

# **Unit 3 Lesson 1**

## **The Need for Programming Languages**

### **Resources**

Name(s) \_\_\_\_\_ Period \_\_\_\_\_ Date \_\_\_\_\_

# Activity Guide - LEGO® Instructions



## Challenge

Create instructions your classmates could use to reproduce a simple arrangement of LEGO® blocks. Do the following:

### 1. Create Your LEGO Arrangement

You will be given 5-6 LEGO blocks which you should connect into a single arrangement. Try to choose something interesting or challenging to test your instruction-giving abilities.

### 2. Record Your Arrangement

Record your arrangement somehow so you can recall it later. Make a simple drawing, take a photo, etc. You'll want an exact record, so make sure you pay attention to color!

### 3. Write Instructions

In the space provided below, write a clear and precise set of instructions your classmates could follow to build this arrangement on their own, without diagrams or pictures. That is, your instructions may only use words, so you cannot use pictures to help you.

## Test Your Algorithm

Exchange algorithms and blocks with another group and try to follow the instructions provided to create the correct arrangement. Afterwards, confirm whether you succeeded by using the other group's image.

## Reflection

Once you've had an opportunity to test one another's instructions, respond to the following reflection questions.

- Were you always able to create the intended arrangement? Were your instructions as clear as you thought?
- Why do you think we are running into these miscommunications? Is it really the fault of your classmates or is something else going on?

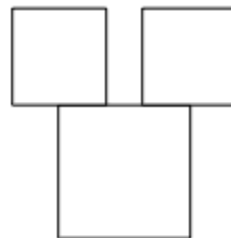
Name(s) \_\_\_\_\_ Period \_\_\_\_\_ Date \_\_\_\_\_

## Activity Guide - Building Blocks of Drawing

**Challenge:** Create instructions your classmates could use to reproduce a simple arrangement of drawn blocks. Similar to the one shown at right.

**Create Your Arrangement:** With a pen, draw a figure of variously sized rectangles (or squares) similar to the one shown at right.

**Design Your Algorithm:** As you draw think about about the fundamental operations - the most basic set of commands - you would need in order to write out a list of instructions for another person to draw it the same way you did. Your instructions may only use words, so you cannot use pictures to help you.



## Compare Your Instructions with a Neighbor's Instructions:

With a neighbor, compare how you each drew the shape, and compare your sets of commands to see where you have things in common.

- Where did the drawer start?
- Where did they end?
- Did they draw it the same way as you?
- Did they pick up the pen?
- How many different commands do you actually need to draw this?

**Reflection:** Once you've had an opportunity to compare instructions with a neighbor, respond to the following reflection questions:

- How were your instructions different from your neighbor's?
- Were your instructions as clear as you thought?

# **Unit 3 Lesson 2**

## **The Need for Algorithms**

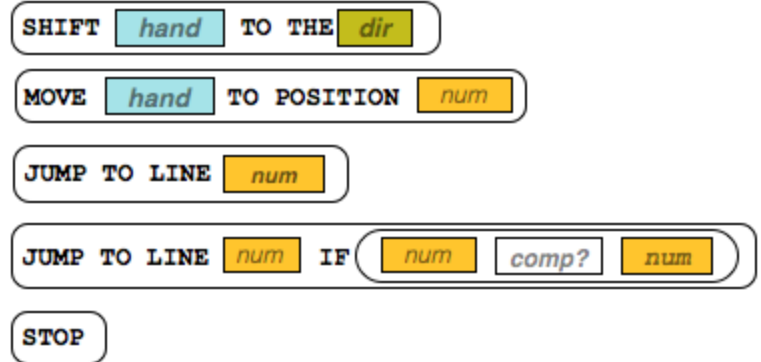
### **Resources**

# The “Human Machine” Language

Here are the beginnings of a more formalized low-level language you can use to create programs for a “Human Machine” to solve problems with playing cards.

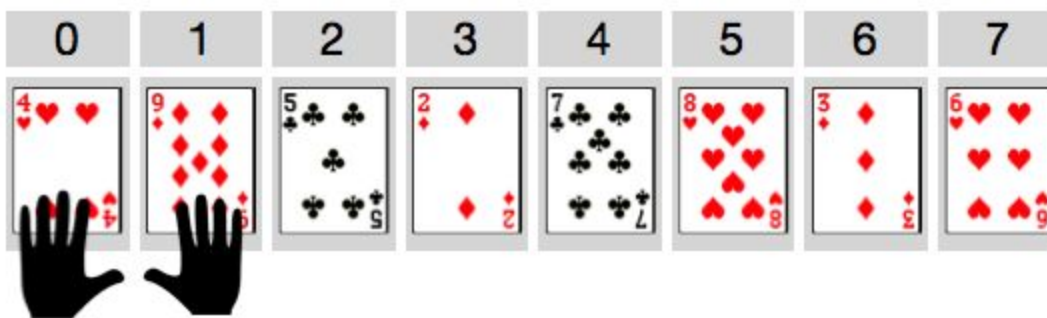
To simplify things we’ll get rid of the need to pick cards up and put them down. Instead leave cards face up and just touch them. The 5 commands you can use are shown to the right. **See the Reference Guide on the next page for descriptions of what these commands do.**

Some of these commands might seem unusual, but we can write programs with just these commands to control the “human machine’s” hands to touch or pick up the cards, look at their values, and move left or right down the row of cards.



## Standard Card Setup

You should assume this standard initial setup. Here is a diagram for an 8-card setup:



- There will be some number of cards with random values, lined up in a row, face up.
- Positions are numbered starting at 0 and increasing for however many cards there are.
- The left and right hands start at positions 0 and 1 respectively.

## Try out some example programs

Get to know the Human Machine Language by acting out the examples on the following page with a partner. For each of the examples on the next page you should:

- Lay out a row of **8 cards** in front of you to test out the program.
- Have one partner read the instructions in sequence starting at line 1, and the other partner act out each command as the human machine.
- Use the **code reference** to answer your questions and verify you’re interpreting the code correctly.
- Give a brief description of what the program does, or its ending state.

