CHAPTER 6

The Physical World

Physical Computing

I'm such a huge fan of hands-on tactile computing experiences. There's just so much satisfaction to be found in taking the ethereal code that lives inside your computer and expressing it in the physical realm. My first explorations into physical computing in the classroom came through the Arduino, an open microcontroller platform released in 2005 that revolutionized teaching with hardware by making it accessible to novices. The ever-expanding family of Arduino-based devices covers lots of different feature sets and form factors, and over the last decade the affordability, accessibility, and capability of competing physical computing platforms have continued to bring down the barriers for teaching CS with hardware.

A decade ago I would recommend that novice CS teachers hold off on physical computing until they have a bit of classroom experience, but today I can, without reservation, recommend that even the greenest of CS teachers give hardware a spin. Moreover, artists are increasingly leveraging these accessible physical computing platforms in conjunction with traditional arts in amazingly innovative and expressive ways. Take for example Miral Kotb, the creator of iLuminate, who combines theater, dance, and CS with programmed light-up costumes that blend the human and the programmed elements of performance (see figure 6.1).

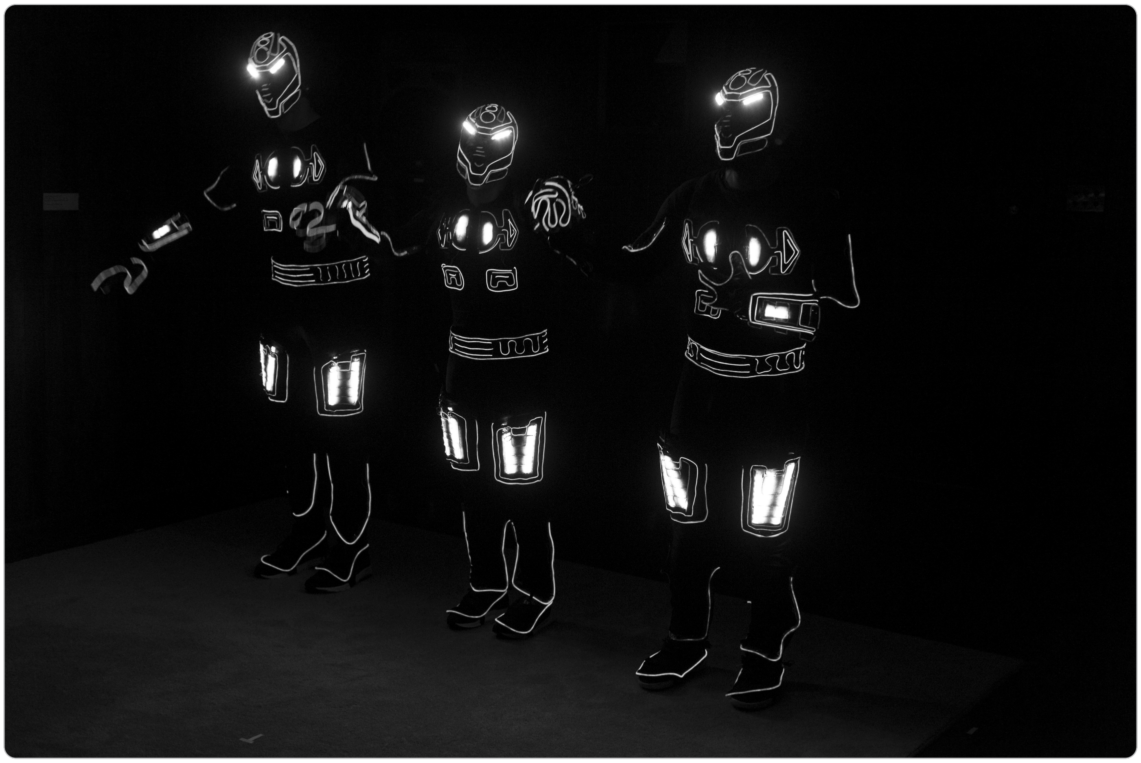


Figure 6.1 iLuminate costumes with programmed lights.

iLuminate was featured on the sixth season of America's Got Talent, and I highly recommend finding a video of their performance to share with students. Despite the impressive and polished performance, the basic elements of these suits are within reach for students to replicate and experiment with using affordable technology.

Tool Exploration: Circuit Playground

Of the many physical computing platforms available today, I have to admit I've got a special spot in my heart for Adafruit's Circuit Playground. I worked closely with the folks there when developing the physical computing unit for Code.org's CS Discoveries curriculum, which was designed around the unique capabilities of the Circuit Playground. I fell in love with this little board because it packed a ton of interesting input and output devices into a sewable, friendly form factor. Where my old Arduino lessons would require students to learn bread boarding and Ohm's law as a prerequisite for even the most basic projects, the Circuit Playground opens up a wide range of engaging projects that don't require a single extra wire.

Don't get me wrong, I absolutely adore bread boarding and electronics, but when it comes to introductory CS I'm always looking for ways to bring students early wins without requiring a lot of new skills all at once, and the Circuit Playground definitely delivers in that respect. Run a scientific experiment by logging data from the temperature sensor over time. Use the light sensor to make a project that activates when its box is opened. Combine the accelerometer and buzzer to make a truly unique musical synthesizer. Any one of these projects would require significant wiring with a traditional standalone microcontroller, but with the Circuit Playground not only can you create them without additional hardware, you can quickly switch between each of them making it much easier to share hardware among multiple students.

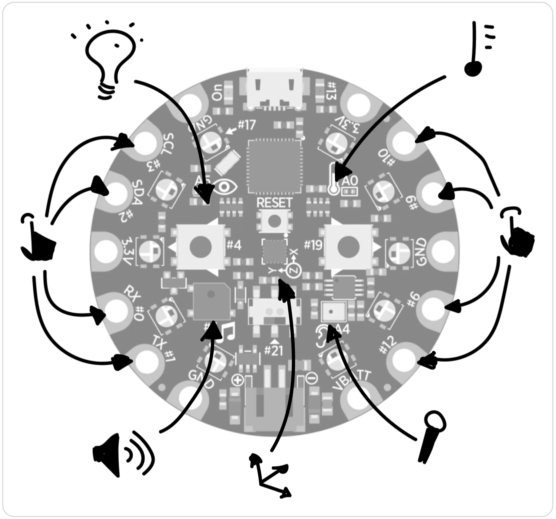


Figure 6.2 Circuit Playground inputs and outputs.

Circuit Playground Programming Model

Here's where things start to veer from the way we were thinking about purely software-focused tools. The way you think about writing a program for the Circuit Playground depends on which programming language or environment you pair it with. While most of the languages available give you pretty similar capabilities, the nuances between can make a big difference. To illustrate I'll overview four programming tools that cover a range of approaches and share pros and cons of using each.

