Idyll Studio: A Structured Editor for Authoring Interactive & Data-Driven Articles

Matthew Conlen
mconlen@uw.edu
University of Washington
Seattle, Washington, USA

Megan Vo
meganv@idyll-lang.org
Idyll Collective
Seattle, Washington, USA

Alan Tan
alan@idyll-lang.org
Idyll Collective
Seattle, Washington, USA

Jeffrey Heer
jheer@uw.edu
University of Washington
Seattle, Washington, USA

ABSTRACT

Interactive articles are an effective medium of communication in education, journalism, and scientific publishing, yet are created using complex general-purpose programming tools. We present *Idyll Studio*, a structured editor for authoring and publishing interactive and data-driven articles. We extend the Idyll framework to support reflective documents, which can inspect and modify their underlying program at runtime, and show how this functionality can be used to reify the constituent parts of a reactive document model—components, text, state, and styles—in an expressive, interoperable, and easy-to-learn graphical interface. In a study with 18 diverse participants, all could perform basic editing and composition, use datasets and variables, and specify relationships between components. Most could choreograph interactive visualizations and dynamic text, although some struggled with advanced uses requiring unstructured code editing. Our findings suggest *Idyll Studio* lowers the threshold for non-experts to create interactive articles and allows experts to rapidly specify a wide range of article designs.

CCS CONCEPTS

• **Human-centered computing** → **Visualization systems and tools**; **User interface programming**; • **Applied computing** → **Hypertext / hypermedia creation**.

KEYWORDS

interactive articles, computational media, explorable explanations

ACM Reference Format:
Matthew Conlen, Megan Vo, Alan Tan, and Jeffrey Heer. 2021. Idyll Studio: A Structured Editor for Authoring Interactive & Data-Driven Articles. In *The 34th Annual ACM Symposium on User Interface Software and Technology (UIST '21), October 10–14, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3472749.3474731

UIST '21, October 10–14, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8635-7/21/10…$15.00
https://doi.org/10.1145/3472749.3474731

1 INTRODUCTION

Interactive articles [31] like Victor's *Explorable Explanations* [73] utilize dynamic text, manipulable controls, and interactive graphics to present information to viewers in an engaging, reader-driven form. This format is a useful medium of communication across domains like journalism [28, 66], education [45], and scientific publishing [20] because interactive articles can promote active reading [73], foster engagement [17, 28], and lead to improved learning outcomes [36, 66]. While some tools exist to support the creation of these articles [14, 61], they typically require that authors are familiar with technical tools like the command-line and often mandate the use of general purpose programming languages like JavaScript or Python. These requirements make it difficult or impossible for those with limited programming knowledge—for example some educators and journalists—to create interactive articles [14]. For those who do have the requisite programming knowledge, the article creation process is still time consuming and cognitively demanding [31].

We present *Idyll Studio*, a novel structured editing interface for authoring data-driven, interactive documents. With this tool, authors can use a graphical, WYSIWYG-style interface to create, edit, and publish interactive articles. We show through a first-use study how, in many cases, the tool eliminates the need to use general purpose programming tools, and how it can streamline the article creation process for both novice users and technical experts.

Our contributions can be summarized as follows:

- **Extensions to the Idyll framework** to support *reflective* documents which can inspect and modify their own programs at runtime. We use these extensions to create a flexible structured editing API which powers *Idyll Studio*.
- ***Idyll Studio***—a structured editor for authoring interactive & data-driven articles—released as open source software that can be downloaded and used today.[1]
- A **first-use study** in which 18 participants with a wide range of backgrounds used *Idyll Studio* to perform rapid prototyping tasks. We found that *Idyll Studio* enabled non-technical users to complete tasks they otherwise could not, and allowed expert users to create articles more rapidly and with fewer cognitive demands compared to existing tools.

[1] https://github.com/idyll-lang/idyll-studio/releases

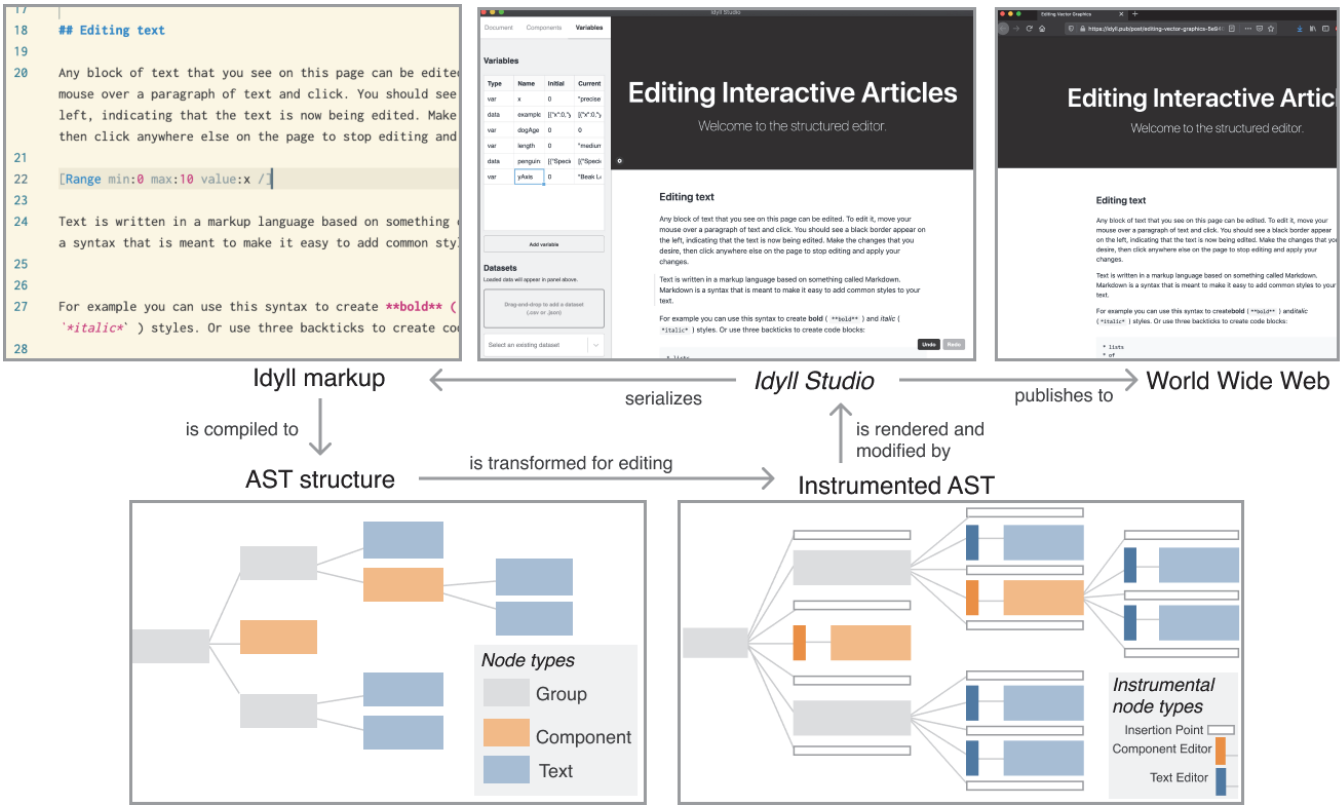

Figure 1: *Idyll Studio* builds on the Idyll markup language by creating a structured editing interface for creating, manipulating, and publishing interactive articles. We implement a tree expansion algorithm to transform arbitrary Idyll programs into structured editing interfaces. *Text editors* allow an author to edit and style text while *component editors* allow for orchestration of component behaviors. *Insertion points* can be used to add new components or text blocks to the article via drag-and-drop.

2 MOTIVATION & RELATED WORK

Idyll Studio builds on research from narrative visualization [67], computational notebooks [60], literate programming [39], user-interface toolkits [50, 72], and word processors [37]. *Idyll Studio* is a structured editor, and can be seen as a hybrid word processor and interface builder tool for interactive articles.

2.1 Narrative InfoVis & Interactive Articles

The information visualization research community has examined the role of narrative in explanatory visualizations, including storytelling approaches [26, 67, 69], the effects of sequence [33] and framing [32], as well as in-depth analysis of usage of different narrative visualization formats such as data comics [4, 5], video [1, 10], and interactive articles [46, 77]. We aim to support narrative visualization authoring in the interactive article format.

A number of systems have been developed with similar goals [42, 48, 53, 61, 74], but existing systems are either limited in their expressivity or require that authors use general purpose programming tools. For example, some WYSYIWG-style editors allow users to insert arbitrary interactive components into web-based articles [12]; however, in previous such systems, components are treated as independent black boxes and cannot communicate amongst one another—a requirement for common design patterns like linked text and graphics [77]. On the other hand, code-based approaches, while sufficiently expressive, have a high threshold for productive use and are often inappropriate for non-technical users like some journalists and educators [42].

