From Simple to Easy

Andrew Cotter

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

Acknowledgements

Thesis Advisors:

Aya Karpinska

Loretta Wolozin

Jesse Harding

Chris Prentice

Special Thanks:

Victoria Hackett

Ramsey Nasser

Sarah Groff Henneigh-Palermo

Zach Lieberman

Justin Bakse

Bryan Ma

Nick Montfort

Abstract

In programming education, students are often first introduced to the simplest concepts of a language before more complex topics. Their initial understanding of data types is applied to understand logic statements, which then abstract up into functions and classes. This approach is often referred to as the 'Procedure-First' method. This problem with this approach arises from a semantic misunderstanding of the difference between 'simple' and 'easy.' Definitionally, something simple is irreducibly complex, meaning that it can't be broken down further. Conversely, something easy is familiar or ready-at-hand. In this way, the progression from simple to complex isn't necessarily congruent to the progression of easy to hard. In this way, what is complex in programming can be easier to understand than what is simple. This is also in line with how people learn many other processes. If you want to teach someone how to drive a car, you don't start with the lower levels of machinery like the carburetor, you start from the highest level of abstraction: the steering wheel. This method of teaching programming is known as the 'Model-First' approach.

Grokking Creative Code aims to test and apply the Model-First approach as an online book of tutorials with interactive sketches, which can help independent learners advance from a post-beginner skill level into an intermediate level. I have been testing this method by showing the tutorials I've written to fellow students. Before and after they go through the tutorial, they describe how they would approach the problem that the lesson addresses. Although an imperfect method of evaluation, this is how I've been able to gauge the relative degrees in confidence and understanding that they've gained. With this feedback in hand, my ambition is to thoughtfully implement the Model-First method of teaching, help popularize the method, and promote its broad application to other levels of creative coding.

Table of Contents

Introduction

The Journey of Learning Creative Code

In my experience of learning creative coding, there was a two-year span of time in which I was feeling like I wasn't getting better at programming. My level of skill was past that of a beginner, but still below what would be considered 'intermediate.' During this plateau, I was spending a lot of my time unlearning what I thought I knew about programming. I had been introduced to functional and object-oriented programming by then, but I was still working in the mindset of procedural programming, trying to solve problems just with variables and if statements. It wasn't until my second reading of *The Nature of Code* by Dan Shiffman that I had really started to internalize what object-oriented programming was about.

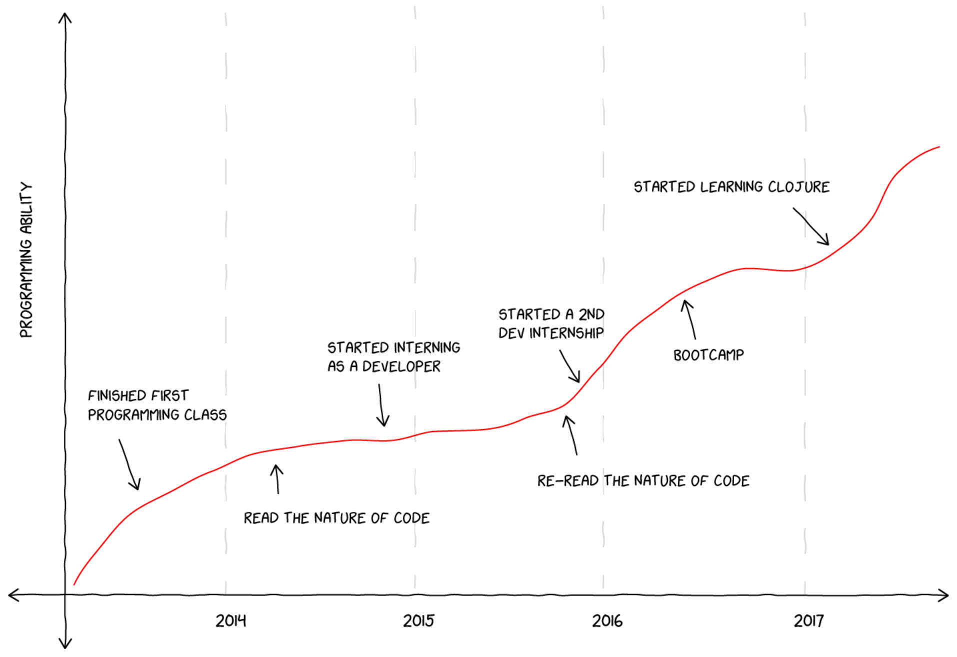


Fig. 1 My Person progression of learning programming over the last five years

This is a process that many of my peers also went through as well. In my practice of programming both in professional and educational environments, I have consistently noticed that many of my peers reach a plateau in skill and understanding for an extended period of time. Typically, students and independent learners at this level have also begun to touch functional and object-oriented code like I had, but still don't have a full understanding of classes or more abstract structures of software.

Pain Points for Students

The length of this is process of going from beginner to intermediate is due in part to a lack of resources that target that skill level. While there are many learning resources that target complete beginners and people who have been programming for many years, there are very few books or courses aimed at people who are in the liminal space between.

In my interview with Sarah Groff Hennigh-Palermo, a self-taught programmer and Engineer at Kickstarter, she recounted to me many of the pain points which exist in the documentation of languages and frameworks. As she described it, many bodies of documentation for languages or frameworks just focus on giving a concise description of what each function or class does discretely, falling short of describing how those functions and objects collate together into a cohesive workflow or process.

The Procedure-First Method

Furthermore, the difficulty I and my peers have been experiencing is due to the shortfalls of learning within the framework of a Procedure-First pedagogical approach. Although this method has been widely used in engineering and computer-science courses, I don't believe it serves people coming from an arts or creative background.

Structurally, the Procedure-First approach first presents students with the simplest parts of a programming language (variables, if statements) before proceeding to the more complex and abstract parts of a programming language (functions and objects). However, as students progress up the ladder of abstraction, they also have to discard parts of what they thought they knew about programming. This is because most of the concepts at the imperative level of code don't necessarily scale up to the systematic levels of programming.

For example, a common challenge given to beginner programmers is the “fizz-buzz” challenge, which has students count up to 100 and print “fizz” if a number is divisible by 3 and “buzz” if it is divisible by 5. This tests the student’s understanding of booleans and loops.

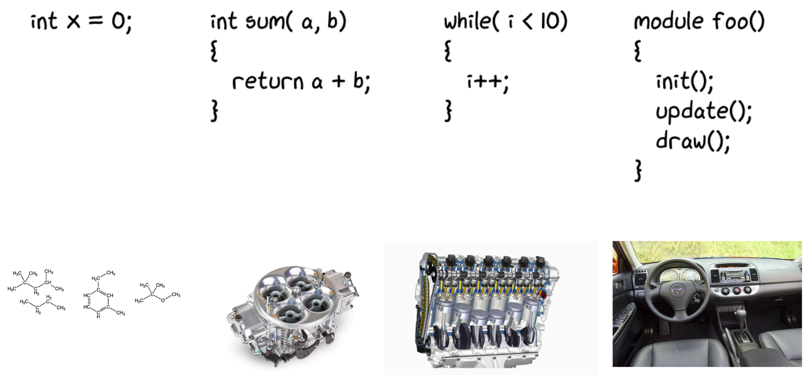
Typically, the next challenge step up from there for creative coding is something like a particle system. This involves making a Particle class with a position, speed, and velocity, then storing several instances of the particle in an array and looping through them to update their positions. This requires an understanding of vectors, some physics, and how to manage arrays.

The understanding required for these tasks have almost nothing in common, and as a result, the simpler and more trivial assignments given to beginner students is almost detrimental

This progression from simple to complex is also not the same as a progression from easy to hard. In his 2011 talk, *Simple Made Easy*, computer scientist and progenitor of Clojure Rich Hickey makes an insightful distinction between the two:

"So the first word is simple. And the roots of this word are sim and plex, and that means one fold or one braid or twist. The other word we frequently use interchangeably with simple is the word easy. And the derivation there is to a French word ... which means to lie near and to be nearby."

What Rich is getting at is that simple concepts are *irreducible* (or at least less reducible) and easy concepts are familiar and at least partly known already. In this way, teaching someone how to drive a car with the procedure first method would start from the chemistry of hydrocarbons and slowly work up to the steering wheel and the gas pedal.



```
int x = 0;
```

```
int sum( a, b)
{
   return a + b;
}
```

```
while( i < 10)
{
   i++;
}
```

```
module foo()
{
   init();
   update();
   draw();
}
```

Fig. 2 Comparing the elements of programming to the elements of driving.

While this progression might be useful to an engineer or mechanic, it's not an expedient way to learn how to drive if all you want to do is pick something up at the grocery store. Furthermore, if this were the only method of learning to drive, the only people who would drive cars would be engineers and mechanics.

To address this, I am writing a series of online interactive lessons and video tutorials aimed at people stuck in this post-beginner / pre-intermediate stage. Specifically, I am aiming to fill the pedagogical niche between bodies of work like Daniel Shiffman's *The Coding Train*, which targets beginners in creative coding, and works for intermediate programmers like *The Book of Shaders* by Patricio Gonzalez-Vivo. I am further differentiating my work by implementing a Model-First approach of teaching.

The Model-First Method

The Model-First method is defined by teaching a concept by starting with a wholistic and abstracted overview of how something works, then chunking out the concept into parts until the instructions have drilled down to the simplest level. In programming this usually involves looking at the entire problem objective, modeling and describing how it works, then finally translating that model into code, using the model as a framework or outline.

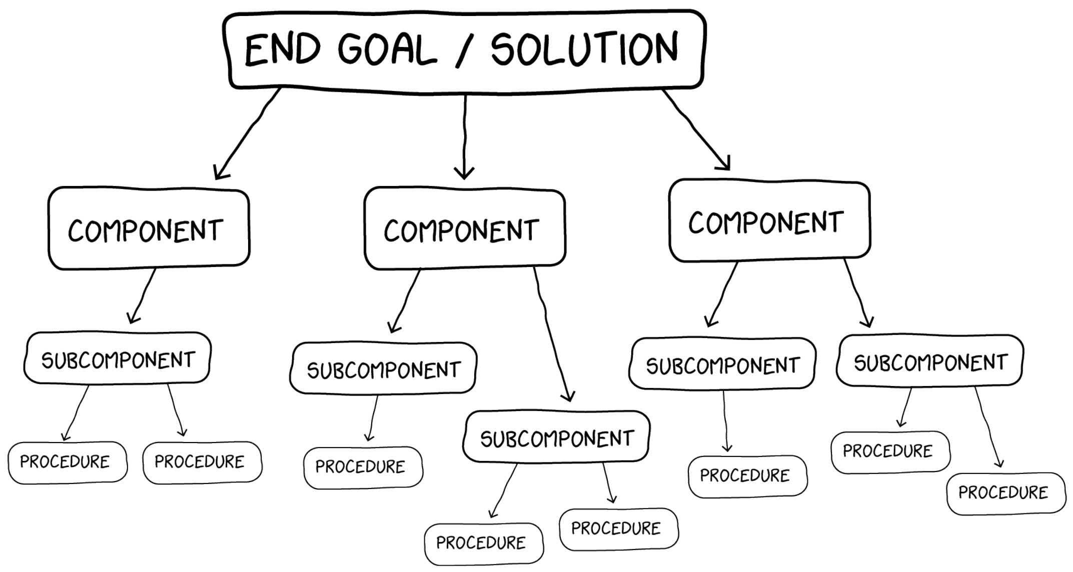


Fig. 3 The structure of the Model-First Method

In *Programming in Context – A Model-First Approach to CS1*, authors Jens Bendeson and Michael E. Casperson define the Model-First approach as:

- Instructing the computer: The programming language is viewed as a high-level machine language. The focus is on aspects of program execution such as storage layout, control flow and persistence. In the following we also refer to this perspective as coding.
- Managing the program description: The programming language is used for an overview and understanding of the entire program. The focus is on aspects such as visibility, encapsulation, modularity, separate compilation.
- Conceptual modelling: The programming language is used for expressing concepts and structures. The focus is on constructs for describing concepts and phenomena.

In much the same way as Bendeson and Casperson outline, my tutorials begin by showing an example of a finished project, define the project in terms of its constituent elements, describes the qualities and behaviors of those elements, then translate those elements into classes that can be written in code.

Through writing, testing, and refining these tutorials, I am seeking to develop a successful implementation of the Model-First method that can properly bridge the gap between beginner and intermediate creative coding.

Grokking Creative Code

http://thatcotter.github.io/thesis

In my current iteration of the project, the non-linear nature of the website is relational among the tutorials rather than among the modules and submodules of the tutorials themselves. The tutorials are structured around three tentpole concepts. In the book, they are referred to as "space, time, and form," but these are amicable stand-in terms for "linear algebra, concurrency, and data architecture."

The tutorials are also structured in a non-linear fashion. There is no 'first' chapter or lesson to read past the introduction, and they can be completed in any order. All of the concepts addressed in the lessons are of a similar difficulty level and any minor pre-requisites from other lessons can be easily linked to easily. This also meant that I had to assume a post-beginner level of familiarity with programming since I wouldn't be dedicating any material to covering topics like syntax.

The key to breaking up the tutorials in this way was to not make the learner start from scratch. In most other programming tutorials, the reader/viewer is given a blank canvas at the beginning, which slowly builds up to the finished product. However, this process often means that many corollary problems need to be solved on the way to the primary problem or concept the tutorial is trying to get to. The tutorials in my current iteration give the reader a pre-existing project which needs to be altered in some way. In this way, the reader can focus on fewer problems per tutorial without having to think about them in a vacuum.

These tutorials were also accompanied by interactive live-code sketches that the student could edit and observe in real time. In this way, the student is able to work in a tighter feedback loop and see what they're doing as they're doing it.

