Data Theater: A Live Programming Environment for Prototyping Data-Driven Explorable Explanations

Sam Lau
UC San Diego
La Jolla, CA, USA
lau@ucsd.edu

Philip J. Guo
UC San Diego
La Jolla, CA, USA
pg@ucsd.edu

ABSTRACT

Explorable explanations (a.k.a. 'explorables') enable readers to learn concepts in domains such as math, physics, and the social sciences by interacting with live visualizations. Despite their popularity, there is currently a high barrier to creating explorables since one must be adept at UI and visualization programming. To learn about these challenges, we interviewed 6 educators who were interested in explorables but lacked the skills to create them from scratch. These interviews gave us design insights to lower some of these implementation barriers. We used these insights to create a live programming system called Data Theater that enables programmers to prototype explorables by writing their simulation logic in Python and mapping Python values to visualization elements using a declarative JSON grammar. To demonstrate the capabilities of Data Theater, we used it to recreate two of Bret Victor's original physics simulation explorables and found that our approach can lower the barriers to prototyping explorables.

INTRODUCTION

Explorable explanations (often called 'explorables') engage learners by embedding interactive visualizations within textual explanations [1, 12]. For example, one popular explorable, *Parable of the Polygons* [6], asks viewers to interact with a simulation of triangles and squares to learn how large-scale social segregation can emerge from small individual biases. The recent popularity of explorables has motivated educators to create explorables for topics ranging from statistical histograms to voting systems. In a classroom, instructors can use live explorables in lessons as active learning activities; they enable students to pose and answer hypotheses in real-time [12].

However, there is currently a high barrier to implementing explorables, which makes it hard for many instructors to make them. Creating an explorable requires expertise in interactive UI programming using specialized visualization libraries [7]. *How can we lower the barrier to creating explorables for instructors who have only basic programming knowledge?*

ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

In our experience with implementing explorables, we found that even simple explorables require writing hundreds of lines of JavaScript. However, the amount of code corresponding to their core logic is usually an order of magnitude smaller than the total code. For instance, a representative statistics explorable within *Seeing Theory* [8] contains under 20 lines of logic to simulate dice rolls intertwined within 200 lines to set up UI event listeners and visualize results using D3 [3].

To surface opportunities to abstract away this boilerplate code, we conducted formative interviews with six instructors of statistics and data science courses. We asked the instructors to make pen-and-paper mockups of explorables based on their lecture slides. These interviews revealed that instructors do not actually need the high degree of UI flexibility to manipulate a simulation that current explorables provide. Instead, instructors were most often interested in visualizing steps of an algorithm, similar to how program visualization tools like Python Tutor [5] allow users to visualize execution steps of code. *These interviews suggest that code visualization techniques might enable instructors to quickly prototype explorables.*

Informed by these insights, we created *Data Theater*, a live programming system that allows instructors to construct custom explorables on top of Python code without requiring knowledge of user interface programming. Data Theater simplifies the explorable authoring process for instructors by separating the logic of an explorable from its visual representation, and by allowing only one simple interaction for manipulating visualizations: stepping through statements of Python code.

To demonstrate the capabilities of Data Theater, we used it to recreate two of Bret Victor's original physics simulation explorables [11] and found that Data Theater's approach can potentially lower the barriers to prototyping explorables.

RELATED WORK

Explorable explanations is an instructional format popularized by Bret Victor, Vi Hart, Nicky Case, and others [1, 6, 12]. Broadly it refers to web-based articles that interleave explanatory prose with interactive visualizations so that readers can actively engage with the material. Journalists sometimes refer to these as *interactive articles*, and notable data journalism teams at The New York Times, Washington Post, and other publications have published such articles for, say, exploring voting patterns [4]. Distill is a web-based academic journal that publishes interactive articles on machine learning [2].

