An Amazing Maze Game

Minimum experience: Grades 3+, 1st year using Scratch, 2nd quarter or later

At a Glance

Overview and Purpose

Coders create a player controlled maze game with multiple, custom levels. The purpose of this project is to introduce conditional statements (if blocks) to create player controls, while reinforcing how to use the image editor to design mazes.

Objectives and Standards

Process objective(s):	Product objective(s):
Statement: ● I will learn how to create player controls in Scratch. ● I will learn how to create custom backdrops in Scratch. **Question:** ● How can we create player controls in Scratch? ● How can we create custom backdrops in Scratch?	**Statement:** ● I will create a player controlled maze with custom levels. **Question:** ● How can we create a player controlled maze with custom levels?
Main standard(s):	**Reinforced standard(s):**
1B-AP-10 Create programs that include sequences, events, loops, and conditionals ● Control structures specify the order (sequence) in which instructions are executed within a program and can be combined to support the creation of more complex programs. Events allow portions of a program to run based on a specific action. For example, students could write a program to explain the water cycle and when a specific component is clicked (event), the program would show information about that part of the water cycle. Conditionals allow for the execution of a portion of code in a program when a certain condition is true. For example, students could write a math game that asks multiplication fact questions and then uses a conditional to check whether or not the answer that was entered is correct. Loops allow for the repetition of a sequence of code multiple times. For example, in a program that produces an animation about a famous historical character, students could use a loop to have the character walk across the screen as they introduce themselves. (source)	**1B-AP-08** Compare and refine multiple algorithms for the same task and determine which is the most appropriate. ● Different algorithms can achieve the same result, though sometimes one algorithm might be most appropriate for a specific situation. Students should be able to look at different ways to solve the same task and decide which would be the best solution. For example, students could use a map and plan multiple algorithms to get from one point to another. They could look at routes suggested by mapping software and change the route to something that would be better, based on which route is shortest or fastest or would avoid a problem. Students might compare algorithms that describe how to get ready for school. Another example might be to write different algorithms to draw a regular polygon and determine which algorithm would be the easiest to modify or repurpose to draw a different polygon. (source) **1B-AP-09** Create programs that use variables to store and modify data. ● Variables are used to store and modify data. At this level, understanding how to use variables is sufficient. For example, students may use mathematical

	operations to add to the score of a game or subtract from the number of lives available in a game. The use of a variable as a countdown timer is another example. (source) **1B-AP-11** Decompose (break down) problems into smaller, manageable subproblems to facilitate the program development process. ● Decomposition is the act of breaking down tasks into simpler tasks. For example, students could create an animation by separating a story into different scenes. For each scene, they would select a background, place characters, and program actions. (source) **1B-AP-13** Use an iterative process to plan the development of a program by including others' perspectives and considering user preferences. ● Planning is an important part of the iterative process of program development. Students outline key features, time and resource constraints, and user expectations. Students should document the plan as, for example, a storyboard, flowchart, pseudocode, or story map. (source) **1B-AP-15** Test and debug (identify and fix errors) a program or algorithm to ensure it runs as intended. ● As students develop programs they should continuously test those programs to see that they do what was expected and fix (debug), any errors. Students should also be able to successfully debug simple errors in programs created by others. (source) **1B-AP-16** Take on varying roles, with teacher guidance, when collaborating with peers during the design, implementation, and review stages of program development. ● Collaborative computing is the process of performing a computational task by working in pairs or on teams. Because it involves asking for the contributions and feedback of others, effective collaboration can lead to better outcomes than working independently. Students should take turns in different roles during program development, such as note taker, facilitator, program tester, or "driver" of the computer. (source) **1B-AP-17** Describe choices made during program development using code comments, presentations, and demonstrations. ● People communicate about their code to help others understand and use their programs. Another purpose of communicating one's design choices is to show an understanding of one's work. These explanations could manifest themselves as in-line code comments for collaborators and assessors, or as part of a summative presentation, such as a code walk-through or coding journal. (source)

Practices and Concepts

Source: K–12 Computer Science Framework. (2016). Retrieved from http://www.k12cs.org.

Main practice(s):	Reinforced practice(s):

