PART 2

A CONTEXTUALIZED

PRE-AP COMPUTER PROGRAMMING CURRICULUM:

MODELS AND SIMULATIONS FOR EXPLORING

REAL-WORLD CROSS-CURRICULAR TOPICS

CHAPTER 2.

THE COURSE OUTLINE FOR CPRWE: COMPUTER PROGRAMMING AS IF THE REST OF THE WORLD EXISTED

Section 1. Introduction

CPRWE (Computer Programming as if the Rest of the World Existed), is a year-long introductory programming course intended to give students (1) a rigorous overview of and basic literacy in the uses of a structured programming language, using the Java-based language *Processing*; and (2) familiarity with algorithmic problem-solving. Within the context of programs of mid-level complexity and size, and cross-curricular fields of application (science, art, humanities), students learn the uses of variables, Boolean expressions, and iterative and conditional control structures. They learn to encapsulate code within methods, pass input (arguments) via parameters, and calculate return values. Students learn to think of programs as interactions of objects having attributes and methods that they describe in classes. They learn software engineering principles for top-down design, resulting in hierarchically organized programs for optimal maintenance, modification and extendibility. They examine criteria for deciding which of competing code styles and algorithms to implement. Equally important, they learn the possibilities for non-trivial applications of programming to study and solve diverse problems across the STEM, Humanities and Arts curriculum. To write accurate programs, students learn and use cross-curricular concepts from such core areas as math (e.g. algebra, trigonometry), chemistry (electro-negativity, covalent and hydrogen bonds) and biology (DNA structure and genetics). In order to give purpose and context to the programming task, students study a film or play that situates the target problem within an historical / societal context. For example, in the Astronomy unit (*Galileo's Revolution*), students build (a) a Copernican simulation of the solar system to understand such observational phenomena as the phases of Venus, Mars in retrograde, and the infrequency of solar eclipses; and (b) a Ptolemaic simulation (using *epicycles*) to prove that such a model cannot account for all the phases of Venus. They then study Bertolt Brecht's play *Life of Galileo,* and consider the repercussions of the discovery of the phases of Venus (a) in accelerating the pace of both the scientific Renaissance and the religious Reformation, and (b) in weakening the Church and eventually the monarchies of Europe.

The course is divided into three sections:

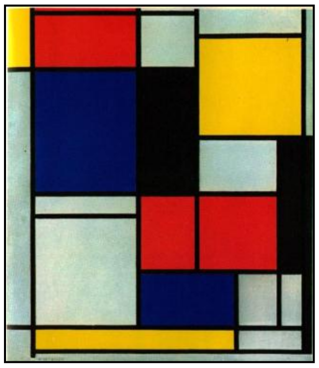
a) Basic Programming Skills and Introductory Projects: Graphics basics, Primitive Methods / Arguments, Coordinate Plane Manipulations, Processing Mouse and Keystroke Events, Animation. setup() Initialization Code; draw() Animation Code, User-Defined void Methods / Blocks / Indentation, Variables, System Variables, Classes / Objects, Arrays, Iteration.

b) Building Programming Skills: Methods that return values, Primitive Types (int, boolean), Method Parameters, Hierarchy / Nested Conditional Statements

c) Intermediate Projects: Software Engineering Principles, Multiple Structural Recursion, Inheritance, Polymorphism.

The first section is intended as a "*whole language*" approach where students learn to recognize and use basic programming components to build four **dynamic art** programs sequenced in increasing levels of sophistication. The second section uses **CodingBat** to help students gain proficiency in programming skills and recognize programming issues that involve Boolean logic, strings, arrays and iteration. The third section is comprised of four multi-week projects where students build scaled down, but functional, **applications of real-life software programs**, and use them to examine or solve specific problems in **government**, **geography**, **astronomy** and **molecular biology**.

Section 2. Piet Mondrian Painting
(PART I: INTRODUCTORY PROJECTS)



Essential Question
How does one **design** a computer program?

Supporting Questions
Where does a computer program begin **execution**?
What is a **method**?
How does a **primitive** method differ from a **user-defined** method?
Does the order of **arguments** in a method **call** matter?
What is **hierarchical** organization?
What are the advantages to **organizing** one's code?
How do **syntax** errors differ from **logic** errors?
How do methods that set **modes** operate?
How does one fix **bugs** in a program?
How does the **RGB** color system – and transparency – work?

Description

The unit introduces students to the **Processing** programming environment and familiarizes them with its **inverted Cartesian coordinate plane** (origin at the upper left corner). They learn basic **drawing** and **mode** methods for rendering regular and irregular shapes. They learn the programming concept of **hierarchical organization** by defining **methods** with **meaningful names** and grouping primitive functions into the **method bodies**. They learn to call these methods sequentially in the program's **entrance point** method **setup**(). They learn **indentation** conventions to organize lines of code for legibility. They learn to add **comments** to their program to clarify intent. They learn that the settings of **mode** methods persist beyond the methods in which they are used – until they are next changed. They learn simple **debugging** techniques for locating the source of **logical** errors. They learn about programming language **syntax**, such as matching parentheses and curly braces, and the order of method arguments; they learn to debug **syntax** errors. Students learn the **RGB** color system and transparency.

Key Assignments

Following an introduction to the Processing programming environment and basic drawing methods, students are given an image of the Piet Mondrian painting and shown how to determine coordinates of rectangle vertices and line endpoints using the system tool **Paint**.

Students then write a hierarchically organized program that renders a full-scale and close approximation of the image. In the course of completing the task, students:

1. Create parameter-less user-defined methods made up of primitive methods.
2. Call primitive methods using the correct coordinates and widths/heights.
3. Call user-defined methods in setup();
4. Use primitive **mode** methods at the beginning of each user-defined method to avoid persistence side-effects.
5. Employ simple debugging strategies to locate and correct syntax and logic errors.

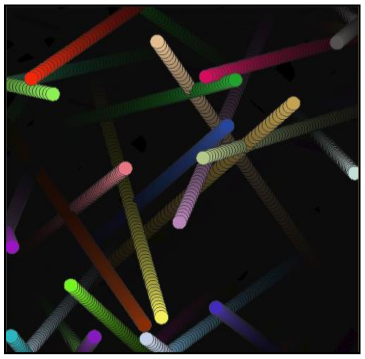
Teaching Strategies

Instructor uses <u>direct whole-class instruction</u> to demonstrate the **Processing** programming environment and its inverted Cartesian coordinate plane (origin in the <u>upper</u> left corner) as students practice at their workstations and instructor and advanced students <u>circulate</u> to help students having problems. Instructor similarly guides students through the use of basic drawing methods, and accessing/reading online documentation. Using <u>guided discovery</u>, students learn by examining program output: (a) how colors are defined using RGB values; and (b) what the various attributes do that **mode** methods set.

Instructor clarifies how primitive methods work by using counterexamples, e.g. a different ordering of primitive methods or method arguments (signatures), to show incorrect or unintended program output.

Instructor helps students individually and via direct whole class instruction to debug syntax and logic errors in their program. Syntax errors in this assignment are limited to orphaned opening or closing curly brackets or parentheses; and using a different ordering or number of method arguments than those specified in the documentation. Instructor teaches (a) students to recognize these specific errors, (b) procedures for avoiding these errors, and (c) simple strategies for locating these errors when they occur (commenting out lines, use of auto-indent feature). The most common logical error in this assignment is calling a **mode** method in one part of the program and not resetting the attribute in a subsequently called method. Teacher shows students that routinely calling mode methods at the beginning of user-defined methods, though seemingly repetitive, avoids this kind of error. Instructor shows students the use of print statements and comments for debugging logic errors.

Section 3. Ricocheting Comets



Essential Question
How does one program the **simulation** of **movement**?

Supporting Questions
How can the **draw**() method be used to simulate movement?
What is difference between using the **background**() statement in setup() vs. draw()?
How are **variables** in programming similar to and different from variables in Algebra?
How and where do you **declare**, **initialize**, **use** and **update** variable values?
What is an assignment statement, and what are the various forms it can take?
What is a **system variable**? [width, height]?
When in the execution of a program do system variables acquire meaningful values?
How can a **conditional** statement detect when an object reaches a specific location?
What's the relationship between a dividend being evenly divisible by a divisor and the **MOD** function?
Why use **parentheses** in conditional expressions if there is no difference in expression evaluation, i.e. if **order of operator precedence** yields the same result?
How do you use a variable to set the **speed** or **direction** of an object?
What does the **random**() method do; in what kind of situations would you want to use it?
How does a **class** allow you to create multiple objects of a given type?
What are **instance variables**?
What is a class **constructor**?
What's the difference between a **class** and an **object**?
How does a class allow you to alter **attributes** of objects so that they look and behave differently?

Description

This unit introduces students to programming strategies that **simulate movement**. Students learn to combine the use of drawing methods, **variables** and **conditional** statements to move a circular object across the screen and make it ricochet off the edges. Students learn the **modulus** function and some of its uses in conditional statements. They learn the advantages to using variables instead of hard-coded values. Students learn to combine simple conditions into complex conditional expressions using the logical AND && and OR || operators. They learn to use **parentheses** inside of complex conditional expressions in order to make the **intent** of their code **clear** and to avoid ambiguity. They learn how to use the **random**() method to dynamically change the color of objects. Students learn to define **classes** and use them to instantiate multiple **objects** of that class.

Key Assignments

The instructor guides students through the construction of a small program in which a circular object moves horizontally, reversing direction when it reaches the left and right edges of the window. Students then:

1. Write a program in which a circular object moves vertically, bouncing off of the top and bottom edges.
2. Write a program in which a circular object moves diagonally and ricochets off each of the four edges.
3. Modify #2 so that the circular object simulates <u>*realistic*</u> ricocheting behavior, i.e. bounces when its outer edge touches the boundary, rather than its center.
4. Calculate algebraic expressions for the slopes of the diagonal segments; and calculate the slope-intercept form of the equations for the lines lying on those diagonal segments.

In the course of completing these tasks, students learn to create different variables, each of which accomplishes a single task. For example, they must create separate variables for horizontal and vertical movement (*position* and *increment*); they must create a radius variable for modeling an object's outer edge.

Instructor introduces students to the **random**() method, and shows them how it can be used to change the color of an object. Students then:

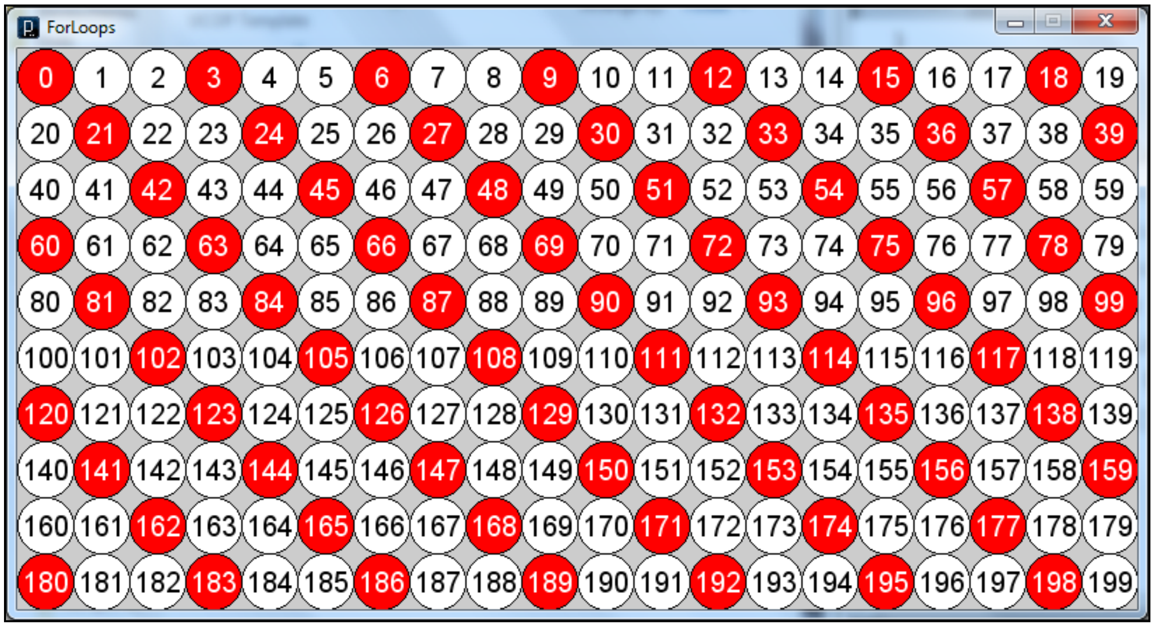
5. Modify program #3 to change the color of the circular object whenever it bounces off of an edge.