Arduino (arduino.cc)

The Circuit Playground isn't actually an Arduino because it is neither produced by the Arduino organization, nor does it use traditional Arduino form factors and processors. Despite that, the board is compatible with the Arduino IDE and programming language, allowing you to leverage the abundance of resources and tutorials designed around the stalwart of consumer-focused open hardware. The Arduino programming model is very similar to Processing and p5, due to some shared lineage between the projects. Most notably you'll find a similar setup/loop structure in Arduino programs, though the looping function is named `loop()` instead of `draw()`. In the same way that Processing was built on Java, the Arduino is programmed in a variant of C. This allows for one to write efficient and performative code that squeezes the most of a puny processor, but it also means that you have to deal with the challenges of a language that wasn't designed for beginners. Arduino sketches are compiled and uploaded to the board over USB, and once uploaded you need only provide power to the board to have it run your program. The folks at Adafruit have also published a nicely abstracted Arduino library that provides easy access to all of the sensors and outputs on the Circuit Playground. To write a program that blinks an LED every second could be written as follows:

```
int ledPin = 13;

void setup() {
  pinMode(ledPin, OUTPUT);
}

void loop() {
  digitalWrite(ledPin, HIGH);
  delay(1000);
  digitalWrite(ledPin, LOW);
  delay(1000);
}
```

Pros:

- Well established language with tons of existing resources, tutorials, and third-party libraries
- Compatible with lots of lots of different microcontrollers

Cons:

- Not the most novice friendly language (C)
- Can be challenging to debug, particularly when you hit compiler errors

Circuit Python

Circuit Python (**circuitpython.org**) is a Python interpreter that has been optimized to run right on your microcontroller. This means that once you have the Circuit Python interpreter installed, adding a program is as simple as mounting the board as a USB drive and dropping a text file with your script onto it. You don't need any software other than a plain text editor, though a lightweight Python IDE like Mu editor (**codewith.mu**) will make your life much easier. The structure of Circuit Python programs is similar to Arduino sketches, except that instead of a named function for the loop you typically use an infinite *while True* loop. The same blinking LED program would look like this in Circuit Python:

```
import board
import digitalio
import time

led = digitalio.DigitalInOut(board.D13)
led.direction = digitalio.Direction.OUTPUT

while True:
    led.value = True
    time.sleep(1)
    led.value = False
    time.sleep(1)
```

An additional benefit of the Python interpreter running directly on your board is that Circuit Python has a REPL, which stands for Read Evaluate Print Loop. This REPL is like a command line for your board. It's a place where you can interactively run Python code without first writing it as a script. This is an incredibly useful feature for tinkering and debugging because you can try code snippets quickly and adjust as you go.

Pros:

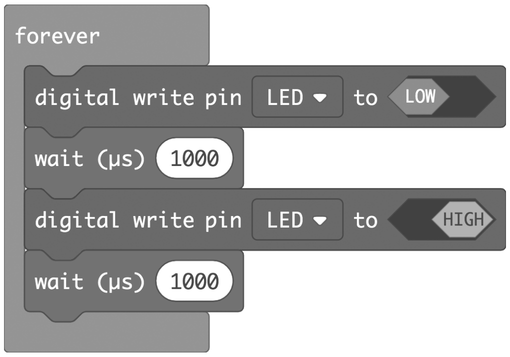
- Python is a fairly novice-friendly language
- REPL makes for easier tinkering and debugging

Cons:

- Relatively new and may have unexpected bugs
- Not a ton of existing resources and tutorials

MakeCode

The Circuit Playground MakeCode environment (**makecode.adafruit.com**) is definitely the quickest and easiest way to get started, and despite the beginner friendly blocks it's a fairly powerful programming tool. Similar to both Arduino and Circuit Python programming, MakeCode relies primarily on an infinite loop, here labeled "forever," as the core program structure. MakeCode however has a few additional useful tricks, such as event handling for the inputs, some slick widgets for dealing with the colored LEDs, and best of all, a board simulator so you can test your program event without physical hardware available. Like Circuit Python, you install your programs onto the Circuit Playground by dragging a file onto a USB mounted drive. Our blinking LED program can been seen in figure 6.3.



```
forever
    digital write pin LED ▾ to LOW
    wait (μs) 1000
    digital write pin LED ▾ to HIGH
    wait (μs) 1000
```

Figure 6.3 Makecode Code.

Pros:

- Blocks-based language is easier for novices
- Board simulator allows multiple students to share a single physical board

Cons:

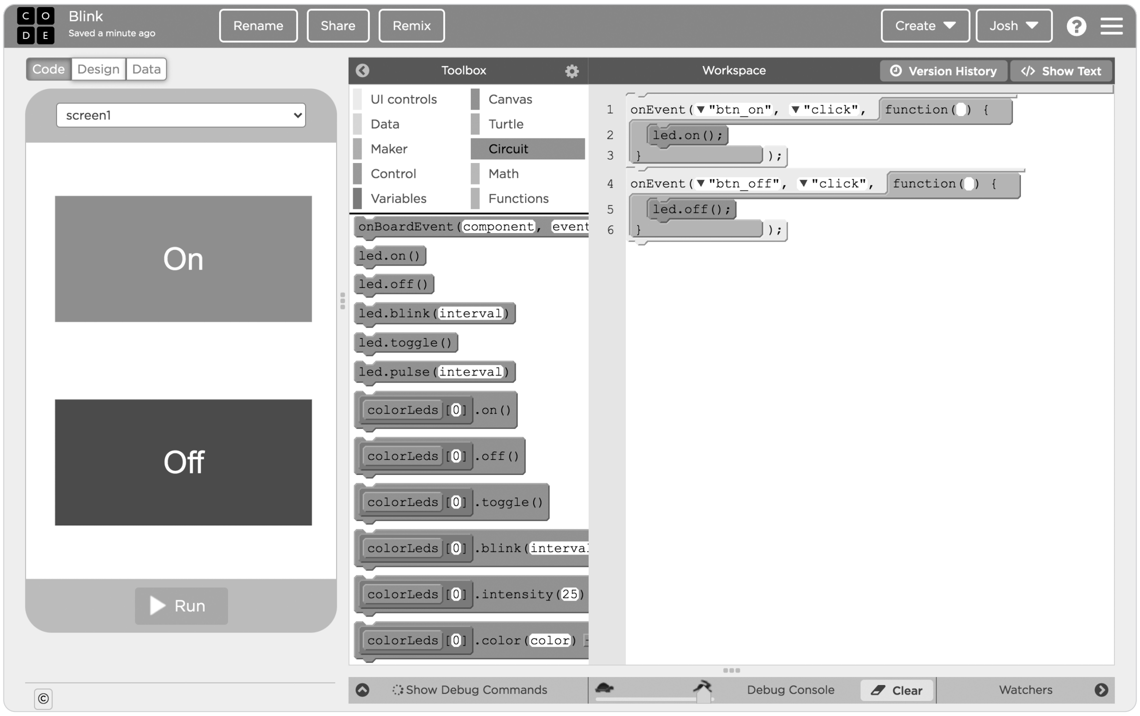
- Blocks can feel constraining as students develop proficiency
- Only a handful of extension libraries available

Code.org Maker Toolkit

The final option is truly a horse of a different color. While each of the previous three environments operated pretty similarly by writing a program structured around an infinite loop and then loading it onto the board, the Maker Toolkit (**studio.code.org/maker/setup**) is designed to control the Circuit Playground from a web app written in a tool called App Lab. The idea behind this approach is to model how hardware interacts with user facing apps without requiring complicated network setup, other wireless communication, or multiple programming languages. All of the code is written in JavaScript, running in your computer's browser, sending messages to the Circuit Playground over USB. The Circuit Playground needs to have a special program called Firmata installed so that it knows how to react to messages from App Lab. There's no infinite loop in this model, and most programming is event driven—for example, if you click a button in App Lab, a light activates on the board. Here's what a blinking LED program looks in App Lab with the Maker Toolkit:

```
led.blink(1000);
```

Okay, that's not a super fair comparison given that some of the other environments also include helper functions that can do simple blinks as well, but App Lab really isn't operating under the same model, so maybe it's worth stretching the example a bit. Take a look at figure 6.4.



```
onEvent("btn_on", "click", function( ) {
  led.on();
});
onEvent("btn_off", "click", function( ) {
  led.off();
});
```

Figure 6.4 Maker Toolkit screenshot.

