Literature Review - Software Instruments

Interactive Construction of Interactive Artifacts

Jasmine Otto

4-27-21

Abstract

I will frame the problem of research debt in terms of design spaces, whose navigation is accomplished via software instruments. In particular, the analysis of specific artifacts is in conversation with the design of models from which they arise. Likewise, the design of specific artifacts is in conversation with the analysis of models from which they arise. Models connect the worlds of design and of analysis.

Contents

List of Tables

List of Figures

This post begins an essay about software instruments, which will be serialized on Tuesday evenings for the next five weeks or so.

Introduction

We describe software instruments as a family of interactive computational mediums (i.e. game-like artifacts) spanning both toys and tools, which all possess a stateful representation of an artifact, and a set of verb-like affordances to manipulate that artifact. Instruments may be used to *discover* or to *characterize* ('analysis') both artifacts themselves, and phenomena arising in artifact space which affect the translation of user intent through operational logics into (paths leading to) artifacts.

We identify 'discovery' with *practices in artifact space*, which may be improvised or rehearsed, but always traces a path from the 'blank canvas' (or another starting point) to the user's goal (e.g. a finished essay). Every performance is mediated through an instrument, which translates user intents into navigation of the artifact space, and user actions into manipulation (i.e. making, or finding) of an artifact.

We identify 'characterization' with *research distillation*, which has to do with the effort of representing artifacts of a given kind (e.g. all taxation policies, or all lighting equations) as points in a navigable artifact space. *Models of behavior* are typical artifacts in both science and industry, whether they are predictive data-driven models (e.g. of climate change, or consumer preference), or specifications of algorithms (e.g. sharding across Amazon Web Services instances).

Skilled users engage with their instruments in a practice. A painter may apply acrylic paint to canvas stretched on wooden boards, or they may press a stylus against a tablet to manipulate a pixel grid shown on an LCD screen. An ecologist may measure the growth of algae by shining a light through their test tube and calculating its absorbance, or they may build a survey grid on a field site and calculate the density

of food from the perspective of a squirrel.

Users typically modify their instruments to alter their expressive ranges, and to explore phenomena related to ease and sophistication of expression, and thus become designers of instruments. They may develop instruments that are useful to navigate complex sociotechnical domains. They may use these instruments to negotiate ontologies, theories, and other characterizations of real-world phenomena.

The next two sections of this essay draw on existing design practices, thoughts on the use of, and conversations with software instruments. We will rely upon developments in complex systems theory since 1980, when Seymour Papert's book titled Mindstorms, - a seminal work in pedagogy using software instruments, - was published. In the future, we will pursue the application of software instruments to the navigation of extremely high-dimensional artifact spaces, especially the design of interactive artifacts, including software instruments.

We will explore data visualization as a form of *interaction design*, and hypertext theory as the design of *tools for thinking*. The future of computational literacy in our technocratic society hinges upon citizen understanding of the models with which we are concerned. The future of work in science also relies on developing expressive forms of reproducibility and proof using software instruments.

Affordance and emergence

From the design standpoint, a software instrument can be said to implement a domain-specific language of interaction, through any combination of keybindings, draggable interface elements, conversational agents, file upload, structured text input, and so on. These inputs may be reframed by the model (as when a Sim paints an image from your files and hangs it in their house), or interpreted by it (as when you direct a Sim to hang up the painting, but they are interrupted by the phone ringing). They may replace other data in the model (e.g. applying a certain file), or act to perturb it (e.g. applying a certain keystroke).

Certain instruments are open-ended, meaning that their artifact space is not just

difficult to characterize (e.g. all configurations of a Rubik's cube), but perhaps impossible (e.g. all songs that can be played on a guitar). From the user standpoint, these different affordances may be combined in ways the designer did not anticipate. The results of an analysis are the skillful result of playing with the instruments of analysis. Therefore, performance - whether from a score, or by improvisation, - is vital to how software instruments are made productive. Written scores and recipes let us understand and exercise the capabilities of a software instrument, in a way that recordings of results alone cannot. We call this capacity that of 'emergence' or 'agency' in a software instrument.