Practice 5: Creating computational artifacts ● "The process of developing computational artifacts embraces both creative expression and the exploration of ideas to create prototypes and solve computational problems. Students create artifacts that are personally relevant or beneficial to their community and beyond. Computational artifacts can be created by combining and modifying existing artifacts or by developing new artifacts. Examples of computational artifacts include programs, simulations, visualizations, digital animations, robotic systems, and apps." (p. 80) ● **P5.2.** Create a computational artifact for practical intent, personal expression, or to address a societal issue. (p. 80) ● **P5.3.** Modify an existing artifact to improve or customize it. (p. 80)	**Practice 2: Collaborating around computing** ● "Collaborative computing is the process of performing a computational task by working in pairs and on teams. Because it involves asking for the contributions and feedback of others, effective collaboration can lead to better outcomes than working independently. Collaboration requires individuals to navigate and incorporate diverse perspectives, conflicting ideas, disparate skills, and distinct personalities. Students should use collaborative tools to effectively work together and to create complex artifacts." (p. 75) ● **P2.1.** Cultivate working relationships with individuals possessing diverse perspectives, skills, and personalities. (p. 75) **Practice 6: Testing and refining computational artifacts** ● "Testing and refinement is the deliberate and iterative process of improving a computational artifact. This process includes debugging (identifying and fixing errors) and comparing actual outcomes to intended outcomes. Students also respond to the changing needs and expectations of end users and improve the performance, reliability, usability, and accessibility of artifacts." (p. 81) ● **P6.1.** Systematically test computational artifacts by considering all scenarios and using test cases." (p. 81) ● **P6.2.** Identify and fix errors using a systematic process. (p. 81) **Practice 7: Communicating about computing** ● "Communication involves personal expression and exchanging ideas with others. In computer science, students communicate with diverse audiences about the use and effects of computation and the appropriateness of computational choices. Students write clear comments, document their work, and communicate their ideas through multiple forms of media. Clear communication includes using precise language and carefully considering possible audiences." (p. 82) ● **P7.2.** Describe, justify, and document computational processes and solutions using appropriate terminology consistent with the intended audience and purpose. (p. 82)
Main concept(s):	**Reinforced concept(s):**
Control ● "Control structures specify the order in which instructions are executed within an algorithm or program. In early grades, students learn about sequential execution and simple control structures. As they progress, students expand their understanding to combinations of structures that support complex execution." (p. 91) ● **Grade 5 -** "Control structures, including loops, event handlers, and conditionals, are used to specify the	**Algorithms** ● "Algorithms are designed to be carried out by both humans and computers. In early grades, students learn about age-appropriate algorithms from the real world. As they progress, students learn about the development, combination, and decomposition of algorithms, as well as the evaluation of competing algorithms." (p. 91)

flow of execution. Conditionals selectively execute or skip instructions under different conditions." (p. 103)	• **Grade 5 -** "Different algorithms can achieve the same result. Some algorithms are more appropriate for a specific context than others." (p. 103) **Modularity** • "Modularity involves breaking down tasks into simpler tasks and combining simple tasks to create something more complex. In early grades, students learn that algorithms and programs can be designed by breaking tasks into smaller parts and recombining existing solutions. As they progress, students learn about recognizing patterns to make use of general, reusable solutions for commonly occurring scenarios and clearly describing tasks in ways that are widely usable." (p. 91) • **Grade 5 -** "Programs can be broken down into smaller parts to facilitate their design, implementation, and review. Programs can also be created by incorporating smaller portions of programs that have already been created." (p. 104) **Variables** • "Computer programs store and manipulate data using variables. In early grades, students learn that different types of data, such as words, numbers, or pictures, can be used in different ways. As they progress, students learn about variables and ways to organize large collections of data into data structures of increasing complexity." (p. 91) • **Grade 5 -** "Programming languages provide variables, which are used to store and modify data. The data type determines the values and operations that can be performed on that data." (p. 103)

Scratch Blocks	
Primary blocks	Event, Control, Motion, Looks
Supporting blocks	Data, Operators, Sound

Vocabulary	
Algorithm	• A step-by-step process to complete a task. (source) • A formula or set of steps for solving a particular problem. To be an algorithm, a set of rules must be unambiguous and have a clear stopping point. (source)
Debugging	• The process of finding and correcting errors (bugs) in programs. (source) • To find and remove errors (bugs) from a software program. Bugs occur in programs when a line of code or an instruction conflicts with other elements of the code. (source)
Event (trigger)	• An action or occurrence detected by a program. Events can be user actions, such as clicking a mouse button or pressing a key, or system occurrences, such as running out of memory. Most modern applications, particularly those that run in Macintosh and Windows environments, are said to be event-driven,because they are designed to respond to events. (source) • The computational concept of one thing causing another thing to happen. (source) • Any identifiable occurrence that has significance for system hardware or software. User-generated events include keystrokes and mouse clicks; system-generated events include

	program loading and errors. (source)
Loop	• A programming structure that repeats a sequence of instructions as long as a specific condition is true. (source) • In a loop structure, the program asks a question, and if the answer requires an action, it is performed and the original question is asked again until the answer is such that the action is no longer required. For example, a program written to compute a company's weekly payroll for each individual employee will begin by computing the wages of one employee and continue performing that action in a loop until there are no more employee wages to be computed, and only then will the program move on to its next action. Each pass through the loop is called an iteration. (source) • The computational concept of running the same sequence multiple times. (source)
Modularity	• The characteristic of a software/web application that has been divided (decomposed) into smaller modules. An application might have several procedures that are called from inside its main procedure. Existing procedures could be reused by recombining them in a new application (source)
Scripts	• One or more Scratch blocks connected together to form a sequence. Scripts begin with an event block that responds to input (e.g., mouse click, broadcast). When triggered, additional blocks connected to the event block are executed one at a time. (source)
Variable	• A symbolic name that is used to keep track of a value that can change while a program is running. Variables are not just used for numbers; they can also hold text, including whole sentences (strings) or logical values (true or false). A variable has a data type and is associated with a data storage location; its value is normally changed during the course of program execution. (source) • Variables play an important role in computer programming because they enable programmers to write flexible programs. Rather than entering data directly into a program, a programmer can use variables to represent the data. Then, when the program is executed, the variables are replaced with real data. This makes it possible for the same program to process different sets of data. (source)

