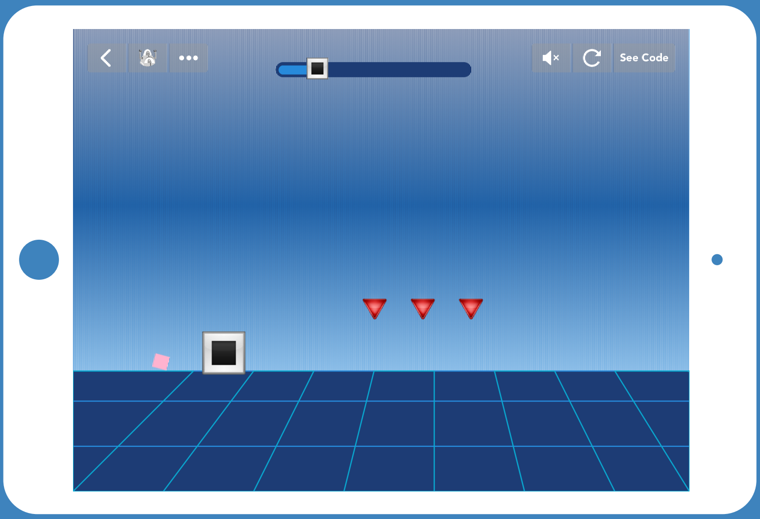
LESSON 2
GEOMETRY DASH



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

TEACHER BRIEF

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.

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?

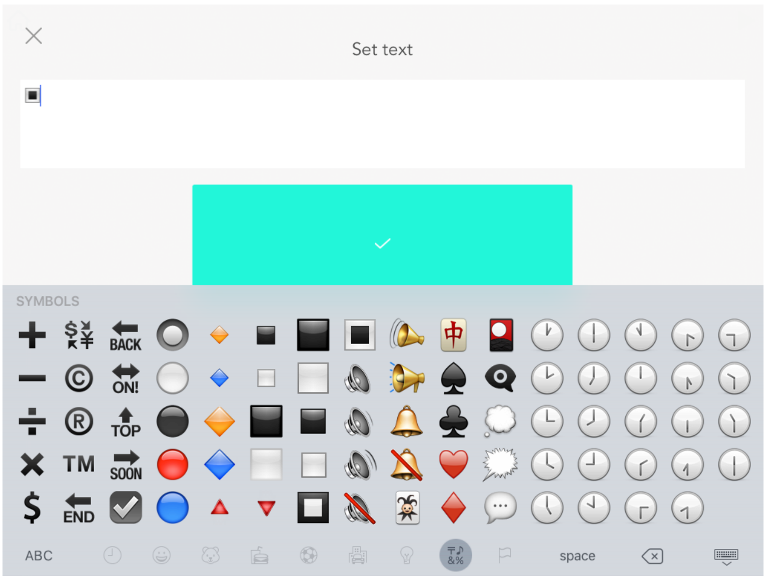
1. Control the hero (ES) (10 minutes)
In Geometry Dash, the player controls a little square that flips and jumps over obstacles.

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.

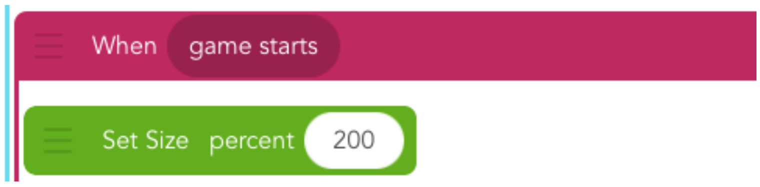
1.1 Add hero object (square emoji) and place it near the bottom left corner of screen



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".

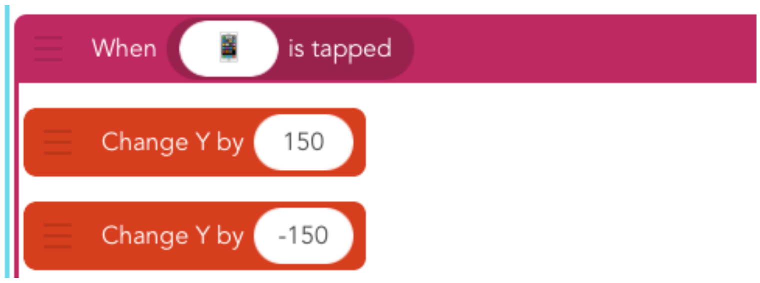
1.2 Add code to hero to make it bigger



```
When game starts
    Set Size percent 200
```

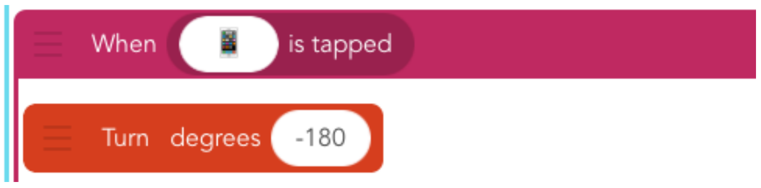
If you choose a number other than 200, all of the other numbers we give will also have to change. This is an opportunity for debugging.

