University of California, Santa Cruz

Experiment 6:

Grammars and Text Art

Dominic Berardi & Nathan Prieto

CMPM 169: Creative Coding

Modes

Due Date 2/21/2023

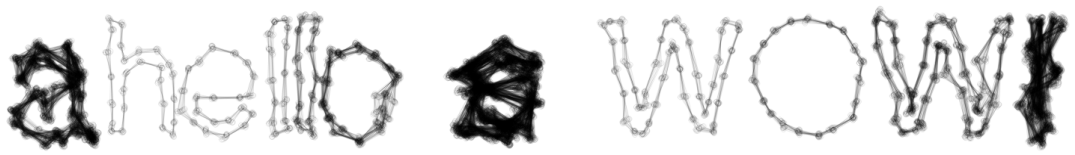
Trippy Text

This experiment uses tracery grammar, Perlin noise, and some other generative methods to create visually odd and disorienting messages.

Playable Link: here
Sketch.js code: https://github.com/domberardi10/cmpm169/blob/master/experiment6/js/sketch.js

Step 1: Imitate

We both talked about what kind of piece caught our attention this week and agreed that the one titled "P_3_2_3_01" by Generative Design was awesome and we wanted to know how it worked. Luckily for us, the piece was created using p5, so bringing it to our p5 editors was simple. But in order to start modifying it, we needed to understand how it works in the first place. http://www.generative-gestaltung.de/2/sketches/?01_P/P_3_2_3_01



The first big piece of the puzzle was this for loop that iterates through an array of "points". This must be what is creating the noise-like movement of the letters– it's not noise at all. Instead, each point's X and Y are randomly moved by a certain "step size" of which the "dance factor" controls how large the steps are.

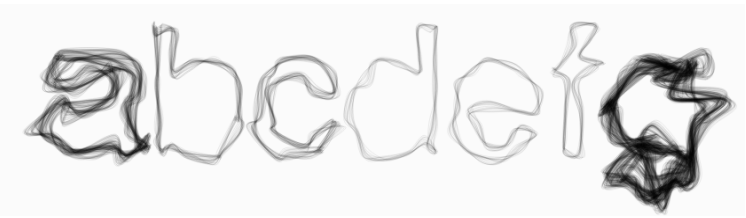
```
if (pnts.length > 0) {
  // let the points dance
  for (var i = 0; i < pnts.length; i++) {
    pnts[i].x += random(-stepSize, stepSize) * danceFactor;
    pnts[i].y += random(-stepSize, stepSize) * danceFactor;
  }
```

But all this does is set the points of the letters. Underneath this code block lies what truly draws the lines which connect the points, and thus make the deformed letter. The points of the current letter are once again iterated through, but this time vertices are created between the points, with an ellipse connecting them. Interestingly, there was a block of code labeled "rounded" that instead uses curveVertex() as opposed to vertex(). Replacing the uncommented block with this one gives awesome results, seen on the right. (Oddly enough, curveVertex() automatically creates lines between points, whereas vertex() does not.)

```
//  ------ lines: connected straight  ------
strokeWeight(0.1);
stroke(0);
beginShape();
for (var i = 0; i < pnts.length; i++) {
  vertex(pnts[i].x, pnts[i].y);
  ellipse(pnts[i].x, pnts[i].y, 7, 7);
}
vertex(pnts[0].x, pnts[0].y);
endShape();

//  ------ lines: connected rounded  ------
/*
  strokeWeight(0.08);

  beginShape();
  // start controlpoint
  curveVertex(pnts[pnts.length-1].x, pnts[pnts.length-1].y);
  // only these points are drawn
  for (var i = 0; i < pnts.length; i++) {
    curveVertex(pnts[i].x, pnts[i].y);
  }
  curveVertex(pnts[0].x, pnts[0].y);
  // end controlpoint
  curveVertex(pnts[1].x, pnts[1].y);
  endShape();
*/
```



We still haven't found how the points of the text are even being found or created. A function titled getPoints() exists further down, which upon further inspection uses methods such as getPath() and resampleByLength() that are not of p5 at all, but instead a JS package called OpenType (https://www.npmjs.com/package/opentype.js/v/0.3.0) and a graphical library (https://g.js.org/ref/resampleByLength.html), respectively. The use of OpenType explains how drastically the text on screen is able to get modified, as the package is designed to give fonts as their letterforms. Oddly, getPoints() is not called during draw() at all– instead, it is called within the setup() function. Even crazier, the function called is looped– *inside of setup*- so that it can understand what the newest letter typed is to function on its' points. P5 documentation explicitly states to avoid using loop() within setup()-- things got a little spicy!

