Idyll: A Markup Language for Authoring and Publishing Interactive Articles on the Web

Matthew Conlen
University of Washington
Seattle, WA
mconlen@cs.washington.edu

Jeffrey Heer
University of Washington
Seattle, WA
jheer@cs.washington.edu

ABSTRACT

The web has matured as a publishing platform: news outlets regularly publish rich, interactive stories while technical writers use animation and interaction to communicate complex ideas. This style of interactive media has the potential to engage a large audience and more clearly explain concepts, but is expensive and time consuming to produce. Drawing on industry experience and interviews with domain experts, we contribute design tools to make it easier to author and publish interactive articles. We introduce Idyll, a novel "compile-to-the-web" language for web-based interactive narratives. Idyll implements a flexible article model, allowing authors control over document style and layout, reader-driven events (such as button clicks and scroll triggers), and a structured interface to JavaScript components. Through both examples and first-use results from undergraduate computer science students, we show how Idyll reduces the amount of effort and custom code required to create interactive articles.

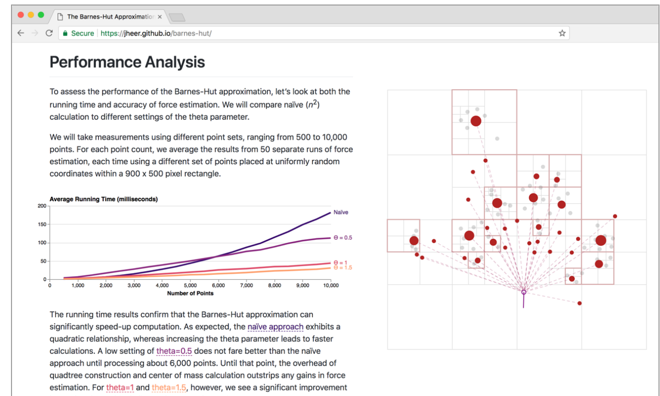

Figure 1. An Idyll article that interactively links text with visual demonstrations to explain the Barnes-Hut approximation for simulating physical forces. The visualization on the right keeps a fixed position, updating as the user progresses through the article and interacts with the text.

Author Keywords

Artifact or System; Prototyping/Implementation; Interaction Design; Programming/Development Support; Storytelling; Visualization; Programming Languages;

INTRODUCTION

Publications like the *New York Times*, the *Washington Post*, the *Guardian*, and *FiveThirtyEight* are known for producing high-quality multimedia narratives. Often referred to as *interactives*, these stories have the potential to engage a large audience: in 2013 the most read story in the *New York Times* was an interactive quiz; in 2014, ten of their forty most read stories were from the *Upshot*, a section known for publishing rich data-driven stories. The data visualization research community has attempted to characterize the techniques used in these articles, and have suggested research opportunities in creating tools to drive the production of interactive narratives.

High-quality production pieces are expensive and time consuming to produce. They require custom code, which is often developed by non-expert programmers under the stress of a deadline. Because of this, the resulting web pages can suffer from performance issues (such as being slow to load, especially on mobile browsers), and the code itself may be poorly structured and hard to reuse. In order to improve code quality and decrease development time, some newsrooms create internal frameworks that facilitate common tasks [31, 40, 41]. These frameworks, however, require both a large initial effort and maintenance over time, making them inaccessible to all but the most well-staffed organizations.

In this paper, we contribute *Idyll*, a novel domain specific language (DSL) designed for authoring interactive narratives. Idyll targets a balance between expressiveness and readability in its programs, while taking production-quality technical requirements into account. We believe these features make Idyll well-suited to provide authors with a way to quickly create and publish interactive articles in a production environment.

Idyll combines a human-readable markup language with a reactive variable system, and provides a straightforward way to embed JavaScript components in-line with text. Idyll focuses on making it easy to orchestrate the presentation of the article based on variable state. This allows article behavior to be declaratively specified, removing a large class of code that typically needs to be written when custom interactives are developed, including code that ties HTML elements to JavaScript variables, custom event listeners that trigger page updates, and custom data bindings in the view layer.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

UIST '18, October 14–17, 2018, Berlin, Germany

© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-5948-1/18/10. . . $15.00

DOI: https://doi.org/10.1145/3242587.3242600

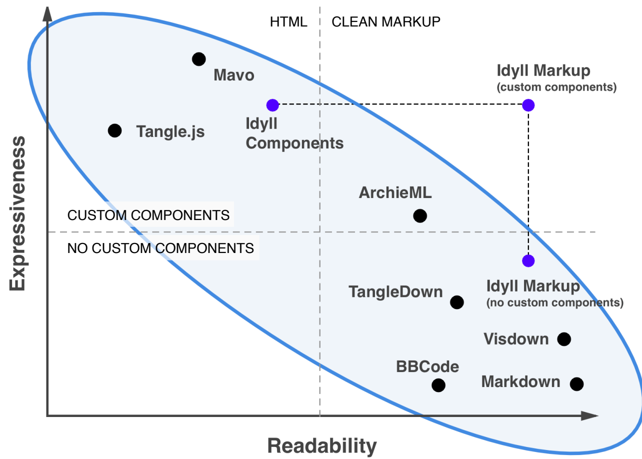

Figure 2. Comparing the readability and expressiveness of web-based publishing tools. Many languages focus on providing a legible authoring experience, or on facilitating the implementation of complex designs. Idyll aims to achieve high readability of its text, while maintaining expressiveness. It does this by imposing a clean, structured interface between text and interactive elements, providing a standard library, and by allowing users to write custom code where necessary.

Idyll includes a standard library of components that can be used to add interactivity to articles without the need to write JavaScript, and we show how these components cover a wide range of use cases common to interactive articles. Idyll exposes high-level language constructs about page state that enable authors to easily parameterize their documents based on scroll events and component properties such as visibility. Idyll aims to be easy for non-technical users to pick up, while still being powerful enough to implement a broad range of designs. We show how, on first use, undergraduate students were able to grasp language concepts and produce high quality interactive articles that leverage a broad range of Idyll features.

Taken together, the human readable markup, reactive variable system, access to page state & events, library of built-in components, and straightforward extensibility are intended to lower the barrier to entry for creating interactive articles, and accelerate their creation by reducing the overall amount of code and setup needed to produce and publish them. The remainder of the paper discusses related work, shares our design process, presents Idyll, and, through a series of examples and a first use study with students, concretely demonstrates how Idyll simplifies the process of creating interactives.

RELATED WORK

Idyll extends previous work through a novel architecture for reactive parameterization of custom components, high-level bindings to page state and events, and by providing the freedom of customization and interoperability necessary to fit the needs of production-quality work.

Figure 2 shows an overview of existing languages and tools for building interactive and data-driven web pages, plotted on the dimensions of *expressiveness* and *readability*. When looking at these dimensions we consider both the threshold and the ceiling [26] that the tools impose for creating interactives. Items below the dotted line on the expressiveness axis have a low ceiling because they do not provide easy ways for users to define custom components, limiting the space of designs that may be produced. Those plotted above that dotted line do allow for users to include arbitrary custom code, and languages that implement features that make it easier to achieve common designs are given higher expressiveness scores. For example, both Tangle and ArchieML are used within standard JavaScript, meaning that the authoring environment is powerful because it has all of JavaScript at its disposal.

The readability axis considers an author writing structured text with each language. Many of the items plotted take advantage of a markup language that reduces the amount of code that an author needs to write to style and add hierarchy to their text. Some of the tools require users to write in HTML or JavaScript directly; these tools are shown as having lower readability. When Idyll is used without custom components, the markup is relatively simple but expressiveness is limited by the set of standard components; users may write custom JavaScript components, which can be arbitrarily complex. A non-technical user can consider Idyll markup without any knowledge of the underlying custom component implementation. From this perspective, the Idyll *markup* will be of a similar complexity regardless of the inclusion of custom components.