Most explorables are now created manually as bespoke JavaScript visualizations using libraries such as D3, React, and other UI frameworks [4, 7]. To ease this process, developers have created specialized reactive frameworks such as Tangle [13] and Idyll [4], which allow UI widgets to be bound to event handlers in a concise declarative way. For instance, one can use these frameworks to easily create an HTML slider which live-updates the value of a numerical parameter that is then fed into a JavaScript-based visualization.

While the above approaches are powerful, we chose to take a complementary approach with Data Theater. Instead of building a reactive framework capable of making arbitrarily sophisticated user interactions, we leverage the fact that the instructors in our target audience (see next section) mostly wanted a much simpler user interaction: they wanted step-by-step algorithm walkthroughs augmented with their own domain-specific visualizations. Thus, we adopted the same metaphor as debuggers and program visualization tools such as Python Tutor [5], where each execution step represents one step in the explorable. We combine this metaphor with a declarative JSON visualization grammar inspired by Vega-Lite [9], which allows the creator to map Python values to custom graphical objects. Finally, to enable rapid prototyping, Data Theater is a *live programming system* implementing Tanimoto's Level-3 liveness [10], which updates the visualization whenever the Python code or declarative JSON visualization grammar is edited. This liveness enables rapid iteration and live coding in front of a classroom to adjust lessons on the fly.

FORMATIVE INTERVIEWS AND DESIGN GOALS

To understand why and how instructors may want to use explorables, we conducted one-hour interviews with six instructors who teach statistics and data science courses at a large U.S. university. Before the interviews, we asked each instructor to send us teaching materials from a representative lesson. During the interviews, we showed them examples of existing explorables related to their lesson topics, then asked instructors to make pen-and-paper sketches for an explorable they would ideally want to use the next time they taught that lesson.

Explorables as worked examples

Our instructors were most excited to use explorables as worked examples[1] during their lessons. Although instructors already included examples in their lecture slides, they found it difficult to predict how many examples their students needed to see before feeling like they understood the topic. This was especially relevant for simulations that relied on randomness. For instance, in one set of lecture slides an instructor simulated dice rolls to count the number of rolls before the first six appeared. When a student asked how the simulation would change if the dice were rolled until either a five or six appeared, this instructor wanted the ability to alter the simulation and re-run it multiple times but could not do this using his static slides. Other instructors also echoed this desire to adjust parameters of a simulation in real-time during a live lesson in response to student questions, citing this as a primary reason they would want to use explorables in their future lessons.

[1] https://en.wikipedia.org/wiki/Worked-example_effect

Barriers to creating explorables

Instructors wished to customize explorables just as easily as they could customize lecture slides. However, none of them knew how to use their existing programming knowledge to create explorables. And as non-specialists in building user interfaces, they thought their time would be better spent on making lesson materials than learning tools like D3 or React.

These instructors expected students to learn programming for the purpose of data cleaning, visualization, and modeling. To this end, they taught programming, ran simulations, and displayed data visualizations during lecture. Because their data analysis code also served as learning artifacts for students, they avoided cluttering up their code with details that they did not expect students to understand. Thus, even instructors who knew about packages for generating user interfaces like ipywidgets[2] and Shiny[3] chose not to use these packages because they did not want to mix UI code with data analysis code.

Guided walkthroughs, not freeform playgrounds

Looking at the sketches instructors made for desired explorables revealed a gap between the interactions that current explorables enable and what instructors wanted in explorables for their lessons. Most notably, explorables like *Parable of the Polygons* [6] are like freeform playgrounds—they give options to change simulation parameters through UI elements, but it is up to the reader to configure and run the simulation; for instance, there are multiple sliders and buttons to change the way the simulation generates and animates shapes. In contrast, our instructors desired to present explorables through *guided walkthroughs*—they wanted to explicitly guide students to pedagogically-meaningful outcomes by changing one parameter at a time. For example, one instructor sketched *separate* explorables for sample size, population data, and sampling method, deliberately choosing to create separate explorables for each parameter rather than a "kitchen sink" explorable.

