

HOPSCOTCH

Curriculum

Learn to Code - Make Cool Stuff

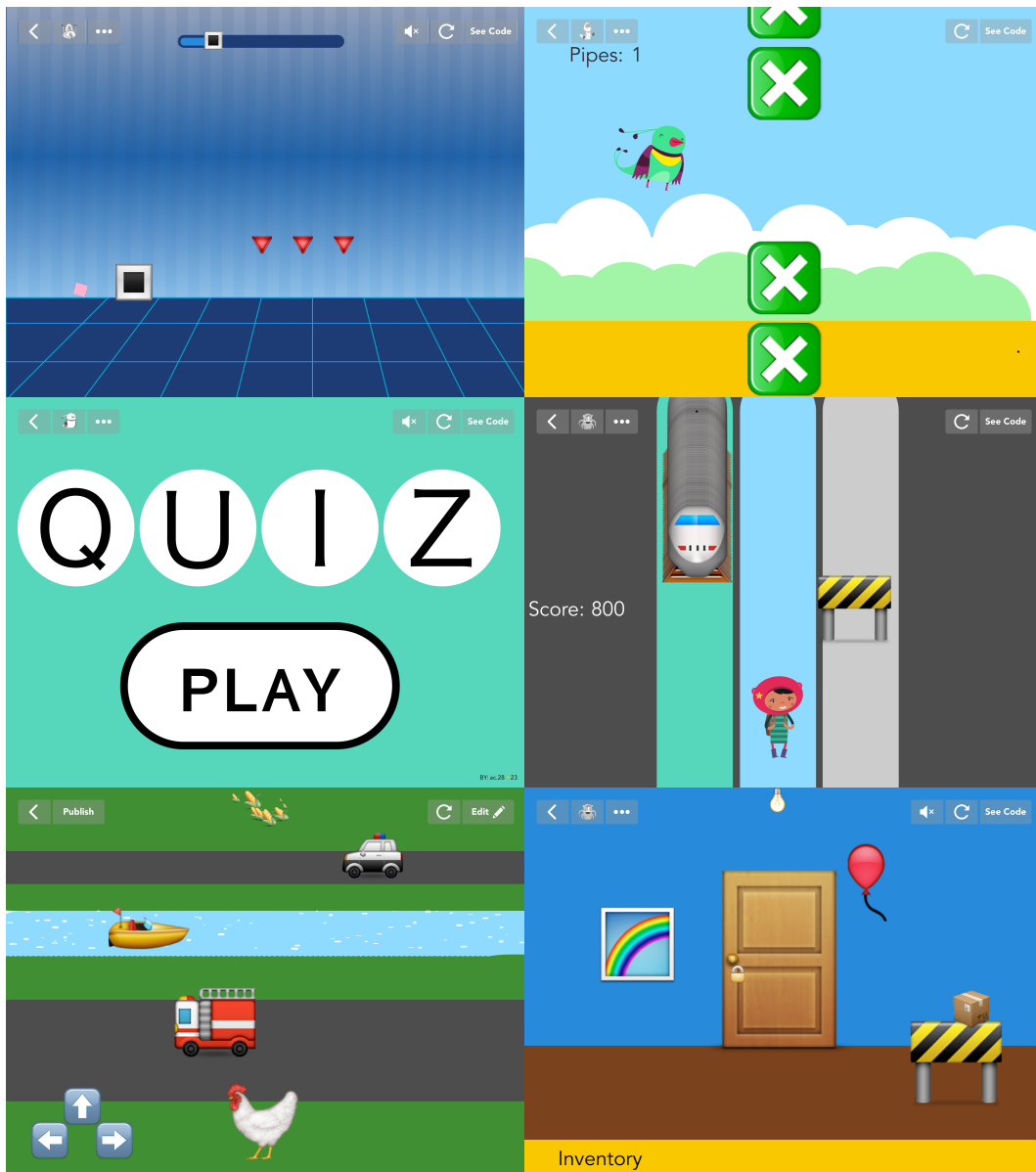


TABLE OF CONTENTS

2	Overview
4	Materials
5	Core Coding Concepts
6	Standards
7	Guide to the Lessons
11	Lesson 1: Crossy Road
23	Lesson 2: Geometry Dash
33	Lesson 3: Which Emoji Are You?
45	Lesson 4: Flappy Bird
56	Lesson 5: Subway Surfers
67	Lesson 6: Can You Escape?
77	Optional Extra Lessons & Extensions
78	Rubric for Evaluating Student Work
79	Glossary for Younger Students
80	Glossary for Older Students
81	References
82	Acknowledgments

Dear Educators,

Hi!

We're *really* excited that you're going to teach your students to program, both for them and for you. Kids have remarkable imaginations, and creating computer programs is an amazing way for them to express themselves. We've seen kids create astonishing things using our simple but powerful tool. We know you'll see the same when using Hopscotch, and hope you share what your students create.

Anyone, regardless of their experience in programming, can teach this curriculum. Just as Hopscotch was built on the principle that anyone can become a great programmer, this curriculum is designed on the premise that anyone can become a great programming teacher.

Programming is a way of thinking, building, and expressing yourself. Just as English is not really about grammar, and history is not memorizing dates, computer programming is not actually about code or computers. Just as we ask students to make connections between events in history, we ask students to investigate the interactions between complex systems in computer science.

But don't just take it from us. Here's what some Hopscotchers have to say:

"The thing I love most about playing Hopscotch is that you can make mistakes and try again and it doesn't matter." — Julia, 10

"Hopscotch is the best platform for expressing our inner creativity!" — Nico, 12

"My kids love working on this app and being able to code has given them a much better understanding of how computers work and has demystified much of the tech in their lives. Now they look at something on the computer and say, 'I could code that!' It has changed their lives for the better." — Jesse, 5th grade teacher

Goals of the Hopscotch Curriculum:

- Equip students with a solid foundation in programming fundamentals
- Expose students to coding culture: Iteration, pair programming, accepting feedback, sharing and attribution
- Enable students to learn transferrable coding skills that prepare them for diving into another programming environment (like Java or Ruby)
- Build self-confidence and comfort taking risks and making mistakes

By learning to program, your students' creative, analytical, and abstract thinking skills will improve, and it will show in their performance in other disciplines. Coding is not just for future software engineers—it's something that anyone can and should explore!

This curriculum builds a foundation in the following Computational Thinking principles:

- Decomposition: Breaking a problem into smaller problems
- Generalization: Seeing the bigger problem
- Abstraction: Understanding significant vs. insignificant details
- Pattern Recognition: Deciding which parts repeat
- Algorithm Design: A process to solve a problem

For more information on Computational Thinking, see the following resources:

<https://computationalthinkingcourse.withgoogle.com>

<http://csta.acm.org/Curriculum/sub/CurrFiles/CompThinkingFlyer.pdf>

Format:

The curriculum consists of six project-based coding lessons and two optional extension lessons.

In each lesson, students will explore the five fundamental computing concepts described above in the process of building a fun game (like the popular Flappy Bird!). We provide an introduction to each game, sample code, and suggested reflection questions. See Guide to the Lessons (page 7) for more details.

We've designed this curriculum for grades 5-8, but it can easily be adapted to meet your students' ages and experience levels.

For younger students, go slower and skip the last part of each lesson. You may also want to consider skipping lessons 5 or 6, which are the most advanced. For older students, encourage exploration and iteration beyond the product completed in the lesson's sample code.

Each lesson is designed to take 45 minutes of code-along instruction. If you have 15 or more extra minutes, use it as free code time for slower students to catch up and for faster students to challenge themselves to embellish their programs. Suggestions for further work are given under the Differentiation section of each lesson.

We hope that you have fun, and look forward to seeing what your students create.

<3,
Dr. Em + the Hopscotch Team

September 2016

You don't need much to teach this curriculum. The most important things to bring to the table are creativity, curiosity, and flexibility. Aside from that, the following resources are all you need:

THE TOOL

The activities in this curriculum require the **latest version of Hopscotch on an iPad or iPhone**. You can download Hopscotch for free directly from the App Store using this link: http://hop.sc/get_hopscotch

Note: We're continually improving Hopscotch. Make sure your version of Hopscotch is fully up to date, or you won't have access to all the code blocks required to make these games!

EMOJIS

We highly recommend using emojis. They are fun, funny, and vastly expand the possibilities of what you can create. You can download the emoji keyboard from the Settings app on your iPad.

You can use any emoji in your project by adding a text object instead of a character. Then tap the smiley or globe in your keyboard to switch to the emoji menu and you can choose what you want from there.

VIDEOS

There are video tutorials for Lessons 1-6. You can absolutely teach this curriculum without them; they are supplementary (though quite fun, if we do say so ourselves :p). They are available on YouTube: <http://hop.sc/hopscotchvideos>

You can use them in a few ways:

- Show the whole video to the class, and after, lead them through the steps to create their games, taking suggestions from the students for what to do next and how to do it.
- Watch the video at home ahead of time to get an idea of a way to lead the class.
- Show the video to the class and have them follow along programming on their own devices, pausing frequently to catch up and discuss the code.
- Have each student self-pace through the video on their own device, with headphones, and code along in their own time. Some students may choose to watch the whole way through once, then code along on the second viewing (Requires robust internet).

CORE CODING CONCEPTS

In this curriculum, each lesson will sequentially explore the following concepts, each of which is fundamental to computer science. Mastery of these ideas will enable students to independently explore more complex programming, including other programming languages. We note what concepts are covered in each section of the lessons with abbreviations.

At the beginning of each sub-lesson, we note which concepts it will cover with the following abbreviations:

Sequence (S) - The order in which instructions are given to the computer

Event (E) - A trigger that a computer recognizes and that causes it to do something

Loop (L) - Code that repeats

Value/Variable (V) - A holder for a number

Conditional (C) - Statements of the form "IF (something is true), THEN (do an action)"

The Hopscotch Curriculum is aligned with both the Common Core Standards for Mathematical Practice and the Next Generation Science Standards for Engineering Practices. They are listed below, and referred to throughout the activities where relevant.

The skill of computer programming itself is deeply rooted in these practices, independent of the content of the program being written. Depending on what app or game a student is making, other content standards may also apply, such as understanding negative numbers, or use of the coordinate plane.

Completion of all eight lessons fulfills all standards.

Common Core Standards for Mathematical Practice

<http://www.corestandards.org/Math/Practice/>

CCSS.MATH.PRACTICE.MP1 Make sense of problems and persevere in solving them

CCSS.MATH.PRACTICE.MP2 Reason abstractly and quantitatively

CCSS.MATH.PRACTICE.MP3 Construct viable arguments and critique the reasoning of others

CCSS.MATH.PRACTICE.MP4 Model with mathematics

CCSS.MATH.PRACTICE.MP5 Use appropriate tools strategically

CCSS.MATH.PRACTICE.MP6 Attend to precision

CCSS.MATH.PRACTICE.MP7 Look for and make use of structure

CCSS.MATH.PRACTICE.MP8 Look for and express regularity in repeated reasoning

Next Generation Science Standards for Engineering Practices

<http://www.nextgenscience.org/sites/ngss/files/Appendix%20F%20%20Science%20and%20Engineering%20Practices%20in%20the%20NGSS%20-%20FINAL%20060513.pdf>

Practice 1 Defining problems

Practice 2 Developing and using models

Practice 3 Planning and carrying out investigations

Practice 4 Analyzing and interpreting data

Practice 5 Using mathematics and computational thinking

Practice 6 Constructing explanations and designing solutions

Practice 7 Engaging in argument from evidence

Practice 8 Obtaining, evaluating, and communicating information

Computer Science Teachers Association K-12 Computer Science Standards

http://csta.acm.org/Curriculum/sub/CurrFiles/CSTA_K-12_CSS.pdf

<http://csta.acm.org/Curriculum/sub/CurrFiles/>

[CSTA Standards Mapped to CC Math Practice StandardsNew.pdf](#)

GUIDE TO THE LESSONS

This curriculum was developed under the Understanding by Design Framework (Wiggins & McTighe 2005), also known as Backward Design. Each lesson was designed to teach one Big Idea as expressed by an explanatory sentence and a short slogan that is easy for students to remember and teachers to evaluate. We have designed six projects that together comprise a survey course of programming fundamentals with an emphasis on transfer goals (skills), and two supplementary lessons that facilitate further synthesis and communication. In addition to teaching computing, this curriculum emphasizes exploratory learning and creative play. And fun. Making something you actually want to use is just as important as learning the vocabulary.

1. Crossy Road

A simple game that introduces events, sequences, and loops, through helping a character navigate across a busy street.

Big Idea: If you can code, you can make things that you like and use, and that may not have existed before. Coding is a superpower!

2. Geometry Dash

A single-button jumping game that focuses on drawing and animation, and increasing the complexity of loops and sequences, including concurrency, so that debugging is required.

Big Idea: Computers do only what you say, because they are not smart enough to figure out what you mean. Be specific!

3. Which Emoji are You?

A customizable quiz that keeps track of your answers and computes a score or outcome using variables and conditionals.

Big Idea: If you know how to use individual blocks like conditionals and variables, you can put them together in powerful ways to build what you want. Little blocks build big programs!

4. Flappy Bird

An exercise in reverse engineering, where students are deeply familiar with the goal, and have to work backward to make it happen. Introduces the concept of a physics engine.

Big Idea: Coding means telling computers what to do, in a language they can understand. Computers speak numbers!

5. Subway Surfers

A complex action game that requires multiple components and design decisions, perfect for introducing the paradigm of pair programming.

Big Idea: There is often more than one solution to a problem, and some solutions are better than others. There may be another way!

6. Can you Escape?

An open-ended point-and-click adventure that connects the ideas of programming logic to real-world logic.

Big Idea: The way to write good programs is to have ideas and make mistakes, over and over. This process is called iteration. Stick to it!

GUIDE TO THE LESSONS

7. Game Design Workshop (Optional)

An opportunity to refine one of the games in lessons 1-6, or start over from scratch with an original idea. Watch Dr. Em's advice on making games at <http://hop.sc/1MwRIID>

8. Game Showcase (Optional)

Share your games in a showcase with others, make a webpage or ad for your game, or write a review of someone else's game. An opportunity to practice sharing and attribution, communication and using appropriate vocabulary, and evaluating the work of others.

The following describes and depicts the format of the lessons and how you might use them in your classroom.

TEACHER BRIEF

At the start of every lesson, there is a Teacher Brief that offers a very high-level summary of the game students will build, goals of the lesson, and concepts covered. Within each lesson there are several mini-lessons that break down the problem of building the complete game into discrete stages.

LESSON

0. Discussion pre-lesson

The first mini-lesson offers prompts for you to set the stage before students begin coding, including discussing the game and the core coding concepts introduced in the lesson. We also recommend showing the students a completed version of the game during this time (or, even better, having them play it!).

1. Mini-lesson overview

Subsequent mini-lessons start with an overview of the game development task to be completed. In this discussion, we define any core coding concepts or vocabulary that are introduced in the mini-lesson and also offer suggestions of ways you can teach them to your class. We recommend that you use the start of each mini-lesson as a way to bring the class back together for instruction between coding sessions.

1.1 Discussion

As students start each stage of building their games, have them discuss the problem they're solving as a class (e.g., "In this stage, we need to add buttons that will let the

player control the character’s movement”). You can ask students to consider potential solutions as a class, in small groups, or on their own. Pseudocoding, or writing out the code on the board or on paper, can be a helpful part of this discussion. Share out potential student solutions and evaluate them as a class. As appropriate, guide them towards a solution.

1.2 Implementation

After a discussion of what needs to be built and, if desired, how it might be coded, students can start coding. Depending on how many iPads you have, you can have students work independently or in pairs (see Page 60 for a discussion of pair programming and why we love it). Students should get into the habit of testing their code frequently by running (playing) it. We recommend that they run their code at every stage of the mini-lesson. It is much easier to find and solve mistakes when you’re constantly testing.

1.3 Using our screenshots

We demonstrate how each task can be implemented in Hopscotch with screenshots of sample code. The code we suggest usually is only one way to build the needed feature; there are often other ways that students can accomplish their goals.

Where appropriate, there are notes that describe the screenshot or functionality depicted.

1.4 Videos

There is a video that accompanies students through the process of making each game. You can use the videos in several ways: as the primary method of instruction by showing them to the class, as a supplement to your instruction, or just as a means to get prepared before teaching. Videos are linked in the materials section of each lesson.

DIFFERENTIATION

(15 minutes, optional)

The format of how you teach each lesson will ultimately be determined by the composition of your class; depending on your students' ages and experience levels, you might want to spend more or less time on discussion, pair-programming, or working independently.

For example, with older or more advanced students, you might always give them an opportunity to code their solutions to the problem on their own or in pairs. For younger or less experienced students, you might always want to give step-by-step directions. You can also give students more freedom as the lesson goes on, or conversely, bring students together to solve harder problems.

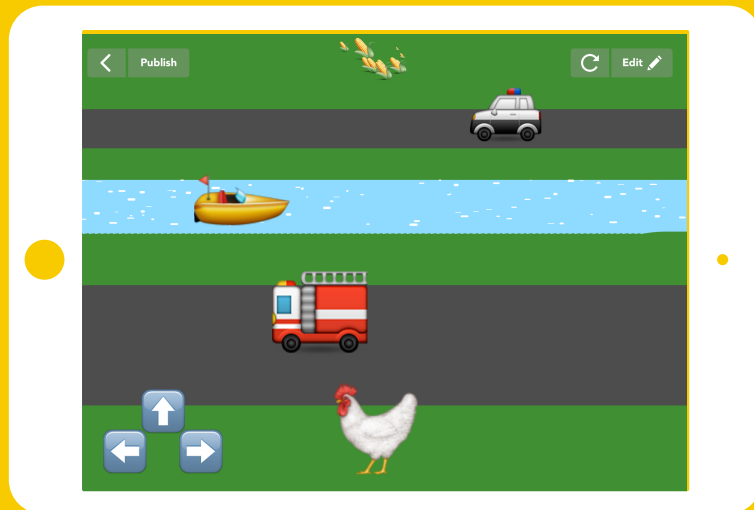
REFLECTION

(15 minutes, optional)

At the end of the lesson, there are suggestions of ways to make the lesson easier or harder, as well as reflection questions.

LESSON 1

CROSSY ROAD



A simple game that touches on each of the core coding concepts and allows students to become familiar with using Hopscotch to build apps and share with others.

TIME

45 minutes, or 60 if you include
15 minutes of free code time

BIG IDEA

If you can code, you can make things
that you like and use, and that may not
have existed before. Coding is a
superpower!

SKILL FOCUS

- Using Hopscotch
- CCSS.MATH.PRACTICE.MP5 Use appropriate tools strategically

KEY VOCABULARY

Event: When something happens

Sequence: A list of instructions, in order

Loop: Code that repeats

Random: The lack of a pattern

Range: The lowest and highest numbers that
Random can choose from

TRANSFER GOALS

1. Students will understand that coding means telling computers what to do, and can think of some things that are made with code (Apps, car software, medical equipment).
2. Students will be familiar with how to use the Hopscotch app to create projects, add objects, and write and edit code.
3. Students will be able to name two Hopscotch Events and understand that an Event is "when something happens".
4. Students will understand that a loop is code that repeats, and be able to see loops in their daily life.