In the world of procedural generation, it is typical for the user and the designer to be the same person, at least at first. In the world of scientific research, this may initially be true, but no longer after research dissemination or cross-disciplinary collaboration. Thus an issue of reproducibility arises, in the same way that having the same guitar and amplifier settings as a famous guitarist does not, by itself, help you to play like them.

Unlike physical instruments, which can probe the world either directly or through simulacra of it (e.g. the use of paint to capture light), software instruments must substitute data structures and modelling constraints for their referent artifact or phenomenon. There is one kind of exception - when a neural network, a game with purely rational agents, or another kind of *artificial life* is investigated. Yet our software instruments are also called upon to describe other kinds of object, ranging from portraits and other avatars of people, to ocean-exploring robots, to global financial and climate systems.

Painting does not demand a one-to-one mapping from, say, specks of paint to grains of sand on a beach. Neither does the characterization of a complex system benefit from extreme detail, e.g. literal representation of every physical molecule in that system. Through an instrument and its simplified representation, the intentional and skillful choice of actions on the representation (e.g. brushstrokes on a canvas) is itself a source of insight. The player of the instrument cooperates with it to discover its 'expressive range' in artifact space.

Any expert using computers may find in this essay a discussion of the qualities in an software instrument which facilitate high-quality creative output, especially the suitability of its affordances and its representations to an open-ended set of domain-specific analyses.

Any artist using software mediums may find herein an attempt to describe the 'conversation' with a digital 'canvas,' which we claim as vital not only to producing compelling artifacts, but also to engaging on shared ground with a community of fellow practitioners.

Contents

The first part of the background section of this essay will address the specific problem of 'research debt,' coined in analogy to technical debt by Olah and Carter [1]. Along the way, we'll critique mathematics pedagogy, learn to tune a platformer game in real time, and discover a better way to code interactive artifacts.

The second part of the background section concerns a case study in shared experiences of navigating (generative) design space, namely the Annals of the Parrigues, written by Emily Short [2]. We will read its appendix, which recounts designing a medium of expression whilst navigating the space it describes, against similar accounts by Vi Hart and Iannis Xenakis.

In the case studies section, we will explore an avatar creation tool from a videogame, and a web-based library for binding data to visual representations.

In the final, theoretical section of this essay, we will introduce terminology for practitioners who intend to develop novel or existing software instruments, and to communicate their results through landmarks, scores, and other methods of orientation in design space using software instruments.

[1] https://distill.pub/2017/research-debt/

[2] https://emshort.blog/2015/12/07/procjam-entries-nanogenmo-and-my-generated-generation-guidebook/ — title: Software Instruments - Definition

excerpt: “We describe software instruments as a family of interactive computational mediums (i.e. game-like artifacts) spanning both toys and tools…” author: Jasmine Otto —

What is a software instrument?

I am revisiting the definition of software instruments today. I intend to return to the question of research debt next Tuesday, and the remainder of last week’s outline subsequently.

Examples of software instruments

Here are some typical examples of software instruments, starting with instruments for static media (i.e. specifying state):

- social media applications with built-in image filters
- dedicated tools for 2D content creation, such as raster-based, brush-based, and vector-based image editors
- character creators in videogames, especially in games where you ‘build and test’ and/or ‘play as’ the resulting character(s)
- text editors

Of instruments for durational media (i.e. specifying state over time):

- digital audio workstations, a kind of development environment for audio content
- multimedia capture and editing tools, such as Open Broadcast Studio and After Effects
- 3D software suites oriented toward content creation, such as Maya and Houdini
- computer-aided design suites, oriented toward 3D design of physical objects

Of instruments for interactive media (i.e. specifying state and its behavior):

- read-evaluate-print loops (REPLs), a kind of software development environment

- software development environments built around computational notebooks
- creative coding environments with first-class grpahical output, such as Processing and its forebearer, Logo
- editors for videogame engines, such as Unity Technologies' Unity (for their Unity engine), or Bethesda's Creation Kit (for their Creation Engine), or Bitsy (for Bitsy games)

Of instruments for organizing media (i.e. managing collections of information):

- digital index cards, such as Scrivener and Anki
- essay-editing software, such as Word and TeX Studio (as specialized text editors)
- communal discovery platforms such as Pinterest, Pinboard, Are.na, and Reddit
- citation management software such as Zotero, browser built-in bookmarks managers, and the hypothetical Memex
- nonfictional hypertexts, such as Wikipedia, certain Twitter threads, and personal Roam Research wikis
- digital whiteboards and mind-mapping software (as specialized vector-based image editors, & may map out a hypertext)

Checklist for software instruments

What if your computer program is in zero or multiple of the genres listed above? This questionnaire will diagnose whether it is a software instrument.

