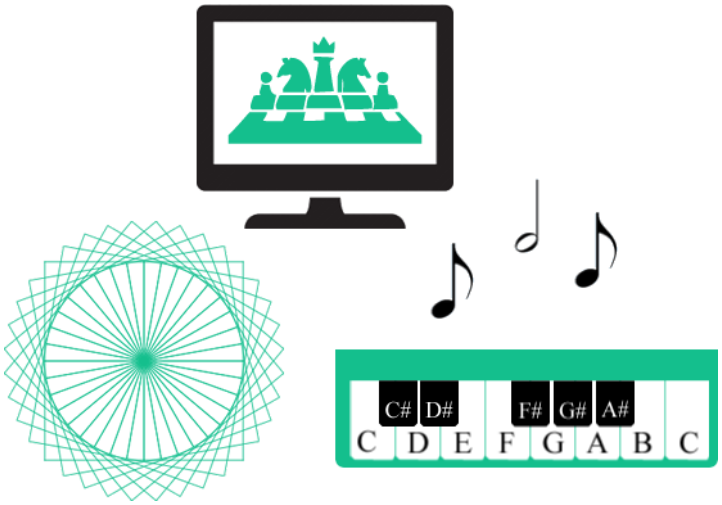
AN INTRODUCTION TO COMPUTATIONAL THINKING THROUGH ART, MUSIC, AND GAMES



Glen Bull, Joe Garofalo, and N. Rich Nguyen

An Introduction to Computational Thinking
through Art, Music, and Games

Editors

Glen Bull, Joe Garofalo, and N. Rich Nguyen

Technical Editor

Jo Watts

Art Consultant

Sheila Cochran

Published by

The Society for Information Technology and Teacher Education (SITE)

ISBN 978-1-9397-9746-9

TABLE OF CONTENTS

An Introduction to Computational Thinking
through Art, Music, and Games

Editors

Glen Bull, Joe Garofalo, and N. Rich Nguyen

Technical Editor

Jo Watts

Art Consultant

Sheila Cochran

Published by

The Society for Information Technology and Teacher Education (SITE)

ISBN 978-1-9397-9746-9

TABLE OF CONTENTS

Introduction

Seymour Papert (1980) introduced the term *computational thinking* in *Mindstorms: Children, Computers, and Powerful Ideas* (p. 182). In an article, "Teaching Children Thinking," that preceded *Mindstorms*, Papert (1970) described his goal of developing educational environments that facilitate use of the computer as a computational object:

> I claim that computation is by far the richest known source of these ingredients. We can give children unprecedented power to invent and carry out exciting projects by providing them with access to computers, with a suitably clear and intelligible programming language and with peripheral devices capable of producing on-line real-time action. (Papert, 1970, p. 2)

Renewed interest in computational thinking was stimulated by an essay in the *Communications of the ACM* written by Jeanette Wing while she was assistant director of the National Science Foundation. Funding provided by the National Science Foundation to encourage integration of computational thinking into the curriculum has yielded results. In 2017, the Commonwealth of Virginia adopted standards requiring that all schools must teach computer science and computational thinking, including computer coding. Virginia's Standards of Learning specify that these concepts must be integrated into content courses at the elementary level and taught as stand-alone courses in middle school and high school (Sawchuk, 2017). Many other states are also requiring computer science at all levels of school or are considering such requirements. The *Code.org* (2018) annual report on *The State of K-12 Computer Science* provides an overview of the current status.

In conversations with teachers, principals, and central office personnel in Virginia's schools, we have found a consensus that integration of computational thinking with subjects such as reading and mathematics will require that these subjects be taught without requiring much more additional class time than otherwise would be needed. Papert's vision of computational thinking embodied the idea that the act of programming a computer can facilitate the process of learning subjects such as mathematics and language arts. However, integrating computational thinking with other subjects requires expertise in both computational thinking and the subject taught.

This book, *Introduction to Computational Thinking*, provides eight examples of ways in which this might be done: two examples in mathematics, two examples in language arts, two examples in social studies, and two examples in science. These initial examples are designed as a starting point rather than an end point. They are designed to stimulate thinking and conversation about ways in which computational thinking might be integrated with subjects taught in schools.

The book is designed to accompany a parallel course, *Introduction to Computational Thinking*, that will be taught at several pilot sites in the coming year. Feedback from teachers and children who explore these activities will be incorporated into a revised edition of this book. Other planned revisions will be to include more open-ended activities and creative extensions. One goal of this effort is to engage diverse audiences in computational thinking. Therefore the activities selected are based on themes such as art, music, and games that are engaging to a wide range of students.

We also would like to be explicit about our goals for the introduction of computational thinking. We are inspired by Brian Harvey, co-developer of the educational programming language "*Snap!*".

> Languages in the Logo family, including Scratch and Snap*!*, take the position that we're not in the business of training professional computer programmers. Our mission is to bring programming to the masses. This has historically led to

> all sorts of informal language: "reporter" instead of "function" (whose CS usage is technically incorrect anyway, to a mathematician), "input" instead of "argument," and so on. Technical jargon is valuable for specialists, but is also a barrier to entry for non-experts. *Retrieved from:*
>
> https://scratch.mit.edu/discuss/topic/4455/?page=164

We have adopted a similar perspective in this work. The goal is to introduce children to ways in which computers can be used to explore school subjects in interesting and engaging ways. This goal is different than the requirements for an introductory computer science course for college students.

That said, we should note that the programming language, *Snap!*, has also been used as the basis for an introductory course for college students, The *Beauty and Joy of Computing*, at the University of California at Berkeley. *Snap!* has a low threshold – it has successfully been used by students in the elementary grades – and a high ceiling; it is also used by college students and adults.

The introduction to the *Snap!* user guide acknowledges the inspiration provided by Scratch and the Lifelong Kindergarten group directed by Mitch Resnick at MIT:

> The brilliant design of Scratch, from the Lifelong Kindergarten Group at the MIT Media Lab, is crucial to Snap!. Our earlier version, BYOB, was a direct modification of the Scratch source code. Snap! is a complete rewrite, but its code structure and its user interface remain deeply indebted to Scratch. And the Scratch Team, who could have seen us as rivals, have been entirely supportive and welcoming to us. (*Snap! Reference Manual*, page 3)

The examples in this book are illustrated in *Snap!* but due to similarities in the two languages, many of these examples could also be created in *Scratch* with minor revisions. However, *Snap!* also has some capabilities that be used to interact with tools such as *SoundScope* that we have developed to support exploration of sound and music. We would like to gratefully acknowledge the support and encouragement that we have received from *Snap!* developers Jens Moenig and Brian Harvey in our efforts to establish connections between *Snap!* and these tools.

This material is based upon work supported by National Science Foundation STEM-C grant # 159457. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors.

References

Papert, S. (1970). Teaching children thinking. *Proceedings of the World Conference on Computer Education*. Amsterdam, NE: International Federation for Information Processing.

Papert, S. (1971). *Teaching children thinking* (MIT Artificial Intelligence Laboratory Memo No. 2247). http://hdl.handle.net/1721.1/5835.

Papert, S. (1980). *Mindstorms, children, computers, and powerful ideas*. New York, NY: Basic Books.

Sawchuk, S. (2017, November 2). Virginia becomes the first state to require computer science instruction. *Education Week.* Retrieved from http://blogs.edweek.org/edweek/curriculum/2017/11/virginia_mandates_computer_science_learning.html

Tabesh, Y. (2017). Computational thinking: A 21st century skill. *Olympiads in Informatics, 11*, 65–70. doi: 10.15388/ioi.2017.special.10

Wing, J.M. (2006). Computational thinking. *Communications of the ACM*, *49*(3), 33–35.

Development Team

An *Introduction to Computational Thinking* is being piloted in schools in Maine, South Carolina, Colorado, and Virginia. The development team includes members with expertise in computer science and in subject areas that include science, mathematics, language arts, social studies, engineering, art, and music.

- Glen Bull, Professor of Education, Curry School of Education, University of Virginia
- Joe Garofalo, Make to Learn Lab, Curry School of Education, University of Virginia
- Jim Cohoon, Assistant Chair, Department of Computer Science, University of Virginia
- Rich Nguyen, Assistant Professor, Department of Computer Science, University of Virginia
- Bill Ferster, Research Professor, Science, Humanities, and Arts Initiative, University of Virginia
- Jo Watts, Manager, Make to Learn Laboratory, University of Virginia
- Sally (Luotong) Yao, Manager, Maker Studio, Curry School of Education
- James Rutter, Director, Fab Lab, Haystack Mountain School of Crafts, Maine
- Michael Littman, Professor of Mechanical Engineering, Princeton University
- Alan Grier, Director, Industrial Electricity and Electronics, Midlands Technical College
- Eric Stein, Research Assistant, Department of Computer Science, University of Virginia
- Katelyn Bosma, Research Assistant, Midlands Technical College
- Christiana Taylor, Post-doctoral Research Associate, University of Northern Colorado
- James Shank, Research Assistant, Blue Ridge Community College
- Natasha Heny, Associate Professor - English Education, University of Virginia
- Paula Cochran, Language Arts Consultant, Kirksville, Missouri
- Sheila Cochran, Retired Special Education Teacher and Art Consultant

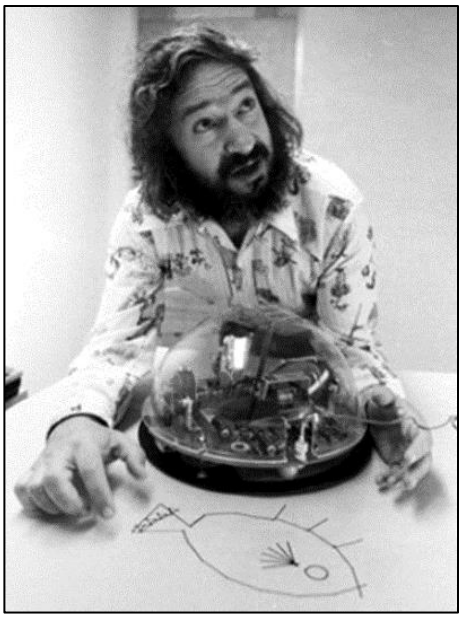
Chapter 1

Digital Designs and Artistic Patterns: Introduction to Turtle Graphics

Glen Bull and Joe Garofalo

Introduction

Robot Turtles



Seymore Papert with a Logo Turtle (credit: Cynthia Solomon, MIT)

Seymour Papert developed *Logo*, the first computing language designed for children, in the MIT Artificial Intelligence Laboratory in 1966. An MIT team led by Mitch Resnick subsequently developed a web-based successor to *Logo* named *Scratch*. With support from the MIT team, a version of *Scratch* named *Snap!* was designed for adults by a team at the University of California at Berkeley led by Brian Harvey working in collaboration with developer Jens Moenig.

Papert's vision of computers in schools embodied the idea that *the act of programming a computer can facilitate the process of learning subjects such as mathematics and language arts.* Papert believed that children can learn by teaching the computer through programming.

The robot turtle is the best-known feature of *Logo*. Papert described the *Logo* turtle as a "computational object-to-think-with." The early *Logo* turtles consisted of Plexiglas hemispheres that could be programmed to move about the floor. Later turtles included a pen that could be lowered to draw figures on rolls of paper.

Screen Turtles

The advent of video displays made it possible to create screen turtles in *Logo*. These turtles sometimes looked like an actual turtle (viewed from above) but often were represented by a triangle on the screen.

Examples of Screen Turtles

Securing a *Snap!* Account

A free account for *Snap!* can be obtained at the following web address:

https://snap.berkeley.edu/snap

Secure a *Snap!* account before continuing. Try each command as you read about it. For a comprehensive guide to *Snap!* commands, please refer to the online manual at the following web address:

https://snap.berkeley.edu/snap/help/SnapManual.pdf

Turtle Commands

The Command Palette

The types of commands available in *Snap!* are displayed in a *Command Palette* at the top of the left-hand side of the screen. For example, the *Motion* commands are currently highlighted in the palette below. Other categories of commands include *Looks*, *Sound*, *Pen*, *Control*, *Sensing*, *Operators*, and *Variables*. Note that each category is a different color. For example, *Motion* commands are purple.

Click on each of the categories in the *Command Palette* to get a sense of the types of commands that are found under each category. Use commands by dragging them into the script space to the right of the *Command Palette*.

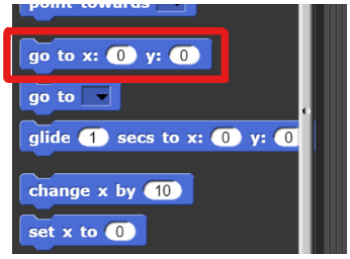
The Move Command

Drag the command **Move 10 Steps** the script space. Then click the code block. The turtle should move 10 steps forward (in the direction that the turtle is pointed) when this code block is clicked.

Try other values such as **100 Steps**. In this case, only one turtle is on the screen. However, it is possible to create multiple turtles. The term *Sprite* is also used as another name for a screen turtle.

Resetting the Turtle

If the turtle went off the screen in the last section, reset its position by clicking on the **Go To X_Y_** code block. You will use this frequently and may want to drag the command block into a corner of the command space for easy access.



The Turn Command

Then try the **Turn** command. Drag the **Turn Right 15 Degrees** code block into the script space. Enter the setting of 90 degrees into the code block.

Turn Right 90 Degrees

Click on the code block to execute the command. Watch the turtle rotate 90 degrees when the command is executed.

This example shows the *Turn Right* command. A *Turn Left* command is also available.

Combining Code Blocks

Combine the **Move** and **Turn** commands can by snapping the two code blocks together. Click any part of the combined code block to cause both commands to be executed in the order that they are listed (even though the order of the execution may not be obvious at this stage).

The name *Snap!* refers to the ability to snap code blocks together in the same manner as LEGO bricks. Practice snapping and unsnapping blocks together to become familiar with the mechanics of *Snap!* command blocks.

Pen Commands

The original floor turtle had a pen in its belly that could be raised and lowered. In a similar manner, **Pen Up** and **Pen Down** commands (found in the green *Pen* palette) enable the screen turtle to draw on the screen. Drag the **Pen Down** command into the script space. Click the **Pen Down** code block, and then click the **Move/Turn** block four times

Clearing the Command Space

Click the **Clear** command to erase any lines you have made. You will use **Clear** frequently and may want to drag the command block into a corner of the command space for easy access.

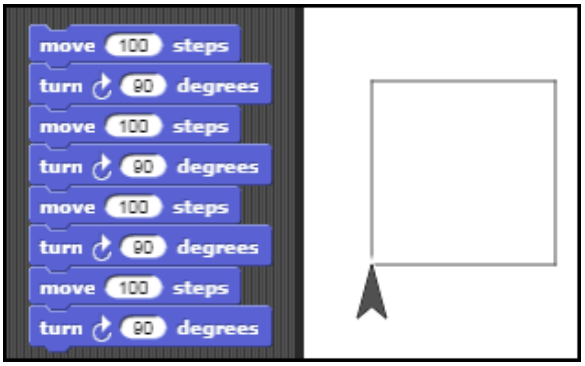
Repeating Commands

Repeating Commands to Draw a Square

Right clicking on any code block in the script space allows you to duplicate it. To duplicate a group of blocks, right click on the topmost block and click *Duplicate*.

Use the *Duplicate* option to create four copies of the **Move** and **Turn** commands.

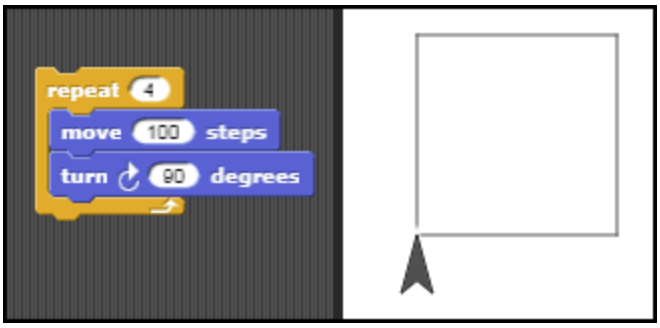
With the pen down, repeat the steps **Move 100 Steps** and **Turn 90 Degrees** four times to draw a square.



Then **Clear** the screen.

The Repeat Command

Use the **Repeat** statement to execute the commands with fewer lines of code. The **Repeat** command is found in the *Control* palette (highlighted in yellow). To use it, drag the command blocks you want to repeat into the empty space of the **Repeat** block and enter the number of times you want them to repeat.



The command **Repeat 4 [Move 100 Turn 90]** achieves the same result as duplicating the **Move** and **Turn** commands four times.

Defining New Commands

Teaching the Turtle a New Word

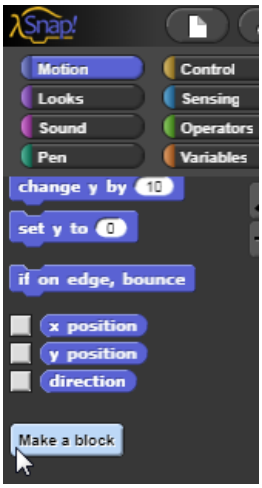
Papert's book, *Mindstorms: Children, Computers, and Powerful Ideas*, describes the way in which subjects such as mathematics can be introduced through turtle graphics:

> "The idea of programming is introduced through the metaphor of teaching the Turtle a new word. This is simply done, and children often begin their programming experience by programming the turtle to respond to new commands invented by the child …"

The family of programming languages that includes *Logo*, *Scratch*, and *Snap!* was designed for children to learn by adding their own extensions to the programming language.

The Make a Block Option

In *Snap!*, the **Make a Block** option is used to "teach the Turtle a new word." This option is found at the bottom of each palette of commands.



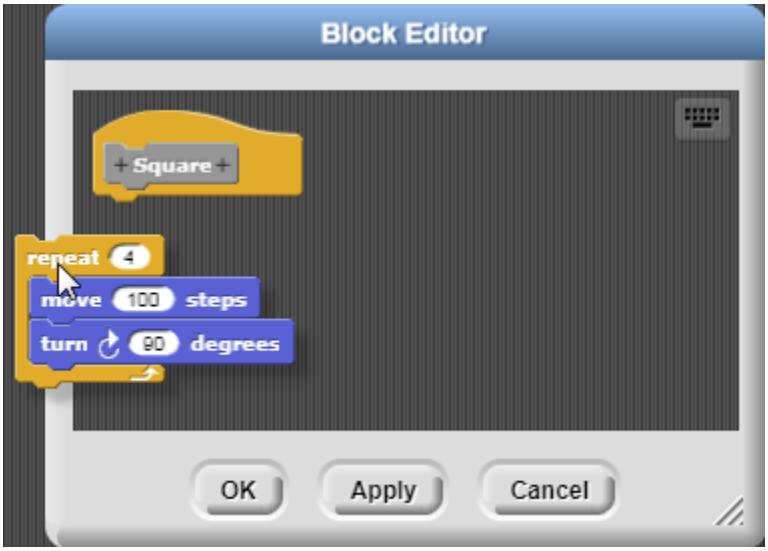
Click the *Make a Block* button to define a new command. Enter *Square* as the name of the new command. In most cases, the "for all sprites" option will be selected so that the new command will work with any sprite. Then click OK.

Defining a New Command - Square

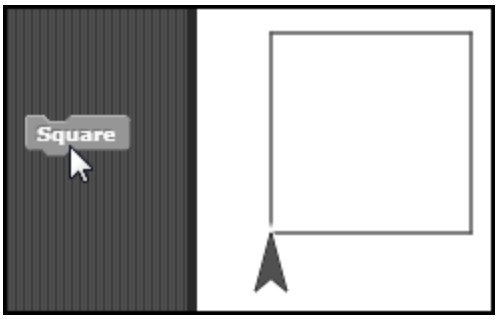
Next drag the previously developed block of code,

Repeat 4 [Move 100 Steps; Turn 90 Degrees]

into the *Block Editor* to define a new command named **Square**.



Click OK. A new command, **Square**, will appear at the bottom of the list of *Motion* blocks. It can now be used as though it were a built-in command.



Abstraction – A Key Computing Concept

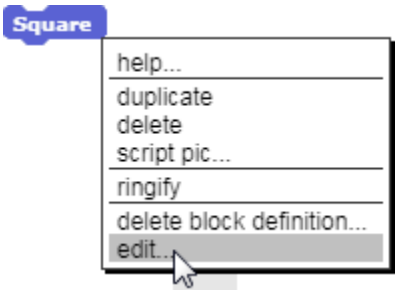
Creation of new commands is one of the most powerful concepts in computing. It makes it possible to work out complex sequences of commands and then assign a name to the code block. This process as known as *abstraction*. The underlying complexity of a long sequence of actions can then be hidden. Assuming the name of the new command is meaningful (and it is important to assign a name that accurately reflects the result that will occur), the programmer no longer has to deal with the underlying details.

The limited capacity of human memory imposes one of the key constraints on development of efficient code. Assigning a name to a complex process, thereby hiding the underlying details, makes it possible to create much more complex programs than otherwise would be possible.

Variables

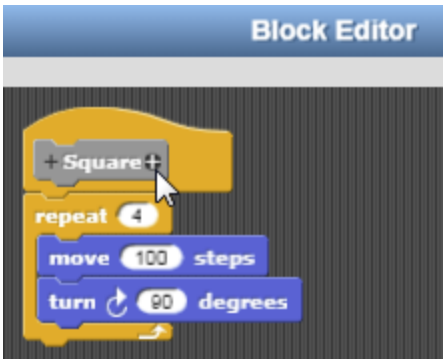
Editing the Square Procedure

The initial **Square** command is somewhat limited. It can only draw a square that is 100 steps on a side. Edit the **Square** command by right-clicking the name of the procedure. This produces a drop-down menu. Click the *Edit* option to edit the procedure.



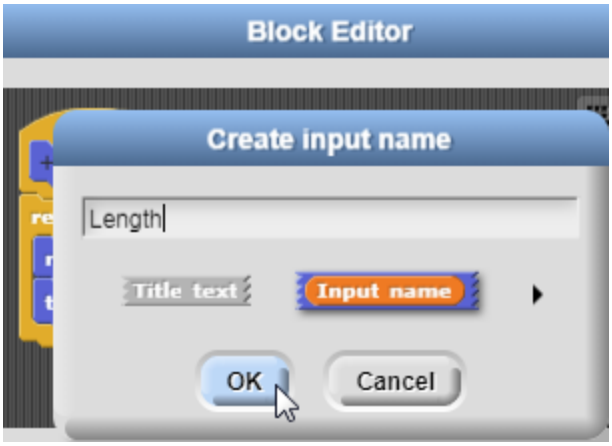
Adding an Input to the Procedure

Click the "Plus" sign (+) to the right of the title ***Square*** in the *Block Editor* to access another option, *Create Input Name*.



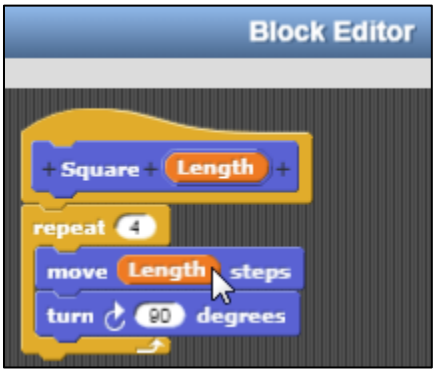
Naming the Variable - Size

Enter the name *Length* as the name of an input to the command. This name was chosen because we are going to use this variable to vary the length that the turtle travels as it completes each side of the square. Click OK to confirm this choice.

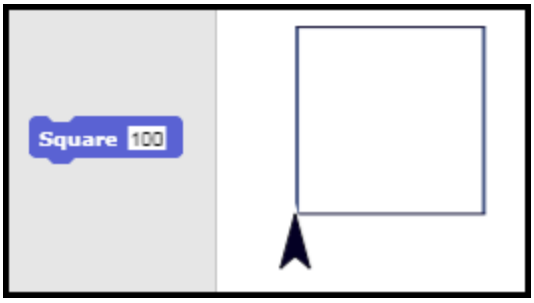


Using the Size Variable in the Square Procedure

Drag the orange oval labeled ***Length*** into the the *step* field of the **Move** command replacing the number.

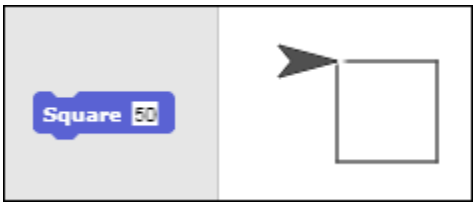


The input to the command **Square** can then be used to specify the length of each side of the square that will be drawn.



Drawing Squares of Varying Size

If 100 is entered as the input, a square that is 100 turtle steps on each side will be drawn. If 50 is entered as the input, a square that is 50 turtle steps on each side will be drawn.



Variables – A Key Computing Concept

The input to the command **Square** is known as a *variable* because it makes it possible to vary the size of the square drawn. Different values can be assigned to the variable known as *Length*. *Variables* are the second big idea in computing. Variables allow the actions of a procedure to be adjusted according to the needs of the situation.

Together, the ability to define new commands and create variables give computing languages much of their power. They enhance the ability of the programmer to create complex applications.

Global Variables

The Make a Variable Option

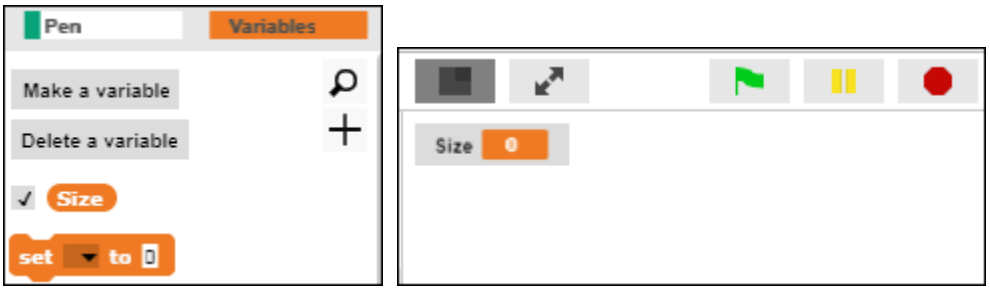
The variable ***Length*** only affects the block of code within the **Square** procedure. It is called a local variable because it *only* affects the local code within the **Square** procedure.

Global Variables can be used *across many* procedures and can also be created. To create a global variable, select the *Make a Variable* option found in the *Variable* command palette (orange). Variables created in this way can be used across multiple blocks of code. For example, a global variable could be used to specify the size of a square and a triangle. It could even be used to control creation of a series of squares of varying size.

The Variable Name Editor

Select the *Make a Variable* to access the *Variable Name* editor. Enter the name ***Size*** as the name of the new global variable. This variable will be used to vary the size of different polygons such as triangles and squares.

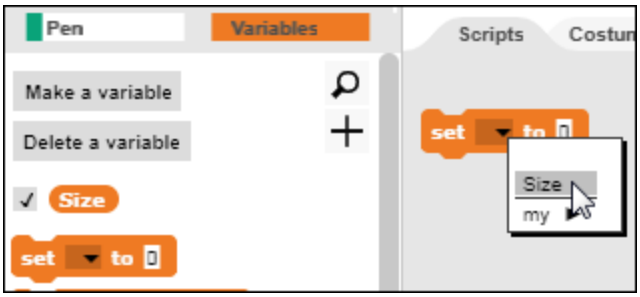
The new global variable, ***Size***, will appear at the top of the *Variables* palette



When the box beside the variable is checked, the value of the variable will be displayed on the stage (on the right-hand side of the screen). A default value of zero is assigned to new variables.

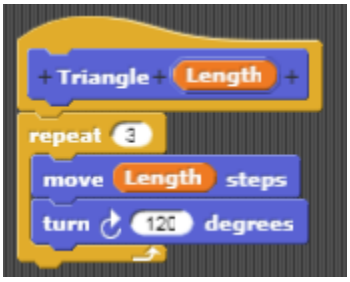
Assigning a Value to a Variable

Use the **Set** command to assign a value to a global variable. The **Set** command is found under the *Variables* palette.

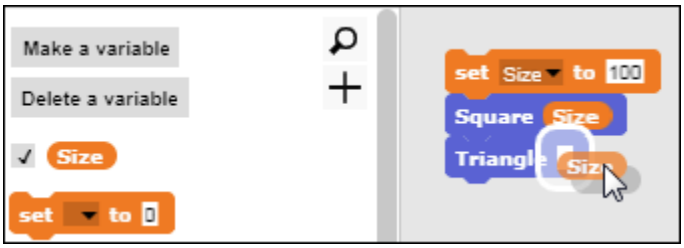


Enter a value of 100 for the global variable size.