MATERIALS

- 1 iPad or iPhone per student, or 1 device per 2 students, for pair programming
- Video available on YouTube:
<http://hop.sc/CrossyRoadVideo>
- Complete project available:
<http://hop.sc/crossyroadproject>

This first lesson prioritizes familiarizing students with Hopscotch as a tool and the exciting idea that they can control their computer. Students will understand what it takes to make a video game, see the results of their work quickly, and feel like a programmer.

We recommend starting with a short discussion of what coding means and an exploration of some of the things we're going to make over the course of this curriculum. If you have a projector, you can show the students examples of some of the finished games. In this lesson, students will build their own version of the popular Crossy Road game in which a character dodges obstacles.

You may want to break this lesson up over two days, because while making this first game, you will be introducing three core coding concepts and will want to spend sufficient time in discussion of these ideas. These concepts are **Events, Sequences, and Loops**.

Don't worry about all your students completely grasping these ideas the first time; ideas will be reinforced in subsequent lessons.

If you choose to spend two days on this lesson, we recommend ending the first lesson after adding a car obstacle, but before discussing randomness.

LESSON

0. Discussion (5 minutes)

The first and most important lesson of computer science is that computers do what they are told, and only what they are told, in the order they are told to do it.

If you fully understand this concept and begin to think of everyday processes (making a sandwich, getting to school) as a set of instructions, you will begin to think like a programmer without trying very hard! A programmer is a person who codes, or writes computer programs. A program is a set of instructions a computer can understand. We refer to these instructions as a **sequence**. This term also refers to the idea that computers must follow the instructions in the order, or in the "sequence", in which they're given.

Ask your students to name some programs they use. Consider all their games and apps, but also the software a DJ uses to mix tracks, the database your doctor uses to keep track of your health, and the video games you play after school. All are programs and all were created by programmers.

How many times a day do you interact with computers? Are there computers in surprising places? How about a car? How about a phone? If you can control these computers and write programs for them, you can make things that millions of people use every day!

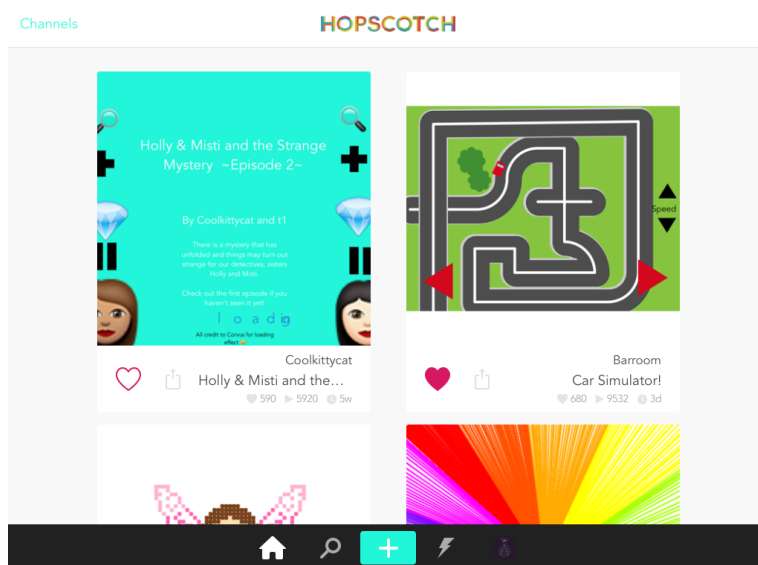
1. Using Hopscotch (5 minutes)

First, get your students acquainted with Hopscotch.

1.1 Finding the Hopscotch app on your iPad

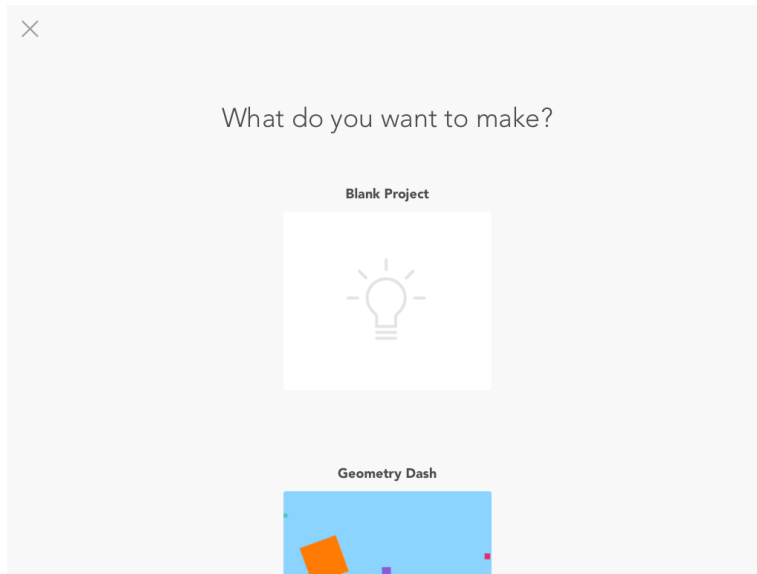
1.2 Signing into your account (students may need to create accounts)

1.3 Making a new project: Tap on the highlighted + on the bottom of the screen



LESSON

1.4 Choose Blank Project



2. Control Pad (E) (10 minutes)

The point of Crossy Road is to navigate a character across a field filled with obstacles (in this case, cars!). The player will use control buttons to direct their character. This control pad is one of the most universally recognizable video game elements.

One of the most important lessons of this activity is learning that the programmer must not only put together all the components of the game (buttons, background, character), but also explicitly tell the computer how they should work. For this, we need to create a **rule**, or code that tells the computer what to do and when to do it. A rule has two components: an **event** and **commands (or action)**.

An **event** is a trigger that the computer recognizes and causes it to do some action. In Hopscotch, all events start with the word "When" and are the first thing you choose when you write a rule. Think of it as completing a "WHEN....., THEN....." sentence.

Events are deeply important for computer engineers because they tell the computer when it should do something. When you touch the phone icon on your home screen, then your phone brings up the interface to make calls. When an Angry Bird hits a block, then the block falls down.

Discuss some events (triggers) that happen in the classroom. Identify the trigger and resulting action: When I raise my hand (trigger), then stop talking (action), when the bell rings (trigger), then put down your pencil and turn in your test (action).

After a general discussion of rules and events, you can transition to talking about programming Crossy Road.

LESSON

In Crossy Road, when the up button is tapped (trigger), we want the hero to move up (action). Pose this challenge to your students and as a class discuss the steps the computer must take to complete it. Once the class agrees on what should happen, you can encourage them to begin working on their own control pads. You can have them do this as a class in several small groups, in pairs, or on their own.

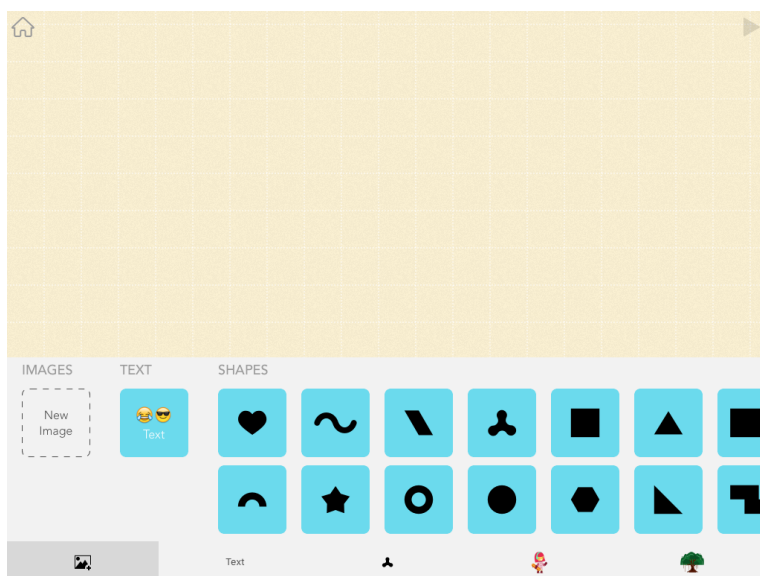
Students should add the buttons that will be used as a control pad (right, left, and up) and a protagonist or “hero” that the buttons will move around the screen. In Hopscotch, we program objects or characters. They can be found by tapping on the “+” button on the bottom of the screen. You can also add a new character from the code editor by tapping “+ New object”. Buttons can be implemented by using a text object and then typing in a block from the emoji keyboard.

For each button they want to include in the game, they will need to add a rule associated with it. They can do this by tapping the character that will be affected by the buttons (the hero) and giving it new rules. Encourage students to explore the events (When) menu in Hopscotch by tapping their hero, then “See code”, and then testing out different events from the magenta menu. They can swipe to see events triggered by the iPad (iPad), interactions between characters (collisions), or logic (conditionals).

Challenge students to complete the code for the other two buttons in pairs or small groups. Ask them to consider: What code will they need to add to create a button that moves the character right when the right arrow is tapped and one that moves the character left when the left button is tapped? To which character should these rules be added? (The hero)

The following is sample code.

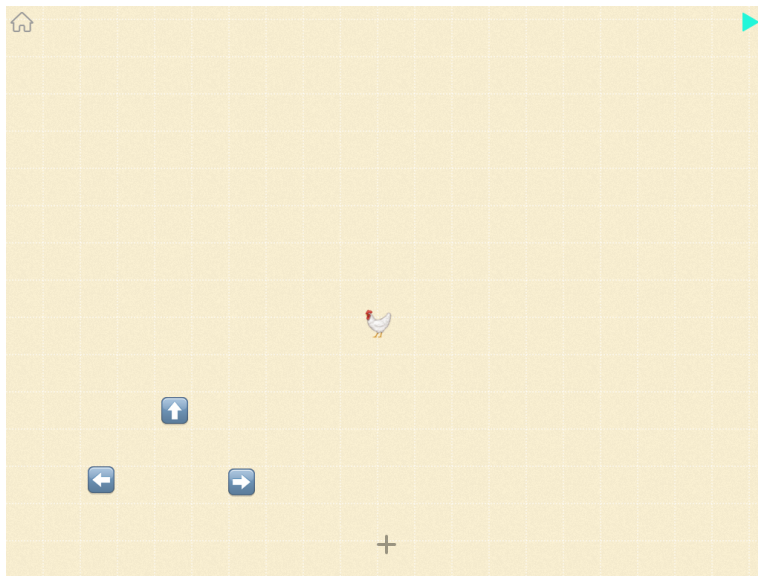
2.1 Add hero object and place at bottom of screen



You can also choose an emoji as your hero by selecting a text object and then choosing from the emoji keyboard.

LESSON

2.2 Add 3 control buttons (up, right, and left) that the player will use to move their character



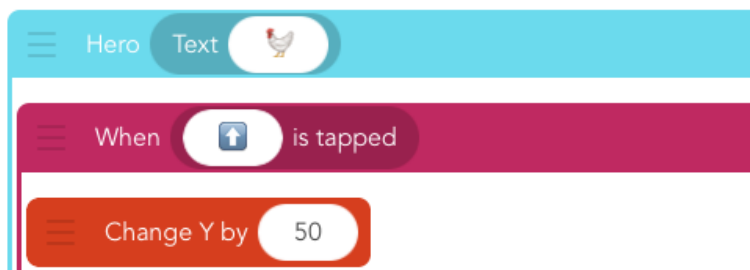
Use three different text objects to create the buttons. You can find the arrow buttons in your emoji keyboard. If you don't have emojis, you can enable them via your iPad settings.

2.3 Name the control buttons "up", "right", and "left"



Tap an arrow button, then the bolded text below it to rename it. Give each arrow button a name that corresponds to the direction it will move the character (up, right, and left). In programming, it is important to name your objects well so that other people can easily read your code in the future.

2.4 Write code to move the hero forward

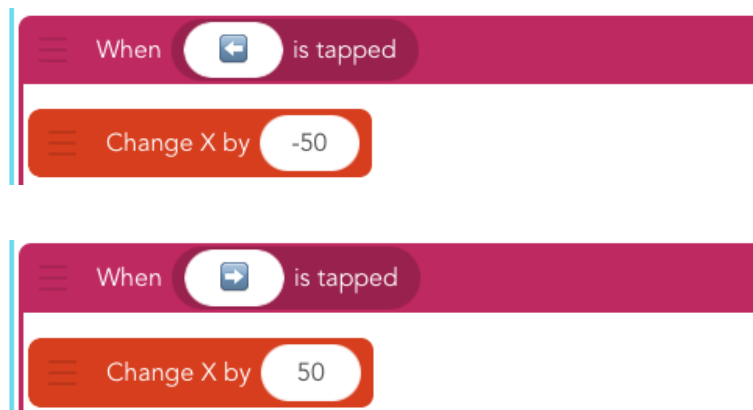


Choose the up button by tapping the iPhone icon in the "When tapped" header.

Review the Coordinate Plane. In Hopscotch, (0,0) is at the bottom left of your iPad screen, so the whole screen is in Quadrant I. Moving up is changing the Y position by a positive amount.

LESSON

2.5 Complete the buttons to move the hero left and right



Moving left is changing the X position by a negative amount, and moving right is changing the X position by a positive amount.

To test out your code, select the play button (turquoise triangle in the upper right corner of your screen).

3. Like a Boss (LS) (10 minutes)

Once the students have their hero working, the next step is to add some drama to the game by introducing a challenge—cars that drive back and forth across the screen indefinitely. These cars will be controlled by the computer (in game design, we call these kinds of automated characters bosses or non-player characters).

This is a good time to discuss **sequence** and **loops**.

Sequence is the order in which instructions are given to the computer. The idea of putting instructions in the correct sequence seems obvious and basic, but it's a vital concept in computer programming.

You can reference a real-life example: making sandwiches for their friends. Ask the class what process they would need to employ in order to make and wrap 10 tuna sandwiches. Does it matter if the process happens in the same order for each sandwich? What if they added mayo after putting canned tuna on bread? Or what if you put the bread in the bag before opening the tuna? Silly, but order matters.

Computers have a finite set of kinds of tasks they can accomplish. But when these tasks are combined properly, amazing things can be built. In addition to running instructions sequentially, computers are very good at repeating sets of instructions. In computer science we call this a **"loop"**, or code that repeats.

Consider using a loop to repeat the sandwich making process: For the number of sandwiches I need: open the tuna, add mayo, stir, put on bread, put in bag.

As a class, discuss the behavior of these cars and together make a list of the steps they take. Ask students to consider the difference between using "Repeat 10 Times" and "Repeat Forever". Which is appropriate for the sandwich? Which is appropriate for the car's movement? Also, consider what happens if instructions are out of order.

LESSON

When students have a hypothesis about how the cars should move, they can begin coding.

3.1 Add car emoji

Name car emoji "Car 1".

3.2 Add new code to car: Make car move back and forth across the screen



(If you decide to break this lesson into two sessions, this would be a good place to stop.)

4. Randomness (5 minutes)

We can make our game more interesting by randomizing the speed of the cars. **Randomness** is a lack of pattern or predictability in events.

The concept of randomness is very important in computer programming because the most useful computer programs must be able to solve generalized (rather than specific) problems. For instance, it is much more useful to write a program that could find the factorial of any random number than a program that could only find the factorial of, say, the number seven. Having one generalized solution that can be used for a variety of specific inputs is at the heart of what makes computer programs powerful. And randomness can be used to test how robust that program is.

Randomness can also be used to make computer programs better. The Roomba vacuum can accomplish its task of cleaning any room anywhere by moving forward until it hits a wall, and then turning in a random direction. Imagine if the people who programmed the Roomba had to write specific directions for it to clean a square room, a rectangular room with two sofas in it, a long and narrow room...you get the idea. It might be more efficient in those specific instances, but they would never be able to account for all the potential rooms the Roomba might have to clean.

Randomness is very useful for programming computer games, because it drives the luck aspects of games. For instance, how often or when a block in Tetris appears is driven by randomness.

LESSON

Randomness depends on giving the computer options to choose from, or a range. You can discuss a real-life example of range. Ask your students to “Pick a number between 1 and 10”. Imagine if you had just told them that you are thinking of a number and asked them to guess it. They would have been guessing for days. Instead, you gave them a range (“1 to 10”), or the lowest and highest number for Random to choose between. We will use randomness to make our games more fun and challenging.

Ask students to consider what would happen if the cars in the game all drove at the same speed. Would the game be fun? What would happen if you randomly set the speed of the cars? Would that make it more fun? (We think so!)

To set the car’s speed, you need to determine its range. Ask your students to play around with the range and see what happens. What if you try (1,10)? What if you try (100,1000)? The default speed in Hopscotch is 400, so a range of (200,600) is pretty good.

4.1 Edit car’s code: Set the speed to random each time



5. Collisions (E) (10 minutes)

The last obligatory element in Crossy Road is to establish collisions and then add more cars. A collision is a type of **event**, and in Hopscotch, it is represented as “When __ bumps __”. When the hero bumps into a car, the hero should disappear.

To finish the game, we need to add at least one other car to make it fun.

Students will need to add the collision rule to their hero, and then add and program more cars. Allow a set amount of time for this activity. In that time, some students will be able to add multiple cars and program their movement and collisions (using the same code as the first car). Others will achieve only one. As a class, discuss collisions and depending on the age of your students, see if they can implement the code on their own. Circulate and help the students who are struggling. This process is repetitive, but offers good practice and gives

LESSON

students a chance to see how one of the most important programming concepts (writing functions) is useful.

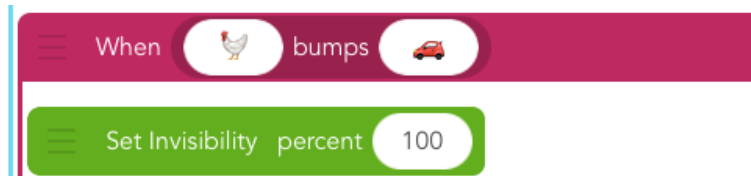
Use the refresh button to start the game over.

5.1 Add code to hero: Disappear when it collides with the car



5.2 Add and program more cars with the rule established above in 4.1

5.3 Add collision code to the hero for each new car



Name each additional car
"Car 2", "Car 3", etc.

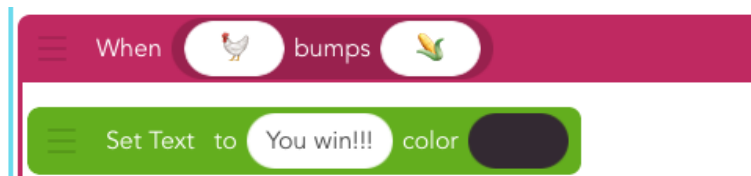
5.4 Test program, adjust position of cars

6. Victory (optional) (E) (5 minutes)

It's not a game if you can't win! Add a goal destination, or target, to give your hero somewhere to go. When the hero bumps the target, the game should say, "You win." We can program a text object to display this message when triggered by the collision.