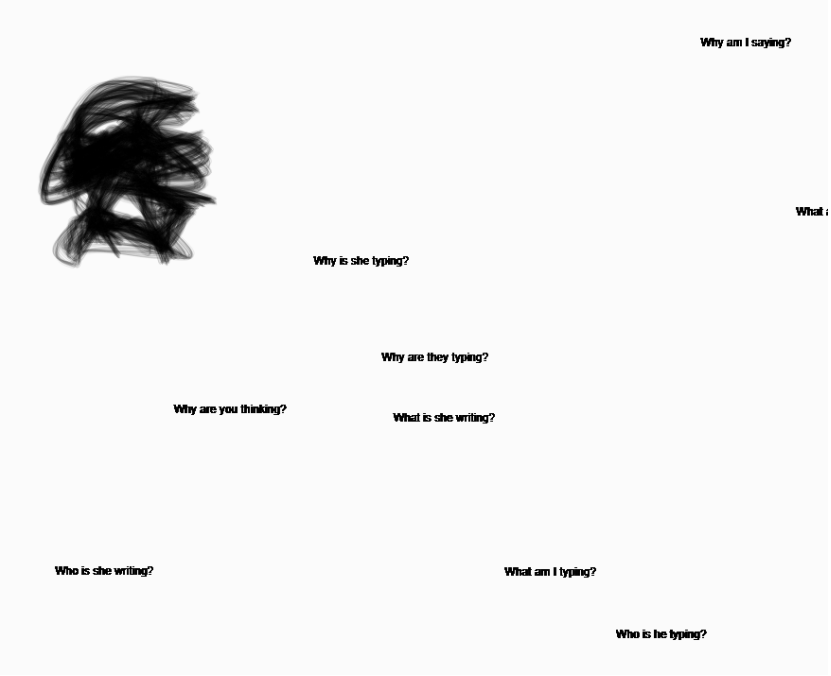
```
function getPoints() {
  fontPath = font.getPath(typedKey, 0, 0, 200);
  var path = new g.Path(fontPath.commands);
  path = g.resampleByLength(path, 25);
  textW = path.bounds().width;
  // remove all commands without a coordinate
  for (var i = path.commands.length - 1; i >= 0 ; i--) {
    if (path.commands[i].x == undefined) {
      path.commands.splice(i, 1);
    }
  }
  return path.commands;
}
```

```
function setup() {
  createCanvas(windowWidth, windowHeight);
  noLoop();

  opentype.load('data/FreeSansNoPunch.otf', function(err, f) {
    if (err) {
      print(err);
    } else {
      font = f;
      pnts = getPoints(typedKey);
      loop();
    }
  });
}
```

Step 2: Integrate

The first thing we had in mind to integrate into the existing project is tracery grammar, something we talked much about in class. Our idea involves using the generated tracery grammars in conjunction with the cool OpenType warping that is going on in the original work. To start, we made a basic JSON that stored various vague questions as its grammar. Then, a new sentence would be created every 2 seconds. This was just to test if tracery would even work with the existing code at all. Luckily, it worked.



A new class was needed in order to store each generated tracery output. This became the class “sentence” which simply stores the generated words as well as a random X and Y position for the sentence to be placed. Every 2 seconds, the new Sentence would be placed into the sentence array, which was iterated through every draw() cycle to place the text onto the screen.

```
  push();
  fill(1);
  //GRAMMAR STUFF
  grammarTimer += deltaTime;
  if (grammarTimer > 2000){
    let output = grammar.flatten("#origin#");
    sentenceArray.push(new Sentence(output, random(0, windowWidth), random(0,
windowHeight)));
    grammarTimer = 0;
  }
  for (let i = 0; i < sentenceArray.length; i++){
    text(sentenceArray[i].words, sentenceArray[i].x, sentenceArray[i].y);
    print(sentenceArray[i].words);
  }
  pop();
```

```
class Sentence{
  constructor(words, x, y){
    this.words = words;
    this.x = x;
    this.y = y;
  }
}
```

