Constructing Mixed-Initiative Tools for Domain Experts from Rich Input Widgets

Jasmine Tan Otto

Oct 8th, 2021

0.0.1 Abstract

This review collects literature relating to mixed-initiative tools used by domain experts to study complex systems through models, and to share knowledge through their work practices. Examples are drawn from the domains of systems thinking, robotics planning, and data visualization. It discusses the wicked problems shared by software tools that support creative work, and characterizes the building blocks of mixed-initiative software tools, by explaining D3.js and other *model engines* (by analogy to game engines). This building-block approach complements participatory design methods, which address the expert's understanding of their model, by supporting the expert's manipulation and rearticulation of their model. This review advances a novel conception of *rich input widgets* comprising a graphical interface, which represents system behavior by connecting domain-specific models.

0.0.2 Introduction

Mixed-initiative tools are used by domain experts (people with expert knowledge and skills) to study and manipulate complex systems. Ant colonies are complex systems, as are Mars rover operations. Experts use instrumentation to sense the system's output (like a telescope) and to formulate its input (like an oscilloscope dial). A well-made *software instrument* combines these functions in an intuitive graphical interface.

This review collects literature relating to software instruments, which function as *boundary artifacts* in work practices, today and tomorrow, in the contexts of both creative professions and scientific computation. Its examples of software instruments are drawn from systems thinking, as well as robotics planning, computational fabrication, data visualization, and procedural generation.

A software instrument is made of multiple widgets, each of which shares some model. This document looks at specific widgets in its case studies, and includes an explanation of Data-Driven Documents (D3.js) and other *model engines*. A software instrument is for domain experts, including not just creatives, but also scientists. Because scientists are designers, they both encounter and generate wicked problems. As such, instruments tend to slot into existing workflows.

A model is necessarily evaluated (by seeing what it does) and executed (by shaking the model around), which can both be difficult steps. Where participatory design methods in the UX of scientific computing address the *gulf of evaluation*, rich input widgets address the *gulf of execution*.

Rich input widgets are elements of a software interface that combine a visualization component with a set of operations. This document builds up to a definition for "rich input widgets" that will be presented in this document's conclusion. And an instrument is a language, which is scaffolded by an interface.

This review advances the idea that, like natural languages, software instruments compose meaning from structured parts, and we need to better understand their vocabularies and grammatical rules. At the same time, these languages are constrained by the set of utterances an instrument can produce, including fewer which are incorrect, effectively increasing the skill of their user. It is interesting to study which languages are effectively constrained and which are not.

0.1 Case Studies

0.1.1 Loopy and the Ladder of Abstraction

Complex systems are made available through their models in diagrams on whiteboards, which capture facets of their dynamic representation in an expert's mind. What if we could write down the simplest dynamics, and communicate them through computers?

i. Bret Victor and Nicky Case, as part of the *explorable explanations* community, have explored this problem through a distinctive hybrid of interaction design (which is itself an emerging field of computational media) and complex systems thinking.

Up and Down the Ladder of Abstraction [36] (Figure 1a) is Bret Victor's interactive essay about breaking a complex model down into a series of interactive diagrams, each one consisting of a few small, reusable widgets. The diagrams are presented in the order of increasing abstraction in the model (i.e. the addition of new parameters, each adding a dimension of complexity to the diagram). Victor realizes an intuition pump for S.I. Hayakawa's ladder of abstraction [1] in the context of data visualization and interactive scientific article writing.

Loopy [6] is Nicky Case's visual editor for dynamical systems, intended for research dissemination and education. It enables direct manipulation of a graphical model (such as the cyclic influence of depression on de-motivation illustrated with Loopy in figure 1), which characteristically occurs in the complex systems literature, intended to accommodate cyclic forms of cause and effect.

ii. Both of these approaches address a non-expert audience, because their common goal is to make a given model approachable to more people more easily. In

[1] See this gloss from Megan Bush Moody's recent course on Writing and the Sciences. https://eatingenglish.community.uaf.edu/the-ladder-of-abstraction/

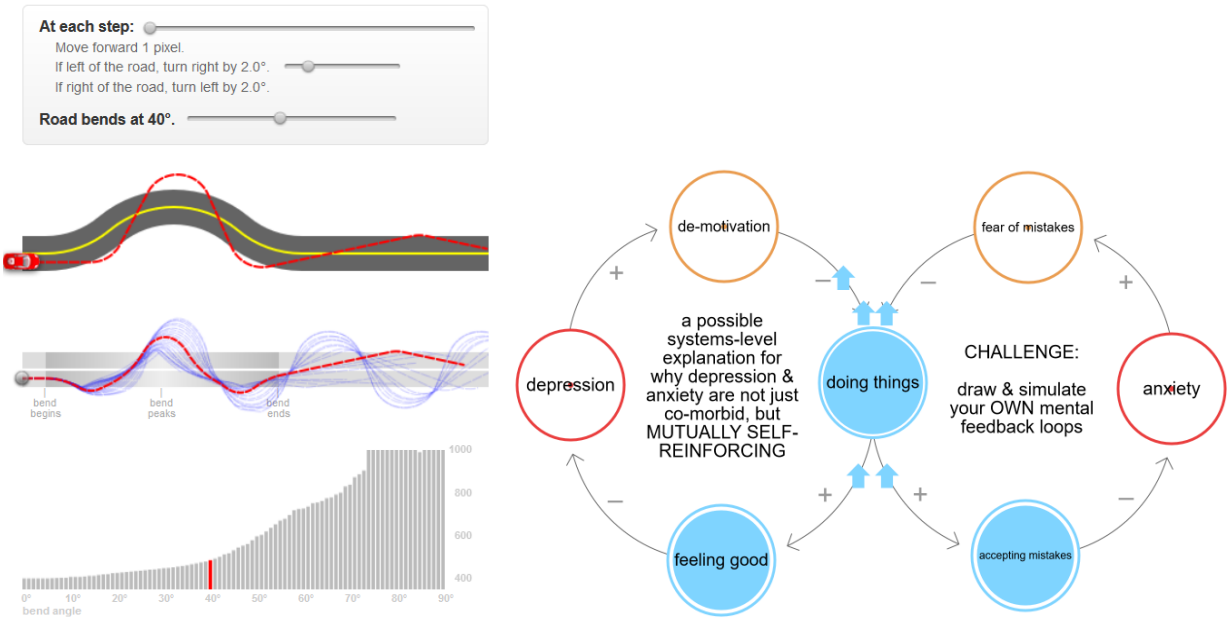

Figure 1: (a) Up and Down the Ladder of Abstraction, one of its self contained interactive diagrams. (b) Loopy, one example of a complex system with cycles.]

real-world projects, this is also a useful goal, because not all stakeholders in a given project are domain experts in every model involved in the project.

Human users in general are highly competent in integrating visual information to deduce cause and effect. People can read diagrams and understand that different entities in the (high-dimensional) system are represented by certain elements of the (two or three-dimensional) diagram. Indeed, people readily and typically ascribe complex feelings and motives to the entities described by certain diagrams of bodies in motion, as in the Heider-Simmel animation [15].

On the other hand, the gaps between the on-screen representation of an entity, its mental realization according to the human user, and its characterization according to the computer model, are always challenging to bridge. (Consider the Eliza effect [37], where a simple model is understood as a complex interlocutor. Also the SimCity effect, where a simple model creates a false confidence that the real-world system it represents is also simple.)

Bret Victor and Nicky Case are both aware of this challenge to communication, and they take opposite approaches. The Ladder of Abstraction uses the example system of a car on a road with a bend in it. The steering behavior of that car, and the degree to which that road bends, determine a family of trajectories that the car might take (many of which fail to clear the bend at all). According to Victor, the intuitive characterization of this family requires not only familiar abstractions such as bar charts and heatmaps, but also semi-literal

representations, such as a road with a 'variable-degree' bend superimposed with the possible trajectories of a car depending on that bend.

Loopy leans into the variety and breadth of real-world interpretations of complex system dynamics, offering lots of text-authoring tools in addition to system-editing tools. The nodes of a Loopy graph are variables in the dynamical system, and their edges are direct or reciprocal relationships from one variable to the next.

But Loopy users are not assumed to be familiar with dynamical systems - because of this, Nicky Case decided not to present the continuous-time solution of the system (which yields a desirable closed-form solution, but is quite challenging to explain), and instead to emit 'units of change' as visible particles which travel around a circuit. Thus, Loopy is a rate-discretized approximation to a system of ordinary differential equations. Units of change can be visually tracked around the circuit of relationships between variables, which the conventional formalism prevents. This is a reasoned design trade-off for increased abstraction (conventionally) versus accessibility of representation (in Loopy).

iii. While Up and Down the Ladder of Abstraction explores an open-ended family of rich input widgets regarding a single concrete model, Loopy realizes a set of rich input widgets on a single abstract model with an exceptional variety of applications.

Each interactive diagram at a certain level of the Ladder of Abstraction involves a few linked rich input widgets, and a slider panel which also exposes each raw parameter. E.g. the 'stepping back down' diagram contains three widgets: (a) car behavior over time, (b) car behavior over time and environmental parameter, and (c) quality of car behavior over environmental parameter.

Up and Down the Ladder of Abstraction is an interactive essay about explaining complex systems models. It demonstrates a strategy for linking interactive diagram, to ratchet up one dimension of abstraction in each subsequent widget, forming an effective intuition pump within the browser-like medium of interactive figures.

The Loopy editor is filled with rich input widgets, including both the nodes themselves, and a set of tools for adding, removing, or changing the length of edges. Node values can be manipulated directly, emitting appropriate rate events, or their initial value and color can be set using an inspection tool.

Loopy is a visual editor for dynamical systems. It can be used to model real-world systems with cyclic forms of cause and effect.

0.1.2 Deep Space Robotics

RSketch [22] is a tool for rover planners to command rovers on Mars based on the Martian landscape, which can cause the rover to roll past its mark or even to tip over. Mars rovers are driven with low-level commands, which are both executed (during the operations window on Mars) and evaluated (after return flight time) on a large time delay.

