HOPSCOTCH
Curriculum

Learn to Code - Make Cool Stuff

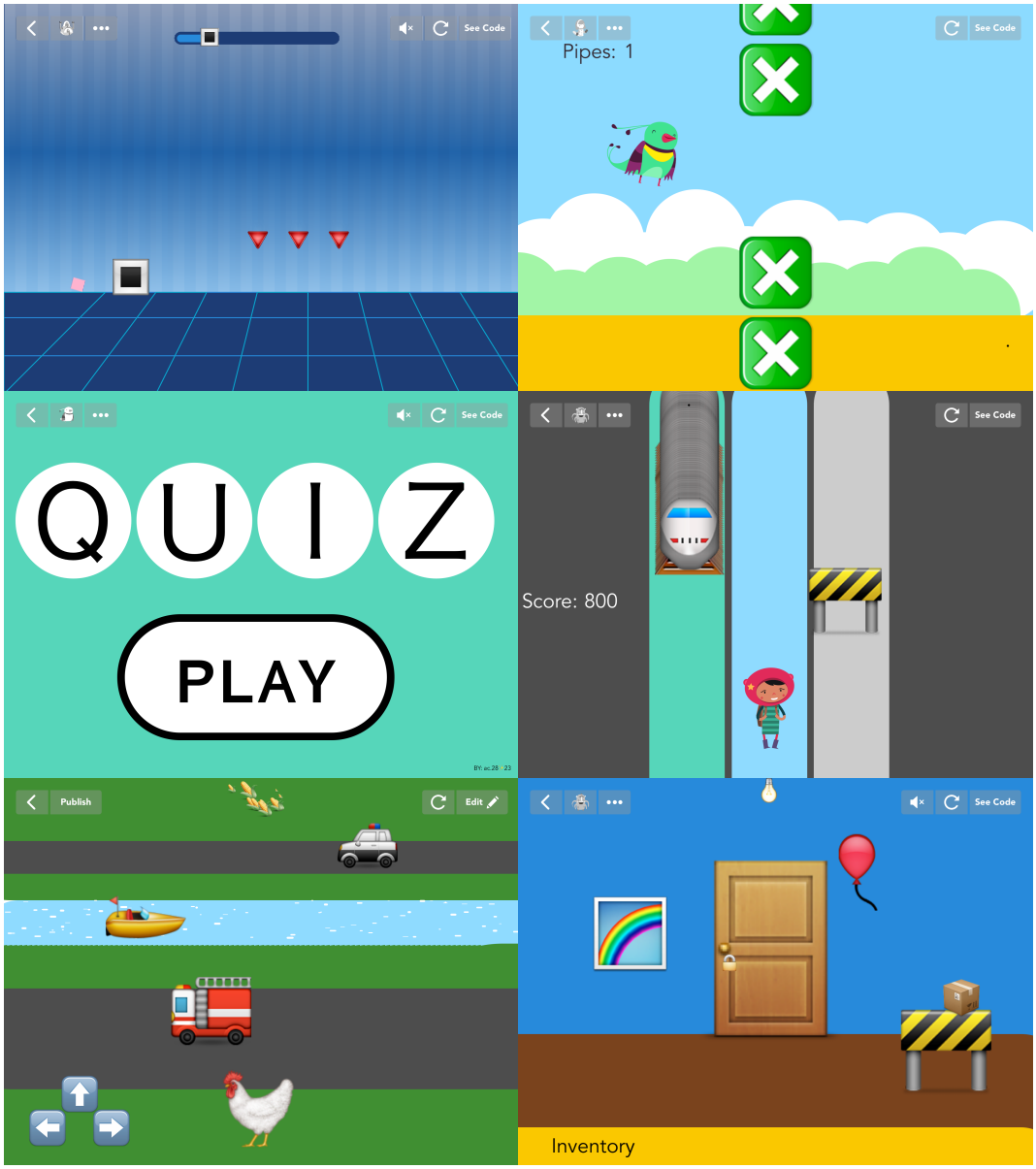

TABLE OF CONTENTS

HELLO!

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!

HELLO!

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

MATERIALS

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 possiblities 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)"

STANDARDS

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!

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.