Narrative Visualization & Explorable Explainables

The visualization research community has investigated the techniques that are used in data-driven storytelling, looking at stories that have been published by news outlets. In 2010, Segel & Heer [37] articulated a design space of narrative visualization that has influenced the space of narrative techniques that Idyll and other tools attempt to support. A challenge for researchers in this space is that the tools and techniques used by practitioners are constantly shifting. Both Stolper et al. [38] and McKenna et al. [24] refine Segel & Heer's design space, updating it to reflect changes in practice. Researchers have noted several opportunities for work to be done in this space [19, 21], including evaluating the effectiveness of data-driven storytelling techniques, and building tools that help authors use these techniques in their stories. Idyll supports both of these goals, serving as a tool for authoring and publishing data-driven stories, but also serving as a platform for researchers to collect and analyze data about how users interacted with the stories.

The use of interactivity and visualization in writing extends beyond journalism, for example into scientific publishing [23, 45]. Bret Victor has written persuasively in favor of adding certain types of interactivity to writing that would traditionally be presented as static text [43]. His essay *Explorable Explanations* [44] illustrates some of the types of interactions that we seek to enable with Idyll. Victor released a JavaScript library, Tangle [46], that helps users add reactive variables to their documents. TangleDown [9] and Fangle [2] both combine Victor's Tangle library with Markdown syntax. Since then, a larger movement has built around creating explorable explanations, many of which are catalogued online [4].

Notably, Distill [14] is an online machine learning journal with interactive articles and seeks to bridge the gap between

techniques common to news media and academic publishing. Chris Olah [28], an editor of Distill, wrote on the value of using interactivity to explain complex topics, arguing that interactive publishing platforms like Distill could expedite the dissemination of new research ideas. Distill publishes posts that utilize visualizations, animations, and linked parameters in text to explain machine learning research. These posts serve as inspiration for the types of multimedia storytelling we hope to enable with Idyll. Distill also publishes the source code for their articles, which reveals just how much work goes into creating them: individual articles often require several hundred custom code commits, made over a period of months.[1]

Interactive Web Frameworks

Several research systems facilitate end-user creation of data-driven websites. These tools are aimed at a more general use-case than Idyll, but are nonetheless informative when considering programs whose primary *output* are webpages. Gneiss [13] is an application that provides a drag-and-drop interface with which users can create GUI components. These components may draw their value from cells on a spreadsheet which may act as a proxy to an arbitrary REST service. Gneiss lets users pick from a predefined set of UI elements, but does not provide a compelling way for users to embed their own custom elements. Mavo [42] is an abstraction on top of HTML and introduces syntax that allows users to bind data directly to templated HTML. The tool allows users to author full data-driven websites without relying on a database service.

Flapjax [25] is a language that brings functional reactive programming (FRP) [8] to the web. Flapjax showed that event-driven reactivity is a natural fit for web applications. FRP has been shown to be an effective means for enabling expressive declarative languages in the domains of animation [15] and visualization [35] specification, and can be similar applied in this domain. React [3] is a JavaScript library for declaratively building user interfaces, centered around reactive components. Idyll is implemented using the React library, and uses an event-driven reactive parameterization of article specifications.

Other web frameworks utilize a similar component-oriented architecture, for example Polymer [30] is a JavaScript library for building applications using Web Components, a new standard being added to web browsers. These frameworks simplify web developement, however they still require users to navigate complex application code. Idyll intentionally separates JavaScript code and editorial copy, so that non-technical users may still make simple edits to the text of an article without navigating complex code.

Computational Documents

Computational notebook environments such as Jupyter [29] are frequently used by programmers and data scientists to create and share graphics and computations. Observable [27] is a service that hosts computational notebooks written in JavaScript, Like Idyll, Observable leverages reactive semantics. However, Idyll targets active reading experiences, including layout, navigation, and styling, whereas Jupyter and Observable focus on the coding experience using an interface consisting of a linear list of code and output cells.

Codestrates [32] is a literate computing approach built on web technologies that allows for more customization of the user interface than previous approaches. Where Codestrates allows users to build arbitrary interactive web content, Idyll targets a more focused range of design outputs. Other computational document formats include Leisure [12] and Hypercard [7], which also target the creation of more general applications than Idyll.

Creating Visualizations & Interactive Graphics

There has been extensive work on tools that facilitate the production of visualizations and interactive graphics. Idyll is designed to supplement, rather than supplant, these tools. For example, D3 [10] is a JavaScript library with which developers can create expressive data visualizations and Vega-Lite [34] is a high-level grammar of interactive graphics. Idyll considers documents in a more unified and structured manner than D3 or Vega-Lite — an Idyll user can easily embed a visualization created with Vega-Lite or D3 in their article.

Ellipsis [33] provides a DSL and graphical interface for building narrative visualizations. TimelineStoryteller [11] is a tool for building narrative timelines. Ellipsis, TimelineStoryteller, and Idyll all share a concern with parameterizing interactive visualizations. However, Idyll supports the orchestration of entire articles, rather than individual components. Several other tools of graphical interfaces for the design of specialized graphics. For example, Apparatus [36] is an editor for creating interactive diagrams; TextAlive [18] is an integrated design environment for creating kinetic typography videos and, like Idyll, targets a dual audience of technical and non-technical users. The systems for creating graphics and other media are complementary, as, for example, Ellipsis and TimelineStoryteller could in theory be used to create augmented visualization components or timelines to be included in an Idyll article.

Editorial Tools for the Web

There have been many efforts to make it easier to write for the web using a simpler and more concise syntax than HTML. Markdown [16] is a popular markup language designed to be fast to write and easy to read. Idyll borrows syntax from Markdown in order to leverage users' familiarity with the language. Jekyll [1] is a blog engine that allows users to write posts in plain text or Markdown files and deploy them to the web. Visdown [17] extends the idea of compiling Markdown to HTML pages by allowing authors to specify data visualizations in their markup. A user can provide a declarative Vega-Lite [34] specification directly in the Markdown file, and this is used to render a chart in the final markup. In contrast with Idyll, these tools do not provide a mechanism that allows JavaScript to be tightly integrated with the text.

Creating *interactive* documents involves writing custom JavaScript and HTML. It can become difficult to balance the narrative portion of a project with the nitty-gritty details of code. To this end, *The New York Times* developed ArchieML [39], a markup language for editors to work with text that will subsequently be used in an interactive article.

[1] E.g. https://github.com/distillpub/post--building-blocks.

ArchieML has been adopted by a number of major newsrooms, and is used by both technical staff and non-technical editors. The widespread use of ArchieML shows that editorial staff are able and willing to pick up simple markup languages in order to work more closely with technical collaborators. While ArchieML makes it easy to pull text into code, Idyll makes it easy to include JavaScript components in text. We contend that our approach makes the relationship between code and text easier to reason about from an editorial perspective, and enables control over where components appear in text and how they interact with the page. This approach allows Idyll to eliminate the need to write a large class of code typically required when producing these articles.

PersaLog [6] is a DSL for dynamic personalization of news articles. Idyll similarly provides a DSL for interactive articles, but targets a broader range of components and article designs.

DESIGNING AN INTERACTIVE MARKUP LANGUAGE

The design of Idyll — a markup language for interactive documents — was informed by prior research and motivated by the first author's domain experience working in the digital journalism industry, where cross-platform multimedia and interactive content must be designed, implemented, and published according to a timely newsroom schedule. We aimed to build a tool that would decrease the overall amount of time necessary to create and publish such interactive content, and that would enable a collaborative production process between technical and editorial users. We iteratively refined a design document for the language in response to feedback from domain experts and early use tests, including interviews with professional journalists and designers.