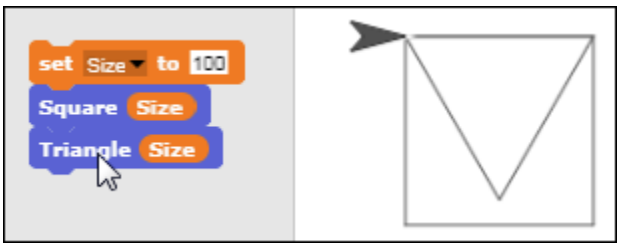
Define a **Triangle** procedure. Enter a value of 3 in the **Repeat** statement and a value of 120 degrees for the angle of each turn. Why does 120 degrees achieve the desired result?



Use the global variable *Size* to control the size of both a square and the triangle. Drag the orange oval labeled Size into the inputs of the **Square** and the **Triangle** procedures.

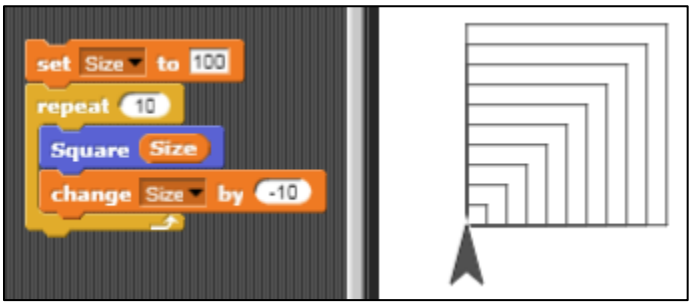


Then click the code block to draw a square and a triangle that are each 100 steps on a side. Experiment with other values for *Size*.



Repeating a Series of Squares

Combine the *Size* variable with a **Repeat** command and the **Square** procedure to create a series of squares. After each square is drawn, the value of the variable is decreased by 10 using the **Change** command. A series of squares decreasing in size results.



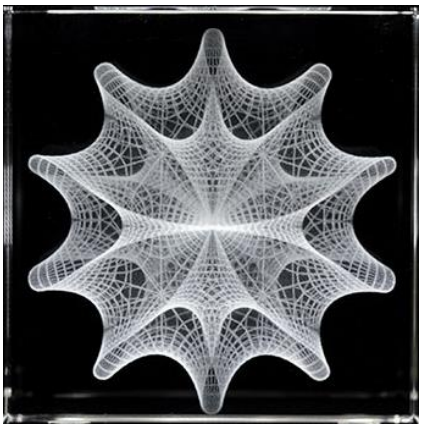
Algorithmic Thinking – A Key Computing Concept

Algorithmic thinking is the process of automating solutions through a series of ordered steps (sequencing) and conditional logic (flow of control), and is an essential characteristic of computing. Developing complex algorithms, such as the one above to draw a series of shapes, is an important skill to develop. *Snap!* provides a simple platform for constructing such algorithms by connecting and sequencing the command blocks. In a later section, we will explore conditional logic and controlling the flow of an algorithm in more depth.

Patterns in Art

Digital Artists

Artists like Bathsheba Grossman use mathematical patterns to create art. Their designs are etched in acrylic or glass using digital fabrication tools such as laser cutters.



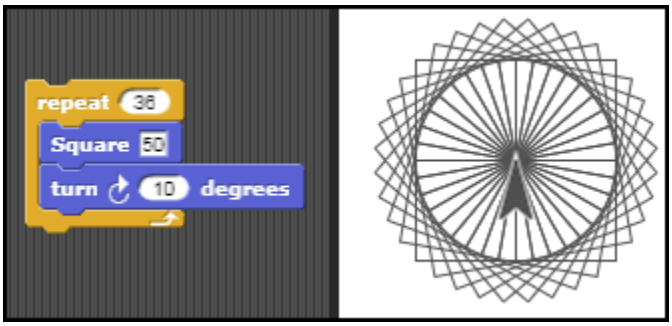
Bathsheba Sculpture Website

Spinning the Square to Create a Pattern

Create a pattern similar to those designed by digital artists. Begin by drawing a series of squares, turning slightly (10 degrees) before drawing each square in the series.

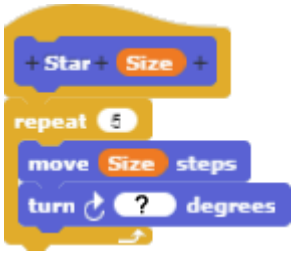
Repeat 36 [Square 50; Turn 10 Degrees]

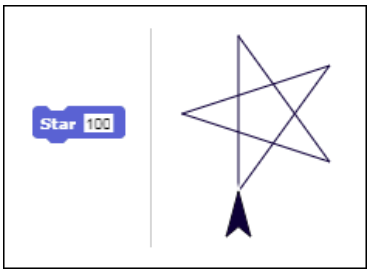
Rotating the turtle as it draws a series of squares results in the following pattern.



Drawing a Star

Use other geometric figures to create other patterns. Begin with a star. A five-pointed star requires a **Repeat** statement with five repetitions. Use experimentation or logic to determine the number of degrees that the turtle must turn.

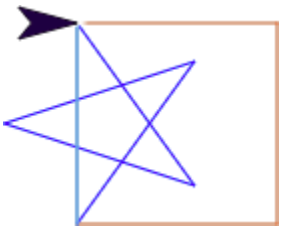




Try two or three different angles for the *Star* procedure. If you have not determined the correct angle after severl tries, consider the following.

Identifying the Correct Angle for the Star Procedure

The turtle turns 90 degrees to draw a square . It must be turned more than 90 degrees to create a star. But how much farther?

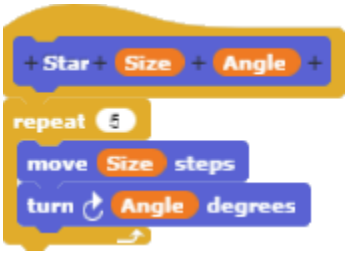


If the turtle does not turn far enough after drawing each line, the figure of the star will not be closed.

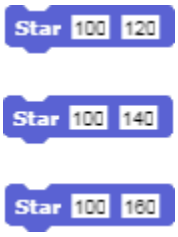
If the turtle turns too far after drawing each line, it will overshoot, creating a star that also is not well formed.

An Iterative Approach

Create a *Star* procedure with Size and Angle inputs.



Then try several angles such as 120 degrees, 140 degrees, and 160 degrees.



Which of the three values (120, 140 or 160 degrees) most nearly resembles a star? What is the most appropriate angle and why?

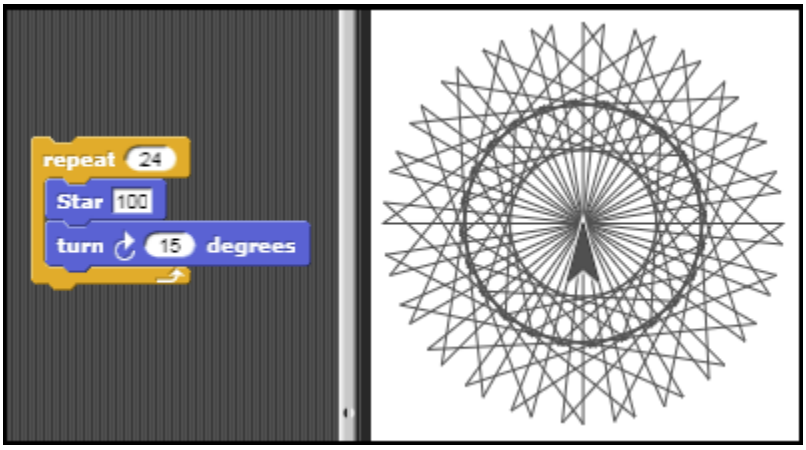
Debugging – A Key Computing Concept

Debugging is the process of identifying and correcting for mistakes, and is another essential to develop in computing. Just as a writer will find and correct grammatical mistakes in a rough draft, a coder will troubleshoot and debug their program. In the above example, we applied a bruteforce

approach to identifying the correct values to use in the **Star** command. Debugging is a process done frequently and occurs at almost every step in the development of a program.

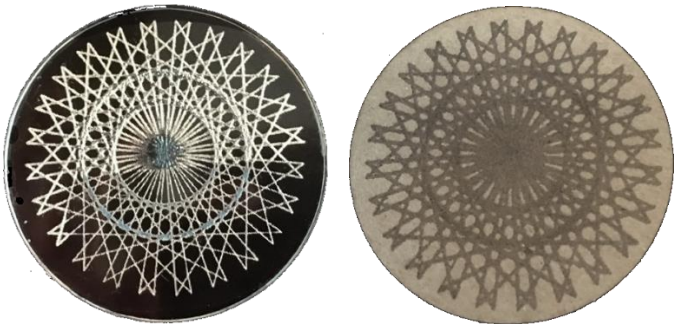
A Snowflake Pattern

Use a Star procedure to create a snowflake pattern. Draw a star; then turn the turtle a few degrees before drawing another star. Repeat until a pattern emerges.



Fabricating Art with Patterns

Tools such as inkjet printers, laser cutters, and other fabrication tools can be used to translate digital designs into ornaments, jewelry, and sculptures in the same manner as digital artists.



Print your design with an ink-jet printer. If you have access to a laser cutter or 3D printer, use the pattern to create a three-dimensional ornament.

CHAPTER 2

Digital Storybooks

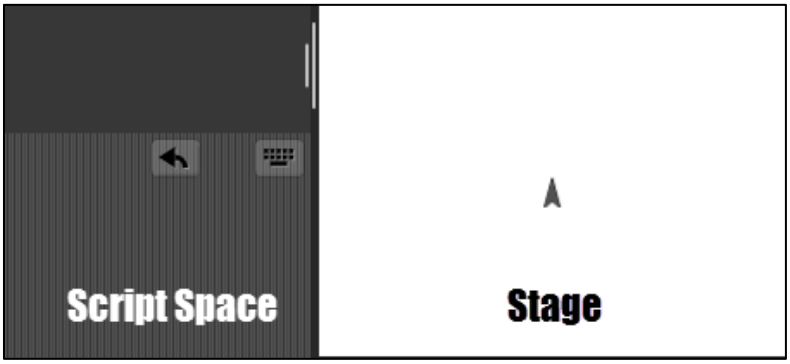
Glen Bull, Sheila Cochran, and Natasha Heny

Computing languages like *Logo*, *Scratch*, and *Snap!* are ideally suited for storytelling. Logo, the first computing language designed specifically for children, included many features for exploring with language and words. Scratch, the web-based successor to Logo, includes many of the same capabilities. Scratch is designed for young children. *Snap!*, developed with the support of the Scratch development team, provides advanced features for exploration of language.

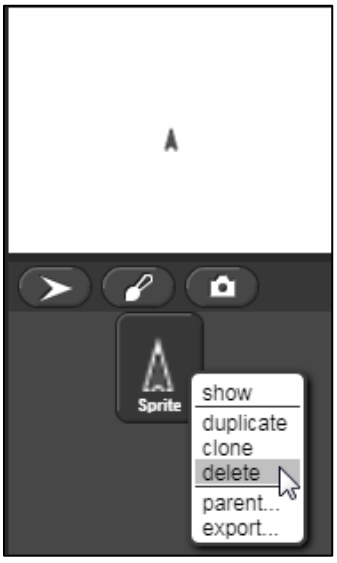
A Digital Storybook

The *Snap!* workspace contains a *Stage* on the right-hand side of the screen and a space for scripts to the left of the stage. Different backdrops can be placed on the background of the stage for different scenes.

Actors called sprites can be placed on the stage. Initially *a screen turtle* (in the form of an arrow) is the default sprite that appears on the stage. (For a more detailed description of *Snap!* robot turtles, see the previous module, *Turtle Graphics*.)



The turtle sprite is not used for creation of the initial digital storybook. Right-click on the turtle icon below the stage to access a menu of options. Select *Delete* to delete the turtle.



Adapting a Story

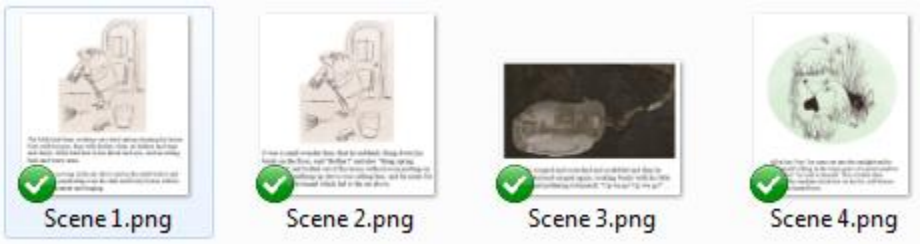
A printed storybook consists of a series of pages with text and images. An electronic storybook enables the text to be read on a tablet or laptop computer. In addition, other features such as speech synthesis can be used to enhance the story.

The story used in this example was adapted from the classic children's novel by Kenneth Grahame, The *Wind in the Willows.* A similar process would be used to create an original story.

A word processor was used to write the text. Images were placed above the text on each page.

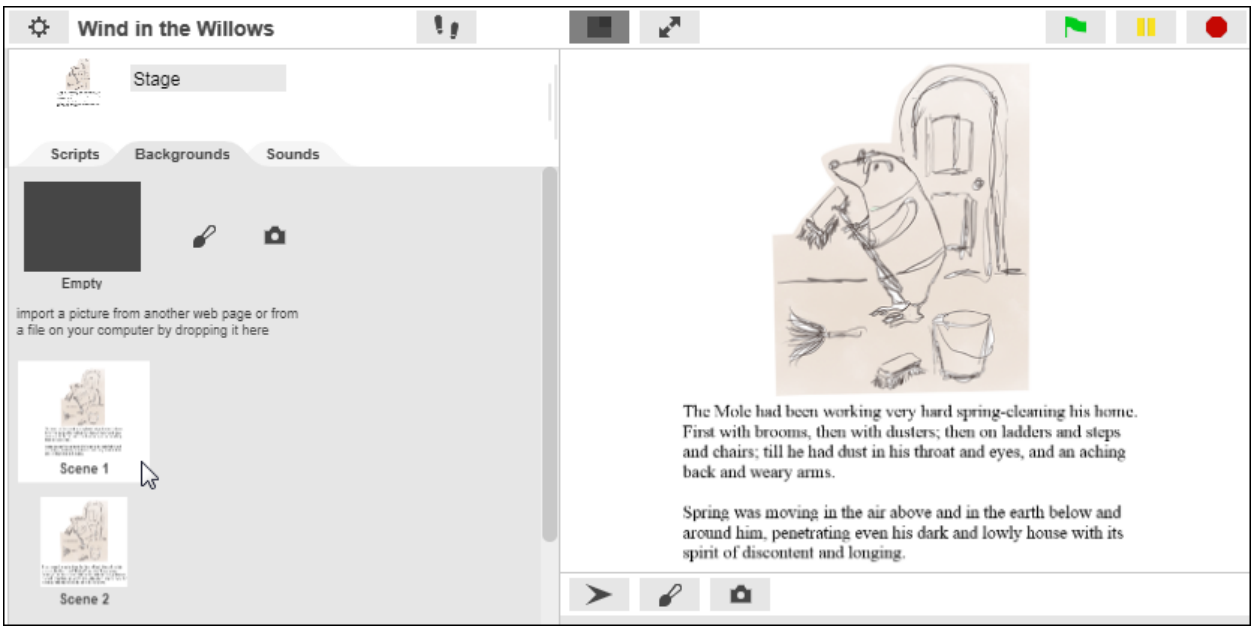
Saving the Story

The *Snipping Tool* in Windows was used to capture each page of the story and save it as an image. A similar screen capture tool could be used to capture each page of the story on a Mac or a tablet. The scenes of the story were saved as images labeled Scene 1, Scene 2, etc.



Importing the Story

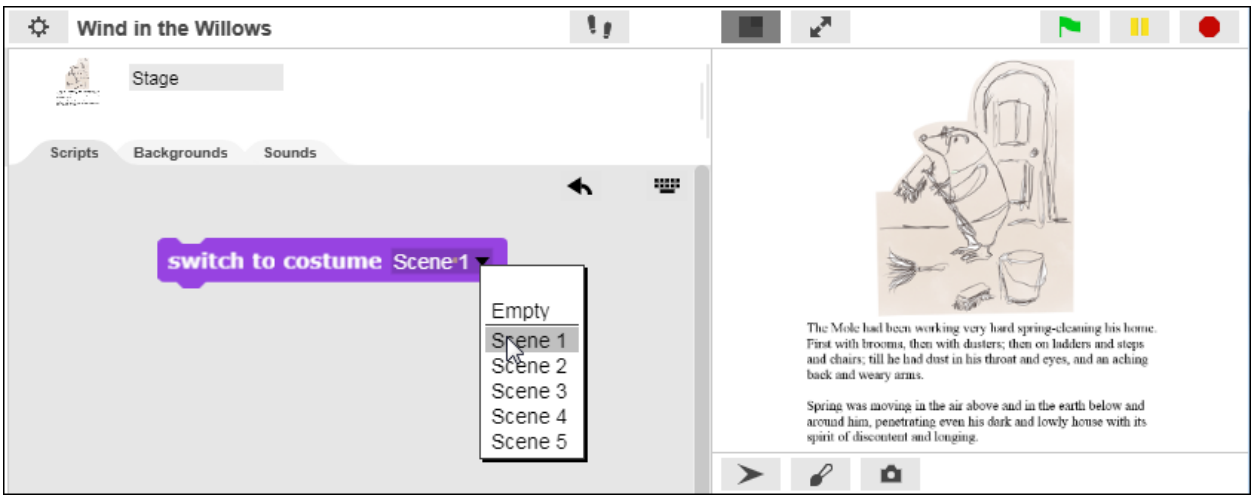
Start by clicking on the *Stage*, and then select the *Backgrounds* tab in the central workspace. Then drag the images into the space below the *Backgrounds* tab.



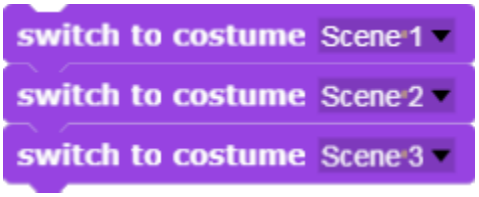
Storybook Script

Use the command **Switch to Costume** to switch from one page of the digital book to the next. This command is found under the *Looks* section of the *Commands* palette.

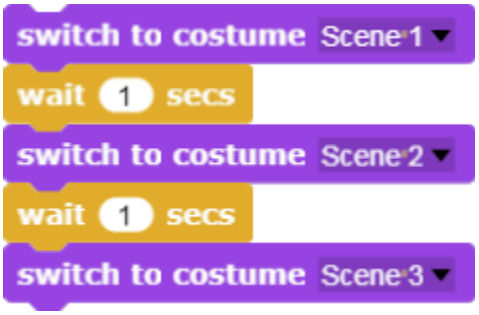
Select the *Scripts* tab (in the central workspace), and drag this command into the script space to the left of the *Stage*. Then select a page and click the command block. The background of the *Stage* will switch to the selected image.



Place the pages of the storybook in order to sequence them.



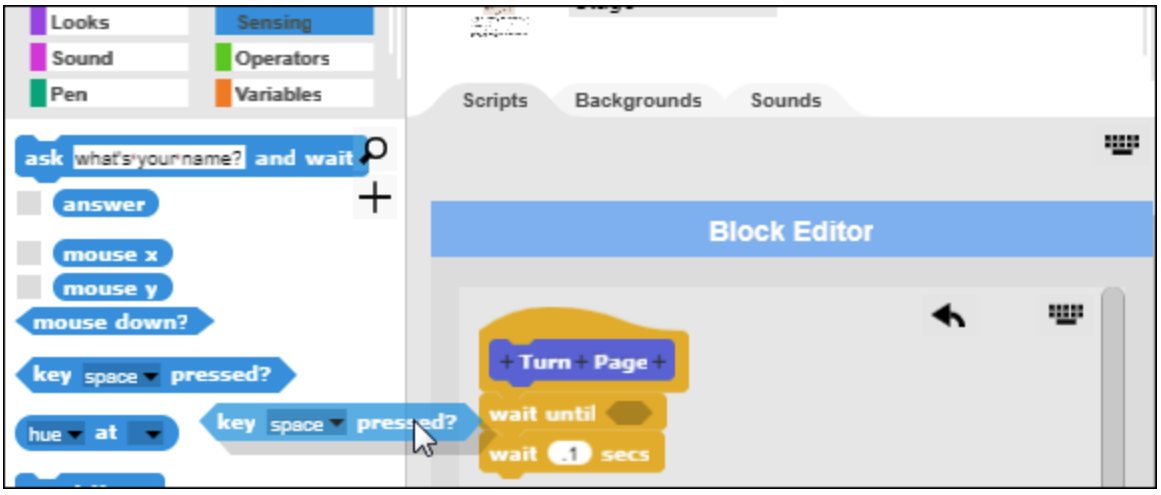
The script will move through the pages too quickly to read. Use the *Wait* command (found under the *Control* section of the *Commands* palette) to slow the presentation of the pages.



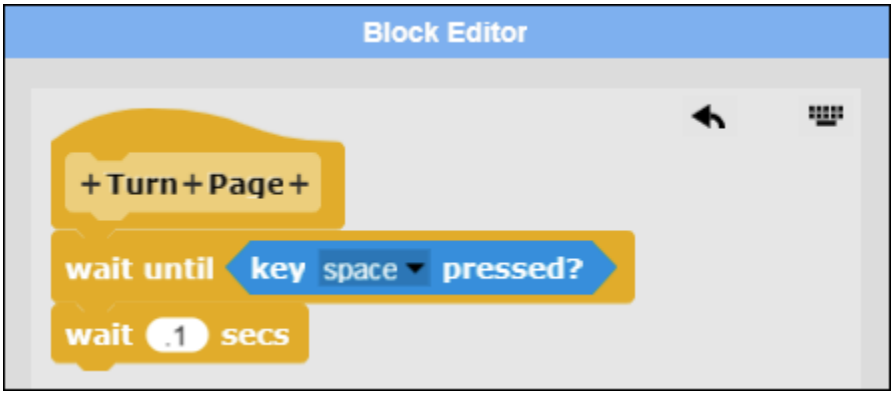
The Turn Page Command

Viewers will read the pages at different rates. Create a *Turn Page* command that enables the reader to turn the page by pressing the spacebar. The command **Wait until Key <Space> Pressed?** Can be used to create a *Turn Page* command.

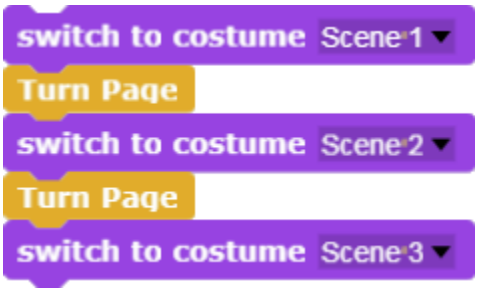
To build this command, create a new block under the *Control* palette called *Turn Page*. Add to it the *Wait Until* command. Then, drag the *Key Space Pressed?* command from the *Sensing* palette into the space on the *Wait Until* command.



A tenth second delay after this command ensures that the program advances only one page at a time. (Without this delay, a single keypress may advance several pages.)

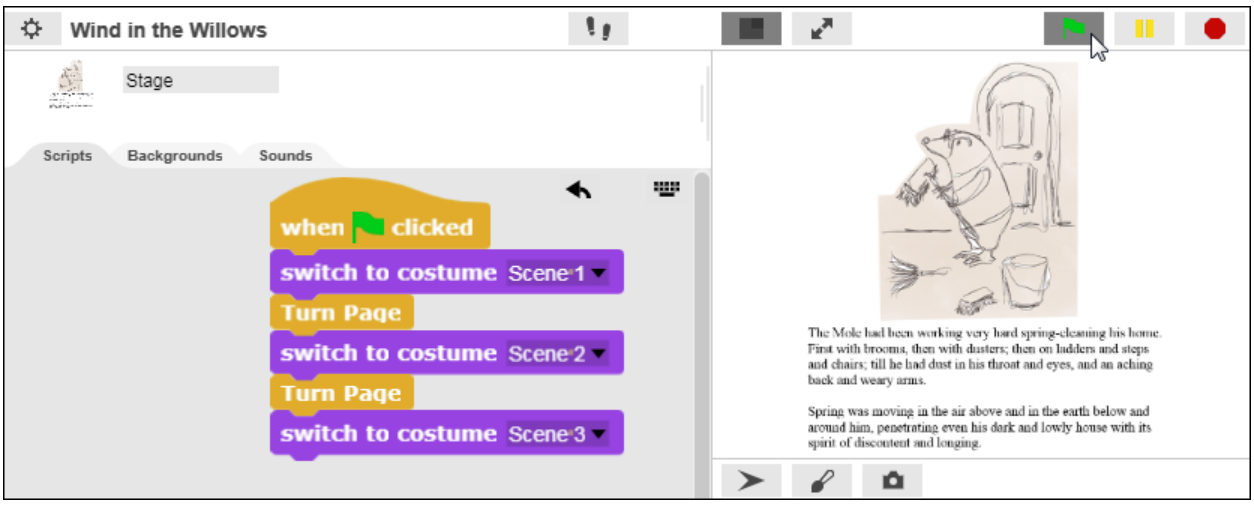


The revised procedure looks like this.



Reading the Story

The completed script begins with the command, **When the Green Flag is Clicked**:



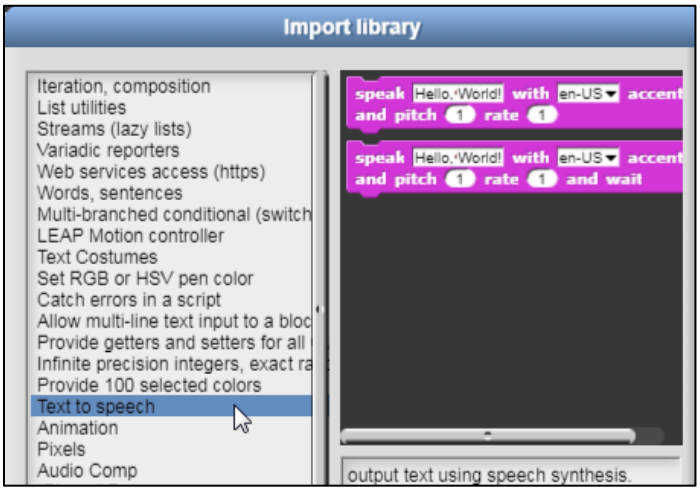
When the green flag at the top of the Stage is clicked, the story will begin.

It is a good idea to include a title page at the beginning with the instruction, *Press Space to Turn Page*.

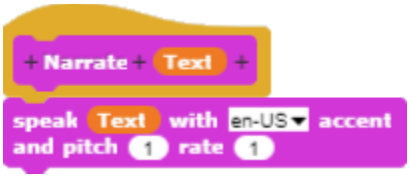
Text to Speech

The file menu in the upper left-hand corner of the screen includes an option to import libraries of additional commands that extend the capabilities of *Snap!*

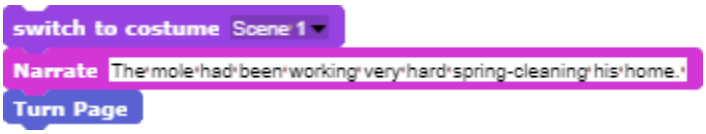
One of the options is a *Text to Speech* extension with commands that can be used to speak the text out loud.



After importing the *Text to Speech Extensions*, create a new command named *Say*. Select the desired pitch and rate of speech (available as options in the *Speak* command).



The script for a scene with an accompanying text-to-speech command might look like this.

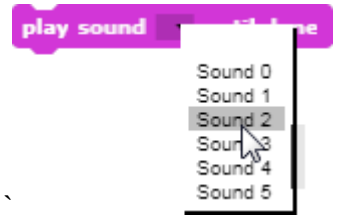


Recorded Speech

The *Sounds* tab in *Snap!* also offers the option of recording speech.

Recorded sounds appear under the *Sounds* tab.