6.1 Add a target object (corn)

6.2 Add a win message text object, and don't set the text



When you add a text object, if you tap "X" on the upper left corner instead of writing a name for your new text, it starts out invisible.

7. Publishing (5 minutes)

Share what you made with the world! Ask students to publish their programs, giving the game a descriptive name that they'll remember and pinching the image to adjust the screenshot. See if they can find their own and each other's projects in the community.

7.1 Publish your program

DIFFERENTIATION

(15 minutes, optional)

- Put in lots of cars
- Draw lanes
- Set speed
- Customize control pad with better emojis, different sizes, or by moving a different amount
- Animate the “You Win” text and give it a cool color

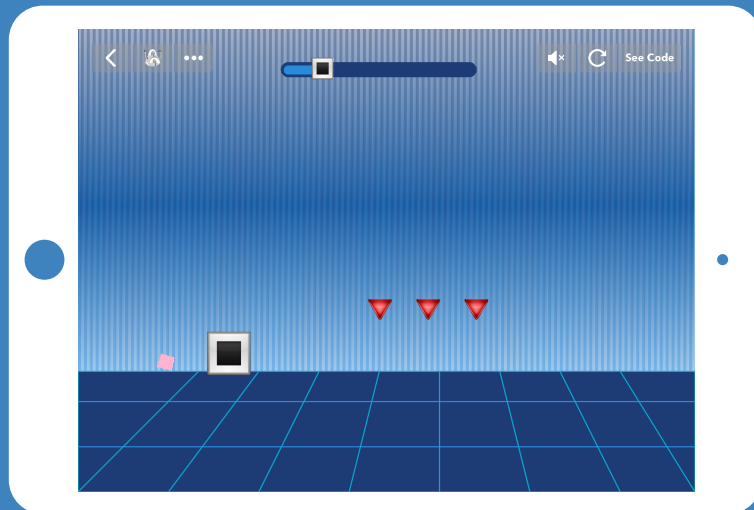
REFLECTION

(5 minutes, optional)

- What is coding? (telling computers what to do)
- What can you make with code? (apps, games, medical software)
- What is an event? (when to do something)
- Can you name some events, in Hopscotch or in real life? (“When the play button is tapped”, “When _ bumps _”)
- What is a collision? (when two things bump into each other)
- What do you think about coding? Is it fun? Hard? Rewarding?

LESSON 2

GEOMETRY DASH



A single-button jumper that includes moving obstacles, drawing a background, and animation

TIME

45-60 minutes (+15 minutes of optional, free code time)

BIG IDEA

Computers can only do what you SAY because they are not smart enough to figure out what you MEAN. Be specific!

SKILL FOCUS

-
- Debugging
 - Make sense of problems and persevere in solving (CCSS.MATH.PRACTICE.MP1)
 - Look for and make use of structure (CCSS.MATH.PRACTICE.MP7)
 - Designing solutions (NGSS Practice 6)

KEY VOCABULARY

Bug: A mistake in your code

Debugging: Finding mistakes and fixing them

Concurrency: Two things that happen at the same time

Random: A surprise

Range: The highest and lowest number for *random* to choose between

TRANSFER GOALS

-
1. Students will become familiar with editing rules
 2. Students will practice testing their programs to find bugs.
 3. Students will practice fixing bugs and verifying that they are fixed.
 4. Students will abstract a problem to design a solution.
 5. Students will develop confidence and persistence.

MATERIALS

-
- 1 iPad or iPhone per student, or 1 device per 2 students, for pair programming
 - Video available on YouTube:
<http://hop.sc/GeometryDashVideo>
 - Complete project available:
<http://hop.sc/geometrydashproject>

As your students learned in Lesson 1, computers are really good at carrying out orders quickly and accurately. They are not good at thinking about what things mean or making decisions for themselves. That means that we have to be very careful when we are giving computers instructions, because they will do exactly what we tell them to do (even if it makes no sense). An inevitable part of programming is introducing mistakes, or bugs, in your code and then having to fix them (debugging).

This second lesson focuses on debugging as a rewarding exercise, and teaches kids to become comfortable making and working through mistakes. Students will get used to testing their programs and editing rules, which will occupy lots of their time for the next four weeks. Finding bugs can be frustrating for even the most seasoned engineer, but the process is ultimately very rewarding and a unique opportunity to learn and practice perseverance—one of the most transferable skills gained through coding. Celebrate bug fixes!

Debugging is made easier by making incremental changes to your code—write one thing, test it, and then write the next. If you code lots of things at once and then figure out it's not working, it's harder to track down which of your changes caused a mistake.

LESSON

0. Discussion: Debugging

In this lesson, students will create their own version of Geometry Dash. While building the game, they will inevitably make mistakes and create bugs. This lesson is equally as much about the process of finding and fixing bugs as it is about making a fun game. You can ask your students to think about this task and imagine themselves as bug hunters.

As programmers, we frequently tell our computers to do something other than what we intended. We call the resulting mistakes bugs, or errors in a program introduced by the person writing it. The process of finding and fixing your mistakes is called **debugging**. One of the most important lessons in coding is remembering that your bugs are not caused by the computer—they're caused by the programmer. And it's totally expected that all programmers will write bugs at different points in the development process.

When real-world programmers are in the process of writing code, the rule of thumb is that it takes 10% of their time to write the first draft, and the other 90% of their time to debug it. There are engineers whose whole jobs are to debug other people's code!

It may be worthwhile at this point to discuss debugging with your class. What are some useful strategies to consider while debugging? The following are just some examples:

- Say what you think your program is supposed to do, see what it actually does, and then describe the difference in your own words.
- Look at your code for ambiguities, or places where your blocks don't say exactly what you want to happen, when you want it to happen.
- Make a checklist of common mistakes: Did you repeat forever? Are the numbers you plugged in correct? Did you use the correct blocks for what you intended? Move Forward vs Change X By? Set Speed vs Set Angle, etc. Does your rule belong to the right object?
- Try to map out the logic of your project. Then see if you've written the right code to create that logic.
- Take a break when you get overwhelmed. We often need distance to see what we've done in its entirety.

What should bug hunters look for? Why is this an important job? When else in our lives have we had to hunt for and solve problems?

1. Control the hero (ES) (10 minutes)

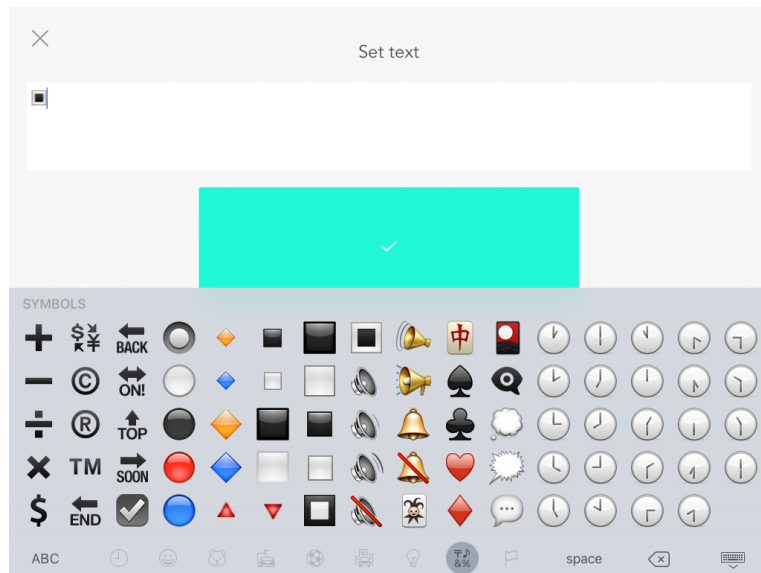
In Geometry Dash, the player controls a little square that flips and jumps over obstacles.

Because the jumping and flipping animations happen at the same time, we say they are **concurrent**. The way to program concurrence in Hopscotch is to make two rules with the same event. That way, they are triggered at the same time.

LESSON

Get students to deconstruct the two steps of jumping (move up, then move down). Does this up and down movement occur along the X or Y axis? Then, ask your students to add their hero object (the square emoji) and tell it to turn and jump when they tap their iPad.

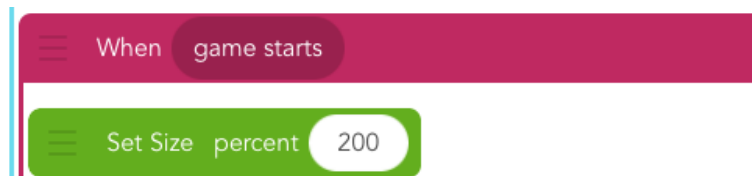
1.1 Add hero object (square emoji) and place it near the bottom left corner of screen



Make sure the emoji keyboard is enabled, which you can do in your iPad's settings.

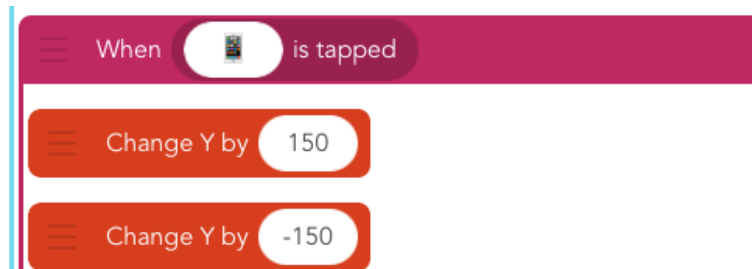
By default, Hopscotch names text objects "Text", "Text 2", etc. Clearly named objects make it easier for you and others to read your code, however, and students should rename each text object according to the role it will serve in the project. Here, rename the square emoji "Hero".

1.2 Add code to hero to make it bigger



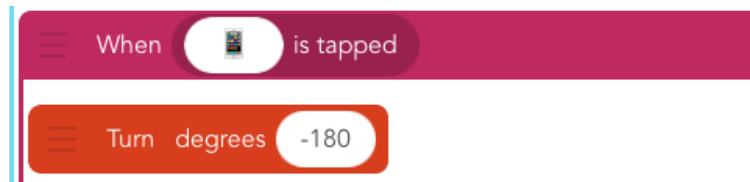
If you choose a number other than 200, all of the other numbers we give will also have to change. This is an opportunity for debugging.

1.3 Add code to hero to jump



LESSON

1.4 Add new code to hero to turn while jumping



What would happen if you picked a non-symmetrical hero? How much would you have to turn it so it landed on its feet? What happens when you choose +180 instead?

2. Background (S) [10 minutes]

Drawing the background is a skill that you can apply to any game. Because drawing is just like any other code, you have to choose an object to be in charge of drawing. It is customary to make this object invisible, so you don't see the thing itself, only the picture it draws. For this reason, it doesn't really matter which object you choose.

In Hopscotch, we draw with a block called "Draw a Trail" that sets the color and width of the line, then executes the code inside – typically "Move Forward" – as if the object were dragging a marker behind it. It will make a dot if it just moves by 1. To color in the whole screen, make a huge dot (width 3000). To make a thick line, you have to set the position to where you want it to start, and then move along the desired path.

This is another opportunity for debugging. Have the students make a prediction about the following questions and then test out changing their code. What happens... if you don't put anything inside the drawing block? ...if you forget to set the width? ...if you set the color to white? ...if you don't set the position before you start?

Then, have students attempt drawing their backgrounds on their own. They can change the artist's speed to draw the background faster.

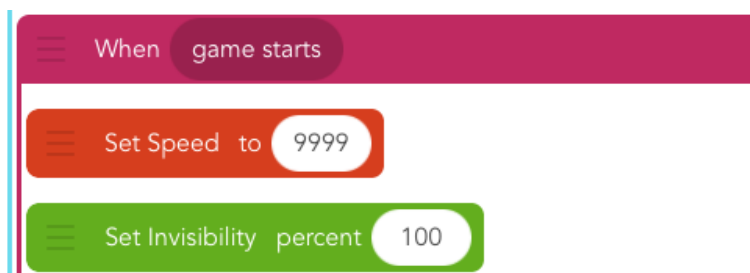
2.1 Add drawing object (choose anything)

2.2 Add code to drawing object



Set the invisibility to 100 so you can't see the painter.

2.3 Edit drawing object's code to draw faster



Change the order of an object's rules by dragging a rule up or down in the editor.

The default speed is 400. 9999 is as high as you ever need to go; that speed is indistinguishable from 999999999...

3. Obstacles (LS) [10 minutes]

In games like Flappy Bird and Geometry Dash, it feels like the hero is moving forward through a stationary world but actually, the hero is stationary and the world is moving backward. Have you ever sat in a stationary car and another car next to you backs up – doesn't it feel, for just a moment, like you're moving forward? In this game, the hero is the car you're in, and the obstacles are the things moving backwards.

Take some time to talk about the movement of the obstacles from one edge of the screen across to the other edge. See if you can come up with the sequence of obstacles' movement rules as a class.

After students agree on the correct code, ask them to try implementing it. Then, bring the class together again and decide as a class at what point the obstacle should be visible and invisible. Discuss why this feels so much more natural (It's because our brains are good at imagining that an object that moves out of our field of view is probably still in motion even though we can't see it).

What if we want to make it look like there are many obstacles but only use one object? This is another great design trick. See if your students can identify the technique to make this possible – putting the code inside a loop.

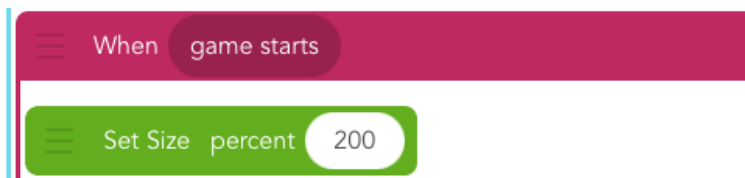
Give the students a few minutes to play their game, and then bring the class together again. Ask for suggestions to make the game more fun and challenging. Like with Crossy Road in Lesson 1, it is boring (and easy!) because it's the same every time! Games are challenging (and fun!) when there is an element of unpredictability. If you make the obstacle wait for a **random** amount of time in between passes, the game becomes more fun.

Debugging opportunity: What is the appropriate range for the random wait time? Try out some different combinations until you settle on one you like.

3.1 Add emoji object for obstacle (triangle)

Rename the triangle emoji "Obstacle".

3.2 Add code to obstacle to make it bigger

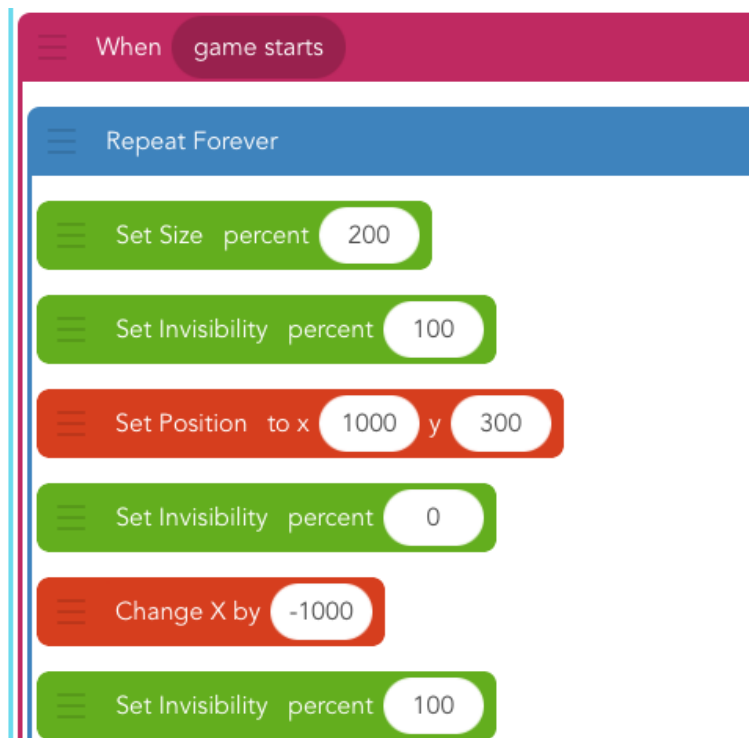


3.3 Edit obstacle's code to move it across the screen



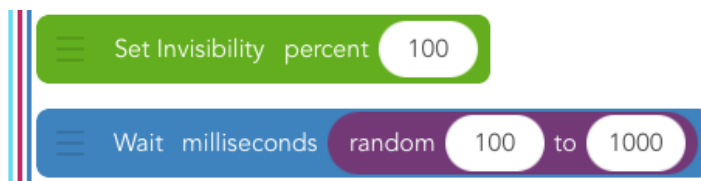
LESSON

3.4 Edit obstacle's code to make sequence repeat forever



When moving code into the repeat block, make sure not to change the order. Students will probably make a mistake here—a good opportunity for debugging!

3.5 Edit obstacle's code to wait random (100,1000)

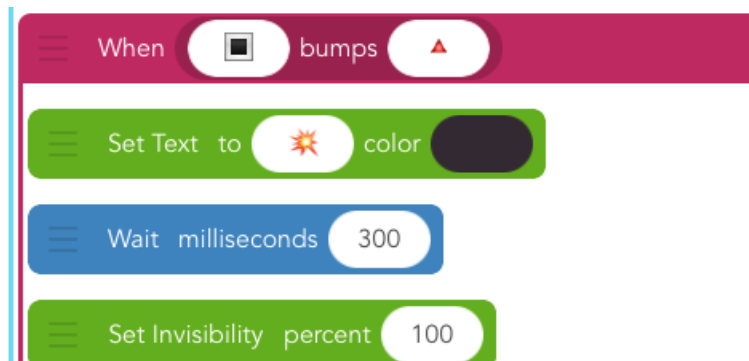


4. Collisions (ES) [10 minutes]

As we learned in Lesson 1, when two objects bump into one another, it is called a **collision**. A collision is a type of **event**, so we can decide what actions should happen when that event occurs. In Geometry Dash, when the hero collides with an obstacle, the game is over.

To designate "game over," upon a collision the hero will explode and then disappear. In Hopscotch, when an object is invisible, it can no longer collide with anything, be tapped, or swiped. Spend some time testing this sequence and getting the timing right, then publish!

4.1 Add new collision rule to hero



You can change the object into an explosion, make it spin around, or drop off the screen like Mario. Turning invisible is necessary, because it stops the game from being playable.