Domains & Precedents

What's out there - Freestanding Resources

In my investigations of existing resources that target the teaching of math and programming creatively (or, conversely, the teaching of creativity formally)

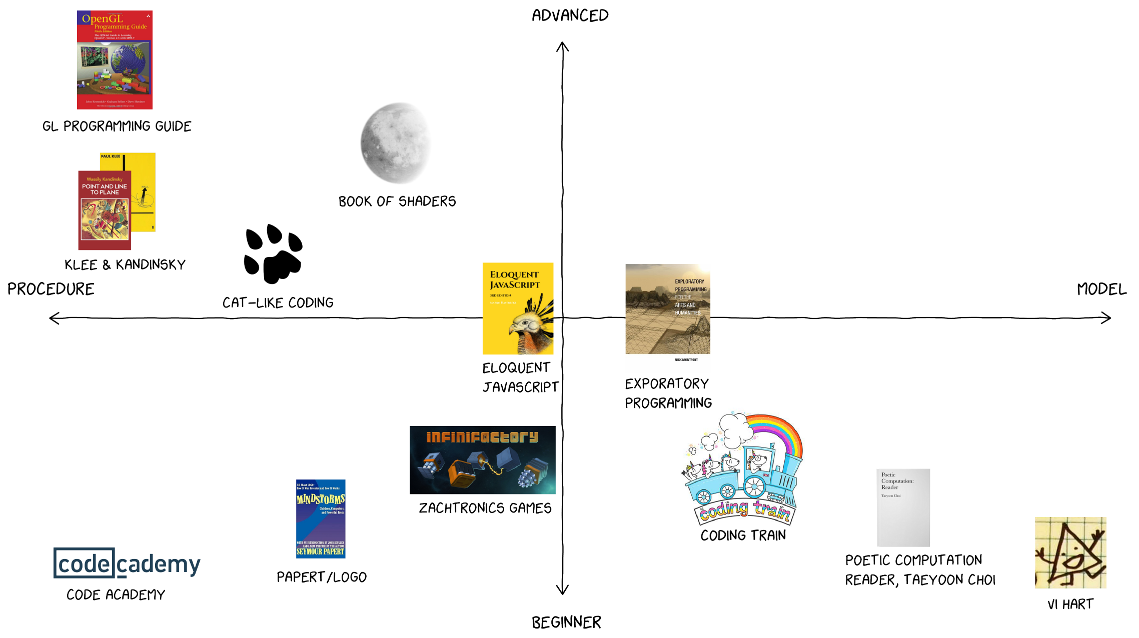


Fig. 4-1 A Qualitative map of some exemplar learning resources for learning programming, from procedure-first to model-first and beginner to advanced.

Beginner & Procedure-First

In the bottom-right, there are a few exemplars of beginner resources that at procedure first. Codeacademy is the lowest-lying fruit for this essay to critique, as it is a resource which merely teaches the syntax of a programming language without teaching anything in the realm of programmatic problem-solving or thinking about the systems or workflow outside of that syntax. It's just learning to code, not learning to program software. There are similar resources in this corner like programming bootcamps, which can extend a little higher and to the right in this grid, but are still largely localized in this corner.

Seymour Papert, his programming language LOGO, and related learning resources like Scratch, are examples of works that have tried to move the needle on the practice of just teaching syntax and representing the semantics of programming visually. As a drawing tool, LOGO operates with the metaphor of moving a turtle with a pen around the screen. Users are able to input commands like "right" and "left" to turn the turtle a specified number of degrees, "penup" and "pendown" for the turtle to raise and lower the pen from the ground, and "forward" to have the turtle move a specified distance.

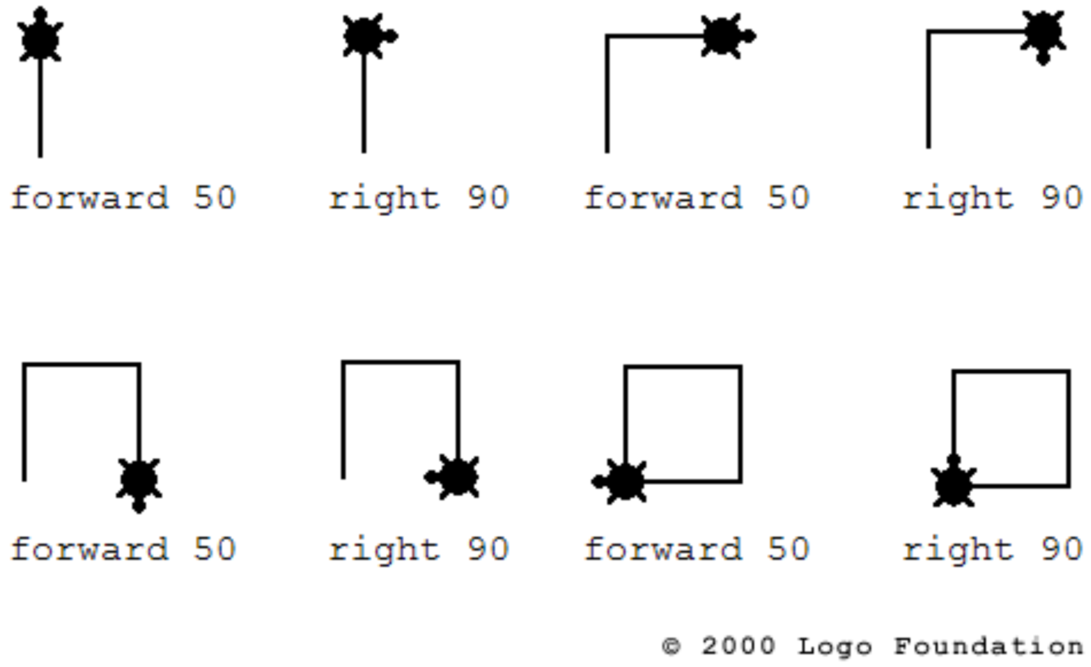
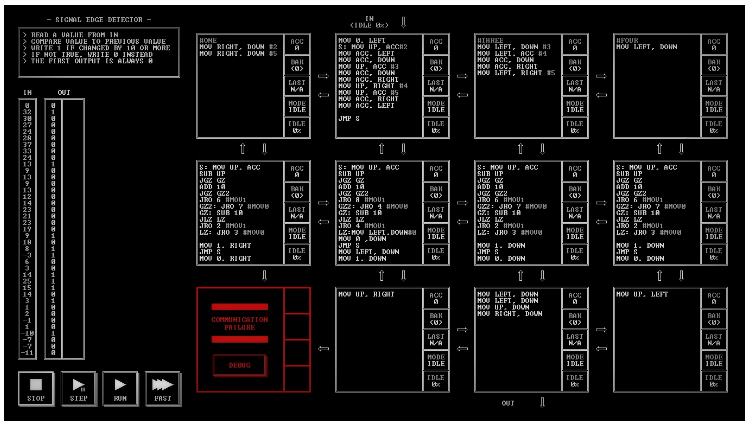


Fig. 5 An example of a turtle graphics program in LOGO

While this does help the user learn how to make instructions for the computer, the highest level of abstraction that the environment offers is the "repeat" command, which lets the user create loops

Zachtronics' games like Infinifactory, and their derivatives like Minecraft's Redstone mechanics, offer something a little further along the spectrum towards a model-first approach by giving players an idea of what the end solution space looks like and the ability to spatially manipulate the mechanisms that they design. In Zachtronics' TIS-100, or Opus Magnum, players are given a goal, and a system of tools with which to get to that goal. Furthermore, each new mechanic is matched a relatively non-trivial challenge in the level that introduces it. However, this is still ultimately a bottom-up approach, as many games tutorials are actually several pages worth of pdf documentation of the syntax of the puzzle systems that the player is solving.



```
- SIGNAL EDGE DETECTOR -

> READ A VALUE FROM IN
> COMPARE VALUE TO PREVIOUS VALUE
> WRITE 1 IF CHANGED BY 10 OR MORE
> IF NOT TRUE, WRITE 0 INSTEAD
> THE FIRST OUTPUT IS ALWAYS 0

 IN    OUT
  0     0
 32     1
 30     0
 27     0
 24     0
 28     0
 37     0
 33     0
 24     0
 13     1
  9     0
 13     0
  9     0
 13     0
 12     0
 14     0
 23     0
 21     0
 23     0
 19     0
  9     1
 18     0
  8     1
 -3     1
  6     0
  3     0
 14     1
 25     1
 15     1
 14     0
  3     1
  1     0
  2     0
 -1     0
  1     0
-10     1
 -7     0
 -7     0
-11     0
```

```
#ONE
MOV RIGHT, DOWN #2
MOV RIGHT, DOWN #5
```

```
MOV 0, LEFT
S: MOV UP, ACC#2
MOV ACC, LEFT
MOV ACC, DOWN
MOV UP, ACC #3
MOV ACC, DOWN
MOV ACC, RIGHT
MOV UP, RIGHT #4
MOV UP, ACC #5
MOV ACC, RIGHT
MOV ACC, LEFT

JMP S
```

```
#THREE
MOV LEFT, DOWN #3
MOV LEFT, ACC #4
MOV ACC, DOWN
MOV ACC, RIGHT
MOV LEFT, RIGHT #5
```

```
#FOUR
MOV LEFT, DOWN
```

```
S: MOV UP, ACC
SUB UP
JGZ GZ
ADD 10
JGZ GZ2
JRO 6 #MOV1
GZ2: JRO 7 #MOV0
GZ: SUB 10
JLZ LZ
JRO 2 #MOV1
LZ: JRO 3 #MOV0

MOV 1, RIGHT
JMP S
MOV 0, RIGHT
```

```
S: MOV UP, ACC
SUB UP
JGZ GZ
ADD 10
JGZ GZ2
JRO 8 #MOV1
GZ2: JRO 4 #MOV0
GZ: SUB 10
JLZ LZ
JRO 4 #MOV1
LZ:MOV LEFT,DOWN#0
MOV 0 ,DOWN
JMP S
MOV LEFT, DOWN
MOV 1, DOWN
```

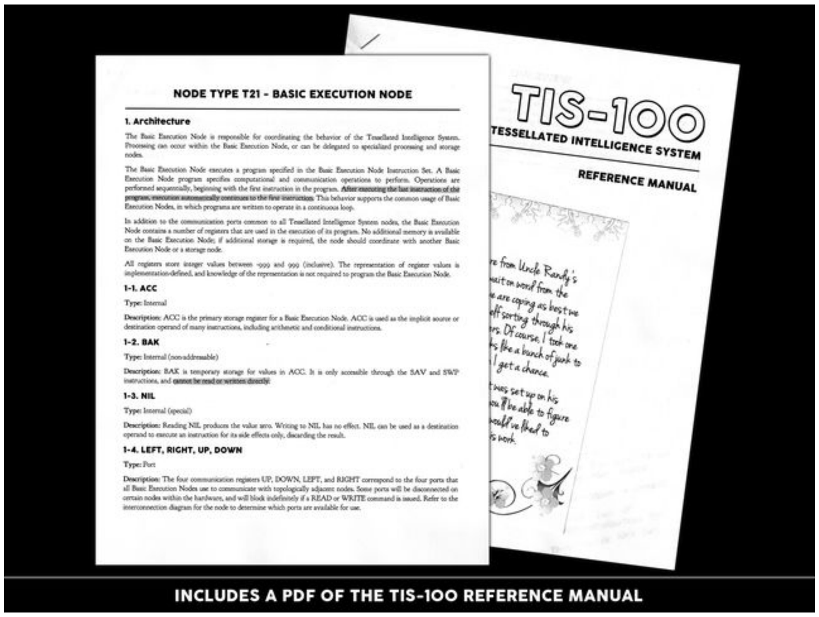
```
S: MOV UP, ACC
SUB UP
JGZ GZ
ADD 10
JGZ GZ2
JRO 6 #MOV1
GZ2: JRO 7 #MOV0
GZ: SUB 10
JLZ LZ
JRO 2 #MOV1
LZ: JRO 3 #MOV0

MOV 1, DOWN
JMP S
MOV 0, DOWN
```

```
S: MOV UP, ACC
SUB UP
JGZ GZ
ADD 10
JGZ GZ2
JRO 6 #MOV1
GZ2: JRO 7 #MOV0
GZ: SUB 10
JLZ LZ
JRO 2 #MOV1
LZ: JRO 3 #MOV0

MOV 1, DOWN
JMP S
MOV 0, DOWN
```

```
MOV UP, RIGHT
```

```
MOV LEFT, DOWN
MOV LEFT, DOWN
MOV UP, DOWN
MOV RIGHT, DOWN
```

```
MOV UP, LEFT
```



TIS-100
TESSELLATED INTELLIGENCE SYSTEM
REFERENCE MANUAL
INCLUDES A PDF OF THE TIS-100 REFERENCE MANUAL

NODE TYPE T21 - BASIC EXECUTION NODE

1. Architecture

The Basic Execution Node is responsible for coordinating the behavior of the Tessellated Intelligence System. Processing can occur within the Basic Execution Node, or can be delegated to specialized processing and storage nodes.

The Basic Execution Node executes a program specified in the Basic Execution Node Instruction Set. A Basic Execution Node program specifies computational and communication operations to perform. Operations are performed sequentially, beginning with the first instruction in the program. After executing the last instruction of the program, execution automatically continues to the first instruction. This behavior supports the common usage of Basic Execution Nodes, in which programs are written to operate in a continuous loop.

In addition to the communication ports common to all Tessellated Intelligence System nodes, the Basic Execution Node contains a number of registers that are used in the execution of its program. No additional memory is available on the Basic Execution Node; if additional storage is required, the node should coordinate with another Basic Execution Node or a storage node.

All registers store integer values between -999 and 999 (inclusive). The representation of register values is implementation-defined, and knowledge of the representation is not required to program the Basic Execution Node.

1-1. ACC

Type: Internal

Description: ACC is the primary storage register for a Basic Execution Node. ACC is used as the implicit source or destination operand of many instructions, including arithmetic and conditional instructions.

1-2. BAK

Type: Internal (non-addressable)

Description: BAK is temporary storage for values in ACC. It is only accessible through the SAV and SWP instructions, and cannot be read or written directly.

1-3. NIL

Type: Internal (special)

Description: Reading NIL produces the value zero. Writing to NIL has no effect. NIL can be used as a destination operand to execute an instruction for its side effects only, discarding the result.

1-4. LEFT, RIGHT, UP, DOWN

Type: Port

Description: The four communication registers UP, DOWN, LEFT, and RIGHT correspond to the four ports that all Basic Execution Nodes use to communicate with topologically adjacent nodes. Some ports will be disconnected on certain nodes within the hardware, and will block indefinitely if a READ or WRITE command is issued. Refer to the interconnection diagram for the node to determine which ports are available for use.