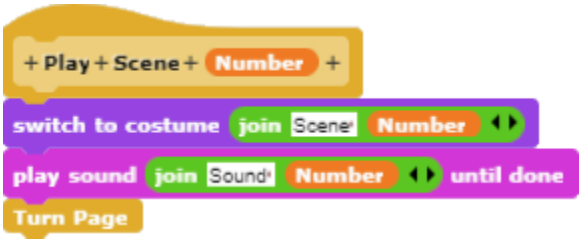
\The *Play Sound* command is used to select recorded sounds and play them in a script.



The *Play Sound until Done* command waits until the sound is completely played before continuing to the next command.

Creating a Master Procedure

Once the basic method for creating one scene in a digital storybook has been created, create a master procedure to play each scene in turn. The *Play Scene* procedure uses the *Join* command to combine the word “Scene” with the number of the current scene.



Similarly, the *Join* command combines the word “Sound” with the number of the sound played in a selected scene. The *Join* command makes it possible to use a variable such as *Number* to create a general-purpose master procedure.

Digital Extensions

The sample scenes from the *Wind in the Willows* can be accessed at:

https://tinyurl.com/Wind-in-the-Willows-Digital

The basic digital storybook consists of a series of imported backgrounds. Switching from one background to the next advances the page. Enhancements can include addition of either synthesized speech or recorded speech.

In addition to the static backgrounds in this initial illustration, characters that move about the stage and interact with one another can be created.

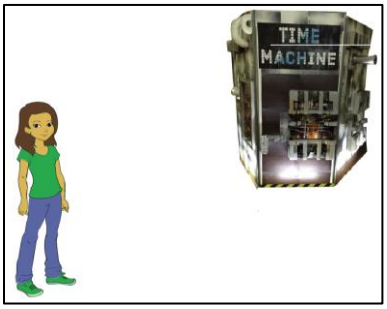
Storytelling – Animating Characters

The basic *Snap!* storybooks described in previous modules enable readers to advance from one page to the next by pressing the spacebar. Interactive stories build on this basic capability, and allow readers to control the direction of the story by making choices.

This module builds on that introduction and introduces new methods that allow *Snap!* to be used to animate characters.

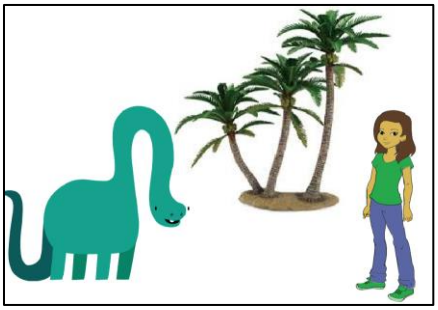
A Sample Story in Three Scenes

A sample story, *Abby and the Time Machine*, illustrates character animation in *Snap!*. The story has three scenes. In scene one, Abby encounters a time machine. In scene two, Abby is transported back in time by a time warp. In scene three, Abby meets a dinosaur in prehistoric times.



Scene 1: Time Machine

Scene 2: Time Warp



Scene 3: Prehistoric Era

The stage backdrop, sound effects, and dialog are outlined in the script below.

Scene 1. Time Machine

Stage Backdrop: A Time Machine (stage right)
Abby: "That looks like a time machine."

Scene 2. Time Warp

Stage Backdrop: Time Warp Effect
Sound Effect: Eerie Time Travel Music

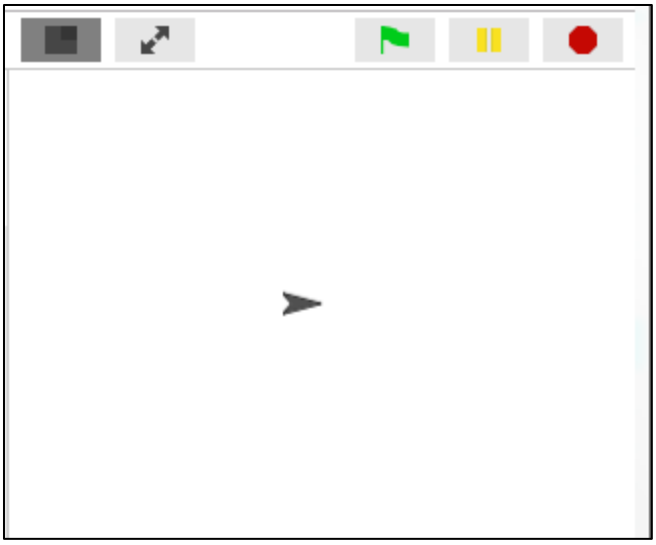
Scene 3. Prehistoric Era

Stage Backdrop: Palm Trees
Abby: "Look … it's a dinosaur!"
Dinosaur: "Are you from the future?"

While this is a relatively short story, it contains all of the elements required to create a digital play.

Scripts for Actors

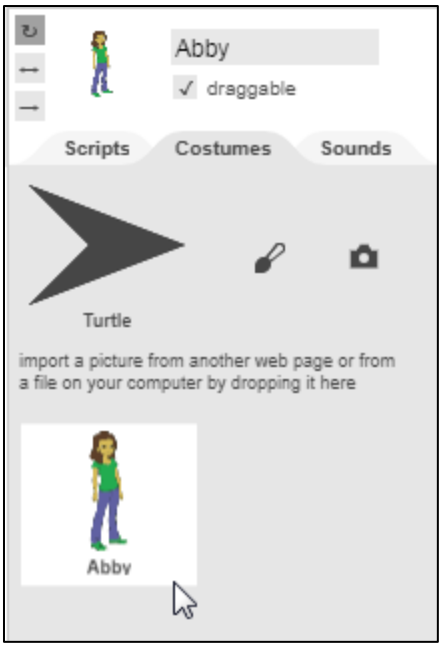
The *Snap!* workspace consists of a stage and actors called *sprites* that move about the stage. The individual sprites can be given directions called *scripts*. A stage manager can also issue commands to sprites to coordinate actions through a master script.



Initially a *screen turtle* (in the shape of an arrow) is the default sprite that appears on the stage. You can add additional sprites in the workspace below the stage. You can also rename each sprite by clicking on it and then changing the text in the box above the central workspace.

You can also switch the sprite costumes to reflect the character or role played. A library of premade costumes is available. You can also create new costumes in the *Snap!* paint program, upload new costumes from a file, or capture new costumes with a camera.

The image for Abby was dragged into the area below the *Costumes* tab for the sprite.

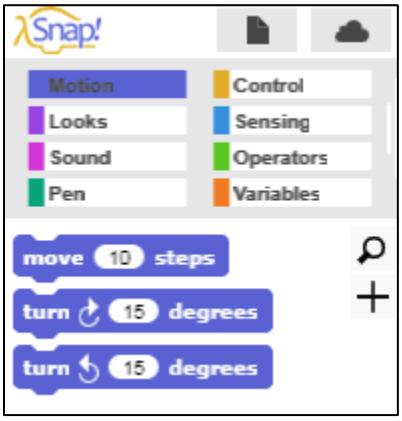


Then a second sprite was created and the image of a dinosaur was dragged into the *Costumes* area for this sprite.

A total of three sprites were created:

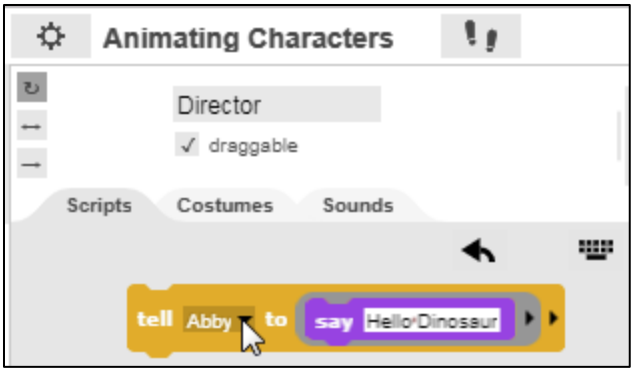
1. Abby
2. The Dinosaur
3. The Director

A code tab in the upper left-hand corner of the *Snap!* workspace displays a variety of commands that can direct the actions of the sprites.

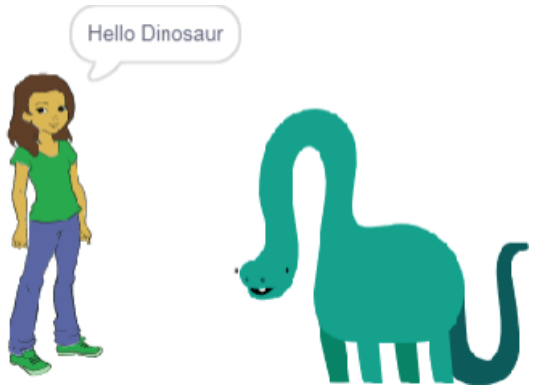


All of the scripts in this example are placed in the Director's script space. (Because the director is off-stage, the costume for the director is blank.) In the illustration below, the Director has issued a command to Abby:

Tell Abby to Say "Hello Dinosaur."

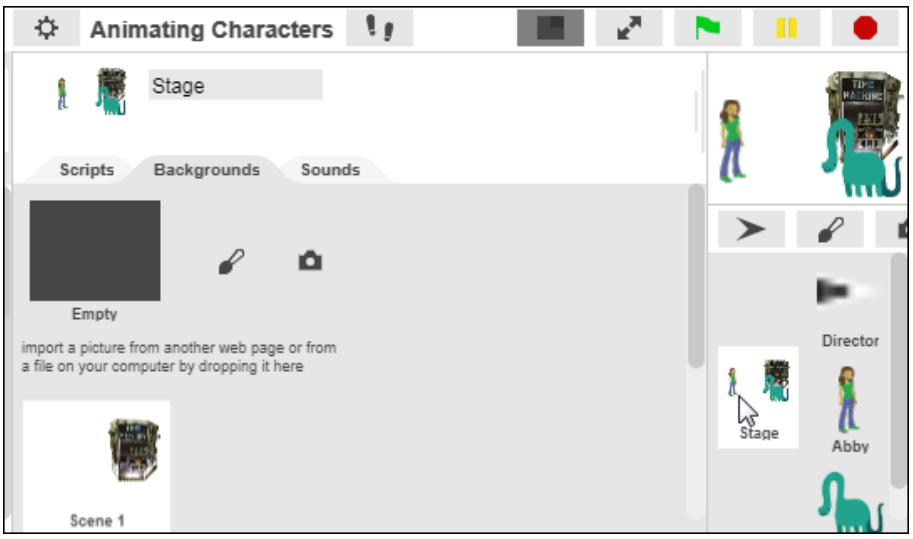


The *Say* command causes a speech bubble with dialog to appear beside a sprite.

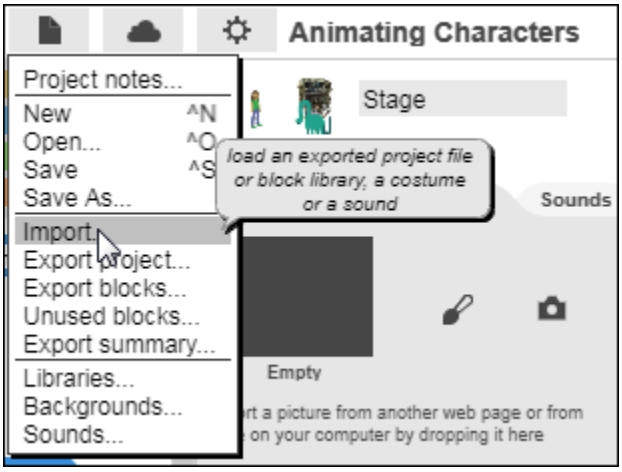


Setting the Stage

When the *Stage* icon (beneath the stage on the bottom right-hand corner of the screen is selected, a *Backgrounds* tab appears next to the *Script* tab.



This option allows images to be imported that can be used as backgrounds for the digital story.



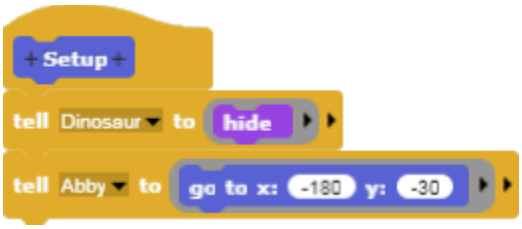
The images used as backdrops for the three scenes in the sample stories can be accessed through the following link: [Backdrop Graphics]

The folder accessed has three backgrounds that have been created for Scene 1, Scene 2, and Scene 3:

Upload these graphics as backdrops for the story:

1. Scene 1: Time Machine
2. Scene 2: Time Warp
3. Scene 3: Prehistoric Era

A *Setup* procedure sets the stage for the first scene. It hides the dinosaur (since it does not appear in the first scene). It also positions Abby on the left-hand side of the stage.



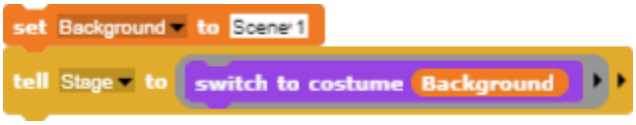
With the stage set, the scenes that follow can be developed.

Scenes

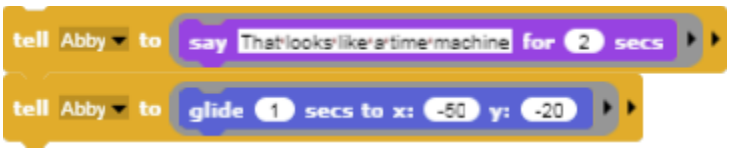
The elements for the digital story include:

1. Two Actors: (1) Abby and (2) the dinosaur
2. Three Stage Backdrops: (1) Time Machine, (2) Time Warp, and (3) Prehistoric Era
3. Actors' Scripts: Directions that tell the actors what to do.

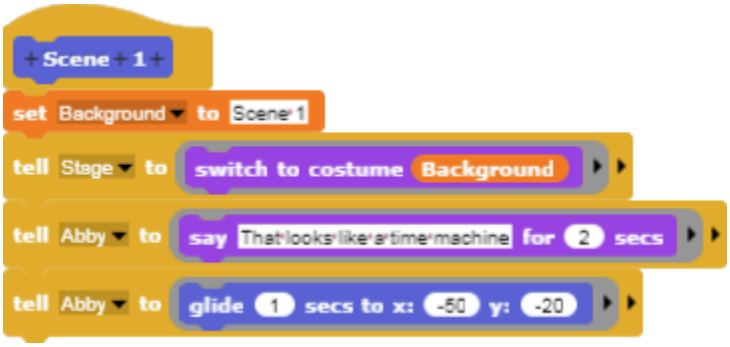
Define a new procedure named *Scene 1* in the script space for the stage. The script will initially tell the *Stage* to set the background to Scene 1.



After the background for Scene 1 is in place, Abby is given her first line, "That looks like a time machine!" Abby is then directed to move toward the time machine.

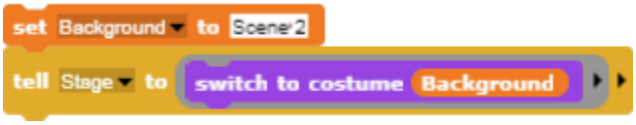


The complete script for Scene 1 looks like this.



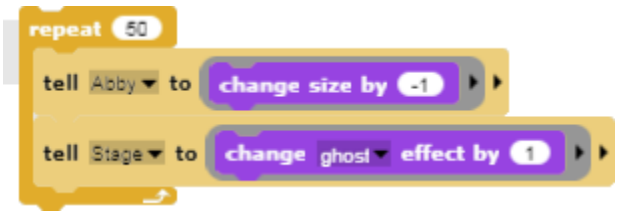
Scene 2

In Scene 2, the backdrop is switched to the *Time Warp* graphic.

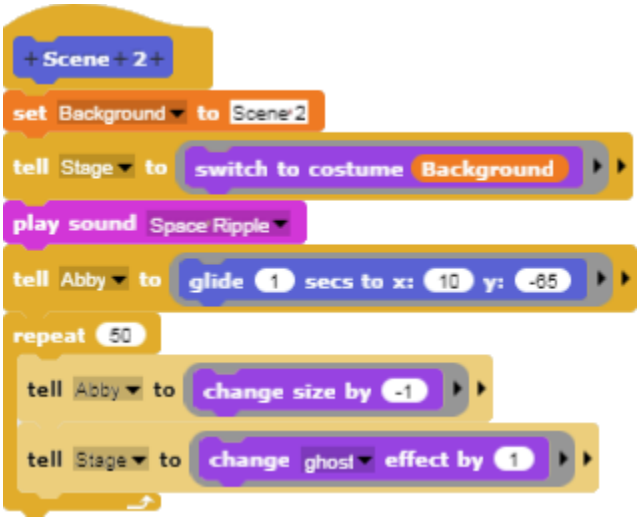


An eerie sound is then played, and Abby is moved to the center of the time vortex.

These actions are followed by a couple of special effects. The command, “Tell Abby to change size by - 1” causes Abby to shrink in size each time the repeat statement is executed. The command, “Tell Stage to change *Ghost Effect* by 1” causes the time warp graphic to gradually dim.



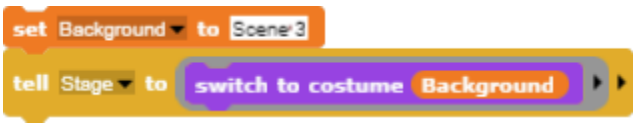
The assembled script for Scene 2 looks like this.



Scene 3

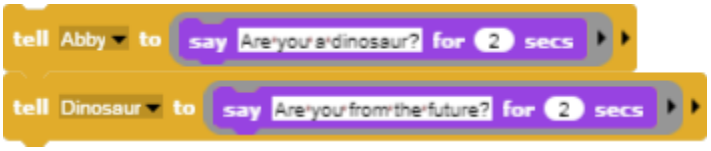
At the beginning of Scene 3, Abbey must be restored to normal size and the Stage must be restored to its normal level of brightness.

The background is then switched to prehistoric times with a palm tree in the background.



The dinosaur is revealed since it appears in Scene 3 and Abby is moved to the left side of the stage.

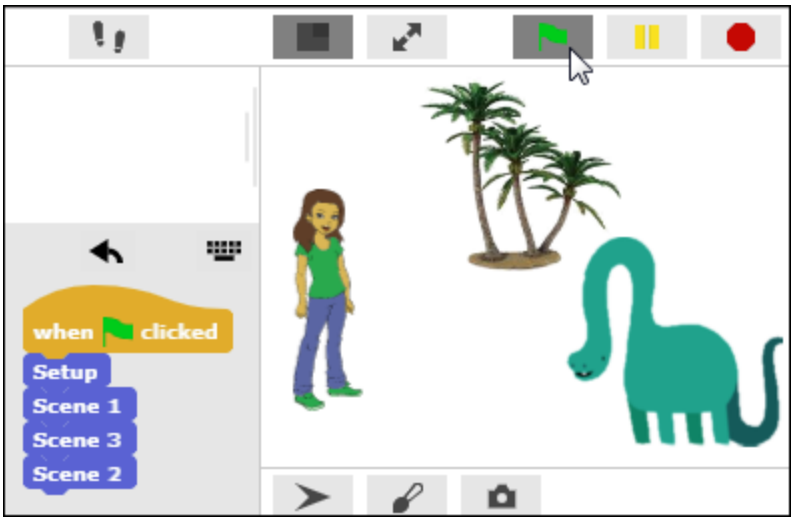
Abby then asks, “Are you a dinosaur?” and the dinosaur replies, “Are you from the future?”



The assembled script for Scene 3 looks like this.

A Master Procedure

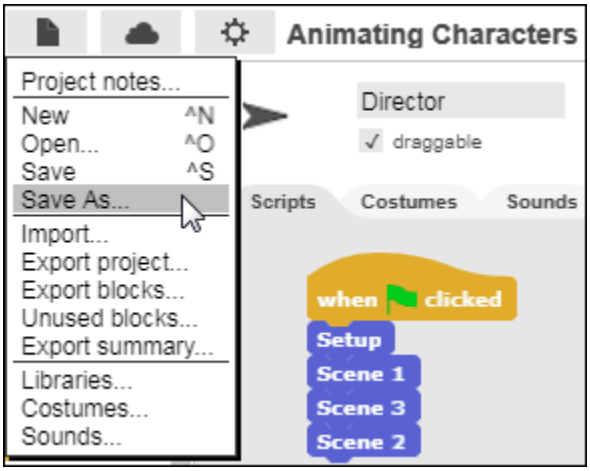
The *Setup* procedure and the three scenes that follow are then combined into a master procedure. Clicking the green flag at the top right-hand corner of the stage causes the scripts to execute in the order in which they are assembled in the master procedure.



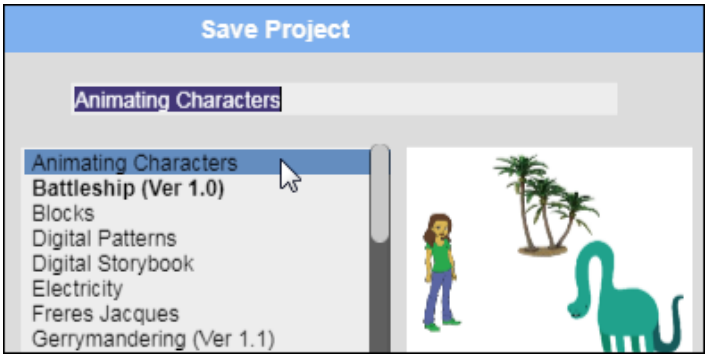
Although this short digital story only includes three scenes and a couple of lines of dialog, it illustrates the principles by which a longer interactive story can be constructed.

Remixing the Project

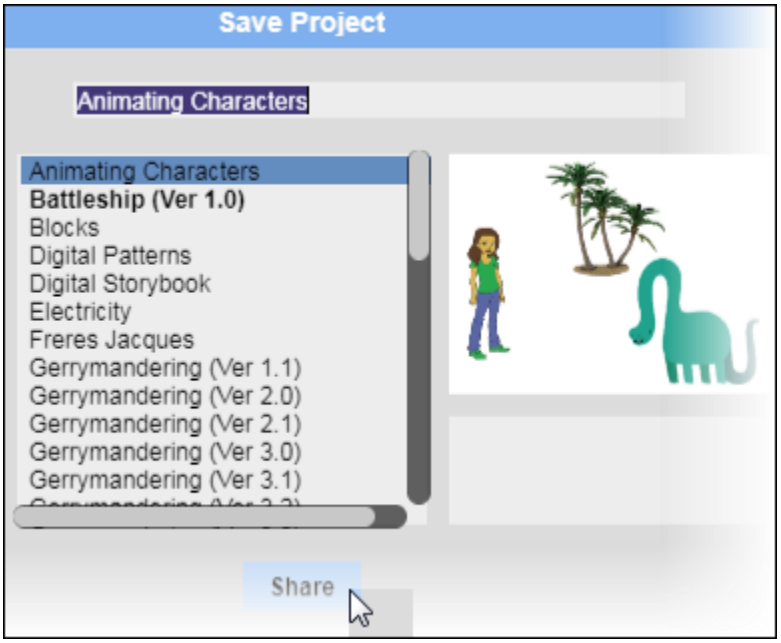
The *Save As* menu provides access to an option that enables a project to be shared.



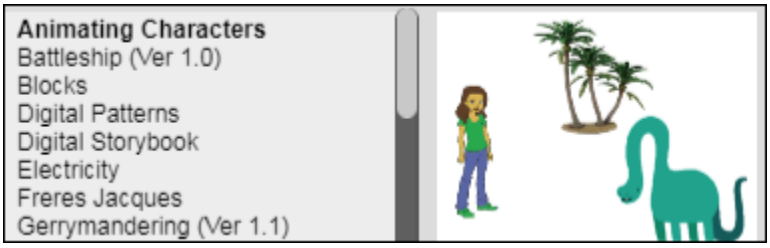
Sharing a project involves two steps. First, select the project in the *Save* menu as shown below.



Then click the *Share* button that is found at the bottom of the dialog box.



After a project has been shared, its title is highlighted with a bold font in the *Save* menu.



The web address for the project can then be shared with others. This allows others to access the shared project and run the script that has been developed. When a project is shared, others can also revise the script, remixing it to enhance it and add their own features.

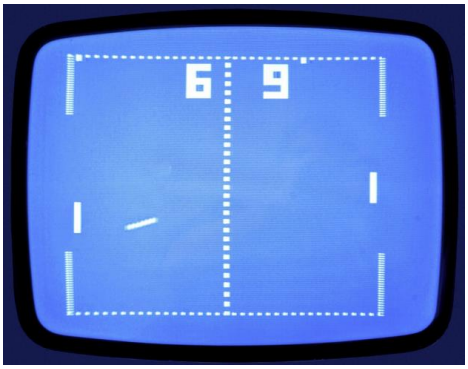
The concept of sharing and remixing is an important idea, because it allows individuals working together in a community to collaborate and build upon one another's work.

CHAPTER 3

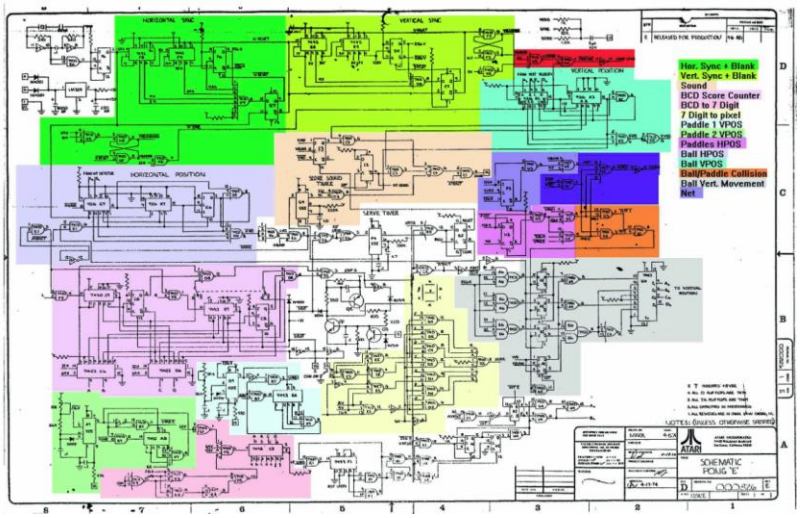
Arcade Games

Glen Bull and Michael Littman

Pong was the first commercially successful video game, and due to its cultural significance, is now part of the Smithsonian's permanent collections. Released by Atari as an electronic table tennis game, Pong helped establish the video game industry.



Atari used discrete digital logic chips to develop the game. In the color-coded schematic, the logic chips responsible for the vertical and horizontal positions of the game paddles are highlighted in green and purple. The logic chips that detect a collision between the ball and a paddle are highlighted in orange.

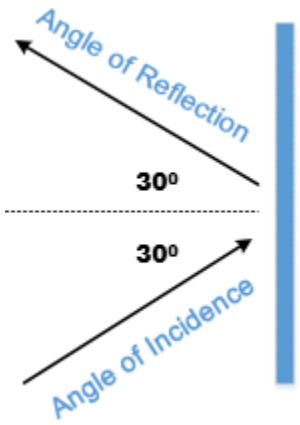


When the Atari Pong game was developed in 1972, computers were too expensive to devote a single computer to a game. Today, small microcomputers like the Raspberry Pi microcomputer capable of implementing the Pong can be purchased for $25.

In this module, a Pong game will be created using *Snap!* Writing a program to recreate Pong provides opportunities to explore concepts related to force and motion. Key concepts include (1) motion is described by an object's speed and direction, (2) changes in motion are related to force and mass, and (3) friction is a force that opposes motion.

Ball Strikes the Paddle

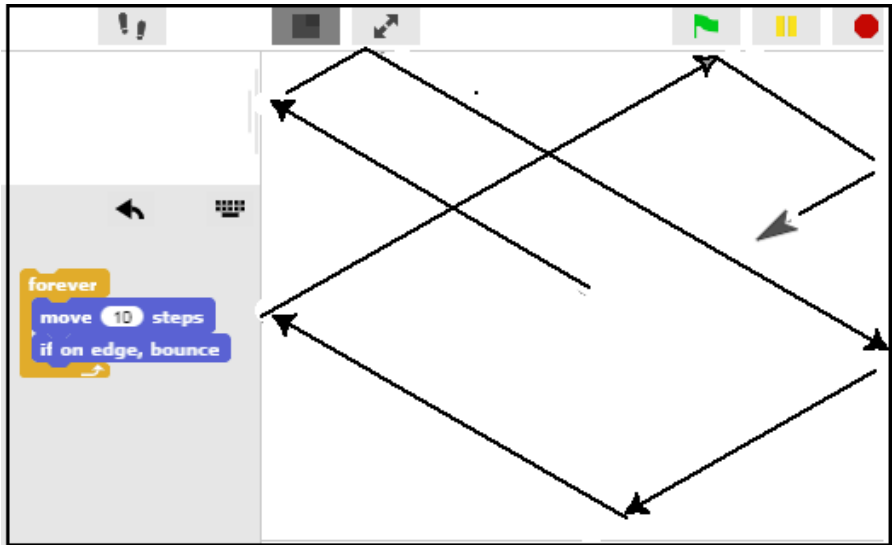
In this example, one sprite (labeled *Turtle*) will rebound when it strikes a second sprite (labeled *Paddle*). When an object such as a billiard ball strikes a surface, its angle of incidence when it strikes the surface is equal to its angle of reflection when it rebounds from the surface.