- Does this program couple a subject to an object in a feedback loop?

The subject must be capable of perception. There must be a data representation of the program input, such as keystrokes on a keyboard or midi controller, otherwise the instrument is not software.

The object must be modifiable by the subject, and this modification must be perceptible to the subject. There must be a data representation of the object's state which contributes to the program's output.

- Does this program represent the object in a form the subject can perceive?

For example, does this program prefer to express an array of floating point numbers as the samples of an audible waveform, or as the pixels of an image? The correct answer isn't obvious. Programs may use idiosyncratic formats, and it may not write all memory into its save files. However, the program must interact with some sound API and/or graphics API to drive its hardware outputs.

Programs can be studied at the level of their graphics buffers, as an in-game photographer ('screenarcher') might to understand a shader modification, or perhaps through a narrative constructed around a series of screenshots. The object may be a performance, an improvisation, an experience. But programs can also be studied at the level of their save files, which in the case of content creation tools, are likely to directly represent the object (a multimedia artifact).

- Can the subject directly manipulate this representation through the program?

Some programs will allow the subject to manipulate the expression of the data representation of their object by clicking and dragging it. Other programs require the subject to manipulate something else (e.g. an array of sliders) and see their object recompiled in real time. Other programs require the subject to manipulate something else, and then press a button and wait to recompile their object. [Not sure whether to use 'compilation' from programming languages here, or stick to the generic but nonstandard 'expression.']

Indirect manipulation of objects is harder to understand the effect of. It may also be necessary if it is difficult to infer which property is being manipulated, i.e. when the property is emergent. It is easier to manipulate emergent properties indirectly when relevant high-level properties are exposed to direct manipulation.

A digression about hardware and data-bending

The object may, in addition to its data representation, also be physically present somewhere (e.g. a robot). However, physical objects can lose synchronization with their data representation (whether due to communications lag, inappropriate envi-

ronment, or any other source of unrepresentable events), which degrades the quality of performance with the software instrument to an unpredictable degree.

An analogous state of 'glitchiness' occurs when the data representation can be understood as though it were physical in some states, but not others. When a frame becomes spliced into an unrelated sequence of action, or a vertex becomes displaced and weirdly stretches the surface it was part of, the human subject is likely to perceive a problem with the object (as a media artifact), even if the program is still able to operate on it happily.

Negative numbers (if regarded as quantities, but not as rate of change) and imaginary numbers (if regarded as rates or quantities, but not as rotations) are examples of data representations that may or may not be 'glitchy,' depending on the object represented.

In general, representations of non-'glitchy' objects are expected to be points in a rugged manifold that sits in the higher-dimensional space of the data representation. An instrument is more effective in a typical sense if it is good at tracing this manifold, i.e. bad at sampling points that aren't on it.

On the other hand, there is a wild unknown of possible artifacts located off-manifold that someone might be interested in reaching with other instruments. It is possible - even likely, in the case of scientific theories, - that the non-glitchy referents of such artifacts have simply not been discovered yet.

Last week, we talked about practice and performance in a computational medium, using instruments to navigate an artifact space. This week, we use our instruments to play in another sense - to 'think the unthinkable,' by embedding skill into our representations of and operations on difficult problems.

Next week, we will have our first case study of a software instrument (The Sims 3 Create-A-Sim). In two weeks, the second (a reading of The Wisdom and Madness of Crowds, as if we were porting it to D3.js). After that, we will take a break for revisions.