Instructor shows students the syntax for defining a **class**. Instructor shows how to write a **constructor** for initializing an object's attributes, and how to transfer the code developed previously into class methods with meaningful names. Instructor shows students how to **instantiate** an **object** of the new class, and how to call the object's methods in **setup**() and **draw**(). Students then:

6. Add 15 objects to the program, each having different starting directions and speeds.

Finally, instructor shows students how to use a combination of transparency values and drawing methods to give the illusion of a fading trail to a moving object. Students then:

7. Modify their programs to give the illusion of fading trails to their moving objects.



```
0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16  17  18  19
20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39
40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59
60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79
80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99
100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119
120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139
140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159
160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179
180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199
```

8. Students solve a series of increasingly complex problems using the **200-circle matrix program** (above), modifying only the conditional statement to produce the correct output of correctly numbered red circles. For example, the conditional statement that produces the output shown above is:

```
if (i % 3 == 0) {
     drawRedCircle(i);
}
```

Teaching Strategies

Instructor uses direct whole-class instruction, and circulates to help students. Students learn by **guided discovery**, observing the runtime output of **counterexamples**. Two cases detailed below show the use of these strategies.

1. When first coding the conditional statement to detect when the object reaches the window's right edge, after initializing the program with a **`size(800,600)`** statement, students generally write: **`if (x == 800)`**. When we change either the starting x position or the increment so that x will leapfrog over 800, students modify this expression to **`if (x > 800)`**. Instructor then demonstrates the system variable **`width`**, and how it takes the value of the first argument to the **`size`**`()` method. Instructor then directs students to change the first size() argument from 800 to 700. Students observe that the object goes past the right edge, but eventually reappears in reverse direction. Instructor asks students to modify the program so that the object will ricochet at the right edge no matter what the width of the window is. Although several students will change the conditional expression to **`if (x > width)`**, a common error is for students to simply replace 800 with 700. Eventually, with enough prodding,

students make the correct change and come to understand the power of variables to make a program behave properly with varied input.

2. When teaching how the **random**() method operates, students are presented with an alternate way of coding the part of their program that changes the color of their objects once they ricochet. The original code is:

```
this.clr = color( random(256), random(256), random(256) );
```

The new code is:

```
     color newColor = random(256);
     this.clr = color (newColor, newColor, newColor);
```

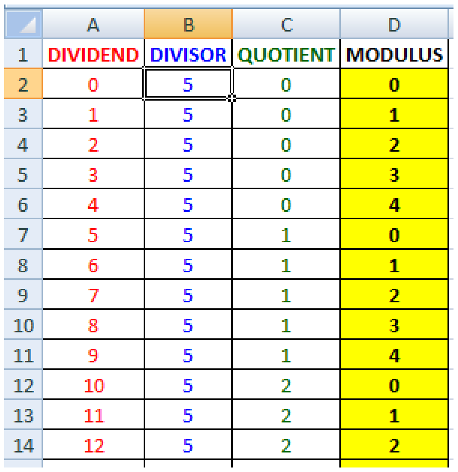
Without running the program, the question is put to the class whether the new code will have output equivalent to the original. After discussion, everyone tests the code and sees the different output, now limited to grayscale colors, rather than the full palette. Each student is asked to write a paragraph explaining why the new code results in different output. Students are then asked to modify the new code, all the while maintaining use of the **newColor** variable, and make it work (spoiler: use 3 variables). It is through these series of experiments that students learn to appreciate not just how the **random**() method works, but to recognize that misunderstanding of an API method can lead to logic errors because the programmer may use it incorrectly.

When teaching students how to reverse an object's direction, students are prompted to come up with several code fragments for reversing the sign of a number, for example:

```
     n = -n;
     n = n * -1;
     n *= -1;
     n = n - (2 * n);    // which simplifies to the first statement
```

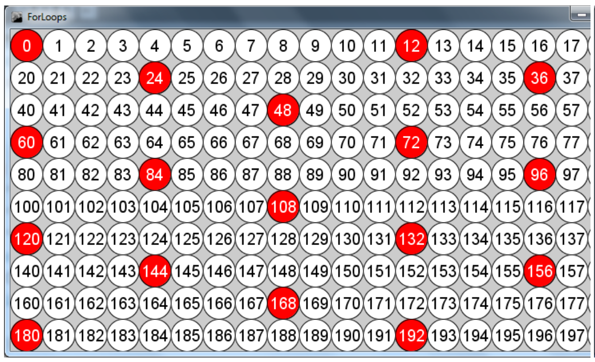
Counterexamples are also used in direct whole class instruction when introducing logical **AND** and **OR** expressions. **Venn diagrams** and **number lines** are used to graphically illustrate the difference between **AND** and **OR**, and to teach that **AND** corresponds to the *SET* concept of **INTERSECTION**, and **OR** corresponds to the *SET* concept of **UNION**.

Instructor distributes the Excel worksheet (at right) that only allows students to change the value in cell B2 (the divisor value). Through experimentation, students study the patterns of output when doing integer division (quotient) and modulus operations given **dividend** and **divisor** as inputs, and deduce that a modulus output of zero indicates that a dividend is evenly divisible by a divisor.



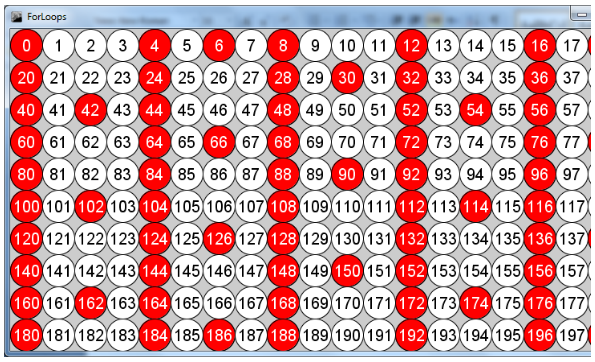
	A	B	C	D
1	DIVIDEND	DIVISOR	QUOTIENT	MODULUS
2	0	5	0	0
3	1	5	0	1
4	2	5	0	2
5	3	5	0	3
6	4	5	0	4
7	5	5	1	0
8	6	5	1	1
9	7	5	1	2
10	8	5	1	3
11	9	5	1	4
12	10	5	2	0
13	11	5	2	1
14	12	5	2	2

In direct whole class instruction, example practice problems using the 200-circle matrix program are demonstrated to get students started on the task. Below is the output showing the difference between && (intersection) and || (union). Students also recognize that the && expression is equivalent to i % 12 == 0 and can be used to reveal the lowest common denominator.



0	1	2	3	4	5	6	7	8	9	10	11	**12**	13	14	15	16	17
20	21	22	23	**24**	25	26	27	28	29	30	31	32	33	34	35	**36**	37
40	41	42	43	44	45	46	47	**48**	49	50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67	68	69	70	71	**72**	73	74	75	76	77
80	81	82	83	**84**	85	86	87	88	89	90	91	92	93	94	95	**96**	97
100	101	102	103	104	105	106	107	**108**	109	110	111	112	113	114	115	116	117
120	121	122	123	124	125	126	127	128	129	130	131	**132**	133	134	135	136	137
140	141	142	143	**144**	145	146	147	148	149	150	151	152	153	154	155	**156**	157
160	161	162	163	164	165	166	167	**168**	169	170	171	172	173	174	175	176	177
180	181	182	183	184	185	186	187	188	189	190	191	**192**	193	194	195	196	197

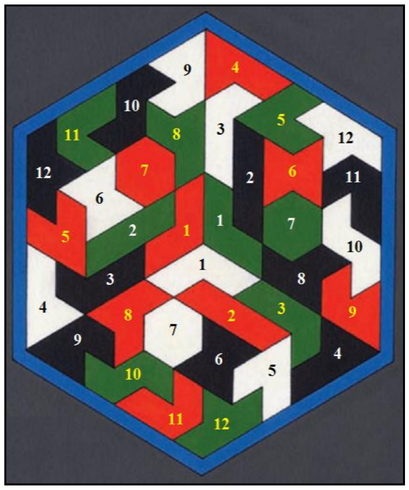
`i % 4 == 0 && i % 6 == 0`
(*intersection*)



0	1	2	3	**4**	5	**6**	7	**8**	9	10	11	**12**	13	14	15	**16**	17
20	21	22	23	**24**	25	26	27	**28**	29	**30**	31	**32**	33	34	35	**36**	37
40	41	**42**	43	**44**	45	46	47	**48**	49	50	51	**52**	53	**54**	55	**56**	57
60	61	62	63	**64**	65	**66**	67	**68**	69	70	71	**72**	73	74	75	**76**	77
80	81	82	83	**84**	85	86	87	**88**	89	**90**	91	**92**	93	94	95	**96**	97
100	101	**102**	103	**104**	105	106	107	**108**	109	110	111	**112**	113	**114**	115	**116**	117
120	121	122	123	**124**	125	**126**	127	**128**	129	130	131	**132**	133	134	135	**136**	137
140	141	142	143	**144**	145	146	147	**148**	149	**150**	151	**152**	153	154	155	**156**	157
160	161	**162**	163	**164**	165	166	167	**168**	169	170	171	**172**	173	**174**	175	**176**	177
180	181	182	183	**184**	185	**186**	187	**188**	189	190	191	**192**	193	194	195	**196**	197

`i % 4 == 0 || i % 6 == 0` (*union*)

Section 4. Rotating McClure Painting



Essential Question
How is a single method able to do different things?

Supporting Questions
What is **iteration**? Why use it?
How do **for-loops** allow you to do repetitive tasks or calculations?
What are the advantages to using a list (**array**)?
How are **members** of a list different from variables of the same type?
Can a Java list contain objects of **more than one type**?
What is the **connection** between iteration and lists?
What kinds of programming errors cause **side-effects**?
How can programmers avoid coding **side-effects**?
How do you determine the **order** that you list **parameters** when defining a method?
Does the order that you do **transformations** (translate, rotate) matter?
Is **math** always involved when writing graphics programs?

Description

Students analyze a geometric image of a hexagonal painting. They recognize that the hexagon is composed of 3 identically shaped rhombuses (although component colors vary), and hypothesize that the program can draw the entire image by coding for the display of just one rhombus, then "rotating the drawing code" twice. Instructor gives students a helper-program that will create the code for a Java array of Points as they click on the 38 vertices of the 12 irregular polygons that make up each rhombus. Students splice this array into the beginning of their program, tweak vertex values for accuracy, and use the indexed points to write 12 methods for drawing the irregular polygons. Student write the methods by **bookending vertex**() method calls – which use indexed points as parameters – between **beginShape**() and **endShape(*CLOSE*)**. Students then encapsulate these 12 methods into a higher-level method called **drawRhombus**(), which is placed in **setup**() [because the image does not (yet) move, there is no need to involve the **draw**() method at this point].