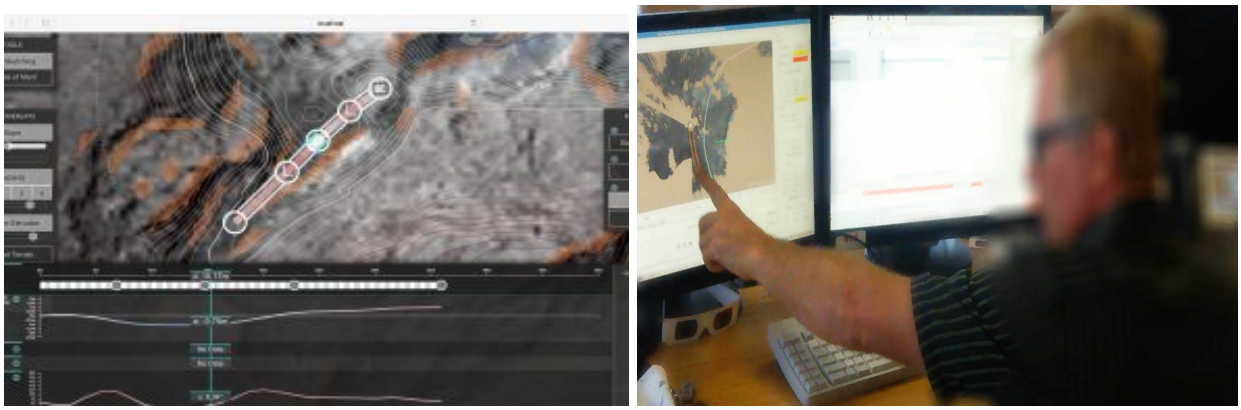

Figure 2: RSketch path planning interface, and a rover operater interacting with the planner.

The Deep Space Network (DSN) Postage Stamp [17] is a situational awareness tool for operators of a worldwide telecoms network used by NASA, which routes mission data between operation centers and spacecraft. This workload is highly 'bursty', meaning that most activity happens during only a short period of the operator's work shift.

Rover planners are able to assess what level of risk a plan poses to the rover's survival and its ability to complete a mission in a certain timeframe. In particular, driving the rover over terrain that is too rough may cause it to roll past (or short of) the intended destination, or to tilt so much that it falls over. The RSketch prototype allows the rover planner team to rapidly evaluate a series of possible routes by seeing what impact the local terrain has on their ability to execute.

Deep space network operators are responsible for continuously maintaining the ability to route information through a complex system of hardware and software distributed around the world. They use their expert knowledge and intuitions to detect, locate, and correct component failures (located somewhere upstream in a complex causal chain) while the system remains online. Operators support projects, which do not have DSN domain expertise, but may request technical support from the operators.

The map panel of RSketch (see Figure 2, reproduced from Alex Sciuto[2]) is a rich input widget. A series of rover commands is represented as a path with nodes on a map of the Martian terrain, which can be directly manipulated. The anticipated path and roll of the rover is visually superimposed on this intended path.

The postage stamp uses the same set of annotations as operators have learned to use on their key artifact, a sequence of events. Since these correspond to discrepancies between the sequence of events data and the live equip-

[2]http://www.alexsciuto.com/design/rsketch/

ment telemetry, which can be calculated automatically, the postage stamp is functionally an output, which the operator reads off. This source of situational awareness feeds into other components of the operator workflow involving kinds of query formulation, a pattern which is supported by rich input.

In general, diagram-like *boundary artifacts* (to be described in Section 0.3.4) like the sequence-of-events (in DSN operation) and rover plans (as built in RSketch) must support annotation. In the absence of the software, the operators and planners would solve their tasks and create consensus by marking up the printed plan with highlighters in a room together.

0.1.3 Domino and Draco

Domino and Draco are two complementary mixed-initiative tools to support visualization design by domain experts. In this section we distinguish between the relationship of Domino to a complex dataset (of multiple tables, each containing one kind of record) and the relationship of Draco to a specific table (yet exploring multiple kinds of chart).

The purpose of Domino [12] is to explicitly represent the relationships between subsets of data (groupings and mappings) in a direct manipulation interface (figure 3). The expert then benefits from a modular representation with respect to their data sources.

Draco [26] is framed as a 'visualization spell-checker' in a basic sense, and a tool for navigating the space of possible visualizations of a given dataset in general. (See figure 4 and live demo [3].) Draco can conceptually be mapped to a grid of possible outputs, which are gradually ruled out as the expert adds constraints (see in the Draco paper Figure 4 and Figure 2 respectively).

Both tools provide expert visualization support to domain experts in some non-visualization field of practice, be it a scientific discipline such as genomics, or another domain of expertise. They bridge a gulf between the question the expert intends to answer (in their own mind or an interlocutor's), and the visual form that the computer graphics system understands it should fill out.

When communicating quantitative data to users and stakeholders in an organization, the data is typically visualized using a kind of chart, which can be decomposed into *marks* bound to *columns* of the dataset under appropriate *groupings of rows* and *mapped columns*. Human experts are trained to understand what transformations on the dataset can achieve a meaningful visual representation, and they must be included in the workflow of making a meaningful chart.

These experts are unlikely, however, to describe a chart from the first principles of data-driven documents (see our discussion of D3 in 0.4.4) - as a collection of marks such as rectangles, circles, lines, and text. Visualization researchers have worked with domain experts in representing their data with the computer, by producing co-creativity support tools, for a long time. Draco's initial ontology, the APT visualization ontology, was first presented in 1986.

[3] https://uwdata.github.io/draco-editor/\#/editor

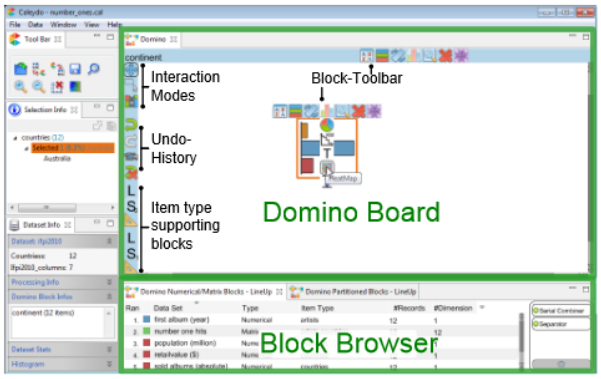

Figure 3: The user-level interface of Domino. Reproduced from [Domino/Fig.7].

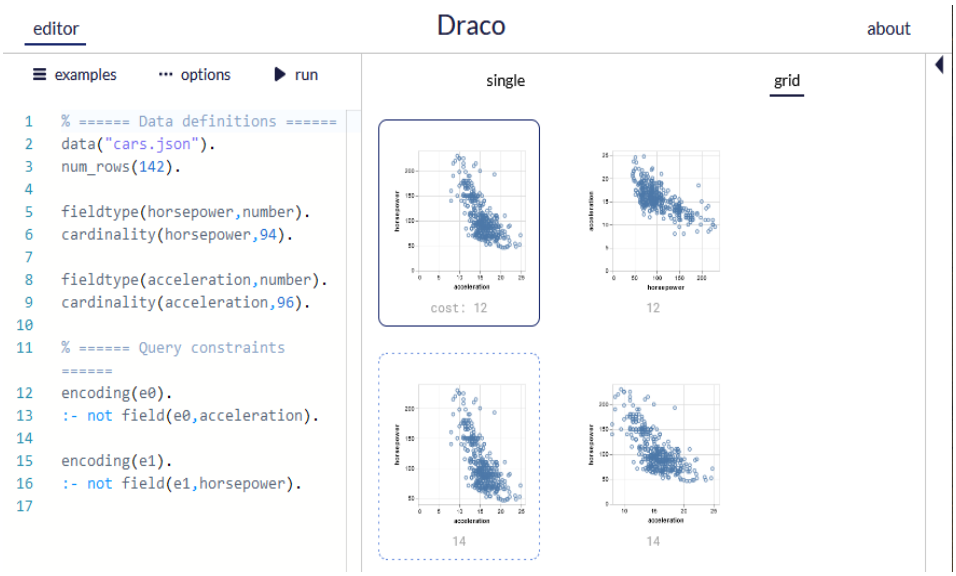

Figure 4: The user-level interface of Draco. Reproduced from the live demo.

Figure 5: Example from the snap-together interface of Scratch.

But domain experts do not use the newest possible visualization design support tools, in practice. Adoption occurs when the tool appears in a software ecosystem that the expert is already using (such as those of R, SciPy, and modern Javascript).

In a **spreadsheet** (such as Excel), suppose a user wants to plot their data. Their plotting grammar in this context is implicit to a wizard, i.e. a hardcoded script with menu-driven parameters. But the highly specialized charts used by domain experts are difficult to produce with these domain-agnostic plotting scripts. Then a specialized software package (such as ArcGIS, or an R library) may be substituted for the spreadsheet software, on an ad hoc basis.

A more modular solution might use Domino to 'slice down' a dataset (from an arbitrary set of data sources) with multiple related tables. Domino supports *multiple coordinated view* approaches for these complex problems. From an interface perspective, it supports *linked brushing* between multiple plots arising from the same dataset. This means it can pass a selection (of particular rows of data) between a number of plots, and that this selection can be specified by the expert through direct manipulation (rather than a query language).

The authors of Domino note that future work on Domino could include adding support for additional plotting grammars. Note that the complexity of the grammar includes not just describing the marks of those plots, but also making them reactive to pointer events, in order to support the exploratory use cases enabled by linked brushing.

Alternatively, in a **computational notebook** (such as Jupyter), chart grammars are typically implicitly (as in ggplot or matplotlib) or explicitly specified (as in Vega-Lite or Observable Plot), i.e. forming a domain-specific language.

The function of Draco is to cut down the space of all possible chart specifications in response to a query by the expert user on behalf of the data shapes they intend to plot.