Instead of blinking at a set rate, this program uses two on-screen buttons to turn the light on and off. With the on-screen app as a new source for both input and output, the Maker Toolkit allows for quickly adding features beyond the built-in hardware. The board has to stay plugged into the computer for this model to work, since it's communicating with the web app live over USB. You are effectively trading physical flexibility (and mobility) for a great deal of software flexibility. Think of it less like writing a program that runs on the board, and more like writing an app that can send commands to, and receive data from, the board. It's not the ideal setup for every physical computing project (particularly those that require significant mobility), but it's a really fantastic way to model the real-life interplay between software and hardware.

Pros:

- Allows students to create user interfaces for physical devices
- Easy transition to hardware if you're already using App Lab (used in Code.org CS Discoveries and CS Principles)

Cons:

- Board must be tethered to your computer while running
- You can only communicate with the board using a specific set of commands, limiting some uses

Classroom Considerations

Of the programming tools we just walked through, only the Maker Toolkit has a classroom focused environment with student account management and a dashboard to track progress. However, now that we're dealing with physical devices there's a whole new tangible realm of concerns to wrestle with. The best organizational approach for you will depend heavily on your classroom layout and whether devices will be shared between students and sections, but I have a handful of guidelines that I've found useful in just about every case:

1. **Track the chain of ownership:** If you're using laptops you may already use some manner of check-in/check-out system—use it for physical computing as well. This isn't just about making sure stuff doesn't go missing; it's also really useful for debugging when a specific device may show a pattern of behavior across multiple students.
2. **Standardize the set:** It may go without saying, but the more you can unify what a hardware kit contains the happier you'll be in the long run. If you have the funds to get a pre-packaged kit then that's a great starting point, but if

you're building up your own kits try to keep it simple and consistent. You can always add on later after you've got some experience.

3. **Keep it all together:** Tupperware containers, tackle boxes, lunch pails, removable filing cabinet drawers—I've seen it all. Get creative and find something that fits the amount of hardware you need (but not too tightly). If you're only using single boards without any accessories, hanging calculator organizers can work too.

4. **Maybe don't keep it *all* together:** If you spend much time doing physical computing projects with kids you'll inevitably find yourself slowly amassing a variety of additional specialized parts like servos, conductive thread, and sensors. Consider defining a "core" set of materials that every kit contains and then separate add on kits for projects that need them. See the list of Extension Kit Ideas for example kits.

5. **Make kids do the inventory:** "The more bits in your box, the harder it is to know when you're losing your bits" is a saying of mine. It happens all the time, you see a stray LED or button on the floor and have no idea where it may have come from, so it goes in the mystery drawer hoping its owner comes looking sometime. Keep an inventory list in your sets and consistently have students confirm their inventory. It won't end the mystery drawer, but it will definitely help curb the tide and ensure that it's easier for wayward bits to find their way home.

Extension Kit Ideas

Wearables:

- Conductive and regular thread
- Coin cell batteries & holders
- Metal snaps (for switches/buttons)
- Velcro strips
- Fabric scraps

Movables:

- Servos
- H-bridges (motor controllers)
- Small motors
- Technics, Erector, K'Nex, or other building toys

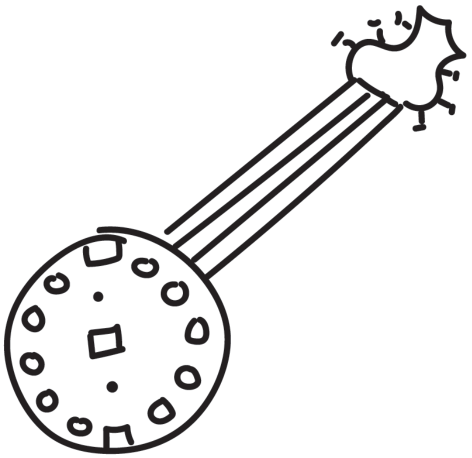
Interactables:

- Switches and buttons
- Light and sound sensors
- Speakers and buzzers
- Resistors
- LEDs

Lesson Sketches

The following sketches can be used with any of the programming tools I mentioned earlier. Because of this I've avoided including sample code but, as always, you can find additional resources on the website.

Invent an Instrument



This activity can pair really well with the activities in the Music chapter, either as an instrument to play alongside a programmed song or to perform with another student live coding. Either way, be ready for some glorious noise when taking this one on.

Preparation

For this you'll want as diverse a set of crafting materials as you can get your hands on. Anything you can imagine kids building an instrument around are fair game—and in particular pay attention to conductive material that can be used with the touch sensors. At a minimum you should stock up on sturdy cardboard and tape, but also consider:

- Styrofoam packing material
- Plastic bottles and containers
- Tubes (old hoses, surgical tubing, etc.)
- All kinds of tape (including conductive copper tape)

If you've got the space in your classroom it never hurts to keep an ongoing collection of potential maker materials—you never know what a student my make magic out of.

Activity

Prime the pump for this activity by asking students to list out as many different musical instruments as possible, keeping a list that the class can refer back to later. In the likely case that your class list is heavily or exclusively comprised of the euro-centric instruments that you'd commonly find in a school band, it's useful to have a handful of instruments from a broader range of cultures to add to the mix, such as the mbira, erhu, or didgeridoo—you can find a slide deck with pictures of these and more on the website for this book. Once you've got a nice diverse set of instruments, ask the students to categorize the instruments by their *interface*—the way a user interacts with the instrument to make music. You're likely to get categories like:

- Blowing
- Buzzing lips or reeds
- Pressing buttons or keys
- Plucking strings
- Bowing strings
- Fretting strings

With that list in place, pair students up with a Circuit Playground and ask them to brainstorm ways in which the inputs and sensors on the board could either mimic some of the instrument "interfaces" they already listed, or provide a whole new way to "play" an instrument. If students haven't already been directly exposed to all of the sensors on the Circuit Playground, encourage them to open up Make Code and look through the "Input" category in the toolbox for ideas.

Once students have had a chance to think through in pairs and then share as a class the ways they could "play" a Circuit Playground instrument, it's time to set them loose making their own. You can leave the scope of this totally open if you like, or specify that students must apply certain concepts that they've learned in class. Either way, I think it's worth having them spend some time planning and mapping out just how their instrument is intended to be played before building and programming. At a minimum this plan should include:

- How the instrument will be held
- How a note/sound will start and stop

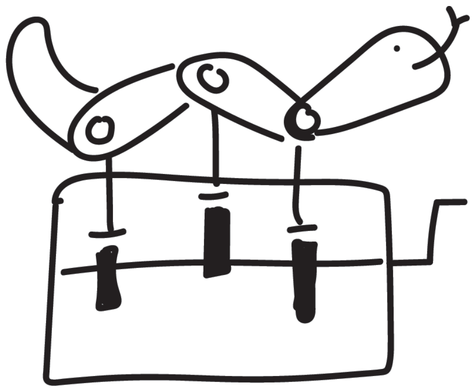
- How the pitch will be controlled
- How the volume will be controlled
- If/how other qualities of the sound will be controlled (e.g. timbre, echo, vibrato, etc.)