Students learn to code for the transformation by using the sequence: **translate**() - **rotate**(), which will **translate** the drawing plane's **origin** to the **center** of the figure, then **rotate** the drawing plane 120º in either direction before drawing the second and third rhombuses. Because *Processing's* **rotate**() method takes radians as input (rather than degrees), students learn the definition of **radian** and the common equivalents for standard angles (multiples of 30º and 45º). They are also shown the **radians**() method, that lets them simply wrap it around the more familiar degrees measures.

Students must also write a method **translatePoints**() that will translate each member of the Points array in the direction opposite to the translated origin in order to keep the hexagon center in the middle of the window. Prior to attempting this task, students gain a working knowledge of the **initialization**, **condition**, and **increment** parts of the regular **for-loop**.

To **paint** the rhombuses with the correct colors, students analyze the image for color patterns. They discern that there are 4 **sets** of 3 identically colored (**red**, **green**, **white**, **black**) polygons in each rhombus (1-5-7, 2-8-11, 3-10-12, 4-6-9). Each rhombus, however, colors the 4 sets differently. Student add four parameters to the **drawRhombus**() method. They then consider two methods for solving the problem: (a) leave the 12 polygon methods in number sequential order, and use 12 color-setting mode calls, one preceding each polygon method, or (b) regroup the 12 polygon methods according to their color set and precede each of the 4 groups with a color-setting **mode** call. For further clarity, students create 4 **local** color variables for use as **arguments** in the **drawRhombus**() method call.

Finally, students make the image dynamic by moving the 3 **drawRhombus**() calls to draw(), and creating a global **angle** variable that is incremented at the end of draw(). This revisits the movement programming mechanism used in ***Ricocheting Comets***, but for **angular**, rather than lateral, movement.

Key Assignments

1. Using the helper-program, students place a functioning code fragment for the **Points array** at the beginning of their McClure program.
2. Using the Points array's indexed point variables, students create 12 parameter-less methods for drawing the 12 irregular polygons in a single representative rhombus, and place these methods in a *working* higher-level user-defined method called **drawRhombus**().
3. Students write a method called **translatePoints**() that recalculates the Points array coordinates so that the center point's coordinates is at the origin (0, 0).
4. Students write code for the **translation** and **rotation** transformations that allow one to draw the other two rhombuses with drawRhombus().
5. Students add **4 color parameter**s to drawRhombus() and modify the method body to paint each rhombus with the correct colors.
6. Students **declare**, **initialize**, **use** (with an additional **rotate**() call) and **update** a variable named **angle** that allows the image to rotate.

Teaching Strategies

Instructor uses **modeling** to help students understand the geometric transformation the program uses to draw the 2nd and 3rd rhombuses. The model likens the graphic drawing plane to a large sheet of paper, and the **drawRhombus**() method to a stamp. If one imagines that the paper does not move, then one must rotate the stamp to draw the 3 rhombuses. Implementing this would require the programmer to calculate a complete set of vertex coordinates for each of the 2 additional rotated rhombuses. Instead, the preferred method is to consider the **drawRhombus**() method as **fixed**, and to simply **rotate the sheet of paper** beneath it before "stamping" it. Instructor also makes an analogy to drawing a circle with a compass, using each of these procedures. Because the point of rotation is the paper's origin (top-left), a **translate** method needs to reposition the **origin** at the **center** of the hexagon prior to the rotation operation.

Using the 200-circle matrix program, Instructor uses guided discovery so that students see the effect of changing the **initialization**, **condition** and **increment** parts of the for-loop code that draws the red circles. In this way, students gain a working knowledge of how these 3 parts work together to perform an iterative task.

Instructor uses **guided discovery** to help students figure out how to write the **translatePoints**() method that will adjust the coordinates of the Points array so that the center point moves to (0, 0), and the remaining 37 points are translated an identical amount. Using a for-loop, students adjust point[0] and subtract its coordinates from the successive 37 points. Output shows that only point[0] has been modified. Instructor directs students to set the initialization part of the for-loop to 1 rather than 0. Students observe that the output is correct for all but the center point. Instructor asks students to figure out why these **side-effects** occur. Students finally discover that they need to save off the original coordinates of point[0], then subtract these from all 38 points in the array. Students are thus made aware of the phenomenon of unintended side-effects that stem from altering a variable referred to <u>**during**</u> the performance of a task.

Section 5. Word Clouds

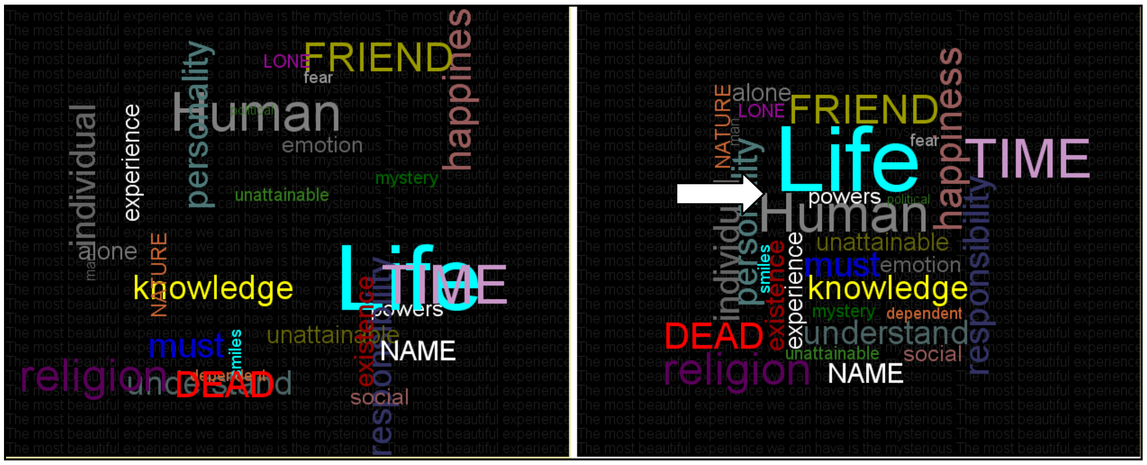
Essential Questions

What are the advantages to using **classes** in the organization of a program?
How does one isolate **side-effects** generated from transformation operations?

Supporting Questions

How is the **pushMatrix**()-**popMatrix**() combination similar in usage to the **beginShape**()-**endShape**() pair encountered in unit 3?
How does the **pushMatrix**()-**popMatrix**() combination prevent side-effects?
How do you use the **random**() method to place a word object along a window edge?
What are the advantages to using a **for-each** loop over a regular **for** loop? Are the two interchangeable?
How do you **synchronize** two or more events?
How do you write code to dynamically **alternate** between objects being in **motion** and then **at rest**?

Description



This unit teaches students how to use write programs that draw **text**. Students learn these new text methods, and are introduced to the **for-each** loop. They learn how to isolate transformation operations needed to render each word from having side-effects on subsequently drawn words by **bookending** commands between **pushMatrix**() and **popMatrix**() calls. The Word Cloud program intertwines these new concepts with the major programming concepts revisited from the first 3 units: **variables, conditional statements, Boolean expressions**, **arrays**, **classes**, **iteration** and **movement**.

Students spend time finding out about and experimenting with word clouds. They find lengthy pieces of text ranging from essays to state documents, and use them as input to any number of Internet word cloud programs referred by the Instructor. The instructor guides the class through the construction a simple program that shows how to **create fonts** and use them to **output text**. These methodologies are then encapsulated in a **DynamicText** class whose constructor takes a list of parameters for text, font, size, position, color, rotational angle and alignment. Students create an array of **DynamicText** objects, and output them using a **for-each** loop. Instructor demonstrates how to create a color-compatible background using text and a for-loop. Students use this code as a model to write a new program that will create a densely packed word cloud design using the most frequently occurring words in a student-chosen text passage.

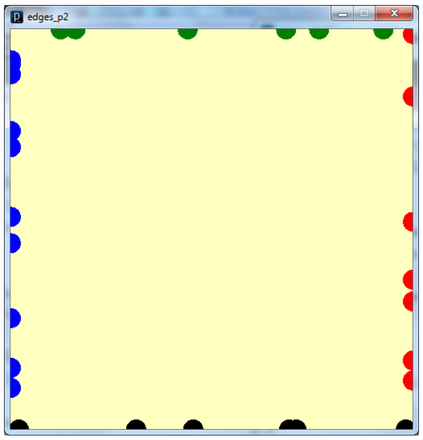
To add motion, the instructor gives students a helper **"*edges*"** program to discover how to write code that will place each word at a random starting location on any of the four window edges. With instructor guidance, students discover a linear equation model for synchronizing the starting and ending times of all words from their initial to final locations. Lastly, students modify the program so that it cycles and spends equal time between two states: (a) text objects moving from random positions on the edges to their final positions, and (b) text objects remaining at the final positions to allow time for appreciation of the final static design.

Key Assignments

1. After whole class instruction, students build a sample program that can output text of varying colors, size, font, rotational angle, alignment, and position.

2. Students build a program that outputs a static Word Cloud design.
3. Students modify their programs to output a dynamic Word Cloud where words appear at random positions on the window's 4 edges, then drift for 3-4 seconds to their final positions, where they come to rest for an equal period of time. Program cycles "forever" between these two states.
4. Using the techniques they learned in #3, students revisit the McClure painting program and modify it so that it will (a) alternate the direction of rotation, and (b) change background colors every time it begins to rotate in the opposite direction.

Teaching Strategies



Using the helper-program *Edges*, students examine two concepts: (a) randomly positioning (text) objects at the four edges of a window; and (b) mathematical variants for defining 4 random intervals, and their resulting constraints on programming style decisions.

Using whole class instruction, teacher guides students to discover what the x and y coordinates have to be if an object is to appear at any random position on the left edge: `x = 0; y = random(0,height);` Students sequester the code in a method called `leftEdge`(), then write similar the method bodies for `rightEdge`(), `topEdge`() and `bottomEdge`().

Instructor next guides students to discover 2 basic variants for defining 4 random intervals of equal size:

```
float n = random(0,4);
if (n < 1) { leftEdge(); }
else if (n < 2) { rightEdge(); }
else if (n < 3) { topEdge(); }
else { bottomEdge(); }

float n = random(0,4);
if (0 <= n && n < 1) { leftEdge(); }
else if (1 <= n && n < 2) { rightEdge();}
else if (2 <= n && n < 3) { topEdge(); }
else { bottomEdge(); }
```

Students are asked to consider the two code fragments for simplicity and clarity. They are then asked to swap lines, e.g. swap lines 2 and 3. Students discover that this has no effect on output for the second code fragment. However, in the first code fragment, no circles appear on the right edge, i.e. the rightEdge() method is never called. Students are asked to explain the phenomenon, and instructor illustrates the concept using (a) the number line, and (b) rearranging a sequence of filters/sieves with increasingly larger holes that are catching balls of various diameters, and so on.

To help explain saving/restoring of the drawing plane's **state** by **pushMatrix**()-**popMatrix**() – used by the program to allows text objects to rotate *independently* – instructor uses a camera metaphor, e.g. taking a snapshot of the drawing surface before any translation/rotation operations, performing the transformations, then restoring the prior state using the snapshot.