This input space (constraint space) consists of a finite set of predicates with arguments like 'a column of the dataset', 'a datatype such as quantitative, ordinal, or nominal', or even 'an operation such as the mean of values sharing a certain key'. It is a variable-length list of strongly typed statements.

Compare this to the snap-together interface of Scratch (Figure 5), whose IDE is aware of its own API and constrains inputs accordingly, thus bridging the gulf of execution.

The Vega-Lite Editor [31] is the clearly intended analogue to the Draco

Editor. By flattening the hierarchical grammar of Vega-Lite, a Draco spec always contains just one scope, which makes it particularly well-suited to a snap-together interface. Or similarly, to a code completion recommender, which knows the type of each 'hole', and therefore the finite set of variables or constants that could fill in.

Overall, there is a great breadth of space in visualization design for mixed-initiative tools to fill in. Further work is necessary to make the tools highly accessible, as well as to make them well-integrated with existing workflows (whether involving data science notebooks, or directly from existing spreadsheets). Tools such as Domino and Draco, if made ubiquitous, will contribute both to expert productivity and to overall data literacy.

0.1.4 Conclusion

In general, the effectiveness of design support tools is limited by the intrinsic difficulty of defining the problem(s) they address. This is common to software tools built for domain experts, and we discuss it in the next section.

0.2 Wicked Problems

0.2.1 Design and Wicked Problems

Richard Buchanan [5] uses "wicked problems" for problems which are not tractable to linear models - in which problem definition is followed by problem solution, - that assume the problem is *definite*. Rittel and Weber [30], in coining the term, give these negative examples:

> The problems that scientists and engineers have usually focused upon are mostly "tame" or "benign" ones. As an example, consider a problem of mathematics, such as solving an equation; or the task of an organic chemist in analyzing the structure of some unknown compound; or that of the chessplayer attempting to accomplish checkmate in five moves. For each the mission is clear. It is clear, in turn, whether or not the problems have been solved.

A common wicked problem is in gleaning insight from some number of datasets. This kind is specifically addressed by the Domino system (see section 0.1.3), which graphically instruments a pipeline of relationships, filters, and charts.

All kinds of wicked problems frequently arise within interdisciplinary collaboration. Buchanan describes four disciplinary areas with which design is concerned: how a sign speaks, how a thing is used, how an action is completed, and how we think together.

> The doctrine of placements will require further development if it is to be recognized as a tool in design studies and design thinking, but it can also be a surprisingly precise way of addressing conceptual space and the non-dimensional images from which concrete possibilities emerge for testing in objective circumstances.

Buchanan quotes the designer Ezio Manzini's proposal in 1989 to arm the designer with a 'microscope and a macroscope'. This aptly describes S.I. Hayakawa's *ladder of abstraction* as Victor abducts it, a description of professional tools and learning tools alike (see section 0.1.1).

Buchanan uses *placement* (the specific) and *categories* (the general) as the poles of his application-versus-theory dichotomy in designerly practice. This document argues from the perspective of domain-specificity, i.e. placement, that good design flows from the ground up. Yet we sit in conversation with an emerging canon of categories - including those of hypertext theory, languages of games and play, and casual creativity support tools.

Each software instrument is a computational medium, capable of carrying and expressing messages in its own data representation(s). According to Timothy Binkley [1], a theorist of digital media:

> Now the distinction between [things and ideas] is not so clear as our creations become agents.

That is, the distinction between *mental models* and *data representations* is not marked - although we will see that the *gulf of evaluation* and the *gulf of execution* between them can be quite large.

0.2.2 Gulf of Evaluation

Collaboration with the computational medium is made difficult by the space between mental models and data representations, in either direction: the gulf of evaluation (in reading the representation), and the gulf of execution (in writing the representation), pictured in Figure 6. In 1986, Ed Hutchins et al. introduced these terms in their paper on *direct manipulation interfaces* [18]. Kate Compton draws on their account of loopy, enactive interfaces to computation in her dissertation [7][4].

i. When we say that direct manipulation bridges the gulf of evaluation, we are describing an act of data visualization (sonification, physicalization, etc.), which forms a read-only computational medium. When a data representation includes some form of input that bridges the gulf of execution (as in data analytics, or player control), that computational medium becomes expressive and reflective.

The mental model (a point in 'expectation space') and the data representation (a point in 'algorithmic space') are brought into alignment through a trial and error process, which is slowed by the gulf of execution (how can I change the data?) and the gulf of evaluation (how did I change the data?).

No wicked problem can be defined (evaluated) without its solutions in mind. In this sense, the solution must precede the problem. This is why the interaction loop is necessary, rather than a linear checklist. The expert user revisits their data again and again to deduce what is 'really the problem' (and 'really the solution').

[4] Also available in zine format: http://www.galaxykate.com/pdfs/galaxykate-zine-casualcreators.pdf

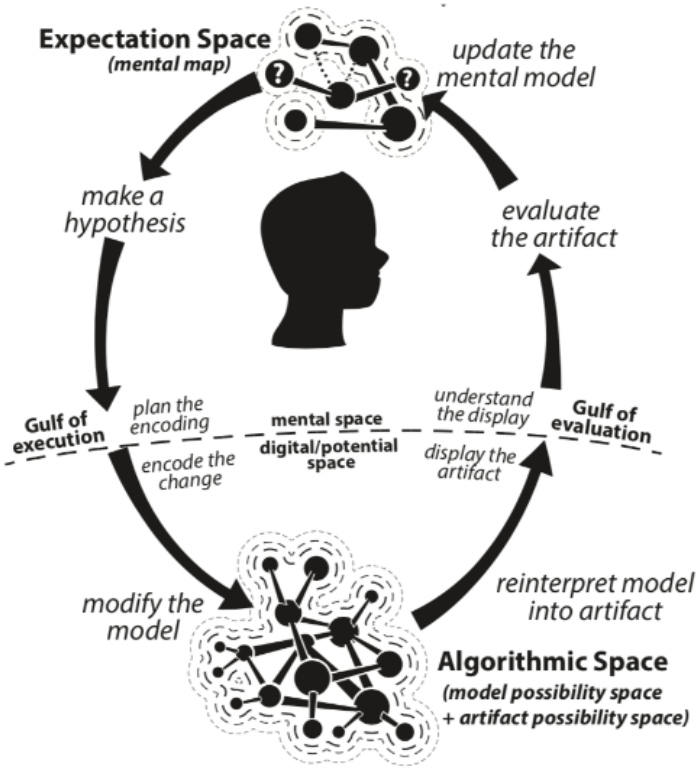

Figure 6: Kate Compton's rendition of the gulf of evaluation and the gulf of execution. Reproduced from their Casual Creators dissertation zine.

A typical example of deduction with expert knowledge is attempting to debug a piece of code based on some combination of re-running the code (to see what kinds of errors happen) and re-writing it (to locate the source of certain errors and to remove them). In compiled programming languages, the compiler acts as a co-creativity support software by flagging type errors prior to runtime. (We revisit this problem in the context of languages which are not compiled, like Python and Javascript, in section 0.4.5.)

ii. The emerging field of *casual creativity support tools*, also known as casual creators, engages deeply with rich input methods as a means of scaffolding models to expose them to more diverse users. Many of those users are in fact deeply involved in their craft, be it computational fabrication, or procedural poetry, or other examples as gathered by Compton [7]. Although these are crafts which are deprofessionalized, these users of casual creativity support tools are fully domain experts.

By contrast, scientific tools are understood to prize *control* over 'power', i.e. the ease of creating some expression of the model - but not necessarily any given one. Domain-agnostic techniques for maximizing control which neglect the gulf of evaluation will also ruin the user experience. Imagine trying to control hundreds of variables at once using only a script in a text editor, and running that script over and over again to discern the effect of your manipulation.

In the earlier case study of rover path planning (section 0.1.2), we saw an intervention to address the gulf of evaluation in a deeply domain-specific way.

0.2.3 Mixed Initiative Interfaces

The simplest obstacle to evaluation is a clunky representation. In this section we address the locus of control that can make a software instrument feel more or less responsive, as the next section will deal with how a certain representation is chosen and validated.

Mixed-initiative software is a class of software whose representation updates in response both to *user actions* and to *its internal state*. For example, a typical Python batch-script accepts a string of arguments at the terminal. While it runs, it can be halted, but accepts no other input (except at explicit, blocking prompts). The user has the initiative when they invoke the script. Subsequently, the computer has the initiative until it finishes, or is interrupted.

Marti Hearst wrote [14] in 1999 about 'mixed-initiative interaction' as an interdisciplinary combination of AI and HCI. Hearst contrasted two existing approaches to interaction with intelligent systems, and attempts to locate the best of both worlds:

- human control, as in design support tools, or
- system control, as in a control systems model, versus
- mixed-initiative control, like a dialog with the software medium.

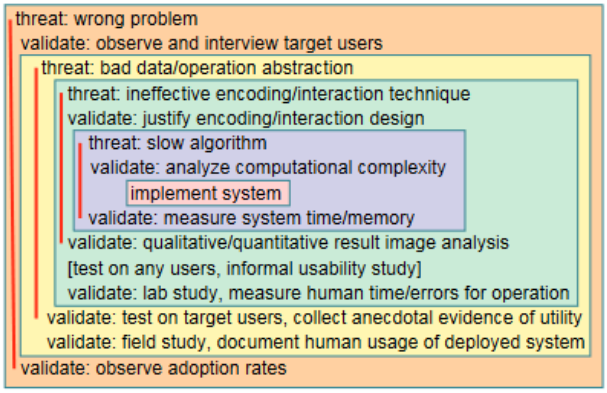

Figure 7: Tamara Munzner's hierarchy of threats to interdisciplinary design projects involving data visualization, reproduced from their paper.