From there set the crowd loose and enjoy the noise. As the instruments take shape encourage students to swap, try to play each other's instruments, and provide constructive feedback. If you're truly brave get everyone to play their instruments together (and definitely share it with me online).

Concept Focus: Decomposition

By analyzing how a variety of musical instruments are played students get practice decomposing a single common element of a set of (potentially) complex systems. This type of decomposition is particularly useful when you're looking for guidance and inspiration when programming in a new context.

Cardboard Automata



To be totally transparent, I hesitated to add this activity because it requires some additional hardware (either servos or motors), but the magic of movement is just too great not to share. The setup here is a little more involved, and while the software side of things is dead simple, the mechanical end can be a bit finicky. All that said, the end result is really fun and well worth the investment. If you don't have servos or motors available but still want to give this a try you can make hand-cranked automata and still embed CT into the design and problem-solving aspects.

Preparation

The automata design here makes use a of a cam and piston to move one (or more) rods. Using the above image for guidance, there are four different elements that you'll need to source for the basic mechanism that you see in figure 6.5.

1. **The cam and follower material.** The cam and follower need to be strong enough to rub up against each other without bending, binding, or breaking. Stacked cardboard wrapped in tape or thick foam work well. If you have access to shop tools, then wood or plastic are great choices as well.
2. **Axle and piston material.** Thin wood dowels can be used both for the axle holding the cam to the frame and the piston that is pushed by the cam follower. I've had good luck with bamboo kabob skewers for this. If you're going the hand-cranked route, these can be used for your crank handle as well.
3. **Piston bushing.** You want to reduce friction where the piston passes through the top of the frame with some sort of bushing. Plastic straws cut in half work great here.
4. **Frame material.** You need something to hold the whole thing together, like small cardboard boxes.

That gives you everything you'll need to build the basic mechanism and structure, but the real fun of this project is designing the automaton itself—the thing that's moving. Any general craft supplies that you have will be useful here.

In order to control your automata with a Circuit Playground, you'll also need a motor or servo. The simplest way to do this is with a 360-degree microservo because they're easy to mount on the build, can be controlled with fair accuracy, and have a lot of other potential applications. If you want to use cheap motors instead, be aware that may have to account for excess power draw by using a motor controller (such as the ones on the Circuit Playground Cricket extension board) and definitely need to slow the motor speed down using gear reduction or a belt and pulley system.

Activity

This first part of this activity is entirely unplugged—first building the structural components of the automaton and then the creative component of whatever will be moving. This can be done in pairs or groups of three, and you should budget

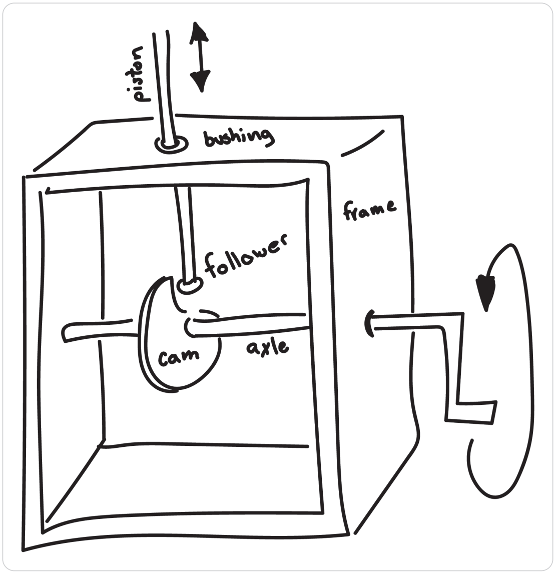


Figure 6.5 A basic cam follower.

time for students to experiment, make mistakes, and revise. Avoid providing students with an overly prescriptive structure, as this activity is a rich opportunity for students to navigate through ambiguity and iteration as they take the broad description of how a cam follower automaton works and then figure out how to apply it with the resources available to them. Start just by sharing a few photos of videos of cam follower mechanisms, which you'll find on the accompanying page for this activity on my website.

With the examples in mind, ask students to describe the behavior of these devices in terms of an algorithm. You may need to clarify that we're not looking for an algorithm that describes how to build an automaton, but rather the algorithm for how the automaton translates input (in terms of turning the crank) into output (the motion of the automaton itself). You should end up with something along the lines of:

- A hand motor turns the input crank
- The crank rotates the cam
- The cam transfers the rotational energy of the crank into vertical movement of the piston, based on the cam's shape
- The piston pushes on part of the automaton (which may cause other parts to move in turn)

This discussion is an opportunity to highlight two important concepts for this lesson. First, an algorithm can be expressed mechanically as well as in software. While students may have done activities describing physical activities algorithmically, it's unlikely that they've really considered the ways in which physical objects might actually execute an algorithm, alone or in conjunction with software. Second, the shape of the cam is the key to creating interesting motion in these devices. Really encourage students to explore how various cam designs, or even multiple cams and pistons, can achieve different kinds of motion in their automata. You can also share this page of helpful tips from Cabaret Mechanical Theatre to get student thinking about the uses for various cam shapes: **bit.ly/35dDdcn**.

Provide students with ample time and space to build, test, and refine their automata before optionally adding in the Circuit Playground and servo to control the crank. The programming for this is quite simple—at a minimum, students need some way to start and stop the servo. This can be done with the buttons or switch, but encourage them to also think about the additional benefits the board gives them, both in terms of the inputs that control the servo (such as using the light or sound sensor) and also the movement of the servo itself. Do you want it to rotate in both directions? Or maybe not rotate all the way, but waver back and forth? Note that the shape of the cam may prevent it from rotating in one direction, so always test gently with your hand before testing with the servo.

When all is said and done, revisit the question of how to describe these mechanisms algorithmically, but now more specifically related to each groups' automaton. If you integrated the Circuit Playground make sure that students include both the software and hardware components of their algorithms.

Concept Focus: Algorithm Design

This activity introduces students to algorithms that span past the bounds of software, which is a key understanding in physical computing and engineering more broadly. Many of the introductory activities in physical computing can feel like the physical component is just a smaller version of a computer—blinking lights instead of shapes on the screen, beeps instead of playing recorded audio. That's not to say these experiences aren't valuable, but when we can add actual physical motion into the mix we move into a space where how things are built has a real impact on what they can do. This is a great jumping-off point for students to explore different ways that motion can be transformed through gears, cogs, pulleys, and linkages and how those physical systems are expressions of algorithms.

Alternatives

There are tons of physical computing boards out there, and it can be pretty difficult to peel apart the real differences. Instead of providing a comprehensive list I'm going to highlight a handful of boards that I like and which fit some different niches.

- **Micro:bit (microbit.org)** Developed by the BBC as part of on effort to bring CS to students in the UK, the Micro:bit does quite a bit at a very low price. It lacks some of the sensors that the Circuit Playground features, but its grid of red LEDs can be used to display text and simple animations, and it has a Bluetooth radio that can be used to write programs that communicate between multiple boards.
- **Makey Makey (makeymakey.com)** This board comes in at a higher price point than either the Circuit Playground or Micro:bit, but brings with it a much simpler (if less flexible) approach to working with hardware. Instead of programming the board itself, the Makey Makey shows up on your computer as a keyboard, allowing you to use it in any programming environment where you could respond to key presses.
- **Arduino Uno (arduino.cc)** A veritable classic, compared to these other boards the Arduino Uno has little more than a processor and a lot of input/output pins. The obvious downside is that you don't have a lot of built-in things to play with, but the tradeoff is both a lot more flexibility to build those things yourself and a ton of tutorials on the internet to help you. This is a compelling route if you want to get into electronics and engineering, but less so if you plan on a briefer physical computing unit.