Requirements

The initial draft of the language was driven by knowledge articulated in prior research and obtained by the authors through industry experience. The first author of this paper spent several years authoring interactive graphics and articles, and building tools to support their production and publication across several major news outlets. We derived a set of requirements from our findings regarding a number of social and technical issues.

Process

Lee et al. [21] elucidate the process of producing data-driven stories in their "visual data storytelling process" model. While this model concerns data-driven storytelling specifically, we find that it can be applied more generally to the production of interactive articles: (1) anecdotes, assets, data, and facts are collected; (2) story copy is written, interactive components are designed and scripted; (3) editors refine the content and presentation of the story; (4) the article is published.

This model provides motivation for separating editorial and technical concerns, a need apparent in the design of other tools such as ArchieML. We want Idyll to support this workflow. *Editors should be able to edit copy and arrange interactive widgets without writing code; developers should be able to easily integrate their code into articles.*

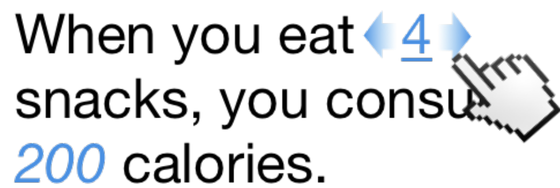

Figure 3. A stereotypical behavior of interactive documents: values can be modified by the reader in order to effect change change elsewhere on the page. In this example, adapted from the Tangle's documentation [46], the number of snacks consumed can be modified, and the sentence updates to show the equivalent amount of calories.

Architecture

Real-world processes are more nuanced than abstract models, and specific workflows and requirements may vary across institutions. To support a wide range of workflows, *Idyll should be modular, customizable, and interoperable with existing (and future) tools for creating interactive graphics*. By designing the tool as a set of composable modules, it may either be used holistically or customized to support specific use-cases. By requiring interoperability with other domain-specific tools, we can build a system that is easy to learn on its own, but also supports a wide range of outputs.

In order to facilitate editing of the article by non-technical users, Idyll should support *declarative specification of text, behavior, and layout*, rather than requiring imperative code to update document state. The specification format needs to be powerful enough to support interactive behavior; for example it must be possible to implement the type of dynamism shown in Figure 3. To achieve this, we draw on functional reactive programming, which has been shown to be an effective means for enabling expressive declarative languages, and can be similar applied in this domain. Idyll uses *a reactive parameterization of the article specification*.

Features

While the design space of interactive web articles is a fast-moving target, prior work has attempted to characterize common narrative patterns used in data-driven stories and interactive articles. McKenna et al. [24] present an analysis of navigational techniques that are often used, such as scroll-based navigation ("scrollytelling"), and step-based navigation (e.g., slideshows). In addition to navigation, designers must often implement custom behavior across graphics in response to reader interaction. With currently available tooling, custom code needs to be written to bind variables to widgets and track changes to internal state as readers interact with them. Idyll includes *a standard component library to make common navigation (e.g., scroll- and step-based) and input techniques straightforward to specify*.

Publication

Content published online should be fast to load, compatible with a wide range of browsers and devices, accessible to screen-readers, and, in some cases, available in multiple languages. Adhering to the best-practices for serving JavaScript may require a complex build process to transform the code that journalists and developers write into something that can

be delivered to a reader. For example, to reduce page load time, developers must limit the number of network requests performed and data transferred. To do this, separate scripts are often compiled into a single file and compressed (e.g., minified). With Idyll, we seek to *support the complex JavaScript build configurations necessary for publication*.

Another concern is the way in which HTML is delivered to the reader. In typical web applications an HTML document is rendered by a server, often by querying a database for information and injecting that into an HTML template. To reduce infrastructure costs and deployment complexity, interactive articles typically do not have a database component, and are completely encapsulated by the HTML, CSS, and JavaScript bundle that is delivered to readers. Article content is included directly in the HTML, and JavaScript code attaches event listeners to this existing content when the page is loaded: this process is known as *hydration*. Following best practices, Idyll should *generate the initial HTML before it is sent to the client, and hydrate interactivity upon page load*.

Interviews

After the initial requirements and candidate language specification were drafted, we solicited feedback from 13 experts from the fields of digital journalism, education, information visualization, and scientific publishing, including journalists and technologists from FiveThirtyEight, The New York Times, and The Washington Post. Twelve of the experts expressed positive reactions and acknowledged seeing value in such a tool (*"Overall the declarative nature of creating the interactive documents here is really neat"*, *"I kinda feel like it's the thing I've always wanted, looking at the [markup] for that page is what convinced me."*). The remaining expert noted that they were *"sometimes puzzled by Markdown specific approaches,"* indicating that they had a hard time seeing how Markdown could integrate into their existing code-heavy workflow.

The experts provided detailed feedback that included suggestions for changes that would improve workflow (*"It might also be worth supporting comments in the syntax. In our ArchieML documents, we've used comments occasionally to make suggestions and to explain the syntax a bit to copy editors"*), requests for additional features (*"It would be a nice feature to have a csv to json converter as part of the build"*), and notes about potential technical hurdles (*"Async stuff is always... interesting in this sort of project, e.g. if you need to make chained network requests and do something with the end result"*). We incorporated this feedback into our requirements document and built the initial version of Idyll.

THE IDYLL LANGUAGE

Idyll is a markup language that targets interactive browser-based articles, generating HTML, JavaScript, and CSS. An Idyll document can contain both *textual markup* using Markdown syntax and *component markup* used to declare variables and interactive components. Listing 1 shows basic usage of Idyll syntax. To produce the output, the Idyll compiler takes in a markup file and creates a list of article components. Figure 4 shows how these components are combined with style information in order to produce a webpage that may be published

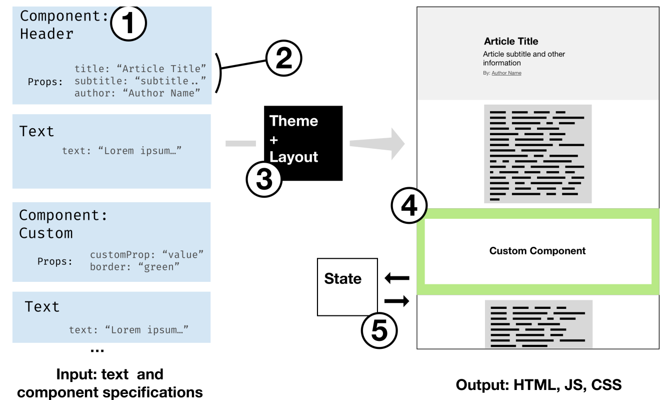

Figure 4. The Idyll article model. (1) The Idyll compiler transforms input markup into a list of document nodes; (2) each of these nodes has a dictionary of properties that determine its behavior (optionally including a list of children which are recursively rendered). (3) The nodes are combined with theme and layout information to (4) construct a static HTML page. When the page is loaded in a browser it is (5) hydrated with event handlers and a reactive state to drive interface updates.

```
# I am a header

Variables can be declared anywhere:
[var name:"x" value:5 /]

and *Markdown* syntax can be used
for inline styling.

The value of x is [Display value:x format:"d" /].

[Range value:x min:0 max:10 /]
```

Listing 1. Example Idyll markup: Markdown syntax is extended to support variable declaration and interactive component specification. This input file compiles to an HTML, JS, and CSS bundle that can be rendered by all major web browsers.

online. Idyll implements a reactive runtime that responds to user input events and updates rendered output in response.

