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.

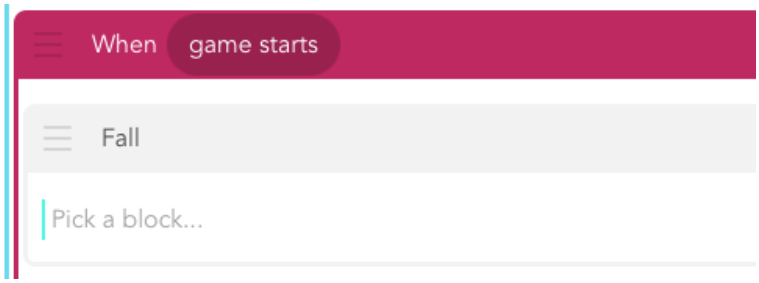
Functions also make your code easier to understand. If anyone wants to look at the code later, they will understand that it, taken together, builds an engine to make the bird fall.

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 psuedocode 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

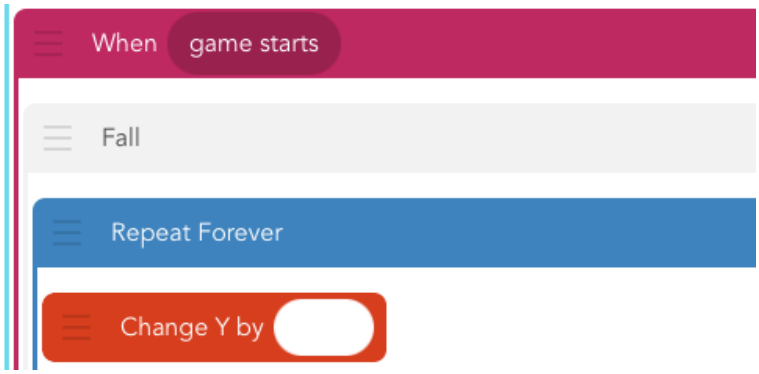
Set bird object to the far left of the screen.

1.2 Add code to bird: create "Fall" function



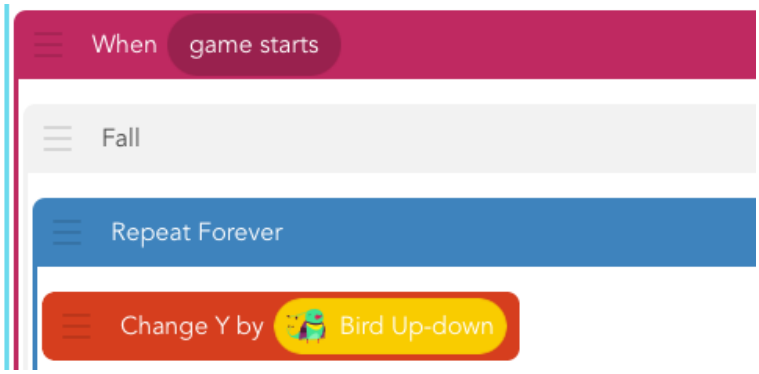
Press New Block to add a new function. Name it something descriptive, like "Fall". All of the code you want to reuse or group as the function should go inside the block. Note that whenever you change a function in one place, it automatically makes those changes wherever else it is used!

1.3 Make bird change Y by [blank] forever



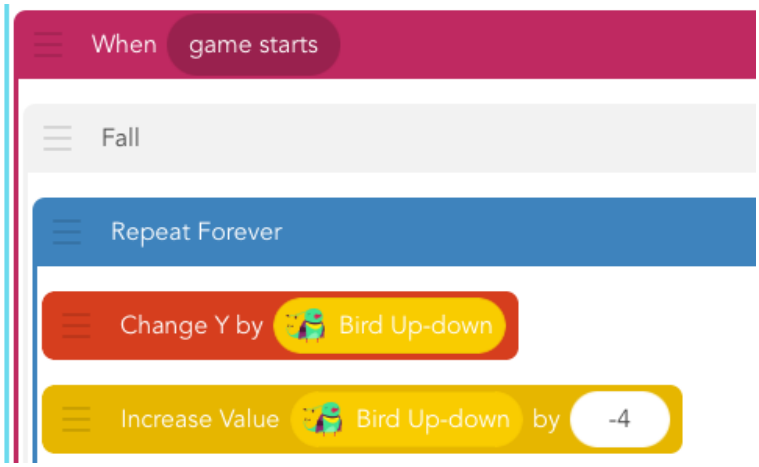
Notice that we don't enter a value in the "Change Y by" block.