1.3 Add code to hero to jump



```
When [▣] is tapped
    Change Y by 150
    Change Y by -150
```

1.4 Add new code to hero to turn while jumping



When [📱] is tapped

Turn degrees -180

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.

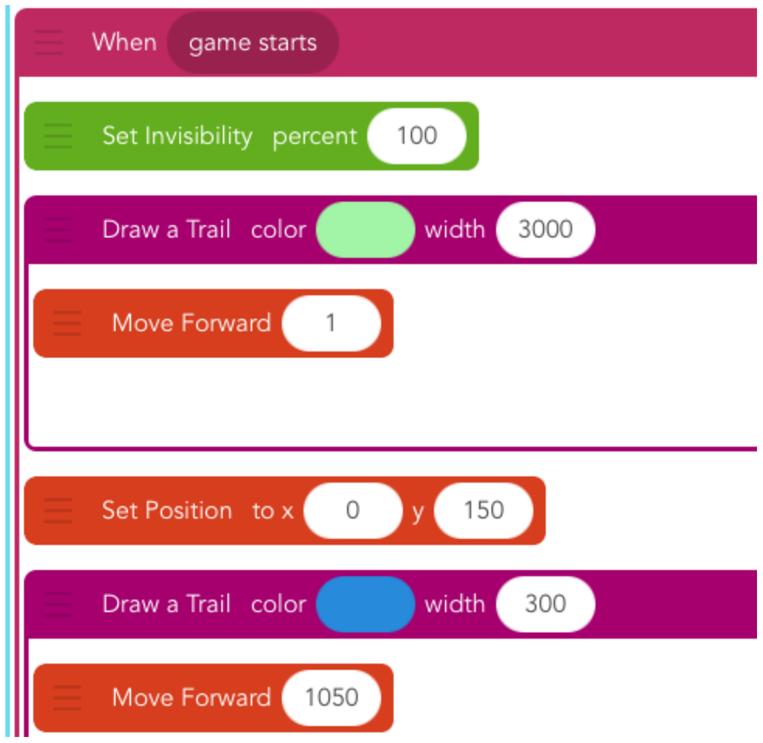
In Hopscotch, we draw with a block called "Draw a Trail" that sets the color and width of the line, then executes the code inside – typically "Move Forward" – as if the object were dragging a marker behind it. It will make a dot if it just moves by 1. To color in the whole screen, make a huge dot (width 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)

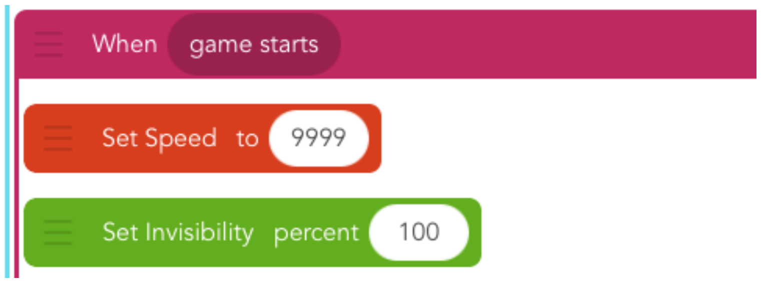
2.2 Add code to drawing object



```
When game starts
    Set Invisibility percent 100
    Draw a Trail color width 3000
        Move Forward 1
    Set Position to x 0 y 150
    Draw a Trail color width 300
        Move Forward 1050
```

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

2.3 Edit drawing object's code to draw faster



```
When game starts
    Set Speed to 9999
    Set Invisibility percent 100
```

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.

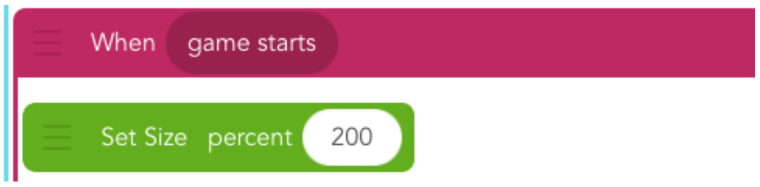
Give the students a few minutes to play their game, and then bring the class together again. Ask for suggestions to make the game more fun and challenging. Like with Crossy Road in Lesson 1, it is boring (and easy!) because it's the same every time! Games are challenging (and fun!) when there is an element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