### NOTES:

- Some of the programs are very simple
- Some of the programs might not ever stop
- The point is simply to practice using the language and executing commands as a “Human Machine”

Example Program	What does it do?
<div><div>1</div><div>SHIFT RH TO THE R</div></div> <div><div>2</div><div>SHIFT RH TO THE R</div></div> <div><div>3</div><div>SHIFT RH TO THE R</div></div> <div><div>4</div><div>SHIFT RH TO THE R</div></div> <div><div>5</div><div>SHIFT RH TO THE R</div></div> <div><div>6</div><div>SHIFT RH TO THE R</div></div> <div><div>7</div><div>STOP</div></div>	
<div><div>1</div><div>SHIFT RH TO THE R</div></div> <div><div>2</div><div>JUMP TO LINE 1</div></div> <div><div>3</div><div>STOP</div></div>	<p><i>Note: this one has a problem, can you find it?</i></p>
<div><div>1</div><div>SHIFT RH TO THE R</div></div> <div><div>2</div><div>JUMP TO LINE 1 IF RHPos ne 7</div></div> <div><div>3</div><div>STOP</div></div>	
<div><div>1</div><div>MOVE RH TO POSITION 7</div></div> <div><div>2</div><div>SHIFT LH TO THE R</div></div> <div><div>3</div><div>SHIFT RH TO THE L</div></div> <div><div>4</div><div>JUMP TO LINE 2 IF RHPos gt LHPos</div></div> <div><div>5</div><div>STOP</div></div>	
<div><div>1</div><div>JUMP TO LINE 5 IF LHCard eq 9</div></div> <div><div>2</div><div>SHIFT LH TO THE R</div></div> <div><div>3</div><div>MOVE RH TO POSITION LHPos</div></div> <div><div>4</div><div>JUMP TO LINE 1</div></div> <div><div>5</div><div>STOP</div></div>	<p><i>Note: there is a potential problem with this one too. But only in certain circumstances. Can you find it?</i></p>



# Human Machine Code Reference Guide

## Hands, Values and Direction

There are some short-hand abbreviations for referring to the human machine, the cards, positions, and directions of movement.

**Hands** - The Human Machine has hands! You can refer to a specific hand abbreviated **LH** or **RH** (left hand or right hand).

**Values** - Each hand has two values you can refer to:

1. **LHPos**, **RHPos** - The hand's position in the list (a number)
2. **LHCard**, **RHCard** - The value of the card the hand is holding (a number)

**Direction** - There are two directions **R** and **L** (right and left) that hands can move along the row of cards.

### Hands

<b>RH</b>	Right Hand
<b>LH</b>	Left Hand

### Values

<b>LHPos</b>	<b>RHPos</b>	Position in the list
<b>LHCard</b>	<b>RHCard</b>	Value on the card

### Directions

<b>R</b>	Right
<b>L</b>	Left

## Commands

Description	Examples
<div>SHIFT <b>hand</b> TO THE <b>dir</b></div> <p>Shift the given hand one position to the right or left along the row of cards.</p>	<div>SHIFT <b>LH</b> TO THE <b>R</b></div>
<div>MOVE <b>hand</b> TO POSITION <b>num</b></div> <p>Move a given hand to a specific position number in the row of cards.</p>	<div>MOVE <b>RH</b> TO POSITION <b>4</b></div> <div>MOVE <b>LH</b> TO POSITION <b>RHPos</b></div>
<div>JUMP TO LINE <b>num</b></div> <p>Jump to a specific line number in the program and continue execution from that point.</p>	<div>JUMP TO LINE <b>1</b></div>
<div>JUMP TO LINE <b>num</b> IF <b>num</b> <b>comp?</b> <b>num</b></div> <p>Jump to line but ONLY IF the comparison of two numbers is <i>true</i>. If the comparison is <i>false</i> then just proceed onto the next line of code.</p> <ul style="list-style-type: none"><li>• For numbers, you can use <b>integers</b> or any of the hand values <b>RHCard</b>, <b>LHCard</b>, <b>RHPos</b>, <b>LHPos</b></li><li>• For comparisons you can use <b>eq</b>, <b>ne</b>, <b>lt</b>, <b>gt</b>, (equal, not equal, less than, greater than)</li></ul>	<div>JUMP TO LINE <b>4</b> IF <b>LHCard</b> <b>eq</b> <b>7</b></div> <div>JUMP TO LINE <b>2</b> IF <b>LHCard</b> <b>lt</b> <b>RHCard</b></div> <div>JUMP TO LINE <b>7</b> IF <b>RHPos</b> <b>gt</b> <b>9</b></div>
<div>STOP</div> <p>End of program. Stop doing anything, stop executing lines of code.</p>	<p><i>This should be the last line of code in the program, or on a line that is jumped to when you want the program to stop.</i></p>

# Challenge: Find Min

Using only the Human Machine Language design a program to find the card with the smallest value in the list of cards.

**Goal:** When the program stops, the left hand should be touching the card with the smallest value.

**Hint:** How do you know you're at one end of the list or the other?

Use the hand position values to check whether the position is 0 or the largest position in the list - you can assume that you know how big the list is ahead of time.

For example, if the last position is 7, then the comparison: **IF RHPos eq 7** would tell you that the right hand was as the end of the list.



Write your program in the space provided below

1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

# Command Cut Outs

Print out this sheet and cut out each command to use as lines of code in the template provided on the previous page. Alternatively, you can just write the commands by hand into the template.

SHIFT  TO THE

SHIFT  TO THE

JUMP TO LINE  STOP

JUMP TO LINE  STOP

MOVE  TO POSITION

MOVE  TO POSITION

SHIFT  TO THE

SHIFT  TO THE

JUMP TO LINE  STOP

JUMP TO LINE  STOP

MOVE  TO POSITION

MOVE  TO POSITION

JUMP TO LINE  IF

JUMP TO LINE  IF

JUMP TO LINE  IF

JUMP TO LINE  IF

Name(s) \_\_\_\_\_ Period \_\_\_\_\_ Date \_\_\_\_\_

## Activity - Minimum Card Algorithm



### “Human Machine” Algorithms - Find Min

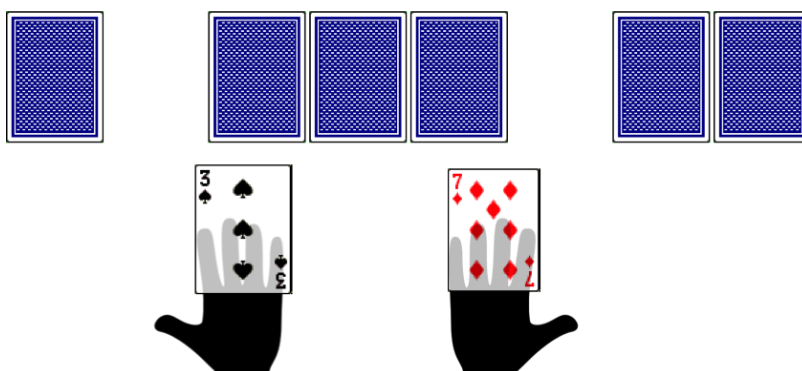
We often get started thinking about algorithms by trying to rigorously act them out ourselves as a sort of “Human Machine”. When acting as a machine, we can keep the limitations of a computer in mind.