4.2 Publish your game

DIFFERENTIATION

(15 minutes, optional)

- Draw a better background
- Make the background colors random
- Add more obstacles (two or three emojis in a row is a possibility, make movement into an ability)
- Set the obstacle size to random each time; pick a good range!
- Print and laminate index cards with debugging strategies and have students check off strategies as they go

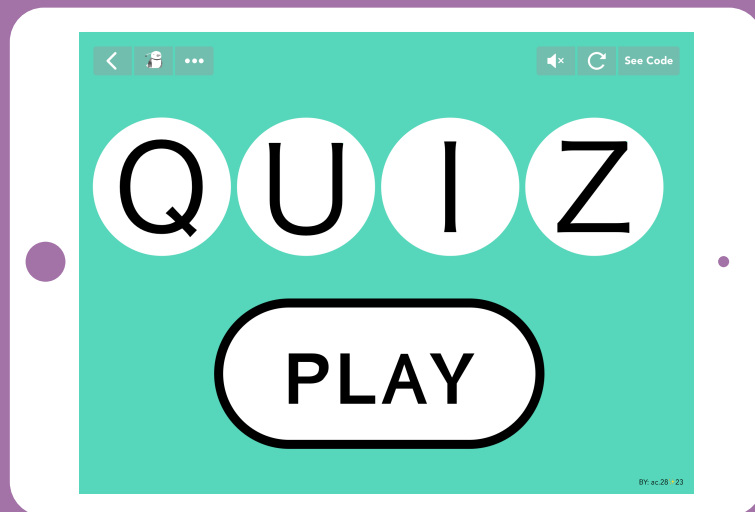
REFLECTION

(5 minutes, optional)

- What are computers good at? What are they bad at?
- How does this compare to what humans are good and bad at?
- Is drawing with a computer easier or harder than drawing with pencil and paper? Why? If it is harder, why do we still do it?

LESSON 3

WHICH EMOJI ARE YOU?



A personalized quiz that keeps track of your answers and calculates results.

TIME

45-60 minutes (+15 minutes of optional, free code time)

BIG IDEA

If you know how to use individual blocks like conditionals and variables, you can put them together in powerful ways to make what you want. Little blocks build big programs!

SKILL FOCUS

- Planning and Execution
- NGSS Practice 3 Planning and carrying out investigations
- NGSS Practice 4 Analyzing and interpreting data
- CCSS.MATH.PRACTICE.MP2 Reason abstractly and quantitatively
- CCSS.MATH.PRACTICE.MP8 Look for and express regularity in repeated reasoning

KEY VOCABULARY

Value/Variable: A number that can change

Conditional: Statement of the form "IF (something is true), THEN (do an action)"

TRANSFER GOALS

1. Students will understand that a conditional checks if something is true and can identify the use of conditionals in their lives.
2. Students will understand that a value or variable is a holder for a number and can identify the use of values in their lives.
3. Students will become comfortable with keeping track of variables and doing operations.
4. Students will make a plan and see it through until the end.
5. Students will recognize patterns in their code and apply them to new situations.

MATERIALS

- 1 iPad or iPhone per student, or 1 device per 2 students, for pair programming
- Video available on YouTube:
<http://hop.sc/EmojiQuizVideo>
- Complete project available:
<http://hop.sc/emojiquizproject>

In this lesson, students will create an if-you-chose-mostly-A's type quiz, like you find in the back of a fashion magazine. In our example, it will help them answer an age-old question: "Which emoji are you?". With a little adjustment, it could be made into a factual quiz that checks the answers for correctness. It is very different from the last two games as it contains only text objects and revolves around logic and order, rather than animation and collisions. This lesson introduces two important computing concepts: **values and conditional statements**.

In this lesson, the example quiz content is "Which emoji are you?". Students can stay close to the example quiz by choosing different emojis and questions. This is also an opportunity to tie in other content they are learning.




LESSON

0. Make a plan

The first step is to design the questions and answers for your quiz. Pass out copies of the blank matrix below, or put it on the projector and have students create their own on paper. Don't worry too much about making them the best questions ever; this is just your first quiz!

	Question 1	Question 2	Question 3
Result A	Answer 1A	Answer 2A	Answer 3A
Result B	Answer 1B	Answer 2B	Answer 3B
Result C	Answer 1C	Answer 2C	Answer 3C

Some ideas include: What job should you have? What fictional character are you most like? What book should you read next? If time is limited, or if a student has a hard time coming up with an idea, use the sample matrix below:

	What are you best at?	Where do you like to hang out?	Would you move to Mars if you could?
	Playing music	Library	No.
	Playing sports	Park	Yes!
	Telling jokes	Pool	Depends, do they have ice cream?

1. Setup and introduction to values (V)

Before coding, discuss how the quiz will work and the tools we will use to build it. The quiz has three main functions: First, it keeps track of what question you're on and advances the question number when the player chooses an answer. Second, it keeps track of how many of each answer you've chosen. Third, it compares the answers at the end to give the player a result. We will use **values** to keep track of all this information.

Values, also known as **variables**, hold pieces of information. Values can be set, changed, or checked. They can be used inside **events** and other blocks, and can stand in any place you can use a number. What makes values so powerful is that you can actually change them programmatically. For example, think of your score in a game like Angry Birds or the number of unread emails in your inbox. Both of these numbers are actually values. As you score more points in Angry Birds your score goes up; the value changes. As you read your email the number of unread emails you have goes down. Values are used to represent some kind of information.

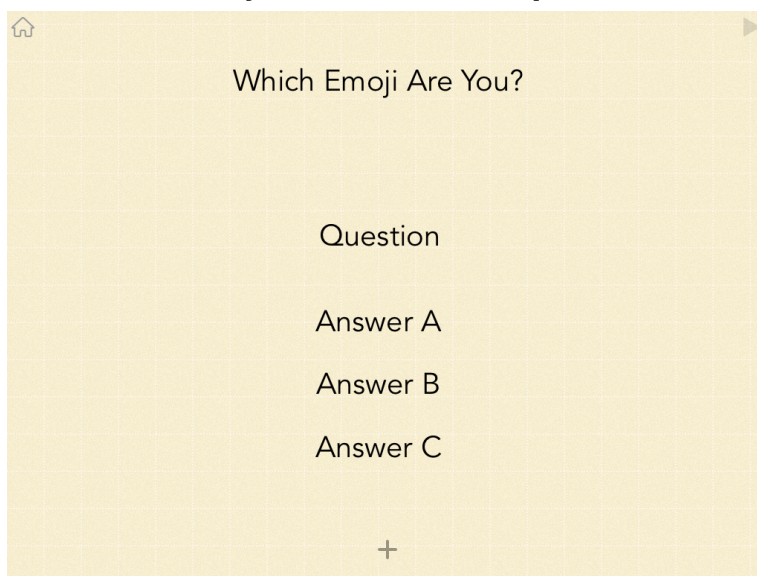
In Hopscotch, the yellow Values tab is to the right of the calculator tab that pops up when it's time to input a number. In addition to the built-in values like "iPad's width" or "Character's X position", you can add new values to your project to keep track of other things like a score or a state. It might be helpful to walk your students through the Values menu and Value blocks.

LESSON

It is super important to name your values well! You'll need to be able to tell them apart later and remember what they're for! If you need to rename or delete a value, press and hold down on it.

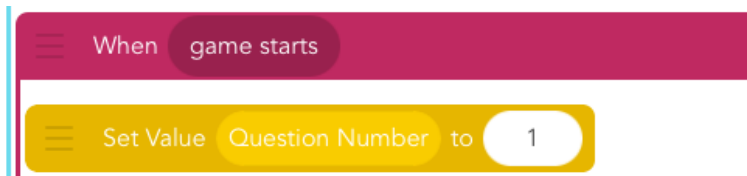
After discussing the concept of values with your students, you might want to go through a list of objects that you will need and their respective initial rules (five text objects: a title, a question, and three answers.) The title (or any object, really) should initialize a value that will keep track of what question the player is on. In the example, we call it "Question Number". We create separate values that will keep track of each answer category. We initialize "Question Number" to 1 and the three "Answer" values to 0.

1.1 Add text objects for the title, question, and three answers (5 objects)



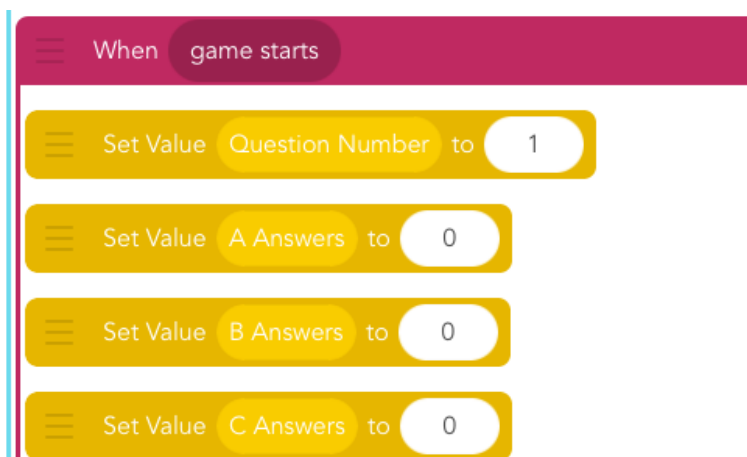
Name each text object according to the role it will serve in the project, e.g. "Title" or "Answer A/B/C".

1.2 Add new code to the Title object: Initialize Question Number to 1



Select "New Value" under the iPad icon and name your new value "Question Number".

1.3 Edit the Title object's code: Initialize all three answer values to 0



Create a new value for each Answer (e.g. A Answers, B Answers, C Answers.)

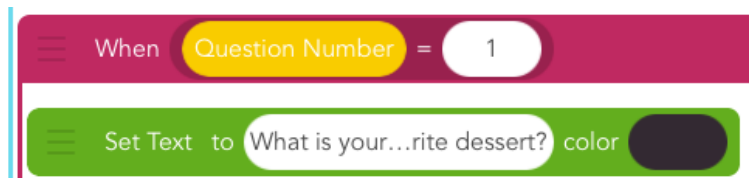
LESSON

2. Value Events (EV)

The four events at the very bottom of the Events menu deal with comparing values. They track whether a value is equal, not equal, bigger than, or smaller than another value or number. When you select the “_ equals _” event, the calculator menu opens. On the right of “Calculator” are values.

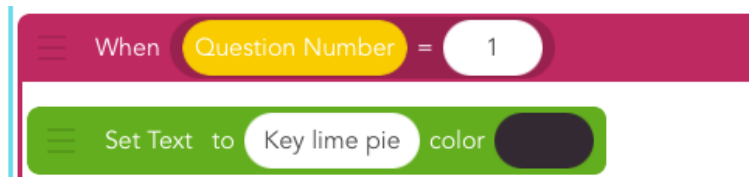
The quiz should display the first question and its answers when the “Question Number” value equals 1. The player chooses an answer and the game should keep track of how many of each kind of answer is chosen. We do that by increasing the appropriate answer value when each answer is tapped. For example, if an “A Answer” is chosen, the value “Answer A” should increase. Show the students these new events, and then discuss the rules that they will need to code before having them start programming. Make sure they choose the right object-value combination!

2.1 Add new code to the question object



Type the first question in the “Set Text” window.

2.2 Add new code to each of the answer objects



Refer to your question matrix to enter each answer in its respective rule. We will do questions 2 and 3 later.

2.3 Add new code to each of the answer objects: Increment the right score when tapped



Tap the iPhone icon in the rule's header to select the corresponding text object.

LESSON

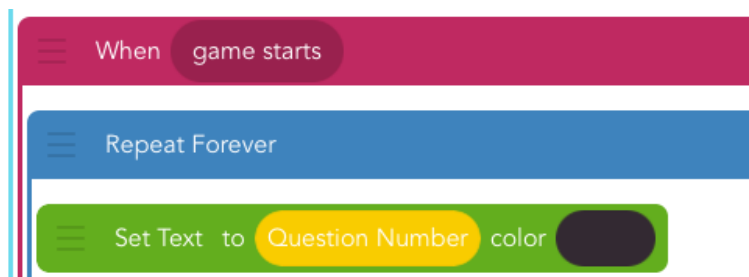
3. Show Values (LV)

It's hard to know whether our values are working properly, so programmers design tests to display them. In Hopscotch, you can do this by displaying the value on the screen. This is an advanced debugging strategy, and the values should be deleted from the finished version of the project once you know everything works. Discuss this technique with students and see if they can figure out how to display a value when their project is played.

We recommend using "Set Text" and putting the appropriate value inside a forever loop to display it when the project is played. You should see the values change as the player progresses through the quiz.

3.1 Add four new blank text objects. Place them next to the question and each of the answers.

3.2 Add new code to the question test object

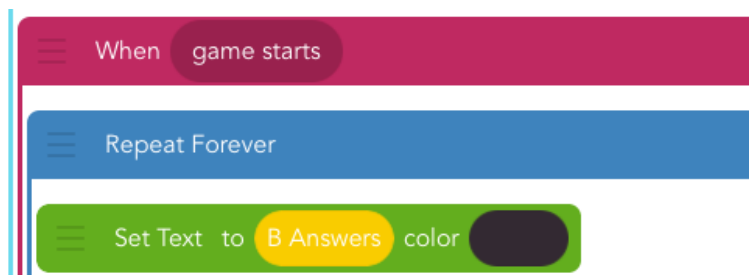


To set text to the value Question Number, tap the three gray bars above "Q" on the keyboard.

3.3 Add new code to Answer A test object



3.4 Add new code to Answer B test object



LESSON

3.5 Add new code to Answer C object

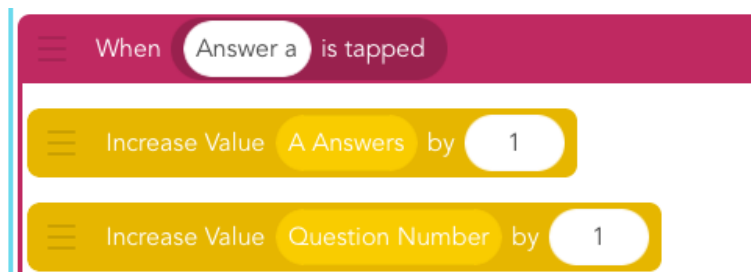


4. Next Question (EV)

The score for each of the answers advances as you tap them, but the question does not change (neither does the "Question Number" value). Decide as a class, what event(s) should trigger the quiz to move on to the next questions. If we make "Question Number" increase, we also need to make the question (and answers) display the correct text for when the "Question Number" value equals 2 or 3. How should this work?

Hint: You advance "Question Number" when any of the answers is tapped! We have already made rules for this event, so we just need to add a block to increase the "Question Number" value. You can do this step as a class and then together or in pairs test your programs to make sure the "Question Number" is advancing. Allow your students to implement the answer code in their own time and to help their neighbors if they finish early.

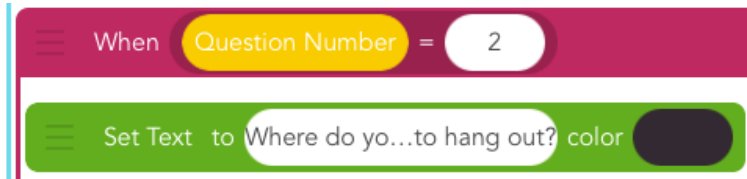
4.1 Edit code for all 3 answer objects to change "Question Number"



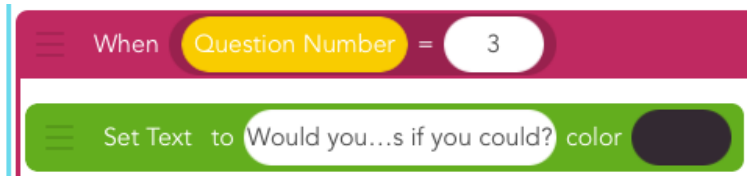
Add the Increase Value block to Answers B and C.

LESSON

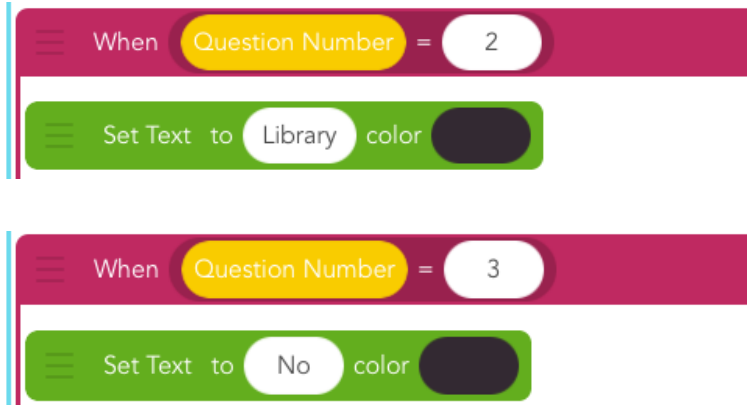
4.2 Add new code to Question object to display second question



4.3 Add new code to Question object to display third question



4.4 Add new code to each Answer object to display second and third answers



For Answer objects A, B, and C, add these same two rules with their respective answers. Refer to your matrix for the text of the answers. In this step, you're creating a total of 6 new rules!

5. Results and introduction to Conditionals (VC)

Now that all the questions are answered, we have to check our results! The correct result to show is the one for which the "Answers value" is the greatest. We can use **conditionals** to compare values.

Conditionals are statements of the form "IF (something is true) THEN (do an action)". It executes code only under the *condition* that you specify, like IF the score is greater than 10 or IF a character is invisible. We use conditionals in our lives all the time. Stoplights are a great example of conditions: "If the light is green, then go." They are also extremely important in programming since computers need explicit directions.

There are two types of conditionals in Hopscotch: "Check Once If" and "Check If... Else".

LESSON

“Check Once If” is used when there are only two possibilities: 1. If the condition is true, do something, and 2. If the condition is not true, move on. “Check Once...Else”, by contrast, lets you check two things before moving on: 1. If something is true, do something, and 2. If this thing is NOT true, do something else, then move on. For example, say you can go in the ocean only if you’re wearing sunscreen or else you will have to put some on first. If you are wearing sunscreen (the condition is true), then you may go into the ocean. If you are not wearing sunscreen (the condition is false), then you must put some on first.

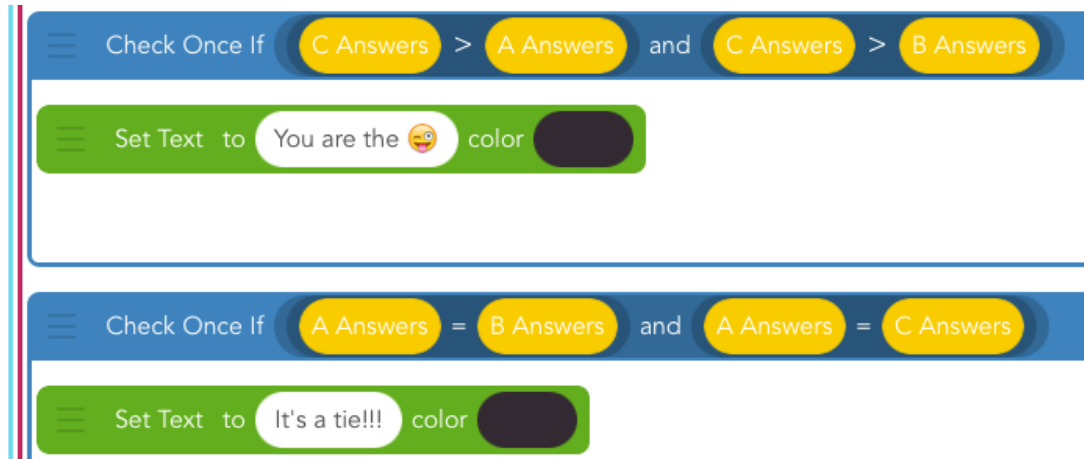
There are several ways to determine the answer to the quiz. The safest way is the one that won’t have to be changed if more questions are added: A is the winner if the “A answers” value is greater than *both* the “B answers” and the “C answers” values (and likewise for B and C.) In Hopscotch, we check if two things are true at the same time by using the “and” expression under Conditionals after choosing Check Once If.