Connections

Integration	Math, media arts
Vocations	There are a wide range of careers in game development that involve coding. For example, coding character movement, player controls, particle and game physics, random world or object generators, sound synthesis, game engines and tools, localization, performance and server optimization, etc. Click here to visit a website dedicated to exploring potential careers through coding.

Resources

- Example project
- Video walkthroughs
- Quick reference guides
- Project files

Project Sequence

Preparation (20+ minutes)

Suggested preparation	Resources for learning more

Customizing this project for your class (10+ minutes): Remix the project example to include your own player controlled maze game.
 	
(10+ minutes) Read through each part of this lesson plan and decide which sections the coders you work with might be interested in and capable of engaging with in the amount of time you have with them. If using projects with sound, individual headphones are very helpful.
	• BootUp Scratch Tips
	  ○ Videos and tips on Scratch from our Google+ community
	• BootUp Facilitation Tips
	  ○ Videos and tips on facilitating coding classes from our Google+ community
	• Scratch Starter Cards
	  ○ Printable cards with some sample starter code designed for beginners
	• ScratchEd
	  ○ A Scratch community designed specifically for educators interested in sharing resources and discussing Scratch in education
	• Scratch Help
	  ○ This includes examples of basic projects and resources to get started
	• Scratch Videos
	  ○ Introductory videos and tips designed by the makers of Scratch
	• Scratch Wiki
	  ○ This wiki includes a variety of explanations and tutorials

Getting Started (6-15+ minutes)

Suggested sequence	Resources, suggestions, and connections
1. Review and demonstration (2+ minutes): Begin by asking coders to talk with a neighbor for 30 seconds about something they learned last time; assess for general understanding of the practices and concepts from the previous project. Explain that today we are going to create a player controlled maze game. Display and demonstrate the sample project (or your own remixed version).	**Practices reinforced:** • Communicating about computing **Video:** Project Preview (1:11) **Video:** Lesson pacing (1:48) This can include a full class demonstration or guided exploration in small groups or individually. For small group and individual explorations, you can use the videos and quick reference guides embedded within this lesson, and focus on facilitating 1-on-1 throughout the process. **Example review discussion questions:** • What's something new you learned last time you coded? ○ Is there a new block or word you learned? • What's something you want to know more about? • What's something you could add or change to your previous project? • What's something that was easy/difficult about your previous project?
2. Discuss (3+ minutes): Have coders talk with each other about how they might create a project like the one demonstrated. If coders are unsure, and the discussion questions aren't helping, you can model thought processes: "I noticed the sprite moved around, so I think they used a motion block. What motion block(s) might be in the code? What else did you notice?" After the discussion, coders will begin working on their project as a class, in small groups, or at their own pace.	**Practices reinforced:** • Communicating about computing **Note:** Discussions might include full class or small groups, or individual responses to discussion prompts. These discussions which ask coders to predict how a project might work, or think through how to create a project, are important aspects of learning to code. Not only does this process help coders think logically and creatively, but it does so without giving away the answer.

	Example discussion questions: ● What would we need to know to make something like this in Scratch? ● What kind of blocks might we use? ● What else could you add or change in a project like this? ● What code from our previous projects might we use in a project like this? ● What kind of sprites might we see in a maze? ○ What kind of code might they have? ● How could we use a keyboard to control a character? ● How could we switch to a different level when we reach a goal? ● What could happen if we touch a wall?
3. Logging in (1-10+ minutes): If not yet comfortable with logging in, review how to log into Scratch and create a new project. If coders continue to have difficulty with logging in, you can create cards with a coder's login information and store it in your desk. This will allow coders to access their account without displaying their login information to others.	**Alternative login suggestion:** Instead of logging in at the start of class, another approach is to wait until the end of class to log in so coders can immediately begin working on coding; however, coders may needs a reminder to save before leaving or they will lose their work. **Why the variable length of time?** It depends on comfort with login usernames/passwords and how often coders have signed into Scratch before. Although this process may take longer than desired at the beginning, coders will eventually be able to login within seconds rather than minutes. **What if some coders log in much faster than others?** Set a timer for how long everyone has to log in to their account (e.g., 5 minutes). If anyone logs in faster than the time limit, they can open up previous projects and add to them. Your role during this time is to help out those who are having difficulty logging in. Once the timer goes off, everyone stops their process and prepares for the following chunk.