In this activity, you'll design an algorithm to find the smallest item in a list. Obviously, if we were really writing instructions for a person, we could simply tell them: “find the smallest item in a list.” But that won't work for a computer.

We need to describe the *process* that a person must go through when they are finding the smallest item. What are they *really* doing?

#### Setup and Rules:

- We'll use playing cards face down on the table to represent a list of items. Start with 8 random cards face down in a row.
- Any card on the table **must** be face down.
- When acting as the machine, you can pick up a card with either hand, but *each hand can only hold one card at a time*.
- You can look at and compare the values of any cards you are holding to determine which one is greater than the other.
- You can put a card back down on the table (face down), but once a card is face down on the table, you cannot remember (or memorize) its value or position in the list.



#### Task:

Write an algorithm to find the card with the lowest value in the row of cards.

- Goal: The algorithm must have a clear end to it. The last instruction should be to say: “I found it!” and hold up the card with the lowest value.
- The algorithm should be written so that it would theoretically work for any number of cards (1 or 1 million).
- Write your algorithm out on paper as a clear list of instructions in “pseudocode.” Your instructions can refer to the values on cards, and a person's hands, etc., but you must invent a systematic way for finding the smallest card.

### **My Algorithm To Find Minimum Card**

Write your algorithm below. We suggest writing it out as a numbered list of instructions to make the sequence clear.

**1.**

# **Unit 3 Lesson 3**

## **Creativity in Algorithms**

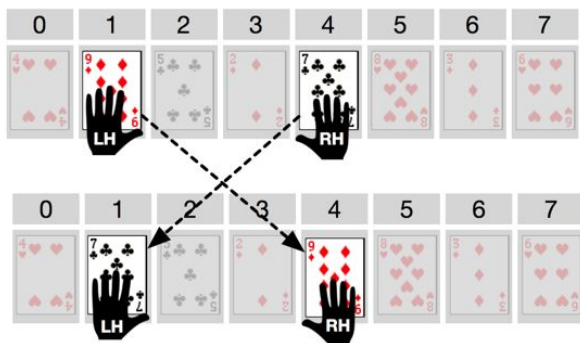
### **Resources**

# Human Machine Language - Part 2

We're going to add one command to the Human Machine Language called **SWAP** - see description below. All of the other commands are still available to you. So, there are 6 commands total in the language now.

## SWAP

Swap the positions of the cards currently being touched by the left and right hands. After a swap the cards have changed positions but hands return to original position.



The human machine action is: pick up the cards, exchange the cards in hand, and return hands to original position in the list with the other card.

## Human Machine Language Reference

SHIFT **hand** TO THE **dir**

MOVE **hand** TO POSITION **num**

JUMP TO LINE **num**

JUMP TO LINE **num** IF **num** **comp?** **num**

SWAP

STOP

## Try an example with Swap

Trace the program below with a partner and describe what it does.

1	MOVE <b>RH</b> TO POSITION <b>7</b>
2	SWAP
3	SHIFT <b>LH</b> TO THE <b>R</b>
4	SHIFT <b>RH</b> TO THE <b>L</b>
5	JUMP TO LINE <b>2</b> IF <b>RHPos</b> <b>gt</b> <b>LHPos</b>
6	STOP

What does this program do?

# Challenge: Min To Front

Using only the Human Machine Language design an algorithm to find the smallest card and move it to the front of the list (position 0). All of the other cards *must remain in their original relative ordering*.

**END STATE:** When the program stops, the smallest card should be in position 0. The ending positions of the hands do not matter, the ending positions of the other cards do not matter. *As a challenge:* try to move the min-to-front and have all other cards be in their original relative ordering.

Cards BEFORE:

0	1	2	3	4	5	6	7
9	4	5	2	7	8	3	6

Cards AFTER (may not be in this order)

0	1	2	3	4	5	6	7
2	9	4	5	7	8	3	6

1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

(If you need more lines, just keep going).



## Optional Challenges

This list of challenges is given in no particular order. Find one that intrigues you and try it out.

For all of these challenges make the following assumptions:

- Cards start randomly valued, and randomly ordered, and are dealt from an infinitely large deck. I.e. you could face a row of all one value, or there could be seven 2s and one 6, and so on.
- Algorithms should work in principle for any number of cards, and any values that are comparable.
- Algorithms must STOP and be in the END STATE given in the challenge description.

Challenge Description	Example
<p><b>Search for 2 <u>or</u> a 10</b></p> <p>Search the list and stop when find EITHER a 2 OR a 10 (you could substitute 2 and 10 for any other two values if you like).</p> <p>END STATE: the left hand should be touching the first 2 or 10 encountered in the list. End state does not matter if there is no 2 or 10, but the program should stop.</p>	<p>BEFORE: 4 3 7 5 10 4 7 2 2</p> <p>AFTER: 4 3 7 5 10 4 7 2 2</p> <p style="text-align: center;">^</p> <p style="text-align: center;">LH</p>
<p><b>Hi-Lo</b></p> <p>Find the min and max values in the list and move them to the first and last positions, respectively.</p> <p>END STATE: The card with lowest value in the list is in position 0, and the card with the highest value is in the last position (position 7 if there are 8 cards). The end state of the hands does not matter, the positions of the other cards does not matter.</p>	<p>BEFORE: 4 3 7 5 10 4 7 1 2</p> <p>AFTER: 1 4 3 7 5 4 7 2 10</p>
<p><b>Search for 2 <u>and</u> a 10</b></p> <p>Search the list and stop once you have found BOTH a 2 AND a 10.</p> <p>END STATE: the left hand should be touching a 2 and the right hand should be touching a 10. End state does not matter if there is not both 2 and a 10.</p>	<p>BEFORE: 4 3 7 5 10 4 7 2 2</p> <p>AFTER: 4 3 7 5 10 4 7 2 2</p> <p style="text-align: center;">^ ^</p> <p style="text-align: center;">RH LH</p>
<p><b>Sort</b></p> <p>Get the cards into sorted order from least to greatest.</p> <p>END STATE: end state of the hands does not matter, but cards should be in ascending order, and the program should stop.</p>	<p>BEFORE: 4 3 7 5 10 4 7 2 2</p> <p>AFTER: 2 2 3 4 4 5 7 7 10</p>
<p><b>Partition</b></p> <p>Call the last card in the list the <i>pivot value</i>. Arrange the list so that the all of the cards less than the pivot value are to the left of it and all the cards greater than the pivot are to the right. (Cards equal to the pivot can go to the left or right of it, your choice).</p> <p>END STATE: The pivot value is in the middle of the list somewhere with all values less than it to the left, and all values greater than it to the right. The ordering of the cards to the left and right do matter. The end state of the hands does not matter.</p>	<p>BEFORE: 8 7 4 2 3 7 5 1 <u>6</u></p> <p>AFTER: 4 2 3 5 1 <u>6</u> 8 7 7</p> <p>BEFORE: 5 4 3 5 4 3 5 4 <u>1</u></p> <p>AFTER: <u>1</u> 5 4 3 5 4 3 5 4</p> <p>BEFORE: 5 4 3 5 4 3 5 4 <u>7</u></p> <p>AFTER: 5 4 3 5 4 3 5 4 <u>7</u></p>
<p><b>Count</b></p> <p>Count the number of 2s in the list and set the right hand so that its position number is equal to the number of 2s in the list.</p> <p>END STATE: the position number of the right hand is equal to the number of 2s in the list. If the count is higher than the possible position numbers (i.e.</p>	<p>BEFORE: 4 <b>2</b> 7 5 10 4 7 <b>2</b> <b>2</b></p> <p>AFTER: 4 2 7 5 10 4 7 2 2</p> <p style="text-align: center;">RH</p> <p>BEFORE: 4 3 7 5 10 4 7 1 1</p> <p>AFTER: 4 3 7 5 10 4 7 1 1</p>