Instructors also pointed out that, when presenting a lesson, simpler interfaces are better. One instructor liked the flexibility in the explorables we showed her but was concerned that she would forget how to manipulate the complex UI during a live lecture. Another mentioned that he would feel more comfortable using explorables in class if he had a slideshow-style interaction to "just press the spacebar."

Design goals

We synthesized these interview observations into three design goals for explorable authoring systems for instructors:

- **D1**: Decoupling of data and visualization: instructors should be able to specify the core logic of an explorable separately from the visual interface of the explorable.
- **D2**: Liveness: tools for authoring instructional explorables should automatically update visualizations to maintain consistency with data during the prototyping process.
- **D3**: Unified interaction: instructors prefer a simple and consistent interface for presenting explorables to students rather than a flexible yet complicated interface.

[2] https://github.com/jupyter-widgets/ipywidgets
[3] https://shiny.rstudio.com/

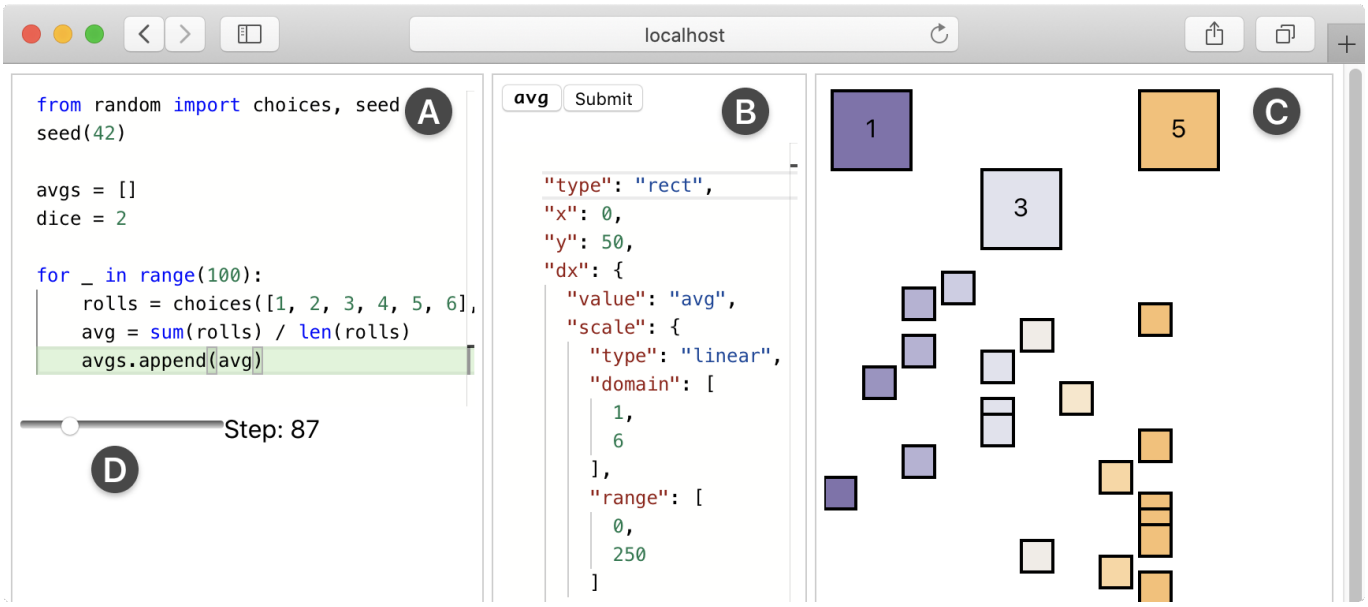


Figure 1. Using Data Theater to create an explorable that rolls two dice, calculates the average of the two rolls, and displays the history of all averages. (A) Instructors implement the core logic using Python code. (B) Instructors visualize the code through a declarative JSON spec that maps Python values to visual glyphs. (C) The visualization live-updates as either the Python or JSON change. (D) Explorables authored in Data Theater contain a single interactive element: a slider that steps forward and backward through execution steps and causes the visualization to animate as Python values change.