To derive expressions that allow the text objects to move (diagonally in most cases) from initial positions to final positions, instructor guides students to calculate a slope/intercept equation for both horizontal and vertical components of the motion. In this case, however, x and y are the dependent variables and *percent completion of motion* is the independent variable, with slope equal to the difference between final and starting coordinates, and the *y-intercept* equal to the starting coordinate. Instructor gives students hints by asking what the x-coordinate would be at 0%, 100%, 50%, 25% (in that order) and so on (Note: although we are calling the variable "*percent*" for ease of instructional discussion, in a strict sense, it is in fact the *fraction* of movement traveled). Students are thus guided to derive the equation for the x-coordinate (below). Once solved, students are directed to derive the expression for the y-coordinate using the same methodology.

```
float percent = this.timeCurrent / TOTAL_TIME;
float xCurrent =
  (this.xEnd - this.xStart) * percent + this.xStart;
```

To make the objects rest for an equal amount of time, instructor directs students to keep incrementing `timeCurrent` to twice the TOTAL_TIME before reverting back to zero. Students observe that this causes each text object to travel twice as far, specifically 100% beyond their final positions. The remedy is to simply cap *percent* at 100% for all values above 100%:

```
float percent = this.timeCurrent / TOTAL_TIME;
if (percent > 1.0) {
     percent = 1.0;
}
float xCurrent =
(this.xEnd - this.xStart) * percent + this.xStart;
```

Section 6. CodingBat: Boolean logic, Strings and Arrays
(PART II: BUILDING PROGRAMMING SKILLS)

<u>Description</u>

> *CodingBat is a free site of live coding problems to build coding skill in Java… The coding problems give immediate feedback, so it's an opportunity to practice and solidify understanding of the concepts. The problems could be used as homework, or for self-study practice, or in a lab, or as live lecture examples. The problems … have low overhead: short problem statements (like an exam) and immediate feedback in the browser.*
> ***- Codingbat.com/about.html***
>
> *… Implicit in this is a* [central] *CodingBat idea: don't add complexity by making a problem which is realistic or has a motivating back-story. Practice problems do not need to be realistic. Instead, you want the description to be short and clear, and you want to have lots of problems so the student can work lots of repetitions (like exercise at a gym), building skill and confidence.*
> ***- Codingbat.com/authoring.html***

At this point, students have had limited practice with most foundational programming concepts within a (hopefully) motivating dynamic art context. Although this should have given students a general framework for how computer programming can be applied, at this point, they need to begin to acquire basic programming competence. The intent is that students will not be learning disembodied skills, but rather will be learning to hone and expand their skill applying specific programming concepts they've already encountered within meaningful contexts.

Students now spend 6-8 weeks solving problems in 4 CodingBat modules: Logic-1, String-1, Array-1 and Array-2. Logic-1 covers Boolean variables, use of conditional statements (IF-ELSE, IF-ELSIF-ELSE, etc.), nested IF-ELSE statements, and common introductory logic problems. String-1 covers the use of the methods **length**, **substring**, **startsWith**, **endsWith**, **isEmpty**, **equals** and **equalsIgnoreCase**, as well as the logic of how to access string index positions from the start, end or middle of a string.

Array-1 covers simple problems in array creation, indexing and swapping of values. Array-2 covers iteration through the members of the array, touching upon operations such as: searching; determining aggregate values; locating specific subsequences; and comparing adjacent member items.

The purpose of the module is to give students real programming competence using basic programming control structures and concepts. Because there are both simple and sophisticated ways of writing code to solve these problems, instructor requires that students first use the simpler, clearer (and longer) coding styles until satisfied that students understand the underlying logic and programming mechanisms. Instructor then shows students how and why the more sophisticated (and shorter) coding styles are equivalent.

Because solutions to CodingBat problems are rife throughout cyberspace, students are only given credit when they pass 4 custom/teacher-written quizzes, one for each module. These custom quizzes have the same format and style as all other CodingBat problems, and are accessible from the teacher's individual CodingBat home page.

In addition, the solutions to some of the problems involve programming issues that will arise in later projects, e.g. circular buffers (a clock). Because the CodingBat website simply glosses over these issues, the unit devotes significant time to exploring different ways of thinking about how one might model and program such systems, and requires expository assignments where students must *clearly* define the problem and explain how to code the solution.

Key Assignments

1. Logic-1 Module and Custom Test
2. String-1 Module and Custom Test
3. Array-1 Module and Custom Test
4. Array-2 Module and Custom Test

Teaching Strategies

Solving the problems in CodingBat is a major hurdle for all students. There are many ways/styles to write code that will solve the problems, and the solutions provided in the *Warm-Up* and *Help* sections of the website do not provide the necessary scaffolding required for most high school freshmen. Therefore, the teacher uses direct instruction to help students understand how to solve the problems in the *Warm-Up* section. Although each problem involves some new aspect or issue, the problem below can be used to illustrate the teaching strategies used:

> *The parameter weekday is true if it is a weekday, and the parameter vacation is true if we are on vacation. We sleep in if it is not a weekday or we're on vacation. Return true if we sleep in.*

The instructor first demonstrates how to build a 2-D table that represents all 4 cases:

	vacation	**!vacation**
weekday	T	F
!weekday	T	T

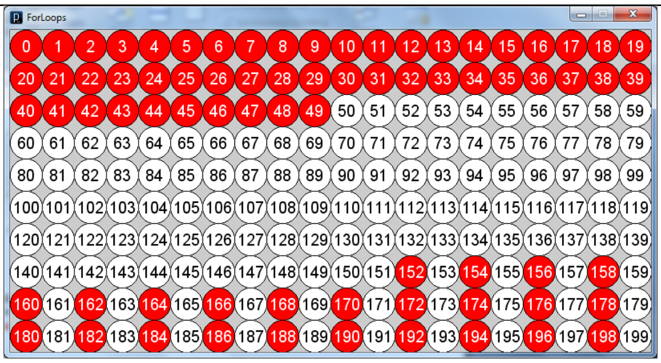
Instructor then shows several ways to code the solution:

```
public boolean sleepIn(boolean weekday, boolean vacation) {
       return !weekday || vacation;
}

public boolean sleepIn(boolean weekday, boolean vacation) {
       if (vacation) {
              return true;
       }
       else {
              return !weekday;
       }
}
```

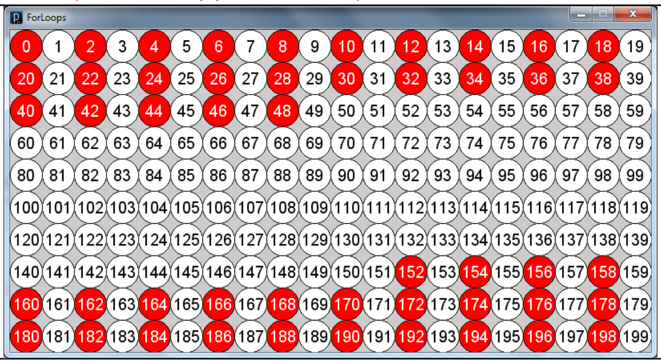
```
public boolean sleepIn(boolean weekday, boolean vacation) {
        if (vacation) {
                if (weekday) {
                        return true;
                }
                else {
                        return true;
                }
        }
        else {
                if (weekday) {
                        return false;
                }
                else {
                        return true;
                }
        }
}
```

For students who initially struggle, the last solution is longer, but simpler to understand because each of the 4 possible combinations is represented. Once students see the code working, the instructor can guide them to realize that the first **if (vacation)** statement can be simplified because it always returns true, i.e. weekday is irrelevant to the return value.



`i < 50 || i > 150 && i % 2 == 0`

`(i < 50 || i > 150) && i % 2 == 0`



Some problems in CodingBat can involve complex Boolean expressions which combine the && and || operators. The **200-circle matrix program** can show how && has higher precedence than ||. The expression

```
i < 50 || i > 150 && i % 2 == 0
```

yields the pattern above top, demonstrating that && binds tighter than ||, even though the expression is evaluated from left-to-right. When one adds parentheses to force the || to be evaluated first, as in

```
(i < 50 || i > 150) && i % 2 == 0
```

the pattern is as shown above bottom. The moral of the story is ALWAYS use parentheses to clarify the intention of the programmer.

Sometimes CodingBat problems inadvertently present common programming issues. The problem below is an example:

> *We have a loud talking parrot. The "hour" parameter is the current hour time in the range 0..23. We are in trouble if the parrot is talking and the hour is before 7 or after 20. Return true if we are in trouble.*

The Boolean expression for the time period between 8 pm and 7 am (non-inclusive), although written exactly as stated in the problem, is not intuitive because it spans the 0/24 boundary in the circular buffer representation of a clock. Normally a time interval between an earlier and later time is written using an AND expression, e.g. `6 <= hour && hour <= 12`, analogous to how one would write a range using an algebraic expression: `6 <= hour <= 12`. However, for intervals that span the boundary, the expression is nonsensical:

```
20 < hour && hour < 7
```

As illustration, the instructor uses Venn diagrams and the number line to demonstrate how a number cannot simultaneously be in two disjoint sets. One solution is to view the interval as the *union of two intervals* on either side of the boundary:

```
(20 < hour && hour < 24) || (0 <= hour && hour < 7)
```

However, because the range of times is limited to 0-23, there is no need for the conditions in red. The expression simplifies to:

```
20 < hour || hour < 7
```

Because *students will revisit such boundary problems in the cross-curricular units*, they are required to write clear, but short, answers to all 4 of the following questions (in general, students who cannot do so do not yet have the abstraction abilities necessary to succeed in a programming course).

1. What is the normal expression for a specific time period on a 24-hour clock?
   (e.g. between 12 and 18)

2. What is the problem with a time period that spans the 0/24 boundary?

3. What is the full expression for a time period that includes both sides of the boundary?
   Explain how you get this expression.

4. Why can you simplify this full expression by dropping the (0 <= hour) and
   the (hour < 24)?

Section 7. Nested For-Loops, Regular Patterns, and T-Tables

Description

This is a short unit that introduces students to **nested for-loops** and a methodology for solving specific kinds of problems where these loops are used. Thirteen problems from the exercises section at the end of **Chapter 2** in *Building Java Programs* were adapted for this unit. The problems involve using **nested** for-loops to produce regular patterns of lined text output. In order to solve the problems, students must use inductive reasoning to determine linear expressions that describe all lines in the output, using line number as the independent variable. Students proceed by creating a T-table describing the number of different categories of characters/numbers for each line. They then graph the categories against line number and determine the **slope** of the resulting line. Plugging in the **slope** and any single **point** into the **slope-intercept equation** allows one to calculate the **y-intercept**. One now has a **slope-intercept expression** for each category. These are used in the **conditional** part of the **inner** for-loops, and at times for the output character itself, if it's a number. Below is an example pattern, a T-table, and the nested for-loops where the derived expressions are used.

```
*|||||
**||||
***|||
****||
*****|
******
```

line	# *	# |
1	1	5
2	2	4
3	3	3
4	4	2
5	5	1
6	6	0
Algebraic expressions	**line**	**6 - line**

```
for (int line = 1; line <= 6; line++) { // outer loop
    // 1st inner loop prints asterisks
    for (int a = 0; a < line; a++) {
        out.printText("*");
    }
    // 2nd inner loop prints vertical bars
    for (int v = 0; v < 6 - line; v++) {
        out.printText ("|");
    }
    // after printing all the characters on the line,
    // go to the beginning of the next line
    out.printTextLine();
```

Key Assignments

13 Problems, including T-tables, graphs, Boolean expressions, and working code.

Teaching Strategies

Instructor uses whole-class direct instruction to go through the example described in the outline.