every value is a 2) then set the right hand to the position past the end of the list. I.e. if there are 8 positions (0-7) the right hand should end in position 8.

RH

BEFORE: 2 2 2 2 2 2 2 2 2  
AFTER: 2 2 2 2 2 2 2 2 2

RH

# Command Cut Outs

Print out this sheet and cut out each command to use as lines of code in the template provided on the previous page.

SHIFT  TO THE

SHIFT  TO THE

SHIFT  TO THE

SWAP SWAP SWAP STOP

JUMP TO LINE

JUMP TO LINE

JUMP TO LINE

JUMP TO LINE

MOVE  TO POSITION

MOVE  TO POSITION

MOVE  TO POSITION

MOVE  TO POSITION

JUMP TO LINE  IF

JUMP TO LINE  IF

JUMP TO LINE  IF

JUMP TO LINE  IF

# **Unit 3 Lesson 4**

## **Using Simple Commands**

### **Resources**

# **Unit 3 Lesson 5**

## **Creating Functions**

### **Resources**

# **Unit 3 Lesson 6**

## **Functions and Top-Down Design**

### **Resources**

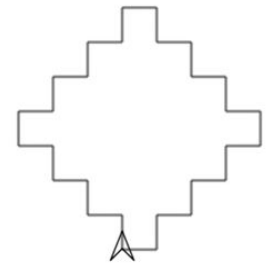
Name(s) \_\_\_\_\_ Period \_\_\_\_\_ Date \_\_\_\_\_

# Worksheet - Top-Down Design



## Functions and Managing Complexity

In the previous lesson, we saw how we could **use functions to manage the complexity of a programming task**. We identified repeated patterns in the diamond that could be programmed as individual functions and then recombined these building blocks to make the full figure. In that example, the problem was broken down for you into layers of functions (and abstraction), but going forward you'll be designing these layers of functions yourself. The question is, "How?"



## Top-Down Problem Solving

One method for approaching programming problems is called "Top-Down Design" or "Stepwise Refinement". This strategy is an informal way of repeatedly dividing a system into simpler subsystems. As these pieces of the original problem get smaller and smaller, eventually you will arrive at pieces that you can program solutions to in a straightforward manner. These pieces of code can then be recombined to solve your original, complex problem. As a result you can eventually solve a large problem, but along the way you only ever had to address smaller ones. Another benefit of this strategy is that it can help you choose names for the different functions you'll design. Here's an example of how we used this strategy to develop our solution to the diamond problem.

Top-Down Design Strategy	Talking through the problem...	Function Name
Look at the big picture...	"Well, I need to draw a diamond..."	drawDiamond()
Identify a sub-task...	"...the diamond has 4 sides I need to draw..."	drawSide()
Break down that sub-task into smaller sub-task(s)...	"...and each side is really 3 zig-zag steps..."	drawStep()
Keep going until you're down to the commands you already have access to.	"...each step is just forward-left-forward-right..."	Stop; this is simple enough to begin programming.

## Writing the Code

Once you've identified the layers of functions you want to design, you begin by writing and testing the bottom layer first and work your way up. Higher layers of functions depend on lower layers working perfectly, since they use them without needing to know the exact details of how they are programmed. The benefit of having these **layers of abstraction** is that it lets you approach increasingly complex problems in an organized way. This typically **makes reasoning about your code much easier**, and your code will read almost like *you are describing your solution to the problem in plain English*.

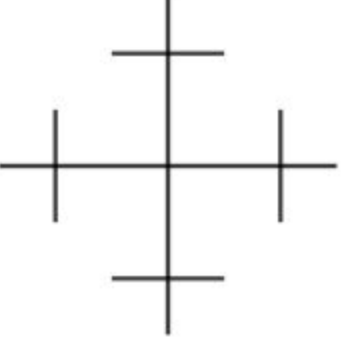
## Personal Expression in Programming

The solution you were presented to the diamond problem is only one of many possible solutions. In fact, when you originally saw that figure, you may have seen a postage stamp or two pyramids stacked on one another. There are always innumerable ways to break the problem into subproblems, name the associated functions, and program their behavior. **How you decide to write a program is a reflection of the way you think and approach problems** and no two people will always do it exactly identically. Programming is a form of personal expression as unique to you as any other way you might express yourself.

## Practice with Top-Down Design

Let's practice using Top-Down Design by breaking down a simple problem.

- With a partner look at the figure that you need to draw.
- Talk through the problem starting from the big picture and identifying sub-tasks.
- Decide the names of the functions you would write to solve this problem.
- Use the space provided to do scratch work.
- Iterate on your ideas; as you discuss the problem you might change your mind about the approach, or come up with better, more descriptive, function names.

The problem: Write a program for a turtle to draw this figure	Functions you would write
	

## Compare with Another Group

Trade your solution with another group. Did you break down the problem in the same way? Do your functions have the same names?