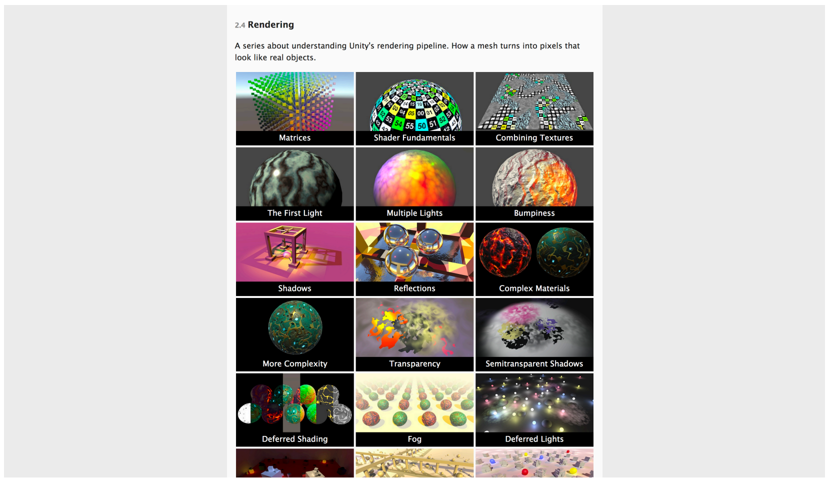
Fig. 6 Gameplay and promotional screenshots [from the Steam Store listing of TIS-100](#)

Advanced & Procedure-First

OpenGL Programming Guide (also colloquially referred to as "The Red Book") is a similar sort of low-lying fruit to critique as Code Academy. The book is written in a vocabulary which assumes the reader is an engineer, or is in the process of earning a Computer Science degree. Furthermore, the actual content merely covers a prescriptive "how" to program specific parts of OpenGL without much concern for the reasoning of why or a description of what possibilities are achievable. These questions of why and what are largely offloaded onto the reader to determine.

As a good counterpoint, *The Book of Shaders* by Patricio Gonzalez-Vivo has a vocabulary well-crafted for artists coming to lower-level graphics programming from the vantage of creative coding. In the first few pages, Patricio goes over the high-level concepts of what shaders are, why they are used, and how programming them is different than more conventional single-threaded programming with tools like Processing.

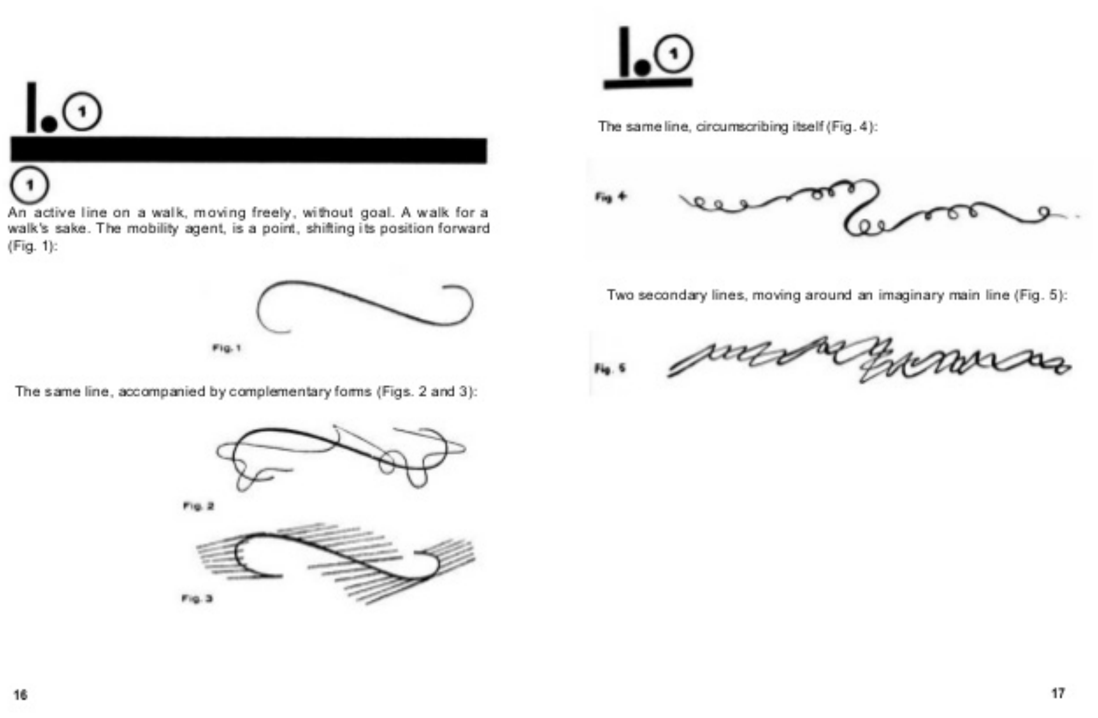
The *Catlike Coding* series by Jasper Flick fits squarely into the procedure-first category, but also gives readers the ability to skip around through different tutorials and sections based on the reader's interest. The actual verbiage of the tutorials is also very attentive, and provides footnotes based on questions that Jasper predicts that the reader might have. In this way, the layout of ideas becomes less linear and more explorable.


2.4 **Rendering**

A series about understanding Unity's rendering pipeline. How a mesh turns into pixels that look like real objects.

Fig. 7 screenshot of the tutorial overviews and links [from Catlike Coding]

The Pedagogical Sketchbook and *Point and Line to Plane* were included in this domain lineup because they informed a precursory project to *Grokking Creative Code*. In both works, the authors are reasoning about the creative practice of art in a way that could almost be described as "computational" or "mathematical." Both start from points as a unit of drawing and work up from there, working through the visual semantics of drawing. Although I didn't have the vocabulary for it at the time, their works both present a procedure-first approach to drawing.



An active line on a walk, moving freely, without goal. A walk for a walk's sake. The mobility agent, is a point, shifting its position forward (Fig. 1):

Fig. 1

The same line, accompanied by complementary forms (Figs. 2 and 3):

Fig. 2

Fig. 3

16

The same line, circumscribing itself (Fig. 4):

Fig. 4

Two secondary lines, moving around an imaginary main line (Fig. 5):

Fig. 5

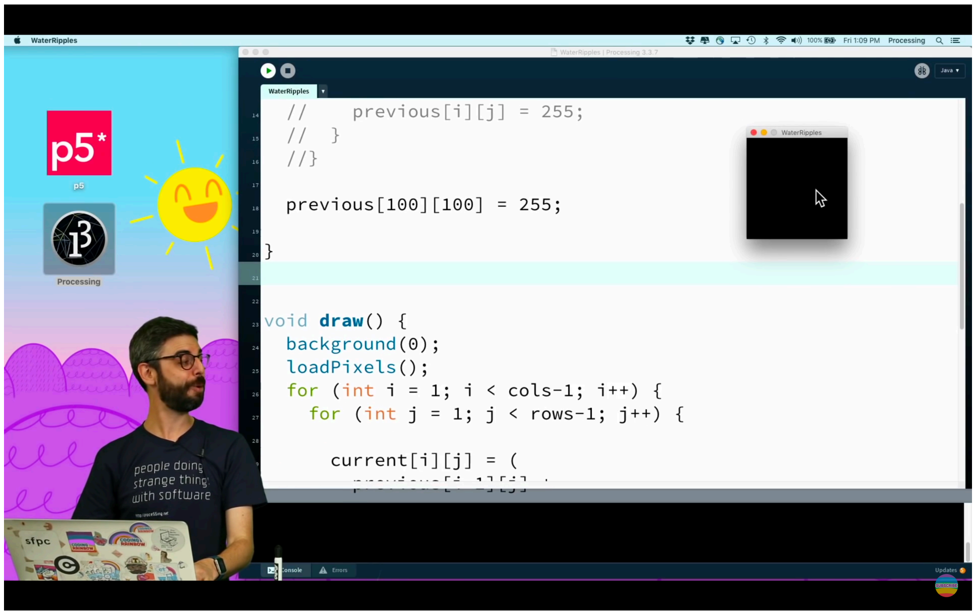
17

Fig. 8 The first two pages from The Pedagogical Sketchbook

Klee's work was also brought up by Zach Leiberman - developer of open frameworks and co-founder of The School For Poetic Computation (SFPC) - in one of my office hours interviews with him. Because of *The Pedagogical Sketchbook's* procedure-first approach to art, it loaned itself well to his class "Re-creating the Past," which focuses of using tools like openFrameworks to reverse-engineer the aesthetic and composition of older works that embody a systematic style of thinking.

Beginner & Model-First

The Coding Train by Dan Shiffman is one of the most comprehensive exemplars for model-first teaching; especially its series of weekly coding challenges. In the coding challenge tutorials, Dan completes a creative coding sketch in Processing or p5.js. Each tutorial is about twenty minutes long, and in the first few minutes, Dan shows the viewer both what the end goal looks like, and what problems need to be solved to get there. Because the videos are also live-streamed, viewers are also offered a chance to see Dan's thought process and contingencies for finding and resolving errors in his code when things go awry. One of the strengths and weaknesses of *The Coding Train* is the relatively low ceiling of difficulty to its content. This means that there are few outliers which would exclude Shiffman's audience, but that also means that there isn't a clear runway into the content past what he is teaching.



```
  //    previous[i][j] = 255;
  //  }
  //}

  previous[100][100] = 255;

}


void draw() {
  background(0);
  loadPixels();
  for (int i = 1; i < cols-1; i++) {
    for (int j = 1; j < rows-1; j++) {

      current[i][j] = (
```

Fig. 9 Screenshot of Dan Shiffman [from Coding Challenge #102: 2D Water Ripple]

In Tayeoon Choi's *The Poetic Computation Reader,* the audience gets a general overview of the history of computation and how it shadows a concurrent history of art. One of the standout features of the book is its use of footnotes to create a non-linear experience for the reader. In this way, the reader can explore topics laterally based on their interest to dig further into particular footnotes. This makes the book much more lattice-like in structure and encourages the reader to be curious and seek out more information when they are curious.

Although not directly about programming, Vi Hart's youtube channel presents math concepts in a fun way. In the channel's *Doodling in Math Class* series, Vi begins each video with "Say you're me and you're in math class…" before doodling in their notebook to distract theirself. Before long, Vi Hart's doodling leads into the math concepts that they were avoiding. In this way, their way of explaining these concepts start from concrete problems like "how do I draw the best spirals?" or "how do I make the squiggliest line?" In each exploration, Vi is working backwards from the leading question and discretizing it into a logical system, making their work a perfect exemplar of the model-first approach.

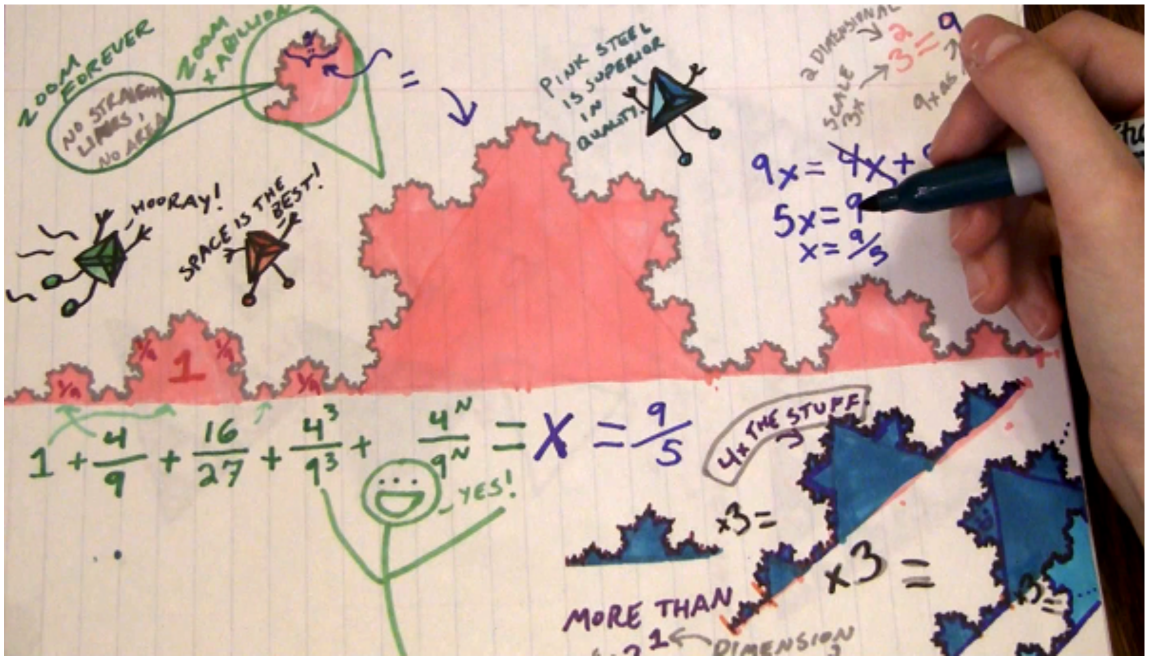


Fig. 10 screenshot from Doodling in Math Class: Dragons

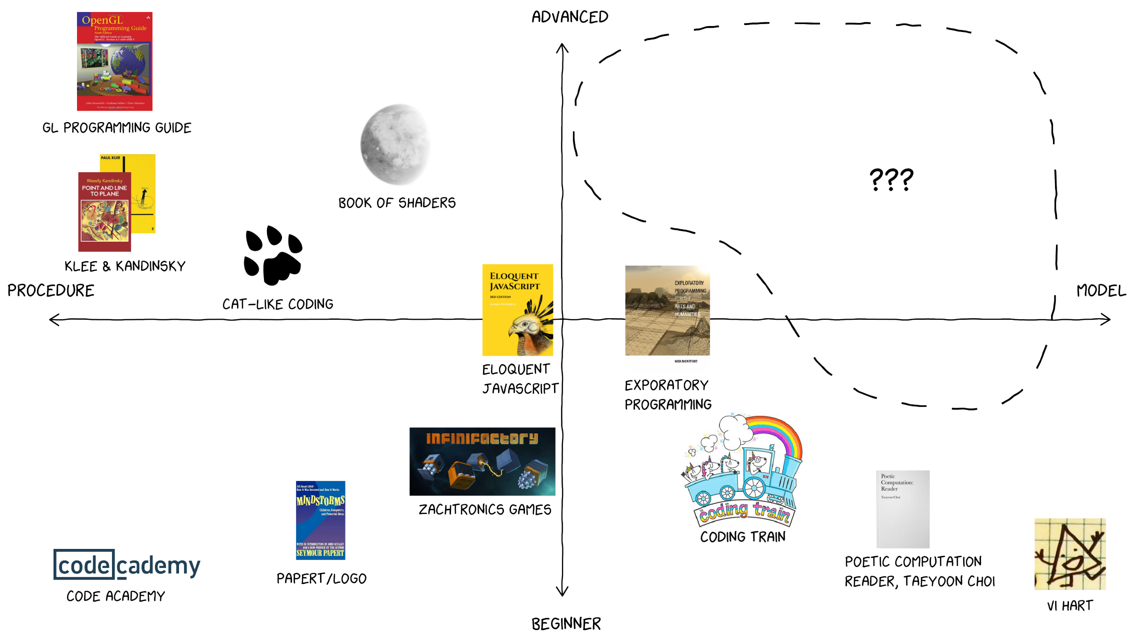


Fig. 4-2 A Qualitative map of some exemplar learning resources for learning programming, from procedure-first to model-first and beginner to advanced. Areas of opportunity are highlighted.