Snap! has a built-in command **If on Edge, Bounce** that automatically applies the correct angle of reflection when the turtle reaches the edge of the stage. This command is useful for preventing the turtle from inadvertently disappearing off the edge of the stage. The following commands will cause the turtle to travel across the stage, bouncing when it reaches an edge:

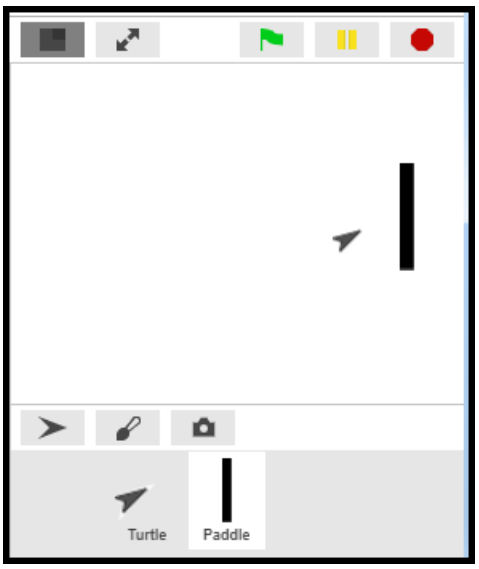
Forever [Move 10; If on Edge, Bounce]

If the pen is down, the turtle will leave a trail that makes it possible to follow its path as it travels from one edge of the screen to the next. The turtle orientation in its starting position should be at an angle as shown in the illustration.



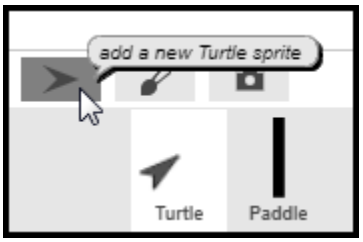
Creating the Game Paddle

In this recreation of the Pong game, the *Turtle* will travel across the screen until it strikes a second sprite (labeled *Paddle*).

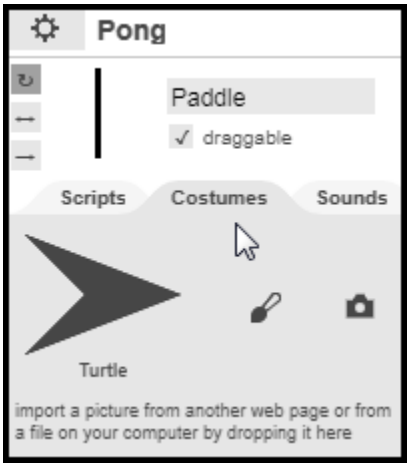


Creating the Paddle

Click on the **Sprite** icon below the stage to create a new sprite.



Name the newly created sprite “Paddle” and then select the *Costumes* tab to access the *Paint Editor*.

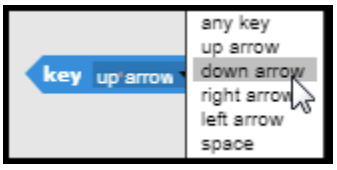


Use the *Paint Editor* to create a vertical bar that will become the game paddle. The *Paint Editor* has tools to adjust the size of the game paddle as needed.

Moving the Paddle

The game paddle sprite should be placed near the back wall of the stage. Move the sprite by clicking on it and dragging the mouse.

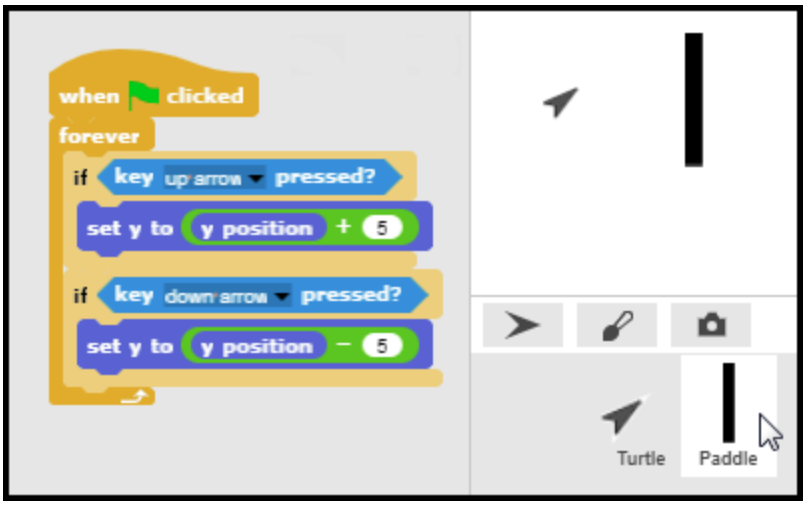
The *Up* and *Down* arrow keys on the key board can be used to move the game paddle up and down. The *Key* command (located under the *Sensing* command palette) can be used to detect a keypress.



The logic of the procedure that moves the paddle states that "If the *Up Arrow* key is pressed, then set the Y (vertical) position of the paddle to its current position plus 5 more steps." This effectively moves the paddle up 5 turtle steps each time that the *Up Arrow* key is pressed.

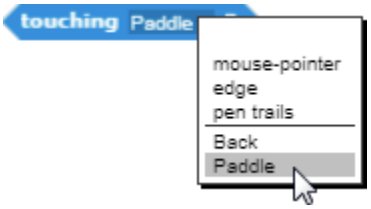
If *Key Up Arrow* is Pressed?
Then Set Y to Y Position + 5

Similar logic is used to move the paddle down when the *Down Arrow* key is pressed. Note that this script must be placed in the script space for the paddle sprite (and not the turtle sprite). This script space can be accessed by clicking the Paddle icon below the stage.

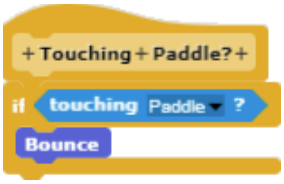


Collision Detection

When the turtle strikes the paddle, it should rebound. To cause this to occur, a means of detecting when the turtle is touching the paddle is needed. One of the commands in the *Sensing* palette can be used to detect this. A drop-down menu can be used to select other objects on the stage.

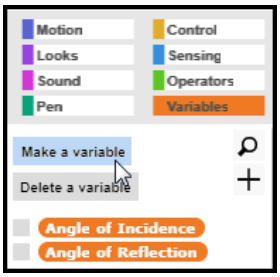


This command can be used to create a *Touching Paddle?* procedure. The *Touching Paddle?* procedure checks to see if the turtle is touching the paddle. When the turtle touches the paddle, the *Bounce* procedure is called. The *Bounce* procedure is described in the next section.

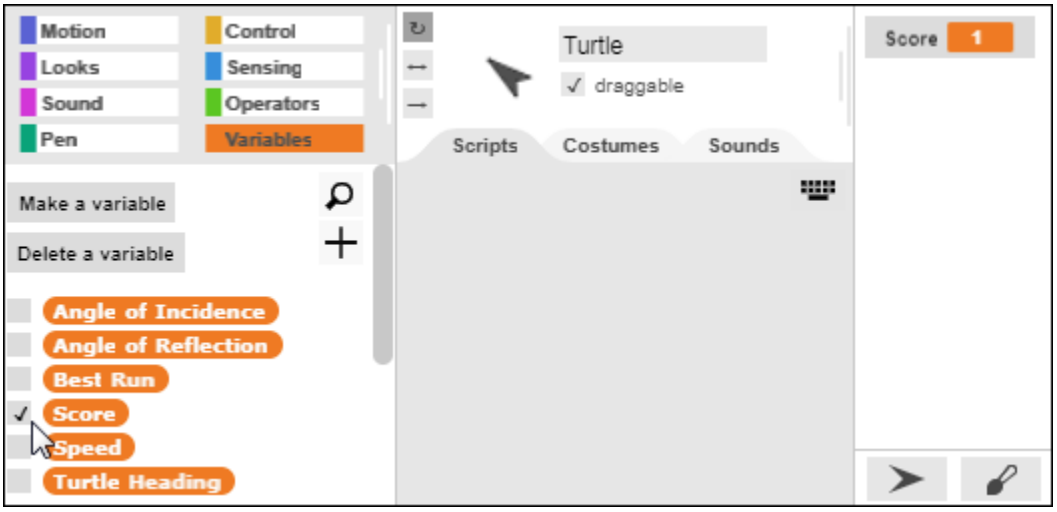


The Bounce Procedure

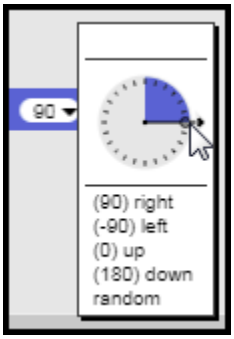
The *Bounce* procedure uses two variables: (1) Angle of Incidence and (2) Angle of Reflection.



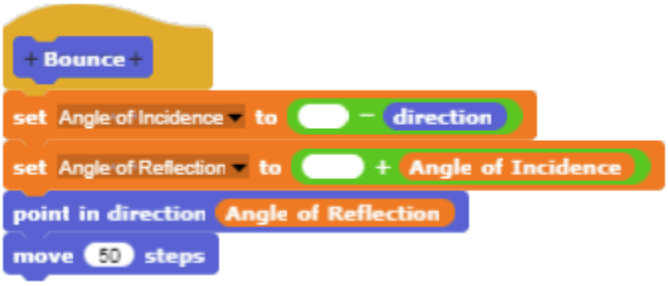
Each variable has a check-box beside it. When the check-box is selected, the value of the variable is displayed on the screen. In the case of a variable such as *Score* (described later), it would be useful to display the player's score on the stage. On the other hand, variables such as *Angle of Incidence* and *Angle of Reflection* that do not change during the course of the game clutter up the screen. Therefore the boxes beside these variables can be unchecked so that they are not displayed on the screen.



The command *Direction* (located under the *Motion* commands palette) can be used to determine the direction in which the turtle is pointed. If the turtle is pointed straight up, the command *Direction* returns a value of 0 degrees. If the turtle is pointed straight down, the command direction returns a value of 180 degrees. If the turtle is pointed to the right, the command *Direction* returns a value of 90 degrees.



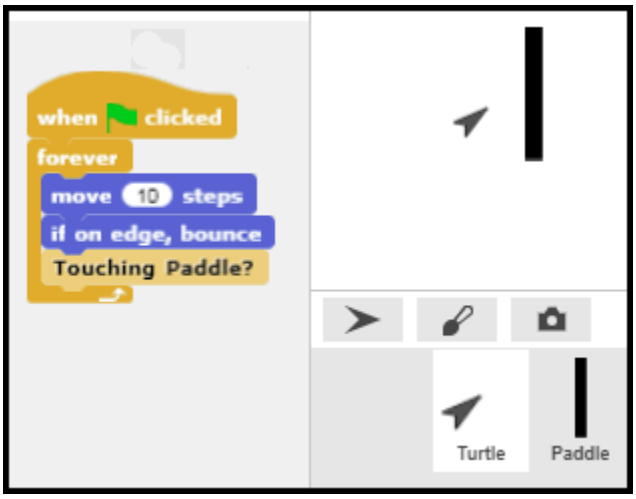
The turtle's direction can be used to determine the *Angle of Incidence* as it travels toward the paddle. The *Angle of Incidence*, in turn, can be used to determine the *Angle of Reflection*. When the *Touching Paddle?* command detects that the turtle has touched the paddle, the *Bounce* procedure uses the derived *Angle of Refection* to point the turtle in the right direction as it rebounds.



The *Bounce* procedure essentially performs the same calculations when the paddle is touched as the built-in *If on Edge, Bounce* command uses when the edge of the stage is reached. By running the procedure with the pen down, it is possible to determine if the angle of reflection when the turtle strikes the paddle is the same as the angle of reflection when the turtle reaches the edge of the stage and rebounds. If both angles of reflection (for the edge of the stage and the paddle) are the same, then the variable *Angle of Reflection* has been correctly calculated.

The Master Procedure

The master procedure (located in the turtle's script space) should look like this. The turtle should be turned to a starting position in which it is facing the paddle at an angle, as shown in the illustration.



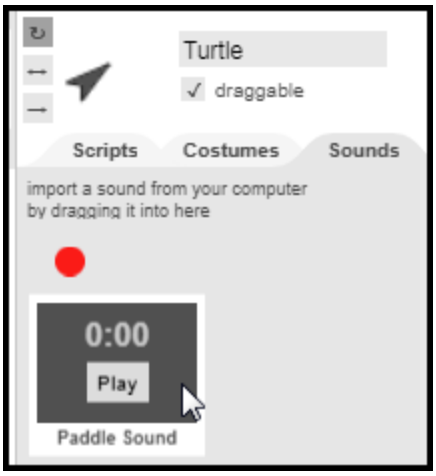
When the *Green* go flag is clicked, the master procedure will move the turtle across the screen until it touches the paddle or the edge of the screen. When the turtle touches the paddle, the *Bounce* procedure will cause it to rebound.

Enhancements - Sound

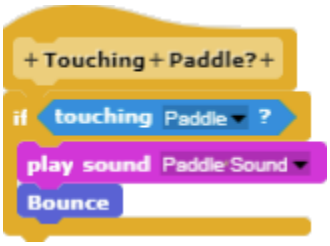
The basic program logic and procedures are now in place. Several enhancements will increase the play value. The Atari Pong game is instantly recognizable because of the electronic blip that occurs when the ball strikes the paddle.

Pong Sounds Folder

Select the *Sounds* tab of the turtle. Then drag the sound labelled *Paddle Sound* from the *Pong Sounds Folder* into sounds space of the turtle sprite.



Once the paddle sound is available, the *Touching Paddle?* script can be updated to play the Pong paddle sound each time the turtle sprite collides with the paddle.

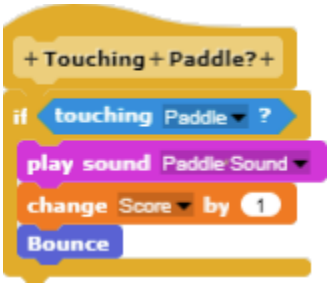


Enhancements – Scoring

A method for maintaining the score is another important enhancement. A variable named *Score* could be added to track the score. The score could be increased by 1 point each time the player successfully block the turtle with the paddle by adding the code:

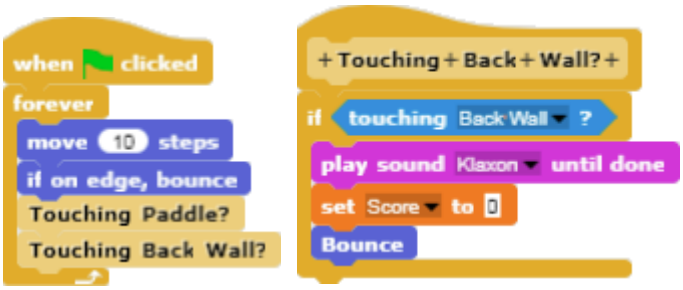
Change Score by 1

to the Touching Paddle? script. This variable would need to be initialized by setting its value to 0 at the beginning of the game.



A third sprite, labeled “Back Wall” could be added to detect when the paddle does not block the turtle.

A *Touching Back Wall?* procedure could detect when the turtle touches the back wall.



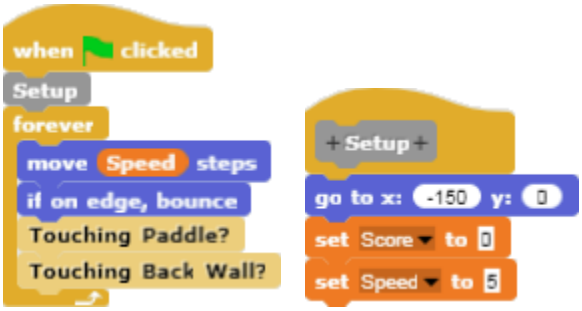
When the turtle escapes the paddle and touches the back wall, the *Touching Back Wall?* procedure could reset the score to zero (or end the game with a *Game Over* display).

Speed

The number of steps that the turtle moves during each iteration of the *Forever* loop determines how fast it moves across the screen. Creation of a variable named *Speed* will make it possible to adjust the speed of the turtle during game play.

For example, in one variation of the recreated Pong game, the speed is increased each time the player successfully deflects the turtle five times in a row. Eventually the turtle may move so fast that the player cannot block it in time. The score at that point becomes the player’s high score.

At this point it may be useful to create a *Setup* procedure to establish the turtle's initial starting location, speed, and score.



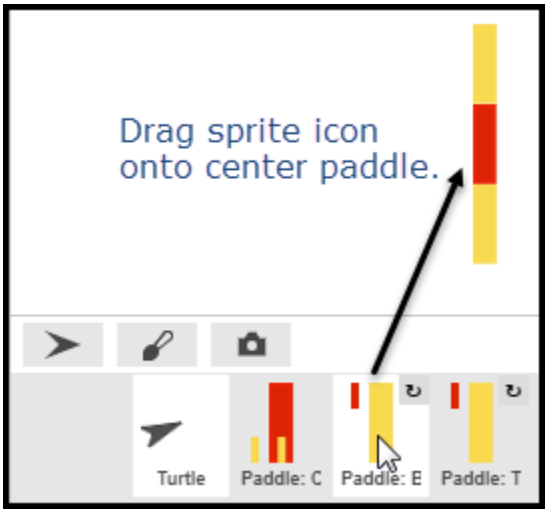
Paddle Surface

Until this point, the assumption has been that no energy is lost in the collision between the turtle and the paddle surface. In actuality, some energy is always absorbed in a collision of this kind. A softer, spongy surface will absorb more energy than a softer surface.

To reward players for accuracy, a composite paddle can be constructed with harder surfaces on the edges and a softer, more absorbant surface in the center. In the illustration, the harder surfaces on the edges are depicted in yellow while the softer, absorbant surface is depicted in red.

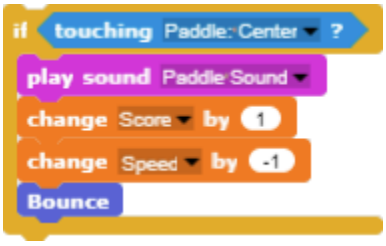
To construct a composite paddle, create two more paddle segment sprites. One sprite will be come the top paddle segment and the other sprite will become the bottom paddle segment. Make sure that there are no scripts in the script areas of these two additional sprites. Then line up the top and bottom paddle segments so that they are aligned with the center paddle segment.

Once the paddle segments are aligned, drag the top paddle segment icon (below the stage) onto the center paddle segment on the stage. Repeat the process for the bottom paddle segment. Once this is done, all three segments will move together as a single unit.



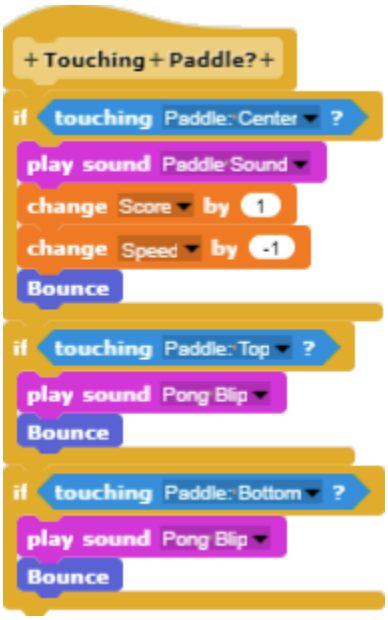
In this illustration, the top paddle sprite has been named *Paddle: Top* and the bottom paddle sprite has been named *Paddle: Bottom*. When the turtle strikes the top or bottom paddle segments, the program plays a sound and turtle rebounds, but the score is not increased.

If the turtle collides with the center paddle segment, the player is rewarded for accuracy. The score is increased by one and the speed is decreased, making the game easier to play.



The updated *Touching Paddle?* procedure:

1. Checks to see if the center paddle segment has been touched, and increases the score and decreases the speed before rebounding if this occurs.
2. Checks to see if the top paddle segment has been touched, but does not increases the score or change the speed before rebounding if this occurs.
3. Checks to see if the bottom paddle segment has been touched, but does not increases the score or change the speed before rebounding if this occurs.

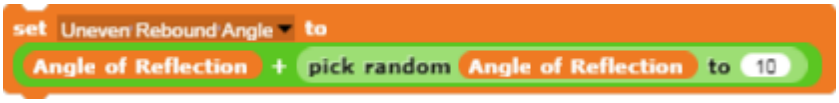


There are other variants that could be implemented. For example, assuming that the outer segments are constructed from brittle material, a counter could be used to count the number of times each segment on the edge of the paddle collides with the turtle. After an edge segment collides with the turtle a specified number of times, it could break off, leaving the player with a smaller paddle.

Angle of Reflection

Thus far an even surface has been assumed. Consequently, the turtle rebounds in the same predictable each time. The angle of reflection is always equal to the angle of incidences. For a more interesting, less predictable game, an uneven paddle surface could be created. This would result in some variation in the angle at which the turtle rebounds.

To accomplish this, a variable named *Uneven Rebound Angle* could be created that was set to the Angle of Reflection plus a random number between -10 and +10. Pointing the turtle in the direction of the Uneven Rebound Angle created in this way would introduce some variation in the angle at which the turtle rebounds from the paddle.



Retroreflection

Paddles with different shapes would cause the turtle to rebound at different angles. For example, a “V” shaped paddle would cause the turtle to rebound at an angle of reflection 180 degrees from the angle of incidence.

Light rays have similar properties. An optical device that reflects light back to its original source is known as a retroreflector.

Retroreflection could be simulated in *Snap!* using mathematical calculations similar to those used to create the Pong game.

CHAPTER 4

Word Play

Glen Bull and Paula Cochran

Alan Turing was an early computing pioneer who worked in the field of artificial intelligence. Turing developed the Turing Test, which proposed that a machine could be described as intelligent if a person communicating with it could not tell if they were speaking with a machine or another person. The first computing languages such as FORTRAN were designed for numeric operations, but sentences used in conversations consist of lists of words rather than numbers. Another computing pioneer working in the Artificial Intelligence Laboratory at M.I.T., John McCarthy, developed a programming language named *LISP* (short for "list processing" language) that was well suited for development of programs designed to address the challenge posed by Turing.

Seymour Papert, who subsequently served as co-director of the M.I.T. Artificial Intelligence Laboratory, developed a dialect of LISP known as *Logo*. (*Logo* is the Greek word for "word.") *Logo* was designed for use by children, and included many features well-suited for exploration of the lists of words and sentences that comprise a language. The descendants of *Logo* such as *Scratch* and *Snap!* retained these list processing capabilities.

The Grammatical Structure of Gossip

Just as a computer program has a structure, language also has a structure. Sentences, poems, and stories all have an underlying composition. The way in which sentences are put together is described as their *syntax*. The structure of the underlying meaning is described as *semantics*. The way in which syntax and semantics are used to form a story can be described as *style*.

In *Exploring Language with Logo*, Paul Goldenberg investigates these concepts through development of a grammar of gossip. In its simplist terms, gossip can be described as:

[Who] [Does What]

Who can be constructed from a list of names:

Sam Sally George Ann

Does What can be constructed from a list of actions.

[procrastinates] [talks all the time] [cheats] [loves to gossip]

In *Snap!* these lists can be constructed in the following way:

Making a List

The *Snap!* list command is found under the *Variables* section of the commands palette. In this illustration, a list of names has been created.

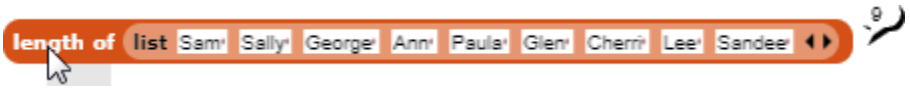
A second list includes a series of actions.

In the *Gossip* game, a name will be randomly picked from the list.

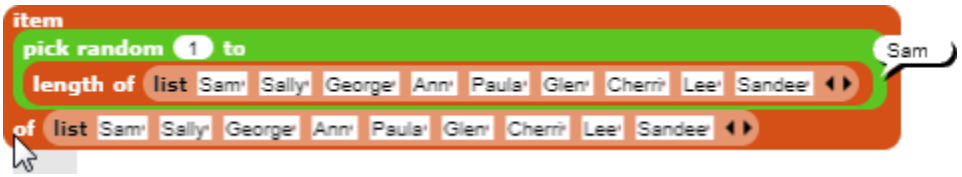
Picking a Random Item from a List

The *Random* function (found under the *Operators* command palette) randomly chooses a number between 1 and some upper bound. Since there are nine words in the list of people, in this case we would like to pick an item between one and nine.

At times there may be a different number of items in the list. The *Length of List* command (found under the *Variables* command palette) can be used to automatically find the length of any list. In this case, the *Length of List* command tells us that there are nine items in the list of people

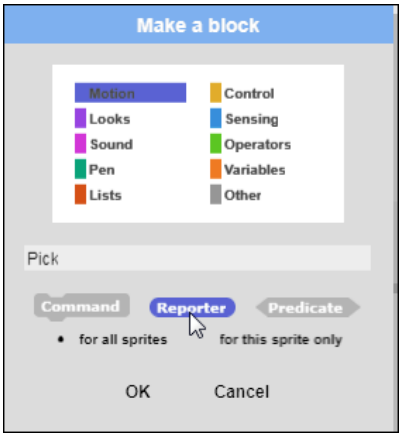


The Random command now picks a number between 1 and the number of items in the list.



The Pick Procedure

These commands enable us to create a more general procedure named *Pick* that randomly picks an item from a list and reports the result. Since the *Pick* procedure reports an item randomly selected from a list the *Reporter* option is chosen when the procedure is created.

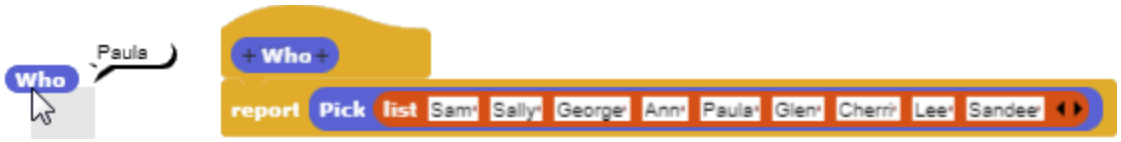


The completed procedure looks like this.

The *Pick* procedure randomly picks an item from a list.

The Pick procedure now makes a procedure that randomly picks a person from a list. The procedure, named Who, reports the name of the person picked from the list. Because the procedure reports an outcome, the oval *Reporter* option is chosen when the procedure is created.

The completed Who procedure reports the name of the person picked from the list.



A parallel *Does What* procedure picks an action from a list

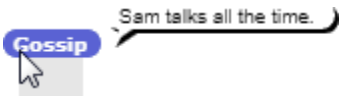
Gossip

The *Join* operator (found under the *Operators* section of the command palette) joins a person and action together in a sentence.

A space can be added to separate the person and the action, and a period can be added at the end to complete the sentence.

These commands form the basis of the *Gossip* procedure.

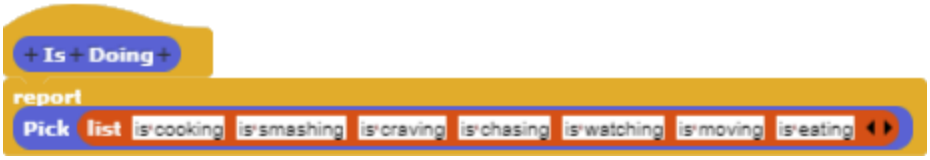
The *Gossip* program reports a person and an action.