3D Printing

I can't tell if the wave of 3D printing in schools has crested and is fading into the background, or if we're just waiting for the right machine at the right price. The first 3D printer I had in my classroom was a MakerBot Replicator sometime around 2012, and my experience using it with students was a constant rollercoaster. When it worked, it was like alchemy; students felt like they were spinning gold from flax (or more accurately key chains out of plastic) and the hum and buzz emanating from the back corner of my room felt like a factory for student creativity. When it didn't work I was spending all of my free time getting the nozzle unclogged, leveling the build plate, or troubleshooting why half of my students' designs wouldn't

Artist Spotlight: Samer Dabra

Samer Dabra is a London-based software engineer and artist who blends the digital and analog worlds with this robot drawn art.

You can see Samer's latest creations on his website **spongenuity.com**.

The beauty of using CS to create art is that we are able to go beyond what is possible with just the human hand and eye to create something greater—and that originated in the human mind! I've always been someone who has both an aptitude for the technical side of things as well as the creative, once I was introduced to creative coding, I was excited to have an outlet for both and a way to express myself that is unique.

—**SAMER DABRA**

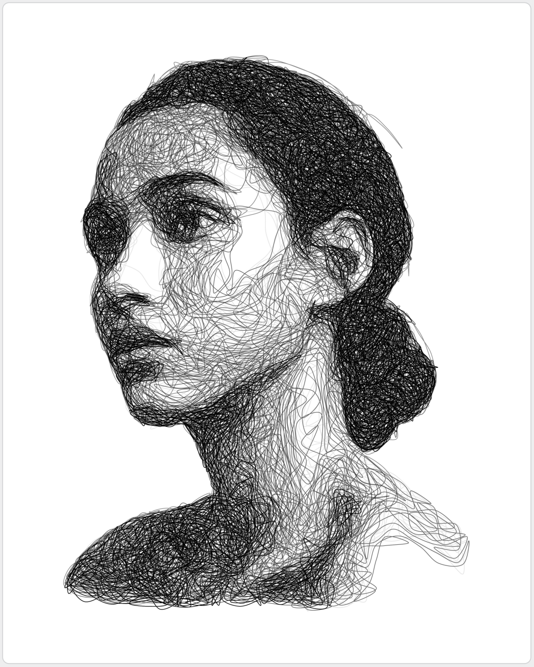


Figure 6.6 Computer science-infused artwork by Samer Dabra.

slice. In these moments the gentle hum and buzz transformed into a gnawing and gnashing that picked away at the focus of everyone in the room. My students experienced some great successes and learned a lot—including printing and building more 3D printers from scratch—but to this day I'm not certain if the overall time investment was worthwhile.

That may be a bleak outlook, and certainly the reliability and stability of commercial 3D printers has come a long way in the near decade since I first started teaching with them, but for an investment that can easily reach into multiple thousands of dollars (before even considering the consumable cost of plastic) it should give us pause. Whether you already have a 3D printer in the classroom or you're looking to get one, there are some critical questions to reflect on first. I bring these up specifically in the context of 3D printing, but they're useful reflections anytime you're tempted to bring something new and shiny into your classroom (a temptation to which I have frequently fallen victim)—just replace every instance of "3D printing" with your shiny thing du jour.

- **What am I trying to teach with this?** Are you looking to teach the thing itself (if so, why?) or is there something important to your curriculum that 3D printing is going to be exceptionally good at teaching? Is it bringing a more meaningful or relevant context for your students? If there's a curriculum that comes along with it, look to see how much of the instructional time is spent on concepts you care about.
- **How am I making room for this new thing?** Is this replacing something in your curriculum because 3D design or printing is the only or best way to do it? If not, what are you cutting to make room for it? Be realistic about the amount of instructional time that you'll need to provide a meaningful experience and the cost of what you're cutting, particularly for underserved students.
- **How will I ensure equitable access for my students?** 3D printers have gotten faster, but they still aren't really *fast* compared to producing something in software. Does your plan account not just for the long time it takes to print, but also the frequency of failure and need to reprint? If the solution is just that some kids will use it, the kids who are faster or ahead, then really reflect on how that impacts the rest of your students, not just in regards to their access to this learning but also how your own time and attention will be diverted from those that may need it more.

The best way to bring 3D printing into a CS classroom in a way that aligns well with other concepts you're teaching, reduces the time bottleneck, and ensures that every student has equitable access to limited hardware is to shift the focus from the printing of things and towards the designing of things that could be printed. It's a slight nuance, but one that has helped me to focus and prioritize my time as a teacher; it's okay to spend time that helps students to design things, less okay to spend that time making sure a given thing is printed. More importantly, it removes the physical device out of the equation in a way that lets you teach this even if your printer is broken, or you don't have one in the first place. There are plenty of print-on-demand services that could be used to print out student work and they don't prevent students from making progress.

You probably think of a 3D model as something visual, something with shape and volume, but actually it's actually all just code. Specifically, most 3D printers operate on something called G-code, which is a set of instructions that control the stepper motors (and other elements) on a 3D printer, CNC mill, laser cutter, or other computer controlled manufacturing device. The G-code to print a single layer square outline might look like the following.

```
M109 S190     ; Heat the nozzle to 190 degrees C
G28           ; Set all three axes (X,Y,Z) to their home position
G1 X10 Y10    ; Move the nozzle to 10mm along both the x and y axes
G1 X30 Y0  E3 ; Move 30mm along the x axis while extruding plastic
G1 X0 Y30  E3 ; Move 30mm along the y axis while extruding plastic
G1 X-30 Y0 E3 ; Move -30mm along the x axis while extruding plastic
G1 X0 Y-30 E3 ; Move -30mm along the y axis while extruding plastic
Z10           ; Move the nozzle out of the way, 10mm up the z axis
```

The first think you're likely to notice about G-code, particularly in comparison to the other languages we've used, is that it's very terse. This is not a language that many people ever write by hand, and it's certainly not the code that students are going to use directly when create their designs. That doesn't mean, however, that it doesn't have a place in the classroom. Showing students the kind of code that actually runs on their printers can be a great place to talk about the layers of abstraction in computing. Every programming language sits somewhere in a spectrum from understandable by a human down to understandable by a computer (see figure 6.7). Languages that we use in the classroom are optimized to be very human readable, so they have to be translated down to further layers of the software stack, which has a performance cost. Getting programs to run faster often requires using languages that are closer to the hardware—the closest being binary machine language. We don't need to program directly in G-code because we can use a program called a slicer to do the work of translating a 3D model to G-code.

Depending on your 3D printer software you may be able to view and edit the G-code before it goes to the printer, or use a live console to send individual G-code commands directly to the printer, as in figure 6.8.