5.1 Add new code to the Title object

The image shows a sequence of code blocks in the Hopscotch programming environment. The first block is a 'When' block triggered by 'Question Number' being greater than 3. It contains three 'Check Once If' blocks, each followed by a 'Set Text' block. The first 'Check Once If' block checks if 'A Answers' is greater than 'B Answers' and 'A Answers' is greater than 'C Answers'. If true, the 'Set Text' block sets the text to 'You are the 🤩 color' with a color picker. The second 'Check Once If' block checks if 'B Answers' is greater than 'A Answers' and 'B Answers' is greater than 'C Answers'. If true, the 'Set Text' block sets the text to 'You are the 😍 color' with a color picker. The third 'Check Once If' block checks if 'C Answers' is greater than 'A Answers' and 'C Answers' is greater than 'B Answers'. If true, the 'Set Text' block sets the text to 'You are the 🤩 color' with a color picker.

LESSON

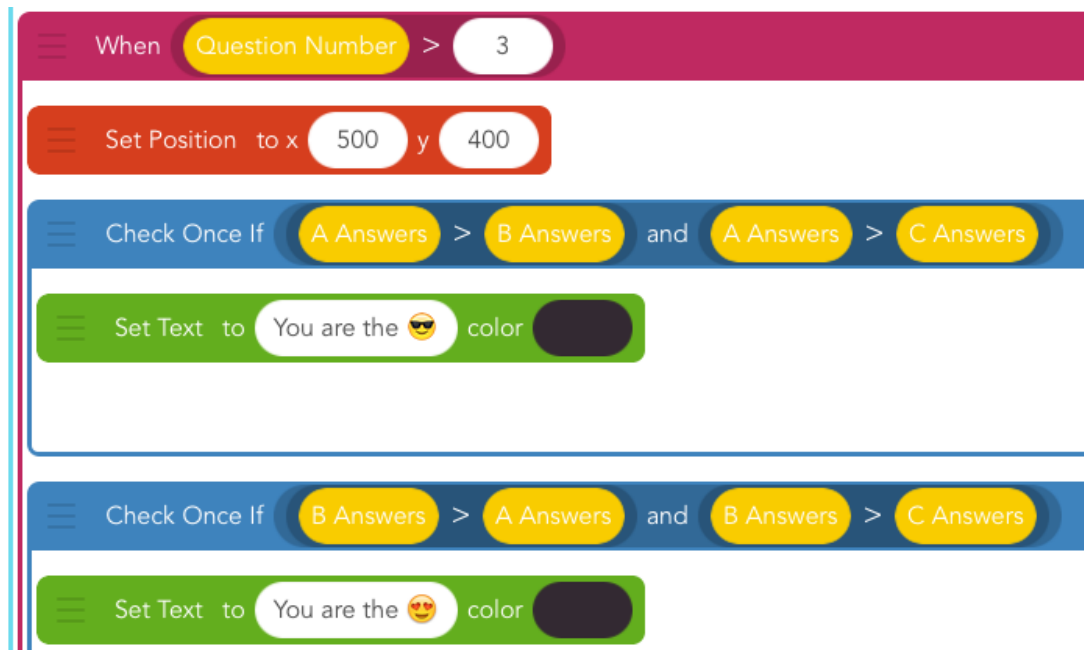
5.2 Edit the Title object's code in case there's a tie (optional)



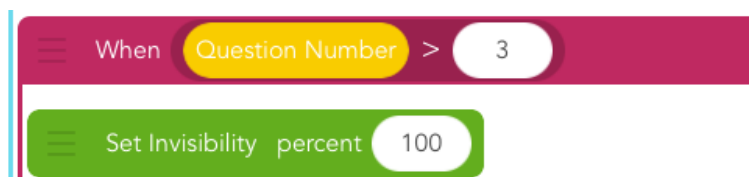
6. Tidying up (ES)

The hard part is done, and there's just a bit more work to do to make the ending look snazzy. One option is to center the results and hide the question and answers at the end. Your students may come up with better ideas!

6.1 Edit Title's code



6.2 Add new code to the question and all three answers



Don't forget to delete your test objects!

DIFFERENTIATION

(15 minutes, optional)

- How would you change the quiz to have more answers or be longer?
- How could you change it to be a factual quiz—to change up whether the correct answer is A, B, or C?
- Turn the quiz into a survey that keeps track of how lots of people have answered—use this data to make a histogram.

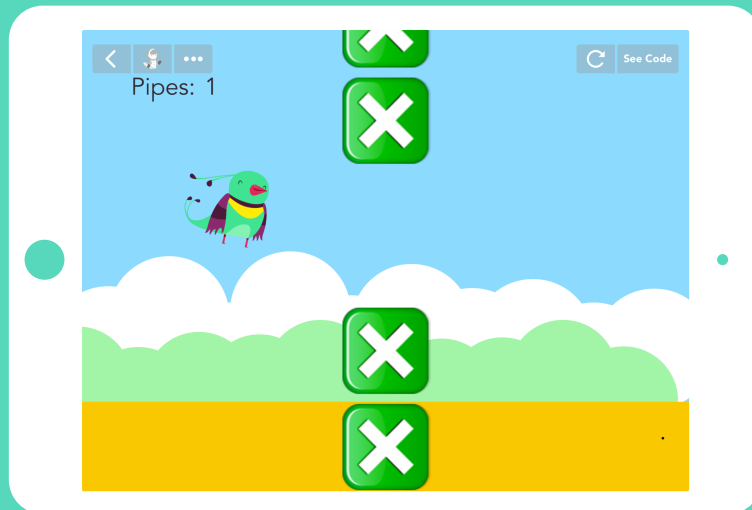
REFLECTION

(5 minutes, optional)

- What is a value?
- What is a conditional?
- How are values and conditionals related or used together?
- How are events related to conditionals? How do you make a custom event? (When play button is tapped, repeat forever, check once if ...)
- What happens if there's a tie? How do we use conditionals to detect a tie? (if $a=b=c$)

LESSON 4

FLAPPY BIRD



An exercise in reverse-engineering, both of a known game and of the physical rules of the real world.

TIME

45-60 minutes (+15 minutes of optional, free code time)

BIG IDEA

Coding means telling computers what to do, in a language they can understand. Computers speak numbers!

SKILL FOCUS

- Reverse Engineering
- CCSS.MATH.PRACTICE.MP4 Model with mathematics.
- NGSS Practice 1 Defining problems
- NGSS Practice 2 Developing and using models

KEY VOCABULARY

Physics Engine: The common set of rules that the objects in the game follow to make the world feel “real”

Reverse Engineering: Examining an existing program or machine and figuring out how it works so that we can reproduce it

TRANSFER GOALS

1. Students will understand that math is an important part of coding.
2. Students will anticipate how changing code will change behavior.
3. Students will be able to test different settings and choose the appropriate one.
4. Students will begin to recognize rules in the world, both those that are constructed (like games) or obligatory (like physics).

MATERIALS

– 1 iPad or iPhone per student, or 1 device per 2 students, for pair programming– Video available on YouTube:

<http://hop.sc/FlappyBirdVideo>

– Complete project available:

<http://hop.sc/flappyproject>

Math is the language scientists use to express the rules of the physical world. From counting tree rings to $E=mc^2$, we use numbers and equations to be specific about how real things move and change. When programmers design virtual worlds, we use math to tell the computer how that virtual world should work. In this lesson, we will make Flappy Bird, and in the process students will create a simple physics engine, using values to create an environment that simulates, or models, the physics of the real world. For those who aren't "math people", don't fret! Most programming math is elementary-level.

Students will also explore the concept of reverse engineering, a process of examining an existing program or machine and figuring out how it works in order to reproduce it. Students should draw from their familiarity with the game or watch it in action before building, and then work backwards to determine the physical elements necessary for the game.

We also introduce a new concept, functions (or abilities, as we call them in Hopscotch) in this lesson.

LESSON

0. Discussion (5 minutes)

Any game that models real-world physics, including falling and gravity, skidding, projectiles, even water, uses what we call a **Physics Engine**. This is the common set of rules that the objects in the game follow to make the world feel “real”. Making a good physics engine takes a lot of trial and error, because the numbers need to be just right to feel real.

What are the physical elements of Flappy Bird? Watch a game of Flappy Bird as a class, and make a list of the objects in the game and the rules that they are following. Write this list on the board and use it as a guide to making the game. The order we give here is just an efficient example, and your class may be better served by a different order:

- First, the bird falls to the ground if left alone.
- Second, the bird flaps when you tap the iPad (thus staying afloat).
- Third, obstacles enter on the right of the screen and travel across to the left.
- Fourth, the game ends when the bird collides with any of the obstacles or the ground.

1. Fall all the time + introduction of physics engine (LV) (10 minutes)

We recommend that you discuss and build this first rule together as a class. The core part of your Flappy Bird physics engine is gravity. The bird should fall when it is left alone. But, it doesn’t just move down the screen at a constant rate: it speeds up as it falls, just like objects fall in real life! We implement this by making the bird move by a value, called “Bird UpDown” in the below code, instead of by a number. That way, we can change the value over time, and the amount the bird moves will update respectively. This will give the appearance of a bird falling at an increasing speed.

Writing this rule is a good opportunity to use a **function** to organize the code for the physics engine. A function is a way to save code so you can reuse it somewhere else. This is a super useful trick that will allow your students to create complex games and programs because they won’t have to write the same code over and over. There is a saying in programming—“Don’t repeat yourself”—which means you should only write something once. With functions, that’s possible.

Functions also make your code easier to understand. If anyone wants to look at the code later, they will understand that it, taken together, builds an engine to make the bird fall.

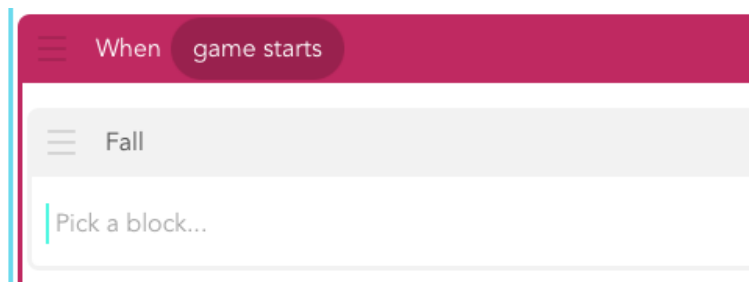
Ask your students why are we using values to change the speed. What is the difference between the “Set Value” and “Increase Value” blocks? How is it like the difference between “Set Angle” and “Turn”? Why are we using a function to group our code? Walk your students through adding the function and then writing the code to make the bird fall at an increasing speed forever. Or, you can have students try using pseudocode to write the rules on their own and then share their ideas with the class. Check out the sample code to see one possible implementation.

1.1 Add bird object

Set bird object to the far left of the screen.

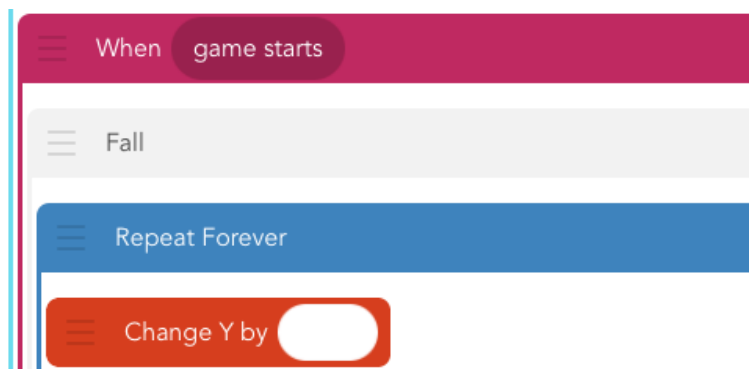
LESSON

1.2 Add code to bird: create "Fall" function



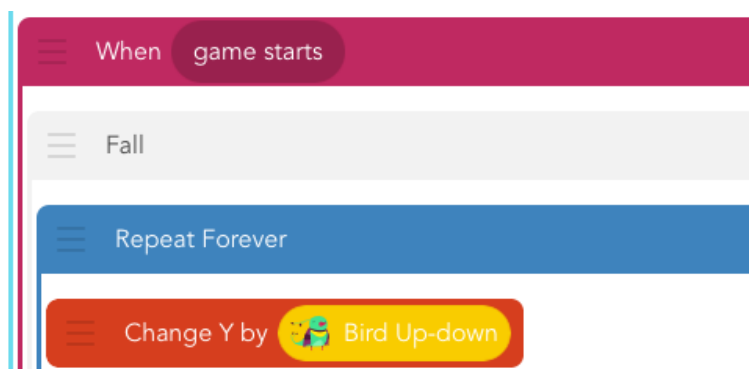
Press New Block to add a new function. Name it something descriptive, like "Fall". All of the code you want to reuse or group as the function should go inside the block. Note that whenever you change a function in one place, it automatically makes those changes wherever else it is used!

1.3 Make bird change Y by [blank] forever



Notice that we don't enter a value in the "Change Y by" block.

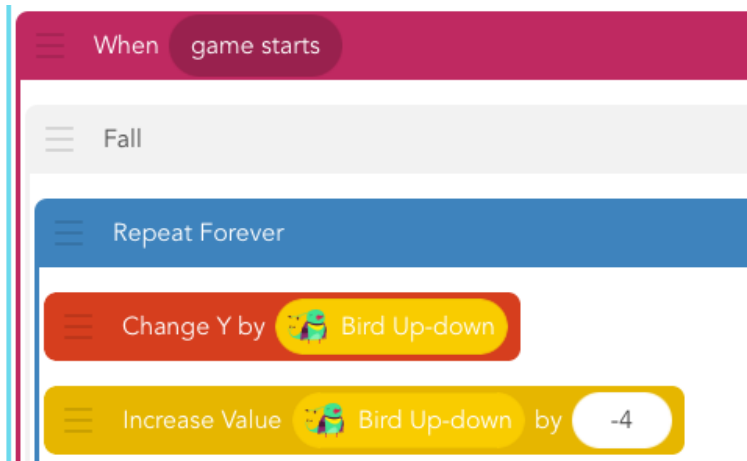
1.4 Create a "Bird Up-down" value and plug it into the "Change Y by" block



This is the crucial moment where you make the bird's movement change over time by using a value instead of a static number. Tap the bubble in "Change Y by" to access your values.

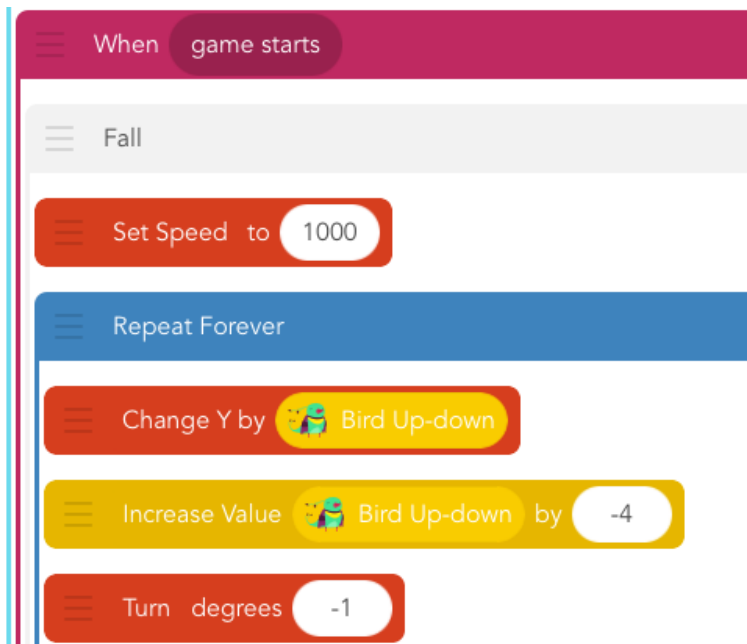
LESSON

1.5 Make “Bird Up-down” decrease forever



Note: we found changing “Bird Up-down” by -4 to work well. Encourage students to experiment with this number.

1.6 Edit falling code: Add a little rotation and some speed



2. Flap when iPad is tapped (ELS) (10 minutes)

As a class, discuss the second component of the physics engine. At this point, the bird just falls nonstop. We need to add the second component of the physics engine that will allow the player to make the bird flap (and stay afloat) by tapping on the iPad. When the bird flaps, it should move up the screen a little, but then keep falling once you stop tapping. You can ask your class to hypothesize how they might make this happen before attempting to code the rule on their own.

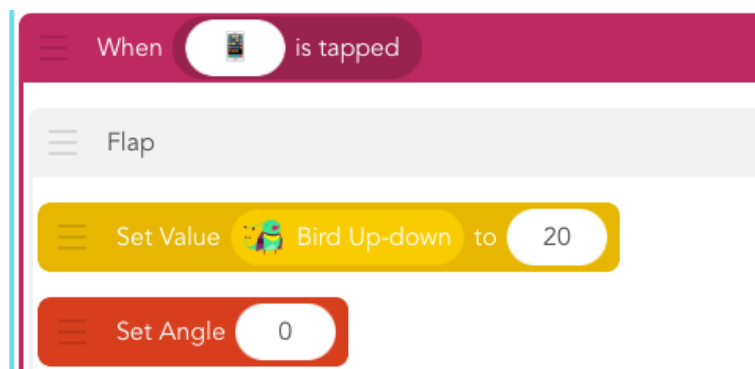
LESSON

We do this by changing the value that controls the bird's position, "Bird Up-down", to a positive number when the bird is tapped. This is the second element of our physics engine. The number we choose here dictates the feel of the game.

You can have students implement this code on their own, in groups, or as a class. Have students experiment with different numbers here and in the falling and turning rules. Fiddle and test until the combination feels right—the bird falls at a believable rate and accelerates up accordingly.

If there is extra time after students have finished their physics engine, add animation to the bird's flapping rule.