Advanced & Model-First

In this compilation of resources, I found that the upper-right quadrant of this domain space was relatively lacking. There are only a couple of works which felt like they could be partly placed here, and even then, there are some detractors which could disqualify them from being where they are.

Eloquent JavaScript does a good job of conceptualizing how javaScript works and how to make the best use of it. Each chapter and section approach hypothetical problems with a given topic in hand as a solution. However, the overall arc of the book still follows a procedure first approach; first covering variables and logic statements before getting into functional and object-oriented programming.

Nick Montfort's *Exploratory Programming* operates under the two key values of "programming as inquiry" and "programming as a practice." Montfort's notion of 'practice' is fairly straightforward; like the practice of the violin or painting, programming should also be practiced. What is more interesting is his notion of inquiry. Here, Montfort means that he wants the reader to continue from his lessons and "learn in whatever specific domains are interesting and compelling." In this way, *Exploratory Programming* and *The Poetic Computation Reader* One relatively minor shortcoming of the work is the lack of any visualizations or illustrations throughout the book to help the reader conceptualize of the problems and tutorials that they're working through.

Existing Creative Coding Tools

Similarly, I also surveyed the available tools for creative computation. As tools become more model-first, they also tend to become less open-source. This makes sense, as the more model-first tools here require more people to maintain them.

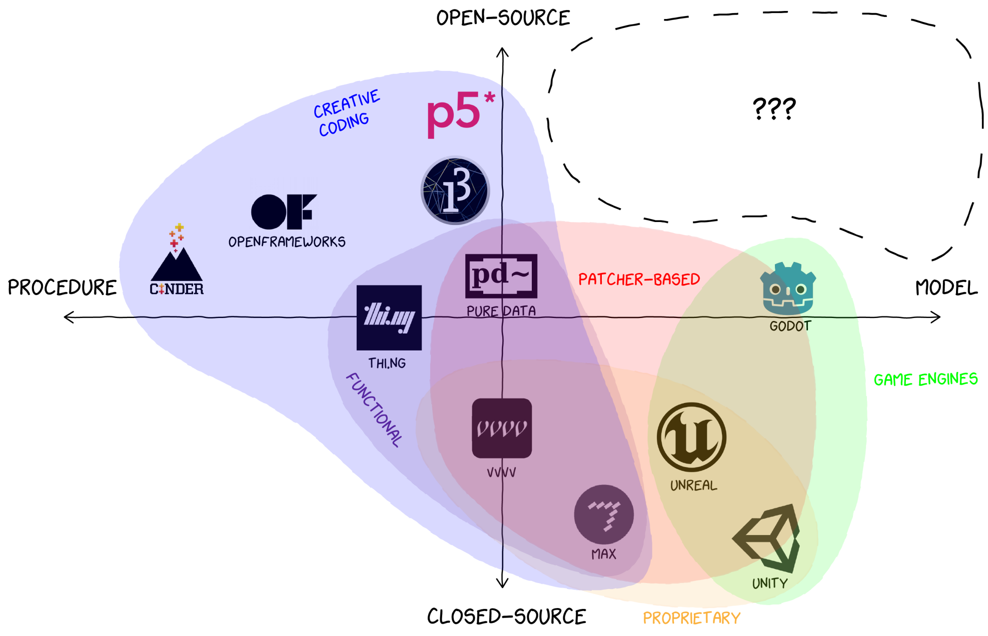


Fig. 11 A qualitative map of existing creative coding frameworks and game engines based on their qualities of being open/closed-source and procedure/model-first

This again leaves a large quadrant open to new tools, aside from Godot, there aren't many tools that can exist in the open-source space sustainably. Given how young Godot is, it will be interesting to see if it can stay in this space, or even thrive. Addressing a gap in tools though is a much larger undertaking than addressing a relatively unpopulated corner of the landscape of learning resources.

Finding a Niche

With that understanding of the gaps and underserved areas in existing learning materials for creative coders, it has been my aim to make a series of tutorials that extend into the model-first space, and past the beginner level. In this way, I am in essence designing a learning resource for my past self when I was having difficulty progressing past the beginner level.

Methodology

The Algorithmic Sketchbook

The first iteration of my project was a short series of tutorials / loose musings on creative coding based on a computational reading of Klee and Kandinsky. Because both artists have a procedure-first approach to art, I thought that they would be a good window to help other artists and creatives approach coding.

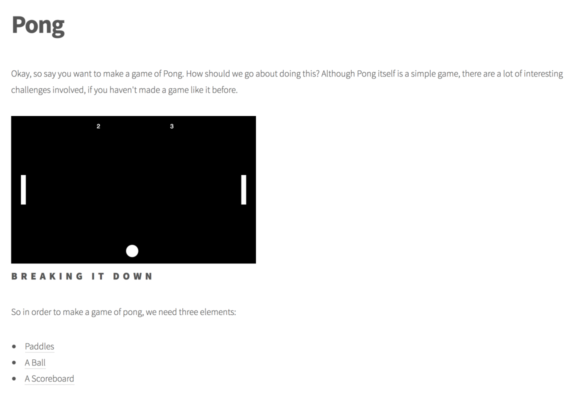
The poetic way in which Klee would imagine a line as "a dot that went for a walk" loaned a quality of playfulness to a well-structured framework of expression. Conversely, Kandinsky had a view of form that seemed much more mathematical and proof driven: "The geometric point is an invisible thing... Considered in term of substance, it equals zero." Both books have a fascinating cascade of concepts which arise into comprehensive treaties on the visual semantics of drawn forms.

However, this iteration was still a procedure-first approach to One of the strengths of starting from what's simple is that, simple concepts are also generalizable. However, the downside to generalization is abstraction. This is not to say that simplicity doesn't matter, but rather that starting from what is simple and abstract can alienate many students.

The most successful part of the project was a live p5 editor which updated programs in real time, and gave users instant feedback on what they were changing. In testing, the students I showed the website to were really excited to play around with the examples provided, but still didn't feel confident in making a p5 sketch from scratch. This is probably due mostly to the fact that I was still giving readers a procedure-first method of learning, which wasn't helping them get past the syntax and imperative mindset and into the semantic and systemic mindset.

Model-First Pong

After my investigation into model-first pedagogy as previously described by Bendeson and Casperson, I then wrote a pong tutorial based on my findings. The goal was to make the tutorial non-linear and broken up into modules which were self-similar to a model of a game of pong: three primary elements (a ball, two paddles, a scoreboard) broken up into their key properties and behaviors.



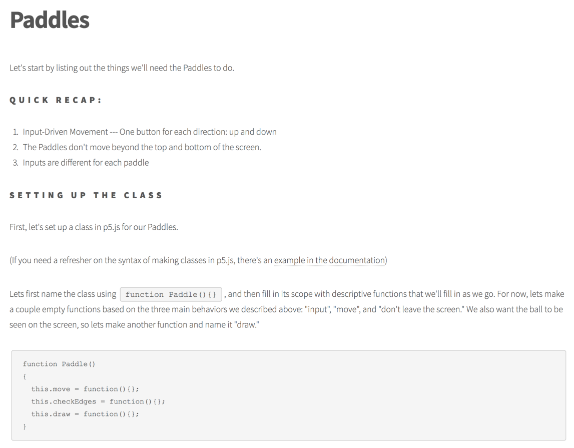
Pong

Okay, so say you want to make a game of Pong. How should we go about doing this? Although Pong itself is a simple game, there are a lot of interesting challenges involved, if you haven't made a game like it before.

BREAKING IT DOWN

So in order to make a game of pong, we need three elements:

- Paddles
- A Ball
- A Scoreboard



Paddles

Let's start by listing out the things we'll need the Paddles to do.

QUICK RECAP:

1. Input-Driven Movement --- One button for each direction: up and down
2. The Paddles don't move beyond the top and bottom of the screen.
3. Inputs are different for each paddle

SETTING UP THE CLASS

First, let's set up a class in p5.js for our Paddles.

(If you need a refresher on the syntax of making classes in p5.js, there's an example in the documentation)

Lets first name the class using `function Paddle(){}` , and then fill in its scope with descriptive functions that we'll fill in as we go. For now, lets make a couple empty functions based on the three main behaviors we described above: "input", "move", and "don't leave the screen." We also want the ball to be seen on the screen, so lets make another function and name it "draw."

```
function Paddle()
{
  this.move = function(){};
  this.checkEdges = function(){};
  this.draw = function(){};
}
```

Fig. 12 & 13 Screenshots from the Model-First Pong Tutorial

I was hoping that the structure of this tutorial would give readers a much more wholistic and top-down view of what they were working on. This exact approach had mixed success, however.

One of the key pieces findings from this iteration was that the tutorials were too non-linear. Navigationally, many users found it too much of a hassle to go back and forth within the same tutorial. In addition, this may have done more harm than good in helping them grasp the relationship between the content in each submodule.

User Feedback

This has led to my current iteration of *Grokking Creative Code*. Throughout each draft of the project, the component of live-coding has remained constant, but I feel confident that I have started to approach an appropriate balance between linear and non-linear tutorial structures in a model-first framework.

To test these tutorials, I've asked students/learners to first talk through or diagram how they would go about solving a problem before and after they one of my tutorials on the same subject. Typically, testers would have an idea of what resources they might look up, or how they might begin to solve the problem in a higher-level environment like Unity3D, but didn't know where to begin as far as writing down the code, or what functions/classes they would write.

After reading the tutorial, the testers would have much more realized ideas of how to solve the problems I asked them, citing which processes they would use and diagramming out how the parts of their program would fit together.

For example, before showing a tester a tutorial about frame-independent animation, I asked them how they would make an animation run at the same speed regardless of the frame rate. At first, they started to pseudocode out their thoughts: "What if I made a counter? Then iterated it in the draw loop? And progress the animation based on that counter?" As they fleshed out this idea, they realized that they were still falling into the pattern of frame-dependent animation.

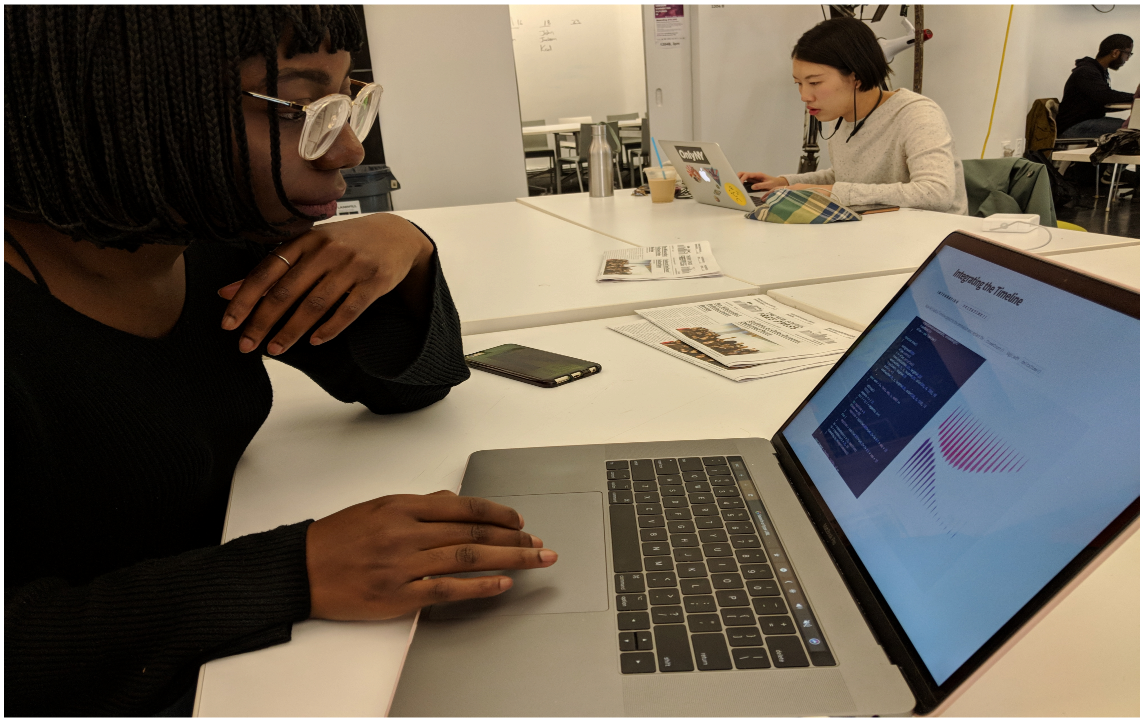
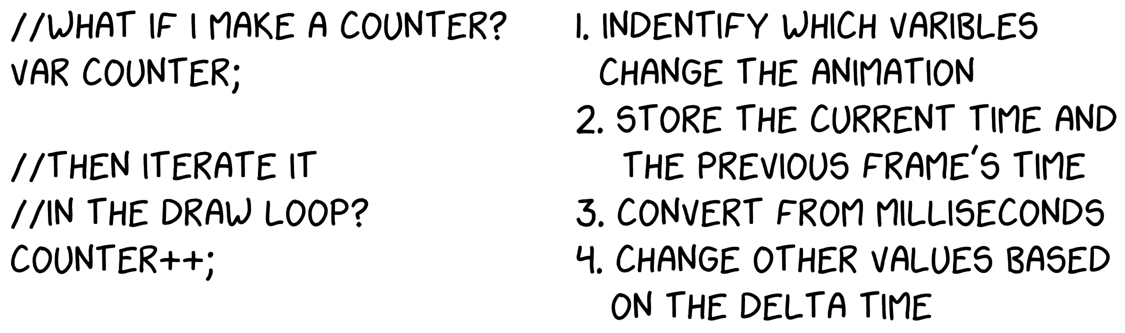


Fig. 14 Documentation from user feedback session



```
//WHAT IF I MAKE A COUNTER?
VAR COUNTER;

//THEN ITERATE IT
//IN THE DRAW LOOP?
COUNTER++;
```

1. INDENTIFY WHICH VARIBLES CHANGE THE ANIMATION
2. STORE THE CURRENT TIME AND THE PREVIOUS FRAME'S TIME
3. CONVERT FROM MILLISECONDS
4. CHANGE OTHER VALUES BASED ON THE DELTA TIME

Fig. 15 Transcribed Notes from the same user feedback session

A while after the student went through the tutorial, I asked them to conceptualize of the problem again. This time, they had a clear, step-by-step process of how they would go about it. Although the way they wrote out their thoughts was not in pseudo-code, it betrayed a more fundamental understanding of how the problem worked, and how the tools they had (in this case p5js) could address it.