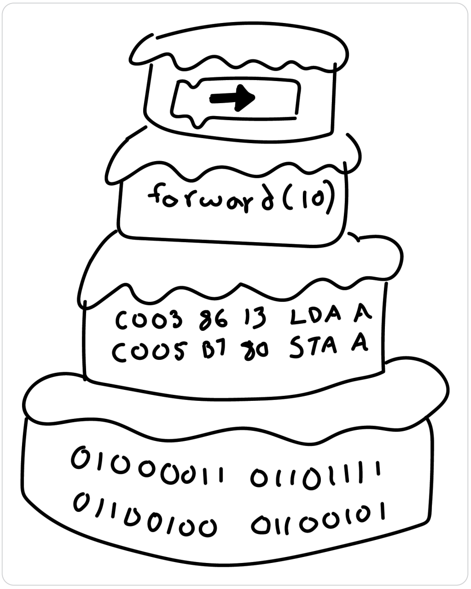
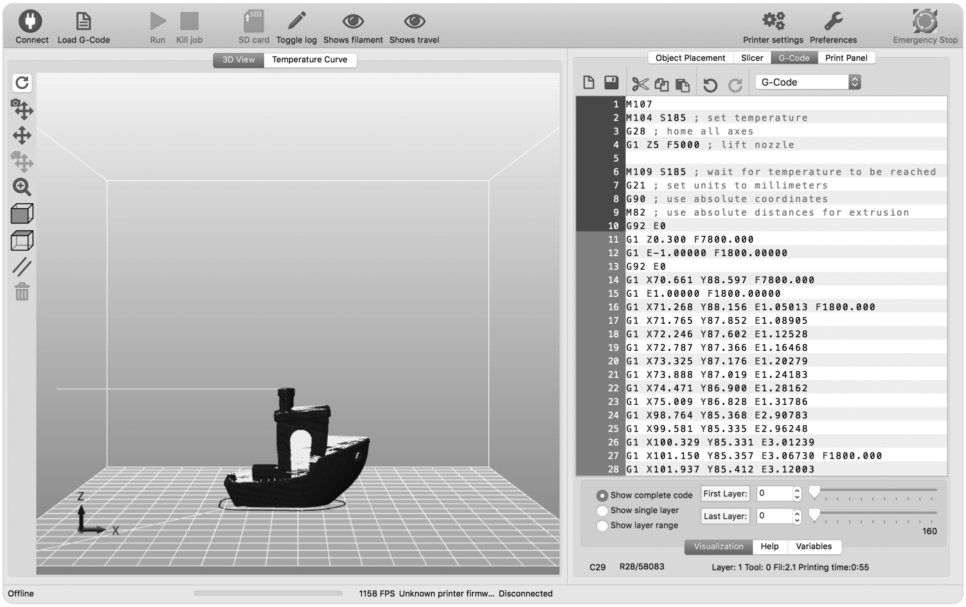


Figure 6.7 The spectrum of programming languages.



```
M107
M104 S185 ; set temperature
G28 ; home all axes
G1 Z5 F5000 ; lift nozzle

M109 S185 ; wait for temperature to be reached
G21 ; set units to millimeters
G90 ; use absolute coordinates
M82 ; use absolute distances for extrusion
G92 E0
G1 Z0.300 F7800.000
G1 E-1.00000 F1800.00000
G92 E0
G1 X70.661 Y88.597 F7800.000
G1 E1.00000 F1800.00000
G1 X71.268 Y88.156 E1.05013 F1800.000
G1 X71.765 Y87.852 E1.08905
G1 X72.246 Y87.602 E1.12528
G1 X72.787 Y87.366 E1.16468
G1 X73.325 Y87.176 E1.20279
G1 X73.888 Y87.019 E1.24183
G1 X74.471 Y86.900 E1.28162
G1 X75.009 Y86.828 E1.31786
G1 X98.764 Y85.368 E2.90783
G1 X99.581 Y85.335 E2.96248
G1 X100.329 Y85.331 E3.01239
G1 X101.150 Y85.357 E3.06730 F1800.000
G1 X101.937 Y85.412 E3.12003
```

[✓ Show complete code] Show single layer Show layer range
First Layer: 0
Last Layer: 0
160

C29 R28/58083 Layer: 1 Tool: 0 Fil:2.1 Printing time:0:55

Offline 1158 FPS Unknown printer firmw… Disconnected

Figure 6.8 G-code for a 'benchy' tugboat.

Tool Exploration: OpenJSCAD

The most common way to create 3D models is using drag-and-drop modeling tools like AutoCAD, but that's not the route we'll be heading down. Instead, we'll use a programming tool called OpenJSCAD to develop models with code. Before going any further, I feel like we need to peel apart the onion layers in that absolute mouthful of a name. Let's start from the end and work our way backward. Here are a few definitions:

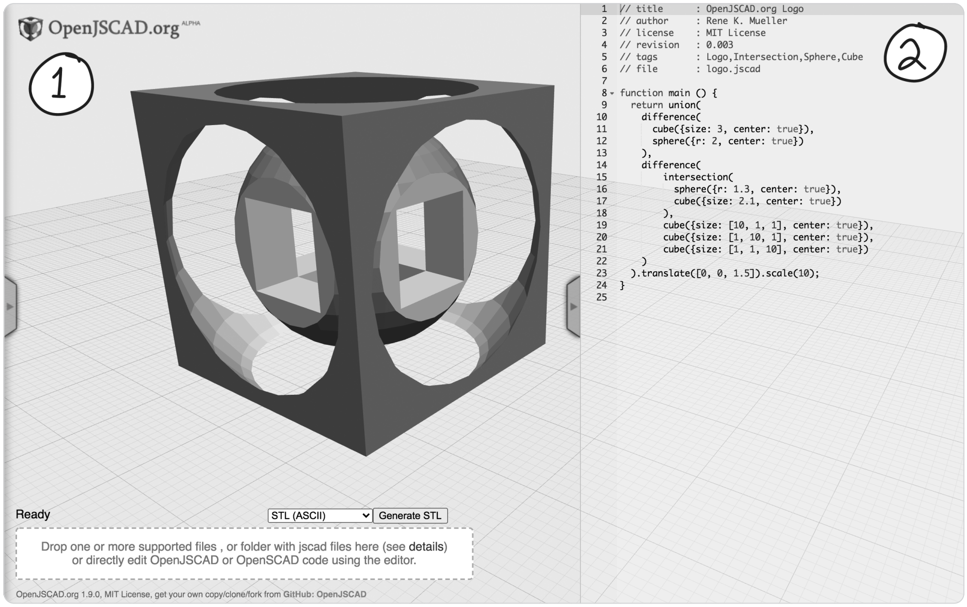
- **CAD** Computer Aided Design is the use of computers to aid in the design of things (clever, yeah?). Architects, engineers, and artists all use CAD software to create detailed and sophisticated designs, blueprints, and models. CAD software like AutoCAD can be used to create models for 3D printing without any need for code, but where's the fun in that?
- **SCAD** *Solid* Computer Aided Design. Like CAD, but solid. As in specifically designed for models that will be physically printed.
- **OpenSCAD (openscad.org)** An open source program designed for scripting the generation of 3D objects that runs on Windows, Mac, and Linux.
- **OpenJSCAD (openjscad.org)** Like OpenSCAD, but in JavaScript.

So, why OpenJSCAD? In short, it runs in the browser and it uses a language that students in my class have already seen (and which is used by many other tools in this book). Functionally OpenSCAD and OpenJSCAD are almost identical, so if you don't want JavaScript you can stick with OpenSCAD. If you don't want text at all then you can check out a BlocksCAD, a Blockly-based version of OpenSCAD available at **blockscad3d.com**.