Instructor also has students reverse the process, i.e. trace through the code for several similar problems and show the output in both a T-table and console table for each iteration and output statement. An example of such a reverse problem is shown below.

```
for (int line = 1; line <= 5; line++) {

  for (int sp = 1; sp <= 5-line; sp++) {
    output.PrintText(" ");
  }

  for (int n = 1; n <= 2*line-1; n++) {
    output.PrintText(line);
  }

  output.PrintTextLine(); // Enter Key

}
```

Line	# spaces	# Number	Number
1			
2			
3			
4			
5			
Expression	5-line	2*line-1	line

Line 1									
Line 2									
Line 3									
Line 4									
Line 5									

Section 8. The Right to Vote
(PART III: INTERMEDIATE-LEVEL PROJECTS)

Essential Questions

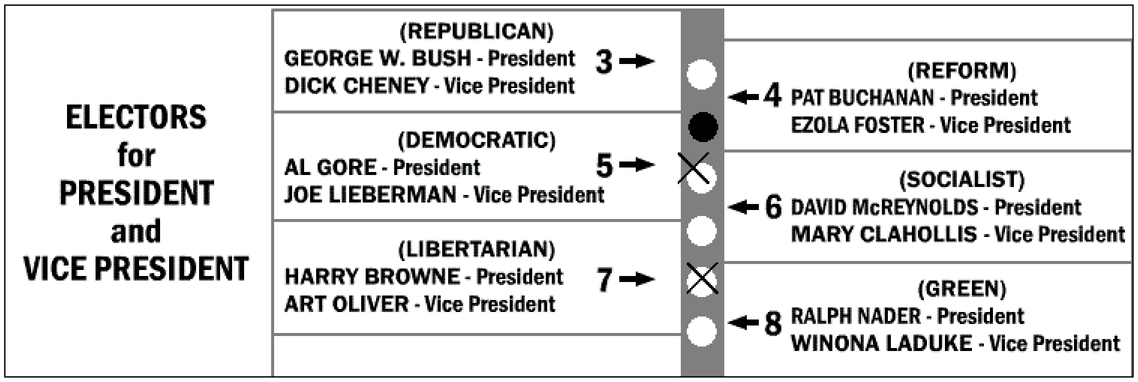
How do you write a program to **simulate** an election, both the **marking** and **counting** of thousands of ballots?
How is a program like this similar to software used to tally **optical scan ballots**?

Supporting Questions

Describe the **flood fill** algorithm?
Define **recursion**. What's the danger inherent in using recursion?
What is the difference between global and local variables?
How would one decide whether to use a global vs. a local variable?

Description

To give students background, they begin the unit by examining Palm Beach, Florida's "butterfly ballot" from the Nov. 7, 2000 presidential election. Students watch the film 2008 HBO film *Recount* about the electoral chaos in Florida which was resolved on Dec. 12, 2000 by the U.S. Supreme Court decision that gave the election to George Bush. Students also watch the 2004 HBO film *Iron-Jawed Angels* which tells the story of the suffragist Alice Paul in the 8 years preceding the passage of the 19th Amendment. Students write an essay about the film in response to the prompt described in Key Assignments.


ELECTORS for PRESIDENT and VICE PRESIDENT
(REPUBLICAN)
GEORGE W. BUSH - President
DICK CHENEY - Vice President
3
(DEMOCRATIC)
AL GORE - President
JOE LIEBERMAN - Vice President
5
(LIBERTARIAN)
HARRY BROWNE - President
ART OLIVER - Vice President
7
(REFORM)
4
PAT BUCHANAN - President
EZOLA FOSTER - Vice President
(SOCIALIST)
6
DAVID McREYNOLDS - President
MARY CLAHOLLIS - Vice President
(GREEN)
8
RALPH NADER - President
WINONA LADUKE - Vice President

At the start of the unit's programming section, students are instructed to figure out a way to mark the ballot above so that the entire white space within a circle is blackened. Students opt for what they know: using the **ellipse**() method to fill in the circle. However, because the border around the circles is pixilated, the rendering either leaves some pixels unmarked, or draws over gray pixels outside of the circle's boundary. At this point, Instructor introduces the **recursive *Flood Fill*** algorithm. Students reorder the recursive calls in the **floodFill**() method in a helper program that slows down the sequential filling in of the pixels; this allows students to see that the direction in which pixels are being drawn corresponds to the order of the recursive calls. Students use this information to write the body to a method named **markBallot**() that completely fills in a white circle for a single candidate. Students write a second method **markBallotX**() method that instead draws an X centered on a random location in the white circle, and consider what criteria should be used to determine the voter's intent, i.e. how many pixels in the white circle need to change color.

Students implement an **Election** class that marks and counts ballots. The method **markBallots**() creates thousands of ballots (**Ballot** class objects) and uses the (revisited) **random**() method to set the parameters for the percentages of the ballots that will be marked for each candidate. They will also use **random**() to mark a certain percentage of the ballots in some invalid way, such as for more than one candidate, or for no clear candidate choice.

Students then implement the **tallyElection**() method, which iterates through the ballot utilizing a **countBallot**() method that determines which candidate the voter selected. The **countBallot**() method in turn must employ a **readBallot**() method that uses (revisited) **nested for-loops** to iterate through rectangular regions of the ballot image's pixels. They consider three algorithms for dealing with invalid ballots when implementing **countBallot**(): (a) Increment global variables for each candidate as marks are encountered. When a ballot marked for two or more candidates is determined to be invalid after it has already incremented their candidates' totals, it will be read a second time to decrement and correct those same variables. (b) Examine a ballot first to determine if it is valid. If so, read it a second time. (c) Read a ballot only once, but use local variables in the **countBallot**() method to first collect counts for all candidates. If the ballot is valid, use the local variable to increment the corresponding global variable.

Key Assignments

1. Students write an essay about the two films *Recount* and *Iron-Jawed Angels*, in response to the prompt: *The protagonists in both films used many strategies in their efforts to reach their goals, both of which concerned voting rights. Describe key strategies that each group used in order to try to attain their goal and how successful each of those strategies was*.
2. Students implement a strategy of their own design to **mark a ballot** for a candidate.
3. Students implement the **flood-fill** algorithm for fully marking a circle.
4. Students implement a method for **marking** the circle with an **X**.
5. Students implement an **Election** class with a method that returns an array of thousands of ballots marked with specified **percentages** for various candidates.
6. Students implement a certain percentage of **invalidly** marked ballots.

7. Students implement the **readBallot**() method that can determine the candidate(s) marked on a ballot.
8. Students implement a **countBallot**() method that throws out invalidly marked ballots and correctly increments the tally for the candidate marked.
9. Students implement the **tallyElection**() method that counts all the votes and reports the election results.

Teaching Strategies

Counterexamples, guided discovery, revisiting concepts and experimentation.

When presenting the **flood-fill** algorithm, a helper-program is distributed to help give students an intuitive feel for **recursion's** depth-first approach. The program performs the **floodFill**() method, but instead of drawing the pixels in real time, saves them in an array for drawing them slowly so that students can observe the sequence. The multiply-recursive method floodFill() appears as follows:

```
void floodFill(int x, int y, color targetClr) {
  MyPoint pt = new MyPoint(x, y);
  color clr = get(x, y);
  if (clr == targetClr && !contains(pts, pt)) {
    pts.add(pt);
    floodFill(x, y-1, targetClr);   // 1
    floodFill(x, y+1, targetClr);   // 2
    floodFill(x-1, y, targetClr);   // 3
    floodFill(x+1, y, targetClr);   // 4
  }
}
```

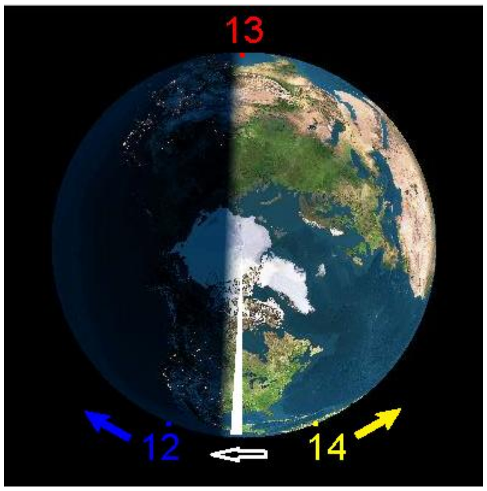
As students experiment with rearranging the numbered lines, they can observe the different directions in which the pixels are drawn.

The **readBallot**() method that examines a ballot's pixels **revisits**/reuses the **nested for-loop** iterative control structure that students learned in the previous unit. The method looks at the circle next to each candidate and compares **white** pixels on the unmarked ballot with corresponding pixels on the marked ballot – if the corresponding pixel is a different color, the voter marked that candidate:

```
// returns candidate voted for (3-8) or 0 if invalid ballot
// x, y is the top left corner of a square bounding the
circle
// wh is the number of pixel rows/cols to process
boolean readBallot(int candidate) {
  int x = 453;
  int y = 34 * candidate - 68;
  int wh = 25;
  for (int row = x; row <= x + wh; row++) {
    for (int col = x; col <= y + wh; col++) {
      color clrU = img.get(col, row);   // Unmarked Ballot
      color clrM = get(col, row); // Marked Ballot (screen)
      if (clrU == color(255) && clrM != clrU) {
        return true;
      }
    }
  }
  return false;
}
```

Instructor should point out the similarity of the structure of this nested for-loop to the problems from Section 7. Instructor then should make the point that in both cases, the code is processing a **2-D space** / **matrix** row-by-row, from top-left to bottom-right.

Section 9. Around the World in 24 Days



Essential Questions
How can software be used to study and solve problems in **Human Geography**?

Supporting Questions
Why is there a need for the **International Dateline**?
How can you use the **sin** and **cos** functions to position objects around the edges of a circle?
Why does one need to treat the area around a **circle's 0°/360° boundary** differently from the rest of the circle?
How can one write a program that uses **the same code** to handle both a stationary and rotating Earth?
What types of discrepancies arise when one tries to use a **discrete model** to represent a **continuous system**? What are some mechanisms to deal with these?

Description

Students build a simulation of a rotating Earth in order to model the phenomenon described in Jules Verne's *Around the World in 80 Days* of an east-bound traveler who circumnavigates the world and experiences one day more than an observer remaining at the starting point. Three observers are placed on the surface, two of whom circumnavigate the globe in opposite directions: (a) an East-bound traveler (yellow); (b) a West-bound traveler (red); and (c) a stationary observer (white). After 24 days, the two travelers return to the starting point to rejoin the stationary observer. The east-bound traveler will have seen 25 sunrises, and the west-bound traveler will have seen 23 sunrises. The simulation sheds light on the reason for the establishment of the International Date Line.

Students begin by downloading 96 satellite images of Earth using the **View from Earth** website (*http://www.fourmilab.ch/cgi-bin/Earth*). These represent snapshots taken over a 24-hour period spaced at 15-minute intervals. So that the surface of the Earth is half in shadow, the date chosen is either the Spring or Autumnal Equinox with Latitude = 90°N as if the satellite is positioned over the North Pole. Longitude is arbitrary, but 72°E was chosen so that Los Angeles is at the top of the simulation.