2.1 Add new code: Set "Bird Up-down" value to a positive number when iPad is tapped



3. Add obstacles (LS) (10 minutes)

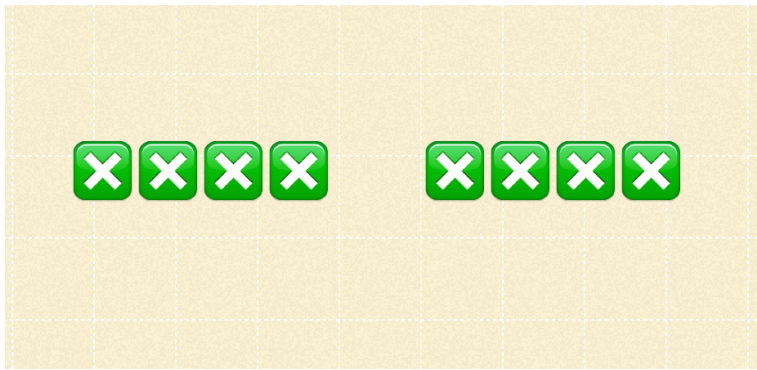
Students next get to reverse engineer the pipes' movement! Reverse engineering is a useful practice in programming in which you examine an existing program or machine and figure out how it works so that you can reproduce it. Using a completed Flappy Bird game as an example, ask your students to discuss how they might make the pipes travel across the screen (while the player attempts to guide the bird through them). Students may remember from Geometry Dash that one obstacle stands in for many and that they travel backward to make the hero look like it's moving forward. But, in this game, the Y position of the pipes is different every time, so you have to use randomness.

Usually we give a lot of freedom in character and emoji choice, but in the case of the obstacles, it's important to follow along exactly, at least in your first version of the game. The code we offer enables the pipes to travel together with a big enough gap for the bird to fly through. This can be changed later once students understand why it works.

Have your students try to code the pipe sequence independently, then compare with their neighbor. Did everyone decide on the same rules in the same order? Did anyone get identical behavior with different rules? The sequence below is one of many possible solutions.

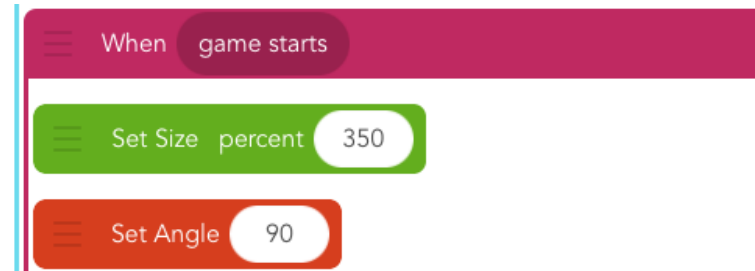
LESSON

3.1 Add obstacle object



To add the obstacle pipes, type in four green emoji squares, then 5 spaces, then another four green emoji squares. Depending on your device, you might need to adjust this. Set the obstacle pipes to the far right and rename the object "Obstacle".

3.2 Add new code to obstacle: Turn and grow



3.3 Add new code to obstacle: Reverse engineer obstacle movement

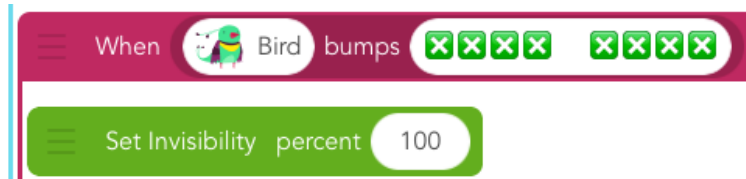


4. End when you hit an obstacle (EV) (5 minutes)

The game ends when the bird collides with the obstacle (pipes). By this point, some students should be able to design a rule that makes the game end on their own. If not, they can work in pairs to build a win state. The simplest implementation is to make both the bird and the pipes disappear upon collision. An advanced programmer could animate the bird to turn toward the ground and fall before disappearing and get the pipes to remain visible but stop moving.

LESSON

4.1 Add new code to bird and obstacle object

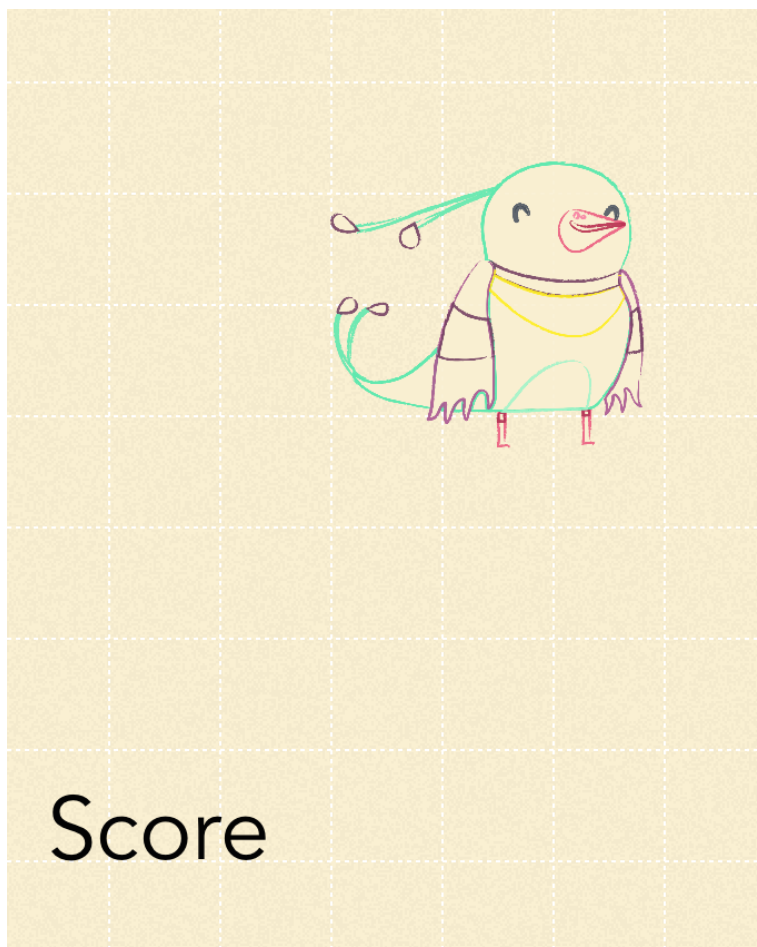


5. Keep score (ELV) (15 minutes) (optional)

If there's extra time, see if students can keep track of the score. The following is a clever way to keep score, and to automatically stop the score from increasing further when the game is over.

It depends on the pipes either stopping or going invisible when they collide with the bird. An object will both display the current score and detect when the score should be increased! It relies on a mechanism in which the player earns points only when the pipes pass the bird without colliding. Conveniently, as soon as the pipes get past the bird, they bump into the Score text, so that event can trigger the score increase. Students might remember how to make a text object display a value from Quiz. If not, guide them to a good solution.

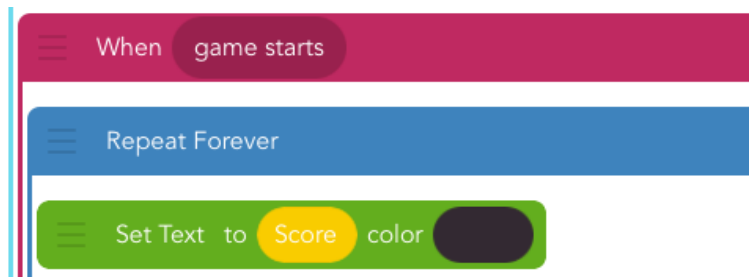
5.1 Add a score object



Name score object "Score" and place it in the bottom left corner of the screen, even further left than the bird.

LESSON

5.2 Add new code: Make score object show score value forever



You will need to create a new value, "Score", for this step.

5.3 Add new code to the score or the obstacle: Increase score



5.4 Publish your game! What's your high score?

DIFFERENTIATION

Differentiation (15 minutes, optional)

- Add a better background
- Add more birds
- Change the speed of the pipes
- Add bonuses or other objects

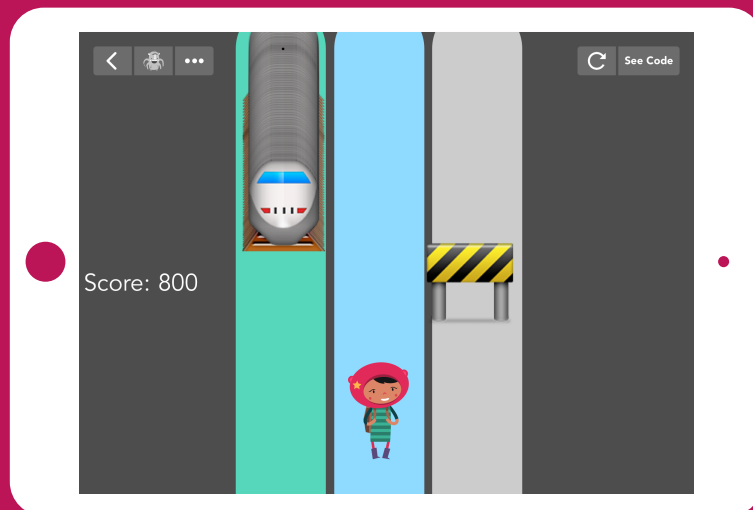
REFLECTION

Reflection (5 minutes)

- Compare Geometry Dash and Flappy Bird.
 - What elements do they have in common?
 - How are they different?
 - How could you use some ideas from one to improve the other?
- What is a physics engine?
- What other games have them?
- Should game physics always be like real-world physics?
 - Why or why not?

LESSON 5

SUBWAY SURFERS



An action game that has multiple possible implementations.

TIME

45-60 minutes (+15 minutes of optional, free code time)

BIG IDEA

There is often more than one solution to a problem, and some solutions are better than others. There may be another way!

SKILL FOCUS

-
- Collaboration
 - CCSS.MATH.PRACTICE.MP3 Construct viable arguments and critique the reasoning of others
 - NGSS Practice 7 Engaging in argument from evidence
 - NGSS Practice 8 Obtaining, evaluating, and communicating information
-

KEY VOCABULARY

Pair programming: A technique in which two people work together on one device

TRANSFER GOALS

-
1. Students will try pair programming.
 2. Students will use appropriate vocabulary to communicate with their partner.
 3. Students will compromise and agree on solutions.
 4. Students will formulate strategies for dealing with disagreement, and compare solutions based on implementing and evaluating, without taking it personally.
-

MATERIALS

– 1 iPad per student, or 1 iPad per 2 students, for pair programming
– Video available on YouTube:
<http://hop.sc/SubwaySurferVideo>
– Complete project available:
<http://hop.sc/subwayproject>

Subway Surfers is an arcade-style game where the player controls a character running in three lanes, dodging trains and other obstacles by swiping to change lanes. We will build a stripped-down version of this game, focusing on the interaction between the hero and the trains. In extra time, students may be inspired to add the other elements of Subway Surfers, like collecting coins and bonuses, and jumping and/or rolling.

This lesson is a great chance to introduce **pair programming**, a technique in which two people work together on one device.

This lesson is an opportunity to work together, using relevant vocabulary, explaining ideas and comparing possible solutions. Assign pairs in whatever way you prefer. Where there is an odd student, create a group of three. The key to working together is understanding that there may be more than one solution to a problem, and the team decides together which solution is best based on how well it works—not on whose idea it was!

The crux of this lesson is deriving the formula for the train choosing one of three lanes to appear in. Be sure to familiarize yourself with it in advance of teaching the class.

LESSON

0. Discussion (5 minutes)

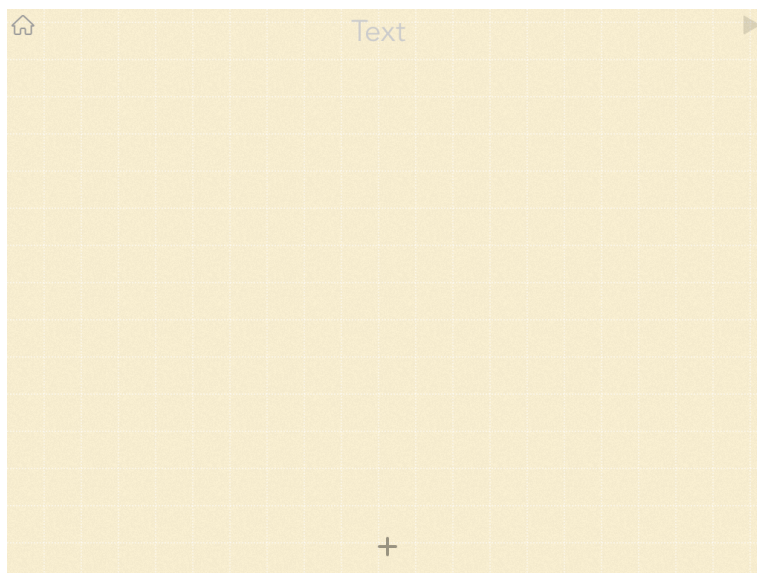
In **Pair Programming**, two programmers take turns coding. The driver holds the iPad and does the coding, and the navigator watches for mistakes, helps come up with solutions, and keeps the pair on task. Every few minutes, they switch roles. Both jobs are equally important! This is especially helpful when making complex games, because there are lots of details to attend to, and an extra pair of eyes can help spot errors before they become bugs. In professional environments, this is often considered a best-practice (and we do it frequently at Hopscotch).

What things would you need to keep in mind while pair programming? Ask your students to make a list of things that they should and should not do while pair programming (e.g. they should ask questions about why the programmer chose a certain block; they should not call each other names.)

1. Draw lanes (S) (10 minutes)

Show your students a completed version of the game. The first step is to create the three colored tracks that run down the middle of the screen. See if your students can draw three random-color lanes by reusing as much code as possible (abilities are helpful for this!). One solution is to make one drawing object, and repeat setting the position and drawing the line. Another possibility is to make three separate drawing objects that each use the same “Draw Lane” ability.

1.1 Add an invisible text object to the top middle of the screen

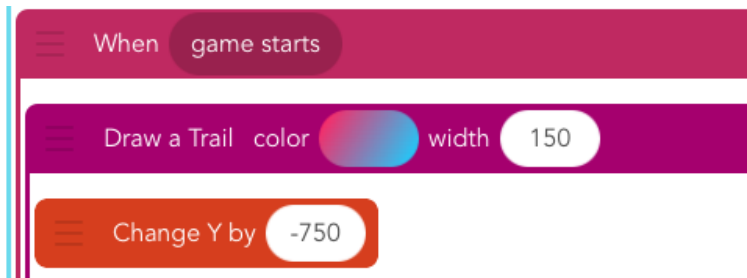


Remember that a shortcut for an invisible object is to make a text object, then tap “X” on the upper left corner when it asks you to set text.

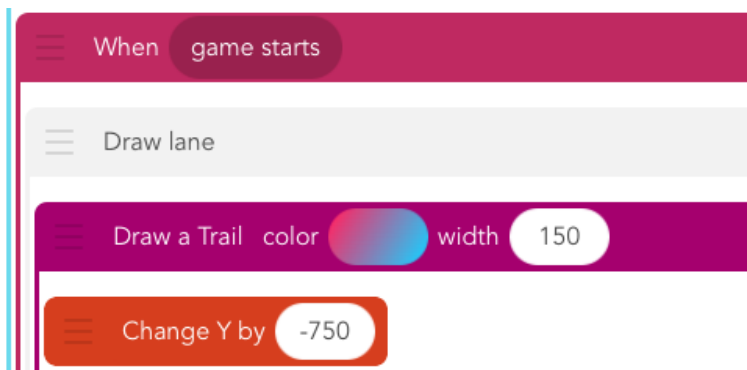
How would your code change if you put the text object at the bottom of the screen? It would have to move by a positive amount.

LESSON

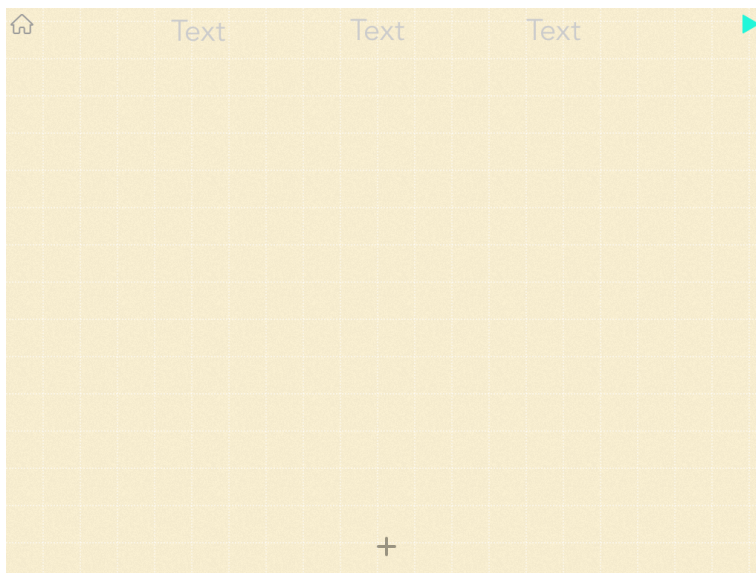
1.2 Add code to text object: Draw lane



1.3 Make it a custom block



1.4 Add two more text objects at appropriate places, give them the "Draw Lane" block



2. Control hero (ELCV) (10 minutes)

As your students saw when playing the sample game, in Subway Surfers the player controls the hero by swiping right and left on the iPad, jumping across lanes to avoid the oncoming objects. In students' versions, they will make the player swipe the hero, rather than the iPad.

LESSON

The rules for animating and controlling the hero are a chance to practice the concepts of events and loops learned in previous lessons. You can discuss the general requirements of this action as a class and then give your students the chance to try coding these rules on their own. The sample code below is just one possible solution.