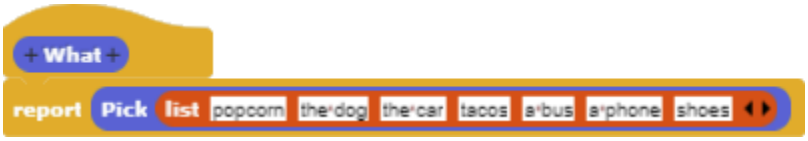
This format serves as a springboard for exploration of more complex grammars. For example, *Does What* could be replaced with a transitive verb, *Is Doing*.

[Who] [Is Doing] [What]

The *Is Doing* procedure includes a list of actions that people are doing.



This can be combined with a word picked from the following list.



This leads to construction of sentences like this.

Computer Poetry

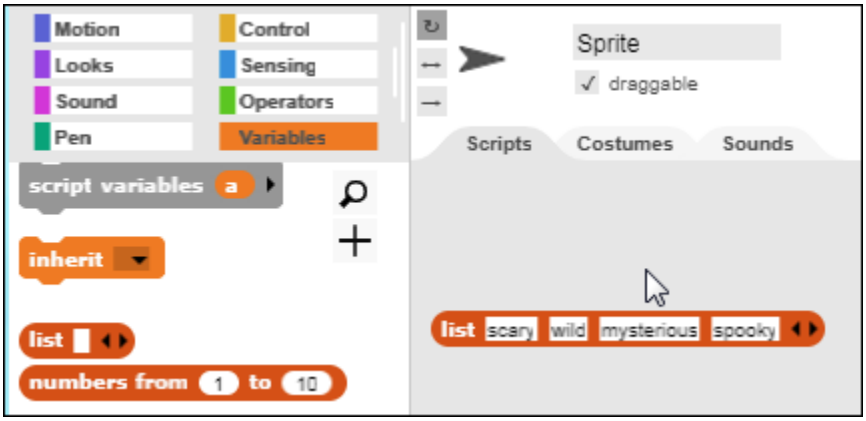
One frequently heard complaint from language arts teachers is that children use boring adjectives. One teacher calls them "the terrible ten" - nice, okay, good, etc. Computer poetry offers an opportunity to learn about parts of speech and explore a more expressive vocabulary. The basic form of a poetry generator used in one teacher's class for Halloween looks like this.

Print Sentence [Halloween is] Adjective
Print Sentence [Halloween is] Now
Print Sentence [Place]

The poem generator generates a sentence such as "Halloween is *Some Adjective*" by picking an adjective from a list of adjectives.

Making a List of Adjectives

The Halloween poem begins by entering adjectives like "scary," "wild," "mysterious," and "spooky" into a list.



The words in the list are assigned to a variable named *Adjective List*.

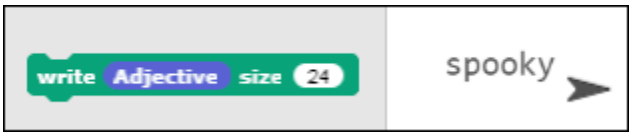
The *Pick* procedure is used to pick an item from the *Adjective* List and report the outcome. In the example shown in the illustration, the word "mysterious" has been picked from the list of adjectives.

This capability creates a procedure that picks an item from a list and reports the result.

Writing Words

The *Write* command (found under the *Pen* section of the commands palette) causes the turtle to write a word on the stage.

The command *Write Adjective* writes an adjective randomly selected from a list.



Joining a Phrase with a Random Word

The *Join* command joins a phrase such as "Halloween is" with the randomly picked adjective to form a sentence.

The addition of a third input to the *Join* command provides a space for inclusion of a period at the end of the sentence.

Printing a Sentence

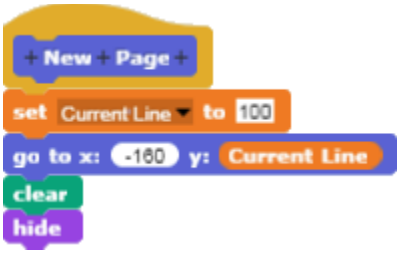
All of the elements for construction of a command to print a sentence are now in place. The *Print Sentence* command prints a sentence consisting of a phrase, a random word, and punctuation at the end of the sentence.

The command that prints the first line of the Halloween poem now looks like this.

To print the next line of the poem, the turtle must be repositioned. The *Print Sentence* procedure leaves the turtle positioned at the end of the sentence. It must be moved back to the left side of the screen, just below the beginning of the first sentence.

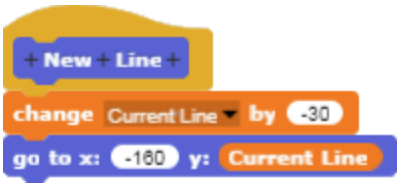
Beginning a New Page

At this point, it would be helpful to position the turtle at the top left-hand side of the screen. In this instance, the turtle will move to the X-coordinate of -160 (i.e., the far left side of the screen) and at the Y-coordinate of 100 (i.e., near the top of the screen). The *New Page* procedure then clears the screen and hides the turtle.



Beginning a New Line

After the first sentence is printed, the *New Line* procedure moves the turtle back to the left side of the screen and down about 30 steps (by decreasing the assigned value of *Current Line* by 30) so that the turtle is now directly beneath the beginning of the first line.



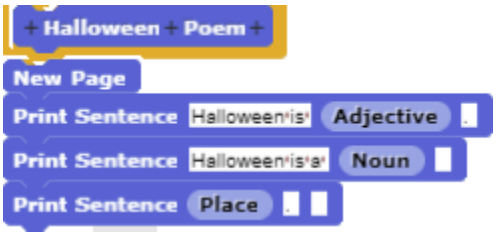
Incorporate the *New Line* procedure into the *Print Sentence* procedure so that after printing a sentence, the turtle moves to the next line and is positioned to print the next sentence.

The Poem Procedure

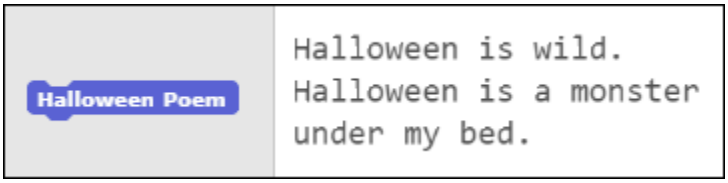
The completed Halloween poem generator clears the page, prints a sentence "Halloween is" *Random Adjective*, followed by the sentence "Halloween is a" *Random Noun"* and a "*Random Place"* on the last line of the poem.

```
To Halloween Poem
    New Page
    Print Sentence "Halloween is" Adjective
    Print Sentence "Halloween is a" Noun
    Print Sentence Place
End
```

The actual Halloween Poem procedure in *Snap!* looks like this.



The Halloween poem generator will produce poems like the one in the illustration.



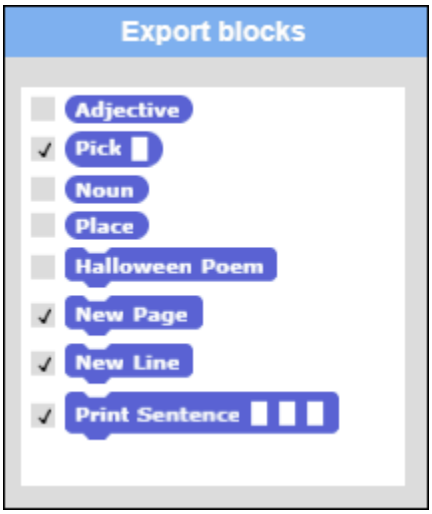
When this activity is undertaken in a group, it is important to solicit a contribution from each participant. Each time a new poem is generated, every member will watch to see if their word appears.

Tool Libraries

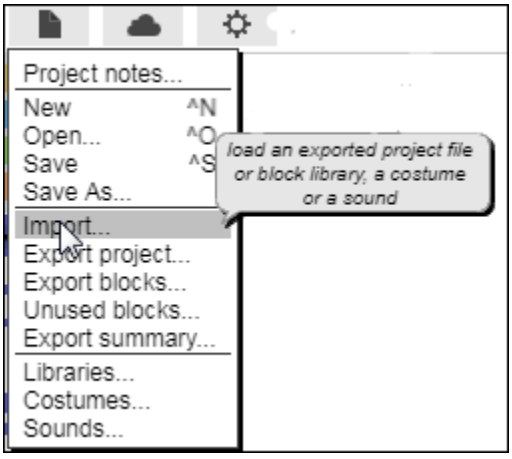
Often the first task in any computational activity involves development of tools to simply the problem. This takes more time in the beginning, but can save time later and make the work more efficient. In this illustration, four tools were developed

1. Pick Item
2. Print Sentence
3. New Page
4. New Line

These tools are general-purpose procedures that can be used in many different language arts activities. *Snap!* provides a mechanism to save a library of tools for reuse in other projects. The *Export Blocks* command is an option found under the *Snap!* File menu.



Once a set of *Snap!* blocks has been exported and saved, the saved block library can be reloaded for use in other projects by using the Import command (found under the *File* menu).



The ability to create libraries of reusable tools is one of the most powerful features of *Snap!*.

Computational Thinking: Abstraction

The ablity to create libraries of functions for projects in this manner is an important aspect of computational thinking known as *abstraction*. Short-term memory constraints of people significantly limit their ability to retain all of the details of a computer program. Abstraction buries the details of a procedures so that once a procedure is developed and tested, the programmer only has to remember the top level functions.

Thus, once the *Pick Item* procedure is developed, the developer no longer has to remember the details of how this procedure works each time it is used. This makes construction of much more complex programs possible than otherwise would be feasible.

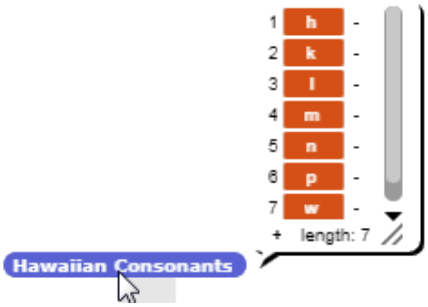
The Structure of Words

Words have an internal structure just as words and sentences do. Often this structure consists of alternating consonants and vowels. The consonant (C) and vowel (V) structure can be represented in the following way.

CVCV

The types of vowels and consonants that make up a language affect the kinds of words that are formed. For example, the Hawaiian language consists of seven consonants (h k l m n p w) and five vowels (a e i o u). The following procedures report Hawaiian consonants and vowels.

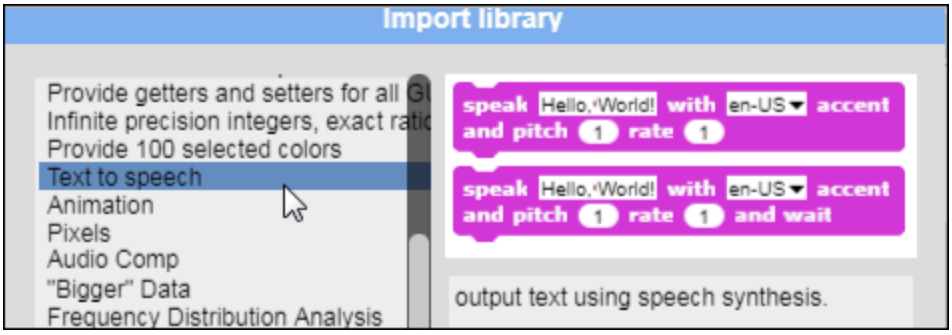
Clicking on the *Hawaiian Consonants* block displays a list of Hawaiian consonants.



The following Hawaiian Word procedure generates a pseudo-Hawaaian word. It joins together a series of alternating Hawaiian consonants and vowels to create a word formed with the twelve letters of the Hawaaian alphabet. Not all of the words formed are actually Hawaiaan words – hence the term psuedo-Hawaaian.

When the Hawaiian Word block is clicked, it outputs a pseudo-Hawaiian word formed from the letters of the Hawaiian alphabet. Because the Hawaiian alphabet has fewer letters than many other languages, the average length of words is longer.

The *Snap!* external library includes a *Text to Speech* option with commands that output speech



The text-to-speech command can be used to speak the pseudo-Hawaiian words generated.

English words, unlike Hawaiian, frequently begin with consonant blends or clusters. Examples of blends which occur frequently in English (and therefore in spelling lessons) are BR, SL, ST, PL, etc.

To adapt the word generator to English, first, change the consonants to include sounds that often occur in English blends (especially R, L, and S). The word generator will generate pseudo-English words that may not actually occur in English, but include consonants that often appear in English words.

To include consonant blends, adjust the structure to include the following combination:

C C V C V

with two consonants at the beginning. This will print words like “stila”.

Other combinations such as “tp” are not possible in English.

Even without formally studying the syllable structures and sound combinations of English, native speakers have knowledge about these structures. A native speaker of the language intuitively knows which of the samples are well-formed without reference to explicit rules. An exercise which emphasizes the consonant blends of English can take advantage of the intuitions about language. Students can test randomly generated CCV combinations, using their own intuitions and a guide:

Combination *English Blend?*

_____ Yes or No

_____ Yes or No

The goal is to make participants more aware of their intuitive knowledge of English. This can make "rules" seem more obvious and easier to use.

Pig Latin

Children often enjoy creation of secret languages that mystify others who do not know the secret of the language. Pig Latin is one of the more common secret languages of this kind. Pig Latin has three rules.

1. If a word begins with a single consonant, move the consonant to the end of the word and add "ay": *pig* becomes *ig-pay*.

2. If a word begins with a double consonant, move both consonants to the end of the word and add "ay": *store* becomes *ore-stay*.

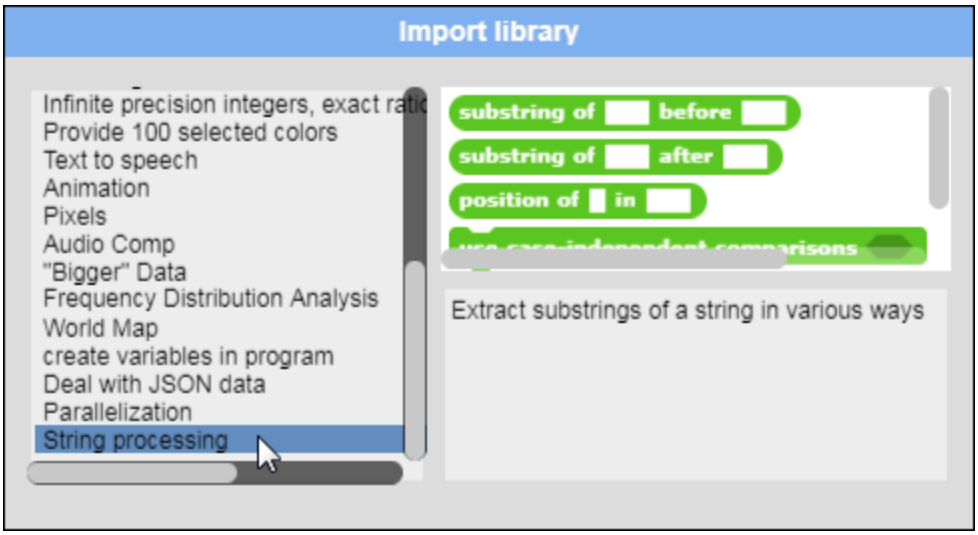
3. If a word begins with a vowel, add "ay" to the end of the word: *egg* becomes *egg-ay*.

Case 3. Words Beginning with a Vowel

The third case –words beginning with a vowel – is straightforward. The *Join* function can join the word ("egg" in this case) with the letters "ay" at the end.

Case 2. Words Beginning with a Single Consonant

The first case – words beginning with a single consonant – requires String Processing commands that can be imported from the *Snap!* library.



The string processing commands process strings of letters in the same way that list processing commands process lists of words. In the case of the word "pig" the result should be:

a. everything but the first letter: "-ig",
b. plus the first letter: "-p", followed by
c. the letters "-ay".

The resulting word in Pig Latin is: "ig-pay".

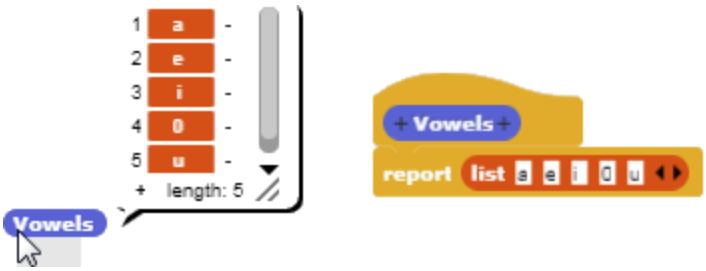
The *Letter* command extracts the first letter of the word:

The *Substring* command extracts everything *but* the first letter:

The *Join* command joins the resulting strings together and combines them with "ay" at the end.

Distinguishing Vowels from Consonants

In order to determine which method to apply, it is necessary to distinguish vowels from consonants. The *Vowels* procedure reports a list of all of the letters that are vowels.

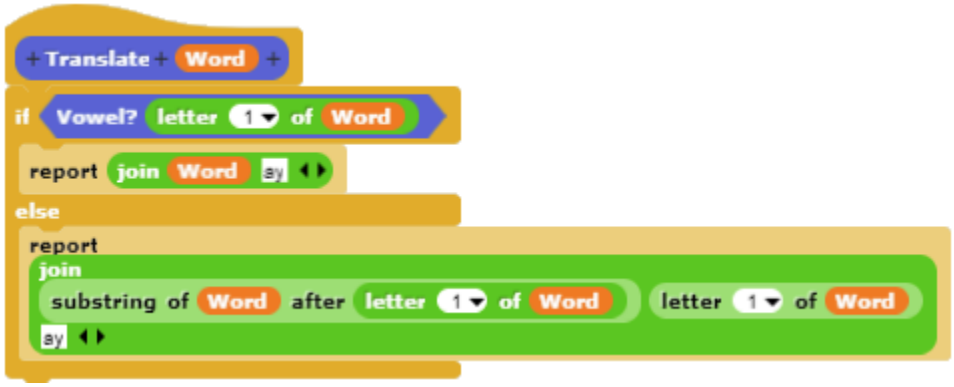


The *Vowel?* Procedure uses the list of vowels to determine whether a letter is a vowel. It reports true if the letter is a vowel and false if the letter is not a vowel.

The Translate Procedure

The *Translate* procedure employs the previously developed methods to translate words into Pig Latin. It uses the *Vowel* procedure to determine if the first letter of the word is a vowel.

1. If the first letter is a vowel, the procedure adds "-ay" the end of the word.
2. If the first letter is not a vowel, it moves the first letter to the end and adds "-ay."



Thus *pig* becomes *ig-pay* and *egg* becomes *egg-ay*.

Extensions

There are a number of enhancements that could extend the basic translation method. For example, the procedure could be extended to check for two consonants at beginning of the word, handling this case in an appropriate way. Another extension could be developed to translate lists of words to form sentences. And, of course, other secret languages could be developed using these same methods.

CHAPTER 5

Graphing Social Data

Glen Bull and Joe Garofalo

The fourth module (*Words and Lists*) introduced lists of words that can be combined to create sentences, poems, and stories. Computing also frequently makes use of lists of numbers. The story of the evolution of the nation is frequently told through both words and numbers.

The electrification of the nation represented one of its most important milestones. The National Academy of Engineers ranked this as the greatest engineering accomplishment of the twentieth century. Electricity is woven into almost every aspect of our civilization, from cell phones to computers. The way in which electrical networks were adopted is a technological story, but it is also a sociological story. For example, adoption of electricity had an impact on entry of women in the workforce.

In the course of writing a biography about President Lyndon Johnson, Robert Caro interviewed women in Texas hill country about their lives before Congressman Johnson brought electricity to that region. He found that their lives prior to electricity had consisted of unending toil. They brought up water "bucket by bucket, from deep wells, since there were no electric pumps; carried it on wooden yokes two buckets at a time; did the wash by hand, since without electricity there were no washing machines … spending an entire day of doing loads of wash, and the next day, since there were no electric irons, doing the ironing with heavy wedges of iron that had to be continually reheated on a blazing hot wood stove." (Robert Caro, *Working*, 2019)

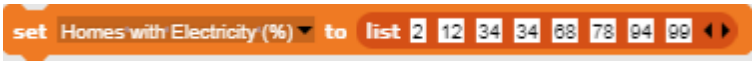
Investigating Rates of Adoption

Thomas Edison developed the first electrical network, beginning with construction of the Pearl Street Power in lower Manhattan in 1882. By the turn of the century, in 1900, only two percent of households in the United States had access to electricity. Even by 1920, only one-third of households had access to electricity. By 1960, almost all homes had electricity.

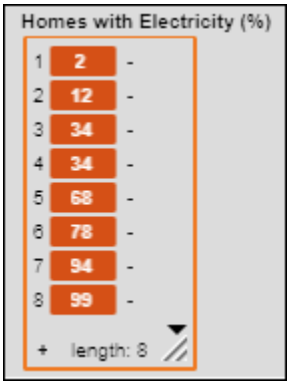
Table 1	
Year	*%*
1900	2
1910	12
1920	34
1930	68
1940	78
1950	94
1960	99

Table 1. Percentage of Households with Electricity

A list of the percentage of homes with electricity can be created in *Snap!*. The variable *Homes with Electricity %* was also created. The values in the list were then assigned to this variable.



After this list of values is assigned to the variable, the values are displayed in a table on the *Snap!* stage.

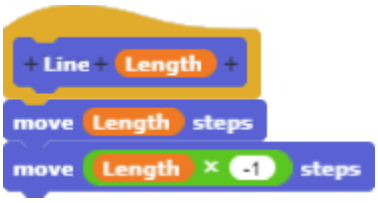


The Line and Over Procedures

A line graph to display these percentages visually can be constructed with two procedures:

1. a *Line* procedure uses the turtle to draw a line that is the same height as the number, and
2. an *Over* procedure to move the turtle over a specified number of steps after it draws each line.

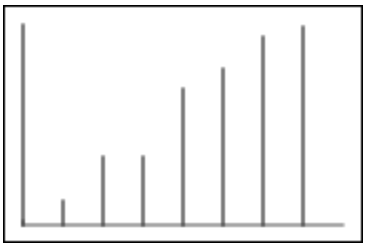
The *Line* procedure is straightforward. It moves the turtle forward a specified number of steps and then back the same number of steps.



The *Over* procedure is equally straightforward. The turtle turns right 90 degrees, moves forward a specified distance, and then turns left 90 degrees.

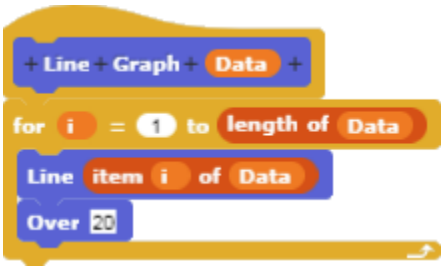
The Line Graph Procedure

The *Line Graph* procedure is three lines long, using the *Line* and *Over* procedures. It produces a graph of the data that looks like the illustration.

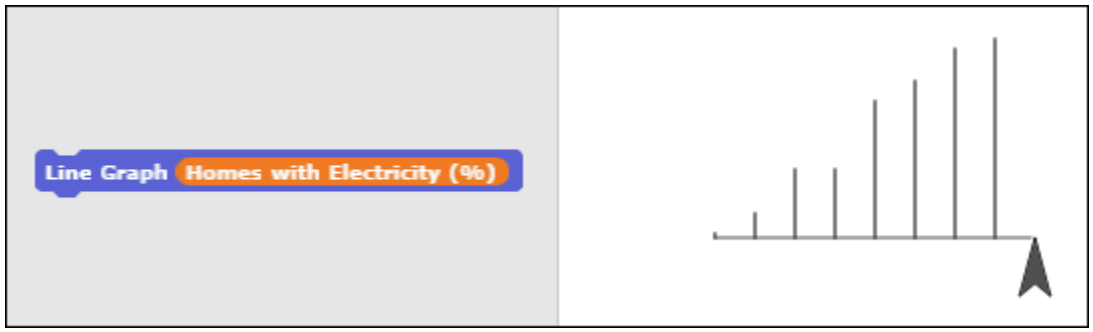


The *Length of List* command that was previously used to identify the length of a list of words (in Module 4: *Words and Lists*) is now used to determine the length of a list of numbers. In this case, there are seven numbers in the list. For each number in the list, from Item 1 to Item 7, the *Line Graph* procedure:

1. Draws a line whose length corresponds to the value of the number for that item, and
2. Moves over 20 steps in preparation for drawing the next line.

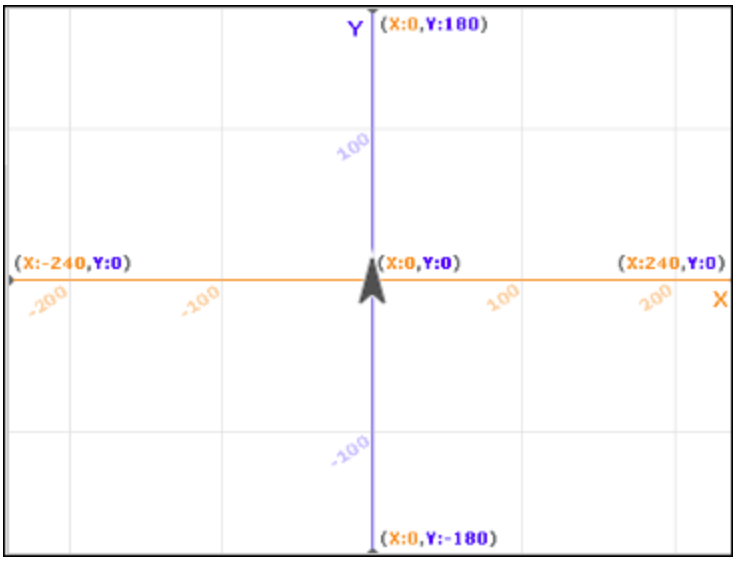


The result looks like this.



Snap! Coordinate System

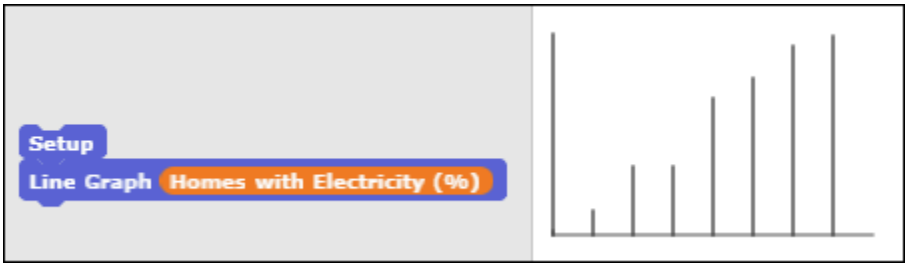
The coordinate system for the default *Snap!* stage spans the distance from -240 to +240 on the horizontal (X) axis, and from -180 to +180 on the vertical (Y) axis.



The Setup Procedure

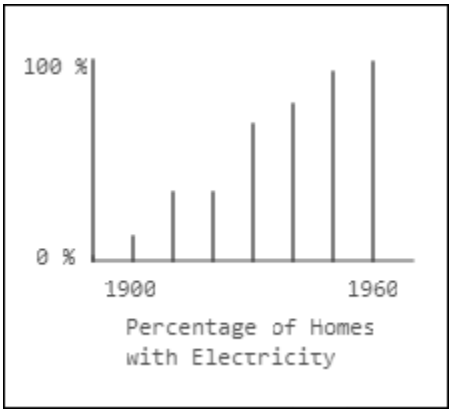
A *Setup* procedure is useful in positioning the turtle on the screen before the graph is drawn. The Setup procedure positions the turtle at -120 on the X-axis (near the left side of the screen) and at -40 on the Y-Axis (just below the middle). It assigns the values in the data list to the variable *Homes with Electricity*, *clears* the screen, and puts the pen down.

The *Setup* procedure also draws a vertical axis on the graph that is 100 steps in length with the following result.



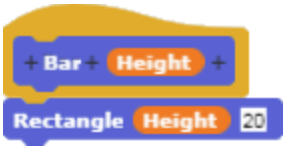
Labeling the graph to show the year associated with each line on the graph and the percentage of homes with electricity that the line represents is a further enhancement that could be implemented. The *Write* command used in the *Word and Lists* module could be used to label the graph.

The completed line graph with label might look like this.