Research debt, and its counters

At first glance, it isn't obvious that we have any skill transfer problems in science or engineering. Maybe the problem with math teaching, what makes people afraid of math, is just a problem with math.

Olah and Carter write [1]: > Often, some individuals have a much more developed version of an idea than is publicly shared. There are a lot of reasons for not sharing it (in particular, they're often not traditionally publishable).

Mathematics is extreme in this regard, because mathematicians have a lot of explanations that only make sense to other mathematicians, and only if that person takes a lot of time to find the background that makes the explanation make sense. (Surprisingly, mathematicians just do this. They aren't busy with other things, and they can't be.)

Mathematics has a lot of research debt, in other words.

A community of practice has research debt when its students and researchers know they need to learn some topic X, but the public exposition on topic X is very dense and requires a few years to digest.

This becomes a problem when people keep re-inventing knowledge about topic X, but using a different terminology that is less inscrutable to them. Usually it is specialized to their use case, which angers people who are using the otherwise inscrutable terminology, if they realize it is the same thing at all.

Any topic can be and will be a topic X, but the more abstraction it contains, the worse this situation gets. I'm going to assert that science and engineering contain a large and increasing amount of abstraction, in the present historical moment.

Grounding

Interactive articles are unreasonably effective at cutting through research debt. They are both demonstration, and instructions for a computer to reproduce that demonstration. They are more portable than other forms of software, because they can

run in a browser or in a computational notebook environment.

Conrad Wolfram has talked about [2] using interactive articles to tackle mathematics pedagogy, a kind of research debt that every schoolchild runs into, using rich visual output and parameter sweeps.

When Wolfram focuses on the pupil and their notebook of calculations, and how much faster a computer is than a page for this usage, he resides in the parts of the interaction loop. When Bret Victor expands on [3] the idea of representation and direct manipulation of the parameters, he resides in the arcs of the interaction loop.

Nielsen's notes on Victor continue [4] to discuss the data visualization operations of linked brushing and drilling down. But he is getting at the building blocks of them now, the operational logics* of an interactive article. > * The operation of "tying" two quantities together, to form a single linked entity. > * The operation of scrubbing a number. > * The operation of "searching" over the graph.

- See Joe Osborn's catalog [5] from his dissertation. Note 'linking' the scrubbing slider to the parameter, 'controlling' that slider with the mouse, and 'spatial matching' over the parameteerized space of graphs.

Reproducibility

Many computational notebooks are not articles themselves, but distributed alongside papers, providing a reproducible computational component. It is better to share notebooks than to share scripts because a notebook can encapsulate more of its own environment configuration, and is therefore more likely to run on someone else's computer. Some scripts (and other scientific softwares) are distributed in Docker containers for this reason.

It is typical to share both software and libraries through a package-management system, such as Python's pip, Javascript's npm, or Scala's sbt (to name a few). Because scientific software written in Python often has non-Python dependencies (such as the Math Kernel Library, or other C++ and Fortran code), the Anaconda distribution of Python has a specialized package manager that can handle such

dependencies.

This is one of the software ecologies that makes it possible to share software instruments (including interactive articles) between computers via the internet. The development of HTML5 is another such story. Two decades ago, most portable software depended on Java (in the scientific use case) or Flash (in the explorables use case), neither of which runs in browsers anymore.

Addressing research debt isn't just a question of software distribution, but also one of computational literacy. The full value of a computational notebook cannot be realized without the skill to rewrite its score (its code) without breaking the data pipeline it describes. Whereas a poorly written article is difficult to understand, a poorly written notebook is difficult to build on.

Olivia Guest describes [6] bad instruments in the professional practice of mathematical modelling, such as a programming language that teaches users to poorly encapsulate their abstractions, and whose entire statefulness leaks immediately into the interaction loop (to my immense dismay as a functional programmer).

[1] https://distill.pub/2017/research-debt/
[2] https://www.computerbasedmath.org/resources/reforming-math-curriculum-with-computers.php
[3] http://worrydream.com/LadderOfAbstraction/
[4] http://mnielsen.github.io/notes/kill_math/kill_math.html
[5] https://eis.ucsc.edu/analyses-and-approaches/operational-logics/
[6] https://neuroplausible.com/matlab

Instrument design

What does it mean to design a 'good instrument' which actually reduces research debt, instead of creating it?