This shift in thinking through the problem was an encouraging initial finding, and one I would like to be able to replicate in future lessons.

Evaluation

Going forward, I want to keep building out this body of tutorials that I have begun to sow. I have started laying the bricks for what is to come, but the final result is still somewhat uncertain, pending future user testing of what explanations make sense, and how non-linear I can make the flow of the lessons while still making them navigable. These are at least the two challenges that I can see going forward as I continue to foster this book.

Future Milestones

A year out from now, I expect to have reached a point of saturation with this body of work. By then, the Space section should provide a good overview of linear algebra (or at least a good visual intuition for it). This would impart concepts in 2D and 3D animation by digging deeper into the mechanics of vector and matrix math.

The Time section will be able to help a reader make their own animation library for p5.js. This progression is inspired from the structure of Zach's Algorithmic Animation course as he described it to me in our office hours meeting. His class was intended to teach students techniques for animating with code, and over time helping those techniques coalesce into a full library that the students could re-use or share.

The Form section will give readers a general understanding of Model-View-Control structures of code and some design patterns. This will be the section of the book most focused on helping the reader understand the wider context of systemic reasoning in programming by helping the reader model larger programs which manage multiple discrete tasks or sub-tasks.

I will also have a better idea as to which way to take my tutorials in terms of demographics. Although it should be possible to continue to drive the difficulty level up to more advanced topics, its also feasible to curb-cut the difficulty down to a neophyte reader. Both options offer compelling challenges, but it may be a more worthwhile test of the model-first paradigm to see how it makes contact with a complete beginner who has not yet internalized the syntax of any programming language.

As I reach these milestones, I also plan on making the tutorials a bit more non-linear, letting readers explore asides and footnotes laterally based on their curiosity. For example, since many of the Space tutorials will incorporate linear algebra, it might be a good idea to make independent pages about vector maths. These pages can be linked to from tutorial pages that touch upon these concepts so that the reader may establish or refresh their understanding of the content.

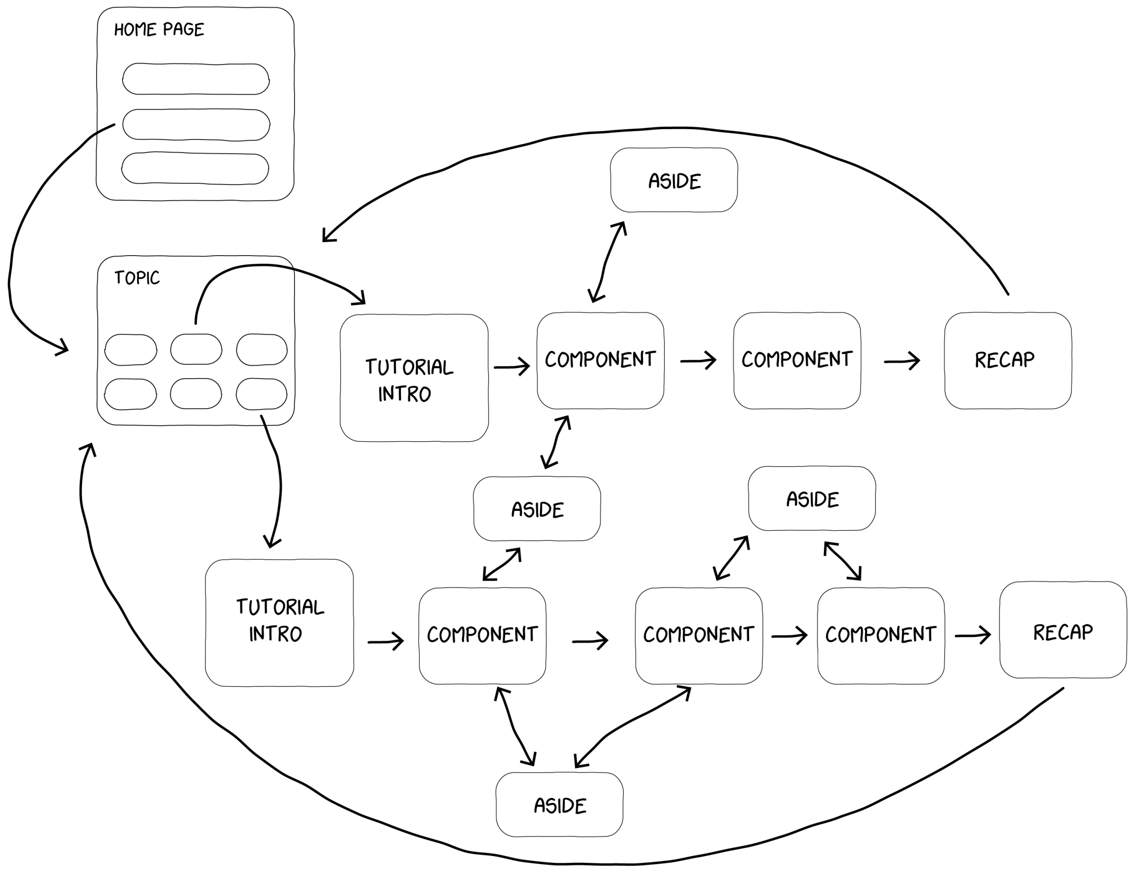


Fig. 16 Intended workflow of the Grokking Creative Code Website when it is complete

After reaching this milestone, I will be able to distill and synthesize some as-yet-unknown key takeaways to apply to my next goal: Creating a model-first tool for creative coding which can be sustainably open-sourced.

A Truly Model-First Framework

As pointed out earlier, there are already model-first tools for creative coders. However, these tools are all proprietary and closed-source, the notable exception being Godot*.

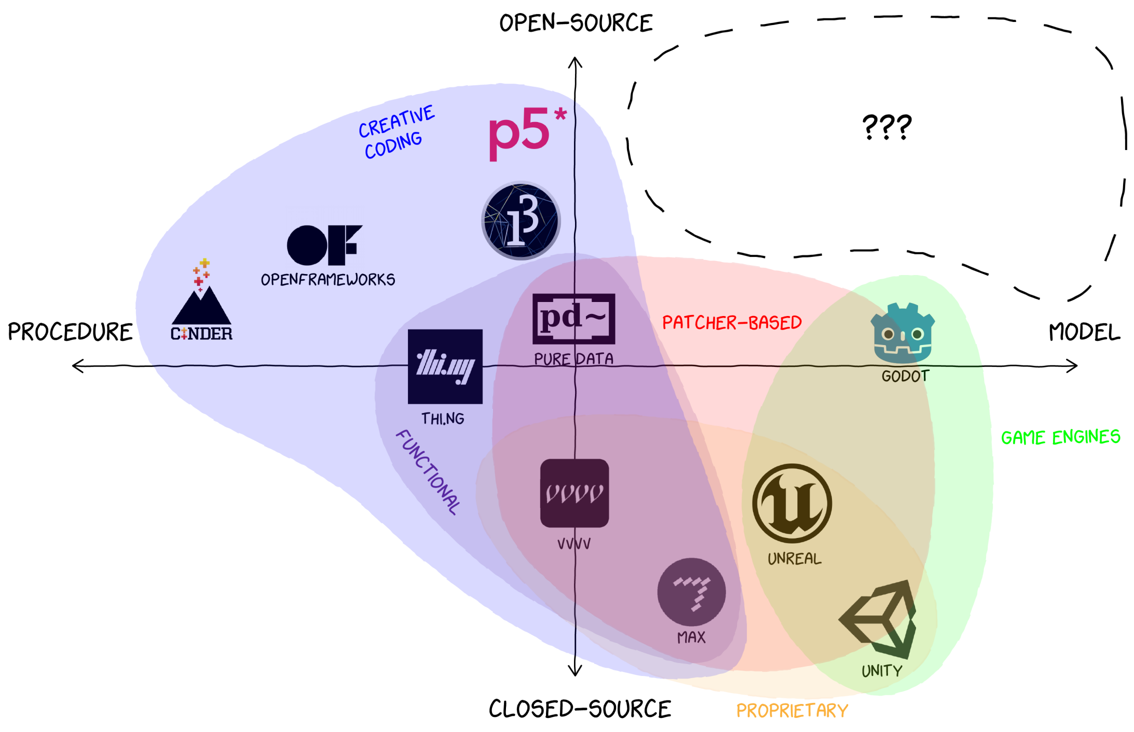


Fig. 11 A qualitative map of existing creative coding frameworks and game engines based on their qualities of being open/closed-source and procedure/model-first

In short, my goal for now is to use what I have learned from teaching and writing to make a tool informed by those pedagogies. And as it looks now, that tool or framework which will be derived from this project will probably inhabit that space. This is because I do not find it satisfactory for a tool to be model-first without being open-source. Although model first tools should start from direct manipulation and chunking of information, they must also make the user curious, and encourage them to explore into the the inner workings of the tool that they are using. This simply isn't possible if the source code is unavailable.

Because of this, I want to use what I have and will learn from this project to make a new tool to contribute to the space of model-first and open-source frameworks for creative coding.

To fund the initial production of this endeavor, I will be seeking out a fellowship or arts residency at organizations like Eyebeam, the Mozilla Foundation, or the Processing Foundation. These organizations are all attractive possibilities for their values of being open-source, inclusion-minded, and often creativity-focused.

Conclusion

From my initial research through to my initial testing of tutorials, it is apparent to me that the Model-First approach will gradually become the new standard, if not the progenitor of a new emerging standard of programming education for learners with goals outside of an engineering or computer science program. While an engineering mindset has its utility, it also has its time and place.

In Creative Coding, problems often do not share an impetus or a size of scope in tandem with the more commercially common forms of development. Because of this, approaching a creative coding problem with an engineering mindset can even be detrimental to the project, letting the developer over-architect and over-scale their work before they even start.

Additionally, Creative Coding is not just about making products, although its certainly allowed to be. Creative Coding as a form of expression is also about opening up the world

The world outside of proper computer science is both vast and varied, and the programming resources available should reflect that. It is my desire to continue to contribute to this frontier, further enriching the space of creative coding.

*Of course, in the time it will take for me to reach my intended inflection point, Unity or the Unreal Engine could surprise me and become open-sourced. Godot could also take off in popularity and become wildly successful without moving to a proprietary model. There could also be an entirely new agent which is introduced to this field which is everything I would have been aiming to make.

Works Cited

"2: How Experts Differ from Novices." *How People Learn: Brain, Mind, Experience, and School*, by John D. Bransford, National Acad. Press, 2004.

Barth, Zach. "Zachtronics." *Zachtronics*, www.zachtronics.com/.

Bennedsen, Jens, and Michael E. Caspersen. "Programming in Context – A Model-First Approach to CS1 ." *Association for Computing Machinery*, 3 Mar. 2004, www.cs.au.dk/~mec/publications/conference/08--sigcse2004.pdf.

"About." *Codecademy*, www.codecademy.com/about.

Gonzalez Vivo, Patricio. "The Book of Shaders." *The Book of Shaders*, thebookofshaders.com/.

Hickey, Rich. "Simple Made Easy." *InfoQ*, www.infoq.com/presentations/Simple-Made-Easy.

Flick, Jasper. "Unity C# and Shader Tutorials." *Catlike Coding*, catlikecoding.com/unity/tutorials/.

Klee, Paul, and Sibyl Moholy-Nagy. *Pedagogical Sketchbook*. Faber and Faber, 1981.

Montfort, Nick. *Exploratory Programming for the Arts and Humanities*. The MIT Press, 2016.

Jackson, Daniel, and Rob Miller. "A New Approach to Teaching Programming." *Psychology of Programming Interest Group*, vol. 18, 2006, pp. 255–265., pdfs.semanticscholar.org/1c75/fd0b1815ec77995aa3ed9da6b180bf1eb78e.pdf.

Papert, Seymour. *Mindstorms: Children, Computers, and Powerful Ideas*. Basicbooks, 1980.

Reas, Casey. "Thoughts on Software for the Visual Arts – Processing Foundation – Medium." *Medium*, Processing Foundation, 1 Feb. 2017, medium.com/processing-foundation/thoughts-on-software-a8a82c95e1ad.

Sajaniemi, Jorma, and Chenglie Hu. "Teaching Programming: Going beyond 'Objects First.'" *Psychology of Programming Interest Group*, vol. 18, Sept. 2006, pdfs.semanticscholar.org/1c75/fd0b1815ec77995aa3ed9da6b180bf1eb78e.pdf.

Shehane, Ronald, and Steven Sherman. "Visual Teaching Model for Introducing Programming Languages." *Journal of Instructional Pedagogy*, vol. 14, Mar. 2014, files.eric.ed.gov/fulltext/EJ1060073.pdf.

Shiffman, Daniel. “The Coding Train.” *YouTube*, YouTube, www.youtube.com/user/shiffman.

Vihart. “Doodling in Math Class: DRAGONS.” *YouTube*, YouTube, 19 Aug. 2013, www.youtube.com/watch?v=EdyociU35u8.

Appendix

Website Pages

Intro

(0-0) WHO IS THIS BOOK FOR?

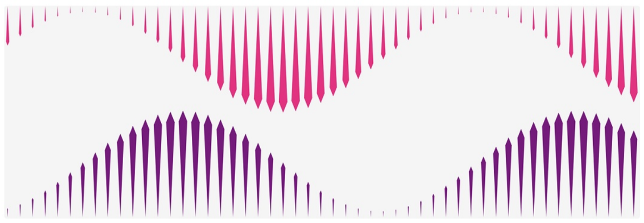
(0-1) HOW THIS BOOK IS ORGANIZED

(0-2) IN CASE YOU LIKE FOREWARDS AND ARTIST STATEMENTS

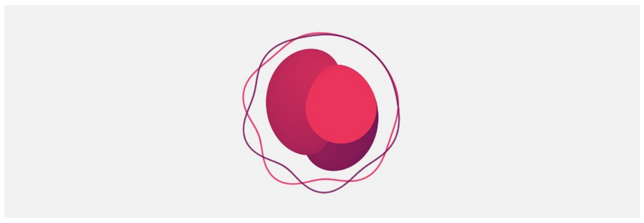
(0-3) GETTING STARTED WITH P5JS

Space

Time



Form



Who is this series for?

