Word Clouds

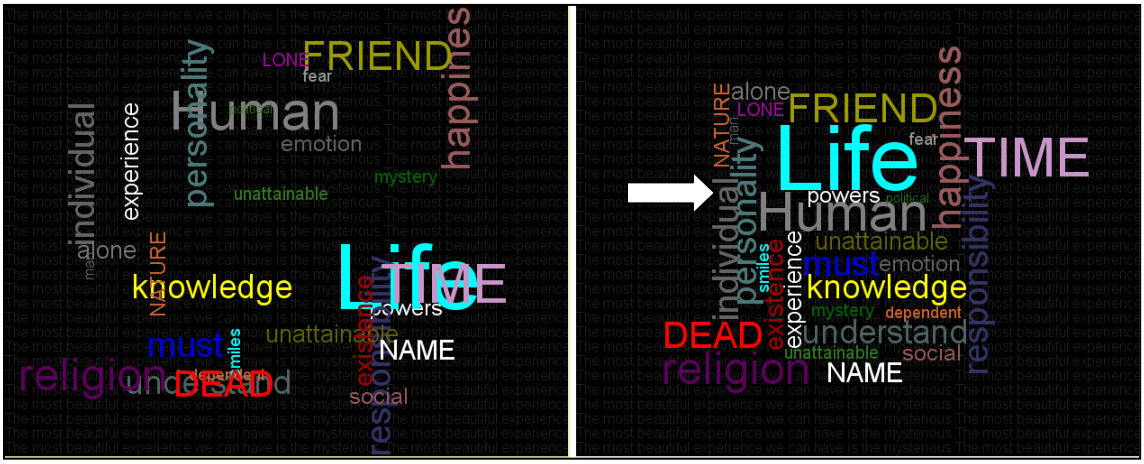
Essential Questions

What are the advantages to using **classes** in the organization of a program?
How does one isolate **side-effects** generated from transformation operations?
How does one assign a word's starting position **randomly** to one of the fours edges of the window?
What are the **mathematics expressions** needed to move each word from a window's edge to its final position?

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 a program to dynamically **alternate** between objects being in **motion** and then **at rest**?
How does the **map**() method achieve the same calculation for a word's movement in time as the full mathematics calculations?

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 **book-ending** 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 (a) the most frequently occurring words in a student-chosen text passage; or (b) key words in a film, play, song, poem, etc.

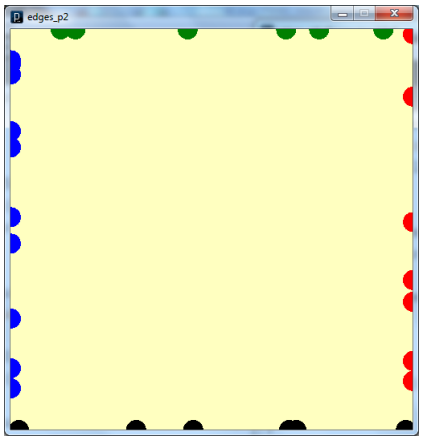
To add motion, the instructor gives students a starting template for a helper **"*edges*"** program to discover how to write code that will place each word at a random starting position on any of the four edges of the window. With instructor guidance, students discover a linear equation model for synchronizing the starting and ending times of all words from their initial to final positions. 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 about 5 seconds to their final positions, where they come to rest for an equal period of time. Program cycles "forever" between these two states.

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 method bodies for `rightEdge`(), `topEdge`() and `bottomEdge`().

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

```
Line 1:     float percent = random(0,100);
Line 2:     if percent < 25) { leftEdge(); }
```

```
Line 3:     else if (percent < 50) { rightEdge(); }
Line 4:     else if (percent < 75) { topEdge(); }
Line 5:     else { bottomEdge(); }

Line 1:     float percent = random(0,100);
Line 2:     if (0 <= percent && percent < 25) { leftEdge(); }
Line 3:     if (25 <= percent && percent < 50) { rightEdge();}
Line 4:     if (50 <= percent && percent < 75) { topEdge(); }
Line 5:     if (75 <= percent && percent < 100) { bottomEdge(); }
```

Students are asked to consider the two code fragments for structure, simplicity and clarity. They are asked to swap lines, e.g. swap lines 3 (rightEdge) and 4 (topEdge). 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 ***fraction*** (of motion completed) 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. 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.

```
final float TOTAL_FRAMES = 300;
float frames = frameCount % TOTAL_FRAMES;
if (frames == 0) {
  this.move = !this.move;
  if (this.move) {
      this.selectStartingEdge();
  }
}
float fraction = frames / TOTAL_FRAMES;
if (!this.move) {
  fraction = 1;
}

float totalDistanceX = this.xEnd - this.xStart;
float distanceXTraveled = totalDistanceX * fraction;
float x = distanceXTraveled + this.xStart;

float totalDistanceY = this.yEnd - this.yStart;
float distanceYTraveled = totalDistanceY * fraction;
float y = distanceYTraveled + this.yStart;
```

To make the objects rest for an equal amount of time, we introduce the boolean **move**, which toggles after each 300 frames (at 60 frames/sec, that's 5 seconds). The modulus calculation and subsequent calculation for **fraction** constrain the movement from traveling beyond the ending position. To make the word rest for an equal amount of time (another 5 seconds), when the value of the **move** variable toggles false, **fraction** is set to 1.

After students demonstrate proficiency in the concept for calculating a word's position-coordinates, they are introduced to the alternate coding below, which utilizes Processing's **map(n,min1,max1,min2,max2)** method, which maps a number relative to a range's minimum and maximum, to a second range. This simplifies the mathematics and offers students an alternate way to think about the calculation for a word's position over time. The variable **fraction** can now be dropped, and full movement is achieved by setting **frames** to **TOTAL_FRAMES** (differences shown below in **RED**).

```
final float TOTAL_FRAMES = 300;
float frames = frameCount % TOTAL_FRAMES;

if (frames == 0) {
  this.move = !this.move;
  if (this.move) {
      this.selectStartingEdge();
  }
}

if (!this.move) {
  frames = TOTAL_FRAMES;
}

float x = map(frames,0,TOTAL_FRAMES,this.xStart,this.xEnd);
float y = map(frames,0,TOTAL_FRAMES,this.yStart,this.yEnd);
```