# **Unit 3 Lesson 7**

## **APIs and Using Functions with Parameters**

### **Resources**

# **Unit 3 Lesson 8**

## **Creating Functions with Parameters**

### **Resources**

# **Unit 3 Lesson 9**

## **Looping and Random Numbers**

### **Resources**

# **Unit 3 Lesson 10**

## **Practice PT - Design a Digital Scene**

### **Resources**

# Design a Digital Scene Project and Programming Rubric



*This rubric does not mimic the Create Performance Task. It covers key practices and programming concepts taught in Unit 3.*

Concept	Limited / No Evidence (0)	Inconsistent Evidence (1)	Strong Evidence (2)	Score
<b>Comments</b>	The program includes no / extremely limited commenting.	The program includes comments but not in all sections. Comments may not effectively clarify the purpose or functionality of the program.	Comments are used consistently. Comments help clarify the purpose or functionality of the program.	
<b>Function and Parameter Names</b>	Function and parameter names do not clearly indicate their purpose. Consistent naming conventions are not used.	Function and parameter names sometimes indicate their purpose. Consistent naming conventions may be used.	Function and parameter names indicate their purpose. Consistent naming conventions are used.	
<b>Functions and Abstraction (Top Down Design)</b>	The program makes limited use of functions. The program does not make use of layers of functions.	The program uses functions but the program may not feature high level and low level functions. There may be missed opportunities to simplify program expression through the use of functions.	Top Down Design clearly used to divide the program into layers of functions. Lower level functions have been further divided into layers when necessary.	
<b>Functions with Parameters</b>	The program does not feature a function with a parameter or the parameters are not used in the function.	A function with a parameter is present, but the parameter is not used in a meaningful way - OR - the function is not called with different values supplied to the parameter (called with the same values every time)	A function with a parameter is present. The parameter controls a meaningful component of the function's behavior. The function is called with different values given to the parameter.	
<b>Loops</b>	The program does not use loops.	The program uses loops inconsistently. There are sections of repeated code that should be placed within a loop.	Loops are used consistently when there is a need to repeatedly run the same block of code.	
<b>Collaboration</b>	Group planning document may be incomplete. In-class communication was limited. Final project may not include code from each member of the team. Comments may not be used to indicate who wrote different sections of the final program.	There is some evidence of effective collaboration. For example the group planning document is complete but in-class communication was weak, leading to program components that do not mesh well.	Group planning guide, classroom participation, and final program code reflect consistent effective collaboration. All team members are assigned significant portions of program. Team members communicated effectively during in-class programming time. Final program includes comments reflecting who completed which sections of the program.	

Name(s) \_\_\_\_\_ Period \_\_\_\_\_ Date \_\_\_\_\_

# Design a Digital Scene - AP Style Rubric



## Tips and Comments:

- This is a modified version of the [2018 Create Performance Task Scoring Guidelines](#), which has 8 rows.
- For each row you either are awarded the point or not. There is no partial credit.
- Row 1 was modified to remove references to the video.
- Rows 4, 5, and 6 are excluded because they deal with algorithms.
- All other rows are identical to the scoring guidelines with slightly different formatting.

Row and Task	Decision Rules	Score and Notes
<b>Row 1</b> <b>Response 2A</b>  The response identifies the purpose of the program	Response earns the point if it explains the function of the program instead of identifying the purpose.  Response earns the point if the illustrated feature runs, even if it does not function as intended.	
<b>Row 2</b> <b>Response 2B</b>  Describes or outlines steps used in the incremental and iterative development process to create the entire program.	<b>Do NOT award a point if any one of the following is true:</b> <ul style="list-style-type: none"> <li>• the response does not indicate iterative development;</li> <li>• refinement and revision are not connected to feedback, testing, or reflection; or</li> <li>• the response only describes the development at two specific points in time.</li> </ul>	
<b>Row 3</b> <b>Response 2B</b>  Specifically identifies at least two program development difficulties or opportunities.  AND  Describes how the two identified difficulties or opportunities are resolved or incorporated.	Response earns the point if it identifies two opportunities, or two difficulties, or one opportunity and one difficulty AND describes how each is resolved or incorporated.  <b>Do NOT award a point if any one of the following is true:</b> <ul style="list-style-type: none"> <li>• only one distinct difficulty or opportunity in the process is identified and described; or</li> <li>• the response does not describe how the difficulties or</li> </ul>	

	opportunities were resolved or incorporated.	
Row and Task	Decision Rules	Score and Notes
Row 4	<i>Omitted for this Practice Performance Task</i>	
Row 5	<i>Omitted for this Practice Performance Task</i>	
Row 6	<i>Omitted for this Practice Performance Task</i>	
<b>Row 7</b> <b>Response 2D</b>  Selected code segment is a student-developed abstraction.	Responses that use existing abstractions to create a new abstraction, such as creating a list to represent a collection (e.g., a classroom, an inventory), would earn this point.  <b>Do NOT award a point if any one of the following is true:</b> <ul style="list-style-type: none"> <li>• the response is an existing abstraction such as variables, existing control structures, event handlers, APIs;</li> <li>• the code segment consisting of the abstraction is not included in the written responses section or is not explicitly identified in the program code section; or</li> <li>• the abstraction is not explicitly identified (i.e., the entire program is selected as an abstraction, without explicitly identifying the code segment containing the abstraction).</li> </ul>	
<b>Row 8</b> <b>Response 2D</b>  Explains how the selected abstraction manages the complexity of the program.	Responses should not be penalized for explanations of abstractions that are not developed by the student.  <b>Do NOT award a point if any one of the following is true:</b> <ul style="list-style-type: none"> <li>• the explanation does not apply to the selected abstraction; or</li> <li>• the abstraction is not explicitly identified (i.e., the entire program is selected as an abstraction, without explicitly identifying the code segment containing the abstraction).</li> </ul>	

# Practice PT - Design a Digital Scene



## Background

In this Practice Performance Task you will use many of the concepts and skills that you have learned through this unit. Here's a quick summary.

**Abstractions** simplify the representation of something more complex. They allow you to hide details to help you manage complexity, focus on relevant concepts, and reason about problems at a higher level.

**Functions** (also called “procedures”) are an example of an abstraction in programming. They are named blocks of code. Grouping and naming many simple commands allows programmers to reason about their program in chunks, rather than having to think about individual commands.

**Top-Down Design** is a programming technique that breaks down a large programming task into chunks. Each chunk becomes a function in your program. If a chunk is still too large to easily program then it is divided into even smaller chunks which are given additional functions of their own. Programs written in this way will have layers of functions with high level functions calling lower level ones.

**Collaboration** in programming is much easier when using Top Down Design. Once a task is divided into many chunks, they can then be assigned to individual programmers who are responsible for writing the functions (or layers of functions) that solve that piece. Their code can later be combined to solve the original problem.