Our work extends Idyll [14], one framework for authoring interactive articles. While Idyll supports the creation of sophisticated interactive reading interfaces [15, 16], it still requires that users learn a markup language and use tools like git and the command-line [31], even to make simple changes to the text of an article. *Idyll Studio* provides a GUI that allows users to perform basic tasks like editing, composing, and publishing documents without requiring general purpose programming tools. *Idyll Studio* reifies [8] the four parts of Idyll's reactive document model (components, text, state, and styles) in elements of a graphical interface that allows users to rapidly sequence text and interactive elements while allowing them to script domain-specific graphics using visualization and user interface libraries when needed. In Idyll, document structure is intentionally separated from the implementation of parameterized components [79]: users write text and arrange components, and then choreograph component interactions by connecting reactive variables to meaningful domain-specific parameters that components expose. Components are created using domain-specific

languages like D3 [9] or Processing [59], visualization grammars such as Vega [64] or Vega-Lite [63], or graphical tools that support the creation of interactive graphics and data visualizations. For example, Lyra [62, 78] lets users specify interactive visualizations by example, while Kitty [34] and Apparatus [65] support the authoring of interactive diagrams through sketching and direct manipulation, respectively. Such tools are complementary to *Idyll Studio*, and can be used to create interactive graphics which are then imported into the system and parameterized using reactive variables [13].

Lee et al. [43] articulate a model of the visual data storytelling process. They describe how multiple collaborators (e.g., data analyst, scripter, and editor) work together to create and present data-driven stories in professional settings, which serves as a guideline for the different roles and tasks that a tool like *Idyll Studio* should support. For example, editors can easily edit text and component properties without needing to understand how the components are implemented, and scripters can use the interface to rapidly iterate on a component's implementation and see how changes look in context. We designed *Idyll Studio* with the assumption that the majority of exploratory data analysis would be done prior to the creation of an interactive article and our structured interviews with user study participants (§4.3) support this.

2.2 Structured Editing

Our system follows a structured editing approach [71], in which authors use an interface to effect changes in the abstract syntax tree (AST) of an underlying Idyll program. In contrast to many structured editors (e.g. [30, 41]), our system allows users to interact with a live running, visual version of their program rather than a structured version of the textual representation. We are not the first to apply structured editing to multimedia documents; for example, the Grif system [57] took a structured approach to formatting complex documents. However, to our knowledge we are the first to support the specification of data-driven, highly interactive hypertext articles through a structured editing interface.

The structured approach eliminates a class of syntax errors [40] (since every change to the AST will result in a valid program) that may make it more beginner friendly, but possible drawbacks include restricting the expressiveness or fluidity of use for expert programmers who prefer to work in unstructured text editors [49]. Sketch-n-Sketch [29] combines unstructured programming with direct manipulation, allowing changes made in either modality to affect the other. Since *Idyll Studio* supports arbitrary components, we do not expose a direct manipulation tool for editing graphics, but rather allow authors to modify any component through general purpose editing instruments [7]. A domain-specific graphics editor like Sketch-n-Sketch could be embedded inside of *Idyll Studio* to allow direct manipulation of a specific set of components using the instrument API we developed (§3.2) or used in an independent but interoperable manner (as in §3.3).

2.3 Runtime Modification of Code & Interfaces

Programs that can inspect and modify their internal structure at runtime support *reflection* [44, 68]. This functionality, available in many modern programming languages [2, 24, 35], allows programs to dynamically adapt based on input from users, environmental sensors, or other sources [27, 47]. To build a structured editor where authors can compose articles with low levels of spatial indirection, we designed an AST transformation to embed *instrumental components* directly in interactive documents and extend Idyll to support reflection, allowing these instruments to modify the underlying program as it runs. Put another way, we augment a running Idyll program to convert it into its own live WYSIWYG editor.

Hypercard [3] also supported customization of application interfaces from within the application itself, allowing end-users to create multimedia presentations including hyperlinks and input forms. *Idyll Studio* supports similar features coupled with a reactive variable system, and allows for the rapid creation of layouts linking text and interactive graphics, as well as full control over the look-and-feel of the generated documents. Both systems achieve this through the modification of source code, but researchers have also developed techniques to let end users customize GUIs that do not explicitly support runtime augmentation. For example, pixel-based reverse engineering [18], runtime toolkit overloading [21], and interface attachments [54] enable modification of user interfaces to support augmented interactions [19], personalization [25] and improved accessibility [22].

2.4 Notebooks & Literate Programming

The ability to mix code and prose exists in other systems such as computational notebooks [55] and literate programming environments [39]. While some notebook environments support similar features to Idyll (e.g., ObservableHQ [52] has a similar reactive runtime and support for embedded graphics), these environments typically target exploratory analysis use cases in which an analyst uses the notebook as a feature-rich REPL to interactively construct visualizations and data transformations to support their analytic needs [56] in a shareable and collaborative format [76]. While Observable and other literate programming environments are centered on programming, our interface centers on direct manipulation, in which authors can compose interactive articles primarily through the use of familiar interactions like point-and-click and drag-and-drop. *Idyll Studio* is designed to support the creation of rich interactive *reading interfaces* targeted at a more general audience than analysts and data scientists, and affords the ability to customize nearly every aspect of the look-and-feel of the final output.

Literate programming environments like Codestrates [58] and Leisure [11] offer a similar editing experience to computational notebooks but offer additional flexibility—for example the ability to customize the user interface—through the ability to use the programming environment to modify its own user interface. By making the entire medium dynamic, literate programs can "blur the line" between authoring environment and application [38]; this technique has been extended to data visualization for ubiquitous analytics [6]. In this work, we extended Idyll to similarly support reflective documents; however, *Idyll Studio* clearly differentiates between the views for authoring and reading an interactive article. When an interactive article created with *Idyll Studio* is compiled for publication, the instrumental elements used by the authors and designers of the document are hidden from the readers' view by default. Because readers are typically interested in the *content* being presented rather than the underlying implementation of the

graphics, model, or user interface [17], centering these "behind the scenes" elements in a published interface can be a distraction and lead to unnecessary cognitive load [70] for most readers.

3 IDYLL STUDIO

Idyll Studio is a structured editor for authoring and publishing interactive and data-driven articles. The software is an open source desktop application that supports Windows, MacOS, and Linux. The tool is built on top of the Idyll [14] markup language and can be used to edit and view existing articles or create new ones. Programs created with *Idyll Studio* are serialized back to idiomatic markup, compatible with any tools used with Idyll markup today.

The interface (Figure 2) consists of two major parts: an article viewer A and an editing panel B on the left-hand side of the interface. The article viewer shows a nearly-WYSIWYG view of the article.[2] The editing panel has three tabs, *document*, *components*, and *variables*. Together these elements reify the Idyll reactive document model into concrete interface objects; in particular it gives authors access to *text*, *components*, *state* (variables and datasets), and *styles* in order to construct customized interactive articles.

3.1 Core Concepts

3.1.1 Compose text and components. A primary task when authoring interactive articles is the composition of text and components. As shown in Figure 1, the modified version of the AST that is rendered into the *Idyll Studio* article viewer includes three types of instrumental nodes: *insertion points*, *text editors*, and *component editors*. These three instrumental elements are used to manipulate existing text and components, as well as to insert new text elements or components. To invoke a text editor C an author can simply click on an existing block of text. This will cause a visual indicator to appear, signifying that the selected text can now be edited, and the text displayed will change from rendered HTML to the underlying Idyll markup. An author can then directly edit this text (including adding markdown tags or components), and when they click away from the text block, it will transition from the editing mode back to displaying rendered HTML. To use the insertion points D to add new content to an article, an author navigates to the component tab on the left-hand sidebar, and then drags a component from the component library displayed in the sidebar onto the article view. As they do this, gray drop targets will appear in the interface where the component can be placed.

3.1.2 Parameterize components. Next to each component in the article view is a small gear icon which can be clicked to invoke the component editor instrument E. This instrument is used to parameterize components by editing the type and value of existing properties or adding additional properties to the component invocation. When the component editor is invoked, an outline appears around the referent component in the article viewer and an editing window appears below it. The editor's properties tab provides a structured interface for editing component properties, as well as a link to component documentation and the ability to delete the

[2]We state "nearly" WYSIWYG because Idyll Studio includes *instrumental* components in the interface which allow an author to edit text and component properties and styles. These components do not affect the layout or styles of the page.