Language Constructs

The basic language primitives in Idyll consist of text, components, reactive variables, and style directives.

Text

The most basic building block of an Idyll article is text. Users can write plain text in Idyll markup and this text will appear in the rendered output. Text can be written using Markdown syntax, allowing users to easily create lists, headers, blockquotes, and stylized text (e.g., bold or italicized). Markdown enjoys widespread use, with a syntax understood by a wide audience that includes non-technical writers and editors.

Components

Idyll extends Markdown with support for embedded interactive components and variables. The components may either refer to a standard Idyll component, a third-party component, or a custom component provided by the author. Components are specified by a name and parameterized by a list of properties, the values of which can be dynamic. When a variable is declared it is added to a global state object that listens for changes to the variable; upon changing, Idyll will re-render

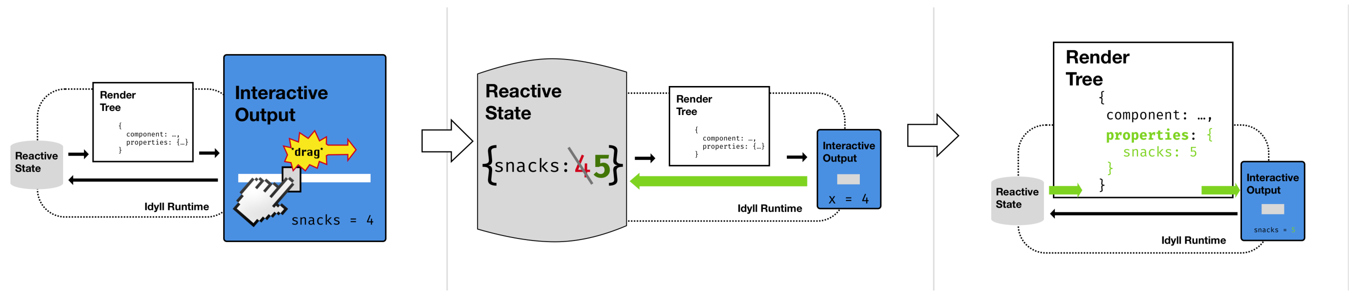

Figure 5. In Idyll, a document's variable values can be bound to control widgets, allowing authors to quickly implement components that respond to user input. This graphic shows how reader interaction propagates through Idyll's state and affects the rendered output.

any components which depend on the value of that variable. Idyll must find an implementation for each component included in the markup: to do this it matches the name against author-provided custom components, a standard library of components, installed third-party components, and — if no matches are found — valid HTML tags.

Idyll's standard library of components consists of 26 components, chosen to help authors implement a range of common design goals without writing JavaScript code. While we don't expect Idyll to eliminate the need to write code, the standard library serves to limit the code that users do need to write for their specific content and story.

The standard library components span four categories:

Presentation components display content on the screen, for example the `Equation` component renders typeset LaTeX code, and the `Table` component renders tabular data.

Input components accept input from users to drive the behavior of articles. For example, the `Range` component adds a slider that can be bound to a variable; the `Select` component displays a list of items in a dropdown menu.

Layout components manipulate how content is displayed. These components can be used as building blocks to create many common designs seen in the narrative visualization space. For example, the `Fixed` component defines content that stays fixed to the screen while a reader scrolls through an article. The second and third examples in the Examples section demonstrate the use of several of these components.

Meta components help with behind-the-scenes tasks and do not modify visible content. Meta components may be used to add document metadata or usage analytics to an article.

Reactive Variables & State

Idyll maintains an internal state that determines how the components are rendered at any point in time. The state consists of user-declared variables and additional information about components with respect to the reader's viewport, for example whether or not a particular component is in view. Variables are declared using a similar syntax to components (Listing 1).

Idyll's variables are reactive: any time the value of a variable is modified, dependent components are re-rendered to reflect the change. Figure 5 shows how this works in practice. The article is rendered as a function of the input content and initial state; an event occurs that causes the state to be updated, for example a "click" event occurs when the user clicks a button; the updated state propagates to the article's components, and the webpage is re-rendered to reflect the changes.

Variables are modified in response to events. Idyll exposes all events provided by the browser's Document Object Model (DOM), including clicks (and taps), mouse hover, and keyboard input. It also adds several high-level viewport events that are useful for defining scroll-driven interactions. For example, the following code specifies that a custom data visualization will only animate while it is fully in a reader's viewport:

```
[var name:"isAnimating" value:false /]

[CustomDataVisualization
  animating:isAnimating
  onEnterView:`isAnimating = true`
  onExitViewFully:`isAnimating = false` /]
```

Variables may also be updated by components: a special function `updateProps` is provided, which can be called to tell Idyll to update the value of a variable which is bound to a particular property of that component. For example, in Listing 1 the `Range` component's property `value` is bound to the Idyll variable `x`. When the user interacts with the rendered range slider, the Range component calls `updateProps({ value: newValue })`, and Idyll subsequently updates the value of `x`.

Style

In web programming, styling of content typically is done using CSS [22], which can be used to specify colors and typography as well as element sizing and layout. Idyll adds additional structure to how articles are styled: the look and feel of an article is determined by its *layout*, *theme*, and any *custom styles* provided by the article author. The layout is responsible for the overall page structure, specifying, for example, if the text container is centered on the page or left aligned, and where the article title should appear. The theme is responsible for colors, typography, and component-specific styles. The separation of form and content offers authors an easy way to view their articles under a variety of different stylings without making code changes, while also offering a way for institutional styles to be applied across an organization's articles without requiring authors to adapt their workflow.

Idyll includes two layouts and three themes, which are suitable for a range of use cases. The *blog* layout utilizes left aligned

body text with a wide right margin to accommodate the positioning of complex graphics; the *centered* layout is a more traditional center aligned article layout. The three themes *github*, *tufte*, and *idyll* are different skins inspired by the style of GitHub READMEs, the print design of Edward Tufte, and Idyll's own visual design, respectively. Users can extend and modify these themes using CSS, and can also install themes that others have shared.

Implementation

Idyll is implemented via a collection of composable modules. A command-line tool is used to compile source files and generate output that can be viewed in a web browser. This output contains a JavaScript runtime that reactively updates article components in response to reader actions. These modules are all available on GitHub as open source software.

Build

Projects are compiled using a node.js-based command-line tool. Users specify an Idyll markup file as input, along with any custom interactive components (written in JavaScript) and datasets. Users can also optionally define custom article themes and layouts to control the look and feel of the resulting output (e.g., using CSS). The input markup is transformed by a compiler into an abstract syntax tree (AST). The nodes in the AST are matched with interactive components which are then sent through a JavaScript bundler and minifier, resulting in a production-ready code package, which can be deployed to any static web host.

Runtime

The Idyll runtime renders interactive output according to the bundled AST and JavaScript components. The runtime provides a reactive variable store implemented using React.js [3], a reactive web framework. Idyll is compatible with any preexisting React components; Idyll users have access to an existing repository of thousands of open source components.

EXAMPLES

We present a number of example Idyll programs to substantiate our claims that Idyll (1) reduces the amount of code and effort needed to create interactives, (2) promotes clear and concise code, and (3) is able to express a wide range of designs. A variety of full examples have been produced with Idyll, available for viewing both online and in the supplementary materials. To open these examples in a browser, see the supplementary materials or visit `http://idyll-lang.org/`.