Project Work (80+ minutes; 3+ classes)	
Suggested sequence	**Resources, suggestions, and connections**
4. Creating levels (25+ minutes, or an entire class) ***5+ minute demonstration*** Click on the stage icon and open the Backdrops tab. Pick a starting location for our sprite, then demonstrate how to use various drawing tools to create a maze with one color. Think out loud how you want to make sure there is enough room for a sprite to move through the maze without touching a wall. Draw a "goal" by choosing a new color and drawing with it at the end of the first level (e.g., a square). Ask coders why all of our walls are one color and our goal is a different color. They may realize this makes it easier for users to figure out what the goal is in their level, and it will make it easier to code by allowing us to determine if we touch a wall or a goal.	**Practices reinforced:** ● Testing and refining computational artifacts ● Creating computational artifacts **Video:** Creating levels (3:03) **Quick Reference Guide:** Click here **Video:** Image editor: Bitmap mode (3:38) **Video:** Image editor: Vector mode (4:31) **Video:** Image editor: Extra tools (4:12) **Facilitation tip:** If you're not comfortable figuring out how to take into account several wall or goal colors, remind coders for this project they need one color for their walls and one color for their goal. If it's not the same across every level, it

Quickly demonstrate one more level, but point out you want to have the sprite start in the same location, so don't put a wall over that location. ***20+ minutes to create custom levels and 1-on-1 facilitating*** Give coders time to create at least three levels using the image editor tools. Encourage peer-to-peer assistance and facilitate 1-on-1 as needed. If coders finish their three levels early, encourage them to add even more, assist others, or walk around to get ideas by looking at other coders' levels.	will make coding a little more difficult (but, certainly possible). In addition, encourage coders to keep the same starting location for every level for the same reasons as above. We could have different starting locations for each level, but that makes it more complicated. **Suggested questions:** ● Where will you sprite's starting location be? (make sure it's the same location for each level) ● Where else might you put your goal? ● Will you make the levels get progressively harder? ● What other shapes could you use to change the way the levels look?
5. Player controls (10+ minutes) ***5 demonstration*** Choose one of the four options presented in the video (4:32) and quick reference guide, then demonstrate how to create player controls for the sprite that will navigate through the maze. Although we could make the controls more complicated, we only need to make it so our sprite can move up, down, left, and right. ***10+ minutes to code their player controls and 1-on-1 facilitating*** Leave your code on the screen and give coders time to create or copy their player controls, then test them out to make sure they're working properly. Encourage peer-to-peer assistance and facilitate 1-on-1 as needed.	**Standards reinforced:** ● **1B-AP-08** Compare and refine multiple algorithms for the same task and determine which is the most appropriate ● **1B-AP-10** Create programs that include sequences, events, loops, and conditionals **Practices reinforced:** ● Testing and refining computational artifacts ● Creating computational artifacts **Concepts reinforced:** ● Algorithms ● Control **Video:** Player controls (4:32) **Quick reference guide:** Click here **Facilitation tip:** Although there are four options presented, the first option would work best for younger coders because it removes the need to understand XY directions. The third option with the forever loop is the best method for responsive player controls, and can be copied by third graders if the code is left on the screen. The fourth option with the or blocks can make things a little complicated for younger coders. **Suggested questions:** ● How could you make this maze two players? ○ What would you need to change in the code to make it two players? ○ What would be similar?
6. Restart function (5+ minutes) ***2 demonstration*** Explain we want to create a restart function that will allow us make Scratch Cat return to the starting position when touching a wall or the goal. Review how to use broadcast and receive message blocks to create a function. Remind the class we need to drag Scratch Cat to the starting position before using our go to block or the X and Y coordinates will be off. ***3+ minutes to code their restart function and 1-on-1 facilitating***	**Standards reinforced:** ● **1B-AP-10** Create programs that include sequences, events, loops, and conditionals **Practices reinforced:** ● Testing and refining computational artifacts ● Creating computational artifacts **Concepts reinforced:** ● Algorithms ● Control ● Modularity