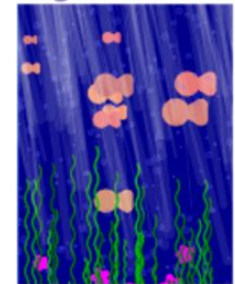
## Target Image



## Broken Into Functions

```
hide();
penUp();
drawBackground();
drawAllSeaGrass();
drawAllStarfish();
drawAllFish();
drawAllFish();
drawAllSeaGrass();
drawAllBubbles();
drawAllSunBeams();
```

## Digital Scene



## Project Description

**Description:** Work with a group to design and program a digital scene. You will use Top Down Design to divide your scene into logical chunks. Then each member of your group will write the code to complete a portion of the scene. Finally your group will combine your code into a single program that draws the scene.

**AP Create Performance Task:** This project is designed to closely match the Create Performance Task. The submission guidelines provide more details on what elements are taken from the Create PT.

**Programming Concepts:** Your program code will need to use many of the programming concepts you learned in this unit. More details can be found in the rubric but you should expect to use layers of functions, loops, parameters, and commenting. The “Under the Sea” project is a good example.

## You will submit individually

- **PDF of program code** with a rectangle around an abstraction that you developed.
- **PDF of written responses** describing the purpose of your program, your development process, and the abstraction that you developed

## You will submit as a group

- **Group Planning Guide**
- **Link to your Digital Scene project.** Only one group member needs to submit this project



# Practice PT Submission Guide - Design a Digital Scene



## Reading these Guidelines

The language in this document is taken in large part from the [Create Performance Task Description](#) (pg 113) and [2018 Create Scoring Guidelines](#). It is highly recommended that you read those documents.

## Written Responses Submission Guidelines

*Selected portions for this Practice Performance Task*

AP Performance Task - Create	Tips and Comments
<p><b>2. Written Responses</b></p> <p>Submit one PDF file in which you respond directly to each prompt. Clearly label your responses 2a, 2b, 2d in order. Your response to all prompts combined must not exceed 550 words, exclusive of the Program Code.</p> <p><b>Program Purpose and Development</b></p> <p><b>2a.</b> Provide a written response that:</p> <ul style="list-style-type: none"> <li>identifies the programming language</li> <li>identifies the purpose of your program</li> <li><i>and explains what the user should expect to see when they run the program.</i></li> </ul> <p><i>(Must not exceed 150 words)</i></p> <p><b>2b.</b> Describe the incremental and iterative development process of your program, focusing on two distinct points in that process. Describe the difficulties and/or opportunities you encountered and how they were resolved or incorporated. In your description clearly indicate whether the development described was collaborative or independent. At least one of these points must refer to independent program development.</p> <p><i>(Must not exceed 200 words)</i></p> <p><b>2c.</b> <i>(omitted for this project)</i></p> <p><b>2d.</b> Capture and paste a program code segment that contains an abstraction you developed individually on your own (marked with a rectangle). This abstraction must integrate mathematical and logical concepts. Explain how your abstraction helped manage the complexity of your program.</p> <p><i>(Must not exceed 200 words)</i></p>	<p><i>These tips and comments are not part of the official Create PT. They are intended to help you better understand the prompts. Many tips are taken directly from the Scoring Guidelines.</i></p> <p><b>2a</b></p> <ul style="list-style-type: none"> <li><i>JavaScript is the programming language used in App Lab.</i></li> <li><i>The purpose can simply be to make a randomly generated drawing.</i></li> <li><i>“Explains what user should expect..” is added to this version of the activity since you will not submit a video of your program.</i></li> </ul> <p><b>2b</b></p> <ul style="list-style-type: none"> <li><i>Keep a few notes as you program. It will make it easier to identify opportunities / difficulties later.</i></li> <li><i>Make sure you describe the entire process, not just the two distinct points</i></li> <li><i>“Incremental and iterative” means that you continuously improved your program based on feedback, testing, or reflection. For credit you need to refer to this feedback, testing, or reflection in your response.</i></li> <li><i>For the two distinct opportunities / difficulties make sure you include how they were resolved</i></li> </ul> <p><b>2d</b></p> <ul style="list-style-type: none"> <li><i>Make sure you copy and paste the code of your abstraction into your written responses, even though you also are drawing a rectangle around it in your code. Switching to text mode may help.</i></li> <li><i>The “how” is important. Don’t just explain what your abstraction is. Explain how it makes your program easier to read or reason about.</i></li> <li><i>Sentence starters: “An abstraction in my program that I developed is the [blank]. It manages complexity in my program by [blank]”</i></li> </ul>

## PDF of Program Code Submission Guidelines

Selected portions for this Practice PT

AP Performance Task - Create	Tips and Comments
<b>3. Program Code</b>  Capture and paste your entire program code into a document. <ul style="list-style-type: none"><li>• Mark with a <b>rectangle</b> the segment of program code that represents an abstraction you developed.</li><li>• Include comments or acknowledgments for program code that has been written by someone else.</li></ul>	<p><i>These tips and comments are not part of the official Create PT. They are intended to help you better understand the prompts.</i></p> <ul style="list-style-type: none"><li>• “Include comments....for program code” means inserting comments (by writing // before a line of code) into the code itself before printing.</li><li>• On the Create Performance Task you are allowed to collaborate with a partner but must carefully indicate whose work is whose with comments.</li><li>• Make sure the abstraction you identify is program code you wrote.</li><li>• In almost all cases the abstraction you choose will be a function you wrote. You’ll want to draw a box around the place where you wrote that function code. Suggest: A good one to choose would be a function that you wrote that calls other functions you wrote, because it demonstrates abstraction and managing complexity.</li><li>• To make a PDF and draw a rectangle, we suggest using the “Code Print” program here: <a href="https://bakerfranke.github.io/codePrint/">https://bakerfranke.github.io/codePrint/</a></li></ul>

## Group Planning Guide Submission Guidelines

Your collaborative programming group will submit a **single copy** of your group planning guide. Make sure that all group members’ names are listed on the document.

## App Lab Project Submission Guidelines

When you have completed your digital scene, submit it by clicking the **Submit Project** button on the proper App Lab project in Code Studio (the project associated with this lesson). Submitting indicates that your project is ready to be reviewed.

# Group Planning Guide

**Choosing a topic:** Before you begin make sure you've reviewed the submission guidelines and rubric. Then pick a scene that has the following features:

- Have several components that allow it to be broken into logical chunks (functions)
- Have repeated elements that will allow you to use loops and random values
- Use a function with a parameter (same figure but different size / color / dimensions, etc.)