This body of work is intended for people who have started programming, but aren't quite sure how to get from beginner to intermediate. The goal of this project is to be the "best second resource" for learning creative coding after Daniel Shiffman's The Coding Train and before Patricio Gonzalez Vivo's The Book of Shaders.

By now, you've already gotten the hang of procedural programming: stuff like `if` , `else` , `while` , `for` , and basic data types. You've also probably been exposed to functions and objects at least once, but maybe you haven't quite gotten the hang of that yet.

Maybe you also still don't feel comfortable making a whole new program from scratch, and instead rely on finding examples to jump off of. This series of tutorials and lessons is made to help you think about the structure of and logic of a program, so it doesn't feel so intimidating when you dive into the code. The tutorials are broken up by their components so that you can ostensibly complete them in any order before tying them all together.

If you've gotten into creative code, you're also pretty familiar with p5.js. If you've used p5, Processing, or openFrameworks, you'll feel right at home here. If you think you need a refresher, check out Hello p5 to get started. There's also a lot of great tutorials on getting started over at The Coding Train

As you go through the tutorials, take your time to read through the explainations. Programming concepts can be unintuitive at times and they don't come to you all at once, so don't expect to just skim through and get it right away.

Copyright © Andrew Cotter 2017 - 2018

Setting up P5.js

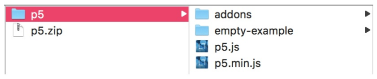
If you're new to the workflow of p5.js, a lot of the drawing fuctions should look familiar if you're coming from Processing or openFrameworks (e.g. rect() and ellipse()). What might not look so familiar is the workflow. This is just a quick guide to help get you familiar with what a p5 project looks like and how it might be structured.

WHAT IS P5?

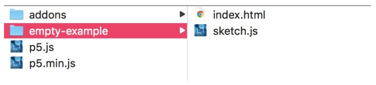
In essence, p5 is a javascript library that lets you make a canvas on a web page and draw on it. p5 also has a lot of libraries which lets you access web apis, interact with the DOM on your webpage, and work with text inputs.

DOWNLOADING P5.JS

Fortunately, there's very little that we need to download or install to set p5 set up. All we need to do is go to p5js.org/download and click the link that says "Complete Library". This gives you a zip file which you can extract into a folder that looks like this:



The first folder has the optional addons for p5, the javascript files at the bottom have the core code for the p5 library, and the "empty example folder"



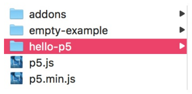
Inside that folder we'll find the two files that you'll usually be working with on smaller p5 projects; `index.html` and `sketch.js` .

But before we start working with these files, lets duplicate them and name the copy.

MAKING A NEW SKETCH

If you're working on a bigger, more professional project, you probably wouldn't use the method I'm prescribing now. In fact, you might not be using p5 at all. But since p5 uses the heuristic of a 'sketchbook,' we'll lean into that, and treat each project folder like a page in our own personal sketchbook.

To do so, lets duplicate the `empty-example` folder and rename it something appropriate to the project we're working on. Since we're familiarizing ourselves with p5, lets call it `hello-p5` .



Now lets open up the folder inside your text editor of choice. (If you don't have a favorite text editor, might I suggest Visual Studio Code?) Before we start in on the `sketch.js` file, lets look at whats going on inside the `index.html` file. It should look something like this.

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" width=device-width, initial-scale=1.0, maximum-scale=1.0, user-scalable=0>
    <style> body {padding: 0; margin: 0;} </style>
    <script src="../p5.min.js"></script>
    <script src="../addons/p5.dom.min.js"></script>
    <script src="../addons/p5.sound.min.js"></script>
    <script src="sketch.js"></script>
  </head>
  <body>
  </body>
</html>
```

So you might have noticed that there isn't a lot going on here, at least not in the `<body>` of the page. Everything here is about tying the javascript files that we have together and running them using the `<script>` tag.

Now let's look at the `sketch.js` file.

```
function setup() {
// put setup code here
}

function draw() {
// put drawing code here
}
```

As the comments suggest, the `setup()` loop runs only once, and the `draw()` loop runs continuously. This is where we'll be adding code for many of the examples in this book.

Typically, the first thing you need to do in a p5 sketch is add a canvas to the page. We do this by calling the function `createCanvas()` and passing in the `x` and `y` dimensions of the canvas. So if we wanted to make a canvas that is 400 by 300 pixels, we'd write:

```
function setup()
{
createCanvas(400, 300);
}
```

Alternatively, if we wanted to make the canvas the size of the browser window, we can write:

```
function setup()
{
createCanvas(windowWidth, windowHeight);
}
```

If you need to brush up on the p5 API, there's a lot neat examples and refences on the p5 site.

2D Camera

So lets say you're helping a friend make a 2D platformer game. They're a little new at this, so they're getting a bit stuck. They have a few things already made, like the player, the ground, the inputs, and they even imported Matter.js to take care of the physics.

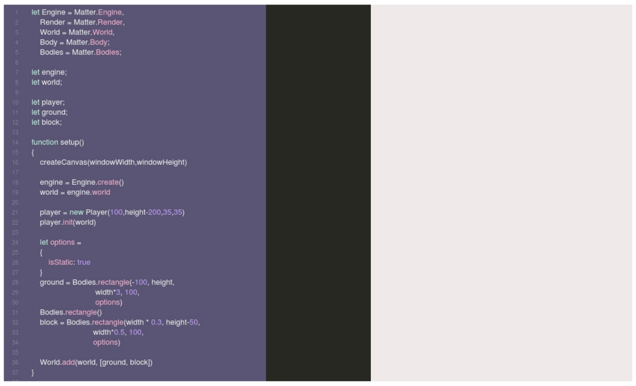
The first thing we should do is take a look at their game and see how it's working. Try interacting with the game a bit, and read through the code that they've written. If you need to restart the game, try editing the code to refresh the canvas. If you feel like you've made a mistake in the code, you can also use command+Z or ctrl+Z to undo your changes.

WAIT, WHAT'S MATTER.JS AGAIN?

If Matter.js is altogether alien to you, it might be worth opening up the documentation or looking through the examples. There's also a good tutorial series on The Coding Train on how to use Matter.js with p5. Your friend doesn't want to spend all day thinking about how to implement physics, so they just settled for using a physics library instead.

Arrow keys: Left / Right

Spacebar: Jump



```
let Engine = Matter.Engine,
    Render = Matter.Render,
    World = Matter.World,
    Body = Matter.Body;
    Bodies = Matter.Bodies;

let engine;
let world;

let player;
let ground;
let block;

function setup()
{
    createCanvas(windowWidth,windowHeight)

    engine = Engine.create()
    world = engine.world

    player = new Player(100,height-200,35,35)
    player.init(world)

    let options =
    {
        isStatic: true
    }
    ground = Bodies.rectangle(-100, height,
                              width*3, 100,
                              options)
    Bodies.rectangle()
    block = Bodies.rectangle(width * 0.3, height-50,
                             width*0.5, 100,
                             options)

    World.add(world, [ground, block])
}
```

The aesthetics of the game could use some polish, but your firend isn't really looking for feedback on that, and they can always improve that later anyway. You might have noticed a bigger problem with this game though: **it's pretty small**. Only the size of the game window, in fact.

"How do I make the level bigger?" your friend might ask. Well, we could simply add more elements outside of the canvas so the player box doesn't fall off into infinity, but that's not really what they're asking here, is it? Their real question is:

HOW MIGHT I SEE MORE OF THIS WORLD AS THE PLAYER MOVES AROUND IT?

Lets take a moment to reflect on this. In p5, everything is drawn in relation to the origin point (0,0). By default, this is in the top left of the canvas.

origin illustration here

This means, in order to see other parts of the game, we'll have to **move the origin**. What's more, we need to move the origin **in relation to the player** so that it stays on the screen. So in order to emulate a camera that is following the player along, we'll need to do both of these.

IN SUMMARY:

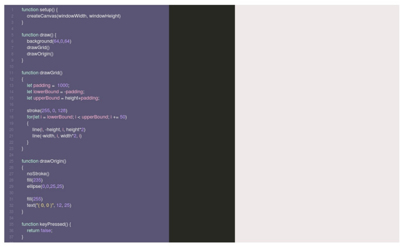
- In order to see more of the game, we need to move the origin
- In order to keep the player on the screen, we need to make the origin movements relational to the player object

Moving the Origin

Fortunately, if we want to change where the origin is on the canvas, we have a few functions built in to p5 that let us do just that. For making a 2D Camera that pans left and right, we can make good use of the `translate()` function.

TRANSLATE()

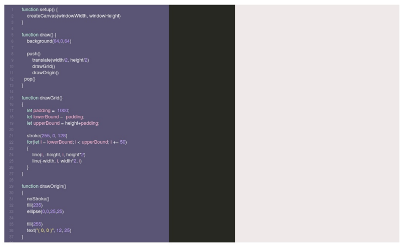
As you already know, **the orgin position `(0,0)` is at the top left of the p5 canvas**. Here's a sketch with a red circle drawn at the origin to illustrate.



```
function setup() {
   createCanvas(windowWidth, windowHeight)
}

function draw() {
   background(64,0,64)
   drawGrid()
   drawOrigin()
}

function drawGrid()
{
   let padding =  1000;
   let lowerBound = -padding;
   let upperBound = height+padding;

   stroke(255, 0, 128)
   for(let i = lowerBound; i < upperBound; i += 50)
   {
      line(i, -height, i, height*2)
      line(-width, i, width*2, i)
   }
}

function drawOrigin()
{
   noStroke()
   fill(235)
   ellipse(0,0,25,25)

   fill(255)
   text("( 0, 0 )", 12, 25)
}

function keyPressed() {
   return false;
}
```

But the cool thing about `translate()` is that we can change where this dot is being draw on the canvas **without changing the coordinates that it's being drawn at**.

To do this, call `translate()` with two arguments. The first is how the origin moves along the x-axis, and the second argument controlls how the origin moves along the y-axis. By using the arguments `width/2, height/2` the circle can move to the center.



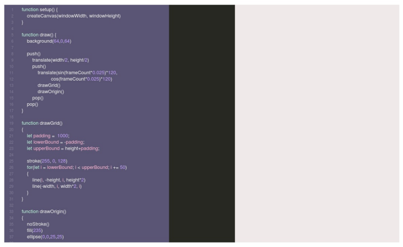
```
function setup() {
   createCanvas(windowWidth, windowHeight)
}

function draw() {
   background(64,0,64)

   push()
      translate(width/2, height/2)
      drawGrid()
      drawOrigin()
  pop()
}

function drawGrid()
{
   let padding =  1000;
   let lowerBound = -padding;
   let upperBound = height+padding;

   stroke(255, 0, 128)
   for(let i = lowerBound; i < upperBound; i += 50)
   {
      line(i, -height, i, height*2)
      line(-width, i, width*2, i)
   }
}

function drawOrigin()
{
   noStroke()
   fill(235)
   ellipse(0,0,25,25)

   fill(255)
   text("( 0, 0 )", 12, 25)
}
```

BUT WHAT ABOUT PUSH() AND POP()?

These functions act as 'bookmarks' of a sort for the `translate()` function. When `push()` executes, p5 is remembering where the origin was when the `push()` function was called. `pop()` then reverts the origin back to where `push()` recorded. It's also common practice (but not required) to indent between each `push()` and `pop()`. In the example above, using them isn't necessary, but it comes in handy if you want to do a whole bunch of translations in a row.

ANIMATE THE TRANSLATION

The amount of translation that the origin does can also be amiated over time, like so:



```
function setup() {
   createCanvas(windowWidth, windowHeight)
}

function draw() {
   background(64,0,64)

   push()
      translate(width/2, height/2)
      push()
         translate(sin(frameCount*0.025)*120,
                  cos(frameCount*0.025)*120)
         drawGrid()
         drawOrigin()
      pop()
   pop()
}

function drawGrid()
{
   let padding =  1000;
   let lowerBound = -padding;
   let upperBound = height+padding;

   stroke(255, 0, 128)
   for(let i = lowerBound; i < upperBound; i += 50)
   {
      line(i, -height, i, height*2)
      line(-width, i, width*2, i)
   }
}

function drawOrigin()
{
   noStroke()
   fill(235)
   ellipse(0,0,25,25)
```

By passing the `frameCount` (the number of frames that the program has completed) into the `sin()` and `cos()` functions, we can make the circle revolve around the center of the screen while still drawinging it at `(0,0)`.

WHY ARE YOU MULTIPLYING THE FRAMECOUNT BY A SMALL NUMBER, AND THE RESULT OF SINE/COSINE BY A LARGER NUMBER?

This largely has to do with the scale of `sin()` and `cos()`'s inputs. Because their inputs are interpreted as radians (from 0 to about 6.28) counting in full inrements would result in a very fast animation of the translation. (Maybe, you want that, but for most that can be a little over-stimulating.) Sine and Cosine always return values that are between -1 and 1, so in order to makethe area in which the circle moves larger, the resulting value is multiplied by a larger number.

NEXT STEPS

Now that you're a little more familiar with doing matrix transformations by using `translate()`, you're ready to start making a camera for your friend.

Making the Camera

Okay, before you start making a camera class, let's go over what it needs to do first.

A 2D CAMERA NEEDS TO:

- move the origin to see the game world
- keep track of the player object's position

The first part we already have a pretty good idea of how to do. We can use `translate()` to move the origin around, but making an object that *just* translates could be a little tricky. Remember the `push()` and `pop()` functions? The canvas could get messy really fast if you don't revert the matrix when the camera is done, so you'll need those too. Because it would be hard to import the draw loop into a camera object, **we'll need separate functions for when the camera transformation begins and ends.**

With that in mind, the first task breaks down even further

- move the origin to see the game world
- save the origin position using `push()`
- `translate()` the origin based on a given value
- revert the origin back to its default position using `pop()`

This means that you'll need to make a **function where the transformation value updates**.

Based on these requirements, we can begin to ouline a class that looks like this:

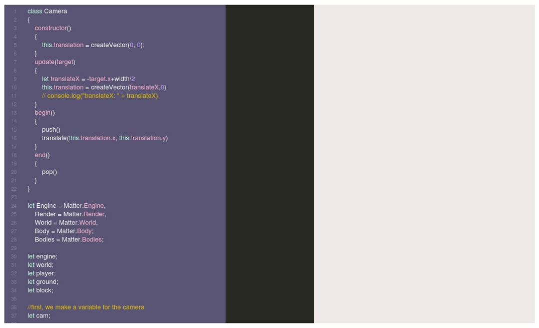
```
class Camera
{
   constructor(){}
   update(){}
   begin(){}
   end(){}
}
```

The `constructor()` and the `update()` functions can be made later when you start making the camera follow the player. For now, you just need to **add `push()` and `translate()` to the `begin()` function**, and put a translation value in the constructor. At this point, the value that you pass it is arbitrary, since you'll relate it to the player object's position later.