Leave your code on the screen and give coders time to create or copy their restart function. Encourage peer-to-peer assistance and facilitate 1-on-1 as needed.	**Video:** Restart function (1:05) **Quick reference guide:** Click here **Note:** If you leave your code on the screen for copying, point out their numbers for the go to block might be different than yours.
7. Don't touch the walls (10+ minutes) ***5 demonstration*** Explain we want to be able to check to see if our sprite touches a wall. If the sprite touches a wall, it will restart to the starting position. Demonstrate how to do this and make note of how to use the touching color blocks (click on the color, move your mouse to the color you want, click again to save the new color). ***5+ minutes to code their project and 1-on-1 facilitating*** Leave your code on the screen and give coders time to add the code to their project. Encourage peer-to-peer assistance and facilitate 1-on-1 as needed.	**Standards reinforced:** ● **1B-AP-10** Create programs that include sequences, events, loops, and conditionals **Practices reinforced:** ● Testing and refining computational artifacts ● Creating computational artifacts **Concepts reinforced:** ● Algorithms ● Control **Video:** Don't touch the walls (1:43) **Quick reference guide:** Click here **Note:** If you leave your code on the screen for copying, point out their touching color block will be the color of their walls, not your walls. I know it seems obvious, but this will likely come up with younger coders. **Potential unplugged lesson:** You could engage in a discussion or lesson on conditionals (if statements) by working through one of the unplugged lessons on conditionals; for example, Conditionals with Cards.
8. Goooooaaaaallllllll (10+ minutes) ***5 discussion and demonstration*** Give a minute or two to see if coders can figure out how to use the wall code for our goal. Explain we want to be able to check to see if our sprite touches our goal so we can switch to the next level. If the sprite touches our goal, it will restart to the starting position and switch to the next backdrop. Demonstrate how to do this and point out the two switch backdrop blocks we need to add (one for switching to the next backdrop when we reach our goal and one for setting our starting backdrop). ***5+ minutes to code their project and 1-on-1 facilitating*** Leave your code on the screen and give coders time to add the code to their project. Encourage peer-to-peer assistance and facilitate 1-on-1 as needed.	**Standards reinforced:** ● **1B-AP-10** Create programs that include sequences, events, loops, and conditionals **Practices reinforced:** ● Communicating about computing ● Testing and refining computational artifacts ● Creating computational artifacts **Concepts reinforced:** ● Algorithms ● Control **Video:** Goooooaaaaallllllll (2:11) **Quick reference guide:** Click here **Note:** If you leave your code on the screen for copying, point out their touching color block will be the color of their goal, not your goal. I know it seems obvious, but this will likely come up with younger coders. **Potential unplugged lesson:** You could engage in a discussion or lesson on conditionals (if statements) by working through one of the unplugged lessons on conditionals; for example, Conditionals with Cards.

	Advanced Note: We can stack multiple if/else blocks inside the "else" portion of the block to check several possible conditions (each "if" block) and execute one default (the final "else" block). Click here for an example where the text changes depending on how close you are to the goal, and click here to see the code in context.
9. Play testing (20+ minutes, or an entire class) ***5+ minute play testing*** Either in pairs or in small groups, give coders a few minutes to take turns trying out each other's maze games and discussing how they used code and the image editor tools to create their mazes. ***5+ minutes to revise their project and 1-on-1 facilitating*** Give coders five or so minutes to revise their projects based on feedback and ideas they gathered from their peers. Encourage peer-to-peer assistance and facilitate 1-on-1 as needed. ***I recommend repeating this process several more times to encourage sharing ideas and getting peer feedback***	**Standards reinforced:** ● **1B-AP-10** Create programs that include sequences, events, loops, and conditionals ● **1B-AP-13** Use an iterative process to plan the development of a program by including others' perspectives and considering user preferences ● **1B-AP-16** Take on varying roles, with teacher guidance, when collaborating with peers during the design, implementation, and review stages of program development **Practices reinforced:** ● Collaborating around computing ● Communicating about computing ● Testing and refining computational artifacts ● Creating computational artifacts **Concepts reinforced:** ● Algorithms ● Control **Facilitation tip:** It may help to model the kind of feedback one might give to a game like this. To practice this, display the maze I created for this lesson or one of the mazes I previously created in this studio. Ask coders what's something they like about the project, what they might be curious about, and what suggestions they might have for improving the project(s). **Note:** When testing out specific levels, a quick way to get to the desired level is to use code to switch to the specific backdrop when the green flag is clicked.
10. Adding in comments (*the amount of time depends on typing speed and amount of code*): ***1 minute demonstration*** When the project is nearing completion, bring up some code for the project and ask coders to explain to a neighbor how the code is going to work. Review how we can use comments in our program to add in explanations for code, so others can understand how our programs work. Quickly review how to add in comments. ***Commenting time*** Ask coders to add in comments explaining the code throughout their project. Encourage coders to write clear and concise comments, and ask for clarification or elaboration when needed.	**Standards reinforced:** ● **1B-AP-17** Describe choices made during program development using code comments, presentations, and demonstrations **Practices reinforced:** ● Communicating about computing **Concepts reinforced:** ● Algorithms **Video:** How to add in comments (1:34) **Quick reference guide:** Click here **Facilitation suggestion:** One way to check for clarity of comments is to have a coder read out loud their comment and ask another coder to recreate their comment using code blocks. This may be a fun challenge for those who type fast while others are completing their comments.