THE DATA THEATER SYSTEM

Informed by these design goals, we are developing a system called Data Theater to enable instructors to prototype explorables using Python and JSON.

Figure 1 shows an overview of Data Theater: (A) To begin creating an explorable, an instructor inputs a short Python program that contains the core logic of the explorable. For example, a programming instructor might input a program that implements insertion sort, while a statistics instructor might input a program that simulates coin flips. Because instructors usually already have these short demo programs written for their courses, Data Theater does not require them to change their Python code to add visualization elements. (B) Instead, an instructor specifies the visualization using a declarative JSON spec that maps Python values within the input program to visual attributes like position and size. (C) Data Theater combines the Python code and JSON spec to render a visualization based on the program's data. (D) To interact with the explorable, use a slider to step forwards and backwards through the program's execution, akin to single-stepping in a debugger; as the data of the program changes between steps, the visualization is automatically animated and updated live to match the data (Design Goal D2).

To show the explorable authoring experience, we describe how a hypothetical instructor named Ani creates a simple explorable that teaches statistics by simulating dice rolls.

Step 1: Inputting Python Code

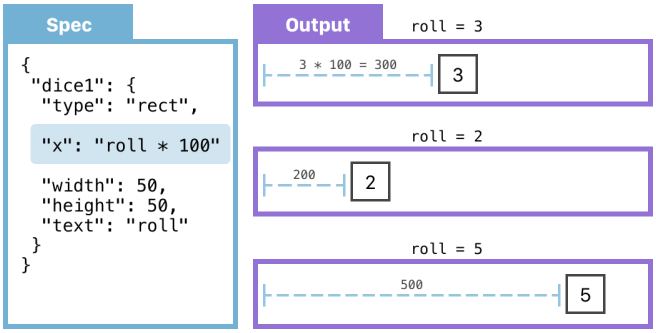
Ani opens the Data Theater web app and copy-pastes her Python code into the left editor pane (Figure 1a). For a previous offering of her introductory statistics course, she has already written Python code to simulate dice rolls to show the central limit theorem. This code rolls a simulated dice 100 times, stores each roll's result (a random integer from 1 to 6) in a `rolls` variable, and appends the running average roll result to an `avgs` list. Without Data Theater, she would normally have printed out the results to a terminal (which is neither visual nor interactive) or needed to write extra code using a graphing library to plot the distribution of roll results.

Using Data Theater, she simply copies that code into the left editor pane and specifies the visualization separately in the next step. That way, her Python code concisely demonstrates the core logic and is not obfuscated by excess boilerplate to set up interactive visualizations (Design Goal D1).

Step 2: Mapping Python values to visual attributes

Ani writes a JSON spec in the middle pane (Figure 1b) to tell Data Theater how to turn her Python code from Step 1 into a visualization. Data Theater provides a declarative JSON-based grammar modeled after Vega-Lite's grammar of interactive graphics [9]. Every spec is composed of one or more glyphs which represent visual objects. Similar to Vega-Lite, Data Theater's grammar supports glyphs for rectangles, ellipses, lines, and text; it also has a collection glyph for grouping.

The diagram below shows a simple demo where Ani creates a `dice1` visualization element as a rectangle glyph (`"type":"rect"`). She sets the width and height to 50 to make it a square. The grammar allows Ani to map the values of Python variables to visual attributes like x-coordinate, y-coordinate, width, height, and color. In the spec below she adds `"x": "roll * 100"`, which maps the x-coordinate of the glyph to the value of the Python `roll` variable multipled by 100. She also sets the text label of the glyph with `"text": "roll"`, which sets the label to the roll value:

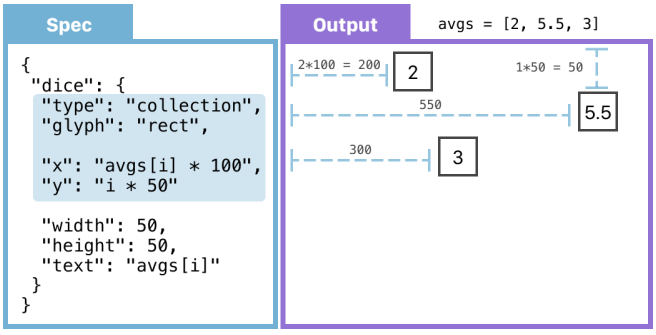


Data Theater parses the spec and renders its result to the output pane, in this case a single rectangle. The visualization always stays in sync with the value of `roll` as the Python code executes. In the above diagram, when roll is 3, the dice's x-coordinate is 300 (`3*100`) and its label is '3'. As the program executes and more random rolls are performed, the rectangle changes positions and labels based on its value (see Step 4).

Note that in this example, `"x": "roll * 100"` constructs a linear scale. Besides manually creating scales with Python expressions, Data Theater supports all scales provided by the D3 library [3]. For example, Ani can use a color scale from D3 to set the rectangle's color based on Python numerical values.

Step 3: Mapping Python collections to visual attributes

Aside from mapping scalar values (e.g., numbers) to visual attributes, Ani can also write JSON to map an entire collection (e.g., a Python list or NumPy array) to visual attributes. The diagram below shows how she can specify how to visualize the entire `avgs` list. She creates a 'collection' type, maps it to a 'rect' glyph, and specifies the x and y coordinates again using Python expressions inside the JSON spec:



For a collection glyph, the special 'i' index variable iterates over all elements in order. The output pane above shows what is rendered if `avgs = [2, 5.5, 3]`. Three rectangles are rendered on-screen, one for each element of `avgs`. The x-coordinate of each is 100 pixels times the element's value (`avgs[i]*100`), the y-coordinate is 50 times the index (`i*50`), and each has a text label with its value (`"text":"avgs[i]"`).

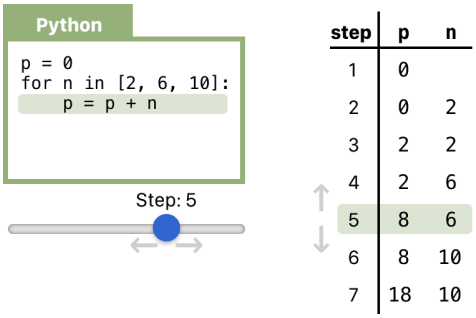
Step 4: Interacting with the prototype explorable

So far we have shown how Ani can map Python code into visual attributes with her JSON spec. This is reminiscent of creating a static data visualization. However, a defining trait of explorable explanations is the ability for users to *interact* with the explorable. Where does this interactivity come from?

Creators of popular explorables [7] implement such interactivity by hand-coding JavaScript event handlers for direct manipulation interactions such as mouseovers, mouse clicks, or drag and drop, and by creating UI widgets such as sliders that are tied to other on-screen elements. However, we found in our formative interviews that instructors did not want to face the steep learning curve of hand-coding such general-purpose interactivity. Thus, we opted to implement a more limited form of interactivity in Data Theater, which matches how instructors often conceptualize animated explanations as step-by-step guided walkthroughs of algorithms (Design Goal D3).