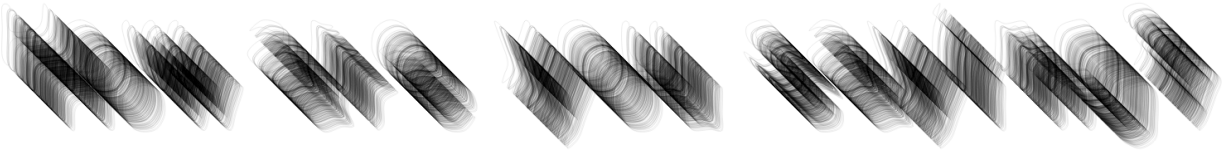
It was not too hard to integrate the use of tracery grammar into the project. Not shown here are a few extra lines in index.html to call the libraries. However, what we really wanted to do is use the generated tracery in conjunction with the OpenType warping.

Step 3: Innovate

For the innovation step of our project we decided to take existing generative methods, such as Perline noise functions, to "animate" the existing lines in unique and unexpected ways. We first attempted to give each vertex of a letter to have an x and y offset that varied with the frame count of the canvas while also being affected by the user's mouse input.

```
switch(this.version){
  case 0:
    for (let i = 0; i < pnts.length; i++) {
      pnts[i].x += noise(frameCount * 50 / mouseX) * 100;
      pnts[i].y += noise(frameCount * 50 / mouseY) * 100;
    }
    return pnts;
```

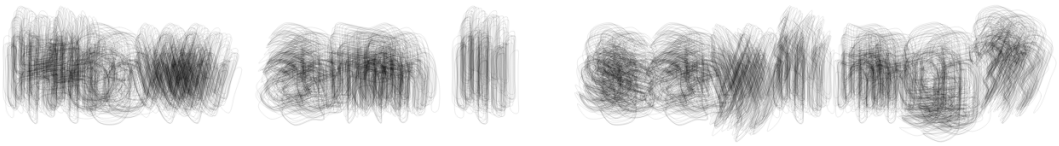
We then got curious about what values and parameters could be changed as well as the results they would yield. We messed around with amplifying and dividing frameCount by different values as well multiplying the result of noise() by varying scalars. Multiplying frameCount by 50 and then multiplying the result of the noise function by 100 produced an interesting 3D effect that affected vertices on both the x and y axes.

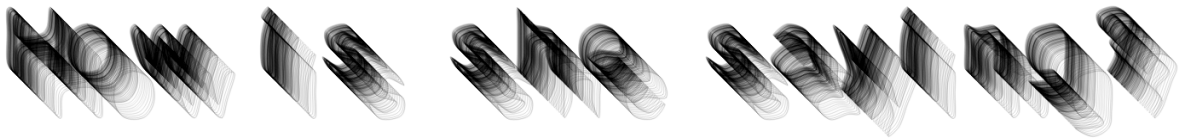


We then came to the idea of creating different versions of vertice manipulations and created 5 different segments of code that affect them in unique ways. We first created a random number between 0 and 5 that defined a sentence's version and when the code obtained points for each letter, it would then call movePoints() which contains a switch statement. This statement controls what values will be added the the x and y positions of each vertice. We decided to keep frameCount within each case as we wanted the points to animate in some way without having the rely on user input.

```
  movePoints(pnts){
    switch(this.version){
      case 0:
        for (let i = 0; i < pnts.length; i++) {
          pnts[i].x += noise(frameCount * 50 / mouseX) * 100;
          pnts[i].y += noise(frameCount * 50 / mouseY) * 100;
        }
        return pnts;
      case 1:
        for (let i = 0; i < pnts.length; i++) {
          pnts[i].x += noise(frameCount * 50) * 100;
          pnts[i].y += noise(frameCount * 50) * 100;
        }
        return pnts;
      case 2:
        for (let i = 0; i < pnts.length; i++) {
          pnts[i].x += noise(frameCount / 50) * 100;
          pnts[i].y += noise(frameCount / 50) * 100;
        }
        return pnts;
      case 3:
        for (let i = 0; i < pnts.length; i++) {
          pnts[i].x += noise(frameCount * (i * 5)) * 100;
          pnts[i].y += noise(frameCount * (i * 5)) * 100;
        }
        return pnts;
      case 4:
        for (let i = 0; i < pnts.length; i++) {
          pnts[i].x += noise(frameCount * 20) * 50;
        }
        return pnts;
    }
  }
}
```

With all of these components combined we created a piece that combines defined grammar conventions supported by Tracery, manipulation of fonts and text using OpenType, and existing generative methods such as Perlin noise.