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



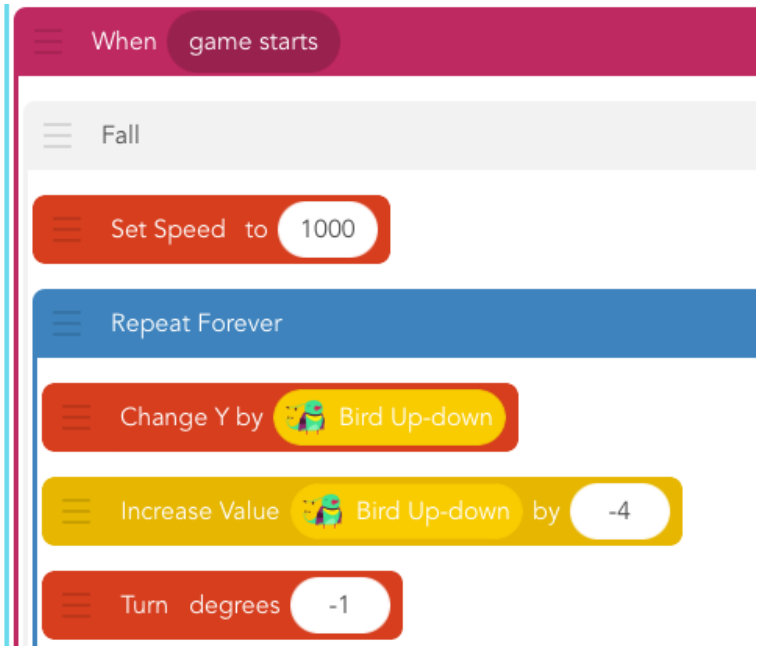
This is the crucial moment where you make the bird's movement change over time by using a value instead of a static number. Tap the bubble in "Change Y by" to access your values.

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)

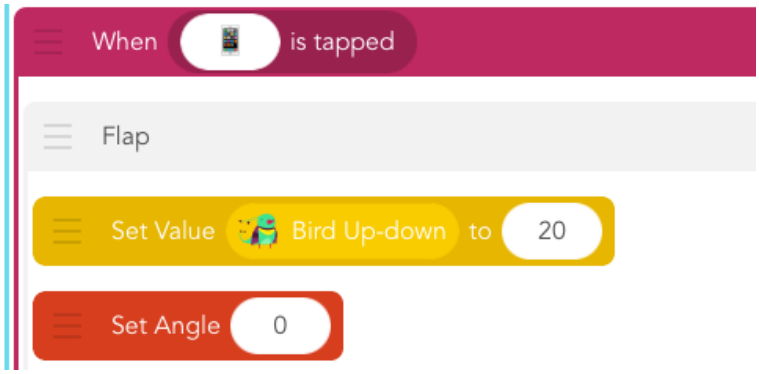
As a class, discuss the second component of the physics engine. At this point, the bird just falls nonstop. We need to add the second component of the physics engine that will allow the player to make the bird flap (and stay afloat) by tapping on the iPad. When the bird flaps, it should move up the screen 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.

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



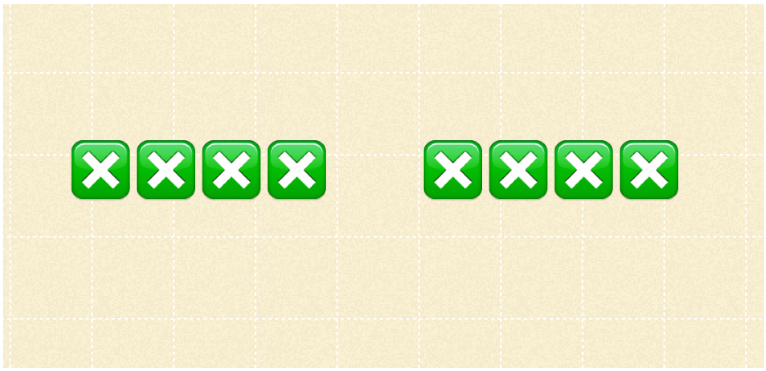
3. Add obstacles (LS) (10 minutes)

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.

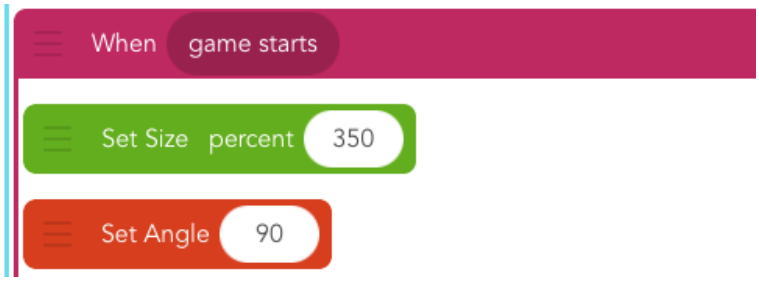
Have your students try to code the pipe sequence independently, then compare with their neighbor. Did everyone decide on the same rules in the same order? Did anyone get identical behavior with different rules? The sequence below is one of many possible solutions.