Assessment

Standards reinforced:

- **1B-AP-17** Describe choices made during program development using code comments, presentations, and demonstrations

Practices reinforced:

- Communicating about computing

Although opportunities for assessment in three different forms are embedded throughout each lesson, this page provides resources for assessing both processes and products. If you would like some example questions for assessing this project, see below:

Summative Assessment *of* Learning	**Formative** Assessment *for* Learning	**Ipsative** Assessment *as* Learning
The debugging exercises, commenting on code, and projects themselves can all be forms of summative assessment if a criteria is developed for each project or there are "correct" ways of solving, describing, or creating. **For example, ask the following after a project:** ● Can coders debug the debugging exercises? ● Did coders create a project similar to the project preview? ○ **Note:** The project preview and sample projects are not representative of what all grade levels should seek to emulate. They are meant to generate ideas, but expectations should be scaled to match the experience levels of the coders you are working with. ● Did coders use a variety of block types in their algorithms and can they explain how they work together for specific purposes? ● Did coders include descriptive comments for each event in all of their sprites? ● Can coders explain how else they might use an "if" conditional block? ● Can coders explain the different affordances and constraints of the various ways for creating user controls?	The 1-on-1 facilitating during each project is a form of formative assessment because the primary role of the facilitator is to ask questions to guide understanding; storyboarding can be another form of formative assessment. **For example, ask the following while coders are working on a project:** ● What are three different ways you could change that sprite's algorithm? ● What happens if we change the order of these blocks? ● What could you add or change to this code and what do you think would happen? ● How might you use code like this in everyday life? ● See the suggested questions throughout the lesson and the assessment examples for more questions.	The reflection and sharing section at the end of each lesson can be a form of ipsative assessment when coders are encouraged to reflect on both current and prior understandings of concepts and practices. **For example, ask the following after a project:** ● How is this project similar or different from previous projects? ● What new code or tools were you able to add to this project that you haven't used before? ● How can you use what you learned today in future projects? ● What questions do you have about coding that you could explore next time? ● See the reflection questions at the end for more suggestions.

• Did coders create a maze game with at at least ## different levels and player controls for the character? ○ Choose a number appropriate for the coders you work with and the amount of time available.		

Extended Learning

Project Extensions

Suggested extensions	Resources, suggestions, and connections
<u>Roguelike challenge (2+ minutes)</u> ***1 minute demonstration*** Roguelike games are a difficult genre of games with permanent repercussions for mistakes. For example, if your character dies in a roguelike, you might have to start the entire game over. Demonstrate how we can turn our maze into a roguelike by adding in one block of code when the sprite touches a wall. ***1+ minutes to code their player controls and 1-on-1 facilitating*** Leave your code on the screen and give coders time to add the switch backdrop to block in their code (if they want to). Encourage peer-to-peer assistance and facilitate 1-on-1 as needed.	**Standards reinforced:** • **1B-AP-10** Create programs that include sequences, events, loops, and conditionals **Practices reinforced:** • Testing and refining computational artifacts • Creating computational artifacts **Concepts reinforced:** • Algorithms • Control **Video:** Roguelike challenge (1:32) **Quick reference guide:** Click here
<u>Adding variables (advanced) (15+ minutes)</u> ***5 minute demonstration*** Demonstrate how to create a variable for keeping track of the number of restarts and a variable that displays how to show the level. Point out that we need to hold shift and click on the display to show a large display of the variable. ***10+ minutes to code their variables and 1-on-1 facilitating*** Leave your code on the screen and give coders time to create new variables and add the various data blocks to their project (if they want to). Encourage peer-to-peer assistance and facilitate 1-on-1 as needed.	**Standards reinforced:** • **1B-AP-09** Create programs that use variables to store and modify data • **1B-AP-10** Create programs that include sequences, events, loops, and conditionals **Practices reinforced:** • Testing and refining computational artifacts • Creating computational artifacts **Concepts reinforced:** • Algorithms • Control • Variables **Video:** Adding variables (2:54) **Quick reference guide:** Click here **Suggested questions:** • What other variables could we use in a maze game? • How could we keep track of how long it takes someone to get through a maze?