Students load the images into an array and implement the animation using a circular queue, which displays a stationary Earth with a moving *terminator* (the boundary line separating day and night). Students are already familiar with Processing's 2-D transformation operations from the Word Cloud and McClure units. They similarly implement rotation by translating the coordinate system origin to the center of the window, performing the rotation, and translating the origin back to the top left corner. As in the Word Cloud unit, students again bracket transformations between **pushMatrix()** and **popMatrix()** to independently rotate several objects simultaneously. The final rotation effect is that the Earth rotates and the **terminator** is stationary. A toggle variable controls rotation.

Earth, **Sunrise** and **Traveler** classes are implemented. A conditional expression for enabling a stationary traveler to detect a sunrise is initially expressed using normalized degree measurements to keep traveler and sunrise angles within the same range. The conditional expression is modified as more cases are accommodated, culminating with a solution for the edge condition at 0°/360°. The final expression implements a **sector-point** intersection model.

Movement for travelers is implemented using a **speed** instance variable which is positive for traveling west; negative for traveling east; or 0 for no movement. Students discover that sunrise detection breaks down for moving travelers: at some point during their circumnavigations - depending upon starting values for sunrise and traveler - the East traveler misses a sunrise and the West traveler clocks a double sunrise. An analogy to an escaping prisoner avoiding detection by a moving flashing searchlight is made. The problem is solved by narrowing or expanding the **sector** by the traveler's speed, and students consider the side-effects that occur when representing a **continuous** system with a **discrete** model.

Key Assignments

1. Students download 96 images of Earth and create an animation showing a complete 24-hour light cycle. The **animation** is that of a stationary Earth and a moving solar terminator.
2. Students implement an **option** for a stationary terminator and a rotating Earth.
3. Students implement an **Earth class** that does the bookkeeping involved in tracking and incrementing its angular position.
4. Students implement a **Traveler class** – although the first Traveler object created is stationary. Like the Earth, the Traveler keeps track of it angular position, and, when the earth rotates, changes its angular position at an identical rate to maintain the same location on the Earth. A Traveler object displays as a **number**, indicating the number of sunrises it has "seen".
5. Students implement a **Sunrise** class to keep track of the terminator's angular position.
6. Students implement a **normalize**() method that keeps all angles in the range 0 <= angle < 360.

7. Students implement **Traveler** methods **seeSunrise**() and **incSunrises**(). The **seeSunrise**() method revisits the boundary problem students first encountered in CodingBat's **parrotTrouble** problem.
8. Students implement **traveling** for a Traveler object, using a **speed** variable.
9. Students **correct** the **seeSunrise**() method to account for the longer or shorter sector width needed to detect a sunrise when a Traveler object moves east or west, respectively. The length of the sector's arc is adjusted by the Traveler object's **speed**.

Teaching Strategies

Modeling, counterexamples, guided discovery and experimentation.

To familiarize students with animation concepts, at the start of the unit, students write a program that animates Eadweard Muybridge's galloping horse photographs. The animation shows that all four feet of a horse are simultaneously off the ground at one point during a gallop cycle. Instructor stresses that this phenomenon is easier to grasp by seeing it in context within an animation rather than as a single still photograph.

Students revisit the edge/boundary problem they encountered when crafting the conditional expressions for CodingBat's **parrotTrouble** problem.

When implementing display() for the Traveler object, students are instructed to keep the number at the same position "height" above the Earth no matter the Traveler's angular position. This involves explicitly calculating the left and top coordinates for the text() method (mathematically centering the number about its (x, y) position), then using the API's **textAlign**() method. When students asked why they had to bother calculating the left and top coordinates, only to comment out the code, the instructor tells them that these are the same calculations the textAlign() method makes to center text horizontally and vertically.

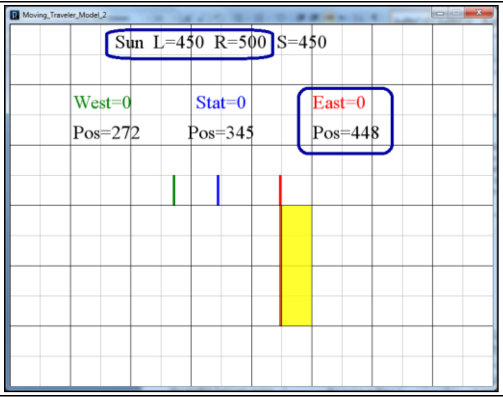
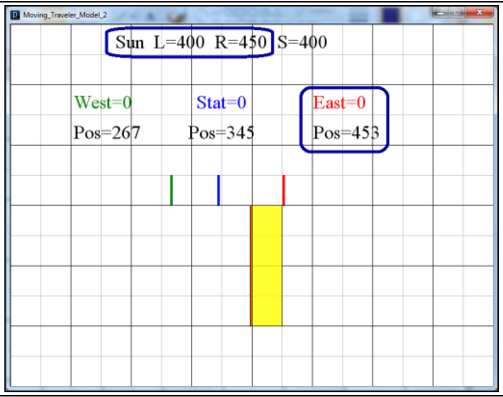
 Moving_Traveler_Model_2
Sun L=450 R=500 S=450
West=0
Pos=272
Stat=0
Pos=345
East=0
Pos=448
	 Moving_Traveler_Model_2
	Sun L=400 R=450 S=400
	West=0
	Pos=267
	Stat=0
	Pos=345
	East=0
	Pos=453

Figure 15A. East bound traveler is just to the west of the sector at Time 1.	*Figure 15B*. East has moved just to the east of the next sector at Time 2. No sunrise is detected!

To *model* the side-effect of using a **discrete model** with a **constant sector width** for traveling objects, the instructor distributes a *Moving Traveler* program that illustrates how, if the coordinates are inauspicious, an east-bound traveler can miss detection of a sunrise (**Figure 15**) and how a west-bound traveler can detect the same sunrise twice (**Figure 16**). The program shows the overlap between sector and traveler and records a sunrise event at these junctions.

The program also allows the user to widen or narrow the sector in order to illustrate how this adjustment can correct the detection errors (**Figures 17 and 18**). Note that the adjustment to the sector width occurs at the tail end (eastern edge), and the amount of the adjustment is the **distance** the traveler moves during an interval.

To assess understanding, students are instructed to write two paragraphs describing how each of the two errors occur and how each is corrected by an appropriate sector width adjustment.

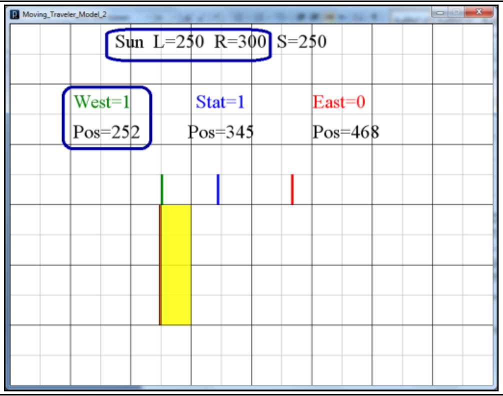
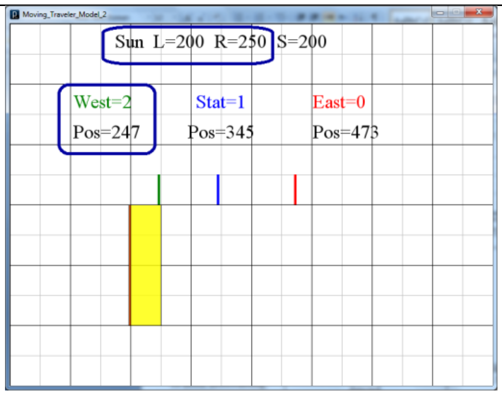
 Moving_Traveler_Model_2
Sun L=250 R=300 S=250
West=1 Pos=252
Stat=1 Pos=345
East=0 Pos=468
	 Moving_Traveler_Model_2
	Sun L=200 R=250 S=200
	West=2 Pos=247
	Stat=1 Pos=345
	East=0 Pos=473

Figure 16A. West bound traveler is at the west edge of the sector at Time 1. A sunrise event is detected.	*Figure 16B*. West has moved to the east edge of the <u>next</u> sector at Time 2. A 2nd sunrise event is detected!

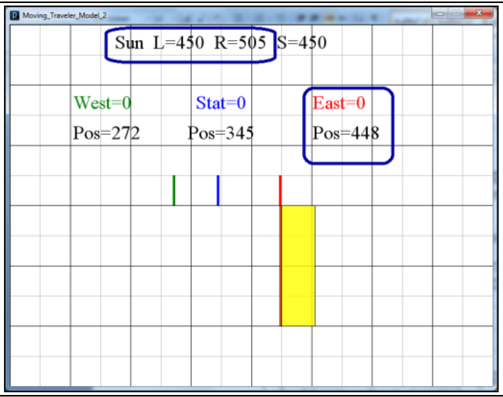
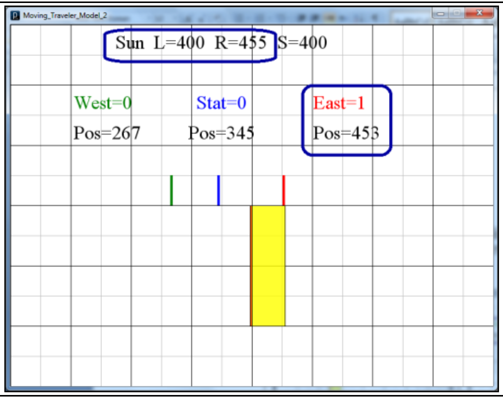
 Moving_Traveler_Model_2
Sun L=450 R=505 S=450
West=0 Pos=272
Stat=0 Pos=345
East=0 Pos=448
	 Moving_Traveler_Model_2
	Sun L=400 R=455 S=400
	West=0 Pos=267
	Stat=0 Pos=345
	East=1 Pos=453

Figure 17A. East bound traveler is to the west of the <u>widened</u> sector at Time 1. No change in behavior.	*Figure 17B*. East is within the <u>widened</u> sector at its east edge at Time 2. A sunrise event is now detected!

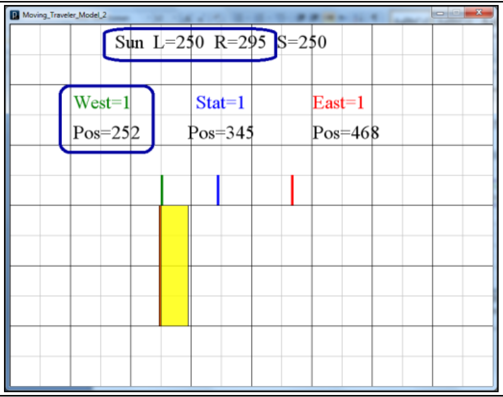
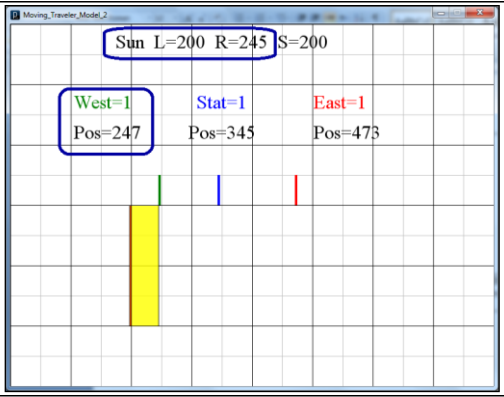
 Moving_Traveler_Model_2
Sun L=250 R=295 S=250
West=1 Pos=252
Stat=1 Pos=345
East=1 Pos=468
	 Moving_Traveler_Model_2
	Sun L=200 R=245 S=200
	West=1 Pos=247
	Stat=1 Pos=345
	East=1 Pos=473

Figure 18A. West bound traveler is at the west edge of the <u>narrowed</u> sector at Time 1. A sunrise event is detected.	*Figure 18B*. West is now just outside the east edge of the next sector at Time 2, because the sector has been narrowed at this end. No 2nd sunrise event is detected!