The examples that follow are based on techniques used in real-world articles, and focus on explaining how specific design techniques can be achieved concisely using Idyll markup. They are illustrative of the way in which Idyll's architecture and language features eliminate the need for much of the code that goes into powering interactive articles, and demonstrate the language's expressiveness. We encourage readers to view the examples in a web browser to gain a more intuitive understanding of the types of designs and interactivity supported.

Reactive Updates to User Input

A common design in the domain of interactive text is to embed text and numbers that change based on a reader's input. The following is an example taken from Tangle's documentation, recreated using Idyll. Figure 3 shows possible output. The example displays the sentence "When you eat 4 snacks, you consume 200 calories." The number `4` is dynamic: a reader may change its value with a drag interaction, in turn updating the displayed number of snacks and calories consumed.

The Idyll implementation of this program is shown in Listing 5. The program starts by declaring the variable `snacks` and initializing its value to 4. The following text renders a sentence with the form "When you eat ______ snacks, you consume ______ calories", using `Dynamic` and `Display` components to fill in the blanks. The `Dynamic` component creates a two-way binding between the `snacks` variable and the number rendered on-screen. The number is rendered with visual cues showing that it can be dragged, and if the user modifies it, the new variable value is propagated to dependent components.

Idyll allows expressions to be passed directly as properties, eliminating the need to create a separate derived variable in many cases. The `Display` component renders the value of an expression, $50 * snacks$. As the user interacts with the `snacks` variable, the value displayed is updated. If another variable is needed for clarity, Idyll's `derived` variables can reactively compute a value based on other variables.

Fixed Sections and Scroll Events

Interactive articles often dynamically update on-screen content as a reader scrolls through the page, modifying the layout in more complex ways than in the previous example. For example, it is common to have a graphic scroll into view alongside text, and stay fixed in view until the reader scrolls through to the next section. To facilitate this type of interaction, Idyll includes a `[Scroller /]` layout component that lets authors declaratively define content that will stay fixed in their readers viewports' while they scroll. Listing 3 shows the Idyll markup necessary to construct such a scrolling experience. Multiple `[Scroller /]` components can be listed to define a series of scroll sections and their associated graphics.

Figure 6 shows an example of an Idyll article that uses this technique. The article, which interactively explains kernel density estimation (KDE), displays a fullscreen fixed graphic that stays in a reader's viewport while they scroll through the article. As they reach certain sections, waypoints are triggered which update Idyll variables, this update in turn causes the graphic to render in a new state.

One of the benefits of declaratively specifying article components in this way is that authors can quickly explore alternative designs. For example, Listing 2 shows the code for displaying the same graphic using a *stepper* design[2]. Note the similarities between Listing 3 and Listing 2: both define a graphic component and a series of steps and both use a variable to track the reader's progression through the content.

Both the Scroller and Stepper components accept properties that allow authors to further customize their behavior. For

[2] A *stepper* can be thought of as a generalization of a slideshow. The graphic shows the content of only one step at any given time, and is updated when the `currentStep` property changes.

```
[var name:"step" value:0 /]

[Stepper fullWidth:true currentStep:
    step]
  [Graphic]
    [CustomComponent state:scrollStep
        /]
  [/Graphic]

  [TextContainer]
    [Step]
      Text for the first section
    [/Step]
    [Step]
      Text for the second section
    [/Step]
    [Step]
      Text for the third section
    [/Step]
  [/TextContainer]
[/Stepper]
```

Listing 2. Idyll markup for stepper-based navigation, in which a user clicks through slides of content. At each stage the `step` variable is updated, causing the custom component to update its state.

```
[var name:"step" value:0 /]

[Scroller fullWidth:true currentStep:
    step]
  [Graphic]
    [CustomComponent state:step /]
  [/Graphic]

  [TextContainer]
    [Step]
      Text for the first section
    [/Step]
    [Step]
      Text for the second section
    [/Step]
    [Step]
      Text for the third section
    [/Step]
  [/TextContainer]
[/Scroller]
```

Listing 3. Idyll markup for scroll-based navigation, in which a graphic will stick in the viewport when the user is scrolling through each step, and release otherwise. The `step` variable updates as each step comes into view.

```
[var name:"state" value:"initial-
    state" /]

[Fixed fullWidth:true]
  [CustomD3Component state:step /]
[/Fixed]

[Scroller currentState:step]
  [TextContainer]
    [Step state:"initial-state"]
      Text for the first section
    [/Step]
    [Step]
      This step will not trigger a
          state update
    [/Step]
    [Step state:"secondary-state"]
      Text for another section
    [/Step]
  [/TextContainer]
[/Scroller]
```

Listing 4. Idyll markup for the layout seen in Figure 6. A graphic stays fixed fully in the background while the reader proceeds through the article. The scroller is used to send state updates to the graphic based on the reader's progress.

```
[var name:"snacks" value:4 /]

When you eat [Dynamic value:snacks /]
snacks, you consume
[Display value:`50 * snacks` /] calories.
```

Listing 5. This Idyll markup produces the behavior shown in Figure 3, displaying a dynamic number which when changed will cause later text in the sentence to update.

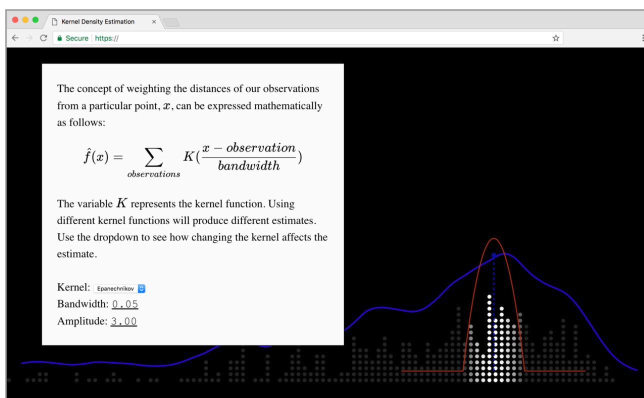

Figure 6. An interactive article explaining Kernel Density Estimation. The article uses scroll-based interactivity defined declaratively in Idyll markup. Listing 4 shows the Idyll markup used to specify this layout. The equations and controls shown in this screenshot were defined in Idyll markup and reactively parameterize the custom graphic.

example, an author can use the Scroller component with the `disableScroll` property to specify the hybrid scroller-stepper navigation characterized by McKenna et al. [24]. This type of navigation, which can be seen in popular data-driven stories such as *Rock n' Poll* by Maarten Lambrechts [20], overrides the browser's default scroll behavior, requiring readers to click to scroll to the next section.

Updating a Custom Visualization

The previous example showed how Idyll's responsive variable system and built-in components can be used together to create interactive experiences without writing custom JavaScript. However, in many real-world scenarios authors will need to write code to create custom components for their articles. In this example we show how responsive variables can be used to parameterize a custom visualization built with D3. The interface exposed by Idyll enforces separation of concerns between editorial issues (the text of the document, order of sections, events that trigger certain content to be displayed, etc.) and specific component implementation details.

Figure 7 shows a screenshot of *The Etymology of Trig Functions*, an interactive blog post made with Idyll. The post uses a custom visualization in order to illustrate the history of some trigonometric terms. This story uses interactive text components to invite the reader to interact with the graphic as they progress through the text. The visualization updates as readers interact with different components on the the page.

Listing 6 sketches how the article in Figure 7 can be created. The `[Action /]` tag displays text that can respond to user events, in this case `onMouseEnter`, which fires when a reader hovers their mouse over the text. The `Action` component is used in conjunction with the `[Fixed /]` component, which causes its contents to remained fixed on-screen in the article's margin as the reader scrolls. We also include `TrigDisplay`, a tag which instructs Idyll to load a custom JavaScript component in a file named `trig-display.js`.