Reflect

Dominic:

The experiment this week was a bit unconventional in comparison to the previous weeks Nate and I have worked together, as he was out of town for a while. To mitigate the amount of work we would have to do later, I started messing around with the code and piece we both decided we liked and broke down it to further understand how it worked. I began to add tracery

grammar, hoping to use it with the existing code, but I was hitting a roadblock with it, to which Nate was able to help figure out when he got back. From there we kept making steady progress on how we could innovate it further, and honestly we're running out of time and really just wanted to get to work on other classes. Though, we were still able to create an interesting end result. The result was reached by a combination of Nate finding out how to easily change the generated tracery grammar a variety of ways and then I suggested we randomize which way the text is warped. Altogether, I am happy with the result though I think we could have kept increasing how cool it is even further, if we had the time.

Nathan:

This project was really fun to work on with Dominic as we spent most of our time editing existing values and parameters that produced some interesting results. I was away for most of the weekend while Dominic had created an adequate starting point for the project by examining some reference code. When I returned we both collaborated on the functionality of the project as well as the overall vision we had in mine when it actually came time to develop. Understanding the reference code took a little bit of time as the original author had written alot of extraneous variables and statements that were very redundant. This made comprehending the base code difficult until we started to implement our own version which cut out many repeat lines in favor of a more succinct control flow. I am overall satisfied with the final product as it incorporates grammar, randomness, and generative methods that creates unique visuals as it runs.

Self-Evaluation

Dominic

Self Evaluation Rubric						
Did you complete the assignment and did you complete it on time?	Submitted on time	Up to 1 day late	Up to 2 days late	Up to 3 days late	4 days late or more	Do you need to clarify? No
	X	☐	☐	☐	☐	
Did you collaborate with a partner?	Worked with partner			Worked alone		Do you need to clarify? No
	X			☐		
Did you put in earnest effort and provide an articulate summary of your experience?	Excellent	Pretty good	About average	Could be improved	Not this time	What supports this? I tried to explain my process as thoroughly as possible. Used many pictures.
	X	☐	☐	☐	☐	

Was the assignment complete, with minimal errors, correct output, and good style?	**Excellent**	**Pretty good**	**About average**	**Could be improved**	**Not this time**	**What supports this?** **Had two people look over each part of the assignment.**
	X		☐	☐	☐	
How much EXTRA effort did you put into the assignment?	**A lot of extra effort**		**Some extra effort**		**Not this time**	**What supports this?** **Just wanted it to be done after a while.**
	☐		**X**		☐	
Reflection in section above.						

Nathan

Self Evaluation Rubric						
Did you complete the assignment and did you complete it on time?	**Submitted on time**	**Up to 1 day late**	**Up to 2 days late**	**Up to 3 days late**	**4 days late or more**	**Do you need to clarify?** **No**
	X	☐	☐	☐	☐	
Did you collaborate with a partner?	**Worked with partner**			**Worked alone**		**Do you need to clarify?** **No**
	X					
Did you put in earnest effort and provide an articulate summary of your experience?	**Excellent**	**Pretty good**	**About average**	**Could be improved**	**Not this time**	**What supports this?** **Our process is well documented with adequate pictures that show every step of the project.**
	X	☐	☐	☐	☐	
Was the assignment complete, with minimal errors, correct output, and good style?	**Excellent**	**Pretty good**	**About average**	**Could be improved**	**Not this time**	**What supports this?** **The writeup is sufficient enough for what the teaching team expects and our code is neatly organized.**
	X		☐	☐	☐	
How much EXTRA effort did you put into the assignment?	**A lot of extra effort**		**Some extra effort**		**Not this time**	**What supports this?** **Understanding the original author's code took and extra**

			X		□	**amount of time which took away from actually developing.**
Reflection in section above.						