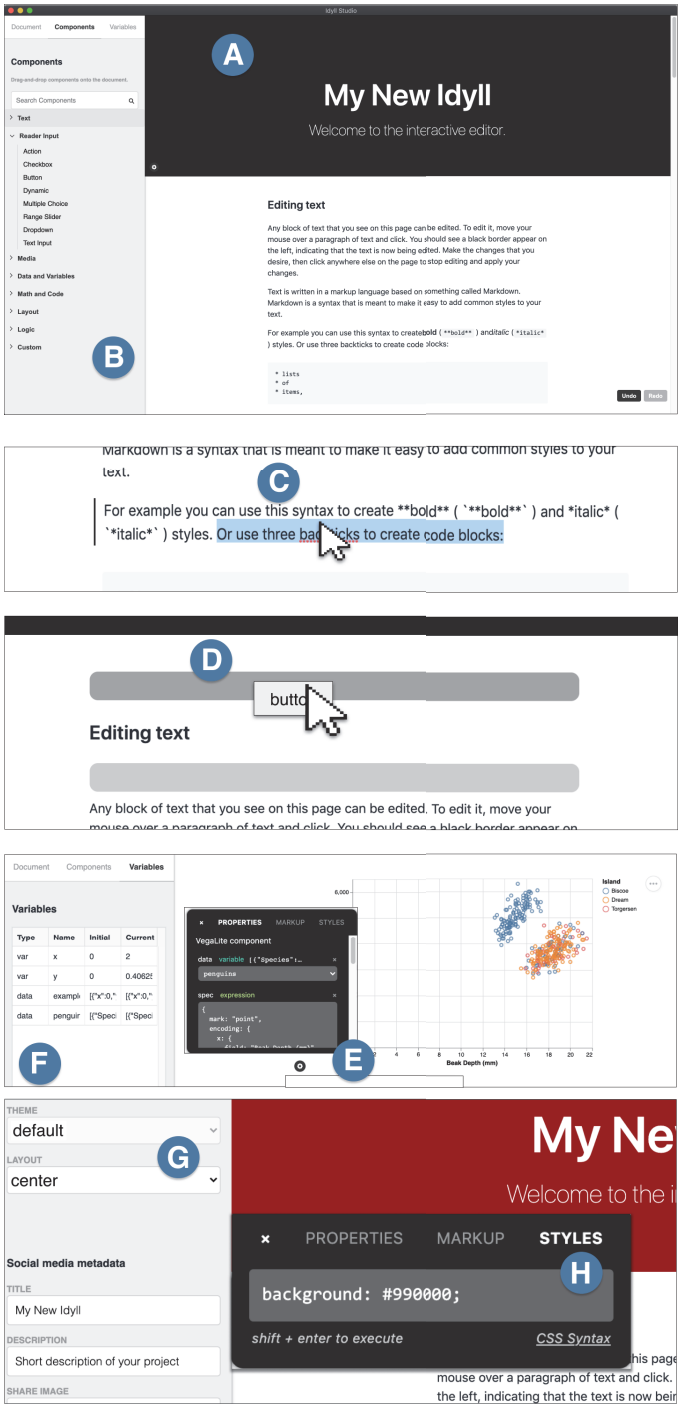

Figure 2: *Idyll Studio* utilizes direct and instrumental manipulation to let authors compose and choreograph interactive and data-driven articles. The interface consists of a live article view (A) and an editing panel (B) on the left-hand side. Authors can use text editors (C), insertion points (D), and component editors (E) to construct their articles. The variables panel (F) lets the author inspect and modify the document's reactive state, and document (G) and component (H) styles can be applied to customize the aesthetics.

component from the article. Properties can have one of several types, including *literal* (e.g., a string, number, or boolean), *expression* (a reactive expression that references one or more variables or datasets), or *variable* types (a two-way binding between the component property and a variable). By default property values are edited using a text input widget, but this varies by property type: for example, properties with the *variable* type are set via a dropdown containing a list of defined variables that can be bound.

3.1.3 Define variables and datasets. In order to use *expression* and *variable* type properties, authors need to define variables and datasets using the variables panel F. The panel shows a list of all variables and datasets currently defined, their initial and current values in a spreadsheet-like interface, and a button to add additional variables. The cells of this view can be edited directly to change the variable's type (*var*, *derived*, *data*), name, initial value, and current value. The panel also includes a *data* view, which can be used to import datasets by dragging and dropping a file from the author's file system (CSV and JSON files are supported) or selecting from a list of built-in examples. Once a dataset is added via the dataset view, a new row with the type *data* will appear in the variable view's grid interface. Any variables or datasets that have been loaded in this view can be referenced in *expressions*, and any with type *var* can be used in two-way bindings. A variable's current and initial value start with the same value, but as an author interacts with variable-bound widgets in the article, the current value will update in real time, giving an overview of the article's most recent state. By setting a variable's current value directly, an author can quickly see how components would look under various configurations.

3.1.4 Customize aesthetics. In addition to specifying the composition and behavior of text and components, authors also need to customize the aesthetics of their article [28]. This can be done in *Idyll Studio* through themes and layouts which affect the entire article, or targeted style rules which only apply to specific component invocations. To modify the theme or layout, users can navigate to the *Document* tab in the left sidebar and choose from existing styles or provide their own G. To add component-specific styles, authors can use the *Style* tab in the component editor instrument and add CSS rules to modify, for example, the color, font, position, and size of elements H. In some cases more complex layouts need to be specified via the composition of built-in components. To achieve this, *layout* components can be dragged and dropped onto the article using insertion points, and then manipulated like other components. For example, an *Aside* component can be added to display content (text or another component) in an article's margin.

3.1.5 Access & modify the underlying Idyll markup. In certain cases it will be desirable to access & modify the underlying Idyll markup. While most programs can be specified by using the structured editor, certain tasks such as invoking logical components are more ergonomic when done using unstructured text. Take for example the *Switch* component, which conditionally displays its children based on a variable property: to modify the contents of children that do not visually appear in the article, an author may choose to edit the underlying Idyll markup directly rather than manually changing the *Switch*'s value each time they want to edit a different child. Users can select the *markup* button on the component instrument panel to reveal that component's Idyll specification. By surfacing only the relevant code snippet directly in the interface, users can make targeted edits without being overwhelmed by seeing the entire program at once. Power users might prefer to edit completely unstructured text as opposed to using our GUI. These users can still access and manipulate the full text of the underlying Idyll program via their file system and their programming tool of choice. We envision that supporting these various interfaces will be important to enable collaborations between less technically inclined users, including journalists and educators, and programmers more comfortable with unstructured editors.

3.2 Implementation

When users edit text or components through *Idyll Studio*, they are using the interface to make changes to the AST of a live Idyll program.[3] In order to support structured editing with minimal temporal and spatial indirection, instruments are placed directly into the document. To do this, we extend the Idyll framework in two major ways. First, we developed a transformation (Figure 1) that expands the AST to wrap text and component nodes with instrumental elements—such as text and component editors—and add insertion targets, which an author can select through a drag-and-drop interaction. Second, we added support for *reflection*, allowing components to effect changes in the document's underlying program while running. Together, these two features allow us to support structured editing with direct interactions by seamlessly inserting instruments *in situ* in a running interactive article.

3.2.1 Instrumentation. We implemented three instrumental widgets: a *component editor*, a *text editor*, and an *insertion target*. By inserting them into the AST, the instruments appear directly in the article where the content that they modify appears. While we use the same component editor widget for all components, additional instruments could be added and associated with specific subsets of components. The tree expansion algorithm performs a single breadth-first traversal of the AST: *(1) For each component and text node, replace that component with an instrumental node that has the original component as its only child; (2) for each component and group node, interleave that node's existing children with "insertion point" instruments (components without any children may or may not have an insertion point added as a child, depending on a parameter in the component's metadata).* While this step allows us to render these *instruments* into the document, we need to add reflection in order for them to apply changes to the program.

3.2.2 Reflection. Reflection is the ability for a program to inspect and change its own behavior at runtime. We extended the Idyll runtime to support this, allowing the active AST to be changed while the document retains its state. We added an API with a single function `updateAST(newAst)`, that when called will update the actively running Idyll program to update to conform to the structure in the new AST. Components can use this function to make changes that modify the behavior of the program at runtime. For simplicity, when passing references to the AST we always use the original,

[3] See https://github.com/idyll-lang/idyll/tree/master/packages/idyll-ast .