The skeleton of the custom `TrigDisplay` component code is shown in Listing 7. A component author must implement `initialize` and `update` functions. The `initialize` function is called when the component is first added to the page. The `update` function is called every time a relevant Idyll variable updates, and accepts the new and previous values of the component properties as input.

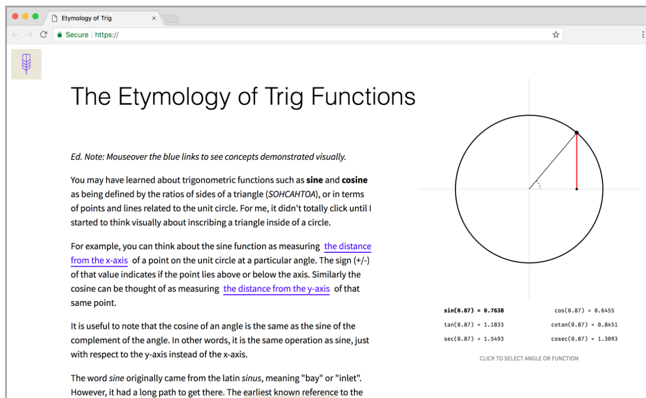

Figure 7. *The Etymology of Trig Functions*, an interactive blog post written with Idyll. This example integrates narrative and graphics through user interaction with the text. As a user hovers their mouse over stylized text, the graphic on the right updates to show a geometric interpretation of the concept being discussed.

```
[var name:"state" value:"init" /]

...lorem ipsum...

[Action onMouseEnter:`state = "showCosine"` ]
  Mouse over this text to see the "cosine" state.
[/Action]

...lorem ipsum...

[Fixed position:"right"]
  [TrigDisplay state:state /]
[/Fixed]
```

Listing 6. Idyll markup similar to that used by *The Etymology of Trig Functions*, shown above. As a user hovers their mouse over the action component (which renders stylized text), the state variable will update, causing the fixed graphic to update.

```
initialize(node, props) {
  // render initial graphics
  this.svg = d3.select(node).append('...')
}

update(props, oldProps) {
  // update based on new page state
  this.svg.selectAll('...')
}
```

Listing 7. JavaScript interface for custom components. A custom Idyll component requires implementing two functions: `initialize` and `update`. The initialize function is called only once, on page load, and is used to render the initial view of the graphic. The update function is called when a component's properties change, for example in response to a user moving a slider or scrolling to a new section.

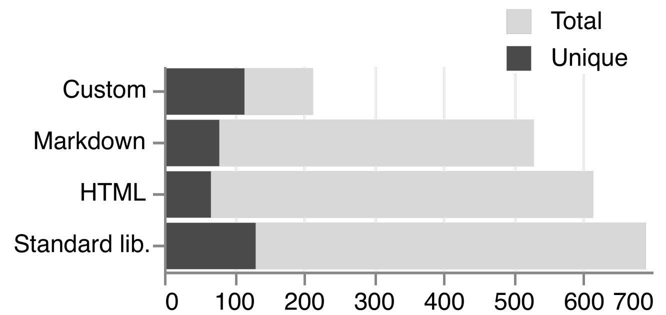

Figure 8. Usage counts of various language features across student groups, along with the total unique uses of features per group. The students were comfortable using the Markdown syntax, and relied heavily on Idyll's standard library of components, reducing the overall amount of code that they had to write.

These same Idyll mechanisms — `Action` links, `Fixed` layout, and custom components — were used to author the article shown in Figure 1, which uses an interactive data visualization to explain the Barnes-Hut approximation of n-body forces. Hyperlinks and input components embedded in the text parameterize the main visualization state, while users can also interact directly with the visualization to explore on their own.

DEPLOYMENT

Idyll has been released as free and open source software, and has been downloaded over 25,000 times [5] since its initial release. The code is available at https://github.com/idyll-lang/idyll, and the language is available for testing in an online editor at https://idyll-lang.org/editor/. The project has received positive feedback from a number of professional journalists and software developers. When asked about an early version of Idyll, one *Washington Post* reporter with JavaScript experience noted, *"I finally played around with Idyll a bit and it seems great. [I] got everything working with no problem and the syntax was super easy to use. I didn't dig too much into building something crazy, but I still was able to create a couple custom components and charts."*

Idyll has been well received by the open source programming community, and has received substantial core code updates from outside contributors: five external contributors have made 280 code commits to Idyll's core modules, and others have helped improve the documentation and examples. Community members have used the language to make their own examples and posts. One graphics programmer uses Idyll to power his personal blog, having converting it from an existing site powered by Markdown. He commented, *"Sharing components and styles seems to be working really well! . . . Very pleasant experience overall. Life gets way simpler when you quit trying to be fancy and just make it work."*

After releasing the initial version of Idyll, the authors worked with Folo Media, a non-profit news outlet, to produce a data-driven story about school funding in Texas [3]. While the story was never published for reasons unrelated to Idyll, the process was illustrative of how the tool can be used in practice.

[3] A draft is available at https://mathisonian.github.io/texas-school-finance/.

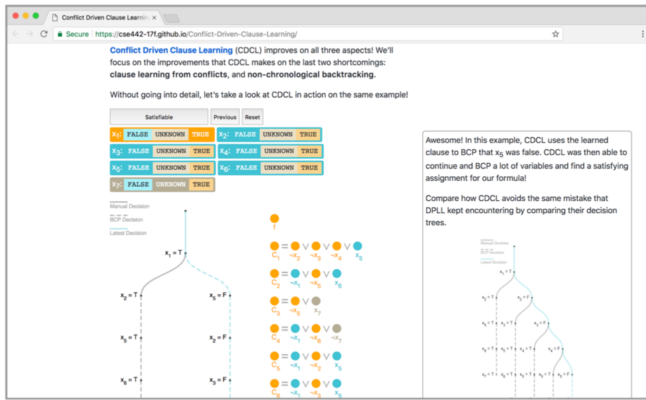

Figure 9. A student-authored Idyll article explaining the conflict-driven clause learning (CDCL) SAT solver. The article displays custom visualizations parameterized by Idyll variables and standard components. The detail shown above visualizes a logical formula; readers can use the buttons to toggle the truth assignment of variables, and see the effects on clauses and the overall formula.

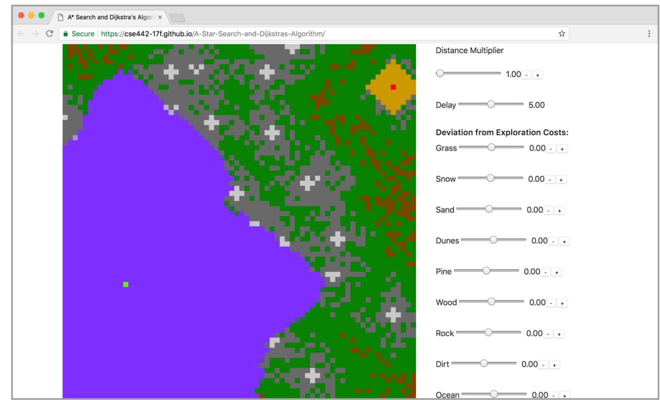

Figure 10. This student article, *A* Search and Dijkstra's Algorithm*, is motivated by the use of path-finding algorithms in video games. The students developed a custom game, controlled via Idyll components, in which players choose optimal search parameters in order to win.