We claim that programming skill alone is insufficient to make a software instrument. The shape of its data must be amenable to representation, and there must be a set

of actions defined on data of that shape which are sensible. In other words, domain expertise is required.

For instance, an avatar creator instrument such as Create-A-Sim must implement actions in a 'face space' of all faces that Sims can have. But it is more sensible to 'place the eyes' or to 'tweak their shape' than to (e.g.) change the position of one vertex comprising part of the eyelid. Our first case study will consider face spaces and Create-A-Sim in greater depth.

In the case of a scientific instrument, most actions are typical of data science, such as 'filter for events matching this condition' or 'show me a timeline of the remaining events.' However, as an example, nonlinear dynamical systems are a kind of model requiring more specialized actions. These systems consist of a 'set of rate equations,' which describe the change in a state vector over time. Every combination of parameters and initial conditions may produce a very different traces (i.e. sequences of states taken on by the system).

For instance, one may 'draw a phase portrait,' which is the plot of one trace of the system. 'Performing a parameter sweep' means to step one (or more) parameters by a fixed amount several times, yielding several conditions. The traces produced by the system in each condition are plotted together (usually in layers). To 'draw a bifurcation diagram' is similar, except instead of the trace, the steady-state solutions of the system (i.e. a description of the phase space) are plotted together (usually along an axis). These 'higher abstraction' visualizations help us to understand the effect of the parameters and their co-dependence with each other, or with certain conditions of the system.

Bret Victor's interactive articles about dynamical systems visualization not only describe and demonstrate these operations, but they also articulate a vehement criticism of the research debt that has piled up in the terminology of nonlinear dynamical systems. I have just now described some simple tasks in this area of research using a volley of complicated terms.

(It is beyond the scope of this document to consider games with 'simulation' or

'idle game' mechanics as research instruments themselves. Consider Dwarf Fortress, Crusader Kings III, or Universal Paperclips. In these cases, the artifact is a simulated life-world, and the research debt lies in an ontology of that life-world that may be shared by only one person. See Kreminski on narrative instruments, and Ryan on simulated life-worlds.)

Community-led discovery

Are instruments self-documenting? How does a community of practice remember?

Interactive articles and video games are both generally meant to be widely accessible, functioning as casual creator support tools. [Cite dissertation zine of Kate Compton.] Other software instruments are not only expected to take years to learn, but they may not even be usable without expert instruction.

For instance, it is unreasonable to expect to learn how to play a guitar without ever receiving instruction from a guitar player. At the same time, many expert guitar players are largely self-taught, because they are highly motivated to play with the instrument, and have discovered many features of its expressive range in that time.

In the course of their practice, a guitar player has presumably also discovered many more action sequences that produce noise than produce songs. Where a beginner's action space may include verbs like 'press down this string at that fret' and 'ready my pick above this string,' an expert will have encapsulated the action sequences they have executed on hundreds of times, resulting in verbs like 'strum this chord' and 'arpeggiate this scale.'

Application programming interfaces (APIs)

Programming languages are software instruments that programmers can use to create software. In practice, it is script libraries (like NumPy and matplotlib) and game engines (like the HTML5 document object model) that are the instruments used to create software instruments.

A script library always consists of methods - certain action sequences that have

been encapsulated as their own verbs, - and objects, which are the data shapes those action sequences expect to operate on. Together, these comprise the API of the library.

Through this 'high-level' interface, the library functions as a programming instrument. But let's define what it means to be 'high-level,' rather than 'close to the machine.' If a well-chosen library is used in a program, then the score of that program has a lower dimensionality (# free parameters, approximated by # lines of code) than the same program written without those encapsulated library methods. [Cite Kolmogorov complexity here.]

This score is now more useful as a control surface for a programming practice, because it is harder to explore the space of all possible programs when you are busy doing code maintenance.

APIs are as difficult to design as any other instrument, which is why we will consider Data-Driven Documents (D3.js) as an exemplar in the second case study.