A Quick Tour

There's not much to the OpenJSCAD interface, so this will be the quickest of quick tours. See figure 6.9.

1. This 3D rending of your code can be panned, zoomed, and rotated to inspect all areas of a model. The arrow on the left edge of the screen pops out a reference panel with links to documentation.
2. The code area on the right is little more than a formatted text box, nothing fancy. To see changes to your code in the rendering area you can click Shift + Enter.


OpenJSCAD.org ALPHA
1
2
Ready
STL (ASCII)
Generate STL
Drop one or more supported files , or folder with jscad files here (see details)
or directly edit OpenJSCAD or OpenSCAD code using the editor.
OpenJSCAD.org 1.9.0, MIT License, get your own copy/clone/fork from GitHub: OpenJSCAD

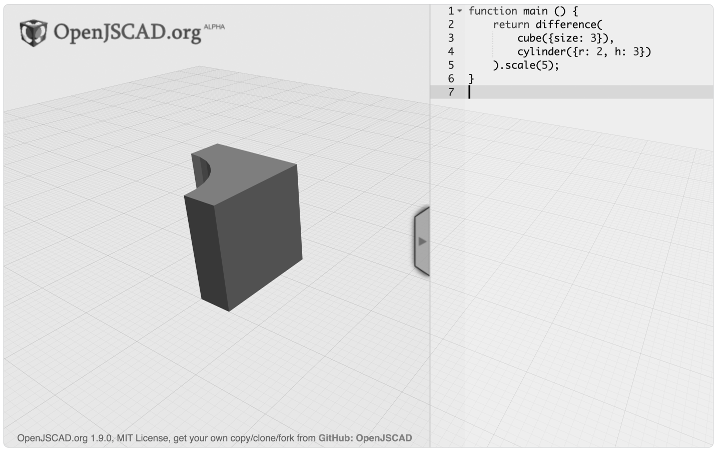
```
// title      : OpenJSCAD.org Logo
// author     : Rene K. Mueller
// license    : MIT License
// revision   : 0.003
// tags       : Logo,Intersection,Sphere,Cube
// file       : logo.jscad

function main () {
  return union(
    difference(
      cube({size: 3, center: true}),
      sphere({r: 2, center: true})
    ),
    difference(
        intersection(
          sphere({r: 1.3, center: true}),
          cube({size: 2.1, center: true})
        ),
        cube({size: [10, 1, 1], center: true}),
        cube({size: [1, 10, 1], center: true}),
        cube({size: [1, 1, 10], center: true})
    )
  ).translate([0, 0, 1.5]).scale(10);
}
```

Figure 6.9 The OpenJSCAD environment.

OpenJSCAD Programming Model

OpenJSCAD uses a programming paradigm that isn't seen very often in K–12 CS called **functional programming**. In this paradigm we treat just about everything in a program as a purely mathematical function. Functions in this approach are considered **first-class citizens**, which basically means that they can be assigned to a variable, passed as an argument to another function, or returned from a function. In practice this means that we are frequently nesting lots of functions together, as one function may be taking another function as an input, which may in turn be taking another function as an input, and so on. It's functions all the way down. This transition in thinking can be a bit difficult for some students, particularly when they've gotten used to thinking of programming as a sequence of commands (an *imperative* programming paradigm). It's worth noting that although OpenJSCAD encourages a functional programming approach, you aren't actually forced to use functional programming exclusively.

So how does this functional programming paradigm impact how we make things in OpenJSCAD? For one, your whole model rolls up into the return value of a single `main()` function, so in order to produce anything more complicated than a single shape you have to nest functions that add and remove shapes from each other. Let's walk through the example in figure 6.10.


OpenJSCAD.org ALPHA
OpenJSCAD.org 1.9.0, MIT License, get your own copy/clone/fork from GitHub: OpenJSCAD

```
function main () {
    return difference(
        cube({size: 3}),
        cylinder({r: 2, h: 3})
    ).scale(5);
}
```

Figure 6.10 A simple OpenJSCAD model.

Basic Principles of Functional Programming

In general you can think of functional programming as code that follows the rules of Algebra (or more properly lambda calculus, but algebra is a more accessible reference for most students). There are lots of times in programming where we do things that *look like* math, but behave quite differently. In functional programming we want everything to behave the way a function in algebra class would function. This includes:

- **Consistent Output.** Every time you provide a given set of inputs to a function you get the same output. In other words, 1 + 1 is always 2.
- **No Side Effects.** The only thing a function can do is return a value based on its input, you can't touch stuff outside of the function, such as changing a global variable.
- **Variables are Immutable.** Once you've given something a name, its value can never change. If I say $x = 1$ then I can't later say $x = x + 1$.

The first thing to notice is that everything is wrapped inside the `main()` function, so at the end of the day the only thing that will get put on screen is whatever value we return from that function. Now instead of working from the top of the return value, I think it's a bit easier to work from the inside out.

- **Lines 3 and 4** are the two shapes at play here, a 3 x 3 x 3 cube and a cylinder with radius 2 and height 3.
- **Line 2** returns the output of the `difference()` function that the cube and cylinder are being passed to. This function will remove the second shape from the first, giving us a quarter cylinder cutout.
- **Line 5** calls the scale method of the shape that's being returned to scale it up 5x.

It can take some time getting used to thinking in this nested functional manner, combining shapes by passing them through various translations and Boolean combinations. My advice is to avoid thinking sequentially from top to bottom and instead try to think from the inside out. The outermost function that represents your shape is the culmination of all of the functions that live inside it. Once you have the basic understanding of how elements are passed through to each other, we can start to talk about the different kinds of elements available. Broadly, they fall into two categories.

1. **3D Primitives:** Basic 3D objects such as cubes, spheres, cylinders, tori, and text. These shapes are already fully three-dimensional.
2. **Extruded 2D Primitives:** These start as 2D paths like squares, circles, or any arbitrarily complex 2D path that is pulled through space to make a 3D object.

```
cube({size: 1}); // Creates a cube that is 1 x 1 x 1

linear_extrude({height: 1}, square({size: 1})) // Creates a 1 x 1
square and then extrudes it by 1 to create a 1 x 1 x 1 cube
```

Compare the two lines above, both of which produce a 1 x 1 x 1 cube. To extrude a square we call the function `linear_extrude` with a call to the function square as one of its arguments. This nesting is crucial to how you think about building up complex shapes using transformations and Boolean operations.

Boolean Operations

You may have introduced Boolean values to your students as the dichotomy of true and false. Perhaps you've also used Boolean operators for something like checking if an age is greater than twelve *and* less than 19. The Boolean operations in 3D modeling are a bit of a departure from the simple world of true/false that we're used to in intro CS. In 3D modeling we use Boolean operations to describe how shapes should be combined. The three Boolean operations in OpenJSCAD are:

- **Union** (Boolean OR): Any area that is included in either one shape OR the other will be included in the combined shape.
- **Intersection** (Boolean AND): Any areas that are part of both the first AND second shape are included in the combined shape.
- **Difference** (Boolean NOT): Any area that is part of the first shape but NOT part of the second will be included in the combined shape. Unlike the other two, order matters here so pay attention to which shape is listed first.