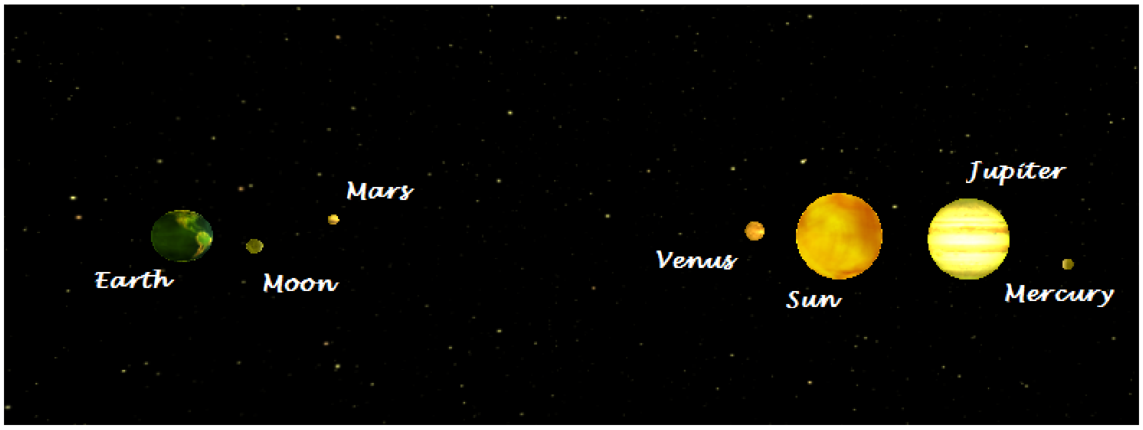
Section 10. Galileo's Revolution and Astronomy

<u>Essential Questions</u>
How can a **software model of the Solar System** help us understand astronomical phenomena such as the phases of Venus, retrograde motion of Mars, and the infrequency of solar eclipses?

<u>Supporting Questions</u>
Why was the **discovery of the phases of Venus** so controversial *<u>and</u>* so significant historically?
How does **texture mapping** work?
What are the math functions needed to describe an **elliptical orbit** in 2D?
What adjustment does one need to make to add a 3rd dimensional vertical movement to an orbit to implement **ecliptic tilt**?
How do you **position** the **camera** to view simulations of astronomical events?

<u>Description</u>

To provide background, the instructor guides students through the use of planetarium software, such as the proprietary Orion *Starry Night* or a free downloadable equivalent. Viewing the skies from any arbitrary location on Earth (such as one's home city) and using time-lapse settings, students view close-ups of Mercury and Venus as they go through their Moon-like phases, and observe Mars in retrograde over a period of roughly six months every two years.



Students then read and discuss Bertolt Brecht's play *Life of Galileo,* and write an essay in response to several possible prompts described in Key Assignments. Major themes are (a) faith vs. doubt; (b) integrity vs. personal ambition; and (c) societal responsibility.

Students build a Copernican/heliocentric simulation of the inner solar system planets plus Jupiter, in order to view astronomical observations from various perspectives. The simulation replicates the **phases of Venus**, **Mars in retrograde**, and the **infrequency of solar eclipses**, both partial and total. Students create a **3-D** simulation and use **transformations** in 3 dimensions to position spherical objects representing the sun, planets and moon. Although the simulation cannot be to scale – because the planets would be too small to see – **distances** from the sun and planet **sizes** are consistent relative to one another. For similar reasons, elliptical **orbits** are approximated as circles. **Orbital tilts** for each planet and the moon are implemented relative to the ecliptic plane. The add-on *Processing* library **shapes3d** is used to add **texture mapped skin images** to the surface of the planetary spheres and to position the entire simulation within a surrounding static sphere whose skin is a **star map**. Students write methods to position and aim the **camera** in any direction in order to view the model not only from Earth, the planets and the sun, but from lateral and overhead views of the solar system. Students write methods to move the camera through a 3-D space. Finally, students write a second program to simulate the **Ptolemaic**/geocentric solar system model in order to understand what phenomena the model did and did not account for, and so help explain its nearly 2,000 year longevity.

Key Assignments

1. Students write an essay about the play *Life of Galileo* in response to several possible prompts:

a. A recurrent theme in the play is FAITH vs. DOUBT. In Scene 1, Brecht discusses this as it relates to science. In subsequent scenes, the references to faith/doubt are related to religion and the political/social order (the nobility ruling over the peasants). Discuss Brecht's ideas about faith and doubt as they come up during the course of the play.

b. Galileo uses his intelligence for three things: (a) trying to find a way to live well and be comfortable, (b) searching for scientific truth, and (c) trying to stay alive and out of trouble with the authorities. Discuss Galileo's use of cunning (shrewdness, cleverness, deception) to do all three as he negotiates the demands of the various authorities from the church and state (including the university and the city) that oppose him.

c. In Scene 7, the Little Monk argues that scientific truth should be abandoned for the sake of the peasants. Discuss his rationale (reasoning/reasons) for this stance and Galileo's vigorous response. Discuss the connection between this scene and the 2 lines at the end of Scene 12:

> *Unhappy is the land that has no hero.*
> *Unhappy is the land that needs a hero.*

2. Students write a 3-D program that places a yellow sphere (**sun**) in the center of the window using transformations.
3. Students use 3-D transformations to implement a **planet** revolving around the sun in a circular orbit.
4. Students implement the 4 terrestrial planets **Mercury**, **Venus**, **Earth** and **Mars**, and the gas giant **Jupiter** using diameters and orbital radii relative to each other. The

planets will be many times larger than scale, and all planets will orbit in the same ecliptic plane.

5. Students implement **inclinations** for all planets except Earth, i.e. their orbits are tilted relative to the ecliptic plane.
6. Students implement the Earth's **Moon** revolving around the Earth (diameter relative to Earth, but orbital radius not).
7. Students implement **rotation** of planets, sun and moon about their axes.
8. Students implement Earth's **axial tilt** (23°).
9. Students collect texture mapped images of the planets, sun and moon, and implement **texture mapping** so that they are drawn with realistic looking surfaces.
10. Students collect an appropriate texture mapped image of the **stars** to wrap on a super-large sphere that encloses the entire simulation.
11. Students write methods to position the **camera above** the ecliptic plane and to its **frontal side**.
12. Students implement methods to **position** the camera on each of the **moving planets** pointing to the sun.
13. Students implement a method to view **Mars** from Earth.
14. Students implement a **second** method to view Mars from Earth, but **fixing** the camera on a point far beyond Mars in order to view Mars' motion in retrograde.
15. Students implement methods to control the **speed** and **direction** of the simulation so that the phenomena of solar eclipses can be viewed.
16. Students implement methods for **moving** the **camera** through space, rotating left and right, up and down, and moving forwards and backwards.

Teaching Strategies

Counterexamples, guided discovery and experimentation.

The heliocentric (Copernican) model of the solar system allows students to view the complete cycle of Venus's phases as seen from Earth. The companion program that simulates the geocentric (Ptolemaic) epicycle model demonstrates the impossibility of observing both a completely dark and fully lit Venus. When students subsequently read Bertolt Brecht's play *Life of Galileo*, they learn that it was this single celestial observation – made possible by the newly invented telescope – that was a pivotal point in the further erosion of papal power, already weakened by the Reformation. Although Galileo himself was put under house arrest for the remainder of his life, his discoveries further loosened the Church's capacity to impede the pace of science during the Renaissance.

On the pedagogic level, this unit taken in its entirety extends and clarifies students' understanding of events 400 years old. It does so through the use of student-written software that is able to clarify the true nature of a celestial phenomenon, one whose logical implications had huge historic, social and political ramifications. Because of its many facets, the unit contains multiple points at which students can make engaging connections.

If placing problems in real-world contexts answers the question "How can I use this knowledge?", providing historical and social contexts allows students to ask "What is the human/societal impact?" At this point, students step back, gain perspective and look at the big picture to observe how their work can be used and misused. This unit brings up powerful ethical questions for scientists and engineers – such as the ethical role of the scientist – that dwarf such standard fare concerns as intellectual property.

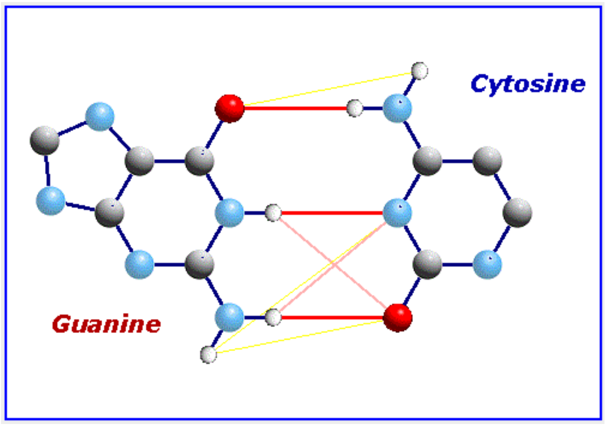
For implementing the movement of the camera to view the simulation from different perspectives, students use "dummy" **Planet** class variables named **eyePlanet** and **centerPlanet** to hold the position and direction of the camera, respectively. In this way, students are introduced to the use of <u>**references**</u> variables that can hold values referring to objects. Implementing a way to observe Mars in retrograde from Earth is done in two ways. One is to simply assign Earth to **eyePlanet** and Mars to **centerPlanet**, the effect of which is to keep Mars statically fixed in the center of the viewport, while the star background moves in reverse direction relative to Mars. The second is to point the camera at Mars, then project a **vector** far beyond Mars (say 100 times the Earth-Mars distance), and capture that position to store in **centerPlanet**. The effect of this second strategy is to fix the camera direction on a static star background, allowing one to observe Mars move and reverse directions. The calculation of this distant position is done with **vectors** (see below).

To implement keystroke-driven camera movement for **left** and **right rotation**, students first consider rotation around an axis parallel to the y-coordinate axis (students will later implement rotation around an axis in any direction using matrices). Students are introduced to the *arctan* function to derive the angle that the camera vector projects onto the X-Z coordinate plane. Students learn, however, that the angles returned from this function are **doubly ambiguous** because tangent values are the same for quadrant pairs I and III, and II and IV. Therefore in order to map the correct angle, students learn that they need to also utilize the <u>signs</u> of the **cosine** and **sine** values in the calculation. Students use an **Excel** spreadsheet to quickly see that the **arctan** returns values in the range $-\pi/2$ to $\pi/2$ (-90° to 90°). They then expand this spreadsheet to show corresponding **sine**, **cosine** and **tangent** values for all 360º, and the (differing) angle values returned by **arctan**. Using the data from each quadrant, students write a camera method that correctly maps degrees based on **arctan**, **cosine** and **sine** values as angular position is incremented or decremented.

To implement keystroke-driven camera movement **forwards** and **backwards**, students need to project component vectors onto all three axes, the same technique used to calculate a distant point when viewing the retrograde motion of Mars. Students will recognize that calculating the magnitude of these component vectors is similar to calculating the slope between two points. To calculate the slope between any two points, students know that it doesn't matter which point is subtracted from the other as long as one is consistent, i.e. $\Delta x = x^2 - x^1$ and $\Delta y = y^2 - y^1$ OR $\Delta x = x^1 - x^2$ and $\Delta y = y^1 - y^2$. This is because the signs of the two differences cancel each other when the two are positioned as a ratio: $\Delta y / \Delta x$. For example, a line defined by the origin and a point in quadrants I or III yields two positive differences or two negative differences, resulting in a positive slope. A line similarly defined in quadrants II an IV yield one positive and one negative difference, resulting in a negative slope.