We restate the introduction in the terms of *data visualization*, and its attendant interests in science and industry.

Introduction

Data Problems

Expert knowledge that is based on data inevitably results in data that is high-dimensional and domain-specific. A physicist might cast the behavior of a car as a function over time in phase space (coupled appropriately to the behavior of other functions representing cars). A mechanical engineer might prefer to represent the car as an arrangement of physical linkages (an engine, a drivetrain, wheels). An experience designer would instead treat the car as an arrangement of driving-affordances exposed by its physical control surfaces (a gearshift, a steering wheel, pedals).

As in the parable of the blind men and the elephant, each of these experts has told us that the car has many different properties, some of them contradictory, such as if we neglect that the hard, metallic car body is different from the soft, leather seats. Here I will address the problem where each data representation is a dissimilar view of the elephant. Perhaps the dissimilar views represent different parts, so of course they're different. Or, perhaps the dissimilar views represent the same part (such as the algebra and the geometry of one system), in which case a data visualization tool can make sense of (and make use of) the translation between them.

That is, while high-dimensional data visualization is a powerful tool for understanding the behavior of complex systems, it is not agnostic to the knowledge domain (the ontology) from which that data arises.

Expert practitioners in a given domain will already be familiar with diverse visual representations of their data - ranging from standardized plots (e.g. in a paper), to

abstract geometric representations (e.g. on a blackboard), to diagnostic illustrations (e.g. the response of bacteria to a Gram stain), to metaphors they personally find useful. All of these representations are in dialog with each other and with certain models within each domain. Those models are valuable to us as tool designers, because they translate between representations.

Another source of models for high-dimensional data, besides explicit readings of expert knowledge, is statistical inference on a given dataset. Popular modelling techniques include 'training' a given machine learning (ML) architecture by performing gradient descent on a very large parameter space. These black-box modelling techniques are certainly data-driven, but are not easy to combine with theory-driven models and inferences.

A taxonomy of data problems in natural language processing (NLP), a field in which ML competes strongly with hand-coded networks of relations, is given by Halvey, Norvig, and Pereira [3]. In order to reconcile the data-driven and theory-driven approaches to coding natural language (and generating it), they show that each cares about the same three orthogonal problems:

- choosing a representation language
- encoding a model in that language
- and performing inference on the model.

Data-driven modelling approaches rely on theory to make good choices of representation, while theory-driven modelling approaches rely on data to make inferences about the measurable world. ## Visible Representation Representation languages decompose artifacts into modular units. An artifact is an object of study. It has at least one kind of data representation, such as trophic networks in ecology (as graphs with weighted edges), or characters in a role-playing game (as character sheets), or crocheted accessories (as knitting patterns) [4].

Specific ML architectures typically incorporate domain-specific representation languages, such as kernel functions (in raster-based graphics), word embeddings (in NLP), or solutions to nonlinear dynamical systems (in reservoir computing). These

representation languages provide gains in data visualization, equally.

If you have plotted data before, you are familiar with first-class visual encodings (e.g. line charts, bar charts, heat maps) for distributions of scalar values varying along one or more dimensions. How should we visually encode non-scalar values, such as kernel functions (as weight matrices?), or word embeddings (as semantic vectors?), or reservoir elements (as neural networks with random weights)? As soon as our values are more complex than scalar quantities, the conventions of axes and color scales become inadequate.

How shall we represent a distribution of non-scalar values, let alone a model, which is a rich parameterization of such distributions? In the domain of procedural generation, the visual analysis of a design space of artifacts (e.g. playable levels in a game) may take the form of an expressive range analysis [5]. I will build on this idea, that distributions of artifacts require first-class visual encodings, to motivate the concept of explorable models as implemented by software instruments.

In this literature review, we will draw on prior theories of embodied interaction and hypertextuality to discuss already-existing evaluation-execution loops conducted by experts with their data, to understand its models, producing both descriptions (representations) and inferences (analyses).

We will discover sets of operational logics that are used by these experts, including well-known patterns such as 'searching for' datum meeting certain criteria, and subsequently 'inspecting' these events or rules or artifacts. When we implement these logics into data-visualizing software, those will function as instruments of analysis.