For example, computers tend to understand geometric forms as arrays of numbers, but displaying those forms to the user as a lengthy sequence of Arabic numerals is inappropriate. The technical infrastructure supporting computer graphics certainly exists as a library for any general-purpose computing language. But low-level libraries such as OpenGL (which has implementations for C++, Java, Python, etc.) are not only very powerful, but also require significant technical expertise to use.

For the visualization designer, a library such as Processing or matplotlib furnishes its API, which defines a language of graphics. The language may refer typically to a raster grid (as Processing understands its canvas), or to a data plot (as matplotlib understands its datasets), or to some other screen representation (such as an oscilloscope driven by an audio buffer, or a geographic map driven by a tile layer).

For the expert user, libraries such as SciPy offer similar leverage on their domain-specific tasks, but the API may not be an appropriate interface to work through.

So a designer needs to wire the domain-specific representations up to appropriate pre-existing visualization and interface components (output and input widgets). When an interface is understood by the expert and correctly wired to its representations, it is possible for them to act on it, and not to work blindly.

0.2.4 Visual Encodings and Operation Abstraction

It is difficult for designers who are not domain experts to align the software instrument with their user's analysis task. Tamara Munzner [27] gives the example that hypertext navigation is not equivalent to a graph visualization problem, despite the resemblance of data types.

Many data wrangling and visualization operations are domain agnostic, because they do not care about data types, beyond which fields are quantitative

(continuous-valued, e.g. 'the temperature is 66 degrees Fahrenheit') and which are nominal (discrete-valued, e.g. 'today is Wednesday'). While domain-specific encodings are harder for the designer to share knowledge about and to evaluate directly, they are critical to the tasks performed by expert users.

It is the responsibility of the widget designer to interview expert users. Otherwise the tool for thought is ineffective. Designers who cannot engage in dialog with expert users due to organizational structure will not be able to produce programs with structure supporting rich input / visual analytics.

In Figure 7, Muzner further separates the reflective work of instrument design (the outer orange and yellow boxes) from the technical work of instrument design (the inner green and blue boxes), addressing both levels of threat to the successful design.

The reflective work is specifically prone to wicked problems. A perfectly functional software instrument may fail as a project because it cannot find its users. So we focus on her distinction between 'domain threats' and 'abstraction threats' - i.e. 'nobody has this problem' versus 'nobody solves it that way'.

If an instrument construes a problem that no one has thought about before, this challenge may be exceedingly hard. In this case, domain fit can only really be established through a series of conversations - similar to apprenticing with a novel paper tool (as we discuss in section 0.2.5, next). This type of project structure requires faith that the instrument will become well-scoped through dialog between designers and experts.

Observational studies and paper prototyping are techniques to address abstraction threats without the up-front work of implementing an entire widget. Both are structured collaborative activities: the observational study requires the designers to observe the experts doing their work, while the paper prototype sees the experts do their work with a mock-up of the software instrument where the designers play the part of the computer.

It is essential that the reflective design work be done with both designers and experts. Disciplinary boundaries within organizations limit the rate at which software instruments can be successfully produced.

In any given scientific community, user experience design methods are (or can be) applied to increase the usability of computational tools, and hence their adoption. These bridge the gulf of evaluation, which we discuss in the next chapter.

0.2.5 Gulf of Execution

The gulf of execution can be defined as the distance between the user having formed their intent, and having communicated it through the software instrument. 'Execution' refers to interaction with a data representation through an interface, such as using a mouse to drive a cursor, or a physical bank of sliders to operate a MIDI controller.

In a technical context, the question the user asks here is: was I able to describe my solution, and did it act on the problem as I intended to? Therefore, together with the gulf of evaluation, the gulf of execution determines the pace

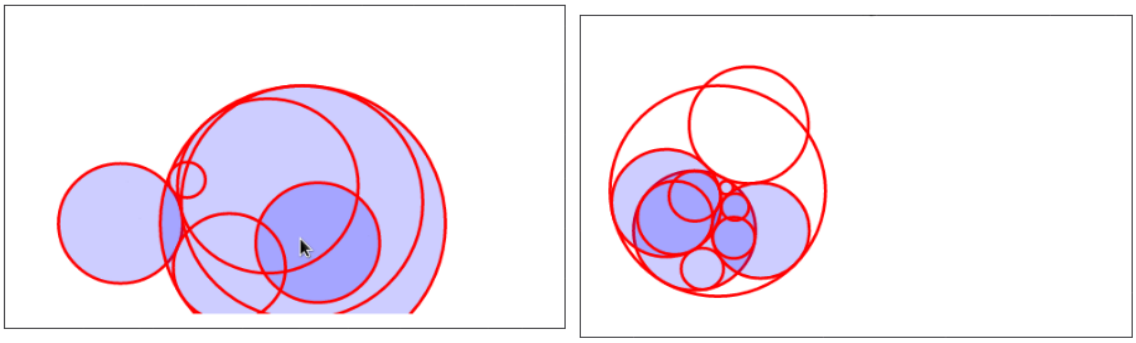

Figure 8: Stills from my interactive diagram of the Circles of Apollonius, running in a Jupyter Notebook backed by phcpy.

of debugging and problem-solving with the data representation. (Programming contexts are an extreme example, because part of the 'data representation' is the program's executable code.)

Visual analytics, as the discipline of data visualization focused on interactive operations, is closely related to paper tools in scientific practices - a concept introduced by Ursula Klein [20]. These include chemical formulae, Feynman diagrams, and other re-writable figures on paper, whiteboards, and similar surfaces.

These figures are not only an image-like output, but also a working representation. Robert Crease [9] describes the need for apprenticeship to learn how to use a novel paper tool such as Feynmann diagrams, especially within the physics community.

As Klein quotes from the historian Jeff Hughes about practices of theoretical physics:

> mathematical tools are open to the same kind of manipulative and interpretive procedures as the experimentalists' instruments and other material resources.

So paper tools also demonstrate how a gulf of execution may arise at the level of design and representation, prior to the implementation of the design.

Crease describes two additional, desirable characteristics of paper tools and how they participate in skillful computation. The tool frames existing projects in new ways; it can be applied to novel projects too, through "improvization and bricolage".

Ultimately, Crease tells us, a paper tool may cause the reification of its representation language - because it is now the common way in which its subject is thought about. Then it becomes harder to notice how the tool necessarily emphasizes parts of the referent system (the physical world), and de-emphasizes others.

Rich input widgets realize paper tools in a computational medium. Automated calculations may bridge the gulf of execution (in the sense of manipulating

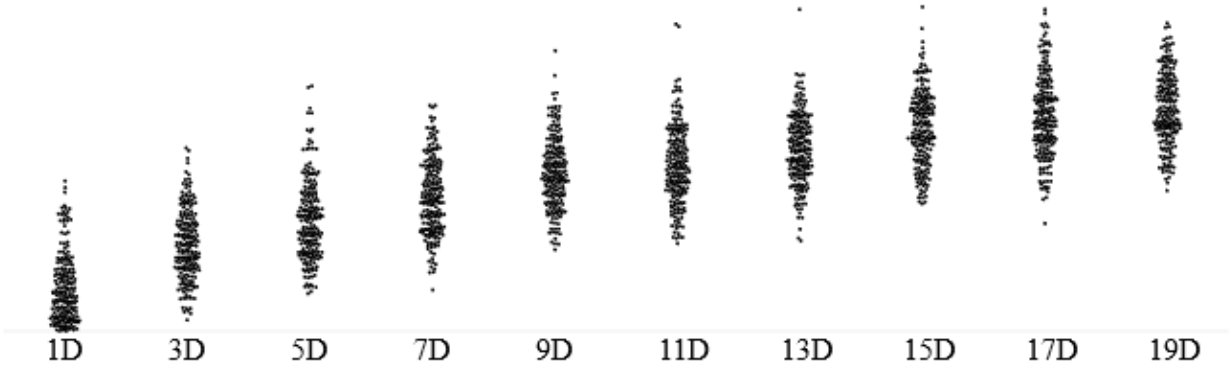

Figure 9: Still frame of Toph Tucker's beeswarm plot of random uniform distributions. *Uniformly* randomly sampled points in high-dimensional spaces are usually 'in the corners', far from the origin.

the referent), as discussed earlier in section 0.2.3. But translating a paper tool to the computational medium requires a notion of rich input as well, to retain the capacity of the paper medium to bridge the gulf of execution (in the sense of selecting a referent).

Execution can be understood as technical skill with an instrument. It is the ability of the expert to have 'said' with the tool what they meant, using the grain of its medium. In scientific domains, an expert typically manipulates a paper tool to plan their complex action, and thereby to bring the problem in line with their mental model.

Our claim echoes Marvin Minsky's conclusion to "Why Programming Is a Good Medium for Expressing Poorly-Understood and Sloppily-Formulated Ideas", as quoted in Software Design for Flexibility by Hanson and Sussman [13]:

> A computer is like a violin. You can imagine a novice trying first a phonograph and then a violin. The latter, he says, sounds terrible. [...] Computer programs are good, they say, for particular purposes, but they aren't flexible. Neither is a violin, or a typewriter, until you learn how to use it.

0.2.6 Curse of Dimensionality

From an engineering perspective, complex problems have very high dimensionality. These are situations in which every row of data has many columns, or there are many interrelated tables of data, or those relations are themselves parameterized or otherwise open-ended (as in the case study of Domino, section 0.1.3), and so forth.

Toph Tucker's interactive article from 2018, There's Plenty of Room in the Corners [32], explores the counter-intuitive consequences of random sampling in very high-dimensional spaces. Sampling is a complementary strategy to analysis, analogous to reading one row of data carefully (or reading one relation carefully across a sampling of possible cases, etc.).

A domain-agnostic solution strategy for analyzing high-dimensionality models is to reduce their dimensionality using topological or statistical methods. Meanwhile, dimensionality reduction is necessary to and implicit in the presentation of a data representation (a sample) through a paper tool or rich input widget, because it fits into two (or rarely, three) spatial dimensions. We claim that the domain-specific strategy of sampling (perhaps following a domain-agnostic analysis) is realized by the paper tools and diagram that each expert brings to bear on their problem.