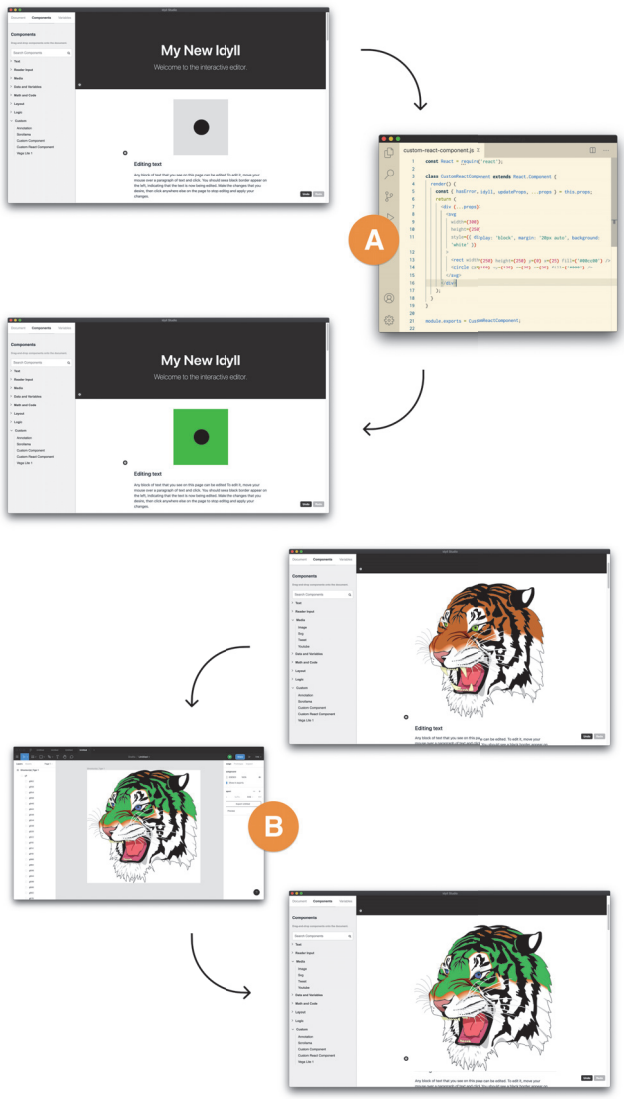

Figure 3: Two examples of how *Idyll Studio* interoperates with existing tools: (A) users can click "edit" to call up their system text editor and modify a component's source code; any changes made are immediately reflected in the article view; (B) similarly, users can edit media assets like images or SVGs by invoking a domain-specific editor like Figma and see changes immediately reflected in context.

non-transformed copy and make changes to that version before re-running the tree expansion. If these methods are accessed from within an instrumental node, the `nodeId` will refer to the instrument's child, making implementation of the component and text editors straightforward. A suite of helper functions to query and transform the AST is available.

3.3 Interoperability

While *Idyll Studio* focuses on specifying the composition and behavior of interactive articles, it may be desirable to utilize other types of editing environments in certain cases, such as when editing a component's implementation (which may involve writing low-level graphics code) or when creating custom vector graphics. While there is an argument to be made that users might prefer this functionality to exist directly in *Idyll Studio*, we think that there will always be cases where a domain-specific editor provides a better experience than a general purpose one that we could provide. Rather than try to build a monolithic system that supports every type of editing functionality, we instead choose to support interoperability with existing tools that have been highly refined for their specific tasks. Figure 3 shows how *Idyll Studio* interoperates with general purpose code-editing and vector graphics editing programs; this functionality can be extended to support other editing environments as well.

In many cases authors want to include their own domain-specific graphics in interactive articles. The most common way of implementing these custom components is through unstructured editing of JavaScript source code, typically in conjunction with a user interface library like React or D3. To support this workflow in *Idyll Studio*, we provide two commands that can be applied to components: *edit* and *duplicate*. To edit a component, the author simply needs to click the *edit* button corresponding to the component that they want to edit; this will cause the author's system-defined text editor (A) to open with the relevant source file loaded, as well as launch a daemon that listens for changes to the source file and reloads the component inside of *Idyll Studio* as changes are made. This allows users direct access to edit their component's source code without needing to worry about details such as launching a local development server or running a JavaScript bundler to prepare the code for use in a web browser. Each time the source file is saved, the component will instantaneously update in the article viewer, allowing authors to rapidly iterate code changes. If an author wishes to edit a component provided by the Idyll standard library, they must first *duplicate* it, and then can freely edit a copy while retaining access to the original implementation.

This interoperability approach is not limited to the modification of component source code, but can be used to streamline the inclusion of other rich media as well (such as vector or raster images) and can be extended to work with any asset that *Idyll Studio* loads from the file system. Figure 3 shows how an author can add an SVG component to their article, and then similarly call out to a system-defined SVG editor (B): first, a user clicks an *edit* button that appears underneath the SVG, calling up a system editor based on file-type, and then *Idyll Studio* instantiates a file-system watcher and reloads the file whenever it changes on disk. As the author manipulates the SVG in their preferred editing interface, the view of the SVG in *Idyll Studio* updates automatically to show how the changes look in context.

4 EVALUATION: FIRST-USE STUDY

We conducted a first-use study with 18 participants with widely varying technical expertise, ranging from writers with little or no programming experience to professional programmers. Participants

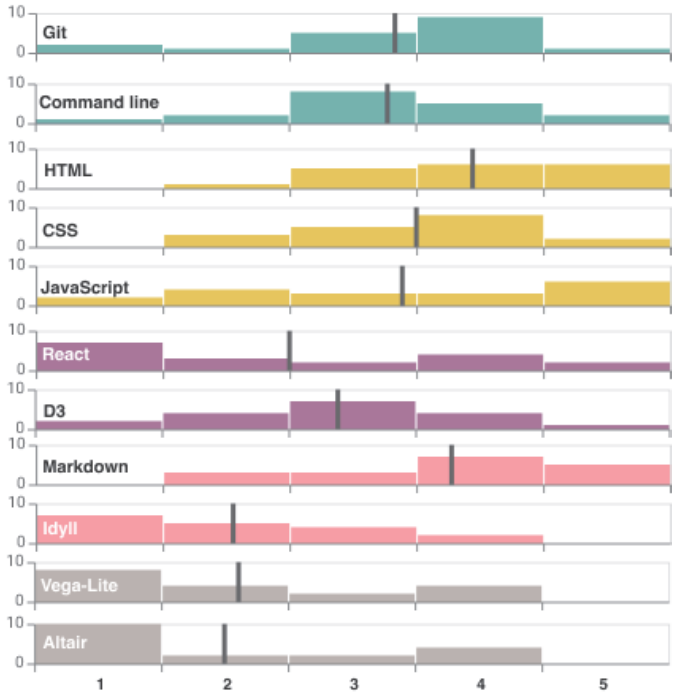

Figure 4: The distribution of self-reported technical expertise of our user study participants by category: developer tools, web technologies, JavaScript libraries, markup languages, and visualization grammars. Overlaid vertical lines convey the mean value.

completed a short survey describing their familiarity with various technologies, used *Idyll Studio* to complete a series of tasks, and then participated in a semi-structured exit interview.

4.1 Methods

To find participants for the study, we advertised it on a social media account associated with the Idyll project, a support chatroom for Idyll users, as well as a public Slack channel for data journalists. We also invited respondents to share the sign-up link with interested colleagues. Each participant completed a short survey in which they rated their familiarity with various technologies on a 1-5 scale, and listed their responsibilities at work and on personal projects.

We ultimately conducted user studies with 18 people. This cohort worked on a variety of relevant tasks (writing, editing, designing, programming), lived across several continents (North America, Europe, and Asia) and used all three major operating systems (Windows, macOS, and Linux). We asked respondents to report their expertise on 11 technologies commonly used in web development, data science, and the construction of data-driven articles on a scale from 1 to 5. Responses are shown in Figure 4. Our study participants have a broad range of familiarity with these technologies, from almost no familiarity to expert levels.

All study sessions were performed remotely over video chat. Each participant was given a link that they could use to download a development version of the *Idyll Studio* desktop application. Participants were given a 10 minute tutorial on how to use the basic functionality, and then were given 30 minutes to complete a series of tasks. The tasks were divided into two categories (*interaction* and *data-driven*) with a basic, intermediate, and advanced task within each category. Participants were given the tasks in order of difficulty, starting with the basic interactive task and ending with the advanced data-driven task (if time allowed) and asked to "think aloud" [23] while they worked through them. After completing the tasks, we performed a short semi-structured interview with each candidate, discussing their experience using *Idyll Studio*, their experience completing similar projects in the past, and if they would consider using *Idyll Studio* in the future.

4.1.1 Interaction Tasks. The three interaction tasks represented common patterns that occur in interactive articles, starting off with a simple example where the participant needed to create a variable and then bind the value of the variable to a *Range Slider* (two-way binding) as well as to a *Display* component (one-way binding), so that when the range slider was manipulated, the display component would immediately show the new value of the variable. The next task was inspired by Victor's *Explorable Explanations* [73] and required that participants build a "dog years calculator," in which text would dynamically update to show the age of a dog in both human-years and dog-years in response to user input. Finally, participants were required to implement *StretchText* [51], where a paragraph of text expands or contracts to display text containing different levels of detail in response to reader input. The *StretchText* task was chosen as it requires that users drop down into unstructured editing of Idyll markup.

4.1.2 Data-Driven Tasks. To complete the data-driven tasks, participants were sent a link to a tabular data file containing information about penguins in several quantitative and categorical columns. The participants were first asked to load this data into the system, and then add a *Table* component to the page that displayed the rows of this dataset. After this, they were asked to construct a simple scatter plot using Vega-Lite [63], showing their choice of quantitative fields on the x and y axes. Participants were then asked to modify this scatter plot to make it interactive, allowing readers to choose which field appeared on the y-axis of the chart. To do this, participants needed to create a new reactive variable, bind it to an input widget, such as a radio button selection, and update the Vega-Lite specification to reference this reactive variable.