Remarkably, these are very much the same operational logics as we encounter in instruments of expression, which allow experts to traverse an expressive range, embedded in a design space, to locate artifacts (or events, or rules). In other words, an expert navigates a model, embedded in a latent space, to locate a datum. ## Contents of This Review In the following sections of this document, I will frame the problem of research debt in terms of design spaces, whose navigation is accom-

plished via software instruments. In particular, the analysis of specific artifacts is in conversation with the design of models from which they arise. Likewise, the design of specific artifacts is in conversation with the analysis of models from which they arise. Models connect the worlds of design and of analysis.

The first part of the background section concerns a case study in shared experiences of navigating design space, namely the Annals of the Parrigues, written by Emily Short [2]. We will read its appendix, which recounts designing a medium of expression whilst navigating the space it describes, against generative design mediums including Logo sketches and Twitterbots.

The second part of the background section of this essay will address the specific problem of 'research debt,' coined in analogy to technical debt by Olah and Carter [1]. Along the way, we'll critique mathematics pedagogy, learn to tune a platformer game in real time, and discover a better way to code interactive artifacts.

In the case studies section, we will explore an avatar creation tool from a videogame, and a web-based library for binding data to visual representations.

In the final, theoretical section of this essay, we will introduce terminology for practitioners who intend to develop novel or existing software instruments, and to communicate their results through landmarks, scores, and other methods of orientation in design space using software instruments.

[1] https://distill.pub/2017/research-debt/

[2] https://emshort.blog/2015/12/07/procjam-entries-nanogenmo-and-my-generated-generation-guidebook/

[3] https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/35179.pdf

[4] https://aiweirdness.com/post/173096796277/skyknit-when-knitters-teamed-up-with-a-neural

[5] https://web.archive.org/web/20120306215536/http://games.soe.ucsc.edu/sites/default/files/smith-expressiverange-fdgpcg10.pdf

— title: Software Instruments - Performance excerpt: "This week is about design

space, and its navigation via generative instruments." author: Jasmine Otto —

This week is about design space, and its navigation via software instruments. This is the second background section of the series. Next week, we will have the first background section on research debt, which wasn't meant to be self-demonstrating in its indefinite deferral.

Pens and turtles

Students are often introduced to imperative programming using the Logo language introduced by Papert, which provides a software instrument called a 'turtle.'

A turtle carries a pen, which traces its movement on the canvas. The turtle can be directed to move forward, or to turn by a certain amount, and to repeat a block of such instructions a given number of times.

It turns out that many remarkable figures can be produced by stringing together these instructions and varying their parameterization. Logo is a prototypical example of a 'generative system.'

The Logo environment converts a sequence of instructions into a turtle-made drawing on the screen. The programmer converts this drawing into a modified set of instructions, because they are trying to make or discover another form. This execution-evaluation loop (per Norman) comprises an exploration of the artifact space (see Compton on expressive range) of Logo.

Practicing with the turtle

All instruments participate in an execution-evaluation loop, as exemplified in 1963 by Sketchpad (cite Sutherland). The loop is the functional unit in a cybernetic system, which consists of a set of subjects, and a set of arcs of interaction between them. For example, a programmer and their Logo environment together comprise a cybernetic system, because the environment executes the programmer's expression of their intent (in terms of Logo affordances), and the programmer evaluates whether the thing they intended really happened or if their instructions went awry somehow.

A (cybernetic) performance comprises of an inner 'practice' loop, involving the player and their instrument, and an outer 'presentation' loop, involving the practice and its audience.

Practices may be improvised (by the player selecting their next action based on the response they anticipate), based on a score (the player selects their next action by consulting a script), or typically combining the two (when the script underspecifies the next action). Performances inherit this property of the practice. In these cases, the player is likely to account for the audience in both their action selection (at the moment-to-moment level of improvisation) and their choice of scores (if they are aware of such choices).

In Reveal Codes, Rita Raley characterizes the navigation of hypertext as a practice with a complex system, one that "functions to separate the digital from the analog," which is echoed in our distinction between the model (whose instance is an artifact) in a software instrument, and its referent which cannot be represented as data.