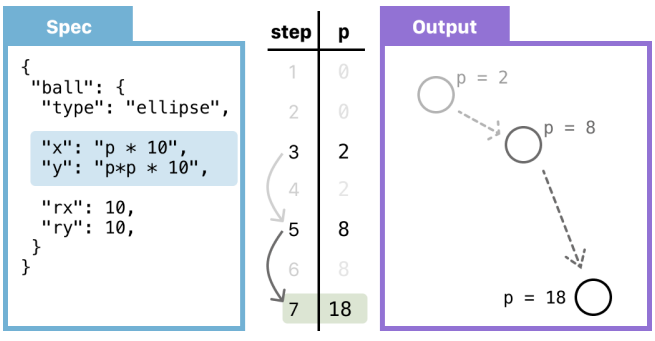
Data Theater provides interactivity via an execution step slider below the code editor (Figure 1d). When the user inputs code, it is executed and all of its run-time values at each step are recorded in a Python Tutor trace [5]. The slider allows the user to scrub back and forth through all execution steps. As the user scrubs the slider, the output pane shows the visualization *with glyph elements corresponding to the values of program data at that step*. For instance, as Ani's example program executes, the `avgs` list gets more and more items appended to it, so the output pane renders all of those items as they enter the list.

To demonstrate the slider at work, consider the small program below that defines two variables 'p' and 'n':



As the user scrubs the slider, 'p' and 'n' take on the respective values at each execution step as shown in the table on the right. By mapping 'p' and 'n' to glyphs using JSON like we will do next, the user can visualize them in arbitrary ways.

Finally, to enable explorables to animate smoothly, whenever an updated Python value causes a visual attribute of a glyph to change, Data Theater will automatically animate the transition. The spec below defines a circle glyph (an 'ellipse' with 10-pixel radius) and maps its x and y coordinates to the values of the Python 'p' variable defined in the code above:



As the user scrubs the slider to step through execution, 'p' takes on values such as 2, 8, and 18. As this happens, the circle in the output pane smoothly animates to move in between the respective x and y coordinates corresponding to the 'x' and 'y' values as indicated by the JSON spec. This allows an instructor to, say, create a physics simulation explorable showing a bouncing ball affected by different gravity settings. (Without smooth animation, it would simply look like the ball disappears and reappears elsewhere, which is jarring.)

From animated visualizations to explorable explanations

In this section we have shown how Data Theater enables instructors like Ani to take Python code, augment it with a JSON spec, and use an execution slider to animate their data-driven visualizations. However, note that an animated visualization is only one part of an explorable explanation; the other key component is the explanatory prose itself that surrounds the visualization. Since Data Theater visualizations are implemented as HTML/CSS/JavaScript web technologies, they can be embedded within web-based presentation slides or articles where authors can write the surrounding explanations. In our example, Ani would likely string together a few Data Theater visualizations of dice rolls to progressively build up more complex ones to show statistics concepts such as the central limit theorem. Since Data Theater is a live programming system, she can build up those examples live in class, step through them, and tweak the code as she makes verbal explanations.

PRELIMINARY CASE STUDY: RE-CREATING BRET VICTOR'S SKATEBOARD EXPLORABLES

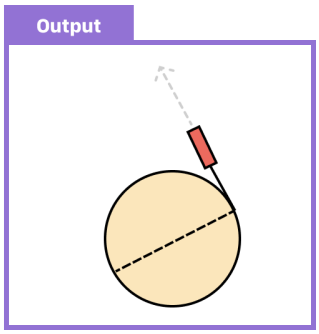
As a preliminary case study to demonstrate Data Theater's capabilities, we recreated the interactive visualizations that power two explorables from Bret Victor's widely-read *Simulation as a Practical Tool* web article [11]. Each took fewer than 20 lines of Python to replicate. These explorables explain a physics simulation where a skateboarder launches from a rotating disk towards a wall. The first explorable in the article simulates the skateboarder traveling towards the wall, and the second one calculates the time it takes the skateboarder to hit the wall based on different initial launch angles.