<b>Scene Description</b> What is the topic? What are the different pieces of the scene?  _____  _____  _____  _____  _____  _____  _____  _____	<b>Scene Sketch</b> Make a quick sketch of the scene. Write notes to clarify points. Use the back of this sheet if you need.
---	--

## Identify Top-Level Functions and Assign to Group Members

Use Top Down Design to identify the major components of your scene. Give each component a top-level function. In the Under the Sea project these were `drawAllFish()`, `drawAllSeagrass()`, `drawAllBubbles()`, etc. Then assign each function to a member of the team to program individually. They may need to further divide their component into smaller functions later.

Scene Component	Function Name	Group Member

(use back side of page if you need more)

# Practice PT Planning Guide - Design a Digital Scene



## Background

The beginning of our programming unit has focused on how to use Top-Down Design as a strategy for thinking about how to design and write programs. Functions, parameters, and loops have allowed us to break down increasingly complex drawings into logical pieces. Most recently, we combined all three of these programming constructs to design an “Under the Sea” digital scene.

**Abstraction:** Everyone uses abstraction on a daily basis to effectively manage complexity. In computer science, abstraction is a central problem-solving technique. It is a process, a strategy, and the result of reducing detail to focus on concepts relevant to understanding and solving problems. Top-Down Design is a technique for discovering the levels of abstraction in your problem so that you can effectively write code to solve it.

**Collaboration:** Top-Down Design also helps by breaking up a problem in a way that lets multiple people **collaborate** and work on it at the same time. Often programming projects are divided among many programmers. Each person is responsible for programming a portion of the final product. In order to do this, everyone on the team must understand the overall plan at a high level, but each person only needs to worry about the nitty gritty details of the portion they are responsible for.

## Project: Collaborate to Design a Digital Scene

For this project, you will be **working with a group to design your own digital scene**. Your group can choose any kind of digital scene to create. You have already seen how the “Under the Sea” scene was created, but here are some guidelines to consider when picking your own:

### Your scene should...

- Be of interest to every member of the group
- Have several components that allow it to be **broken into logical** chunks
- Have repeated elements that will allow you to use **loops** and potentially **random** values
- Have elements that would benefit from **creating a function** with a parameter (same figure but different size / color / dimensions, etc.)

Each member of your group will **write the code to complete a portion** or portions of the scene. Then **bring all the code together** to compose the final image! You will **submit your own project and written reflection**, but it must use code written by your teammates.

## General Process

- **Get together with your group** (recommended groups of 3 or 4 people)
- **Do Group Project Planning** (see below)
  - Brainstorm ideas, break the problem down, delegate tasks
- **Do your individual programming and share with teammates**
  - Program your portion of the project
  - Share your code with teammates, and incorporate others' code into your project
- **Complete your digital scene, write responses to reflection questions, and submit**
  - See: [Practice PT Project Overview and Rubric - Design a Digital Scene](#)
  - Submission includes final program and completed reflections questions

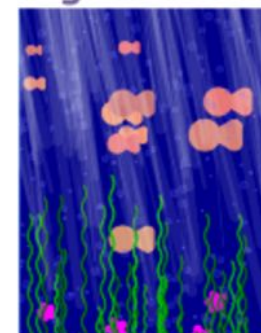
## Target Image



## Broken Into Functions

```
hide();
penUp();
drawBackground();
drawAllSeaGrass();
drawAllStarfish();
drawAllFish();
drawAllSeaGrass();
drawAllBubbles();
drawAllSunBeams();
```

## Digital Scene



# Group Project Planning

Below we have laid out a process for your group to brainstorm, plan, and eventually create your Digital Scene project. Go through each step together. The outcome of the process will be a project plan (see next two pages) that indicates what you're trying to create and who is doing what. **Get started and have fun.**

## Understand the problem & timeline

Identify the problem to solve. It may sound obvious, but it's hard to solve a problem if you don't deeply understand it.

- Review the steps of the process for completing this project (see above and below)
- Review [Practice PT Project Overview and Rubric - Design a Digital Scene](#) make sure you understand what is required of you and your teammates.
- Make sure you also **understand the time constraints**. There's always too much to do and too little time! This will help you prioritize work later.

## Brainstorm

What kind of digital scene will you create? Make it rain with ideas! Whether you're in a group or on your own, take the time to generate lots of different ideas for your digital scene

- Put a time limit on how long you will brainstorm
- No criticism of ideas, only building up on ideas

Once you have decided on what your scene will be, **complete the Project Description below**. It must contain a brief description and at least a rough sketch of what you are trying to create.

## Break it down

Use Top-Down Design to **identify the major components** you will wish to include in your image. At this point **you only need** to break down the image at a high level.

- **Decide as a group what high-level functions** you will need.
- For example, in the Under the Sea project, the high-level functions for each component were things like: `drawAllFish()`, `drawAllSeagrass()`, `drawAllBubbles()`, etc. These high-level functions are all you need to think about at this point.

## Assign tasks & prioritize

Each of the high-level functions you identified **will be programmed by one member of the group**.

- Assign the high-level functions you identified evenly among your group.
- If people have more than one thing to do, **you should prioritize**: what can be cut if you run out of time or if there is an unanticipated bump in the road?
- Once you have assigned functions, each group member can begin working individually to program the components they have been assigned.

**Complete the Project Component Table below**; this is where you will declare what functions you will write *and* who will do what.

Name(s)\_\_\_\_\_ Period \_\_\_\_\_ Date \_\_\_\_\_

## Project Description

Once you have chosen your topic, write a brief description of it below and include either a sketch or digital image that shows what you are aiming to draw. It's fine if your image includes more components or detail than you will be including in your digital scene. You can also have more than one representative image.

**Scene Title:**

**Group Members:**

**Short Description:**

---

---

---

---

**Sketch / Digital Image:**

Name(s)\_\_\_\_\_ Period \_\_\_\_\_ Date \_\_\_\_\_

## Project Component Table

With your group, identify the major components you would like to include in your digital scene. For each component, provide a descriptive and meaningful name for the function that will draw that component, provide a short description of what the function does, and assign the function to a member of the team.

You do not have to fill this table, but you can, and you can add more rows if necessary. This is just a template.

Component	Function Name	Description of Function Behavior	Group Member

# Practice PT Overview and Rubric - Design a Digital Scene



## Overview

You will submit this project and write responses to the reflection questions in the style of the AP® Create Performance Task. The document below has been constructed to mimic the AP Create Performance Task. Some but not all of the language is pulled directly from the AP document. Some of the prompts have been modified slightly or simply omitted for clarity and to better fit the *Design a Digital Scene* project.