	• Could you figure out how to create a low or high score? • Could you make it so it displays how long it takes to clear a level? • What other variables could we keep track of?
Cleaning up with functions (10+ minutes) ***5 minute demonstration*** Pull up your code with the forever loop and ask the class to figure out what the three sections of the forever loop are (movement, checking if touching a wall, and checking if touching the goal). Tell the class we can make this much easier to read by putting each of these sections into their own functions (message blocks) with a descriptive name. Demonstrate how to do this with each of the three sections and make note we need to use broadcast message and wait blocks in order to make sure it doesn't move on to the next algorithm until the previous algorithm is completed. ***5+ minutes to clean up their code and 1-on-1 facilitating*** Leave your code on the screen and give coders time to clean up their code. Encourage peer-to-peer assistance and facilitate 1-on-1 as needed.	**Standards reinforced:** • **1B-AP-10** Create programs that include sequences, events, loops, and conditionals **Practices reinforced:** • Testing and refining computational artifacts • Creating computational artifacts **Concepts reinforced:** • Algorithms • Control • Modularity **Video:** Cleaning up with functions (3:12) **Quick reference guide:** Click here **Suggested questions:** • How do functions and comments help you understand the parts of a project? ○ What's the difference between the two? • What other projects could you make easier to read by creating functions for each part of the code? • When shouldn't you use functions and just keep a longer algorithm?
Adding even more (30+ minutes, or at least one class): If time permits and coders are interested in this project, encourage coders to explore what else they can create in Scratch by trying out new blocks and reviewing previous projects to get ideas for this project. When changes are made, encourage them to alter their comments to reflect the changes (either in the moment or at the end of class). While facilitating this process, monitor to make sure coders don't stick with one feature for too long. In particular, coders like to edit their sprites/backgrounds by painting on them or taking photos, or listen to the built-in sounds in Scratch. It may help to set a timer for creation processes outside of using blocks so coders focus their efforts on coding.	**Standards reinforced:** • **1B-AP-10** Create programs that include sequences, events, loops, and conditionals **Practices reinforced:** • Testing and refining computational artifacts • Creating computational artifacts **Concepts reinforced:** • Algorithms • Control **Facilitation Suggestion:** Some coders may not thrive in inquiry based approaches to learning, so we can encourage them to use the Tips Section and Block Help to get more ideas for their projects; however, we may need to remind coders the suggestions provided by Scratch are not specific to our projects, so it may create some unwanted results unless the code is modified to match our own intentions. **Suggested questions:** • What else can you do with Scratch? • What do you think the other blocks do? a. Can you make your project do ____? • What other sprites can you add to your project? • What have you learned in other projects that you could use in this project? • Can you add more user control than demonstrated? • How else might you use "if" conditional blocks in your project?

	• Could you add in other sprites as enemies or power ups? • Could you make it so you have to collect items to unlock a door that allows you to escape a maze? • Could you make your game multiplayer?
Similar projects: Have coders explore the code of other peers in their class, or on a project studio dedicated to this project. Encourage coders to ask questions about each other's code. When changes are made, encourage coders to alter their comments to reflect the changes (either in the moment or at the end of class). Watch this video (3:20) if you are unsure how to use a project studio.	**Standards reinforced:** • **1B-AP-10** Create programs that include sequences, events, loops, and conditionals • **1B-AP-12** Modify, remix, or incorporate portions of an existing program into one's own work, to develop something new or add more advanced features **Practices reinforced:** • Testing and refining computational artifacts **Concepts reinforced:** • Algorithms **Resource:** Studio with mazes I created (some are very complicated) **Note:** Coders may need a gentle reminder we are looking at other projects to get ideas for our own project, *not to simply play around*. For example, "look for five minutes," "look at no more than five other projects," "find three projects that each do one thing you would like to add to your project," or "find X number of projects that are similar to the project we are creating." **Generic questions:** • What are some ways you can expand this project beyond what it can already do? • How is this project similar (or different) to something you worked on today? • What blocks did they use that you didn't use? a. What do you think those blocks do? • What's something you like about their project that you could add to your project? • How might we add player controls to this project? • How might you use "if" conditional blocks in this project? • If the project is not a game, could you turn this project into a game? • If the project is a game, could you turn it into a different kind of game?

Differentiation

Less experienced coders	More experienced coders
Demonstrate the example remix project or your own version, and walk through how to experiment changing various parameters or blocks to see what they do. Give some time for them to change the blocks around. When it appears a coder might need some guidance or has completed an idea, encourage them to add more to the project or begin following the steps for creating the project on their own (or with BootUp resources). Continue to facilitate one-on-one using questioning techniques to encourage tinkering and trying new combinations of code.	Demonstrate the project without showing the code used to create the project. Challenge coders to figure out how to recreate a similar project without looking at the code of the original project. If coders get stuck reverse engineering, use guiding questions to encourage them to uncover various pieces of the project. Alternatively, if you are unable to work with someone one-on-one at a time of need, they can access the quick reference guides and video walkthroughs above to learn how each part of this project works.

If you are working with other coders and want to get less experienced coders started with remixing, have those who are interested in remixing a project watch this video (1:57) to learn how to remix a project.	If you are working with other coders and want to get more experienced coders started with reverse engineering, have those who are interested watch this video (2:16) to learn how to reverse engineer a project.

Debugging Exercises (1-5+ minutes each)

Debugging exercises	Resources and suggestions
Why don't we switch to the next level when we touch the goal (the green rectangle)? ● The "touching color" block needs to be green, not teal Why does Scratch Cat move to the right instead of the left when we press the left arrow? ● We need to change x by -5, not +5 Why do we stay on level 1 even when we reach the goal? ● We need to switch to the next backdrop, not the backdrop labeled "Level 1" Even more debugging exercises	**Standards reinforced:** ● **1B-AP-15** Test and debug (identify and fix errors) a program or algorithm to ensure it runs as intended **Practices reinforced:** ● Testing and refining computational artifacts **Concepts reinforced:** ● Algorithms ● Control **Suggested guiding questions:** ● What should have happened but didn't? ● Which sprite(s) do you think the problem is located in? ● What code is working and what code has the bug? ● Can you walk me through the algorithm (steps) and point out where it's not working? ● Are there any blocks missing or out of place? ● How would you code this if you were coding this algorithm from Scratch? ● Another approach would be to read the question out loud and give hints as to what types of blocks (e.g., motion, looks, event, etc.) might be missing. **Reflective questions when solved:** ● What was wrong with this code and how did you fix it? ● Is there another way to fix this bug using different code or tools? ● *If this is not the first time they've coded:* How was this exercise similar or different from other times you've debugged code in your own projects or in other exercises?