```
class Camera
{
   constructor()
   {
      this.translation = createVector(0, 0);
   }
   update(){}
   begin()
   {
      push()
      translate(this.translation.x, this.translation.y)
   }
   end()
   {
      pop()
   }
}
```

Now that you have your camera class started, let's make an instance of it in our game. First, we make a variable for our camera with `let cam;` , then we call it's constructor in the setup loop with `cam = new Camera()` .

Finally, you can add the `cam.begin()` and `cam.end()` functions before and after drawing the other objects in our scene.



```
class Camera
{
   constructor()
   {
      this.translation = createVector(0, 0);
   }
   update(target)
   {
      let translateX = -target.x+width/2
      this.translation = createVector(translateX,0)
      // console.log("translateX: " + translateX)
   }
   begin()
   {
      push()
      translate(this.translation.x, this.translation.y)
   }
   end()
   {
      pop()
   }
}

let Engine = Matter.Engine,
   Render = Matter.Render,
   World = Matter.World,
   Body = Matter.Body;
   Bodies = Matter.Bodies;

let engine;
let world;
let player;
let ground;
let block;

//first, we make a variable for the camera
let cam;
```

Try playing around with the **x value of the camera's `this.translation` property** and see how the canvas changes. If you're feeling fancy, why not try `sin(frameCount)` to animate the translation?

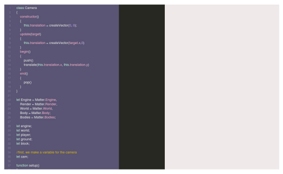
When you're ready to move on, we can add the player-following behavior to the camera.

Following the Player

Now that the camera is set up and can translate the world, now it's time to start translating it in relation to the player. In order to do this, the camera first needs to **know where the player is**. This can be done in the `update()` function that you outlined earlier.

By passing in a `target` value into `update()`, the update function can manipulate `this.translation`. Since we only need the `x` coordinate of the player object, you can leave the `y` value of the translation at `0`.

```
class Camera
{
  constructor()
  {
    this.translation = createVector(0, 0)
  }

  update(target)
  {
    this.translation = createVector(target.x,0)
  }
  ...
}
```

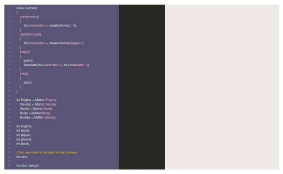


```
class Camera
{
  constructor()
  {
    this.translation = createVector(0, 0);
  }
  update(target)
  {
    this.translation = createVector(target.x,0)
  }
  begin()
  {
    push()
    translate(this.translation.x, this.translation.y)
  }
  end()
  {
    pop()
  }
}

let Engine = Matter.Engine,
  Render = Matter.Render,
  World = Matter.World,
  Body = Matter.Body;
  Bodies = Matter.Bodies;

let engine;
let world;
let player;
let ground;
let block;

//first, we make a variable for the camera
let cam;

function setup()
```

PASSING IN THE PLAYER POSITION

Now lets see how that looks when we bring the update function into the draw loop. In the draw loop, you'll need to call `cam.update()` and pass in the position of the player object's position (`player.body.position`)

```
...
function draw()
{
  background(51);
  Engine.update(engine)
  player.update()

  cam.update(player.body.position)
  cam.begin()
  ...
  cam.end()
}
...
```



```
class Camera
{
  constructor()
  {
    this.translation = createVector(0, 0);
  }
  update(target)
  {
    this.translation = createVector(target.x,0)
  }
  begin()
  {
    push()
    translate(this.translation.x, this.translation.y)
  }
  end()
  {
    pop()
  }
}

let Engine = Matter.Engine,
  Render = Matter.Render,
  World = Matter.World,
  Body = Matter.Body;
  Bodies = Matter.Bodies;

let engine;
let world;
let player;
let ground;
let block;

//first, we make a variable for the camera
let cam;

function setup()
```

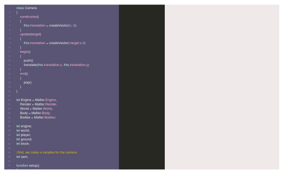
WHY `.BODY` ?

Because the player object is a matter.js object, it has a `.body` property that contains all of the physics information that the rest of matter.js needs to run the

WAIT, THE TRANSLATION IS GOING THE WRONG WAY!

Since the translation is moving the origin, we need to move it in the **opposite direction of the thing we're tracking.** Lets change the camera's `update()` function to reflect this.

```
class Camera
{
  constructor()
  {
    this.translation = createVector(0, 0)
  }

  update(target)
  {
    this.translation = createVector(-target.x,0)
  }
...
}
```

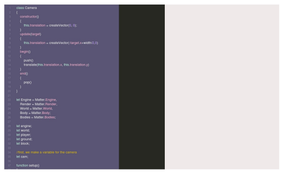


```
class Camera
{
  constructor()
  {
    this.translation = createVector(0, 0);
  }
  update(target)
  {
    this.translation = createVector(-target.x,0)
  }
  begin()
  {
    push()
    translate(this.translation.x, this.translation.y)
  }
  end()
  {
    pop()
  }
}

let Engine = Matter.Engine,
  Render = Matter.Render,
  World = Matter.World,
  Body = Matter.Body;
  Bodies = Matter.Bodies;

let engine;
let world;
let player;
let ground;
let block;

//first, we make a variable for the camera
let cam;

function setup()
```

HMM, STILL NOT QUITE RIGHT

Looks like the player ends up right at the edge of the canvas now. To fix this, let's add an offset to the translation so the player is in the center. Adding half of the width to the translation should be enough.

```
class Camera
{
...
  update(target)
  {
    this.translation = createVector(-target.x+width/2,0)
  }
...
}
```



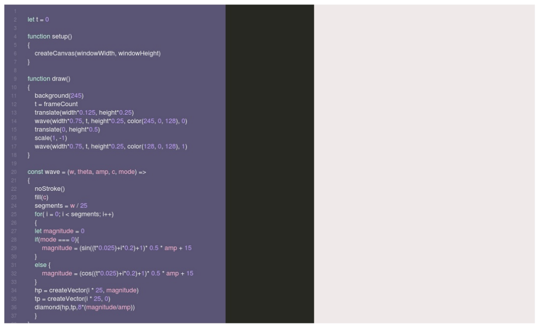
```
class Camera
{
  constructor()
  {
    this.translation = createVector(0, 0);
  }
  update(target)
  {
    this.translation = createVector(-target.x+width/2,0)
  }
  begin()
  {
    push()
    translate(this.translation.x, this.translation.y)
  }
  end()
  {
    pop()
  }
}

let Engine = Matter.Engine,
  Render = Matter.Render,
  World = Matter.World,
  Body = Matter.Body;
  Bodies = Matter.Bodies;

let engine;
let world;
let player;
let ground;
let block;

//first, we make a variable for the camera
let cam;

function setup()
```

Great! Now the camera works! The rest of the level could use a bit more designing and fleshing out, but we can leave that to your friend's creative discretion

FrameRate

MAKING AN ANIMATION FOR A CLIENT

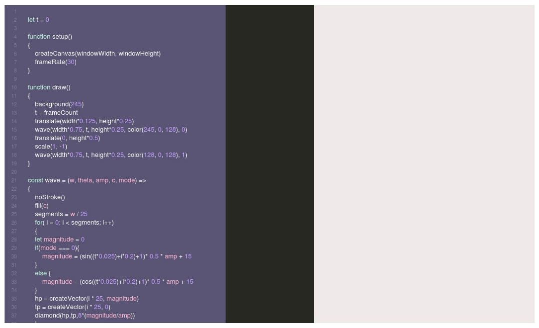
Let's say that you're working on an animation for a client. They wanted an animated canvas that loops. So far, you have this:



```

let t = 0

function setup()
{
   createCanvas(windowWidth, windowHeight)
}

function draw()
{
   background(245)
   t = frameCount
   translate(width*0.125, height*0.25)
   wave(width*0.75, t, height*0.25, color(245, 0, 128), 0)
   translate(0, height*0.5)
   scale(1, -1)
   wave(width*0.75, t, height*0.25, color(128, 0, 128), 1)
}

const wave = (w, theta, amp, c, mode) =>
{
   noStroke()
   fill(c)
   segments = w / 25
   for( i = 0; i < segments; i++)
   {
   let magnitude = 0
   if(mode === 0){
      magnitude = (sin((t*0.025)+i*0.2)+1)* 0.5 * amp + 15
   }
   else {
      magnitude = (cos((t*0.025)+i*0.2)+1)* 0.5 * amp + 15
   }
   hp = createVector(i * 25, magnitude)
   tp = createVector(i * 25, 0)
   diamond(hp,tp,8*(magnitude/amp))
   }
```

The trouble is, the client just informed you that this animation might be running on older hardware, so it has to run at 30 frames per second. Fortunately, you can just use the `frameRate()` function in the setup loop to fix that

```
framerate(30)
```



```

let t = 0

function setup()
{
   createCanvas(windowWidth, windowHeight)
   frameRate(30)
}

function draw()
{
   background(245)
   t = frameCount
   translate(width*0.125, height*0.25)
   wave(width*0.75, t, height*0.25, color(245, 0, 128), 0)
   translate(0, height*0.5)
   scale(1, -1)
   wave(width*0.75, t, height*0.25, color(128, 0, 128), 1)
}

const wave = (w, theta, amp, c, mode) =>
{
   noStroke()
   fill(c)
   segments = w / 25
   for( i = 0; i < segments; i++)
   {
   let magnitude = 0
   if(mode === 0){
      magnitude = (sin((t*0.025)+i*0.2)+1)* 0.5 * amp + 15
   }
   else {
      magnitude = (cos((t*0.025)+i*0.2)+1)* 0.5 * amp + 15
   }
   hp = createVector(i * 25, magnitude)
   tp = createVector(i * 25, 0)
   diamond(hp,tp,8*(magnitude/amp))
```

OH NO, IT'S TOO SLOW NOW!

You changed the framerate from 60 to 30, but now the animation plays **half as fast**. This is because the animation that's being done is **frame-dependent**. That means that the speed of change in the animation is directly proportional to the framerate of the canvas. That means we'll need to use the rate in change of time rahter than the rate in change of frames. This will make the animation run the same way, regardless of the frames per second (FPS) that the sketch runs at.

THE FASCINATING PROBLEM OF PROGRAMMATIC ANIMATION

This is one of the more interesting challenges of doing animations with programming. In conventional software engineering context, time is a much less interesting issue. If you're writing a server or the backend of an application, the main challenge you need to solve for is getting the program to run as quickly as possible. That isn't to say making a program run as quickly as possible can't be a rewarding challenge, meerly that its a much more linear and monotonic challenge than what we're trying to do.

SUMMARY

In short, in order to make the animation frame-independent, we need to:

- track what time it is each frame
- compare the change in time between frames
- use the change in time to influence the theta

Timeline Class

SETTING UP THE CLASS

Okay, lets set up a class that do all of the tasks we just outlined.

This tutorial is going to name the class 'Timeline' since more animation functionality will be added to it in future tutorials.

```
class Timeline{}
```

Since the object only needs to compare two points in time (the current frame and the previous frame), it'll need to store those two values.

```
class Timeline
{
    constructor()
    {
        this.now
        this.then
    }
}
```

GETTING THE CURRENT TIME

Conveniently, javascript has a native method for getting the time: `Date.now()` . This returns the numer of milliseconds since January 1st, 1970. Why 1970? It's mostly boring reasons having to do with UNIX, which you can read more about here, if you're curious.

ASIDE: `PERFORMANCE.NOW()`

There's also `performance.now()` if you want to get a more precise timestamp value, but you probably don't need it for this tutorial.

UPDATING `THEN` AND `NOW`

Cool, so now that you know what method will get you the time, it time to put it to use!

Start by adding it to your constructor, then you can make an update function to reassign the values.

```
class Timeline
{
    constructor()
    {
        this.now = Date.now()
        this.then
    }
    update()
    {
        this.then = this.now
        this.now = Date.now()
    }
}
```

DELTA TIME

Great! Now that there's a `then` and `now` being recorded, the difference between them (or the delta) can be found. All you have to do is make a function that subtracts `then` from `now`

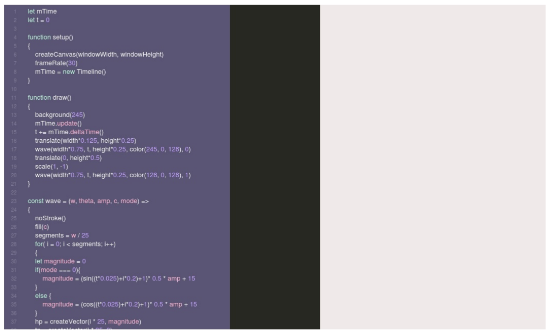
```
class Timeline
{
    deltaTime()
    {
        return this.now - this.then
    }
}
```

Now you're ready to start integrating this with the rest of the animation!

Integrating the Timeline

INTEGRATING `.DELTATIME()`

Now let's add a Timeline object to the animation and replace the `frameCount()` logic with `deltaTime()`.