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

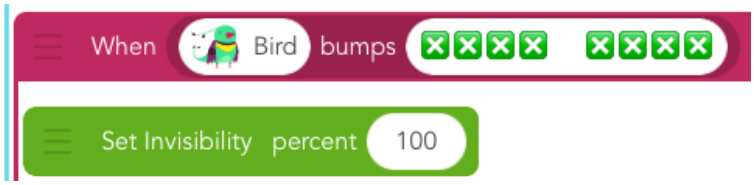


3.3 Add new code to obstacle: Reverse engineer obstacle movement

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.

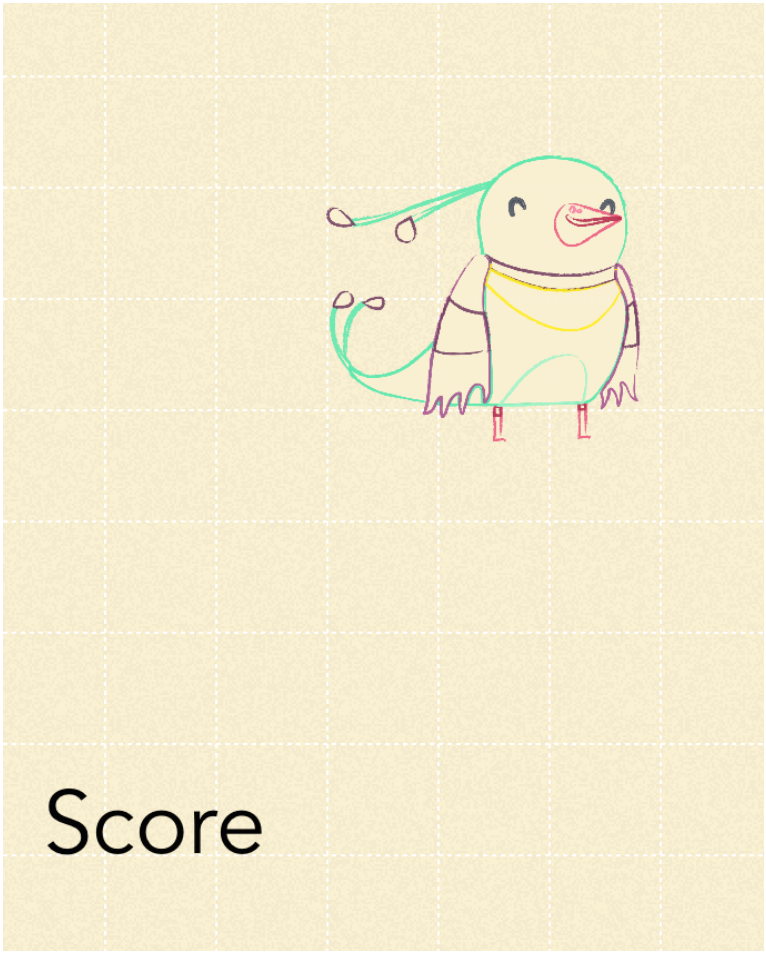
4.1 Add new code to bird and obstacle object



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.

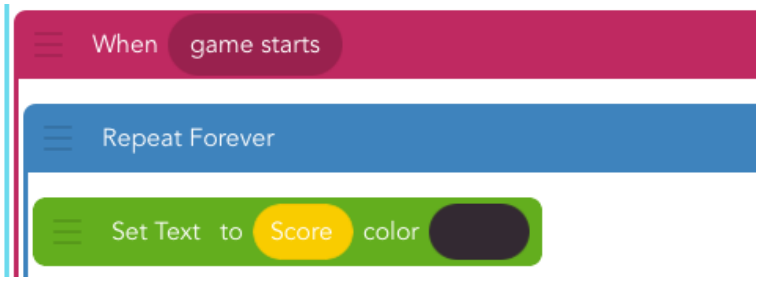
It depends on the pipes either stopping or going invisible when they collide with the bird. An object will both display the current score and detect when the score should be increased! It relies on a mechanism in which the player earns points only when the pipes pass the bird without colliding. Conveniently, 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



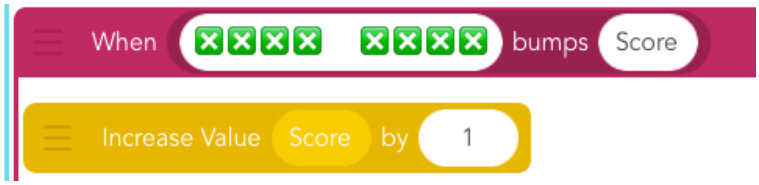
Name score object "Score" and place it in the bottom left corner of the screen, even further left than the bird.

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?