Explorable 1: skateboard path simulation

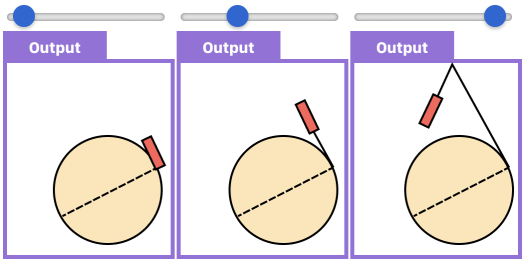
To recreate the first explorable, we begin by defining Python variables that record the disk position, disk rotation, initial skateboarder position, and initial skateboard velocity. Then we write a loop to simulate the skateboard moving forward as time elapses. In total, the Python script is 18 lines of code (11 lines to initialize variables and 7 lines for the simulation). Data Theater allows us to focus our Python code on the essence of the simulation instead of tangling it with visualization code.

We then specify the JSON visualization spec, which contains three glyphs: an ellipse for the disk, a rectangle for the skateboarder, and a line showing the skateboarder's path over time. To create the ellipse glyph, we map Python values of the disk object to x-position, y-position, radius, and rotation. To create the rectangle, we map Python values of the skateboard object to x- and y-positions. To create the line glyph, we map Python values of the history of skateboard positions to the x and y-values for the line. In this example, all values are mapped directly without scaling; the final visualization spec is under 25 lines long. The diagram below shows a summary of the JSON spec and example output at one execution step:

variable	description
`disk`	position of disk
`r`	radius of disk
`rotation`	rotation of disk
`x`	x-pos of skateboard
`y`	y-pos of skateboard
`xs`	all previous x-pos
`ys`	all previous x-pos



Since the Python simulation code updates the skateboarder's position as execution progresses, we animate this explorable by using the slider to step forward. As we step through execution, the skateboarder (orange rectangle) launches off the disk, heads straight to the wall, and bounces off at the same angle:



By tweaking Python variables, we can alter the physics of the simulation. This interactive visualization now matches the behavior of Victor's original explorable, except the graphics are not as polished since they are simple shapes; allowing the user to import sprite graphics could make this higher-fidelity.

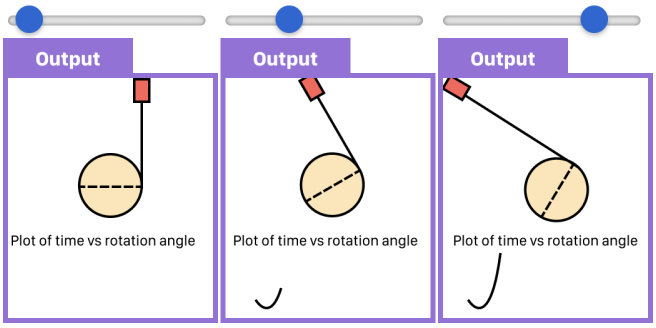
Explorable 2: initial disk angle simulation

The second explorable in *Simulation as a Practical Tool* [11] builds on the first one by allowing the user to change the initial rotation of the disk and observe how the trajectory of the skateboarder changes. It also uses a line graph to plot the time taken to reach the wall for different starting rotations.

To recreate this explorable, we modify our Python script to iterate over disk rotations rather than over time. For each starting rotation, we record the final x- and y-positions of the skateboarder and the amount of time it takes to reach the wall. The final Python script is 17 lines long, with many lines copied verbatim from the original script.

The visualization for this explorable contains the same glyphs as the previous one and with an additional line glyph to plot the time taken to reach the wall. To visualize the disk, skateboarder, and trajectory, we copy the glyphs from the previous explorable. We use them nearly verbatim with minor modifications to account for changed Python variable names. To visualize the time taken to reach the wall across rotations, we create a line glyph and map the rotations to x-values and times to y-values. This JSON visualization spec is 38 lines long.

The diagram below shows three example steps through the simulation as the user scrubs the slider. Note that the line plot at the bottom gradually builds up as the user steps through different rotation angles, which matches Victor's original:



Comparing Data Theater to original explorables

Reflecting on our case study experiences, we found it intuitive to recreate simple physics simulations such as these as long as we adopted the mindset that Data Theater uses execution steps as a 'one-dimensional slider.' In the first explorable, we used the slider to scrub through time steps, and in the second one, we used it scrub through initial disk rotation angles.