```
let mTime
let t = 0

function setup()
{
   createCanvas(windowWidth, windowHeight)
   frameRate(30)
   mTime = new Timeline()
}

function draw()
{
   background(245)
   mTime.update()
   t += mTime.deltaTime()
   translate(width*0.125, height*0.25)
   wave(width*0.75, t, height*0.25, color(245, 0, 128), 0)
   translate(0, height*0.5)
   scale(1, -1)
   wave(width*0.75, t, height*0.25, color(128, 0, 128), 1)
}

const wave = (w, theta, amp, c, mode) =>
{
   noStroke()
   fill(c)
   segments = w / 25
   for( i = 0; i < segments; i++)
   {
   let magnitude = 0
   if(mode === 0){
      magnitude = (sin((t*0.025)+i*0.2)+1)* 0.5 * amp + 15
   }
   else {
      magnitude = (cos((t*0.025)+i*0.2)+1)* 0.5 * amp + 15
   }
   hp = createVector(i * 25, magnitude)
```

BUT NOW THE ANIMATION IS WAY TOO FAST!

Oh no! Now the animation is going much faster than it was before. This is because when `frameCount*0.01` was being used, `theta` was increasing by about 0.02 each frame. now that `deltaTime` is being used, it's increasing by about 32 (at 30 FPS) each frame. That's because 32 milliseconds have elapsed since the last frame.

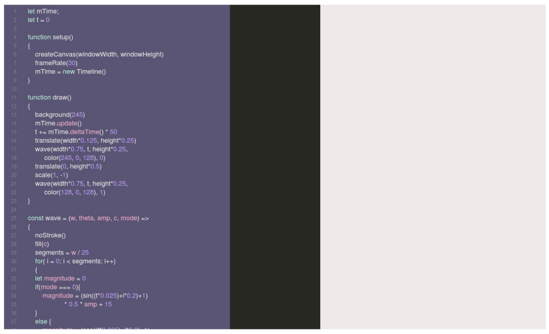
RETURNING SECONDS INSTEAD OF MILLISECONDS

The solution to this is to convert the deltaTime to miliseconds. This can be done by simply dividing the deltaTime value returned by 1000. You could divide each call to `deltaTime()`, but it'll probably save you more time in the long run to add that division to the Timeline function.

```
deltaTime()
{
    return (this.now - this.then) * 0.001
}
```

From there, you can scale up the speed of the animation by multiplying `deltaTime()` before adding it to `t`. The decision on how much to multiply the deltaTime by is a matter of aethetics and personal preference (and the preference of your hypothetical client).

```
t += mTime.deltaTime() * 5
```



```
let mTime;
let t = 0

function setup()
{
   createCanvas(windowWidth, windowHeight)
   frameRate(30)
   mTime = new Timeline()
}

function draw()
{
   background(245)
   mTime.update()
   t += mTime.deltaTime() * 50
   translate(width*0.125, height*0.25)
   wave(width*0.75, t, height*0.25,
      color(245, 0, 128), 0)
   translate(0, height*0.5)
   scale(1, -1)
   wave(width*0.75, t, height*0.25,
      color(128, 0, 128), 1)
}

const wave = (w, theta, amp, c, mode) =>
{
   noStroke()
   fill(c)
   segments = w / 25
   for( i = 0; i < segments; i++)
   {
   let magnitude = 0
   if(mode === 0){
      magnitude = (sin((t*0.025)+i*0.2)+1)
               * 0.5 * amp + 15
   }
   else {
```

Great! Now the animation will run at the same speed, regardless of its framerate. Try changing the `framerate()` in the setup loop, and see how the animation plays out when you change the argument passed in.

Slideshow

SHOWING OFF YOUR WORK

Let's say that you're going to an open-mic/open-projector event in your area to show off some of the smaller projects and sketches you've been working on. The trouble is, when you practice presenting, you have to keep switching between your slide deck and the sketches. If only there was some way to add the p5 sketches you've made to the slideshow.

Oh, wait! What if you did the opposite though, and added your sketches and slides into the same p5 program?

That might take a lot of content migration; copying and pasting the text from your slides into the javascript program. So for now, let's just put some of these placeholders

```
```

title placeholder
```
```

```
```

Project Description placeholder
```
```

```
```

Thank you slide
```
```

As for the sketches you'll be showing off, here's a few placeholders you might recognize from elsewhere in this book.

```
```

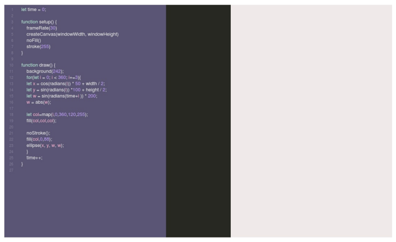
example 1
```
```

```
```

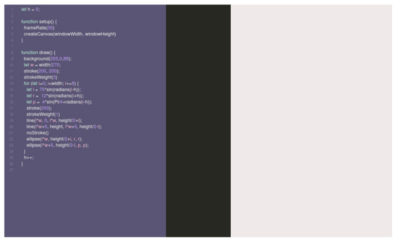
example 2
```
```

```
```

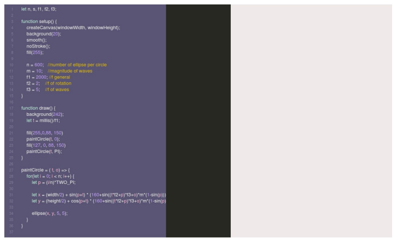
example 3
```
```



```
let time = 0;

function setup() {
   frameRate(30)
   createCanvas(windowWidth, windowHeight)
   noFill()
   stroke(255)
}

function draw() {
   background(242);
   for(let i = 0; i < 360; i+=3){
   let x = cos(radians(i)) * 50 + width / 2;
   let y = sin(radians(i)) *100 + height / 2;
   let w = sin(radians(time+i )) * 200;
   w = abs(w);

   let col=map(i,0,360,120,255);
   fill(col,col,col);

   noStroke();
   fill(col,0,88);
   ellipse(x, y, w, w);
   }
   time++;
}
```



```
let h = 0;

function setup() {
  frameRate(30)
  createCanvas(windowWidth, windowHeight)
}

function draw() {
  background(255,0,88);
  let w = width/270;
  stroke(200, 200);
  strokeWeight(3)
  for (let i=0; i<width; i+=8) {
    let l = 75*sin(radians(i-h));
    let r =  12*sin(radians(i+h));
    let p =  4*sin(PI/4+radians(i-h));
    stroke(200);
    strokeWeight(1)
    line(i*w, 0, i*w, height/2+l);
    line(i*w+5, height, i*w+5, height/2-l);
    noStroke()
    ellipse(i*w, height/2+l, r, r);
    ellipse(i*w+5, height/2-l, p, p);
  }
  h++;
}
```



```
let n, s, f1, f2, f3;

function setup() {
   createCanvas(windowWidth, windowHeight);
   background(20);
   smooth();
   noStroke();
   fill(255);

   n = 600;   //number of ellipse per circle
   m = 10;    //magnitude of waves
   f1 = 2000; //f general
   f2 = 2;    //f of rotation
   f3 = 5;    //f of waves
}

function draw() {
   background(242);
   let t = millis()/f1;

   fill(255,0,88, 150)
   paintCircle(t, 0);
   fill(127, 0, 88, 150)
   paintCircle(t, PI);
}

paintCircle = ( t, o) => {
   for(let i = 0; i < n; i++) {
      let p = (i/n)*TWO_PI;

      let x = (width/2) + sin(p+t) * (160+sin((t*f2+p)*f3+o)*m*(1-sin(p)))
      let y = (height/2) + cos(p+t) * (160+sin((t*f2+p)*f3+o)*m*(1-sin(p)

      ellipse(x, y, 5, 5);
   }
}
```

WHAT WE NEED

- Slides that can encapsulate the behavior of each or your sketches that you want to present
- One type of slide that is static (text/image)
- Another type of slide that is animated (p5 sketches)
- A container for all of your slides
- Something that keeps track of where you are in the slideshow
- Holds a variable for your place in the slideshow
- A way to change which slide is being shown

Slide Classes

WHAT WE NEED

- Slides that can encapsulate the behavior of each or your sketches that you want to present
- One type of slide that is static (text/image)
- Another type of slide that is animated (p5 sketches)
- A container for all of your slides
- A way to keep track of what slide you're on
- A way to go forward and backward in your slide deck

STATIC SLIDES

These slides can afford to be a little more generic since text and images can be passed in as arguments. The only trouble is, you might not want to pass in every argument every time you define a slide. Having the arity for that many parameters would also give you argument lists a mile long!

It might make sense to make all of your arguments optional by passing in one JSON object, then having the slide's constructor define each property conditionally, like so.

```
class StaticSlide
{
  constructor(options)
  {
    if (options.header) {
      this.header = options.header
    }
    if (options.subHeader) {
      this.subHeader = options.subHeader
    }
    if (options.bulletPoints) {
      this.bulletPoints = options.bulletPoints
    }
    if (options.imgLink) {
      this.imgLink = options.imgLink
    }
    if (options.background) {
      this.background = options.background
    } else {
      this.background = color(242)
    }
    if (options.fillColor) {
      this.fillColor = options.fillColor
    } else {
      this.fillColor = color(255, 0, 88)
    }
    if (options.imgLink) {
      this.imgLink = options.imgLink
    }
    if (options.font) {
      this.font = loadFont(options.font)
    } else {
      this.font = loadFont('assets/circular-book.otf')
    }
  }

  setup()
  {

  }

  draw()
  {
    background(this.background)
    // textFont(this.font)
    fill(this.fillColor)
    if(this.header) this.drawHeader();
    if(this.subHeader) this.drawSubHeader();
  }

  drawHeader()
  {
    textSize(48)
    text(this.header, width*0.05, height*0.22)
  }

  drawSubHeader()
  {
    textSize(36)
    textFont
    text(this.subHeader, width*0.05, height*0.33)
  }
}
```

Then in the main sketch, you can make an object and pass in that object as an argument

```
let titleOptions = {
  "header": "HELLO, WORLD!",
  "subHeader": "subtitle",


}
let titleSlide = new StaticSlide(titleOptions)

let endOptions = {
  "header": "Thanks! (   ✿)"
}
let endSlide = new StaticSlide(endOptions)
```

ADDING TO THE SLIDES

Now you have couple slides that have headers and subheads, but how could we add a bit more to it? You can certainly adjust the layout in the draw loop, but what about images? Or bullet points? Feel free to add to it as you see fit.

DYNAMIC SLIDES

These slides will be a little harder to organize since each one will have to be specifically tailored to the sketch which is being drawn. Because each of the draw functions that they perform would be radically different, we can't make a single class for all of them.

There could be an individual class for each sketch slide, but that would defeat the purpose of object-oriented programming.

Instead, the sketches can be defined as json objects, then have their setup and draw functions appended with dot notation which can be aceive this like so:

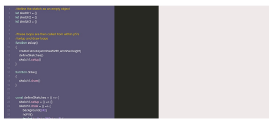
```
//define the sketch as an empty object
let sketch = {};

//add the sketch's setup loop
sketch.setup = () =>
{
  ...
}

//add the sketch's draw loop
sketch.draw = () =>
{
  ...
}

//these loops are then called from within p5's
//setup and draw loops
function setup()
{
  sketch.setup()
}

function draw()
{
  sketch.draw()
}
```

This doesn't save us from writing all of the code that we had before, but now you have the convenience of encapsulating the code for each sketch into two discrete methods.

Now you can do this for each sketch that you have.

```
//define the sketch as an empty object
let sketch1 = {}
let sketch2 = {}
let sketch3 = {}

sketch1.setup = () =>
{
  ...
}

sketch1.draw = () =>
{
  ...
}

sketch2.setup = () =>
{
  ...
}

sketch2.draw = () =>
{
  ...
}

sketch3.setup = () =>
{
  ...
}

sketch3.draw = () =>
{
  ...
}
//these loops are then called from within p5's
//setup and draw loops
function setup()
{
  sketch1.setup()
}

function draw()
{
  sketch1.draw()
}
```

Hmm, now that we look at that approach fleshed out though, there seems to be a lot of code before the `setup` loop. It might be good to swrap them in a function that can be put underneath the `draw` loop, or in a different file.



```
//define the sketch as an empty object
let sketch1 = {}
let sketch2 = {}
let sketch3 = {}


//these loops are then called from within p5's
//setup and draw loops
function setup()
{
  createCanvas(windowWidth,windowHeight)
  defineSketches()
  sketch1.setup()
}

function draw()
{
  sketch1.draw()
}


const defineSketches = () => {
  sketch1.setup = () => {}
  sketch1.draw = () => {
    background(242)
    noFill()
```

Modeling the State

Okay, now that there are slide classes set up, we need a way to keep track of them. After all, we can't call the draw method of every slide every frame. For one, we would only ever see the last slide, and all of those draw calls would certainly slow down your program.

WHAT WE NEED:

- An array containing all of our slides
- A variable that keeps track of which slide needs to be shown
- A way to to forward and back in the slideshow based on user inputs

With that in mind, lets outline the constructor and method of the Model class:

```
class Model
{
  constructor(){
    this.slides
    this.state
  }
  setup(){}
  getInput(){}
  next(){}
  back(){}
  draw(){}
}
```

CONSTRUCTOR

So Model has to keep track of the slides, and which one to show. This means that t**he slides have to be passed in as an array, and the state needs to start on the first slide**.

```
...
constructor(slides) //later, you'll pass an array into this argument
{
  this.slides = slides //the array wil then be assigned to this.slides
  this.state = 0 //the state will start at 0 to line up with the first index o
}
...
```

SETUP

Since all of the slides in `this.slides` need to run their setup loops. You could just use a regular for loop to go through all your slides:

```
setup()
{
  for (let i = 0; i < slides.length; i++)
  {
    slides[i].setup()
  }
}
```

But it might be a little more neat and compact if you use a `for of` loop:

```
setup()
{
  for (let slide of this.slides)
  {
    slide.setup()
  }
}
```

DRAW

Now that you have an array of slides, you can use the state of your model to determine which slide is drawn.

```
draw()
{
  this.slides[this.state].draw()
}
```

Because `this.state` is initially zero, the next behavior you'll need is

BACK AND NEXT

The methods that manipulate `this.state` are pretty straightforward: increment or decrement the state, but don't go past the first or last slides.

```
next()
{
  if(this.state < this.slides.length-1)
  this.state++
}

back()
{
  if(this.state > 0)
  this.state--
}
```

GETINPUT

Now that you have methods to change the state, now you need a function to call them based on key inputs.

```
getInput(input)
{
  switch(keyCode){
    case 37://left arrow
    this.back()
    break

    case 39://right arrow
    this.next()
    break
```

Putting it all together

Cool! Now you have a model to handle your slides. All you need to do now is to bring it into your sketch

```
let myModel; //make a variable to store your model in
let sketch1 = {};
let sketch2 = {};
let sketch3 = {};

function setup() {
createCanvas(windowWidth, windowHeight)
defineSketches() //

let titleOptions = {
   "header": "HELLO, WORLD!",
   "subHeader": "subtitle",
}
let titleSlide = new StaticSlide(titleOptions)

let endOptions = {
   "header": "Thanks! (   ✿)"
}
let endSlide = new StaticSlide(endOptions)
myModel = new Model([titleSlide,
                sketch1,
                sketch2,
                sketch3,
                endSlide])
for (slide of myModel.slides) {
   slide.setup()
}
}
```

DRAWING

```
function draw() {
myModel.draw()
}
```

INPUT

```
function keyPressed(){
myModel.getInput(keyCode)
}
```