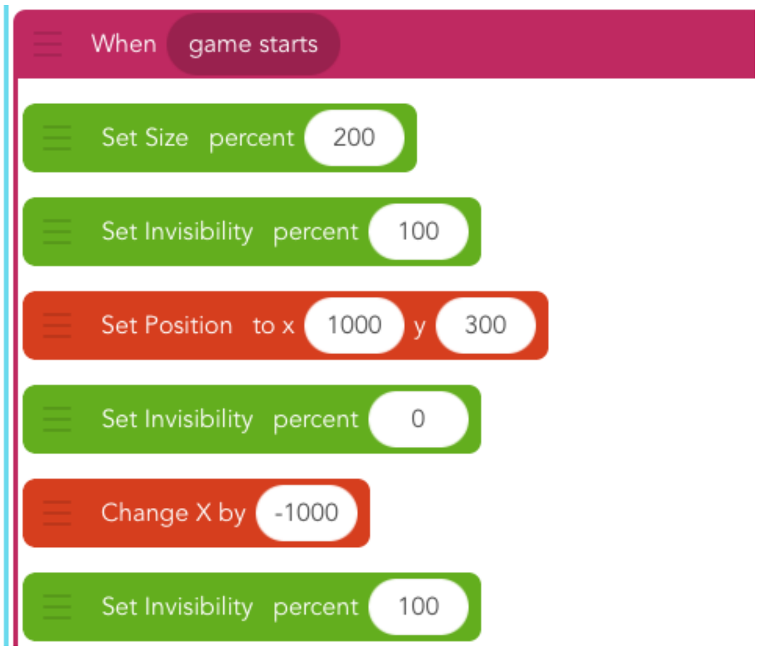
Rename the triangle emoji "Obstacle".

3.2 Add code to obstacle to make it bigger



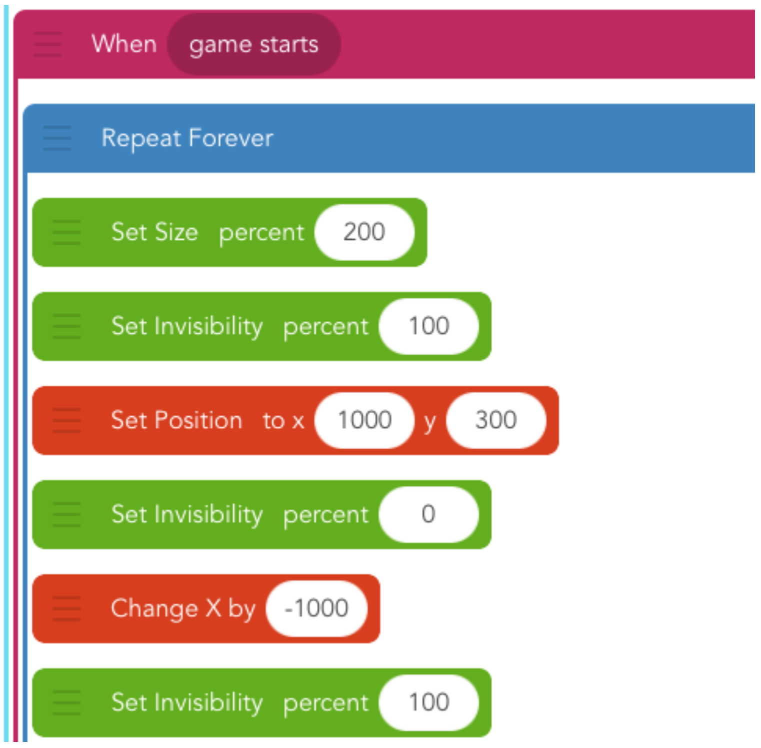
```
When game starts
    Set Size percent 200
```

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



```
When game starts
    Set Size percent 200
    Set Invisibility percent 100
    Set Position to x 1000 y 300
    Set Invisibility percent 0
    Change X by -1000
    Set Invisibility percent 100
```

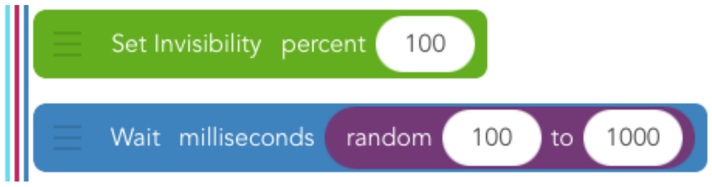
3.4 Edit obstacle's code to make sequence repeat forever



```
When game starts
  Repeat Forever
    Set Size percent 200
    Set Invisibility percent 100
    Set Position to x 1000 y 300
    Set Invisibility percent 0
    Change X by -1000
    Set Invisibility percent 100
```

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)



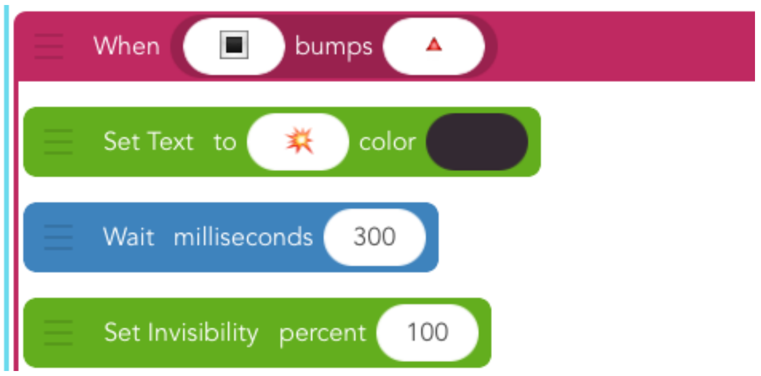
```
Set Invisibility percent 100
Wait milliseconds random 100 to 1000
```

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



```
When ■ bumps ▲
  Set Text to 💥 color ●
  Wait milliseconds 300
  Set Invisibility percent 100
```

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?