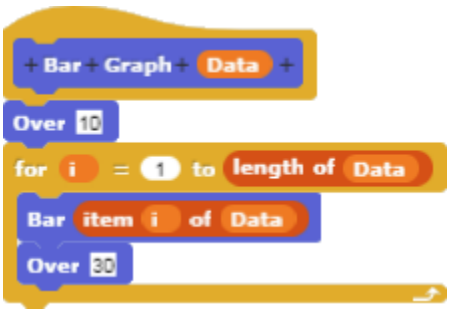


Bar Graph

A bar graph is similar to a line graph, but – as its name implies – bars are used in place of lines to chart the data. A *Bar* procedure can be created using a *Rectangle* procedure. The *Rectangle* procedure should have two inputs: width and height. (See the first module, *Turtle Graphics*, for more information about creating a procedure that draws a rectangle.) In this instance, the width of the bar has been set to 20 steps in the *Bar* procedure.

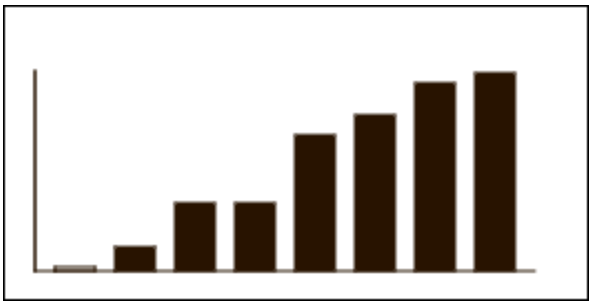


The *Bar Graph* procedure is similar to the *Line Graph* procedure, except that the *Line* procedure is replaced with the *Bar* procedure.



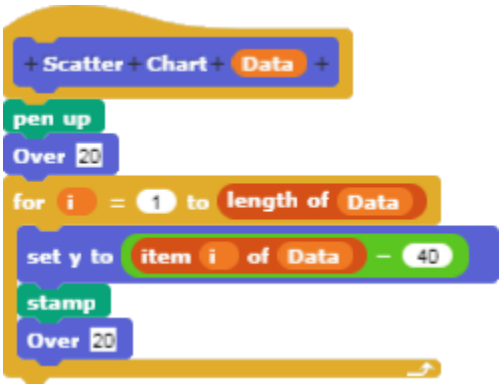
Executing the B*ar Graph* procedure using the *Homes with Electricity* list creates a graph that looks like this.

The *Fill* command (found under the Pen section of the commands palette) can be used to create solid bars.

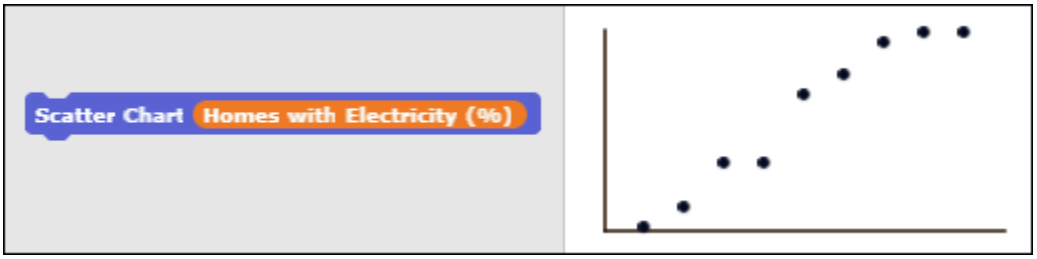


Scatterplot

Another type of chart is a scatterplot. The scatterplot in the example below marks the location of each data point with a dot. The Y-coordinate (height) of the turtle is set to the value (i.e., percentage of households) of each item in the table. The turtle's costume is changed to a dot; the *Stamp* command (found under the *Pen* section of the command palette) is used to stamp a copy of the dot at each location. Because the initial Y-coordinate of the turtle is set to -40 in the *Setup* procedure, this amount is subtracted from the position of each Y-coordinate in the chart. (This type of adjustment is known an *offset*.)

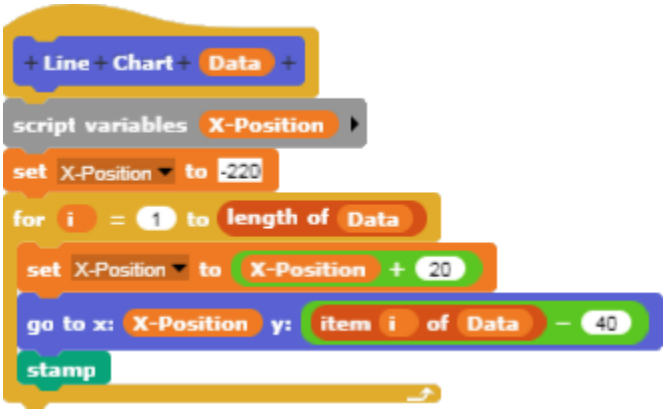


The resulting graph looks like this.

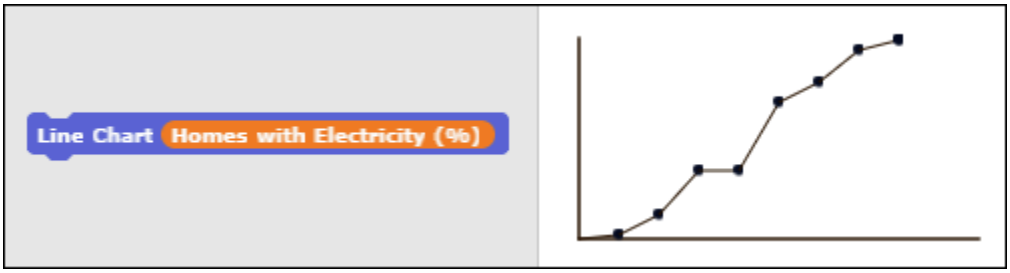


Line Chart

In some cases, the dots are connected with a line. This type of graph is known as a *line chart*. The procedure for a *Line Chart* is similar to the *Scatter Chart* except that the turtle's pen is down as it moves from dot to dot, drawing a line between each dot as the turtle moves from one to the next. The *Go To X Y* command is used to move the turtle from one dot to the next. The value of the X-Coordinate is increased by 20, moving the turtle over horizontally between each dot.



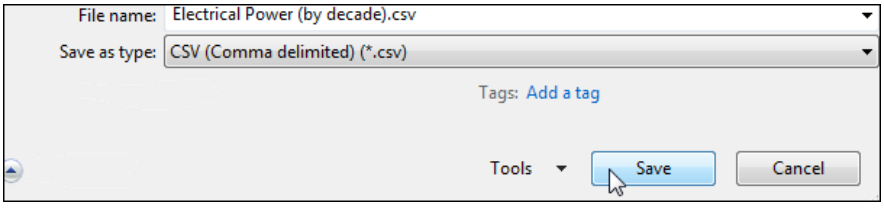
The line chart with connected markers looks like this.



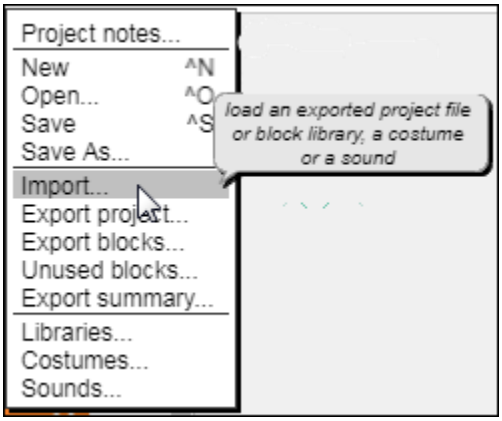
A scatterplot with connected lines implies that changes during each decade were linear. In instances in which the changes are not linear, it may be not be appropriate to connect the data points with lines.

Importing Data

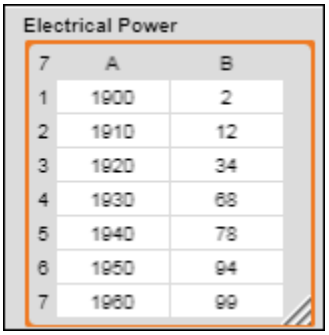
In many cases, data is available in the form of a spreadsheet. The form in which the data for adoption of electrical power in the U.S. was originally acquired in the form of a spreadsheet. Data in spreadsheets can be imported into *Snap!* in the form of a list of *Comma Separated Values* (CSV format). Most spreadsheets offer the option of saving data in this form.



The CSV file can then be imported into *Snap!*



The data in the imported file looked like this.



Indexing Lists of Lists

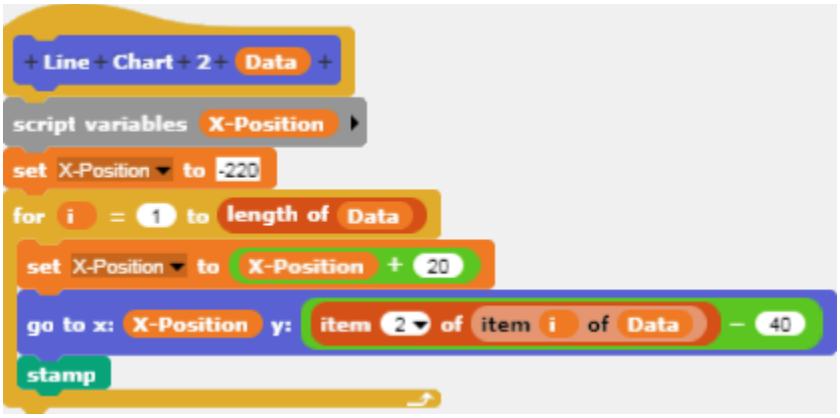
This table has seven rows (1 through 7), with two items in each row (year and percent). From the perspective of *Snap!*, the data is visualized as a list of seven additional two-item lists (numbered 1 through 7 in the illustration).

Calling Item 4 in the list yields two items: the year and the percent:

Calling Item 1 of Item 4 yields the year.

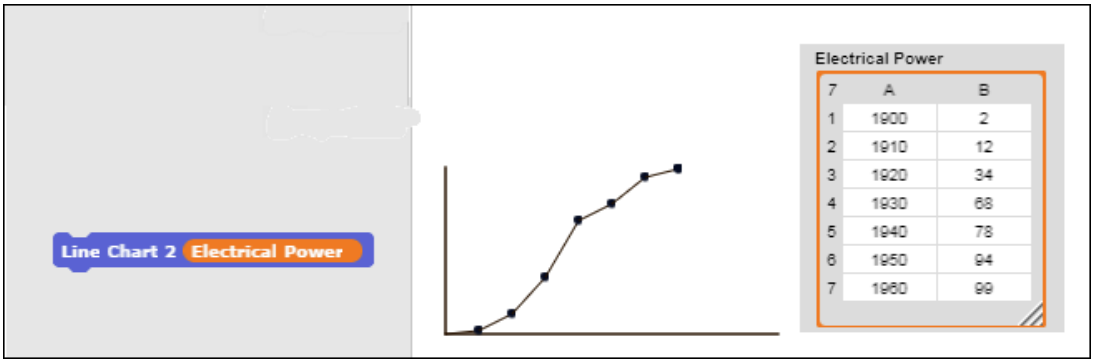
Calling Item 2 of Item 4 yields the percentage associated with that year.

A second procedure, *Line Chart 2*, to access the second item in each list *[Year, %]* would be written this way.



The ability to work with lists of lists in this manner is a first step to working with complex data structures that are often found in computing.

Execution of the *Line Chart 2* procedure produces a graph that looks like this.

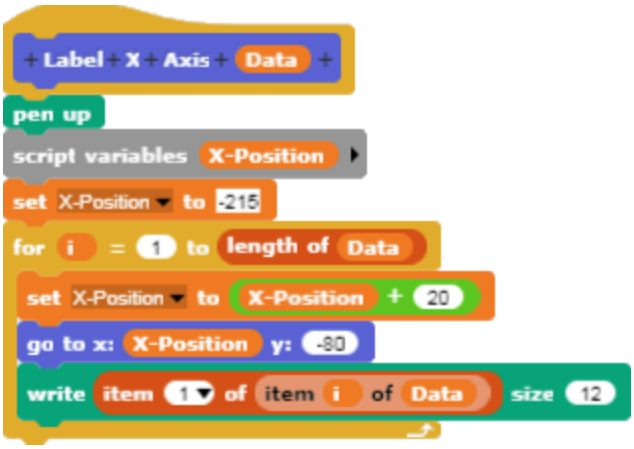


Labeling the Graph

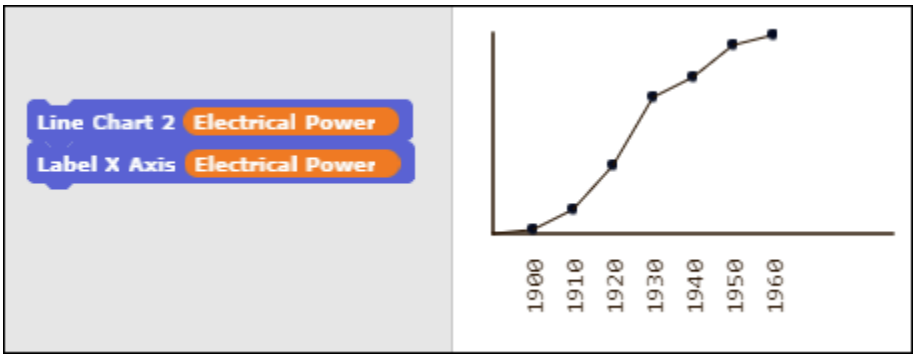
The inclusion of the year associated with each percentage makes it possible to label the X-axis (i.e., the horizontal axis) of the graph. The *Line Chart 2* procedure can be adapted to label the X-axis by making three modifications in the procedure.

1. The year is accessed by selecting *Item 1 of [Item i of Electrical Power]*.
2. The turtle is positioned below the X-axis (at the Y-coordinate of -80).
3. Instead of stamping the turtle to leave a dot, the *Write* command is used to label the year.

In *Snap!*, these modifications would be expressed in the following way.



The result of executing the *Line Chart 2* and *Label X Axis* procedures looks like this.



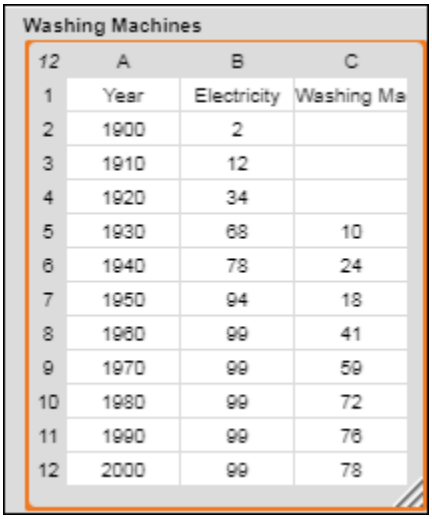
Electricity brought immediate benefits to households, most notably electric lights that extended the day and electric water pumps that made it no longer necessary to haul water. The U.S. Department of Agriculture estimates that the average rural family of five individuals required approximately 40 gallons of water per person or about 200 gallons of water per day. A gallon of water weighs slightly more than 8 pounds. Hence, a family hauled more than 1,600 pounds of water per day (nearly a ton). Electrical water pumps eliminated this back-breaking chore.

Adoption Rates for Washing Machines

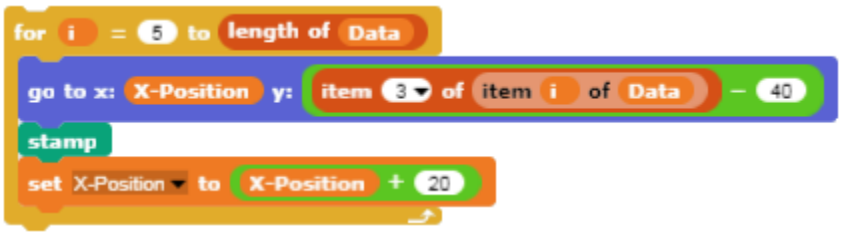
Acquisition of other electrical appliances took longer. The first washing machines sold for the equivalent of about a thousand dollars in today's currency, a sizeable percentage of the average family income at the time. Consequently, although nearly three-fourths of households had electricity by 1940, only about one-fourth of the families could afford a washing machine.

A	B	C
Year	Electricity	Washing Machines
1900	2	
1910	12	
1920	34	
1930	68	10
1940	78	24
1950	94	18
1960	99	41
1970	99	59
1980	99	72
1990	99	76
2000	99	78

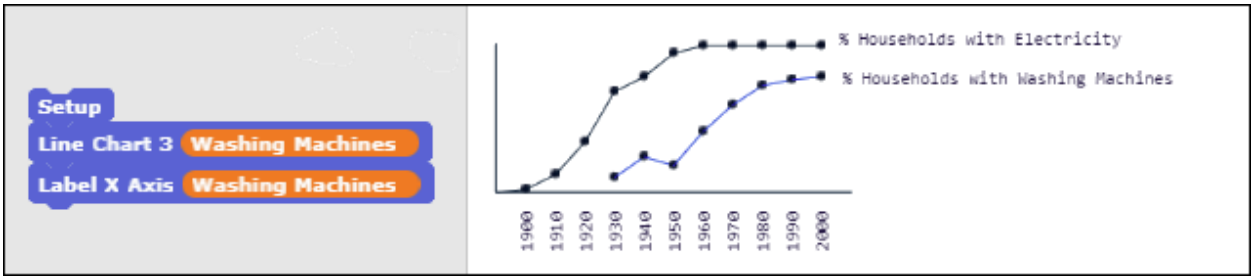
This data looks like this when it is imported into Snap!



A procedure to plot adoption of washing machines would access *Item 3 of Item I of the Data*. Also, since the data for adoption of washing machines only begins in 1930 (row 5 of the table), the initialization of the *For* loop begins with 5: *For i = 5 to Length of Data*.



If this code is added to the Line Chart 3 procedure, a plot like this can be produced.



Social Implications

Today nearly every household in U.S. either has a washing machine or has access to one in a nearby laundromat. Hans Rosling, a Swedish professor of global health who taught at the Karolinska Institute in Sweden, described the impact of the washing machine that his family acquired when he was four years hold. Rosling's mother, freed from the labor of hand washing, used the freed time to take him to the library and read to him. Rosling noted that only about two billion of the seven billion people in the world can afford washing machines. The remaining five billion still wash their clothes by hand just as everyone did centuries before.

Adoption Rate Data

Data for technology adoption by households in the United States is available at:

https://ourworldindata.org/technology-adoption

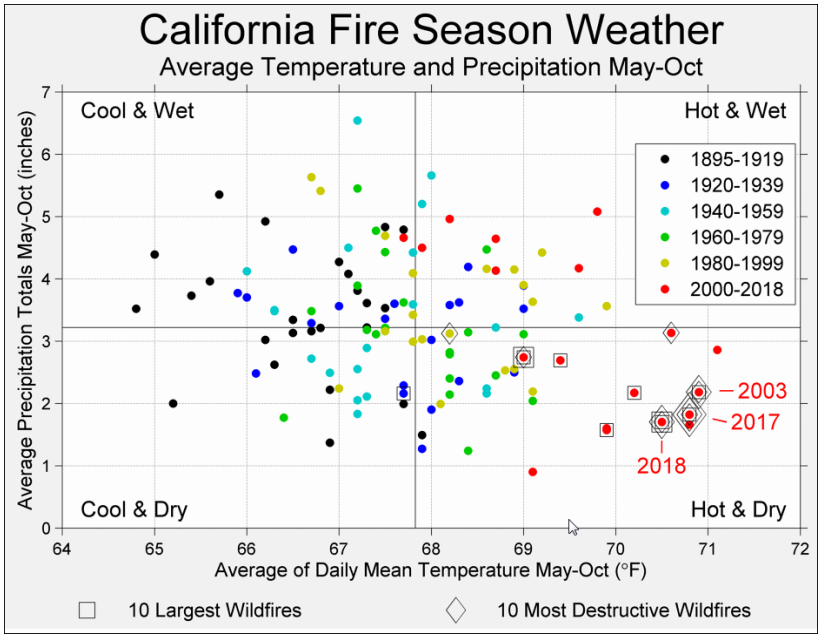
This site provides data for adoption rates such as refrigerators, computers, cell phones, etc. in the form of a spreadsheet. Subsets of the data can be exported in CSV format for analysis and display in *Snap!* using procedures similar to the ones described above.

Computer Languages and Computational Thinking

All of the plots described above could be quickly generated in a spreadsheet, raising the question, "Why use a computer language to accomplish this goal?" There are several benefits:

1. Tools like spreadsheets are written in a computer language. Analyzing the methods used to process data in this way provides a look under the hood.
2. Special-purpose tools like spreadsheets have boundaries. A general-purpose computing language can accomplish anything that that can be imagined if the user is proficient enough.

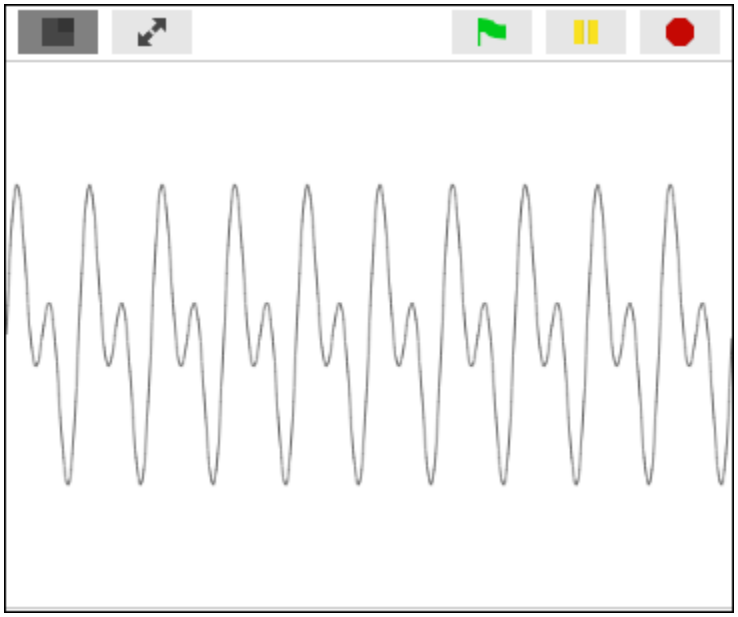
For example, a graph depicting the relationship between (1) Precipitation, (2) Temperature, and (3) Incidence of Fires in California. The graph displays temperature on the horizontal axis and the precipitation on the vertical axis. To display the dimension of time, the graph fills in the scatterplot over time.



Source: http://berkeleyearth.lbl.gov/downloads/CaliforniaFire.gif

This type of graph can be created with a computer program and tailored to the exact need.

3. The data processing methods learned provides scaffolding for other, more complex activities. For example, a subsequent module will explore sound and music. In the process, sound acquired from the microphone will be stored in a list, analyzed, and graphed.



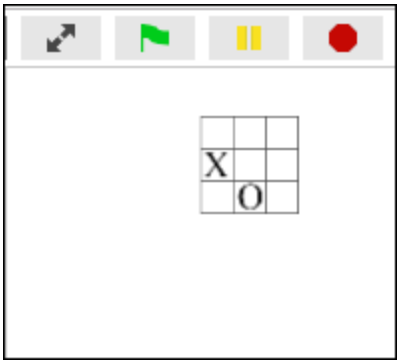
Experienced programmers develop a re-usable tool set over time. These tools initially take more time to develop. As a set of tools is developed, however, they can be reused in many different applications, saving time in the long run.

CHAPTER 6

Board Games

Glen Bull

Many board games involve moving tokens across a grid of squares. Chess, checkers, tic-tac-toe, and *Battleship* all follow this format.

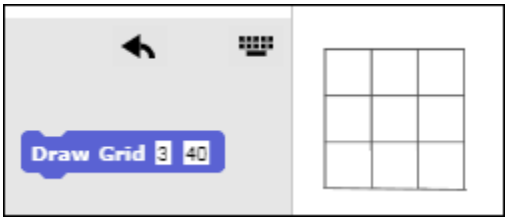


Reconstruction of tic-tac-toe as a digital game involves two basic coding tasks:

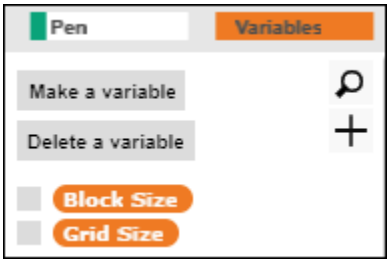
1. Drawing a three-by-three grid, and
2. Procedures to place the tokens.

Drawing a Tic Tac Toe Grid

The *Draw Grid* procedure was constructed using the *Line* and *Over* procedures developed for the prior module, *Graphing Data*. It has two inputs, *Blocks per Side* (3) and *Block Size* (i.e., the size of the blocks in the grid).



In this instance, the grid was constructed with a *Grid Size* of 3 (i.e., 3 rows and 3 columns) and a *Block Size* of 40. These values were assigned to global variables so that they could be used by other procedures.

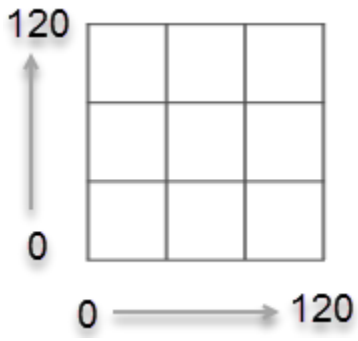


Creating and Identifying the Location of Game Tokens

The game tokens for tic-tac-toe consist of an "X" and an "O". Two sprites were created, one for each shape.

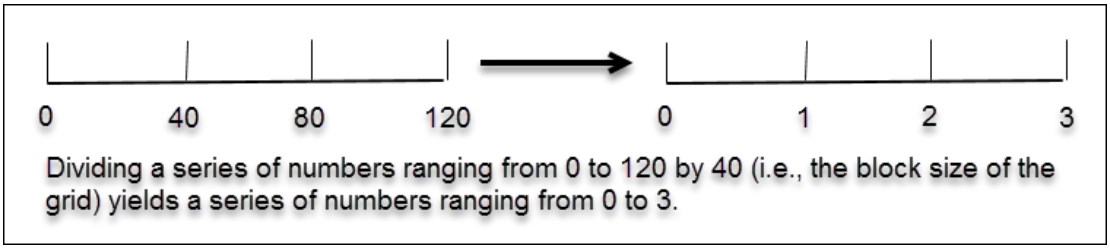
You can access a *Snap!* file with the tokens installed here: Battleship Kit.

The built-in *X Position* and *Y Position* commands (found under the *Move* section of the commands palette) can be used to identify the row and column in which a token has been placed.



A grid of three rows and columns with a block size of 40 steps spans the distance from 0 to 120. Dividing the X and Y coordinates by the block size yields the row and column in which a sprite is placed.

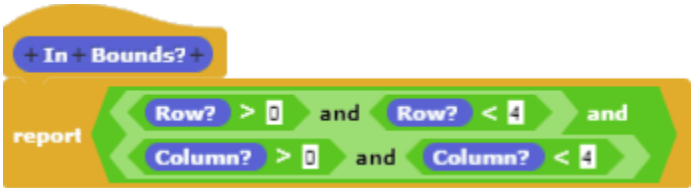
For example, if a token is placed at the X-coordinate of 60, dividing by 40 yields the result of 1.5. Rounding this result yields the integer 2. This result indicates that the token has been placed in the second column. Eventually the token will need to be placed in the center of the square.



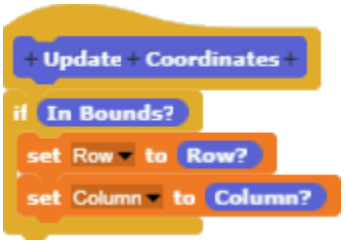
Numbers between 0 and 0.5 round to 0, which would place the token outside the grid. This can be corrected by adding an offset of 0.5 to yield the correct result of "Column 1". The code to accomplish this is shown in the *Column?* procedure.

A similar method is used to identify the row in which the token has been placed.

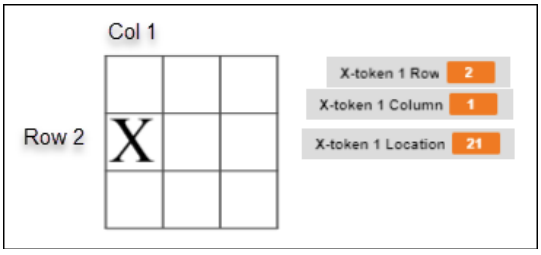
It will be important to know whether the token is on the grid or outside its boundaries. The *In Bounds?* procedure reports whether the result for row and column is between 1 and 3.