(Raley describes 'performance' as sequences of actions with a hypertextual instrument but not necessarily an audience, so we will call this sense *practice*. She also describes 'practice' as a form of reading and writing hypertext, implying a long-form conversation within a hypertext community, so we will call this sense *performance*.)

We can substitute any artifact for Raley's hypertext, only if it is the object of a software instrument. We will find that this instrument's expressive range is a complex meaning-making system, as she has described.

Performing with the turtle

What is so surprising about practices - that is, action sequences taken in an interaction loop with the artifact that they describe, - is that they illuminate the 'grain of the medium' (cite Barthes), i.e. an attractor structure on the artifact space.

This is most clearly demonstrated by the anecdote about 'failing quickly' as a pedagogical method. Those students who were instructed to create one hundred pots over a semester, were creating higher-quality pots by the end of it than students

who had focused their entire energies that semester on one pot.

Recall that generative systems, including Logo and similarly 'rich output' programming environments, describe artifacts in terms of action sequences written in a domain-specific language (e.g. the turtle's vocabulary of 'move forward this far,' 'turn by this many degrees,' 'set pen up or down'). We call these descriptions 'scores,' after the scripts that musicians consult to help them learn a given song.

Musical scores are incomplete, because many inflections and dynamics in the song as it is played are not written down and must be learned 'by ear,' i.e. imitation of a presently-existing instance of the song, which is transient. Also, scores are partially transferrable between instruments, as it is possible to play a score written for one instrument on a related instrument, but only by skillfully modifying the score to not make it unrecognizable.

We will discover these properties in software instruments later. For now, imagine the difficulty of learning Photoshop without seeing someone's demo, or of re-writing a Python script into Javascript.

Analysis of turtle-like systems

Emily Short's characterization of the generative meta-medium (and specifically, her own hypertextual machine collaborator), in the appendix to Annals of the Parrigues, breaks it into the following five complementary lenses ('aspects'):

- Salt, which governs structured processes like crystal growth and tilings.
- Mushroom, which governs branching processes like rhizomic growth and Markov chains.
- Venom, which governs spiky distributions of semantic weight, like when 'distribution' and 'semantics' appear in a sentence together.
- Beeswax, which governs mass-curated processes and crowdsourced collections.
- Egg, which governs singly-curated processes and moments of authorship.

These lenses of analysis apply to instruments, which impose on generative processes enacted with them in related ways. For instace, salt implies structure, mushroom

implies self-propulsive energy, and venom implies standout moments. Now think of a generative text which continuously produces utterances that use the same grammatical structure, but which vary in their payload, from 'blossom petals' to 'nuclear runoff.'

By combining aspects and a medium, we have just described a Twitterbot, a kind of artifact described (along with its accompanying creative scene in the 2010s) by Veales and Cook at length. Beeswax seems to describe the 'like' count of a generated tweet, while egg describes the choice by someone to retweet it. Salt, then, also describes the predictable layout of the bot's Twitter page.

We should not be surprised that a characterization of the 'practice' (i.e. bot-making) is equally applicable to the 'performance' (i.e. Twitter presence), which is another interaction loop inside which practice is nested.

It is not possible to circumscribe the possible action sequences with a given instrument in the abstract. Although practice and theory (which we will discuss next section!) together indicate 'rules of design' (goals and constraints on action sequences) that make good artifacts easier to find, these rules are almost never hard and fast.

For example, one affordance of a guitar is to play semitones on the fretboard. Because it sounds dissonant if you play notes together that do not share a scale (a patterned subset of semitones, characterized as a sequence of intervals), so players learn to move within e.g. the blues scale. As long as the key of the song doesn't change, the player can "improvise freely" over the matching blues scale, meaning that they can choose arbitrarily between next actions that follow the rule without making any mistake. But it is equally important that this rule can be learned, and that it can be unlearned - because it would be very boring to play only one blues scale all the time, and besides, you might want to play in a different key.

With our intuition pump primed in this way, we hope to have motivated a future research project of describing software instruments, whose operational logics serve to navigate the 'rugged landscape' of artifact space.

References