Tucker writes:

> You don't walk around in a high-dimensional space, but you'd certainly be described in one. The center is all alike, but every corner is a corner in its own way. This helps to formalize certain intuitions. Finding yourself right in the middle is as strange as being way, way out on the tails. Almost nobody *is* what people typically *are*.

Uniformly sampling the expressive range of a very high-dimensional model is going to throw a bunch of perceptual spikiness at you. But the volume of the (hyper)cube is very large due to combinatorial explosion, i.e. it is expensive to make a spanning set of samples. If your goal is to control variation, this is a point when you want to do principle components analysis, or another form of dimensionality reduction.

> One time, maybe a year ago, my friend Colin casually mentioned that in high dimensions, the volume of a sphere goes to zero.

By contrast, sampling instances of that model from a normal distribution (regardless of its center) is going to give you relatively similar results, but not cover the expressive range. This is a really powerful technique in an expressive practice where you're learning what each direction of variation tends to do - like using a paintbrush to make a mark that tapers in a different way, or with a dirty brush to break up the form more.

0.2.7 Conclusion

While wicked problems resist clean solutions, there are interaction strategies that offer points of leverage that make them less bad.

0.3 Domain-Specific Solution Strategies

0.3.1 Introduction

This is my conceptual toolbag for addressing the wicked problems of the previous section, consisting of design patterns that support mixed-initiative human-

computer interfaces in software instruments.

- **Rich input** is the design pattern where pointing at the output is understood as a form of input (i.e. pointer-based gestures). E.g. the Spore Creature Creator, which allows players to define the bodyplan of a creature by dragging limbs to its torso from a box.
- **Player verbs** are the representation of operations which the expert (or any user) can perform directly, whose effect is the bridge between their mental model and the data representation of the world or artifact. E.g. the toolbox in Photoshop, which contains various raster grid selections, transforms, and brushes.
- **Boundary objects** are artifacts and processes that are understood in different ways by two or more sets of domain experts (across a sociotechnical boundary), allowing them to do work together by moving fluidly between their dual understandings of the object. E.g., a data notebook which consumes a data source to produce a plot that a domain expert reads.
- **Direct manipulation** is the design pattern of extending visual data representations to function as tactile digrams, such as replacing a parameter weight (or other single float value) by a knob. E.g. in analog synthesizers, where most parameters are implemented by potentiometers.

0.3.2 Brushes and Rich Input

In 2011, Bret Victor wrote about their project to Kill Math [35]. In Victor's framing, the difference between paper tools and 'a new interface' is not whether quantity and change are present, but whether rich input is enabled.

> We are no longer constrained by pencil and paper. The symbolic shuffle should no longer be taken for granted as the fundamental mechanism for understanding quantity and change. Math needs a new interface.

Rich input to domain-specific languages is an emerging technique supporting mixed-initiative co-creative work between domain experts and their models. By rich input, I mean bespoke and deeply domain-specific interfaces with real-time feedback. These enactive loops between human and machine reveal the 'grain' of the model, as it becomes both a kind of navigable space, and simultaneously an expressive medium.

We draw our first examples from mixed-initiative tools in the domains of digital painting and sculpting. We will zoom out to co-creativity support in the domains of mathematical thinking and in general.

Jennifer Jacobs and her collaborators write about [19] supporting expressive work in digital art through direct manipulation of the canvas, in direct contrast with code-based and GUI-based programming tools (like Processing and Rhino).

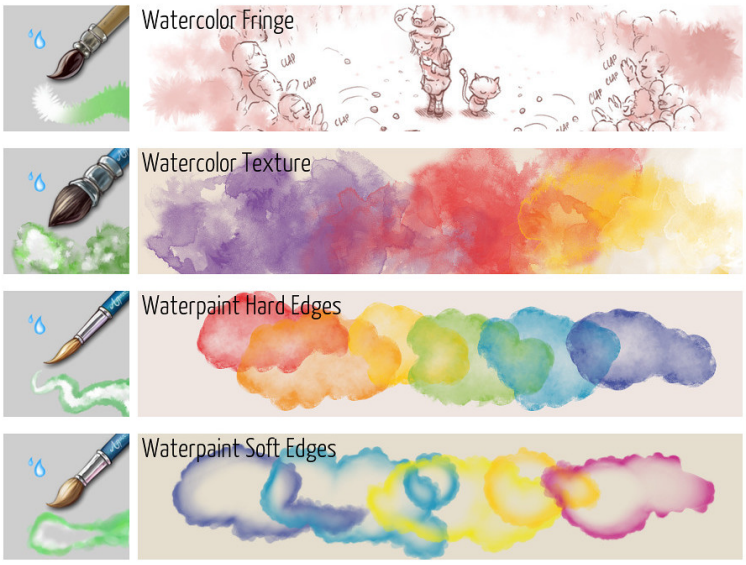

Figure 10: Built-in watercolor brushes for Krita 4. Compare with Photoshop / GIMP / other 2D raster editor software mediums.

They argue that the brushes they implement in Para serve to bridge the gulf of execution in digital painting and illustration.

> Creative coding applications like openFrameworks [23] and Processing [35] are extremely expressive; however, these tools require people to learn numerous textual programming conventions and lengthy APIs before producing basic artwork.

Although mostly unlike physical brushes made of bristles (or physical paint exhibiting fluid flow), digital brushes similarly enable immediate evaluation of the effect of each mark as it is added to the image. Jacobs et al. write:

> Our approach is to augment the conventions of graphic art software to support the creation and modification of procedural relationships exclusively through manipulation of concrete artwork. [...] This goal is challenging because it requires developing a procedural representation that supports expressive creation, yet is compatible with direct manipulation conventions.

Rich input translates immediately between the visible diagram (the geometry) and its symbolic representation (the algebra), in other words.

As an example, the Spore creature creator is a set of 'magic crayons' (see Chaim Gingold's talk [11], written up by [25]) that function more like clay than a finicky mesh and rigging. This constrains the possible outputs relative to other digital sculpture, yet bridges the gulf of execution.

Figure 11: The Spore Creature Creator, with spine handles visible. The eyes and other parts visible in the bin to the left can be dragged directly onto the metaball surface, and will animate automatically. By contrast, sculpting tools like ZBrush afford much more detailed manipulation of the mesh surface, but provide no animation rigging out of the box. Although it is possible to create a vastly greater variety of creatures in ZBrush than in the Creature Creator, it is much harder to produce one at all.

The architecture of any given rich input software can be understood as a collection of interactive widgets which share and communicate through a set of underlying data models. A graphics editor has a brush which creates marks on a canvas. A DAW has a patch bay that specifies which inputs connect to which outputs. Yet neither is very useful without the hands and eyes of a domain expert, who can make skillful and intentional marks (on the canvas) or connections (between musical boxes).

Yannakakis et al. [39] characterize mixed-initiative co-creativity with the machine as supporting *lateral thinking*, which reveals new moves in navigable design space.

> The user's (e.g. designer's) mind is extended onto the diagram and reasoning proceeds through structural (rather than semantic or syntactical) entailment. One therefore thinks through the diagram rather than its use as a simple image. According to diagrammatic lateral thinking, the process of constructing a diagram (an image, a map, or a character) is more important that the final product [32].

This coincides with Victor's claim in 'Kill Math' [35], that the 'symbolic shuffle' of syntax manipulation is secondary to the ability to perform structural operations with a diagram of any sort - perhaps a language, but perhaps an image, or another kind of instrument. Grounding out the ladder of abstraction produces a rich input widget, such as a canvas, or a piece of 'digital clay' in the Spore creature creator.

Figure 12: Sims 1 bobblehead menu. This menu exposes most of the same affordances of objects in the world that Sims perceive and evaluate, and has been retained throughout two decades of Sims games. While users perceive these verbs as a combination of text on the screen and the animated interactions of their execution, Sims perceive them as a signal gradient weighted by which motives that Sim needs to satisfy soon.

This widget has a narrower range of expression than the software library (written in a programming language) that it exposes, in exchange for achieving meaningful expressions faster. For example, Loopy grounds out dynamical systems by allowing users to wire variables together directly, and to observe their state over time.

0.3.3 Operations and Player Verbs

To bridge the gulf of execution, meaning the ability of users to do what they meant, enables the model to express correct intent and the user to create new meaning. It requires the discipline of taking a language, and making it into an instrument, by creating an interface.

In 2020, a team at Georgia Tech and University of Washington describes explorable explanations or *interactive articles* as a distinct medium of computer-assisted cognition technologies and communication tools [16], used by designers from academia and journalism. We might imagine these 'media for thought' as technical demos, except they do not function for the passive observer.

Why is that? In the case of natural language, current literature in psychology [33] suggests that (domain-specific) languages are always enactive, and are taught by interaction rather than by viewing. Interlocutors who worked together on a task learned to share the language of the task. A video recording of the same exact interaction does not create the shared language, without access to the intention behind the utterance.

Martens and Hammer write in 2017 about player verbs as the constrained

affordances produced by the designer in the software instruments we understand as videogames [24]:

> Then, the syntax and semantics are determined by each game individually, depending on what meaning they give to each control input. This language defines the verbs of the game, which may include moving, selecting inventory items, examining world items, applying or using items, entering rooms, and combat actions.

The expert's mental model is where the verb's intention resides, even though its expression is also present in the game world. Worlds are loaded with semantics by the expert who is familiar with their representation (whether in the form of rooms and items, or more abstract diagrams).

> The syntax of a game is its space of recognized player intentions. Note that intention is different from action in the sense that we don't necessarily expect each well-formed intention to change anything about the game state [...]

Software instruments should be expressive, just as paper tools, which afford direct manipulation without understanding it. A software instrument enables and understands these complex, richly contextualized verbs and utterances.