Using these three operations we can build up quite complex shapes from simple shapes. I find it really helps to speak the whole thing out to help internalize how they work. For example, for the operations in figure 6.11 I might say:

- The *union* of the circle and square includes any are that is part of the circle *or* the square.
- The *intersection* of the circle and the square includes only the area that is part of the circle *and* the square.
- The *difference* of the circle and the square includes the area that is part of the circle but *not* the square.
- The *difference* of the square and the circle includes the area that is part of the square but *not* the circle (note how the order changed).

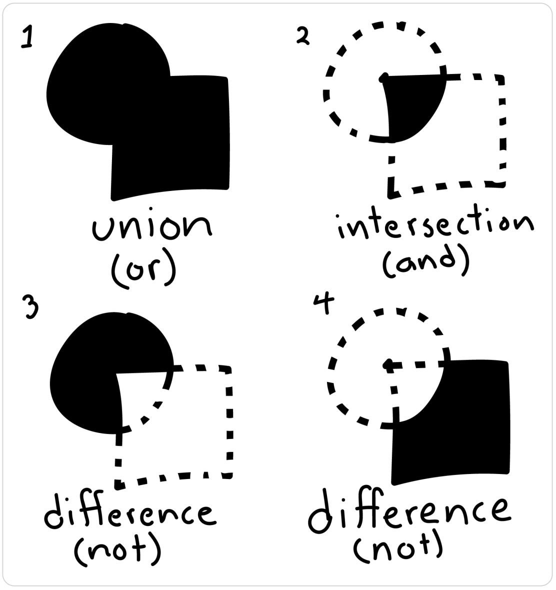


Figure 6.11 Boolean operations.

Transformations

Finally, there are a number of transformations that you can apply to shapes to further modify their location, rotation, and size. Transformations can be applied either in the functional model (as a function that you pass a shape into) or in an object-oriented fashion as a method attached to the end of your shape with dot notation. Let's compare the two below.

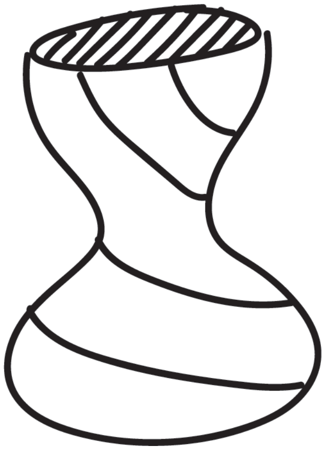
```
// Make a cube and move it 1 along the x axiz, 2 along the y, and 3
along the z
translate([1, 2, 3], cube({size: 3}));

// Do the same thing, but with a method
cube({size: 3}).translate([1, 2, 3]);
```

Aside from `translate`, which moves a shape from its default location, there are functions to rotate, scale, and center your shapes.

Lesson Sketches

The following sketches are a good way to have students experiment with Booleans and models that can be printed with a 3D printer.

Make a Container



Of the many dozens or even hundreds of 3D printed trinkets that students left on my desk, the ones that stuck around fell into two camps, the quirky and the useful. The quirky ones are hard to predict and widely varied, and they survived the trash can because honestly who is going to toss out a plastic model of their own head on a rubber duck body? The useful, on the other hand, survived the trash because they made themselves indispensable. As a collector of many things (quirky prints included), I often find myself in need of containers like boxes and vases. Not only do they give you (or your students) a good place to stash your goodies, they are a really simple introduction to the way in which basic shapes can be combined with Boolean operations to make more complicated objects.

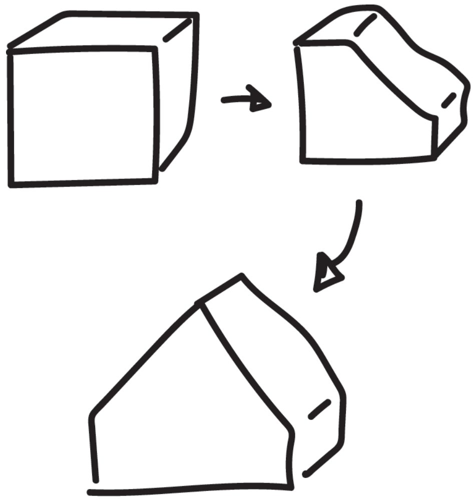
Activity

This activity works best without a ton of guidance, just a broad goal and a lot of room for students to explore how best to reach that goal. Specifically, we want to create containers by removing the inside of a shape using the `difference()` function, for example you could remove the volume of a cylinder from a slightly larger cylinder to create a basic straight sided vase. That's really all there is to it. Each student can play with the different shapes and transformations to make boxes, cups, bowls, or any other container they can imagine.

Concept Focus: Booleans

As I mentioned in the tool overview, the way that we approach the concept of Booleans in this context is richer and more nuanced than we may typically see in introductory CS. This activity specifically lends itself to a Boolean *not* to subtract the volume of one shape from another.

Evolving Shapes



Imagine a game of telephone, but instead of a verbal message getting mangled and morphed over time, it's a model that gradually changes and evolves as it passes through more hands. That's the basic idea here. This activity can really help students understand the nested nature of functional programming as they gradually add to (or subtract from) their collective models.

Activity

Place students into groups of three to four and ask each group to come up with a relatively simple model that they want to create. You want something that's going to require multiple different shapes combined in interesting ways. I would tend towards something that feels a bit past your students' reach rather than something that's too simple because it's just more fun that way. Reach for the stars; even if you miss, you'll end up with an interesting-looking mangle of shapes.

Once every group has selected a goal model, they should spend five to ten minutes breaking down that shape into its component pieces, keeping OpenJSCAD's primitive shapes in mind. It's a pretty safe bet that students will naturally tend towards thinking of things from a purely additive viewpoint—in other words, it's likely that they'll focus on building up a shape out of building blocks and miss opportunities to use other Boolean operators like difference. Prompt them as necessary to encourage thinking past the "building block" model.

Is it Printable?

Designing a 3D model is one thing; ensuring that it can actually print effectively is another. As students work on their evolution, encourage them to constantly question whether or not it will be printable. This inquiry becomes more natural and nuanced as students get more experience with 3D modeling and printing, but early on it can be as simple as making sure that there's actually a bottom layer. An object that's just floating in air won't stick to the printer's build plate!

The next step is to start actually building out the models, and this is where it gets fun. In the spirit of telephone, each group should define a sequential order they'll work in. Each student gets to add only one additional function before passing the program to the next one. Building the models in this way is challenging, but encourages groups to think systematically about how to structure their programs.

Concept Focus: Decomposition

The key to building a moderately complex model from basic shapes is breaking down that shape at the start. It's tempting for students to jump right into a problem without spending the time up front on decomposition, which is exactly why this activity is structured to push that decomposition to the forefront.

Alternatives

- **BlocksCAD (blockscad3d.com)** This Blockly-based implementation of OpenSCAD can be used as a drop in replacement for any of these activities and comes with the added benefit of removing syntax from the equation for students who need that additional scaffold.
- **Tinkercad Codeblocks (tinkercad.com/learn/codeblocks)** Tinkercad is an easy to use interface for building 3D models from a drag and drop GUI, and the Codeblocks extension brings programming to the platform as well.
- **Rhino Grasshopper (rhino3d.com)** Rhino is a professional quality CAD tool that could be used for a much more advanced 3D design course. Grasshopper is an integrated programming tool that allows you to build sophisticated scripts that allow for advanced and detailed 3D models.