions on*, 17(12):2301–2309, Dec 2011.

[5] S. P. Callahan, J. Freire, C. E. Scheidegger, C. T. Silva, and H. T. Vo. Provenance and annotation of data and processes. chapter Towards

Provenance-Enabling ParaView, pages 120–127. Springer-Verlag, Berlin, Heidelberg, 2008.

[6] S. Carter, G. Aisch, and M. Bostock. Svg crowbar, 2015. Available: http://nytimes.github.io/svg-crowbar/. [Accessed: 13-Feb-2016].

[7] G. Caviglia, M. Mauri, M. Azzi, and G. Uboldi. Raw: The missing link between spreadsheets and vector graphics, 2013. Available: http://raw.densitydesign.org/. [Accessed: 13-Feb-2016].

[8] R. Costa and D. K. Sobek. Iteration in engineering design: inherent and unavoidable or product of choices made? In *ASME 2003 International design engineering technical conferences and Computers and information in engineering conference*, pages 669–674. American Society of Mechanical Engineers, 2003.

[9] J. D. Denning and F. Pellacini. Meshgit: Diffing and merging meshes for polygonal modeling. *ACM Trans. Graph.*, 32(4):35:1–35:10, July 2013.

[10] H. Fa-Zhi, W. Shao-Mei, and S. Guo-Zheng. From wysiwyg to wysiwis: research on cscw based cad. In *Communications, 1999. APCC/OECC '99. Fifth Asia-Pacific Conference on ... and Fourth Optoelectronics and Communications Conference*, volume 2, pages 1095–1096 vol.2, Oct 1999.

[11] L. Grammel, C. Bennett, M. Tory, and M.-A. Storey. A Survey of Visualization Construction User Interfaces. In M. Hlawitschka and T. Weinkauf, editors, *EuroVis - Short Papers*. The Eurographics Association, 2013.

[12] J. Harper and M. Agrawala. Deconstructing and restyling d3 visualizations. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, UIST '14, pages 253–262, New York, NY, USA, 2014. ACM.

[13] S. Huron, Y. Jansen, and S. Carpendale. Constructing visual representations: Investigating the use of tangible tokens. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2102–2111, Dec 2014.

[14] R. Krishnamurthy and M. Zloof. Rbe: Rendering by example. In *Data Engineering, 1995. Proceedings of the Eleventh International Conference on*, pages 288–297, Mar 1995.

[15] B. Lee, G. Smith, N. H. Riche, A. Karlson, and S. Carpendale. Sketchinsight: Natural data exploration on interactive whiteboards leveraging pen and touch interaction. In *2015 IEEE Pacific Visualization Symposium (PacificVis)*, pages 199–206, April 2015.

[16] D. Lloyd and J. Dykes. Human-centered approaches in geovisualization design: Investigating multiple methods through a long-term case study. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2498–2507, Dec 2011.

[17] I. Meirelles. Diagram: Acm siggraph 2010 conference, 2010. Available: http://isabelmeirelles.com/acm-siggraph-2010/. [Accessed: 13-Feb-2016].

[18] D. Ren, T. Hollerer, and X. Yuan. ivisdesigner: Expressive interactive design of information visualizations. *Visualization and Computer Graphics, IEEE Transactions on*, 20(12):2092–2101, Dec 2014.

[19] C. Robinson, S. Schube, and D. Savitzky. The HBO recycling program. Available: http://grantland.com/features/the-hbo-recycling-program/. [Accessed: 13-Feb-2016].

[20] S. F. Roth, J. Kolojejchick, J. Mattis, and J. Goldstein. Interactive graphic design using automatic presentation knowledge. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '94, pages 112–117, New York, NY, USA, 1994. ACM.

[21] S. Santagata, M. L. Mendillo, Y.-c. Tang, A. Subramanian, C. C. Perley, S. P. Roche, B. Wong, R. Narayan, H. Kwon, M. Koeva, A. Amon, T. R. Golub, J. A. Porco, L. Whitesell, and S. Lindquist. Tight coordination of protein translation and hsf1 activation supports the anabolic malignant state. *Science*, 341(6143), 2013.

[22] A. Satyanarayan and J. Heer. Lyra: An interactive visualization design environment. *Comput. Graph. Forum*, 33(3):351–360, June 2014.

[23] A. Satyanarayan, R. Russell, J. Hoffswell, and J. Heer. Reactive vega: A streaming dataflow architecture for declarative interactive visualization. *Visualization and Computer Graphics, IEEE Transactions on*, 22(1):659–668, Jan 2016.

[24] M. Savva, N. Kong, A. Chhajta, L. Fei-Fei, M. Agrawala, and J. Heer. Revision: automated classification, analysis and redesign of chart images. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, pages 393–402. ACM, 2011.

[25] C. Scheidegger, H. Vo, D. Koop, J. Freire, and C. Silva. Querying and creating visualizations by analogy. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1560–1567, Nov. 2007.

[26] C. Sun, S. Xia, D. Sun, D. Chen, H. Shen, and W. Cai. Transparent adaptation of single-user applications for multi-user real-time collaboration. *ACM Trans. Comput.-Hum. Interact.*, 13(4):531–582, Dec. 2006.

[27] R. Teal. Developing a (non-linear) practice of design thinking. *International Journal of Art —& Design Education*, 29(3):294–302, 2010.

[28] W. F. Tichy. Design, implementation, and evaluation of a revision control system. In *Proceedings of the 6th International Conference on Software Engineering*, ICSE '82, pages 58–67, Los Alamitos, CA, USA, 1982. IEEE Computer Society Press.

[29] J. Walny, J. Haber, M. Dörk, J. Sillito, and S. Carpendale. Follow that sketch: Lifecycles of diagrams and sketches in software development. In *Visualizing Software for Understanding and Analysis (VISSOFT), 2011 6th IEEE International Workshop on*, pages 1–8, Sept 2011.

[30] J. Walny, S. Huron, and S. Carpendale. An Exploratory Study of Data Sketching for Visual Representation. *Computer Graphics Forum*, 2015.

[31] C. Weaver. Building highly-coordinated visualizations in improvise. In *Proceedings of the IEEE Symposium on Information Visualization*, INFOVIS '04, pages 159–166, Washington, DC, USA, 2004. IEEE Computer Society.

[32] K. Wongsuphasawat, D. Moritz, A. Anand, J. Mackinlay, B. Howe, and J. Heer. Voyager: Exploratory analysis via faceted browsing of visualization recommendations. *Visualization and Computer Graphics, IEEE Transactions on*, 22(1):649–658, Jan 2016.