One limitation to Data Theater's 'one-dimensional slider' interaction model is that it does not have the capability to exactly replicate the richer interactions in the original explorables. For example, the first explorable plays a sequence of steps as a movie and the second rotates the disk based on the user's mouse position. Data Theater's only interactive UI element is the execution slider to step through Python code. That said, an instructor can use the slider to approximate these interactions. For the first explorable, moving the slider forward at a steady rate animates the visualization like a movie. For the second, moving the slider back and forth creates a similar animation as moving the mouse left and right in the original explorable.

Directly comparing the amount of code we wrote to Victor's original code is hard since his explorables are implemented in Adobe Flash (which is now unsupported in modern web browsers) without available source code. Based on our prior experiences, we estimate that fully recreating these two explorables using JavaScript libraries would take at least a few hundred lines of code. In contrast, our implementation of these two explorables in Data Theater is less than a hundred lines of code combined.

CONCLUSION

We have presented Data Theater, a live programming system that combines step-by-step Python execution with a JSON visual grammar to let instructors quickly prototype explorable explanations in the browser. The key simplifying insight that enables our approach is to eliminate extra degrees of interface design freedom by using the execution slider as a unified way to interact with the explorable. In closing, although our approach can greatly reduce boilerplate code, it still cannot eliminate the intrinsic complexities of writing code to express core simulation logic and writing suitable prose to encapsulate it in a high-quality explanation. We are currently investigating ways to provide better support for these ongoing challenges.

Acknowledgments: This material is based upon work supported by the National Science Foundation under Grant No. NSF IIS-1845900.

REFERENCES

[1] 2014. Explorable Explanations, a hub for learning through play! `https://explorabl.es/`. (2014). Accessed: 2020-09-17.

[2] 2018. Publishing in the Distill Research Journal. `https://distill.pub/journal/`. (2018). Accessed: 2020-09-17.

[3] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. 2011. D3: Data-Driven Documents. *IEEE Transactions on Visualization and Computer Graphics* 17, 12 (Dec. 2011), 2301–2309.

[4] 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 (UIST '18)*. Association for Computing Machinery, New York, NY, USA, 977–989.

[5] Philip J. Guo. 2013. Online Python Tutor: Embeddable Web-based Program Visualization for CS Education. In *Proceedings of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13)*. ACM, New York, NY, USA, 579–584.

[6] Vi Hart and Nicky Case. 2014. Parable of the Polygons: A playable post on the shape of society. `https://ncase.me/polygons/`. (2014). Accessed: 2020-09-17.

[7] Fred Hohman, Matthew Conlen, Jeffrey Heer, and Duen Horng (Polo) Chau. 2020. Communicating with Interactive Articles. `https://distill.pub/2020/communicating-with-interactive-articles/`. (2020).

[8] Daniel Kunin. 2020. Seeing Theory: a visual introduction to probability and statistics. `https://seeing-theory.brown.edu/`. (2020). Accessed: 2020-09-17.

[9] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. 2017. Vega-Lite: A Grammar of Interactive Graphics. *IEEE Transactions on Visualization and Computer Graphics* 23, 1 (Jan. 2017), 341–350.

[10] Steven L. Tanimoto. 2013. A Perspective on the Evolution of Live Programming. In *Proceedings of the 1st International Workshop on Live Programming (LIVE '13)*. IEEE Press, 31–34.

[11] Bret Victor. 2009. Simulation as a Practical Tool. `http://worrydream.com/SimulationAsAPracticalTool/`. (2009). Accessed: 2020-09-17.

[12] Bret Victor. 2011. Explorable Explanations. `http://worrydream.com/ExplorableExplanations/`. (2011). Accessed: 2020-09-17.

[13] Bret Victor. 2013. Tangle: explorable explanations made easy. `http://worrydream.com/Tangle/`. (2013). Accessed: 2020-09-17.