The story was written by a journalist (who was familiar with Markdown and had basic experience with the programming language R, but was otherwise non-technical), was edited by several non-technical staff members at Folo Media, and required the creation of several custom components, which were programmed by the first author. The journalist was able to pick up Idyll markup basics without issue and saw the value of the workflow that Idyll enabled, although he hoped for a more streamlined way to incorporate charts created during data analysis in R. *"Is anyone working on an R wrapper for Idyll? I feel like your language is like the ideal way for developing interactive narratives. I know folks have been working on ways to better integrate JavaScript into the R universe."* The editors did not directly engage with the Idyll markup, instead looking at drafts of the article in a web browser and sending targeted notes via chat and email, but they were able to participate in an iterative design dialog enabled by Idyll's standard component library (*"What if we made this component a Stepper instead of a Scroller?"*) and decided to add additional content (audio interviews) to the article after realizing Idyll's support for such rich media.

Idyll in the Classroom

An undergraduate data visualization class used Idyll for their final projects. The computer science students were required to form groups of four and construct an interactive article that explained an algorithm of their choice. While students are not the intended end users of Idyll, they represent a reasonable proxy for the technical users that we wish to engage. The students are technically proficient but far from expert web developers; the study does not speak to usage by non-technical editorial users. Of the 20 groups, 19 of them successfully used Idyll to create an interactive article. (One group decided to focus on making video graphics and opted to embed these videos directly in HTML.)

Figure 8 shows how the students utilized different language features in their articles. All student groups were comfortable using Markdown, and all took advantage of Idyll's built-in components. The high use of Idyll's standard components indicates that our standard library reduces the overall amount of code that authors need to write, as they don't need to implement a range of functionality themselves. We found some of the student usage of HTML tags to be due to students needing to work around issues in Idyll's compiler, (for example, many groups used the `[br /]` tag to insert extra whitespace where the compiler had stripped it away), though they also used semantic tags such as `[section] [/section]` to group their content.

All but one of the groups included their own custom components using Idyll's JavaScript bundling infrastructure. The group that did not include any custom Idyll components wrote their JavaScript separately and included it in the article via `[iframe /]` tags. The majority of the custom components that were developed were algorithm-specific visualization components. The basic components provided by Idyll covered a wide range of use cases for capturing reader input and describing document layout and scroll behavior, allowing students to focus on writing code for custom algorithm visualizations.

Figure 9 presents an article about *conflict driven clause learning* (CDCL), an algorithm for solving the boolean satisfiability problem. Using the game Sudoku as motivation, the student authors walk readers through the steps of CDCL and related algorithms. The piece concludes with an interactive Sudoku solver that compares CDCL with another solver. The students were able to combine standard Idyll components with custom visualizations to create an engaging visual explanation.

Teams were able to use Idyll to create sophisticated interactive experiences. Figure 10 shows a screenshot from an article about A* search and Dijkstra's pathfinding algorithm. The narrative discusses the importance of path-finding algorithms in video games, and uses interactive graphics to display how the algorithms would find paths to varying points in game levels. The students included an interactive game, *aStarCommando*, where players tweak parameters of a search algorithm in order to navigate to a safe house located across the map before they are killed or captured. The game was parameterized by Idyll's reactive variables and controlled by Idyll input widgets.

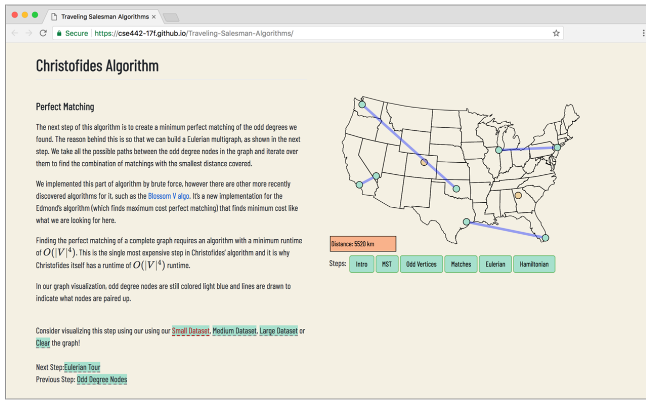

Figure 11. A student article explaining the Travelling Salesman Problem. The students leveraged Idyll's features for modifying styles and layout. They were able to create a highly personalized page, incorporating both scroll- and step-based navigation.

Some students took advantage of Idyll's layout components and flexibility with styling. Figure 11 shows an article on the classic Travelling Salesman Problem (TSP). The student authors of this article customized the color theme, typography, and standard component styles. They used Idyll's variables and components to power interactive maps, upon which different TSP algorithms are demonstrated. The students used a combination of scroll-based and step-based navigation to produce an article with a sophisticated look and feel.

While students had few issues picking up the basic syntax and workflow, some faced initial difficulties adjusting to the declarative, reactive nature of Idyll markup. Rather than coordinate interaction via Idyll variables and component properties, some students attempted to code coordination themselves by setting and reading global variables in JavaScript. Others sought to invoke methods defined on custom components directly from the markup.

LIMITATIONS

While the results of the classroom study indicate that technical users are able to use Idyll to create a range of designs, the study does not speak to usage by non-technical editorial users. We believe that the widespread adoption of ArchieML, along with the feedback we've received from journalists, show that these users are willing and able to adopt simple markup languages. A more targeted study is needed to say how usable Idyll is for editorial users in its current form, and what additional training, if any, would be necessary for them to use the tool effectively.

Idyll does not eliminate the need for writing JavaScript code to create custom graphics. Custom Idyll components can grow to become arbitrarily complex, and beyond providing reactivity, Idyll doesn't do much to decrease the complexity of these custom graphics. A closer integration with tools that focus on the creation of custom graphics may be needed in order to lower the threshold for creating articles that include custom components. Idyll might also benefit from a closer integration with other existing data science tools, as many data journalists use tools like R and Python to create static graphics. A more streamlined workflow might, for example, allow a user to embed R code directly in their markup and have a static graphic be generated at compile time.

Idyll's two-way variable binding allows components to trigger page updates by modifying Idyll variables via a special function, however two-way binding to derived variables is not supported. This distinction may be confusing to some users, and it may be addressed in a future version by adding a constraint solver to Idyll's runtime.

FUTURE WORK

While Idyll is already a useful tool, we see a variety of opportunities for future research and extensions.

Evaluation. As Kosara & Mackinlay [19] point out, *"Controlled studies today are often done in the lab, and typically within a relatively short time frame. Evaluation of stories will require a very different approach, to account for different scenarios and to reflect real-world uses."* Idyll offers a path forward in increasing our understanding of the effectiveness of stories: because of the centralized state management, tracking reader interactions with a story just means listening for changes to the central store. Idyll offers an API to listen for these changes, making it straightforward to collect structured analytics and empowering future work on evaluation.

Retargeting. The declarative nature of Idyll markup makes it possible to use alternative renderers to target additional output formats. Idyll can be adapted to generate interactive native iOS and Android applications (using the React Native renderer); we also have early support for LaTeX output. These alternative renderers suggest that it is possible to extend Idyll such that authors can choose from a list of possible output targets (e.g., web, mobile web, mobile native, PDF), generating output that is interactive where possible and gracefully degrading in environments that do not support interactivity. This flexibility is particularly desirable for scientific publishing, where authors are required to submit static documents to conferences and journals but may also wish to provide a web-based interactive version of their work.

Visual Editing. Non-technical users may prefer to write text in a visual what-you-see-is-what-you-get (WYSIWYG) editor. Such an editor has the potential to lower the threshold for adding dynamism to static text, but there are major design challenges when building an editor to not only specify text, but also interactive behavior and layout. Idyll markup could provide a convenient output target for such an editor and serve as a file format, abstracting away technical issues involved in compiling interactive output, lowering the barrier to create such an editor.