0.3.4 Boundary Objects and Explainable AI

In anthropology, a *boundary object* is understood as a set of artifacts and processes which support cooperative work across disciplinary boundaries or social worlds [21]. We gloss Conlen et al. [8], who worked on tools of visual analytics in operations at NASA JPL, as guiding the design and development of future boundary objects in analogous contexts.

Star wrote about their work with Griesemer regarding why the continuing existence of (interdisciplinary) work arrangements motivates a definition of boundary objects:

> Consensus was rarely reached, and fragile when it was, but cooperation continued, often unproblematically. How might this be explained?
>
> - The object [...] resides between social worlds (or communities of practice) where it is ill structured.
> - When necessary, the object is worked on by local groups who maintain its vaguer identity as a common object, while making it more specific, more tailored to local use within a social world, and therefore useful for work that is NOT interdisciplinary.
> - Groups that are cooperating without consensus tack back-and-forth between both forms of the object.

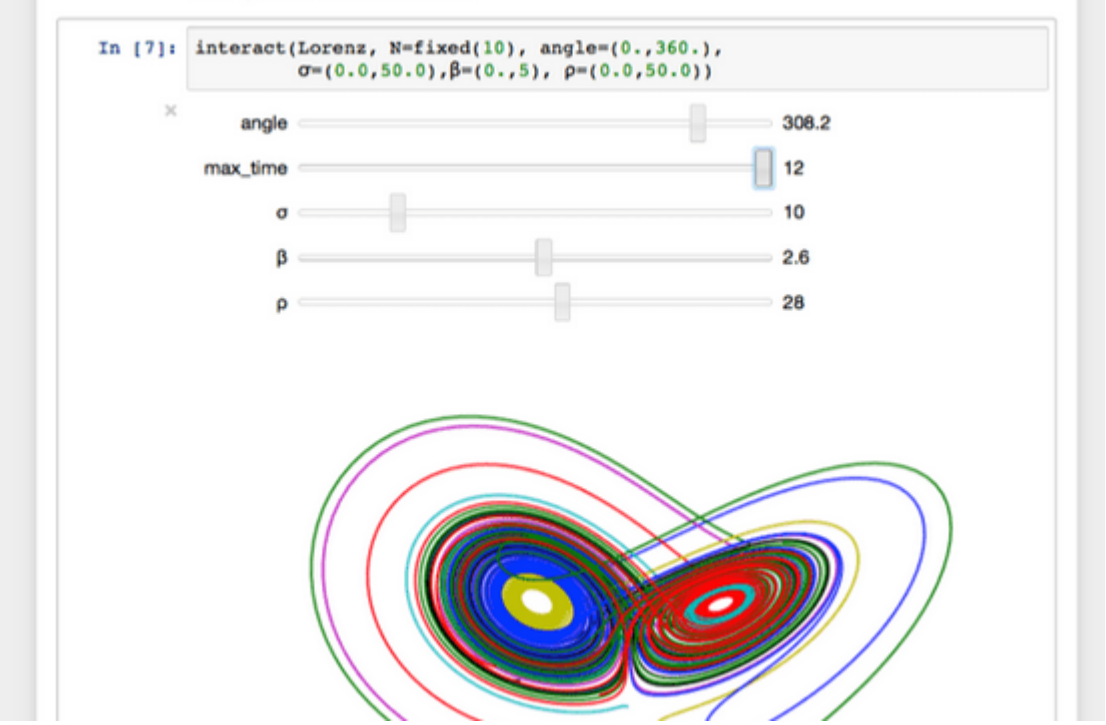

Figure 13: A code cell defining a rich input widget in a Jupyter notebook, which is displayed inline as the result of evaluating this cell. Reproduced from the Project Jupyter webpage.

Conlen et al. produce a detailed set of design principles for boundary objects in the visual analytics domain. They base these on their own work developing a tool for satellite operators responsible for maintaining communications with Mars rovers. Note that operations contexts involve expert operators, and data analytic systems are mixed-initiative tools:

> We offer the following seven principles to guide designers who create visual analytic systems for operations contexts:
>
> - Data must come with context
> - Visualize relationships as well as individual data points
> - Let users travel through the data along the most important decisional dimension
> - UI flexibility needs to be deeper than linking and brushing
> - [. . .]

Let's pause here to unpack their top-level principles. Context is just those features of the data representation which are not valuable to the software algorithm, but only to the human expert. To 'visualize relationships' is actually to visualize subsets of data points, which have been selected by a predicate or *sifting query*.

The binding of query structure to UI elements (including dropdown menus and rich input widgets) is what enables travel and navigation through the dataset. Linking and brushing is a form of rich input, and going beyond that means exploring more rich input strategies.

In this case, designers are describing a tool that is used by (and built by) designers and operators in very different ways - that is, they describe a boundary object. The designers have written a fundamentally different paper about this system than the operators would. For example, decisions are framed in terms of operative dimensions and the distribution of data between plots of different possible densities:

> By isolating plots in small multiples instead of layering them [...] This visual encoding places an emphasis on their consensus rather than a particularly good reading from one or the other.
>
> In addition to picking a satellite, users needed to select a specific heading at which to orient the rover [...] We therefore redesigned the interface to center around the heading as the operative dimension.

Decisions are framed in terms like plot density (to support an uncertainty visualization task) and the ordering of filters in a sifting pipeline (to support a prediction task). These principles all support the operators' need to navigate fluidly around the dataset.

The conversation between models and experts is crucial to the future of work, especially the development of boundary artifacts as software systems which mediate between many stakeholders with different positionalities and perspectives.

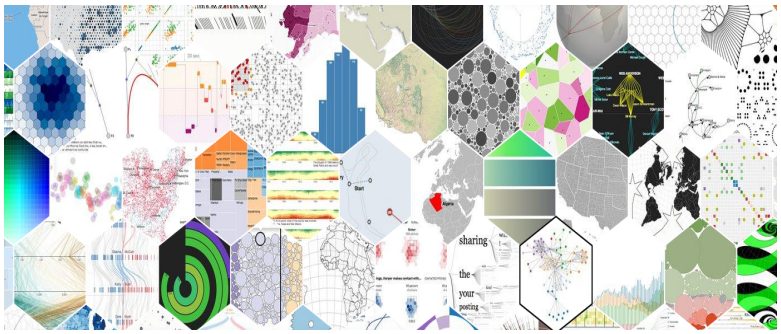

Figure 14: Subset of the D3.js visualization gallery as of its 7.0 release. Note the mix of plot-like (scatter, histogram, treemap, parallel coordinates, heatmap, etc) and domain-specific (polar clock, beeswarm, faceted plots, geospatial) data representations.

In these 'data analytics' projects, data visualization serves to mount an interaction design.

Indeed, many distinctive ecosystems of rich input software exist and are the subject of professional practice in scientific and creative domains. Just as audio production depends on the digital audio workstation (DAW), and digital illustration depends on graphics editors like Photoshop, so too does game development depend on game engines like Unity, and the operation of Mars rovers on planning and simulation tools like Mapgen and R-Sketch.

0.3.5 Direct Manipulation and Visual Form

Hutchins, Hollins, and Norman write in 1985 [18] that "the best way to describe a direct manipulation interface is by example." Their paper introduces the gulfs of execution and evaluation as two complementary phenomena that together index the experience of conversation with an expressive medium, i.e. direct manipulation.

Their first example is that of plotting a matrix of numbers (i.e. support a domain-agnostic visualization task) using a visual programming language. The key point is not whether this language is equipped with task-appropriate verbs, but that these verbs can be dragged to certain subsets of the data and otherwise flexibly manipulated.

It is relatively easy to design plots that abstract several parameters, i.e. which are several levels of abstraction high, compared to reading those plots in a meaningful way. Often an expert discipline will converge on one standard plot that encapsulates a great deal of domain-relevant abstraction, such as phase diagrams in chemistry.

The process of writing a diagram on a blackboard and talking through how each mark is positioned, as a function of others, conveys useful information that is superimposed on but not present in the final image.

Diagrams, like paper tools, are more effective when they're tactile, i.e. executing a move produces rich feedback. Abstract concepts acquire form and tangibility when they are situated in a space of design moves.

Direct manipulation tends to fail in very high-dimensionality problem spaces. A synthesizer with a hundred knobs is harder to control than one with ten banks of ten knobs. This kind of semantic grouping is the start of a domain-specific language, which makes claims about which knobs should be in which bank, typically by distinguishing the roles of different banks.

Domain-specific languages are ubiquitous in the design and development of complex interactive systems, such as videogames [34, 24]. If drawing is the art of visual design and communication, then videogames are the art of the computational meta-medium, and a means to discover what kinds of conversation the computer can hold.

0.3.6 Conclusion

In 2011, Bret Victor wrote about their project to Kill Math [35] as an interaction designer, from outside the institutional context of mathematics pedagogy. Explorable explanations are typically produced by interaction developers within or apart from the institutional sites of conceptual domains[5], often from existing paper tools such as payoff matrices, plots of the complex plane, musical notation, etc.

Hohman et al. explain that interactive articles are used in many contexts [16] - including pedagogy, but also to support experts' conversations with their systems, data representations, and colleagues across disciplines. Like other software instruments, articles need to be easy for experts to share and to run.

Expressive tools are typically the medium of a skillful art practice, such as those explored by Jacobs et al. [19]. They need to solve deeply domain-specific problems faced by experts, and do so with the same interaction strategies as an interactive article, explorable explanation, or another software instrument would.

After Timothy Binkley [1], I argue that these *software instruments* supporting rich input reside in the *computational meta-medium*, meaning they are themselves the subject of a co-creative process of programming. But the domain-specific languages used to define meaning-making behaviors sit higher in the 'stack' than general-purpose languages like Python and JavaScript. They are the languages of games and play [34], of data analytics [8], of interactive articles [16], and of other fields aligned with computational media.

The next section addresses how the software instruments that support and realize these languages can be built.

[5] see the Explorable Explanations gallery at https://explorabl.es/

0.4 Building Blocks

0.4.1 Why The Browser?

In this section, we will describe a stack of technologies corresponding to 'data engines', including libraries built for the web browser in JavaScript.