Unplugged Lessons and Resources

Although each project lesson includes suggestions for the amount of class time to spend on a project, BootUp encourages coding facilitators to supplement our project lessons with resources created by others. In particular, reinforcing a variety of standards, practices, and concepts through the use of unplugged lessons. Unplugged lessons are coding lessons that teach core computational concepts without computers or tablets. You could start a lesson with a short, unplugged lesson relevant to a project, or use unplugged lessons when coders appear to be struggling with a concept or practice.

Master list of 100+ unplugged lessons and resources

Incorporating unplugged lessons in the middle of a multi-day project situates understandings within an actual project; however, unplugged lessons can occur before or after projects with the same concepts. **An example for incorporating unplugged lessons:**

Lesson 1. Getting started sequence and beginning project work

Lesson 2.	Continuing project work
Lesson 3.	Debugging exercises and unplugged lesson that reinforces concepts from a project
Lesson 4.	Project extensions and sharing

Reflection and Sharing

Reflection suggestions

Coders can either discuss some of the following prompts with a neighbor, in a small group, as a class, or respond in a physical or digital journal**.** If reflecting in smaller groups or individually, walk around and ask questions to encourage deeper responses and assess for understanding. Here is a sample of a digital journal designed for Scratch (source) and here is an example of a printable journal useful for younger coders.

Sample reflection questions or journal prompts:

- What's something we learned while working on this project today?
  - What are you proud of in your project?
  - How did you work through a bug or difficult challenge today?
- What other projects could we do using the same concepts/blocks we used today?
- What's something you had to debug today, and what strategy did you use to debug the error?
- What mistakes did you make and how did you learn from those mistakes?
- How did you help other coders with their projects?
  - What did you learn from other coders today?
- What questions do you have about coding?
  - What was challenging today?
- Why are comments helpful in our projects?
- How is this project similar to other projects you've worked on?
  - How is it different?
- Why are keyboard keys easier to use for player controls than buttons?
- What's the difference between user controls and player controls?
- What other projects could you turn into a game?
- How else might you use an "if" conditional block?
- More sample prompts

Sharing suggestions

Standards reinforced:

- **1B-AP-17** Describe choices made during program development using code comments, presentations, and demonstrations

Practices reinforced:

- Communicating about computing
- Fostering an inclusive culture

Concepts reinforced:

- Algorithms
- Control
- Modularity
- Program development

Peer sharing and learning video: Click here (1:33)
At the end of class, coders can share with each other something they learned today. Encourage coders to ask questions about each other's code or share their journals with each other. When sharing code, encourage coders to discuss something they like about their code as well as a suggestion for something else they might add.

Publicly sharing Scratch projects: If coders would like to publicly share their Scratch projects, they can follow these steps:

1. Sharing Your Project (2:35)
   a. Quick reference guide
2. **(Advanced)** Creating a thumbnail (3:26)
   a. Quick reference guide

An Amazing Maze Game

Coder Resources

Project Sequence

(complete each step before moving to the next)

1. **Sign in and create a new project**
2. **Create levels**
    a. **Additional resources:**
        i. **Video:** Image editor: Bitmap mode (3:38)
        ii. **Video:** Image editor: Vector mode (4:31)
        iii. **Video:** Image editor: Extra tools (4:12)
3. **Create player controls**
4. **Create a restart function**
5. **Detect the walls**
6. **Create a goooooaaaaalllllllll**
7. **Add in comments**

Project Extensions

(pick and choose extensions that sound interesting)

1. **Create a roguelike challenge**
2. **Add variables (Advanced)**
3. **Clean up your code with functions**
4. **Share your project**
5. **Create a thumbnail**
6. **Learn even more Scratch tips**

Debugging Exercises

(practice your debugging skills by solving these bugs)

1. Why don't we switch to the next level when we touch the goal (the green rectangle)?
2. Why does Scratch Cat move to the right instead of the left when we press the left arrow?
3. Why do we stay on level 1 even when we reach the goal?
4. Even more debugging exercises

Example Project and Files

(use these resources to see the original project, learn how to remix the project, or to challenge yourself)

1. **Project**: Example project
2. **Video:** Project Preview (1:11)
3. **Video**: Remixing a project (1:57)
4. **Video**: How to reverse engineer a project (2:16)