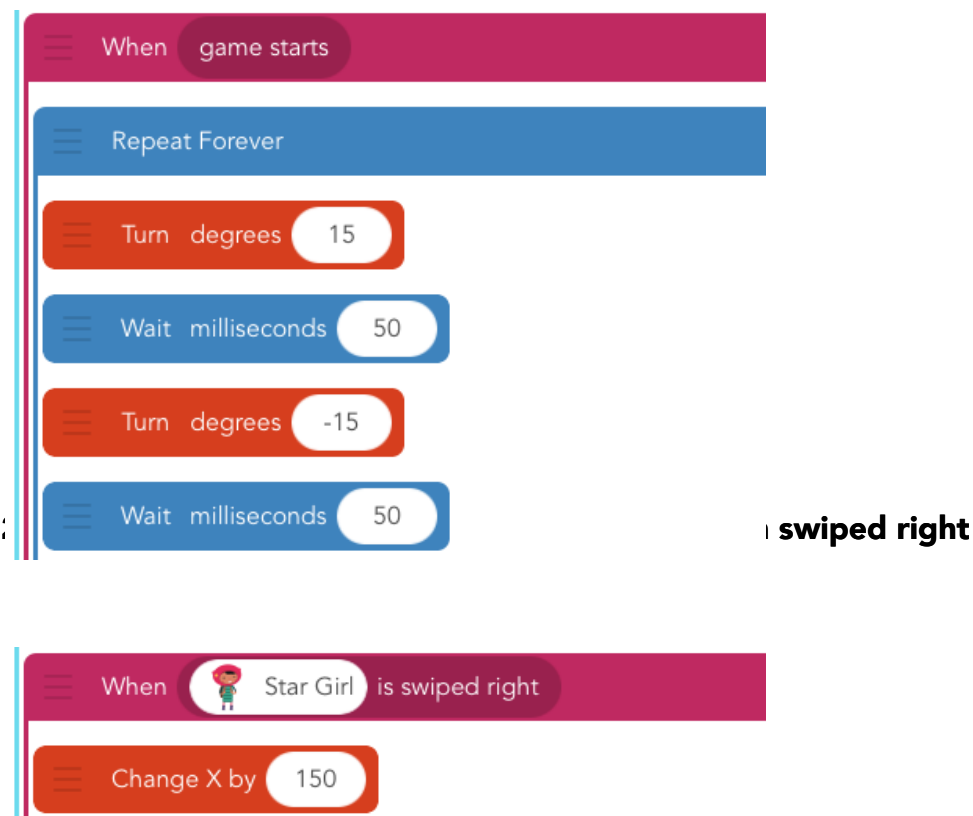
Check in to compare the results and make sure everyone's is working and, if your students have not brought it up, then raise the issue that nothing is stopping the player from swiping their hero out of the lanes and "off the board". In order to establish boundaries, we can introduce logic that only allows the character to move within a certain zone.

In our game, the character can move in one case but not another; we will tell the computer to check and see if a condition is true before moving on to execute the command. We do this with a **conditional** that checks the character's position on the X axis.

As a class, decide the range of X positions for which the character may be moved right and left, respectively. Derive the inequalities from these numbers (e.g. If character's X position is greater than [some number] you cannot move it further to the right). Code these rules as a class, in pairs, or as individuals. The numbers in the sample code are a possible solution.

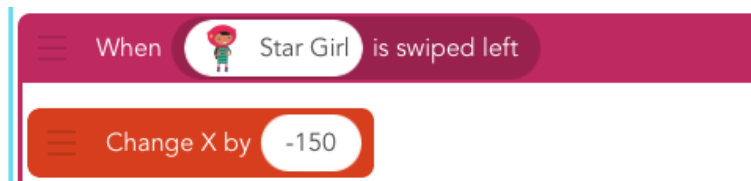
2.1 Add hero object

2.2 Add new code to hero: animate hero object to run forever

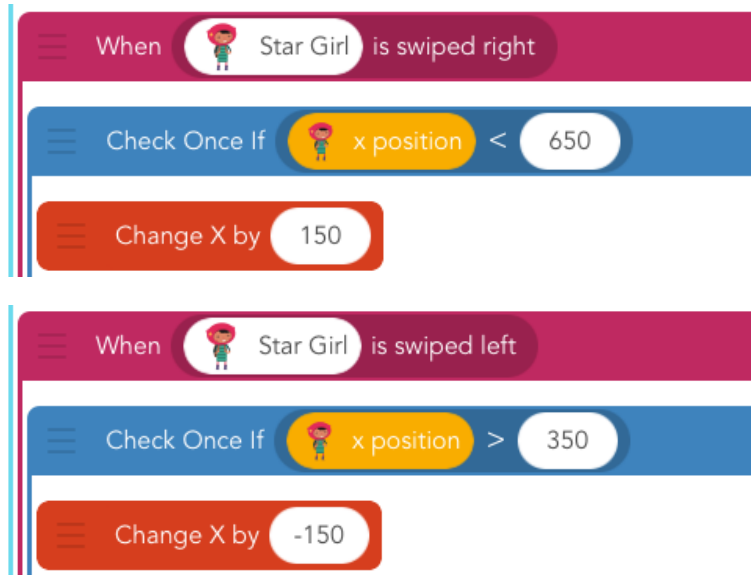


LESSON

2.4 Add new code to hero: Change X by -150 when swiped left



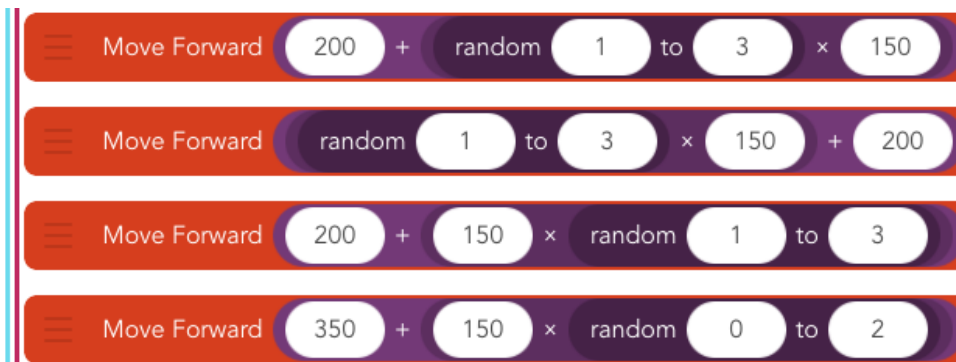
2.5 Edit swipe code to check for hero's X position



3. Add trains (SV) (10 minutes)

The most challenging aspect of this game is setting the trains to appear as challenging obstacles. Students should start, in pairs, by discussing how to program one train to move along one lane, starting at the top. If they forget the height of the screen, they can use the built-in value, "iPad's height".

Once they have a train running in one lane, they will need to program it to randomly appear in different lanes. This is accomplished by choosing a range of X-positions in which it may appear, using one of these formulas for the X position:



Note: If you're on an iPhone, you will need to adjust the width of the lanes to fit the smaller screen.

LESSON

If these formulas are confusing, check out the code below. For older students, work out this formula in pairs. Younger students may need to be shown or led to discover the formula. In either case, test the formula out with the possible values on paper before trying it in Hopscotch. Have students code it when they think they have it right, then evaluate the formula by testing its effect in the game.

You can also ask the class how they would change the code if their lanes were wider. (change the X-position by 150 to accommodate lane width, make 200 lower). How would they change it if they had more than 3 lanes? (Change second number range to number of lanes, make 200 lower) For older students: Consider that these changes affect more than just one number; can you think of a general-purpose equation using variables for "NumberOfLanes" and "LaneWidth"?

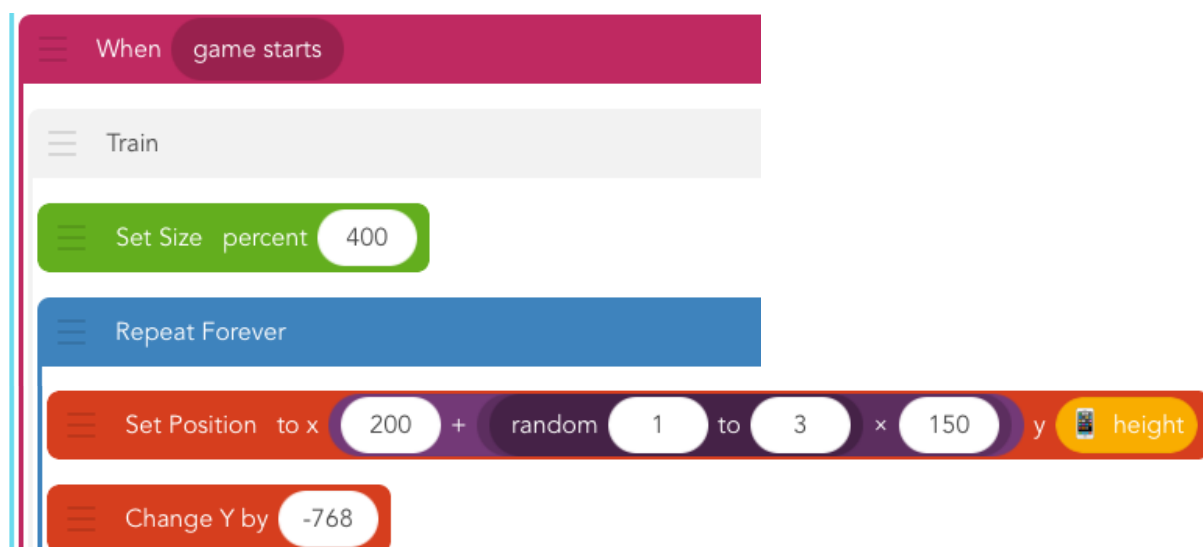
3.1 Add train object and make it bigger



3.2 Add new code to train: Make it run along one track

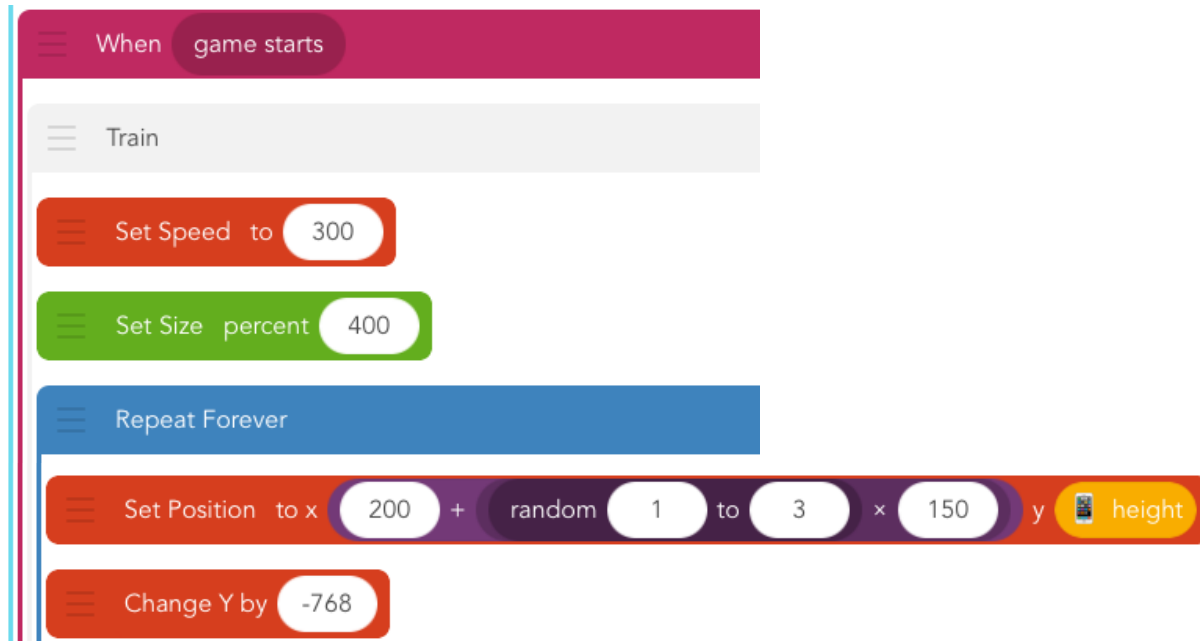


3.3 Edit code to choose random lane



LESSON

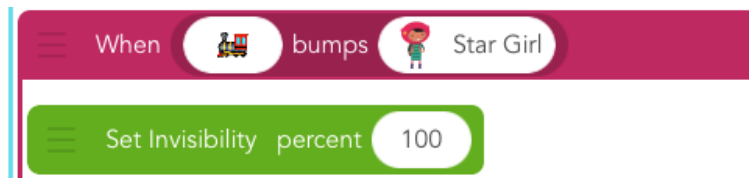
3.4 Adjust speed



4. Collisions (ES) (5 minutes)

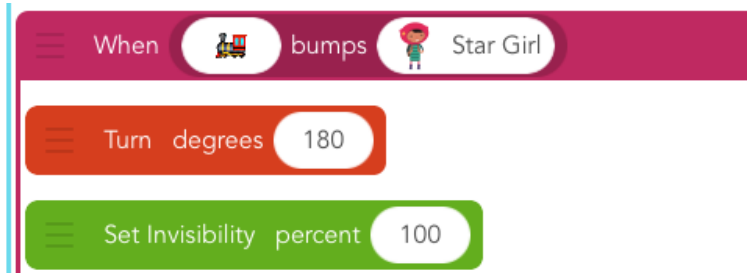
Ending the game is an opportunity for creativity. How do students want to indicate to the player that a collision has occurred? If you want, for example, to grow and spin at the same time, that's concurrency, and should be implemented with two different rules with the same event. Ask students to come up with a good indication that a collision has occurred, code it, and show it to a partner. The partner can help debug if necessary.

4.1 Add new code to hero: Disappear when you collide with a train



LESSON

4.2 Edit hero's code: Animate collision



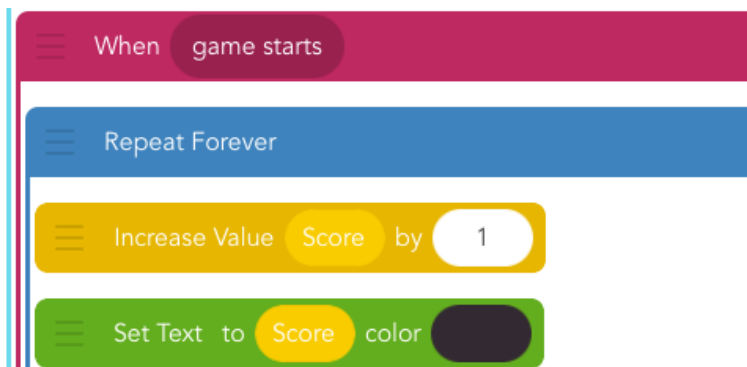
5. Keep score (ELV) (5 minutes)

As students might recall from earlier lessons, we use values to store numbers that may be changed. To create a score, students should introduce a text object that will keep track of and display the score. In this case, though, the score shouldn't keep increasing forever; it should stop when the hero collides with a train. You can ask students to discuss a solution to this problem. One possible way to implement the score is to change the event that triggers this rule, so instead of happening forever, it only happens when the hero is visible on the screen (e.g. has not collided with the train).

5.1 Add a score object

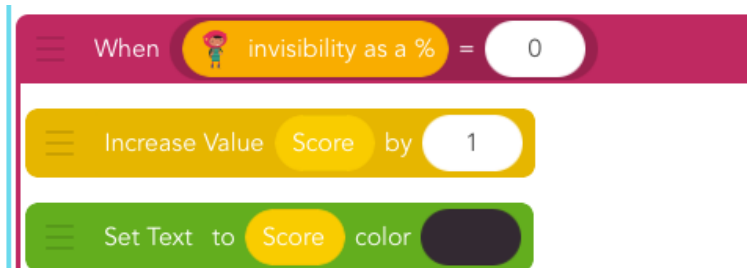
Name score object "Score".

5.2 Add new code to score: Make it count up by 1 forever and display score



The score is increasing as fast as the computer can go. Is that too fast? Add a wait block to slow it down. Find a good value for wait.

5.3 Change the event and delete the loop



The reason you can delete the loop is that this event is already repeating!

5.4 Publish your game!

DIFFERENTIATION

(15 minutes, optional)

In Subway Surfers, you have the option of jumping over or rolling under barriers, in addition to the option of avoiding them. Can you add this functionality to your game? Devise a strategy in pairs, either on paper, or in code. Share your ideas with the class and see how they are similar. Can we come up with a better solution by combining our ideas?

- Add hurdle object
- Make train's behavior a function, then give hurdle the same train rule
- Give hero a swipe down rule to roll - make "rolling value"
- When hero bumps hurdle, check if rolling
- Can you jump over the hurdles by swiping up?
- Can you make the trains speed up as the game goes along?

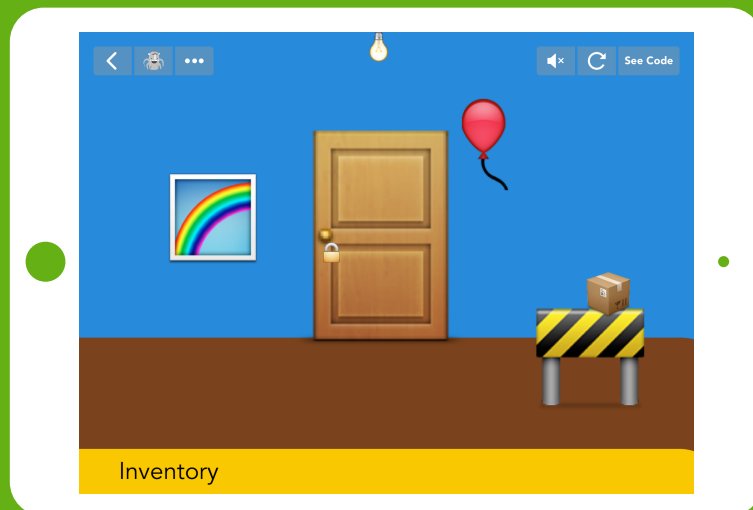
REFLECTION

(5 minutes, optional)

- What is pair programming? Why is it useful?
- Can you see a pattern in the order we use to build a game? Why did we choose that order? Is it the best, or can you think of improvements?
- Why are some values a different color than others? (Some are built-in and you change them with dedicated blocks, others are the ones you made and you have to change them by using set value or increase value.)

LESSON 6

CAN YOU ESCAPE?



An open-ended logic game, enabling students to make the connection between real-world logic and programming logic.

TIME

45-60 minutes (+15 minutes of optional, free code time)

BIG IDEA

The way to make good programs is to have ideas and make mistakes, over and over. Stick to it!

SKILL FOCUS

- Iteration
- CCSS.MATH.PRACTICE.MP1 Make sense of problems and persevere in solving them
- CCSS.MATH.PRACTICE.MP6 Attend to precision
- Practice 1 Defining problems
- Practice 6 Designing solutions

KEY VOCABULARY

Logic: The process of making decisions

Iteration: Having ideas and making mistakes, over and over

TRANSFER GOALS

1. Students will generate ideas for game elements and design solutions.
2. Students will test solutions and design improvements.
3. Students will recognize that some ideas take more than an hour to implement.
4. Students will be able to define the 5 Core Coding Concepts and give examples of their uses.
5. Students will recognize the programming convention of 1=True, 0=False.

MATERIALS

- 1 iPad or iPhone per student, or 1 device per 2 students, for pair programming
- Video available on YouTube:
<http://hop.sc/EscapeTheRoomVideo>
- Complete project available:
<http://hop.sc/escapetheroomproject>

An Escape Game is an adventure game in which, surprise surprise, the player is locked in a room and must locate objects and solve puzzles to escape. Can You Escape? is a current example, but The Room is related, and Myst and Monkey Island are ancestors. This type of game relies heavily on logic, order of events, and attention to detail. It pulls together the programming concepts introduced in previous lessons and gives students a great opportunity to be creative. Students will create **values** to keep track of the state of the game, then use **conditionals** and **events** to change the behavior of the game, depending what state it is in.