CONCLUSION

We contribute Idyll, a novel domain-specific language for creating interactive narratives. Idyll combines a simple markup language with reactive JavaScript components to reduce the amount of code and effort needed to produce and publish interactive articles. Code, examples, and documentation are available online at `https://idyll-lang.org/`.

REFERENCES

1. 2008. Jekyll. (2008). Retrieved August 1, 2017 from https://jekyllrb.com/
2. 2013. Fangle. (2013). Retrieved September 19, 2017 from https://github.com/jotux/fangle
3. 2015. React. (2015). Retrieved August 1, 2017 from https://facebook.github.io/react/
4. 2017. Explorable Explanations Web Catalog. (2017). Retrieved August 1, 2017 from http://explorabl.es/
5. 2017. Idyll download count. (2017). Retrieved August 1, 2017 from https://npm-stat.com/charts.html?package=idyll
6. Eytan Adar, Carolyn Gearig, Ayshwarya Balasubramanian, and Jessica Hullman. 2017. PersaLog: Personalization of News Article Content. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17)*. ACM, New York, NY, USA, 3188–3200. DOI: http://dx.doi.org/10.1145/3025453.3025631
7. Bill Atkinson. 1988. *Hypercard*. Apple Computer.
8. Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. 2013. A Survey on Reactive Programming. *ACM Comput. Surv.* 45, 4, Article 52 (Aug. 2013), 34 pages. DOI:http://dx.doi.org/10.1145/2501654.2501666
9. Nicholas Bollweg. 2011. TangleDown. (2011). Retrieved September 19, 2017 from https://github.com/bollwyvl/TangleDown
10. Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. 2011. D3: Data-Driven Documents. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)* (2011). http://vis.stanford.edu/papers/d3
11. Matthew Brehmer, Bongshin Lee, Benjamin Bach, Nathalie Henry Riche, and Tamara Munzner. 2017. Timelines Revisited: A Design Space and Considerations for Expressive Storytelling. *Transactions on Visualization and Computer Graphics (TVCG)* 23 (September 2017), 2151 – 2164.
12. Bill Burdick. 2011. leisure. https://github.com/zot/Leisure. (2011).
13. Kerry Shih-Ping Chang and Brad A. Myers. 2017. Gneiss: spreadsheet programming using structured web service data. *Journal of Visual Languages & Computing* 39, Supplement C (2017), 41 – 50. DOI: http://dx.doi.org/https://doi.org/10.1016/j.jvlc.2016.07.004 Special Issue on Programming and Modelling Tools.
14. Distill. 2017. Latest articles about machine learning. (2017). Retrieved September 19, 2017 from https://distill.pub/
15. Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. In *International Conference on Functional Programming*. http://conal.net/papers/icfp97/
16. John Gruber. 2004. Markdown. (2004). Retrieved August 1, 2017 from https://daringfireball.net/projects/markdown/syntax/
17. Amit Kapoor. 2016. Visdown. (2016). Retrieved August 1, 2017 from http://visdown.amitkaps.com/
18. Jun Kato, Tomoyasu Nakano, and Masataka Goto. 2015. TextAlive: Integrated Design Environment for Kinetic Typography. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)*. ACM, New York, NY, USA, 3403–3412. DOI: http://dx.doi.org/10.1145/2702123.2702140
19. Robert Kosara and Jock Mackinlay. 2013. Storytelling: The next step for visualization. *Computer* 46, 5 (2013), 44–50.
20. Maarten Lambrechts. 2016. Rock 'n Poll: Polls explained with interactive graphics. https://web.archive.org/web/20180307013513/http://rocknpoll.graphics/. (2016).
21. 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.
22. Hakon Wium Lie and Bert Bos. 2005. *Cascading style sheets: designing for the Web*. Addison-Wesley Professional.
23. Kwan-Liu Ma, Isaac Liao, Jennifer Frazier, Helwig Hauser, and Helen-Nicole Kostis. 2012. Scientific storytelling using visualization. *IEEE Computer Graphics and Applications* 32, 1 (2012), 12–19.
24. S McKenna, N Henry Riche, B Lee, J Boy, and M Meyer. 2017. Visual Narrative Flow: Exploring Factors Shaping Data Visualization Story Reading Experiences. In *Computer Graphics Forum*, Vol. 36. Wiley Online Library, 377–387.
25. Leo A Meyerovich, Arjun Guha, Jacob Baskin, Gregory H Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. 2009. Flapjax: a programming language for Ajax applications. In *ACM SIGPLAN Notices*, Vol. 44. ACM, 1–20.
26. Brad Myers, Scott E. Hudson, and Randy Pausch. 2000. Past, Present, and Future of User Interface Software Tools. *ACM Trans. Comput.-Hum. Interact.* 7, 1 (March 2000), 3–28. DOI:http://dx.doi.org/10.1145/344949.344959
27. Observable 2018. Observable. https://observablehq.com/. (2018).
28. Chris Olah and Shan Carter. 2017. Research Debt. *Distill* (2017). Retrieved August 1, 2017 from http://distill.pub/2017/research-debt
29. Fernando Pérez and Brian E. Granger. 2007. IPython: a System for Interactive Scientific Computing. *Computing in Science and Engineering* 9, 3 (May 2007), 21–29. DOI: http://dx.doi.org/10.1109/MCSE.2007.53

30. Polymer Project 2017. Polymer Project. https://www.polymer-project.org/. (2017).
31. National Public Radio. 2016. dailygraphics. https://github.com/nprapps/dailygraphics. (2016).
32. 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 (UIST '17)*. ACM, New York, NY, USA, 715–725. DOI: http://dx.doi.org/10.1145/3126594.3126642
33. Arvind Satyanarayan and Jeffrey Heer. 2014. Authoring Narrative Visualizations with Ellipsis. *Comput. Graph. Forum* 33, 3 (June 2014), 361–370. DOI: http://dx.doi.org/10.1111/cgf.12392
34. 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
35. Arvind Satyanarayan, Kanit Wongsuphasawat, and Jeffrey Heer. 2014. Declarative Interaction Design for Data Visualization. In *ACM User Interface Software & Technology (UIST)*. http://idl.cs.washington.edu/papers/reactive-vega
36. Toby Schachman and Joshua Horowitz. 2016. Apparatus. https://github.com/cdglabs. (2016).
37. Edward Segel and Jeffrey Heer. 2010. Narrative Visualization: Telling Stories with Data. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)* (2010). http://vis.stanford.edu/papers/narrative
38. 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).
39. Michael Strickland, Archie Tse, Matthew Ericson, and Tom Giratikanon. 2015. Archie Markup Language (ArchieML). (2015). Retrieved August 1, 2017 from http://archieml.org/
40. Tampa Bay Times. 2016. lede. https://github.com/tbtimes/lede. (2016).
41. The New York Times. 2017. kyt. https://github.com/NYTimes/kyt. (2017).
42. Lea Verou, Amy X. Zhang, and David R. Karger. 2016. Mavo: Creating Interactive Data-Driven Web Applications by Authoring HTML. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology (UIST '16)*. ACM, New York, NY, USA, 483–496. DOI: http://dx.doi.org/10.1145/2984511.2984551
43. Bret Victor. 2006. Magic Ink: Information Software and the Graphical Interface. Retrieved August 1, 2017 from http://worrydream.com/MagicInk/
44. Bret Victor. 2011a. Explorable Explorations. Retrieved August 1, 2017 from http://worrydream.com/ExplorableExplanations/
45. Bret Victor. 2011b. Scientific Communication As Sequential Art. http://worrydream.com/ScientificCommunicationAsSequentialArt/.
46. Bret Victor. 2011c. Tangle: a JavaScript library for reactive documents. Retrieved August 1, 2017 from http://worrydream.com/Tangle/