With the component vectors, however, the order in which one subtracts the points is critical because (a) we are calculating the effective contribution of each vector separately, and (b) a vector has not only magnitude but direction. The instructor therefore has students calculate the extrapolated point using both possible orientations, so that students discover that the correct order is **destination** value minus **source** value – or for the camera variables, **centerX** minus **eyeX**, etc. Students are asked to write a short paragraph explaining how to do these calculations.

Section 11. Molecular Modeling and DNA



Essential Questions

How can we use Molecular Modeling software to explain how the opposing strands of a DNA double helix are **structurally** held together?
How can Molecular Modeling software help explain how the hydrogen bonds between the DNA bases provide the mechanism by which **genes** are faithfully **replicated**?
How can we use Molecular Modeling software to help us understand how **point mutations** arise?

Supporting Questions

What kinds of social and emotional obstacles might hinder **scientific collaboration**?
Which **geometry concepts** are needed to draw the DNA bases?
How can you use **sine** and **cosine** functions to determine the positions of atoms at irregular angle positions?
What are the advantages/disadvantages in keeping many separate, but corresponding, lists of properties rather than a list of objects, each of which contains all the properties?
What are the advantages in using **superclasses** and **subclasses**?
How can the **additive trigonometry identities** be used to implement **rotation**?
What are advantages/disadvantages to **implementing** translation and rotation using **math**, as opposed to the **transformation methods**?
Is there a way to implement **reflection** using Processing's transformation methods?
How do you program the GUI for **selection** and **movement** of objects using the **mouse**?
How do you implement **multiple selection**?

Description

Students build a 2-D molecular modeling program to examine the hydrogen bonding, between purine and pyrimidine bases, that holds the two anti-parallel strands of the DNA double helix together.

The unit begins with students familiarizing themselves with the freely available 3-D molecular modeling program **MolSoft ICM- Browser**, and exploring ways to configure the four DNA bases Adenosine, Guanine, Cytosine and Thymine within the program. The molecule files will be downloaded from the **NYU Library of 3-D**

Molecular Structures. Students then proceed to calculate the angles of the pyrimidine (hexagon) and imidazole (pentagon) rings and use **BYOB** (UC Berkeley) to correctly position each base's ring and functional group atoms. During this process, students design their code to reflect the biochemical nomenclature of the molecule's major features. They also abstract shared features of the molecules into common methods for building pyrimidine rings, imidazole rings, and adding functional groups at any of the 6 pyrimidine atoms.

Once they are familiar with the structure of the 4 bases, students go about building the 2-D molecular modeling program in **Processing**. They use the **sine** and **cosine** functions to create a method to position atoms using polar coordinates. They use getter methods to encapsulate the x- and y-coordinates of each atom. These methods become the central repository for calculating translated, rotated, and mirrored coordinates for each atom. Students use a geometry proof to find the additive angle formulas for sine and cosine. These are used to derive the rotation formulas for x- and y-coordinates, which are then implemented in the program.

Students study the chemistry of polar bonds and hydrogen bonds. They write methods for deciding which hydrogen atoms are **electropositive** and which nitrogen and oxygen atoms are **electronegative**. The optimal distance for intermolecular hydrogen bonds is indicated by color and line thickness. Students program the GUI for object selection with mouse, control key and lassoing. Move, rotate, and mirror-image actions are driven by mouse events.

At unit's end students ***use their programs*** to display normal A-T and G-C pairings. They also perform **predictive** tasks, i.e. find configurations for rare A-C and G-T pairings, which represent point mutation situations.

To anchor this project in a social setting, students study the **BBC** film ***Double Helix***, which relates the little known story of the discovery of the DNA double helix by Watson and Crick, who used the x-ray diffraction data of the biophysicist Rosalind Franklin for building their model. Although her data was crucial to their calculations, which won them the Nobel Prize, they did not acknowledge her contribution until long after her death. Students write an essay about the film in response to the prompt described in Key Assignments.

Key Assignments

1. Lesson 1: Students study the **geometry** of **hexagons** and **pentagons** in order to write a BYOB program that correctly draws the angles and bond lengths for the 4 DNA base *in pyrimidine and purine ring nomenclature atom order*.
2. Lesson 2: Students organize and simplify their BYOB programs by using iteration and sequestering duplicate code into parameterized methods that have meaningful biological and/or chemical significance, e.g. **drawPyrimidine**(), **drawImidazole**(), **addAmine**().
3. Lesson 3: Students write a *Processing* program – organized along the principles learned in the BYOB lessons – that correctly displays classes for the 4 DNA bases at the origin (translated to the center of the drawing window).
4. Lesson 4: Students modify their *Processing* program to implement polymorphism, using the common parent class Molecule.

5. Lesson 4: Students implement **translation**, so that the user can move/reposition the DNA bases. Students program the GUI to use the **Arrow** keys for gross movement, and the **Ctrl + Arrow key combinations** for finer movement.
6. Lesson 4: Students complete a **geometric proof** for the **Additive Trigonometric Identities**, and use these to derive expressions for implementing 2-D **rotation**.
7. Lesson 4: Students implement **rotation**. The **PageDown** and **PageUp** keys are used for rotation clockwise and counter-clockwise, respectively, with the option of using them in combination with the **Control key** for finer movement.
8. Lesson 4: Students implement **reflection**, so that users can flip molecules horizontally. The 'M' key (for "mirror") will toggle the state of the reflected molecule.
9. Lesson 5: Students implement **mouse events** in the GUI. Students implement single and multiple mouse **selection** using (a) **clicking**; (b) clicking in combination with the **Shift** and **Control** keys; and (c) **lassoing**. They implement the **mouse wheel** for rotation. They implement **drag-and-drop**.
10. Lesson 6: Students implement **electropositive** and **electronegative** chemical properties into the bases, so that they can form – and display – **intermolecular hydrogen bonds**.
11. Lesson 6: Students use the program to show the normal hydrogen bonds between A-T and G-C, and to discover at least one configuration each for point-mutation-causing hydrogen bonds between A-C and G-T.
12. Lesson 7: Students write an essay about the film *Double Helix* (a.k.a. *Life Story*) in response to the prompt:

The film *Double Helix* takes place in the years 1951-1953. At the beginning of the story, Dr. Rosalind Franklin returns from Paris to London to take a research position at King's College. There, as one of the few women researchers, she experiences first-hand the effects of a work place imbued with sexist attitudes and where men have traditionally been in charge. One of the effects of this suffocating environment is that she feels isolated in her work.

a) Think of scenes where Franklin feels, or is in fact, isolated or marginalized (not treated seriously or equally). Contrast these scenes with those where Franklin finds ways out of her isolation to form connections with other supportive characters.

(b) In the film, Franklin's character resists a system that puts her at a disadvantage. Think of scenes where these she pushes back and how effective her efforts are in terms of successfully getting the changes (in behavior, or legally) that she might have wanted. Analyze these actions/efforts and discuss reasons why they might have been either effective or ineffective.

(c) At the end of the film, Bragg tells Franklin: "This race, this winning and losing, it's not the way I was taught to do science." This remark speaks to the main characters' motivations for doing science. Compare and contrast the motivations of Crick, Watson, Franklin and Wilkins. Remember that the characters' motivations refer not just to abstract issues about the pursuit of science, but equally – if not more – about what they enjoy about their day-to-day work

Teaching Strategies

Counterexamples, guided discovery, experimentation and **CONNECTIONS**.

One central strategy is the use of cross-curricular concepts, especially geometry and biology, to write a program that has both descriptive and predictive value vis-à-vis the bonding between the anti-parallel strands of DNA. Students must connect concepts from other disciplines in order to write an accurate 2-D molecular modeling program. Students will also see that CS is an engineering discipline, one that can be used to solve problems in other academic fields.

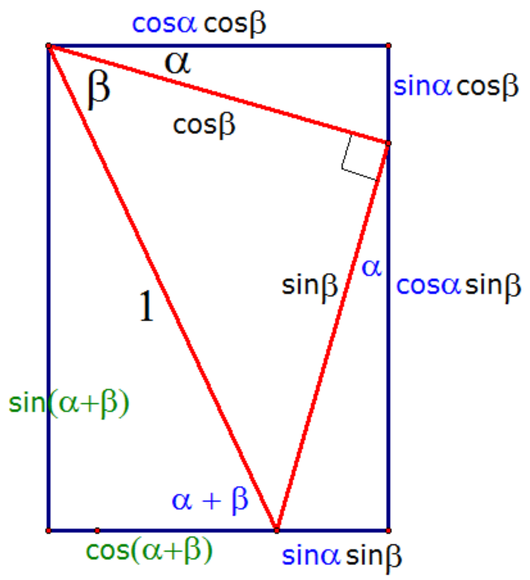


Figure 19. Derivation of Additive Trigonometric Identities

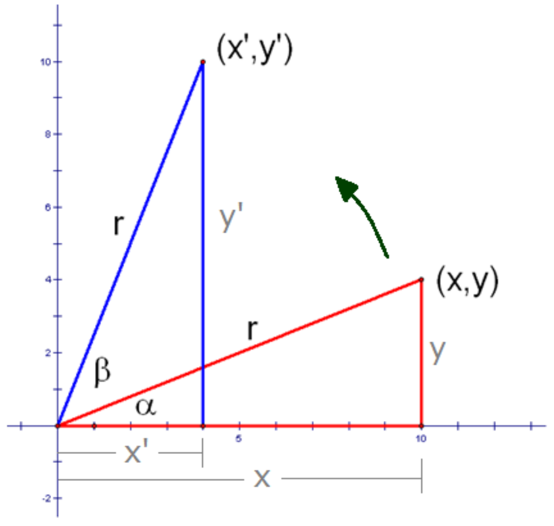


Figure 20. Derivation of Formulas for Calculating New Coordinates after Rotation about the Origin

One powerful example of making ***CONNECTIONS*** to other disciplines is the section where students implement molecule rotation. Changing the coordinates of a molecule's atom is an alternate means for rotating objects as opposed to using *Processing's* transformations. To do so, students learn – through guided discovery – a ***geometric proof*** for the Additive Trigonometric Identities, **sin**(α + β) and **cos**(α + β). Because the prerequisite for the course is proficiency in Algebra 1, students will have already taken concurrently the lion's share of a Geometry course – or an Algebra 2 course – by the time this final unit is encountered near the end of the school year. Moreover, in the *Around the World* unit, students have already been learned about and used the **sin** and **cos** functions. The proof involves nothing more than using the definitions of **sin** and **cos** on a specific right triangle, equating opposite sides, then isolating **sin**(α + β) and **cos**(α + β) (***Figure 19***). Once students complete the proof, they **<u>use</u>** the identities in another geometrically constructed diagram that sets up the theory for rotating a single point about the origin (***Figure 20***). Students can now derive the formulas for the x- and y-coordinates (x_2, y_2) for a point (x_1,y_1) that is rotated about the origin by substituting in the additive trig identities, then substituting in the definitions of **sin** and **cos** (**$x' = x\cos\beta - y\sin\beta$** and **$y' = y\cos\beta + x\sin\beta$**). Finally, students implement these simple formulas in their program and visually confirm that the molecules they've constructed rotate when the programs are run. Students are thus able to see that theoretical math does indeed have concrete applications.