The foundation of this game, and computing as a field, is **binary logic**. In Escape the Room, students will use binary logic to assess whether the player has completed the activities required to escape. While they have not, they cannot leave (the escape condition is untrue, or 0). Once they have completed the activities required, the escape condition becomes true (1) and the player can escape!

This activity is an excellent opportunity for customization and extra time. It can be worthwhile to start with a quick summary of these concepts and prepare students to implement them on their own. This process of making small, incremental changes to your code—**iteration**—is a key tenant of programming.

LESSON

0. Discussion of Escape genre (5 minutes)

Some of your students will be familiar with the escape game genre, and some will not. Take a few minutes to get everyone on the same page. Look at some examples on the projector, or elicit descriptions of good escape games from the class. Talk about why they are fun. A good escape game should be challenging, but not impossible, and have a few puzzles, but not too many. Are there other important elements, like an interesting setting, good animation, or sound effects?

Remind your students that the best programmers make lots of small changes to their code and test them frequently. This process, **iteration**, helps you identify bugs early and make sure that everything works as you intend. We, at Hopscotch, are always reminding ourselves to slow down and check our work. :)

1. Introduction to basic logic (EVC) (10 minutes)

Discuss the basic premise of the game with your class: the player is trapped in a room that has a locked door and they, initially, do not have the key. The player's task is to find the key and escape, but this is complicated by series of puzzles that they have to solve to find the key.

Since the door cannot open unless the player has found the key, the game must know whether or not it's been found at all times. We will create and use a value "HasKey" to keep track of this condition with the help of one of computing's most important concepts: **binary logic**.

In binary logic, we use 0 to represent FALSE and 1 to represent TRUE. These are known as truth values. Imagine a lamp: When the switch is in the ON position, 1 light is on, so the sentence "a light is on" is TRUE. When it is in the OFF position, 0 lights are on, so the sentence "a light is on" is FALSE. Now, if you have two lamps, you can deal with them both: If the first switch is in the ON position (1) and the second is in the OFF position (0), the sentence "a light is on" (0+1) is TRUE! If you've ever heard jokes about computing being all 0s and 1s, this is why...

Start by having students independently establish the room by adding a key and a door that opens when swiped. Then, in pairs or as a class, discuss the logic required to only open the door when the key is found. Depending on your students' experience level, you might ask them to come up with and share out potential logic rules.

Make sure everyone is on the same page and then have students code the logic rules. In the escape game, when the key is found (tapped), "HasKey" will equal 1 (True), and when it has not, "HasKey" will equal 0 (False). The door should only open when "HasKey" equals 1. If "HasKey" does not equal one, the door should signal to the player that it cannot be opened. This is a good opportunity to discuss the difference between "Check Once If" and "Check Once If // Else".

LESSON

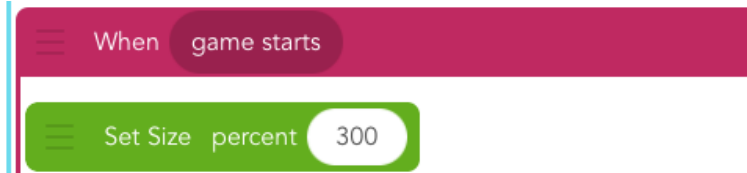
The door cannot be opened unless the player has the key ("HasKey" = 1). Students will use conditionals to create two states:

The door opens if the player has the key.

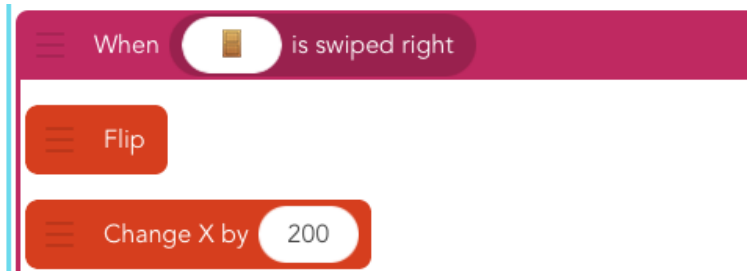
The door wiggles (but does not open) if the player does not have the key.

1.1 Add door & key emojis

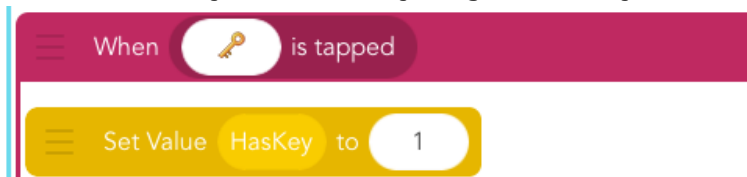
1.2 Make the door bigger



1.3 Animate door opening



1.4 Set HasKey to 1 when you get the key



You will need to add the "HasKey" value. Remember that all values are 0 by default before you set them.

1.5 When door is swiped, check if "HasKey" = 1, then execute the opening code.



Check out that "Else" condition!

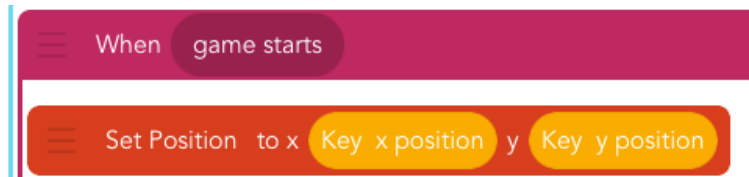
You will need to create your own Wiggle ability. Here is one way to do it.

LESSON

2. Two-step logic (ES) (10 minutes)

This game is too easy! Let's start adding things to make the key harder to get. One possible solution is to obscure objects by putting one in front of the other and revealing the second only when the first (outermost) has been moved. The below code shows how to do this. Note that objects show up in the order you added them. If you want one character to be in front of another, use "Bring to Front". There is also "Send to Back". What does that do? Discuss where in your code you would use these (generally as the first action).

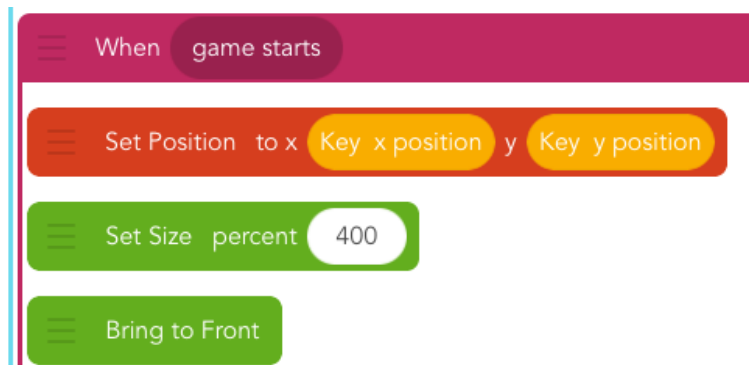
2.1 Add portrait emoji at same position as key



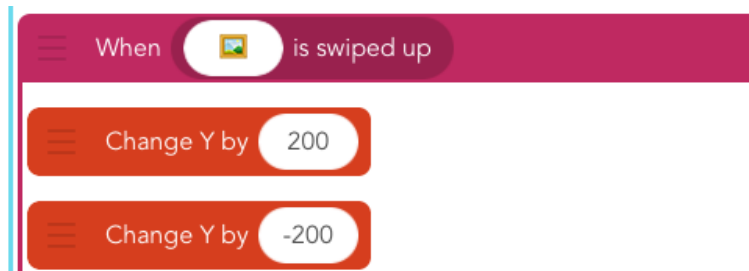
2.2 Make portrait bigger



2.3 Bring to Front



2.4 Make portrait slide up, then back down, when swiped



LESSON

2.5 Test and adjust positions of objects

3. Inventory (ESV) (10 minutes)

An Inventory is a widely-used game element that helps the player keep track of the things they "have". There are multiple ways to implement an Inventory; one way is to create a designated storage place on the screen with copies of all Inventory items, and then to show or hide each item depending on whether it's been collected. We already keep track of this state using the "HasKey" value!

Students should start by drawing an Inventory box, and then add a new key emoji. In pairs, see if they can identify the two rules this key needs (hint: one for each state). As a class, make sure everyone has the right rules. Then, together, adjust the door's rule to change the "HasKey" value back to 0 when it has been used to open the door. Test the program and check whether the Inventory key is appearing and disappearing at the right times.

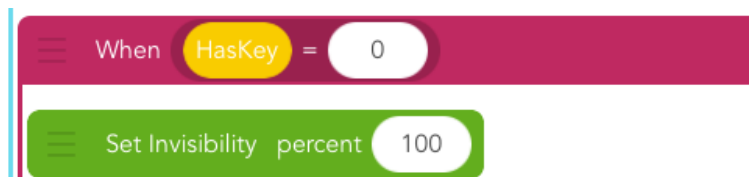
3.1 Add Inventory label (optional)

3.2 Draw Inventory box (optional)



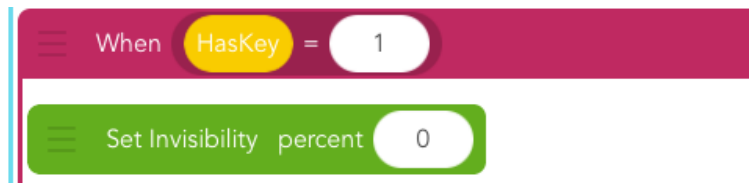
3.3 Add a new key emoji to Inventory

3.4 Add code: Do not display key in Inventory when condition is false

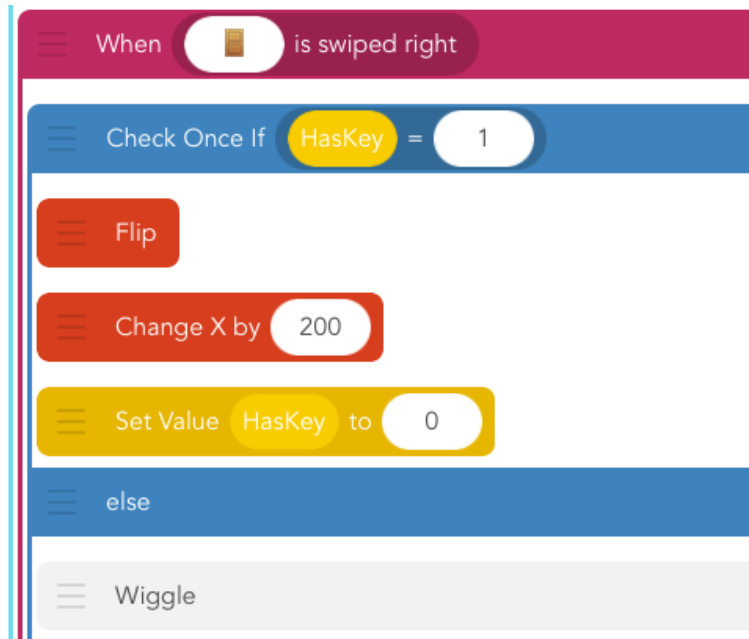


LESSON

3.5 Add code: Display key in Inventory when condition is true



3.6 Edit the door's swipe code to set "HasKey" to 0 (when you've used it up)



4. Win State (EV) (5 minutes)

It's important to tell the player when he or she has won. We'll keep track of whether the player has won using a new value, "Ending". When the game is finished, "Ending" will be 1. Ask your students to consider which event would trigger the value to increase from 0 (starting value) to 1?

As in previous games, ask students to create a "win state" text that will appear when the "Ending" value is 1.

As a class, discuss what would happen if there were another set of conditions that set "Ending" to 2 or 3 and caused different endings to happen.

LESSON

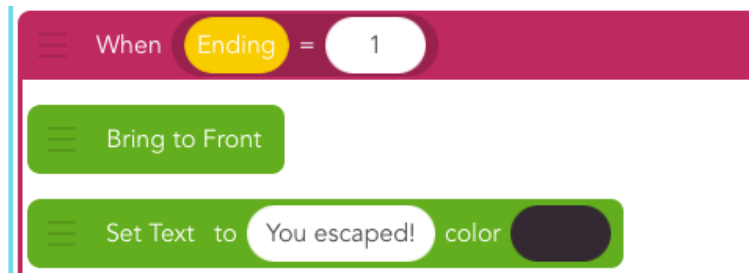
4.1 Edit door's code: Opening door sets "Ending" to 1 (remember all values are 0 by default when you make them)



Create a new value "Ending".

4.3 Add new text object, cancel text

4.4 Add new code to text object



4.5 Publish your game!

DIFFERENTIATION

(15 minutes, optional)

- Show how you could have two different endings
- Add appropriate sounds to each event
- Add more puzzles to the game (search for escape games in the Hopscotch Community for inspiration)
- User testing: Watch silently while other people play your game and observe how they play it
- Active feedback: Ask players what they thought of your game
- Make a real-life escape game in your classroom!

REFLECTION

(5 minutes, optional)

- What is logic? Is there a difference between coding logic and real-life logic?
- What's the difference between IF and IF...ELSE?
- How do you check if two things are both true? (Nested conditionals)
- How do you choose how to respond to feedback? Do you have to make all the changes someone suggests?

LESSONS 7 & 8

Now that your students have completed Lessons 1-6, they should have a solid grasp of the core coding concepts we've covered and of Hopscotch as a tool. These lessons are designed to give students an opportunity to practice their skills and apply them in new ways. You can determine the right amount of structure based on your class's experience and needs. Younger kids might need more concrete assignments, whereas older students may relish the opportunity to make something from their imagination.

An important idea to explore in creative coding is identifying which problems you can solve with a computer (a computable problem) and which are better solved by a human. For example, a video game can easily be solved by a computer, but a mind-reading game cannot.

Here are some suggestions of how you can guide your class; do what is best for your goals:

Go deeper with games from previous lessons:

- Refine one of the games you made in lessons 1-6
- Design and make your own game, using pieces from these lessons
- Form a game dev team and make a big game together, assigning roles to each team member
- Have a game showcase with another class

Link the coding experience to other subjects:

- Write a review of someone else's game
- Make a commercial for your game: video, website, magazine ad
- Make a video tutorial for how to use a certain block, or define a term
- Write a persuasive essay about two different ways to do something, and which one is better
- Record your project as an animated gif (make a meme)

Explore Hopscotch in greater detail:

- Do the tutorial videos in the app
- Find a game you like in the Community and remix it
- Design a game as a team and implement it, using the Forum to ask for help

	Unsatisfactory	Competent	Proficient	Distinguished
Execution	Program does not work, or has major flaws that prevent its intended use	Program mostly works, and has only minor flaws	Program works in the way the student intended	Program is functional and refined, with extra features that add functionality or improve upon the original design
Content	Program lacks understanding of core concepts and skills	Program shows some understanding of core concepts and skills	Program reflects understanding of concepts and skills	Program shows synthesis of new and old concepts and skills
Reflection	Student cannot describe how their code works	Student can mostly describe how their code works	Student can describe how their code works and can make changes that have desired effects	Student can describe how their code works and how they wrote it, and help others debug their code
Habits of mind	Student is not aware of the goal of the program, is frequently off-task, does not offer their own ideas, and gives up when it is difficult	Student is aware of the goal of the program, returns to the task when asked, has some ideas when prompted, asks for help when stuck	Student understands the goal of the program, has their own ideas, rarely goes off-task, and attempts to solve problems first before asking for help	Student embraces the goal of the program and chooses to try out new ideas and multiple solutions, even when they are challenging

Inspired by <http://www.edutopia.org/pdfs/blogs/edutopia-yokana-maker-rubric.pdf>

GLOSSARY FOR YOUNGER STUDENTS

Ability: Code that can be reused

Algorithm: A recipe for a program

Coding: Telling computers what to do

Concurrence: Two things happening at the same time

Conditional: Statements of the form "IF (something is true), THEN (do an action)"

Debugging: Finding mistakes in your code and fixing them

Event: When something happens

Iteration: Having ideas and making mistakes, over and over

Logic: The process of making decisions

Loop: Code that repeats

Operator: A mathematical symbol that makes an equation

Program: A set of instructions a computer can understand

Programmer: A person who writes programs

Programming Language: A set of rules or blocks that can be used to write any program

Random: When there's no pattern

Range: The highest and lowest number random can choose between

Rule: Instructions that tell your computer what to do (the command) and when to do it (the event)

Sequence: The order in which instructions are given to the computer

Object: A character or text with its own rules

Value/Variable: A holder for a number

GLOSSARY FOR OLDER STUDENTS

Ability/Function/Procedure/Subroutine: A saved set of blocks. What we call abilities in Hopscotch are known as functions or subroutines in other programming languages. Easily replicable routines are a key concept in computer programming, and allow you to scale your code and create complex programs

Algorithm: Algorithms are at the heart of computer science; they are the recipes that computers follow to solve problems

Bug: An error that a programmer has made in his or her code

Coding: Writing the rules of behavior for a computer to follow automatically; programming

Concurrency: Two or more things happening at the same time, or triggered by the same event

Conditional: Statements of the form "IF (something is true), THEN (do an action)"

Debugging: Finding mistakes in your code (bugs) and fixing them

Event: A trigger that the computer recognizes and causes it to do some action. In Hopscotch, events include "When the iPad is tapped" or "When the play button is tapped"

Iteration: The repetition of a process

Logic: The science of the formal processes of thinking and reasoning

Loop: A repeating set of instructions

Operator: A mathematical symbol that produces a value

Program: A set of instructions a computer can understand

Programmer: A person who writes programs

Programming Language: A set of words, rules, blocks or instructions that can be used to write a program

Random: Any number or item among a set. The lack of a pattern among items in a set

Range: The highest and lowest number random can choose between

Rule: Rules tell your object what to do and when to do it. When you make an ability and pair it with an event, you create a rule

Sequence: An ordered list of things (instructions, blocks, numbers, etc) which can be triggered by an event or repeated

Object: A character or text with its own rules on screen

Value: A holder for a number. Also known as a variable

REFERENCES

Wiggins, Grant P., and Jay McTighe. Understanding by design. Ascd, 2005.

<https://books.google.com/books?id=hL9nBwAAQBAJ&dq=understanding+by+design+apple+unit>

<https://computationalthinkingcourse.withgoogle.com>

<http://csta.acm.org/Curriculum/sub/CurrFiles/CompThinkingFlyer.pdf>

<http://www.corestandards.org/Math/Practice/>

<http://www.nextgenscience.org/sites/ngss/files/Appendix%20F%20%20Science%20and%20Engineering%20Practices%20in%20the%20NGSS%20-%20FINAL%20060513.pdf>

<http://www.edutopia.org/pdfs/blogs/edutopia-yokana-maker-rubric.pdf>

ACKNOWLEDGMENTS

Huge thanks to Dr. Emily Thomforde, David Dulberger, Jesse Beutow, Thomas Abend, Ashley Gavin, Miranda Gohh, Elizabeth McDonald, Jessica Wertheim, Redwood City Public Library, Taft Community School, and all of the educators who have provided ideas for and feedback on this curriculum.

Most importantly, thanks to Hopscotchers everywhere for making amazing things and reminding us that learning should be fun. This curriculum is inspired by and dedicated to you <3.