4.2 Quantitative Results

The results of the user study are shown in Table 1. All of the participants were able to complete both of the basic tasks and at least one of the intermediate tasks: only one participant didn't complete both intermediate tasks. The completion rate of the advanced tasks was lower, with 13 out of the 18 participants completing both of those tasks. The "Tech." column in Table 1 shows the average value of the participant's self reported score across the 11 different technologies that were asked about in our pre-study survey, which serves as a proxy for a participant's overall comfort with programming and technical tools in general. The table is sorted from highest to lowest *tech* score and shows that a participant's prior familiarity with web

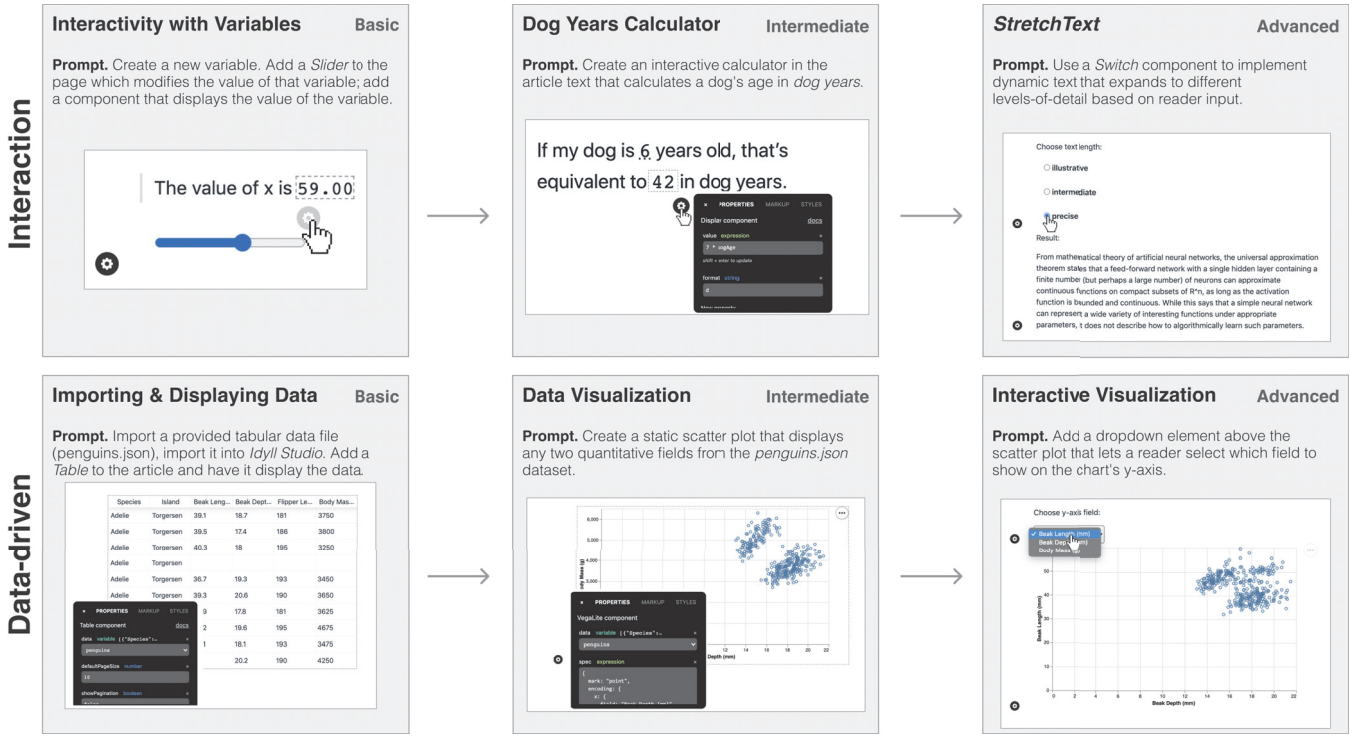

Figure 5: Our first-use study asked participants to complete a series of tasks that involved adding interactivity and data-driven elements to their articles. The tasks were chosen to represent common design patterns found in interactive articles.

development or data science tools was highly predictive of their performance during this user study.

This may indicate that the interface is still not intuitive enough for non-technical users, or that these users needed more time to familiarize themselves with the concepts involved in creating interactive articles. The participants had only 10 minutes of tutorial and 30 minutes to complete all of the tasks. It is encouraging that even the less technical participants were able to complete the intermediate tasks because it suggests that they have an understanding of the basic ideas needed to complete these tasks, but may need more time to synthesize how to compose them in the interface. For example, the advanced interactive data visualization task built directly on concepts that were needed to complete the previous tasks (defining variables, connecting variables to input widgets, referencing variables in expressions, and using Vega-Lite specifications). Given that participants were comfortable using these ideas in isolation, these more advanced tasks may be learnable for these users given more time with the system, and are likely in their zone of proximal development [75].

4.3 Qualitative Results

Immediately after spending up to 30 minutes completing the tasks described, we conducted semi-structured interviews where we asked participants to discuss their experience with the tool. The interviews ranged in length from 15 minutes to about an hour. We prepared three areas of questions for each participant: *What did you find confusing or counterintuitive using the tool? Describe the last time you worked on a project involving the communication of data. What tools did you use? What was the workflow? Could you see this tool fitting into your personal workflow in the future? Why or why not? What about in a collaborative workflow?* We engaged in open discussion with participants based on their responses to elicit more feedback, better understand their needs, and ask how the system did (or did not) work for them. We used an open coding procedure to analyze the interviews and synthesized the resulting codes into several major themes.

1. The interface enabled users to rapidly create designs with less stress. Despite some users not completing all the tests, the general sentiment was positive and participants expressed that they valued the ease-of-use of the system regardless of their technical abilities: "What I really like is to drag and drop [components] and then wire them up with point and click" (P15), "It's fully something I would use - I don't have the time or energy to code from scratch" (P17), "Adding the interactivity and putting it into text - if you wanted to do it in another way you could but it's more work and more stressful" (P9), "Really powerful and easy to use. I feel empowered" (P5). The interface was frequently described as intuitive and participants appreciated that the ability to publish to the web after they had completed an article: "You've smoothed so many of the tough things away from what I usually do. Publishing is a pain in the butt and hey — it's just done!" (P3).

2. But some users struggled to synthesize multiple concepts. Others needed time to understand the reactive model. While nearly all

Table 1: We conducted a first-use study with 18 participants. The *tech* column represents a participant's average familiarity with various related technologies; the subsequent columns each represent a task: interactive (*I*) or data-oriented (*D*), and basic (*1*), intermediate (*2*), or advanced (*3*).

	Role	Tech.	I1	D1	I2	D2	I3	D3
P1	comp. biologist	4.2	y	y	y	y	y	y
P2	programmer	4.2	y	y	y	y	y	y
P3	hci researcher	3.8	y	y	y	y	y	y
P4	vis. practitioner	3.7	y	y	y	y	y	y
P5	hci researcher	3.5	y	y	y	y	y	y
P6	journalist	3.4	y	y	y	y	y	y
P7	hci researcher	3.4	y	y	y	y	y	y
P8	lab manager	3.4	y	y	y	y	y	y
P9	programmer	3.2	y	y	y	y	y	y
P10	vis. practitioner	2.9	y	y	y	y	y	y
P11	editor	2.9	y	y	y	y	y	y
P12	cs/journ. student	2.5	y	y	y	y	y	y
P13	data scientist	2.5	y	y	y	y	n	n
P14	analyst	2.3	y	y	y	y	n	n
P15	designer	2.3	y	y	y	y	n	n
P16	writer	1.8	y	y	y	n	n	n
P17	ux researcher	1.8	y	y	y	y	y	y
P18	designer	1.3	y	y	y	y	n	n

participants completed both basic and intermediate tasks, five less-technical users failed to complete any of the advanced tasks in the allotted time. Difficulties typically occurred when users needed to perform unstructured editing of an expression or manipulate a component's property type. All users were comfortable writing simple one-line expressions, even ones that referenced variables (for example, an expression like `dogAge * 7` was needed to complete the dog years calculator). However, when needing to add a reference to a variable in a more complex expression, as in a Vega-Lite specification, some users struggled and felt overwhelmed: "I didn't feel comfortable or grounded in this part of the test. I just felt like I was trying different things and seeing how the graph would respond" (P18). Participants expressed that they needed some time to internalize how the interface worked: "Everything logically made sense but it took some time to think through things" (P17), "I needed some time to clarify" (P8). For some this did not last long ("I came in with a slightly wrong preconception, but that was all of 30 seconds or so" (P3)), but others needed more time ("I would have to do it a few times to get that to work" (P15)).