If the token is in bounds (i.e., on the game board), the *Update Coordinates* procedure assigns the row and column in which the token has been placed to the global variables *Row* and *Column*.

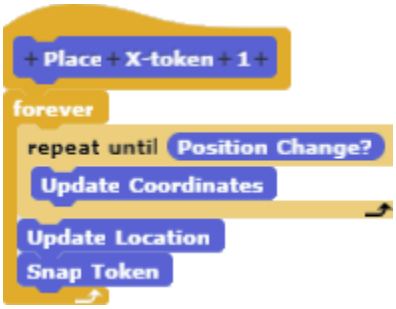


These values are used by the *Update Location* procedure to locate the token on the board. For example, if the token has been placed into Row 2 and Column 1, the *Update Location* procedure assigns a value of 21 to the token's *Location*.



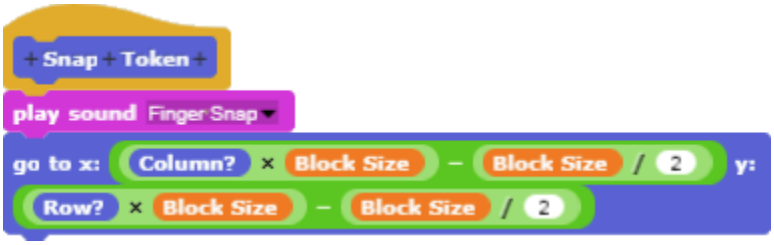
Place Token

The foregoing procedures are used to snap the token into place on the correct square of the game grid. The *Place Token* procedure repeatedly checks to see if the position of the token has been changed.



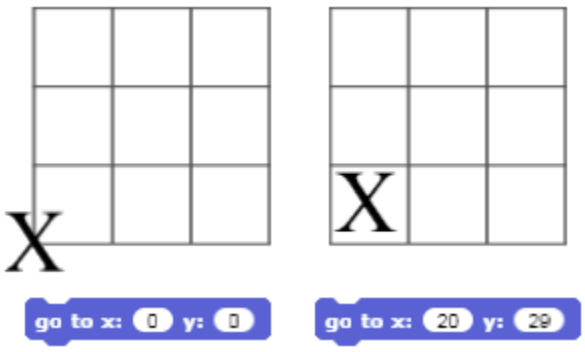
A position change can be detected by checking to see if the most recent coordinates (i.e., row and column) of the token match the previously recorded location of the token.

When a position change is detected, the recorded location is updated using the *Update Location* procedure, and the token is snapped into place on the new game square. The *Snap Token* procedure snaps the token into place.



The procedure provides a sound effect as feedback to indicate that the token is being snapped into place.

- The command *Go to X = 0, Y = 0* sends the token to the corner of the board.
- The command *Go to X = 20, Y = 20* sends the token to the center of the first square.



Therefore, multiplying the row number by the block size and then subtracting half the block size will place the token in the center of the row. For example, if the token is in Row 1:

Row 1 x 40 (i.e, the block size) = 40; 40 – 20 = X coordinate of 20

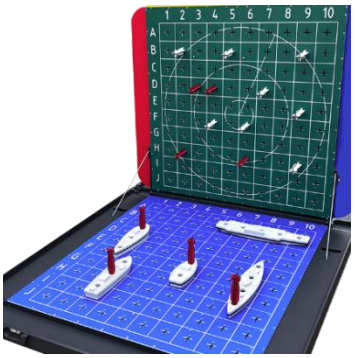
Similar calculations can be used to determine the center of the column.

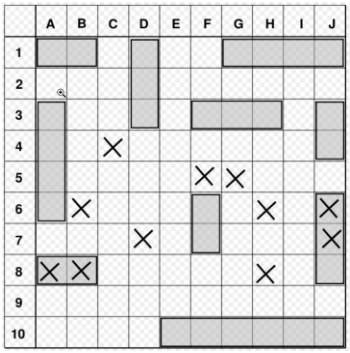
Column 1 x 40 (block size) = 40; 40 – 20 = Y coordinate of 20

Thus, the *Place Token* procedure aligns the token with the center of the square at the intersection of the row and column in which the token has been placed.

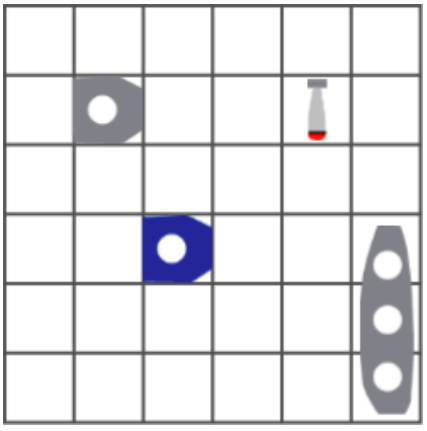
Battleship

Battleship is another board game that is played on a grid. A fleet of ships concealed from an opponent is placed on a grid. The opponent attempts to torpedo ships by calling out coordinates that are marked on the grid. *Battleship* has been played as a pencil-and-paper game on a grid of paper and has also been manufactured in several commercial versions with plastic pieces.





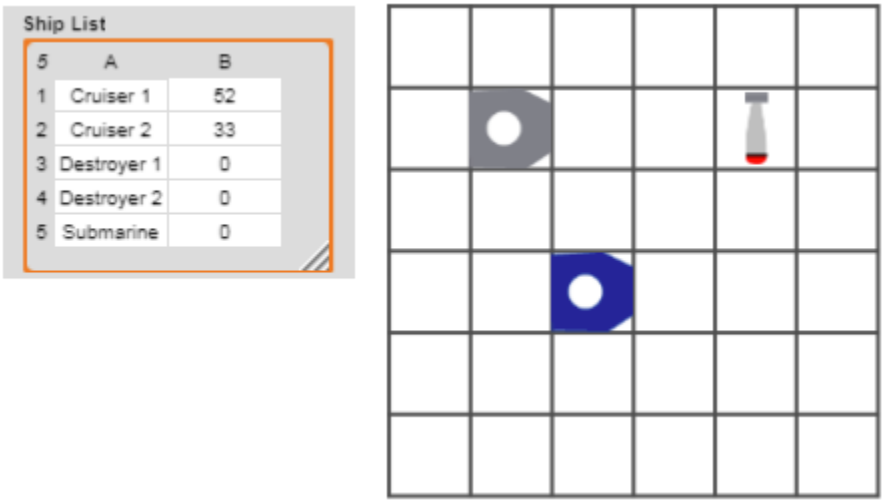
The *Snap!* version of *Battleship* uses ships and torpedoes placed on a grid drawn on the computer screen. The grid is similar to the tic-tac-toe grid except that it is a 6 x 6 grid instead of a 3 x 3 grid. The outline of various ships (cruisers, destroyers, submarines) are used in place of the "X" and "O" tokens used in the tic-tac-toe game. These images drawn with the paint program can be accessed here*:* Battleship Images



One player (or, optionally, the computer) places ships on the grid. The ships are then hidden (using the *Hide* command found under the *Looks* section of the command palette). The opposing player then places

a torpedo on a square in attempt to hit a ship. If a ship is hit, an explosion occurs. If a ship is not hit, an "X" is placed in the square to indicate that the square is empty.

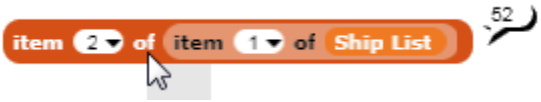
In order to determine if a ship has been hit, a list of squares on which ships have been placed will be needed. (Lists were first introduced in Module 4, *Words and Lists*. Additional attributes of lists were discussed in Module 5, *Graphing Social Data*.) In the illustrated example, the *Ship List* indicates that ships have been placed on *Square 33* and on *Square 52*.



The procedure to place *Cruiser 1* on the grid is identical to the *Place X-token* procedure (from the preceding section on tic-tac-toe), with one enhancement. After the token is snapped into place and the location is updated, the procedure records the location of C*ruiser 1* on the ship list.

Item 1 of Ship List consists of two items: (1) the name of the ship (Cruiser 1) and (2) the location of the ship.

Item 2 of *Item 1* of the *Ship List* points directly to the location of *Cruiser 1*:



Therefore, the command *Replace Item 2 of Item 1 of Ship List with Location* will update the value of the location of *Cruiser 1* with the location of the square on which it has been placed.

Collision Detection

The *Place Torpedo* procedure is identical to the *Place Ship* procedure except that at the end, the *Torpedo* procedure checks to see if there has been a collision with one of the ships.

The *Torpedo* procedure checks for collisions by checking to see if the *Torpedo's* current location is the same as the location of any of the ships on the *Ship List*. It accomplishes this by using the *List Contains Thing* command (found under the Variables section of the command palette) to see if the *Torpedo's* location is also listed among the locations recorded on the *Ship List*.

If the *Torpedo's* location and the location of a ship on the *Ship List* are the same, the sound of an explosion can be used to register that a hit has occurred.

Other details can extend the depth of the game. For example, some ships are larger and cross more than one square, so a method for recording all of the squares that a ship occupies is needed. Also, a method for determining when all the ships on the list have been destroyed is needed in order to determine when to end the game. However, these are housekeeping details that can be worked out once the basic game procedures have been implemented and tested.

CHAPTER 7

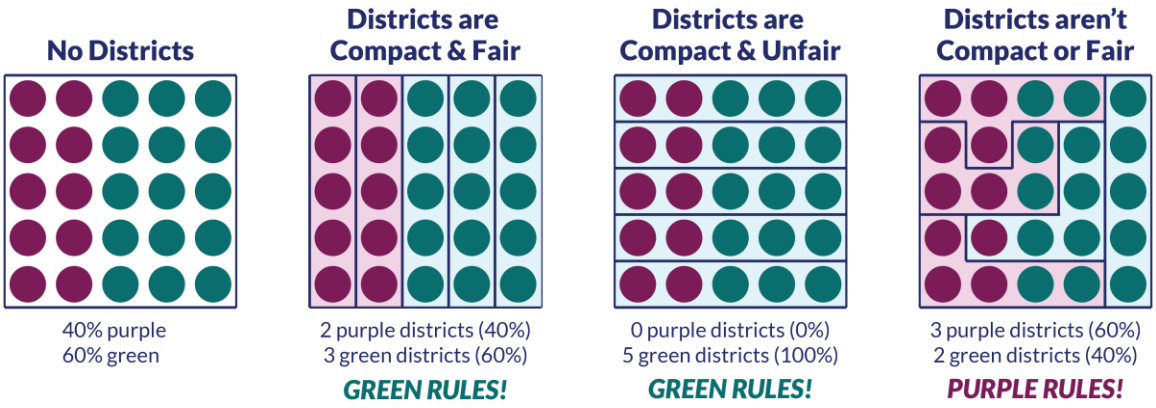
Simulations: The Gerrymandering Game

Glen Bull and Joe Garofalo

Congressional districts can still be drawn in a way that allows some votes to count more than others. This practice dates to the origins of the nation. In 1812, Governor Eldridge Gerry signed a bill that redistricted Massachusetts in a way that benefited his political party, the Jeffersonian Democratic-Republicans. As a result, although the Federalist party received more votes statewide, the state senate remained in the hands of the Democratic-Republican party. An article in the Boston Gazette, printed March 6, 1812, used the term *Gerrymandering* to describe the process of drawing a legislative district in a way that benefits one political party over another party.

The idea behind gerrymandering is fairly straightforward: one party, typically the party in power, packs their opponents' supporters together into very few districts. Then the party in power makes other districts relatively more balanced — but place enough of their supporters in most of these districts to give the party in power an advantage. The anticipated result is that the party in power loses a few districts hugely, yet wins a majority of districts comfortably.

The Virginia Public Access Project, a non-profit educational association, provides a simplified explanation of Gerrymandering on its website, illustrated in the graphic below.



Gerrymandering Simplified: https://education.vpap.org/section/redistricting/

Additional information about redistricting in Virginia is provided on the Virginia Public Access Project web site. A YouTube video by CGP Gray provides an animated explanation of the way in which Gerrymandering is used to ensure that some votes will count more than others.

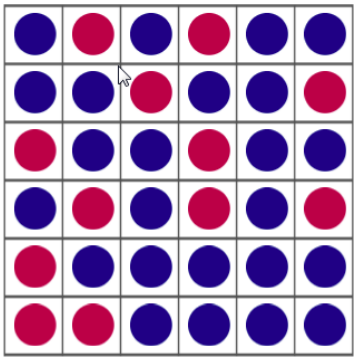
YouTube Video - Gerrymandering Explained:
https://youtu.be/Mky11UJb9AY

In the YouTube video, Gray uses hypothetical animal populations to describe ways in which district boundaries can be manipulated to achieve a desired political outcome.

Gerrymandering Simulation

The process of drawing district boundaries lies at the heart of democracy and representational government. The perceived legitimacy of a government rests upon the belief that each vote is counted equally. Extreme cases of Gerrymandering can represent a risk to democracy if voters perceive that the process is rigged. At the same time, it must be recognized that all parties will seek partisan advantage when given the opportunity.

The *Make to Learn Gerrymandering Game* simulates the gerrymander process through a computer program. In the simulation, a six-by-six grid represents a state with 36 counties. Counties with majority population voting for the Red party are designated with a red token. Counties with a majority population voting for the Blue party are designated with a blue token.



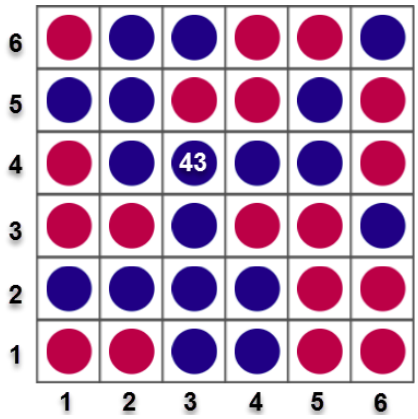
Twelve congressional districts consisting of three counties each are formed. The counties in each congressional district must be adjacent (i.e., touch one another). If two or more counties in the district are blue, the district will elect a congressman representing the Blue party. If two or more counties in the district are red, the district will elect a congressman representing the Red party. The goal of the Gerrymandering game is to draw the lines of the congressional districts in a way that benefits one party over another.

Gerrymandering Procedures

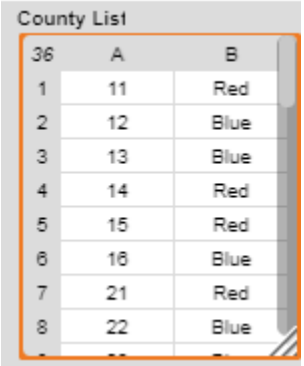
The basic gerrymandering board with tokens is provided at:

Gerrymandering Board

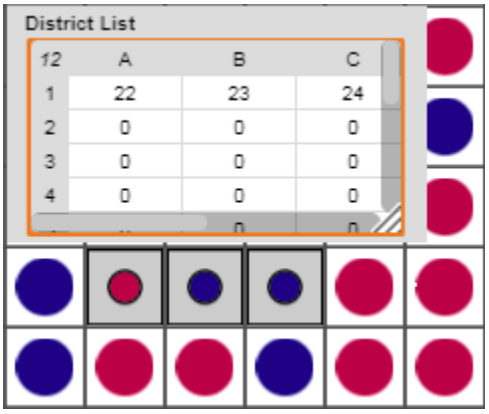
The Gerrymandering simulation uses the *Grid* procedure developed in the previous module (*Board Games*) to draw a six-by-six grid. A *Populate Grid* procedure places a randomly selected red or blue token in each of the 36 counties. The location of each county is designated by its row and column. For example, the location of the county at the intersection of the fourth row and third column would be designated as “Location 43”.



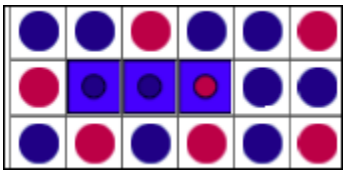
The variable *County List* is used to track the location of each of the 36 counties on the grid and the color of the token (red or blue) in each county.



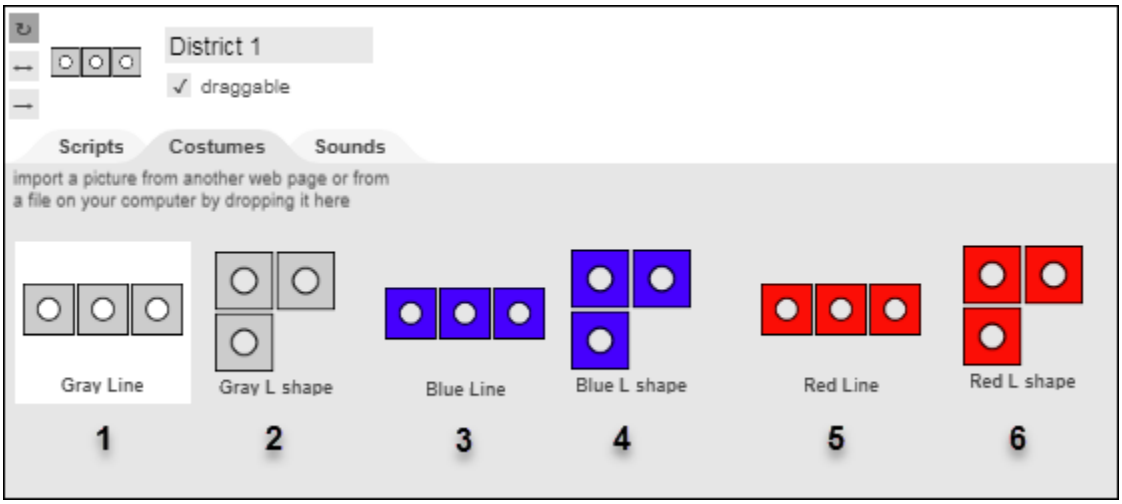
A second variable, *District List*, is used to track the location of each of the three counties in the voting district, and the status of the district (red or blue). For example, the illustration below indicates that *District 1* consists of the three counties located at positions 22, 23, and 24 on the grid.



Twelve district tokens represent the twelve voting districts. District tokens can be dragged onto the game board to create a district of three contiguous counties. The district token is initially a neutral gray shade before it is dragged onto the game board. Once it is aligned with a set of counties, a shade of blue is assigned to the voting district if two or more counties have populations with a blue majority. A shade of red is assigned to the voting district if two or more counties have populations with a red majority.



Each district token, therefore, has six costumes representing the possible shapes and shades of color: two in a neutral gray (linear and L-shaped districts), two in blue, and two in red.

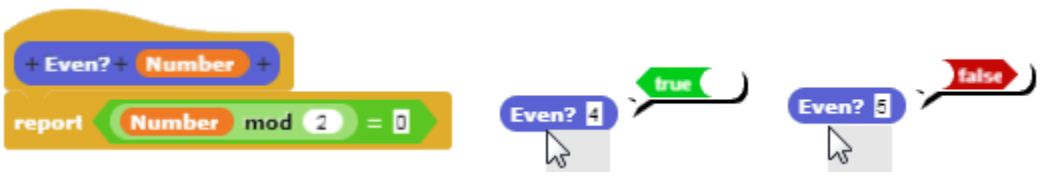


Each costume has a number assigned at the time of creation. The *Costume #* command (found under the *Looks* section of the commands palette) can be used to determine the costume worn by a sprite.

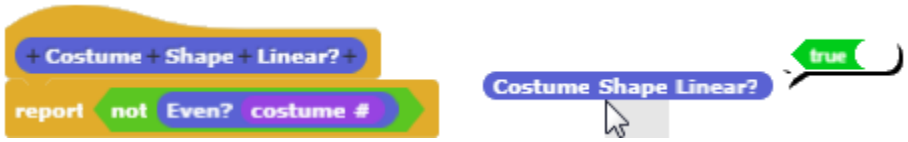
By design, the L-shaped district shapes have been assigned even numbers (2, 4, and 6) and the linear district shapes have been assigned odd numbers (1, 3, and 5). This design choice makes it possible to use the *modulo function* to determine the shape of the district. The modulo function can be used to determine if a number is even or odd. The modulo function returns the remainder that results when one number is divided by another. A whole number that is even will return a remainder of 0 when it is divided by the number 2. The modulo function is often used to create an *Even?* procedure:

```
To Even Number?
    Report Number Mod 2 = 0
End
```

This can be expressed in *Snap!* in the following way.



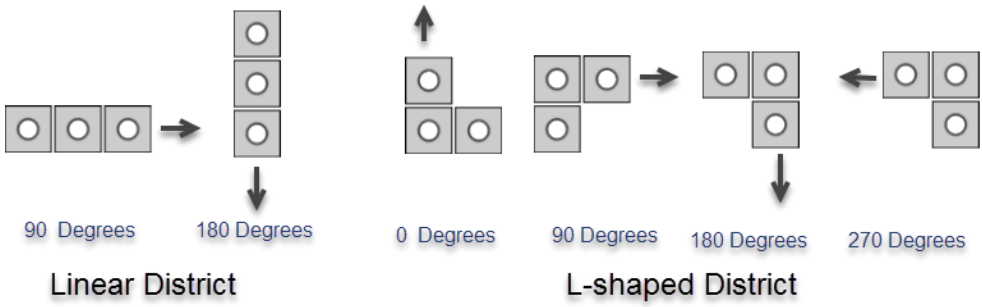
The *Even?* function can be used to determine if the costume shape worn by the sprite is linear. If the costume is not even, it must be a linear shape (Costume # 1, 3, or 5). The ability to determine the shape of a district will be useful in construction of the gerrymandering simulation.



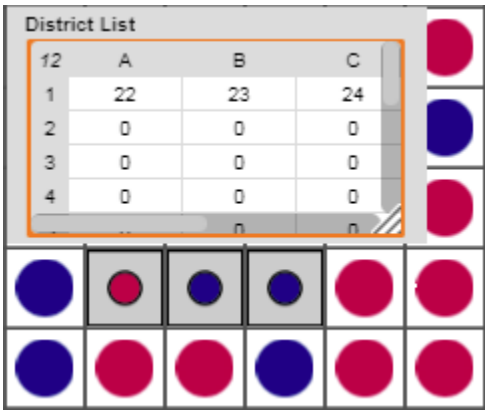
A district comprised of three contiguous counties can be formed in six different ways:

a. A linear district (either horizontally or vertically aligned) or
b. An L-shaped district (oriented in one of four directions.

The illustration depicts the six possibilities. Two shapes (linear and L-shaped) can be rotated to create all of the possibilities.



The shape and orientation of the currently selected district is needed to identify which three counties are covered by the district token. For example, the horizontal linear district placed at Location 22 also covers Location 23 and Location 24.

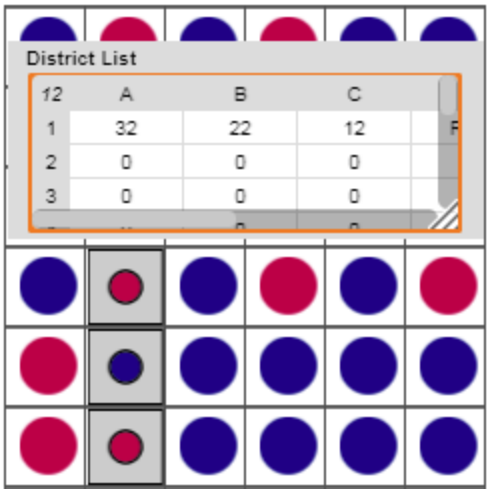


This can be expressed more generally in the following way.

A Horizontal Linear District Covers
a. Current Location
b. Location + 1 (i.e., one square to the right)
c. Location + 2 (i.e., two squares to the right)

Therefore, if the current location of the district token is Location 22, then it follows that Location 23 and Location 24 will also be covered.

In contrast a Vertical Linear district will cover the current location plus two counties directly below that location. In this example, the district token has been placed on Location 32. Therefore Location 22 and Location 12 are also covered.



This can be expressed more generally in the following way.

A Vertical Linear District Covers

a. Current Location
b. Location - 10
c. Location - 20

By using this approach, the area covered by a district token can be identified for each shape and orientation. The process of developing a numerical representation of a geographic area is known as *mapping*.

Area Covered by District Token					
	Shape	*Orientation*	*District Area*		
1	Line	90 Degrees	Location	+ 1	+ 2
2	Line	180 Degrees	Location	-10	-20
3	L shape	0 Degrees	Location	+ 1	+10
4	L shape	90 Degrees	Location	+ 1	-10
5	L shape	180 Degrees	Location	-1	-10
6	L shape	270 Degrees	Location	-1	+10

With the information in this table, it will be possible to determine the location of the three counties covered by each district token if the type of area (indicated by the combination of district shape and orientation) can be identified. A procedure to identify the shape of the district (linear vs. L-shape) was

previously developed: "*Costume Shape Linear?*". The orientation of the district token is provided by the command *Direction* (found under the *Motion* section of the commands palette). This information can be used to identify each of the area types (1 through 6).

The *Type of Area?* procedure can be implemented in the following way.

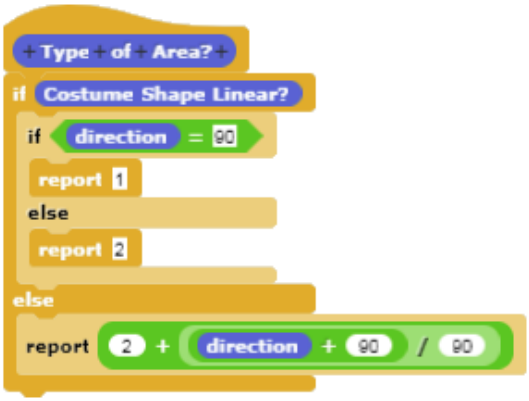
If the Costume Shape Is Linear
If the Direction = 90 Degrees Report 1
Otherwise Report 2
Otherwise Report 2 + (Direction + 90 / 90)

If the district area is linear and the direction is 90, the procedure outputs "1".
If the district area is linear and the direction is not 90, the procedure outputs "2".

Otherwise the procedure adds 90 to the current Direction and divides by 90.

1. If the direction is 0: (0 + 90) / 90 = 1; adding 2 to this result yields 3.
2. If the direction is 90: (90 + 90) / 90 = 2; adding 2 to this result yields 4.
3. If the direction is 180: (180 + 90) / 90 = 3; adding 2 to this result yields 5.
4. If the direction is 270: (270 + 90) / 90 = 4; adding 2 to this result yields 6.

This logic can be expressed in *Snap!* in the following way.



The *Type of Area?* procedure reports which of six area types (based on shape and orientation) is occupied by the district token. This result is used to access the information in the *District Area* list. Columns A, B, and C represent the three counties in the area occupied by the district token.

District Area

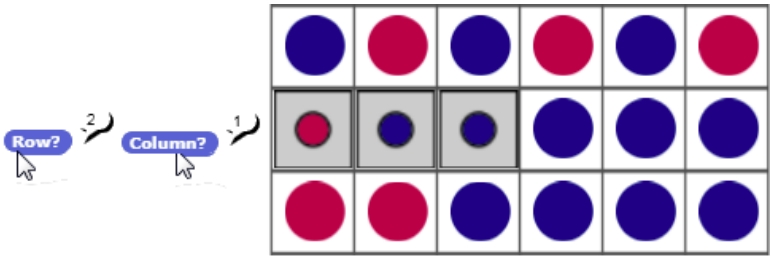
	A	B	C
1	0	1	2
2	0	-10	-20
3	0	1	10
4	0	1	-10
5	0	-1	-10
6	0	-1	10

Each row represents one of the six area types. For example, the first row represents a horizontal linear district. The numbers in this row indicate the values that will be added to determine the location of all three counties in the district. For example, if the district token is placed on Location 24, the *District Area* table can be used to determine the locations of the counties occupied by the district.

County A = Location 24 + 0
County B = Location 24 + 1
County C = Location 24 + 2

A variable, *Current District*, keeps track of the number of the district that is being placed on the game board. When the mouse pointer is placed on the game token and the mouse button is pressed, a script in the script space of the sprite assigns the appropriate value to the variable. For example, in the case of *District 1*, a value of 1 would be assigned to the variable *Current District*.