## Programming Requirements

### Process

The process of creating your program includes individual and collaborative work. Individual work means some portions of the design, development, and implementation of your program must be completed independently. Collaboration can take different forms and can occur at different times in the program development process.

**You will be required to respond to prompts about your collaboration, as well as to identify the portions of your program that were created independently. The following are examples of different forms of collaboration:**

- A. Collaboration can take the form of brainstorming and sharing ideas before the process of writing code begins. Partners can then choose to work together or independently at selected times during the programming process.
- B. Collaboration can take the form of working together to develop an idea, beginning the programming process together, and then working independently to add different features to the collaboratively-developed portion of the program.
- C. Collaboration can involve Pair Programming, in which one partner “drives” (enters code) while the other “navigates” (recommends and reviews code entered by driver), with the partners changing roles after designated time intervals.
- D. Collaboration can involve each partner developing pieces of the program and combining those pieces during the development process.
- E. Collaboration can blend any or all of the above techniques and may include an iterative process in which one or more of these techniques, or other collaboration techniques, are employed several times in the program design, development, and testing phases.

### Program

Your program must demonstrate a variety of capabilities and implement several different language features that, when combined, produce a result that cannot be easily accomplished without computing tools and techniques. **You will be required to respond to prompts about the program development process and your program code, including questions about the abstractions you used. The program must demonstrate:**

- Use of several effectively integrated programming elements from the programming language you are using
- Use and creation of abstractions to manage the complexity of your program (e.g., functions/procedures; abstractions provided by the programming language; APIs)



## Submission Requirements

### 1. Group Planning Document

As a group, you will submit a **single copy** of your group planning document. Make sure that all group members' names are listed on the document.

### 2. Program Code

When you have completed your digital scene, submit it to your teacher by clicking the "Submit Project" button on the proper App Lab project in Code Studio (the project associated with this lesson). Submitting indicates to the teacher that your project is ready to be reviewed.

Your final digital scene **code should:**

- contain the functions you wrote and the functions you got from your teammates
- make calls to both the functions you wrote and your teammates' functions in your program.

**Note:** It is OK if you had to alter your teammates' code slightly to make it work in your program, or to improve its functionality.

### 3. Individual Written Responses

After completing your project, respond to each of the following reflection questions. **Your response to any one prompt must not exceed 300 words.**

- Provide an overview of the purpose of your program and how your program code works. Describe the most important program features, rather than providing a line-by-line summary of the program code.
- Describe the most difficult programming problem you encountered while writing your individual code. What was the difficulty? Explain how you resolved it.
- Identify an abstraction used in your program and explain how it helped manage the complexity of your program.
- Explain in detail points in your development process where collaboration was used.
  - Describe the form of collaboration you used. Refer to Process section A-E in your description.
  - Explain how this collaboration affected your program development. Cite specific examples from the collaboration, such as how the group worked together to arrive at solutions, or feedback that you gave and received.

# Rubric - Design a Digital Scene



Component	1	2	3	Score
<b>Group Planning Document</b>				
<b>Project Design</b>	The description and/or sketch/digital image of the design are simplistic and lacking in details. Not enough information is given to realistically build a program from.	The description and/or sketch/digital image are limited in details. While it might be possible to program from the design, there are too many details missing for the programming task to be easy.	The description and/or sketch/digital image are rich in details. A programmer would have few questions and find it easy to work from this design.	
<b>Top-Down Design</b>	The image has not been broken into logical components, or most components have not been assigned a high-level function with a descriptive name.	Most aspects of the image have been broken into components; however, the components are not distinct or logical and do not represent top-down design, or the functions are poorly named.	The image has been broken into logical components that represent top-down design; each component has been assigned a high-level function with a descriptive name.	
<b>Task Assignments</b>	Tasks have not been evenly divided among team members and/or tasks have not been prioritized.	Tasks have been divided among members but the assignments and prioritizations do not reflect a realistic estimate of the time constraints or anticipate problems that might arise.	Tasks have been prioritized and evenly divided among members with considerations made for timing or problems that might arise.	
<b>Program Code</b>				
<b>Functions and Abstraction</b>	The program does not make use of the high-level functions agreed upon by the team.	The program makes limited or inconsistent use of levels of abstraction and the functions created by the team members.	Appropriate levels of abstraction are expressed in code using the high-level functions created by the team. (Modifications of functions are allowed.)	
<b>Functions with Parameters</b>	The program does not feature a function with a parameter.	Program contains a function with a parameter that is used in a limited or trivial way.	Program contains at least one function with a parameter to control behavior in a meaningful way.	
<b>Loops</b>	The program does not feature a loop.	A loop is used in a limited or trivial way to repeatedly execute portions of code.	A loop is used in a meaningful way to repeatedly execute portions of code.	

<b>Functionality</b>	There is a weak connection between the output of your function(s) and the function descriptions agreed upon by the group.	There is a moderate connection between the output of your function(s) and the function descriptions agreed upon by the group.	There is a clear and obvious connection between the output of your function(s) and the function descriptions agreed upon by the group.	
<b>Final Scene Incorporates Work from Teammates</b>	Final digital scene does not make use of code written by other teammates.	Final digital scene uses code written by some of the other teammates.	Final digital scene uses code written by each of the teammates. (Note: Students may make alterations to their teammates' code as needed to function correctly in the final scene.)	
<b>Individual Written Responses</b>				
<b>a. Program Overview</b>	The connection between the program and its purpose is unclear. Or it is unclear how the program features connect to the purpose.	There is a logical connection between the program and its purpose. Or the purpose of the program is weakly supported by the features identified.	There is a compelling connection between the program and its stated purpose, supported by details of the important features identified.	
<b>b. Difficulties Encountered</b>	The response generally describes the development of the program without clearly describing any specific problems.	The response describes the developmental steps of the program, but it includes little or no information about how any problems were addressed.	The response fully describes the development details that enable the reader to understand the the difficulties that were encountered and how they were resolved.	
<b>c. Abstraction</b>	The explanation of how the selected code illustrates abstraction is incorrect or incomplete.	The explanation of how the selected code illustrates abstraction is mostly complete but lacks a clear explanation of how it helps manage the complexity of the program.	The explanation of how the selected code illustrates abstraction is well-supported by details and clearly describes how it helps manage the complexity of the program.	
<b>d. Collaboration</b>	The response describes a non-collaborative process, or an ineffective collaborative process. The explanation does not describe how the process affected the program development.	The response cites some effective collaboration, but it is unclear how the collaboration affected the program development or any feedback was provided or incorporated.	The response describes effective collaboration and cites specific examples of how collaboration impacted the program development. Examples of providing and incorporating feedback are included.	