We target web browsers because domain experts already use them, and because it is relatively easy to package a client-side web application with its own dependencies, allowing it to be used without installation.

Most data science ecosystems exist in interpreted languages with a read-evaluate-print loop (REPL), including JavaScript and other languages of data science, such as Python, Scala, and R. Computational notebooks are document format that combines a REPL with the ability to save and reuse blocks of code ('cells').

Although the data science ecosystem of JavaScript is not as mature as that of R or Python, it is the standard for front-end work, and therefore extremely portable. It is much easier to share instruments that run in the browser, which is much better for sharing results with other research groups [28].

Jupyter is a browser interface to Python, R, and various other 'kernels' in languages of data science & their package ecosystems. JupyterLite implements the Python kernel in WebAssembly, running entirely in-browser. A number of data science libraries have their own ports into JavaScript, such as TensorFlow.js and Immutable.js.

Browsers are an appropriate prototyping environment, where it is possible to prototype novel rich input widgets in a span of days. Technical primers on using JavaScript this way can be found online [10]. In this document we are concerned with why this works, and what the technical frontiers are to rich input widget development.

0.4.2 The Update Loop

An update loop is the minimal amount of program structure necessary to support mixed-initiative software. Any dynamic object present in the software has to be redrawn to the screen each frame (that is, a pixel grid has to be populated with RGB values), immediately after any updates (either for the passage of time, or the arrival of user input).

Libraries like Processing[6] (for Java and Javascript), Pygame [7] (for Python), twgl [8] (for Javascript), and the HTML API [9] (also for Javascript, and to be discussed later) encapsulate the logic of updates and event handlers, similarly to game engines. Unlike game engines that introduce a lot of extra features such as physics engines (to enable collisions between objects), these 'data engines' avoid introducing a dependency on their own internal object ontologies.

[6] https://processing.org/
[7] https://www.pygame.org/news
[8] https://twgljs.org/
[9] https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model

Software built with Processing must define a 'setup' function (which runs once to initialize the program state and canvas) and a 'draw' function (which runs indefinitely many times, until the user closes the software). Similar bindings exist in other data engines.

User input is handled by defining input handlers, which can change selections in the dataset (or other variables in the program) before the next frame is rendered. Callbacks and other event handlers are an appropriate model of asynchronous execution for input widgets and other interactive prototypes.

When the designer is in the drawing logic of their software on a frame-by-frame basis, their library is more like a graphics API, and works in immediate mode. If the designer is instead changing the software's behavior by modifying a dataset that it loads in, then their library is more like a data engine, and works in retained mode.

Most 3D scenes in software are defined through scene graphs, i.e. are in retained mode. In 2D graphics, the utility of defining a (hierarchical) scene graph such as the HTML document object model (DOM) arises from the ability to identify incoming cursor inputs (i.e. a given pair of screen coordinates) with specific entities (e.g. a token that can be dragged and dropped).

0.4.3 Document Object Model

What's a document object model (DOM) [10]? It's the structure (object model) of a document (typically an HTML page, or an SVG canvas on that page) which the interaction designer connects to a data source.

The DOM is ideally the site of all your user-facing mutable state, just as the data source is where all of your immutable state lives. In this world, all intermediate results are functional, i.e. re-computed on the fly when mutable state changes, like cells in a spreadsheet [23].

i. The 'View' type in ObservableHQ [3] defines an input widget as a pair of objects: a user-facing DOM node that is mounted inline with the cell (and is rarely re-evaluated), with a value that is its stateful representation (and is frequently re-evaluated). According to this architecture, all mutable state has a user-facing view and a designer-facing model, which are strongly coupled by the definition of the controller.

'Mounting' a controller is exposing its model through a view. Models based on a data source (such as a CSV file, or a database of multiple tables) may be transformed based on other inputs, such as selection filters. When any of these inputs is integrated with the view (e.g. a line chart where you can mouse over an x-coordinate to gets its y-value), the widget supports 'rich input'.

ii. Browsers maintain a real-time, 1-1 mapping between selection state and visible interface. The DOM API is a GUI toolkit where the representation is explicit, in the form of HTML, rather than implicit (as in Java Swing [11] and

[10] https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction
[11] https://docs.oracle.com/javase/tutorial/uiswing/

other APIs integrated with desktop windowing).

This explicit representation allows direct inspection and debugging through the browser, using the 'inspector' functionality. The inspector is functionally a REPL (as discussed in 'Why the Browser?').

D3.js (Data-Driven Documents) is a syntactic sugar for the HTML DOM API which vectorizes operations on lists of objects (rows of data) which bind each object to a DOM element.

A typical scatter plot might draw 1,000 circle marks to the screen. If circles are indexed by their relative position in a list, this process is prone to error when data points are added or removed. If circles are identified with a row of data instead, this is much safer.

D3.js can be used just like Processing, if it is called within an update loop. However, D3 does not have an immediate mode - because the DOM is always retained across frames, - which makes it harder to teach. D3 code is hard to debug without the HTML inspector, which enables the designer to distinguish between the various possible kinds of blank canvases ('nothing shows up'):

- Selection failed to grab the root element.
- Selection never entered any datum.
- Selection entered each datum, but some visual property is missing.
- Selection entered each datum, but some of them have calculated a visual property to be NaN or similar.

iii. Browsers are an appropriate debugging environment for interactive documents. Debugging can be broken down into developing a robust view (on the front-end) of a sane model (on the back-end).

Developing interactive documents in the browser is a prototyping process, which can be high-fidelity or low-fidelity, depending on whether the marks are representative of data bindings (which we cover in the next section), or are static stand-ins for real data (such as whiteboard or Figma sketches).

0.4.4 Selection Predicates

Bostock et al. write in 2011 about the design of D3.js [2]:

> [...] the overhead of mapping an internal representation to the DOM introduces performance bottlenecks. Intermediate representations can also complicate debugging, as the mapping between code (written in terms of abstract graphical marks) and inspectable output (e.g., SVG elements in the DOM) is often unclear.

In other words, internal representations create an intractable gulf of evaluation, due to the combined statefulness of both a scene graph and multiple in-memory representations. Therefore, D3.js implements a model of data binding based on the HTML selectors API, which is tightly integrated with HTML classes and

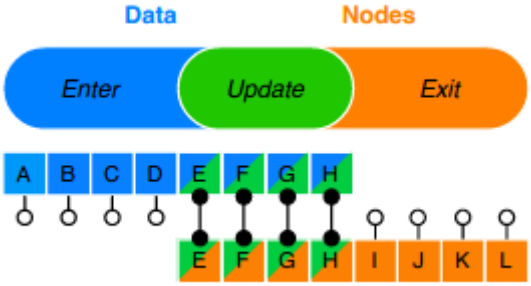

Figure 15: Data binding semantics, reproduced from Bostock et al., 2011, for the second invocation of the 'data' operator on the same selection. Datum E-L are in the old dataset, and datum A-H are in the new dataset. To keep the scene graph in 1-1 correspondence with the dataset, the designer will remove the 'exit' subselection nodes and append the 'enter' subselection nodes. (Equivalently, they can use the 'join' subselector in version 5 of D3.)

CSS styles. Its 'data' operator joins rows of data with the children of a given HTML element, creating a 1-1 correspondence.

Prior to D3, rich input techniques depended either on HTML forms as an out-of-the-box solution, or on binding the DOM with its own API or another library that wraps it, such as JQuery.

This made dynamic datasets difficult to visualize, because enter, update, and exit semantics (see figure 15) were unavailable. While the familiar HTML form elements (e.g. dropdowns, sliders, and text inputs) handle domain-agnostic rich input, they are data-driven in just a single variable (their user-editable value) - except when form validation techniques are used, or when they are bound through a data science library such as IPython widgets [29] [12] for the Jupyter notebook.

D3 supports defining reactive properties of bound data (i.e. the mapping from datum to representation) using functional operators. Bostock et al. write:

> Once data is bound to elements, it is passed to functional operators as the first argument (by convention, d), along with the numeric index (i). [...] Data is "sticky"; once bound to nodes, it is available on subsequent re-selection without again requiring the *data* operator.

In particular, the datum that was bound by D3 to a given DOM node (visible representation) is immediately available when that node generates an event (e.g. mouse-over). Then the datum can be used immediately as an input source (e.g.

[12] Paper describes IPython REPL, see ipywidgets documentation too: https://ipywidgets.readthedocs.io/en/stable/examples/Widget%20Low%20Level.html

in a preview widget, or a linked brushing selection), and its representation can be updated to give a visual feedback.

> Critically, D3 also introduces features that may inform other visualization frameworks: query-driven selection and data binding to scenegraph elements, document transformation as an atomic operation, and immediate property evaluation semantics.

Although a rich input widget tightly encapsulates its own state, it remains to be seen how that state propagates through the software instrument. In the next section, we address how to tightly couple many models to many views through a reactive data model.

0.4.5 Reactive Data Models

Spreadsheets are a familiar example of the *reactive data model*, where changes to values that other values depend on (are downstream of) propagate immediately and automatically.

For the designer, it is desirable to build mixed-initiative software instruments in reactive programming frameworks, both to increase the velocity of prototyping, and to minimize the gulf of evaluation when debugging. One such framework is the ObservableHQ notebook [4], which is distinguished from the Jupyter notebook by its explicit resolution of dependencies between cells.

According to this reactive model, the entire software state should be either part of the working artifact (typically an immutable dataset), stored in the DOM or scene graph, or part of the selection state within a rich input widget.

A rich input widget is able to propagate input into the reactive model using a 'view/value' abstraction, such as the 'viewof' operator in ObservableHQ, which allows the view (a scene graph) to update rarely while the value (a stateful representation of input from the widget) updates often.

Functional programming models naturally support reactivity because they minimize side effects, and therefore the amount of state that depends on other state. (Otherwise, a circular definition is likely, which in the reactive model causes an error.)