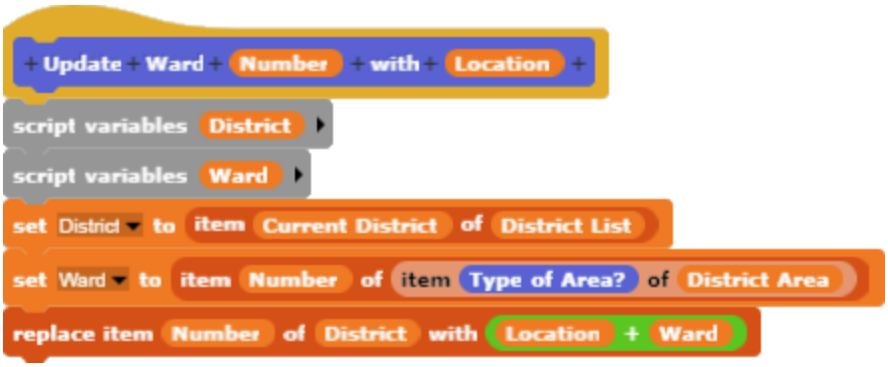
The procedures *Row?* And *Column?* report the row and column in which the district token is placed. These values are obtained by dividing the X and Y coordinates of the token by the block size.



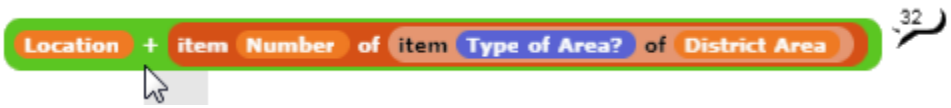
The location is formed by combining the row and column. For example, if the district token is placed at the intersection of Row 2 and Column 1, the Location obtained by combining these values would be 21.

An *Update Location* procedure records the location of the three counties in the district in the *District List*. It first obtains the current location of the token. It then updates the location of each of the three voting wards in the district. (For the purposes of the gerrymandering simulation, a voting ward is equivalent to a county.)

The *Update Ward* procedure makes use of the previously constructed *Type of Area?* procedure to identify the location of each voting ward in the district. (The term *ward* is used as a variable in this local script to avoid confusion with use of *county* as a variable in other contexts.)



The *Type of Area*? function is used to identify the square on the grid corresponding to each county covered by the district token. In this instance, the result indicates that the first square covered by the district token is at Location 32 on the grid. To improve readability, the second half of the function in the illustration below is assigned to a variable named *Ward* in the procedure above.



This value is recorded in the *District List* along with the locations of the second and third wards in the district.

Since two of the three wards are red, the value of *Red* is recorded in the last column of the *District List*. The procedure *Update District Color* checks the color of each of the three wards. If two or more wards are blue, a value of *Blue* is recorded. Otherwise a value of *Red* is recorded.

The procedure also changes the color of the district token to match this value. A procedure to update the total number of blue and total number red districts, *Update District Total*, then records the outcomes for each district on which a token has been placed.

Assigning Colors to Counties

Half of the 36 counties in the grid are assigned red tokens. The other half of the counties are assigned blue tokens. The *Assign Colors* procedure first builds a temporary list named *List of Colors* in which half of the colors are red and the other half of the colors are blue.

Once a temporary list of colors has been constructed in this way, the second half of the *Assign Colors* procedure then randomly selects a color from the list of colors and assigns it to the first county in the *County List*. The token in the corresponding block in the grid is also changed to that color.

Once a color has been selected from the *List of Colors*, it is deleted. The process is repeated until all of the colors have been assigned to each of the 36 counties in the grid.

Appendix

Procedures

- *Grid*

  Draws a grid with a specified number of rows and columns.
  Grid calls two procedures: Grid Setup and Draw Grid

  - *Grid Setup*

    Establishes the size of each block in the grid, and the location of the grid (determined by X and Y-offset variables.

  - *Draw Grid*

    Draw Grid calls *Draw Grid Lines* which uses the procedures *Line* and *Over* to draw the grid.

    - *Line* – Draws a line of specified length.
    - *Over* – Moves the turtle over a specified number of steps.

- *Populate Grid*

  Places a red or blue game in each block of the grid.
  Records the color of each county (as marked by a token) in the grid in *County List*.

- *Place District*

  Places a district token that covers three contiguous blocks on the grid.
  A district token can either be linear or L-shaped. *Place District* performs the following actions.
  It continually updates the coordinates of the district until the district is moved. When moved:\

  - *Snap Token* – snaps the district token into place in the new location.
  - *Update Location* – records the location (row and column) of the token.
  - *Update District Total* – tallies the number of red and blue districts.

Functions

- *Row?* records the row of the currently selected district token.
- *Column?* records the column of the currently selected district token.
- *In Bounds?* evaluates whether the district token has been placed on the grid.
- *Position Change?* determines whether the district token has been moved.
- *Which County?* reports the county on which the district token has been placed.
- *County Color?* reports the color of the county on which the district token has been placed.
- *Costume Shape Linear?* reports whether the district token is linear or L-shaped.
- *Type of Area?* reports the costume shape and the orientation of the district.

Lists

- *County List* records the location and color of each county.
- *District List* records the location of each of the counties within a district and the overall color.
- *District Area* records the location of the three counties within a district based on district type (based on costume shape and orientation).

Variables

Global Variables – Main Program

- *Current District* – the currently selected district
- *Blue Districts* – the number of blue districts
- *Red Districts* – the number of red districts

Global Variables - Setup

- *Grid Block Size* – the size of each block in the grid
- *Grid X-offset* – the horizontal offset of the grid (relative to 0 on the x-axis)
- *Grid Y-offset* – the vertical offset of the grid (relative to 0 on the y-axis)

Local Variables (local to the currently selected district)

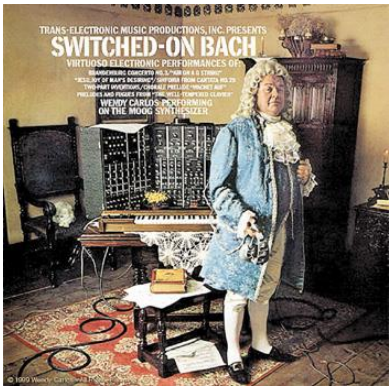
- *Row* – the row in which the current district token has been placed.
- *Column* – the column in which the current district token has been placed.
- *Location* – the location (row and column) in which the current district token has been placed.

CHAPTER 8

Electronic Music

Glen Bull and Joe Garofalo

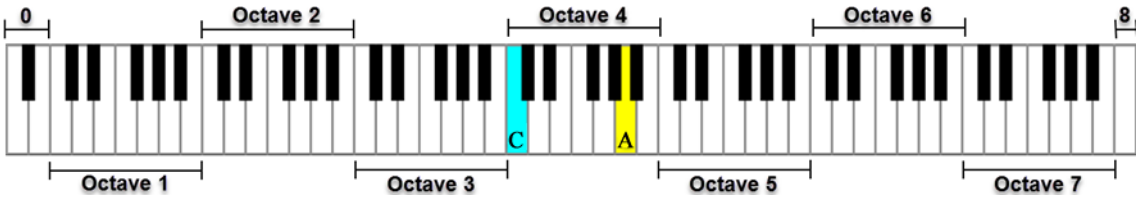
This chapter introduces electronic music. The invention of the transistor made music synthesizers practical. These synthesizers can produce electronic music with effects impossible to replicate with traditional musical instruments. Robert Moog developed the first commercially available music synthesizer, which was showcased on the album, *Switched-On Bach*. This work became only the second classical music album to sell more than a million copies.



The album, Switched-On Bach, was created with a music synthesizer.

Today's software synthesizers allow anyone to begin exploring electronic music with considerably less difficulty and expense. Using a program like *Snap!* to create music requires some understanding of both coding and music described in the sections that follow.

A standard piano keyboard has 88 keys divided into octaves.



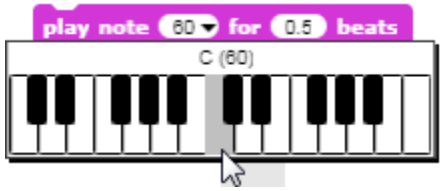
There are twelve notes in each octave – seven white keys and five black keys. Each octave begins with the note "C". The first octave begins with a note referred to as C1, the second octave begins with C2, etc. The note known as "Middle C" falls into the fourth octave (i.e., in the middle of the keyboard). By convention, this note can also be written as "C4" to indicate that it is the note "C" in the 4th octave of the piano keyboard.

Orchestras tune their instruments to the note of A above middle C. Each note has a frequency associated with it. For example, when the note A above middle C is played, the corresponding piano string vibrates at a rate of 440 times per second. In honor of the pioneering scientist Heinrich Hertz, 440 times per second can also be written as 440 Hertz (abbreviated Hz).

As digital technologies made new types of electronic instruments possible, a *Musical Instrument Digital Interface* (MIDI) was developed to enable MIDI instruments to communicate with one another and with computers. The MIDI numbering system includes 20 notes that are below the lowest note on a piano keyboard. The width of a piano keyboard was constrained by the range of notes that a pianist could easily reach, but a computer program does not have these same constraints. Table 1 provides an overview of the relationship between notes on a piano keyboard and the corresponding MIDI number for the octave beginning with Middle C.

Table 1. Piano Keys and MIDI Numbers for the Octave from C4 to C5													
Note	*C4*	*C♯*	*D*	*D♯*	*E*	*F*	*F♯*	*G*	*G♯*	*A*	*A♯*	*B*	*C5*
Piano Key	40	41	42	43	44	45	46	47	48	49	50	51	52
MIDI #	60	61	62	63	64	65	66	67	68	69	70	71	72

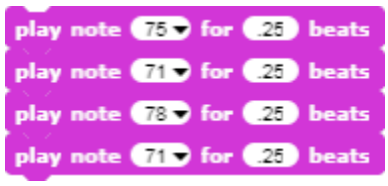
The *Snap!* command *Play Note* (found under the *Sounds* section of the command palette) uses the MIDI numbering system. A dropdown menu displays a piano keyboard that shows the note and the corresponding MIDI number as each key on the virtual keyboard is pressed.



A series of musical notes can be combined to play a musical sequence. The song *Thunderstruck* by the band AC/DC begins with a lead-in note (B) followed by this sequence of notes: "D♯, B, F♯, B". This sequence corresponds to the following MIDI notes in the octave above Middle C: "75, 71, 78, 71."

Table 2. MIDI Notes in the Opening Notes of Thunderstruck											
Notes	***B***	*C*	*C♯*	*D*	***D♯***	*E*	*F*	***F♯***	*G*	*G♯*	*A*
MIDI #	**71**	72	73	74	**75**	76	77	**78**	79	80	81

The following *Snap!* commands will play this initial sequence of notes.

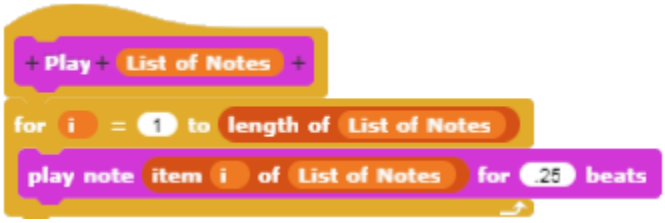


This four-note sequence appears eight times during the opening sequence of notes:

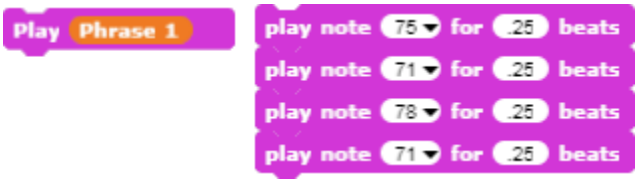
D♯, B, F♯, B - D♯, B, F♯, B - D♯, B, F♯, B - D♯, B, F♯, B
D♯, B, F♯, B - D♯, B, F♯, B - D♯, B, F♯, B - D♯, B, F♯, B

A list named *Phrase 1* stores the opening four-note sequence.

The following procedure plays a list of notes such as the notes recorded in *Phrase 1*.



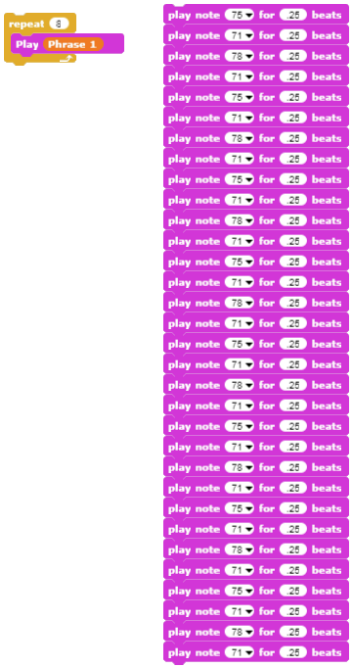
Once the notes are assigned to the list named *Phrase 1*, the command *Play Phrase 1* (on the left in the illustration below) has the same effect as the four individual *Play Note* commands (on the right in the illustration).



The opening sequence of notes is repeated eight times.

Repeat 8 [Play Phrase 1]

The command has the same effect as repeating the sequence using 32 *Play Note* commands. It is more efficient to record the notes in a list and then play them using the *Play List of Notes* command.



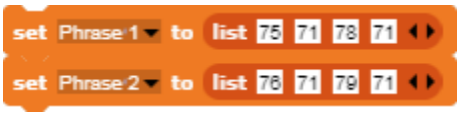
The table below lists two sets of four-note sequences that comprise the beginning of *Thunderstruck*.

Table 3. Repeated Phrases in the Opening Sequence					
	Repeated	*Note 1*	*Note 2*	*Note 3*	*Note 4*
Phrase 1	8 Times	D#	B	F#	B
Phrase 2	8 Times	E	B	G	B
Phrase 1	8 Times	D#	B	F#	B
Phrase 2	8 Times	E	B	G	B

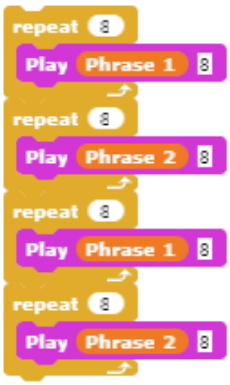
The notes used in the second phrase are highlighted in the table below.

Table 4. MIDI Notes in Phrase 2											
Notes	***B***	*C*	*C♯*	*D*	*D♯*	***E***	*F*	*F♯*	***G***	*G♯*	*A*
MIDI #	**71**	72	73	74	75	**76**	77	78	**79**	80	81

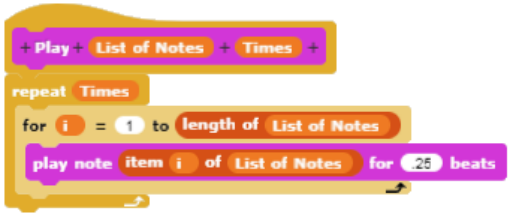
The notes in the first and second phrase can be assigned to lists named *Phrase 1* and *Phrase 2*.



Phrase 1 and Phrase 2 could then be repeated 8 times in each of four iterations.

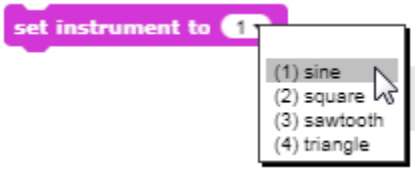


However, it is more efficient to modify the *Play List of Notes* command so that it can repeat the list of notes a specified number of times. Addition of the variable *Times* makes this possible.

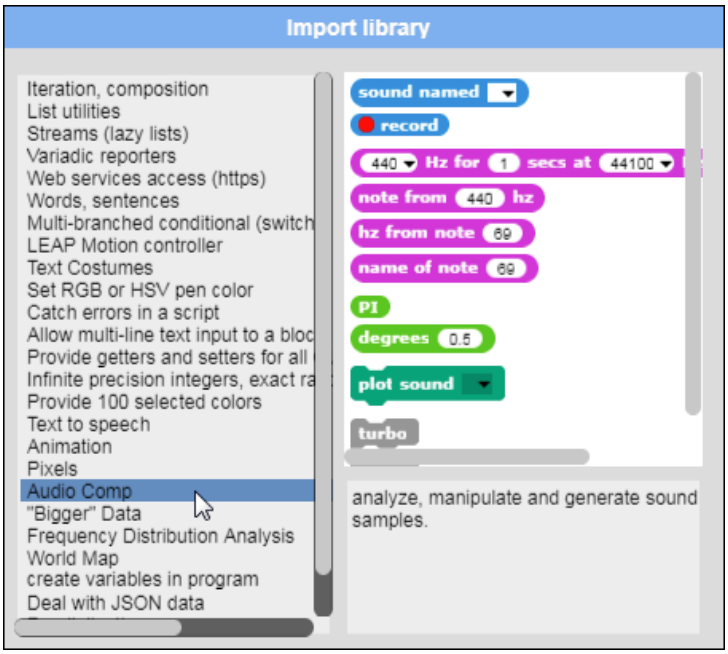


This enables the sequence of 128 notes to be played using four lines of code. The first line in the illustration below states, *Play Phrase 1 a Total of 8 Times*.

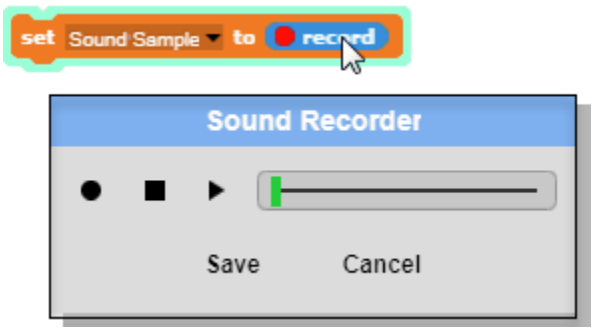
The choices in the *Sound* command palette include a *Set Instrument* command. This command allows the sound played to be set to one of several different sound waveforms, including sine wave, sawtooth wave, etc.



None of these choices sound a great deal like an actual instrument. However, additional audio commands can be imported from the *Snap!* library, including an option to record and play a sample from an actual musical instrument.

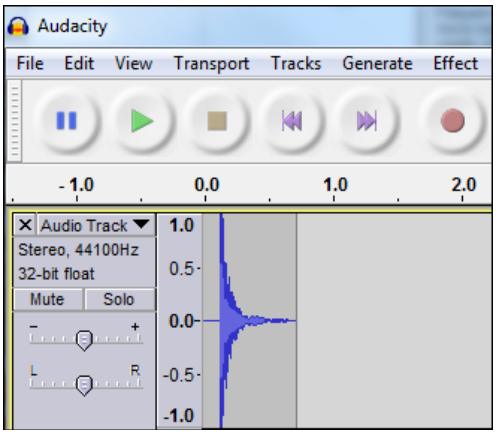


The *Record* command (found under the *Sensing* command palette after the *Audio* library of *Snap!* commands has been imported) records a sound with the microphone of the computer and assigns the recorded sound to a variable. In the example below, the recorded sound has been assigned to a variable named *Sound Sample*.

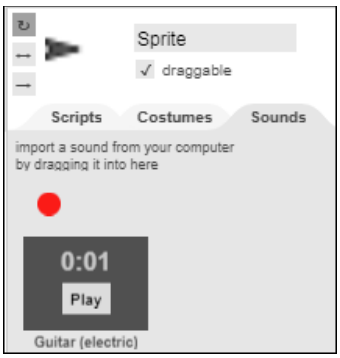


The recorded sound can then be played in *Snap!* using the *Play Sound* command.

Alternatively, sounds can be recorded in a program such as Audacity (available as a free open-source audio application that can be downloaded at https://www.audacityteam.org/).

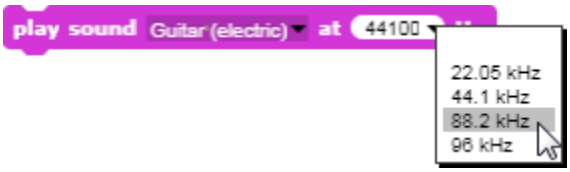


Sounds recorded with applications such as *Audacity* can then be imported into *Snap!* when the *Sounds* tab is selected.



Sounds imported into *Snap!* in this manner will then appear on the dropdown menu on the *Play Sound* command.

One approach to using sampled sounds to play a tune would be to record a sample for each note. However, the *Play Sound at _____ kHz* offers a path to a more economical approach to use of sampled sounds.



The default playback rate on the dropdown menu is 44.1 kHz. When a sound is recorded at this rate, 44,100 samples per second are digitized and recorded. When the sound is played back at the same rate, the pitch of the played back sound is the same as the pitch at which it was recorded. However, when a sound is recorded at 44,100 samples per second, and played back at a rate 88,200 samples per second, the pitch of the played back sound doubles.

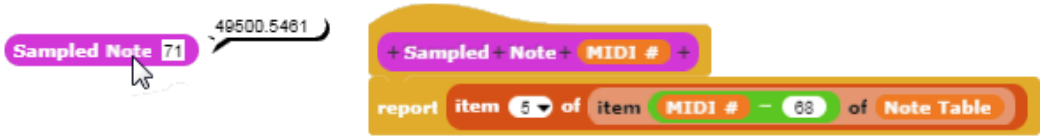
For example, the note A in the fourth octave (that is A above middle C) is 440 Hz. This means that the violin string vibrates at a rate of 440 times per second. The note A in the fifth octave is 880 Hz, or double the frequency of the same note one octave below. If the note A is recorded at a sampling rate of 44,100 Hz and played back at a rate of 88,200 Hz, the frequency will double. By adjusting the playback rate for a sampled sound, the frequencies of all of the notes in the octave can be obtained with a single sample.

The note A4 played on an electric guitar was recorded and imported into *Snap!* and saved under the name "Guitar (electric)". The playback ranges for the notes for the octave spanning A4 to A5 were then imported into a list assigned to the variable *Note Table*.

Note Table

13	A	B	C	D	E
1	69	A	440	1	44100
2	70	A#	466.164	1.05946	46722.3464
3	71	B	493.883	1.12246	49500.5461
4	72	C	523.251	1.18921	52444.0207
5	73	C#	554.365	1.25992	55562.492
6	74	D	587.33	1.33484	58866.4841
7	75	D#	622.254	1.41421	62366.8214
8	76	E	659.255	1.49831	66075.3307
9	77	F	698.456	1.5874	70004.34
10	78	F#	739.989	1.68179	74167.0793
11	79	G	783.991	1.7818	78577.2798
12	80	G#	830.609	1.88775	83249.6748
13	81	A	880	2	88200

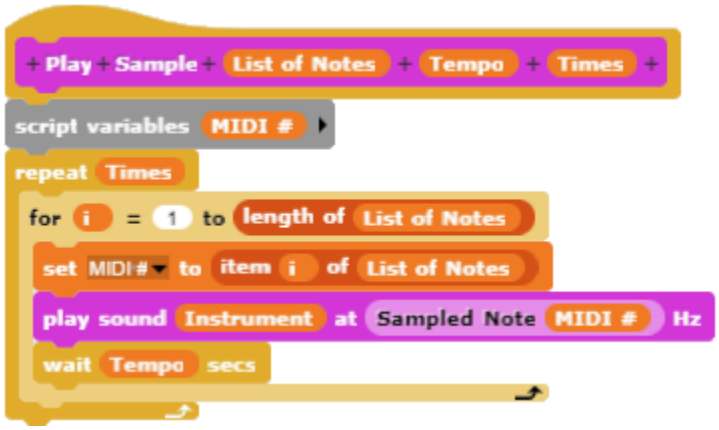
Sampled Note makes use of the *Note Table* to report the rate that plays back the sampled sound at frequencies corresponding to the notes in the octave from A4 to A5.



The recorded electric guitar sample was assigned to a variable named *Instrument*.

The tools constructed in this manner were then used to create a *Play Sample* procedure. The *Play Sample* procedure plays the list of notes in *Phrase 1* and *Phrase 2* just as the *Play List of Notes* command does. However, *Play Sample* plays the notes using the sampled guitar sound. It has three inputs:

1. List of Notes (e.g., Phrase 1, Phrase 2, etc.)
2. Tempo (the tempo at which the notes are played)
3. Times (the number of times that the list is repeated)



For example, *Play Sample Phrase 1 [0.1] [8]*, plays the list of notes in *Phrase 1* with a delay of one-tenth of a second (0.1 second) between each note, and repeats the played list eight times.

Within the structure that has been developed, a song can be envisioned as a list of phrases.

1. A phrase is a list of notes
2. A song is a list of phrases

The first eight phrases at the beginning of Thunderstruck are listed in the table below.

Table 5. Initial Phrases at the Beginning of Thunderstruck

	Repeated	*Note 1*	*Note 2*	*Note 3*	*Note 4*
Phrase 1	8 Times	D#	B	F#	B
Phrase 2	8 Times	E	B	G	B
Phrase 1	8 Times	D#	B	F#	B
Phrase 2	8 Times	E	B	G	B
Phrase 3	1 Time	B5	B	A	B
Phrase 4	1 Time	G#	B	A	B
Phrase 5	1 Time	G#	B	F#	B
Phrase 6	1 Time	G#	B	E	B
Phrase 7	1 Times	F#	B	D#	B
Phrase 8	3 Times	E	B	D#	B

These numbers can be translated into the following MIDI numbers.

Table 6. MIDI Numbers for Thunderstruck Opening Phrases

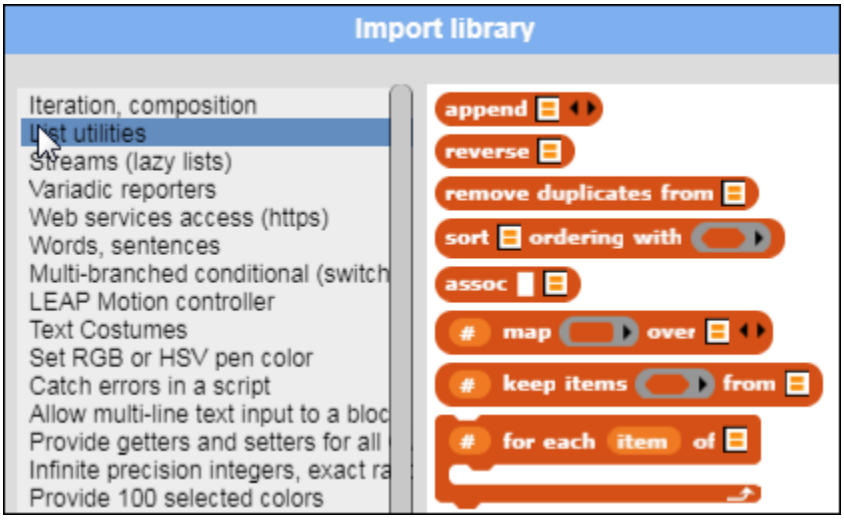
	Repeated	*Note 1*	*Note 2*	*Note 3*	*Note 4*
Phrase 1	8 Times	75	71	78	71
Phrase 2	8 Times	76	71	79	71
Phrase 1	8 Times	75	71	78	71
Phrase 2	8 Times	76	71	79	71
Phrase 3	1 Time	83	71	81	71
Phrase 4	1 Time	80	71	81	71
Phrase 5	1 Time	80	71	78	71
Phrase 6	1 Time	80	71	76	71
Phrase 7	1 Times	78	71	75	71
Phrase 8	3 Times	76	71	75	71

Once each of the phrases has been assigned to a corresponding variable name, then the list of phrases in the opening sequence can be assigned to a list named *Thunderstruck*.

The procedure *Play Song* accepts the song list (i.e., list of phrases in the song) assigned to the variable *Thunderstruck* and plays each of the phrases until the song is complete.

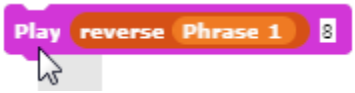
The completed *Play Song* procedure plays the phrases assigned to the list named *Thunderstruck*.

Once the song structure is in place, there are a number of ways in which the song can be manipulated. For example, the Snap library includes a procedure to reverse a list among the additional list utilities blocks.

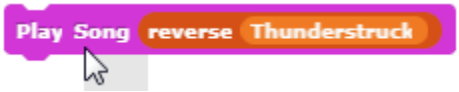


For example, *Phrase 1* consists of the MIDI notes [75 71 78 71]. The *Reverse* function generates a list in the reverse order.

When this function is combined with the command *Play*, the notes in *Phrase 1* will be played in the reverse order.



Similarly, *Play Song Reverse Thunderstruck* will play the phrase in the list Thunderstruck in the reverse order.



Another function can shift a list of notes to a different octave. The Map utility is found in the Variables section of the *Commands* palette. It can apply an operation to every item in a list. In the illustration below, the map function as added 12 to each number in the list (i.e., increased each number by one octave).

The command *Play [Map +12 over Phrase 1] [8]* adds 12 to each MIDI note before playing it, shifting the phrase up one octave.