3. The interface should provide more visual feedback, promote experimentation, provide documentation on demand, and have guardrails to prevent likely mistakes. To make the interface more forgiving and promote learning, the interface should do more to prevent mistakes and guide users in the right direction. One designer without programming experience described a desire for more visual feedback: "Since I'm so unskilled, all the visual cues I could get would be appreciated. I was just operating on a really crude understanding of algebra and how things mapped together. I would definitely take all the hand holding I could get in the UI" (P18). The different types that properties could have were not adequately explained or discoverable in the interface: "[The types] could be more intuitive. It's not obvious what they mean or how to navigate them" (P11). The interface shined when it promoted experimentation ("In *Idyll Studio* everything seemed to work very nicely. You could fiddle around and get things working quite easily" (P14)) but sometimes users would unintentionally overwrite an important part of their document, for instance the dataset that they had loaded (P6, P15).

4. The interface empowered users and provided better support to less technical users compared to existing tools. Despite the issues, participants found value in what the interface allowed them to do compared to existing tools. Some participants could not replicate what they had done in the tasks with tools they currently use: "I would paste a screenshot of a graph into a deck" (P18), "Sometimes I use PowerPoint, but it is limiting. I can write a little HTML too but that is also limiting" (P18). Others found existing tools frustrating: "I don't have a good workflow. I've tried other tools and just gave up. I have used Observable notebooks—it's a bit confusing, I get overwhelmed using their stuff" (P17). Despite the learning curve of *Idyll Studio*, users found the drag-and-drop interface and structured editing easy to work with, and were excited by its potential: "Once I took the time to learn it would be useful" (P18), "It's not a completely entry level tool—at least there are uses that are not entry level. For people who develop some facility it can be very powerful" (P8).

5. Participants valued ownership of code, data, and styles, and the tool fit well with existing workflows. Several of our participants noted that they appreciated that the application was distributed as a desktop application so they could use it with private data files and without relying on third-party infrastructure (P4, P7, P8). The ability to take the generated source files and send them to other tools was also important. Participants—especially journalists and researchers who frequently published their work online—wanted the ability to host their articles wherever they wanted and have complete control over the look-and-feel (P1, P3, P6, P11, P15). The fact that *Idyll Studio* could be used to customize styles and layouts, and produced web-standard HTML, CSS, and JS files that could easily be accessed, were crucial features for these users. As we expected, most users treat the tasks of exploratory analysis and narrative visualization separately, and when creating interactive articles typically start with clean JSON or CSV files to be used in their article (P3, P6, P7, P9, P10, P12, P13). However, there were some requests for more data transformation utilities; several participants wanted to have access to a data-frame API (P2, P3, P8) to make it easier to access and filter data field names in expressions.

5 EVALUATION: EXPRESSIVENESS

While the first-use study is a good indication that the editor can be used to specify small sections of self-contained interactivity, it is also important to understand how the interface scales to more complex articles. Here we describe how *Idyll Studio* can be used to implement common *Scroller* and *Stepper* patterns [46] (Figure 6), and to reimplement an existing explorable explanation (Figure 7). These examples are available online.[4]

[4] *Scroller/Stepper*:
https://idyll.pub/post/scroller-stepper-example-81e37de0fde8487fdb64c378/
Barnes-Hut:
https://idyll.pub/post/barnes-hut-re-implement-e0587578480b3cc37f89ec62/

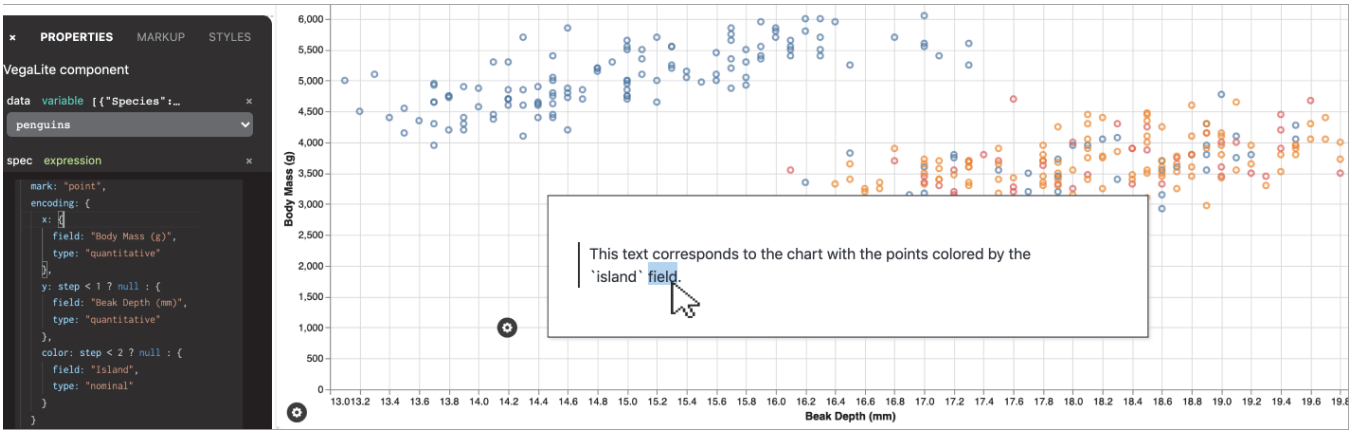

Figure 6: Specification of a *Scroller* layout, which displays a sequence of text blocks over a visualization that updates as a reader scrolls. Text is edited in place and the chart's specification is parameterized with a reactive variable that tracks scroll depth.

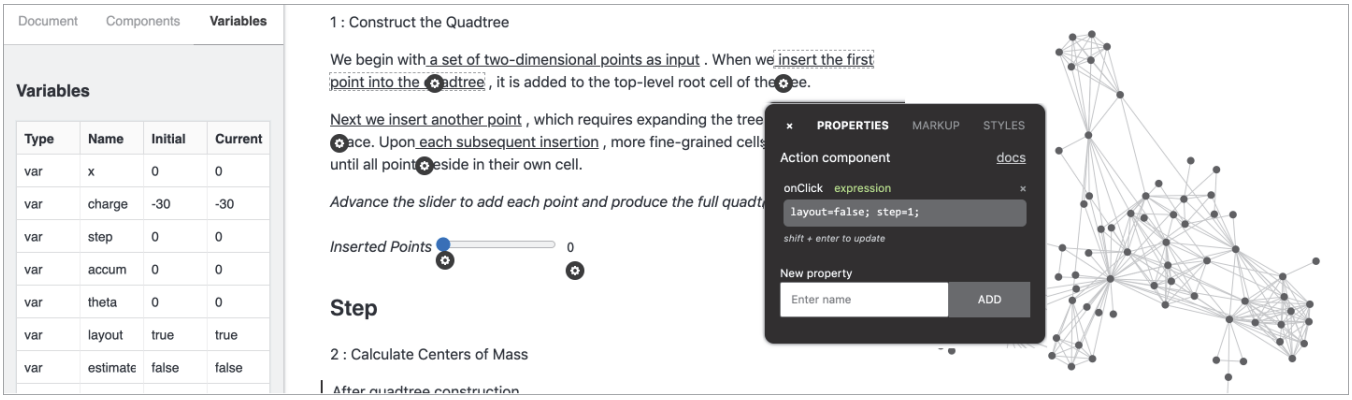

Figure 7: A view of *The Barnes-Hut Approximation*, an existing explorable explanation reimplemented using *Idyll Studio*. A graphic stays fixed to the right side of the page as readers scroll, and responds to links and input widgets placed in the text.

5.1 Scroller & Stepper

A common design pattern in narrative visualization is to animate through a series of visualizations, each of which has a snippet of corresponding text. This can be presented as a slideshow requiring the user to click through each step, or triggered via a scroll interaction. To implement the scroll version, an author would first drag the *Scroller* from the components tab of the left-hand panel onto the document. This immediately adds an example to the screen with sections of text that scroll over a chart in the background. The chart specification can be parameterized using the scroll position by creating a new reactive variable and binding it to the Scroller's 'currentStep' or 'currentPosition' property, then using that variable in the chart specification. The text associated with each step is directly editable via the *Text Editor* instrument. To add steps, they can be dragged onto the article from the components panel. A *Stepper* design can be implemented in a similar way: the component exposes a similar API to bind a reactive variable, and existing text can be edited in place, although new steps must be added by unstructured editing of the Stepper component's Idyll specification.

5.2 *The Barnes-Hut Approximation*

The Barnes-Hut Approximation is an existing Idyll document that presents an explorable explanation of an algorithm used to perform force-directed graph layout calculations. The article consists of a fixed graphic on the right-hand side with text that scrolls by on the left. As the reader progresses, the graphic updates in response, and parameters can be manipulated through controls embedded directly in the text. We downloaded the article's graph visualization component and reimplemented the layout, text, and interactions of the article through the *Idyll Studio* interface using direct manipulation, structured editing, and simple expressions to handle input (typically things like 'layout = false; step = 1;' in a click handler) and create links between components and sections of text. To recreate the layout, we used the drag-and-drop interface to add a *Fixed* component to the right margin and embed the graph visualization inside of it. We then used the *Action*, *Display*, *Range*, and other components to link sections of the text to specific states in the graph, and elicit user input to specify algorithm parameters. The exercise shows that the user interface can scale to support more complex narrative visualizations, though there were some pain points: the component

gear icons may need to be positioned in a more structured way when there are many components on the screen to avoid overlap; we also discovered some minor bugs in our markup serialization implementation. It can be difficult to edit a side-by-side design on a small display—in general, we have not optimized *Idyll Studio* for screens smaller than a 13-inch laptop.