Debugging an interactive software is hard because state can accumulate over time, leading to unpredictable errors due to events from a long time ago. This is especially bad because software with an interactive front-end should never crash on the back-end and lock the user out from interaction.

Even in JS and Python (both interpreted languages that do not have a compilation step with type-checking), it is desirable to define operations that are type-safe. Strongly typed data ensures that all possible representations are valid models.

Moreover, when intermediate values are always functional, and never stateful, it is possible to debug the software at a single logical time [38], rather than having to reason across many possible execution traces simultaneously.

Game engines rarely support functional and reactive programming. But we claim theses are desirable features in all data engines that bind data to

interactive representations (including both the selector language of D3, and the cell-based structure of a computational notebook).

0.4.6 Conclusion

Now we know some ways to make rich input widgets, and put real artifacts together in conversation with wicked problems, according to our four solution strategies.

To create boundary objects, we can take advantage of the browser's ubiquity and familiarity. To create experiences of direct manipulation, we need to support reactive models of data beyond just plots.

0.5 Rich Input Widgets

0.5.1 Definition

In this document, we have only barely touched on the full breadth and depth of work with mixed-initiative co-creativity support software in contemporary domains of skillful practice. We observed that a software instrument defines a language over its data representation, just as games define a language of play with respect to their virtual world. Finally, we have described how a software instruments may be rapidly composed from rich input widgets by using computational notebooks in the browser.

A rich input widget combines a data representation with an interface element, just as paper tools like Feynmann diagrams do in the whiteboard medium. Multiple widgets are connected by levels of abstraction, to support patterns like 'drilling down' to individual artifacts, or 'zooming out' to study the effect of several parameters of variation.

The modularization of representations and operations through responsive widgets serves to bridge the gulfs of execution and evaluation that otherwise make interfaces to large datasets challenging to use and to understand. Expert users benefit from the flexibility of widgets to define (and indicate) the context of the data on which they choose to operate. In professional work practices, we understand software instruments as boundary objects, i.e. artifacts and processes that necessarily relate the domain-specific languages of two or more disciplines.

0.5.2 Conclusion

We address both interaction designers (who maintain and create software instruments) and domain experts (who use and adapt software instruments) with our definition of rich input widgets. Mixed-initiative creativity support is a model for the cooperative and pro-social use of AI, yet its realization is constrained by wicked problems in interface design and cross-disciplinary communication.

In this document, we discussed multiple successful software instruments that support the specific domains of visualization design, robotics planning, and systems thinking. These sociotechnical ecosystems benefit from the presence of

boundary artifacts that could not exist without interdisciplinary thought and collaboration. We reviewed an interactive article (Up and Down the Ladder of Abstraction) and an interactive article development environment (Loopy), which use their own rich input widgets to enactively convey complex systems design principles. We also reviewed recent interface design and data wrangling work at JPL and in academic data visualization.

We argued that software instruments are valuable to address wicked problems, which are intractable to fixed definitions of 'problems and solutions' in a given domain of work, and therefore require domain experts to guide the instrument designers. We presented solution strategies for domain-specific problems that afford the expert to observe and to operate on the data representation, yet constrain the novice to meaningful utterances within the domain.

Software instruments are increasingly supported by a rich software ecology which makes them easy to share with experts and other stakeholders. We seek to guide more technical work towards mixed-initiative applications, and to guide more interaction designers towards these highly expressive and deeply compositional tools.

Rich input widgets are a challenging genre of software to design and develop, yet rewarding for experts to use and to share. This document aims to demonstrate a path toward the rapid prototyping of software instruments in all domains of skillful practice.

References

[1] Timothy Binkley. The Vitality of Digital Creation. *The Journal of Aesthetics and Art Criticism*, 55(2):107–116, 1997. Publisher: [Wiley, American Society for Aesthetics].

[2] M. Bostock, V. Ogievetsky, and J. Heer. D3 Data-Driven Documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, December 2011.

[3] Mike Bostock. Introduction to Views, October 2018.

[4] Mike Bostock. Observable's not JavaScript, June 2019.

[5] Richard Buchanan. Wicked Problems in Design Thinking. *Design Issues*, 8(2):5–21, 1992. Publisher: The MIT Press.

[6] Nicky Case. LOOPY: A Post-Mortem, March 2017.

[7] Katherine E. Compton. *Casual Creators: Defining a Genre of Autotelic Creativity Support Systems.* PhD thesis, UC Santa Cruz, 2019.

[8] Matthew Conlen, Sara Stalla, Chelly Jin, Maggie Hendrie, Hillary Mushkin, Santiago Lombeyda, and Scott Davidoff. Towards Design Principles for Visual Analytics in Operations Contexts. In *Proceedings of the 2018 CHI*

Conference on Human Factors in Computing Systems, CHI '18, pages 1–7, New York, NY, USA, April 2018. Association for Computing Machinery.

[9] Robert P. Crease. How Feynman diagrams transformed physics. *Physics World*, (December 2019), December 2019.

[10] Maya Gans, Toby Hodges, and Greg Wilson. *JavaScript for Data Science*. Chapman and Hall/CRC, 1st edition, January 2020.

[11] Chaim Gingold. SPORE's Magic Crayons, 2007.

[12] Samuel Gratzl, Nils Gehlenborg, Alexander Lex, Hanspeter Pfister, and Marc Streit. Domino: Extracting, Comparing, and Manipulating Subsets Across Multiple Tabular Datasets. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2023–2032, December 2014. Conference Name: IEEE Transactions on Visualization and Computer Graphics.

[13] Chris Hanson and Gerald Jay Sussman. *Software Design for Flexibility: How to Avoid Programming Yourself into a Corner*. MIT Press, Cambridge, MA, USA, March 2021.

[14] M.A. Hearst, C.I. Guinn, and E. Horvtz. Mixed-initiative interaction. *IEEE Intelligent Systems and their Applications*, 14(5):14–23, September 1999. Conference Name: IEEE Intelligent Systems and their Applications.

[15] Fritz Heider and Marianne Simmel. An Experimental Study of Apparent Behavior. *The American Journal of Psychology*, 57(2):243–259, 1944. Publisher: University of Illinois Press.

[16] Fred Hohman, Matthew Conlen, Jeffrey Heer, and Duen Horng (Polo) Chau. Communicating with Interactive Articles. *Distill*, 5(9):e28, September 2020.

[17] Alexandra Holloway. *Design of a data-driven micro-display for situation awareness in bursty environments (when not much is happening most of the time) - ProQuest*. PhD thesis, University of California Santa Cruz, June 2015.

[18] Edwin L. Hutchins, James D. Hollan, and Donald A. Norman. Direct Manipulation Interfaces. *Human-Computer Interaction*, November 2009. Publisher: Lawrence Erlbaum Associates, Inc.

[19] Jennifer Jacobs, Joel R. Brandt, Radomir Meeh, and Mitchel Resnick. Dynamic Brushes: Extending Manual Drawing Practices with Artist-Centric Programming Tools. In *Extended Abstracts of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI EA '18, pages 1–4, New York, NY, USA, April 2018. Association for Computing Machinery.

[20] Ursula Klein. Paper tools in experimental cultures. *Studies in History and Philosophy of Science Part A*, 32(2):265–302, June 2001.

[21] Susan Leigh Star. This is Not a Boundary Object: Reflections on the Origin of a Concept. *Science, Technology, & Human Values*, 35(5):601–617, September 2010. Publisher: SAGE Publications Inc.

[22] Pei Liew, Alex Sciuto, and John Thompson. R-Sketch: Mars Rover Path Planning.

[23] lord. How to Recalculate a Spreadsheet, November 2020.

[24] Chris Martens and Matthew A. Hammer. Languages of play: towards semantic foundations for game interfaces. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*, FDG '17, pages 1–10, New York, NY, USA, August 2017. Association for Computing Machinery.

[25] Ross Miller. Spore's power struggle: freedom vs. beauty, March 2007.

[26] Dominik Moritz, Chenglong Wang, Greg L. Nelson, Halden Lin, Adam M. Smith, Bill Howe, and Jeffrey Heer. Formalizing Visualization Design Knowledge as Constraints: Actionable and Extensible Models in Draco. *IEEE Transactions on Visualization and Computer Graphics*, 25(1):438–448, January 2019. Conference Name: IEEE Transactions on Visualization and Computer Graphics.

[27] Tamara Munzner. A Nested Model for Visualization Design and Validation. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):921–928, November 2009. Conference Name: IEEE Transactions on Visualization and Computer Graphics.

[28] Chris Olah and Shan Carter. Research Debt. *Distill*, 2(3):e5, March 2017.

[29] Fernando Perez and Brian E. Granger. IPython: A System for Interactive Scientific Computing. *Computing in Science Engineering*, 9(3):21–29, May 2007. Conference Name: Computing in Science Engineering.

[30] Horst W. J. Rittel and Melvin M. Webber. Dilemmas in a general theory of planning. *Policy Sciences*, 4(2):155–169, June 1973.

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

[32] Toph Tucker. There's Plenty of Room in the Corners, February 2018.

[33] Jasper van den Herik. Attentional actions - An Ecological-Enactive account of utterances of concrete words. *Psychology of Language and Communication*, 22(1):90–123, January 2018.

[34] Riemer van Rozen. Languages of Games and Play: A Systematic Mapping Study. *ACM Computing Surveys*, 53(6):1–37, February 2021.

[35] Bret Victor. Kill Math, April 2011.

[36] Bret Victor. Up and Down the Ladder of Abstraction: A Systematic Approach to Interactive Visualization, October 2011.

[37] Noah Wardrip-Fruin. *Expressive Processing: Digital Fictions, Computer Games, and Software Studies*. Software Studies. MIT Press, Cambridge, MA, USA, July 2009.

[38] Hillel Wayne. Physical vs Logical Time, June 2021.

[39] Georgios N Yannakakis, Antonios Liapis, and Constantine Alexopoulos. Mixed-initiative co-creativity. page 8.