6 CONCLUSION AND FUTURE WORK

We presented *Idyll Studio*, an expressive WYSIWYG-style editor for authoring interactive articles. We conducted a first-use study in which 18 participants were asked to perform rapid prototyping tasks and validated the utility of this type of GUI environment for a broad range of users to author interactive articles. Our system helped both technical experts and non-technical users create interactive articles, making the process less stressful and lowering the barrier to entry by reducing the complexity of the task and limiting the need to use general purpose programming tools. The participants appreciated being able to use familiar *point*, *click*, and *drag* interactions to author interactive articles without needing to learn a new language or memorize a library of components. We found that the system fit well with users' existing workflows and supported them better than existing tools, although there was a learning curve for less technical users. We also demonstrated the expressivity of the system by implementing design patterns commonly found in interactive articles, and by using it to recreate an existing explorable explanation.

Idyll Studio is a structured editing environment for authoring interactive and data-driven articles that reifies the four parts of a reactive document model (text, components, state, and styles). In order to create this system, we extended Idyll to support reflective documents and developed an AST expansion algorithm that allows us to augment any Idyll document with instrumental components, allowing for structured editing of the document using interface elements that reside directly in the document itself. This functionality is accessible via a flexible API which powers *Idyll Studio*. While general AST transformation is a mainstay of programming languages, to our knowledge it has not previously been utilized for reflective WYSIWYG interface design tools. The approach is conceptually simple, powerful, and generalizable, and so could be adopted by future structured design tools.

There is still much work to be done in this area. Idyll Studio provides an open-source platform for implementing and testing new techniques, and other researchers may use Idyll Studio as a starting point for further extension (e.g., support collaborative authoring, provide guidance to authors, flag likely mistakes, etc.) or integrate it with new research systems or component authoring tools. Despite the positive results of our user study and our findings that the system lowers the threshold for less technically inclined users, the system could still do more to limit the amount of unstructured text editing that needs to be done and make specific concepts of the interactive document model (for example, component property types) more intuitive. Certain interactions can not easily be specified in our tool without writing markup. Identifying patterns that allow users to author such interactions entirely via point-and-click is an interesting area of future research that would continue to benefit interactive article authors.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers, the user study participants, and all of the open source contributors for their generous support of this research.

REFERENCES

[1] Fereshteh Amini, Nathalie Henry Riche, Bongshin Lee, Christophe Hurter, and Pourang Irani. 2015. Understanding data videos: Looking at narrative visualization through the cinematography lens. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. 1459–1468.

[2] Anders Andersen. 1998. A note on reflection in Python 1.5. *Lancaster University* 46 (1998).

[3] B Atkinson. 1987. HyperCard [computer program]. *Cupertino, CA: Apple Computer* (1987).

[4] Benjamin Bach, Natalie Kerracher, Kyle Wm Hall, Sheelagh Carpendale, Jessie Kennedy, and Nathalie Henry Riche. 2016. Telling stories about dynamic networks with graph comics. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. 3670–3682.

[5] Benjamin Bach, Zezhong Wang, Matteo Farinella, Dave Murray-Rust, and Nathalie Henry Riche. 2018. Design patterns for data comics. In *Proceedings of the 2018 chi conference on human factors in computing systems*. 1–12.

[6] Sriram Karthik Badam, Andreas Mathisen, Roman Rädle, Clemens N Klokmose, and Niklas Elmqvist. 2018. Vistrates: A component model for ubiquitous analytics. *IEEE transactions on visualization and computer graphics* 25, 1 (2018), 586–596.

[7] Michel Beaudouin-Lafon. 2000. Instrumental interaction: an interaction model for designing post-WIMP user interfaces. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. 446–453.

[8] Michel Beaudouin-Lafon and Wendy E Mackay. 2000. Reification, polymorphism and reuse: three principles for designing visual interfaces. In *Proceedings of the working conference on Advanced visual interfaces*. 102–109.

[9] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. 2011. D3: Data-Driven Documents. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)* (2011). http://idl.cs.washington.edu/papers/d3

[10] Judd D Bradbury and Rosanna E Guadagno. 2020. Documentary narrative visualization: Features and modes of documentary film in narrative visualization. *Information Visualization* 19, 4 (2020), 339–352.

[11] Bill Burdick. 2011. leisure. https://github.com/zot/Leisure.

[12] Jordi Cabot. 2018. WordPress: A content management system to democratize publishing. *IEEE Software* 35, 3 (2018), 89–92.

[13] Matthew Conlen. 2017. Using Apparatus with Idyll. Retrieved March 1, 2021 from https://mathisonian.com/writing/apparatus

[14] Matthew Conlen and Jeffrey Heer. 2018. Idyll: A markup language for authoring and publishing interactive articles on the web. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*. 977–989.

[15] Matthew Conlen and Fred Hohman. 2018. The Beginner's Guide to Dimensionality Reduction. *Workshop on Visualization for AI Explainability (VISxAI) at IEEE VIS* (2018). https://idyll.pub/post/dimensionality-reduction-293e465c2a3443e8941b016d/

[16] Matthew Conlen and Fred Hohman. 2019. Launching the parametric press. (2019).

[17] Matthew Conlen, Alex Kale, and Jeffrey Heer. 2019. Capture & analysis of active reading behaviors for interactive articles on the web. In *Computer Graphics Forum*, Vol. 38. Wiley Online Library, 687–698.

[18] Morgan Dixon and James Fogarty. 2010. Prefab: Implementing Advanced Behaviors Using Pixel-Based Reverse Engineering of Interface Structure. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Atlanta, Georgia, USA) *(CHI '10)*. Association for Computing Machinery, New York, NY, USA, 1525–1534. https://doi.org/10.1145/1753326.1753554

[19] Morgan Dixon, James Fogarty, and Jacob Wobbrock. 2012. A General-Purpose Target-Aware Pointing Enhancement Using Pixel-Level Analysis of Graphical Interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Austin, Texas, USA) *(CHI '12)*. Association for Computing Machinery, New York, NY, USA, 3167–3176. https://doi.org/10.1145/2207676.2208734

[20] Pierre Dragicevic, Yvonne Jansen, Abhraneel Sarma, Matthew Kay, and Fanny Chevalier. 2019. Increasing the transparency of research papers with explorable multiverse analyses. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. 1–15.

[21] James R Eagan, Michel Beaudouin-Lafon, and Wendy E Mackay. 2011. Cracking the cocoa nut: user interface programming at runtime. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*. 225–234.

[22] Leah Findlater, Alex Jansen, Kristen Shinohara, Morgan Dixon, Peter Kamb, Joshua Rakita, and Jacob O Wobbrock. 2010. Enhanced area cursors: reducing fine pointing demands for people with motor impairments. In *Proceedings of the 23nd annual ACM symposium on User interface software and technology*. 153–162.

[23] Marsha E Fonteyn, Benjamin Kuipers, and Susan J Grobe. 1993. A description of think aloud method and protocol analysis. *Qualitative health research* 3, 4 (1993), 430–441.

[24] Ira R Forman, Nate Forman, and John Vlissides Ibm. 2004. Java reflection in action. (2004).

[25] Krzysztof Z Gajos, Daniel S Weld, and Jacob O Wobbrock. 2010. Automatically generating personalized user interfaces with Supple. *Artificial Intelligence* 174, 12-13 (2010), 910–950.

[26] Nahum Gershon and Ward Page. 2001. What storytelling can do for information visualization. *Commun. ACM* 44, 8 (2001), 31–37.

[27] Mohamed G Gouda and Ted Herman. 1991. Adaptive programming. *IEEE Transactions on Software Engineering* 17, 9 (1991), 911–921.

[28] Esther Greussing and Hajo G Boomgaarden. 2019. Simply bells and whistles? Cognitive effects of visual aesthetics in digital longforms. *Digital Journalism* 7, 2 (2019), 273–293.

[29] Brian Hempel and Ravi Chugh. 2016. Semi-automated svg programming via direct manipulation. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*. 379–390.

[30] Brian Hempel, Justin Lubin, Grace Lu, and Ravi Chugh. 2018. Deuce: a lightweight user interface for structured editing. In *Proceedings of the 40th International Conference on Software Engineering*. 654–664.

[31] Fred Hohman, Matthew Conlen, Jeffrey Heer, and Duen Horng Polo Chau. 2020. Communicating with interactive articles. *Distill* 5, 9 (2020), e28.

[32] Jessica Hullman and Nick Diakopoulos. 2011. Visualization rhetoric: Framing effects in narrative visualization. *IEEE transactions on visualization and computer graphics* 17, 12 (2011), 2231–2240.

[33] Jessica Hullman, Steven Drucker, Nathalie Henry Riche, Bongshin Lee, Danyel Fisher, and Eytan Adar. 2013. A deeper understanding of sequence in narrative visualization. *IEEE Transactions on visualization and computer graphics* 19, 12 (2013), 2406–2415.

[34] Rubaiat Habib Kazi, Fanny Chevalier, Tovi Grossman, and George Fitzmaurice. 2014. Kitty: Sketching Dynamic and Interactive Illustrations. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology* (Honolulu, Hawaii, USA) *(UIST '14)*. Association for Computing Machinery, New York, NY, USA, 395–405. https://doi.org/10.1145/2642918.2647375

[35] Holger M Kienle. 2010. It's about time to take JavaScript (more) seriously. *IEEE software* 27, 3 (2010), 60–62.

[36] Yea-Seul Kim, Katharina Reinecke, and Jessica Hullman. 2017. Explaining the gap: Visualizing one's predictions improves recall and comprehension of data. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. 1375–1386.

[37] Matthew G Kirschenbaum. 2016. *Track changes: A literary history of word processing*. Harvard University Press.

[38] Clemens N Klokmose, James R Eagan, Siemen Baader, Wendy Mackay, and Michel Beaudouin-Lafon. 2015. Webstrates: Shareable dynamic media. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*. 280–290.

[39] Donald Ervin Knuth. 1984. Literate programming. *Comput. J.* 27, 2 (1984), 97–111.

[40] Amy Ko, Htet Htet Aung, and Brad A Myers. 2005. Design requirements for more flexible structured editors from a study of programmers' text editing. In *CHI'05 extended abstracts on human factors in computing systems*. 1557–1560.

[41] Amy Ko and Brad A Myers. 2006. Barista: An implementation framework for enabling new tools, interaction techniques and views in code editors. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*. 387–396.

[42] Sam Lau and Philip J Guo. 2020. Data Theater: A Live Programming Environment for Prototyping Data-Driven Explorable Explanations. In *Workshop on Live Programming (LIVE)*.

[43] Bongshin Lee, Nathalie Henry Riche, Petra Isenberg, and Sheelagh Carpendale. 2015. More than telling a story: Transforming data into visually shared stories. *IEEE computer graphics and applications* 35, 5 (2015), 84–90.

[44] Pattie Maes. 1987. Concepts and experiments in computational reflection. *ACM Sigplan Notices* 22, 12 (1987), 147–155.

[45] Richard E Mayer. 2002. Multimedia learning. In *Psychology of learning and motivation*. Vol. 41. Elsevier, 85–139.

[46] Sean McKenna, N Henry Riche, Bongshin Lee, Jeremy Boy, and Miriah Meyer. 2017. Visual narrative flow: Exploring factors shaping data visualization story reading experiences. In *Computer Graphics Forum*, Vol. 36. Wiley Online Library, 377–387.

[47] Philip K McKinley, Seyed Masoud Sadjadi, Eric P Kasten, and Betty HC Cheng. 2004. Composing adaptive software. *Computer* 37, 7 (2004), 56–64.

[48] Graham McNeill and S Hale. 2019. Viz-Blocks: Building Visualizations and Documents in the Browser. (2019).

[49] Jens Monig, Yoshiki Ohshima, and John Maloney. 2015. Blocks at your fingertips: Blurring the line between blocks and text in GP. In *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*. IEEE, 51–53.

[50] Brad Myers, Scott E Hudson, and Randy Pausch. 2000. Past, present, and future of user interface software tools. *ACM Transactions on Computer-Human Interaction (TOCHI)* 7, 1 (2000), 3–28.

[51] Ted Nelson. 1967. Stretchtext – hypertext note #8. *Project Xanadu* (1967).

[52] Observable 2018. Observable. https://observablehq.com/.

[53] Yoshiki Ohshima, Aran Lunzer, Bert Freudenberg, and Ted Kaehler. 2013. KScript and KSWorld: A time-aware and mostly declarative language and interactive GUI framework. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*. 117–134.

[54] Dan R. Olsen, Scott E. Hudson, Thom Verratti, Jeremy M. Heiner, and Matt Phelps. 1999. Implementing Interface Attachments Based on Surface Representations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Pittsburgh, Pennsylvania, USA) *(CHI '99)*. Association for Computing Machinery, New York, NY, USA, 191–198. https://doi.org/10.1145/302979.303038

[55] Fernando Pérez and Brian E Granger. 2007. IPython: a system for interactive scientific computing. *Computing in science & engineering* 9, 3 (2007), 21–29.

[56] Jeffrey M Perkel. 2018. Why Jupyter is data scientists' computational notebook of choice. *Nature* 563, 7732 (2018), 145–147.

[57] Vincent Quint and Irene Vatton. 1986. Grif: An interactive system for structured document manipulation. In *Text Processing and Document Manipulation, Proceedings of the International Conference*. 200–312.

[58] Roman Rädle, Midas Nouwens, Kristian Antonsen, James R Eagan, and Clemens N Klokmose. 2017. Codestrates: Literate computing with webstrates. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*. 715–725.

[59] Casey Reas and Ben Fry. 2007. *Processing: a programming handbook for visual designers and artists*. Mit Press.

[60] Adam Rule, Aurélien Tabard, and James D Hollan. 2018. Exploration and explanation in computational notebooks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. 1–12.

[61] Arvind Satyanarayan and Jeffrey Heer. 2014. Authoring narrative visualizations with ellipsis. In *Computer Graphics Forum*, Vol. 33. Wiley Online Library, 361–370.

[62] Arvind Satyanarayan and Jeffrey Heer. 2014. Lyra: An interactive visualization design environment. In *Computer Graphics Forum*, Vol. 33. Wiley Online Library, 351–360.

[63] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. 2017. Vega-Lite: A Grammar of Interactive Graphics. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)* (2017). http://idl.cs.washington.edu/papers/vega-lite

[64] Arvind Satyanarayan, Ryan Russell, Jane Hoffswell, and Jeffrey Heer. 2016. Reactive Vega: A Streaming Dataflow Architecture for Declarative Interactive Visualization. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)* (2016). http://idl.cs.washington.edu/papers/reactive-vega-architecture

[65] Toby Schachman and Joshua Horowitz. 2016. Apparatus. https://github.com/cdglabs.

[66] Pascal Schneiders. 2020. What remains in mind? Effectiveness and efficiency of explainers at conveying information. *Media and Communication* 8, 1 (2020), 218–231.

[67] Edward Segel and Jeffrey Heer. 2010. Narrative visualization: Telling stories with data. *IEEE transactions on visualization and computer graphics* 16, 6 (2010), 1139–1148.

[68] Brian Smith. 1982. Reflection and semantics in a procedural language. *Technical Report, TR-272, MIT Laboratory for Computer Science* (1982).

[69] Charles D Stolper, Bongshin Lee, N Henry Riche, and John Stasko. 2016. Emerging and recurring data-driven storytelling techniques: Analysis of a curated collection of recent stories. *Microsoft Research, Washington, USA* (2016).

[70] John Sweller. 2011. Cognitive load theory. In *Psychology of learning and motivation*. Vol. 55. Elsevier, 37–76.

[71] Tim Teitelbaum and Thomas Reps. 1981. The Cornell program synthesizer: a syntax-directed programming environment. *Commun. ACM* 24, 9 (1981), 563–573.

[72] Andries Van Dam. 1997. Post-WIMP user interfaces. *Commun. ACM* 40, 2 (1997), 63–67.

[73] Bret Victor. 2011. Explorable Explorations. Retrieved March 1, 2021 from http://worrydream.com/ExplorableExplanations/

[74] Bret Victor. 2011. Tangle: a JavaScript library for reactive documents. Retrieved August 1, 2017 from http://worrydream.com/Tangle/

[75] Lev Semenovich Vygotsky. 1980. *Mind in society: The development of higher psychological processes*. Harvard university press.

[76] April Yi Wang, Anant Mittal, Christopher Brooks, and Steve Oney. 2019. How data scientists use computational notebooks for real-time collaboration. *Proceedings of the ACM on Human-Computer Interaction* 3, CSCW (2019), 1–30.

[77] Qiyu Zhi, Alvitta Ottley, and Ronald Metoyer. 2019. Linking and layout: Exploring the integration of text and visualization in storytelling. In *Computer Graphics Forum*, Vol. 38. Wiley Online Library, 675–685.

[78] Jonathan Zong, Dhiraj Barnwal, Rupayan Neogy, and Arvind Satyanarayan. 2020. Lyra 2: Designing interactive visualizations by demonstration. *IEEE Transactions on Visualization and Computer Graphics* (2020).

[79] Douglas Zongker. 2003. *Creating animation for presentations*. Ph.D. Dissertation. University of Washington.