University of California, Santa Cruz

Experiment 2:

Vector Art, Animation, and Interactivity

Dominic Berardi

CMPM 169: Creative Coding

Modes

Due Date 1/24/2023

Self-Driving Lightcycles

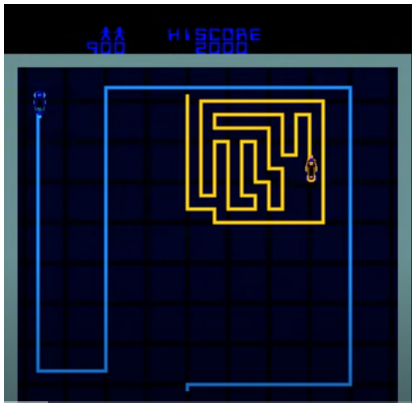
This experiment asked us to create vector art that also has animation and interactivity built into it. My piece is inspired by the 1982 video game *Tron*, attempting to recreate the lightcycle game into a work of art. May or may not have succeeded, depending on how you look at it. Click on the canvas to add more lightcycles.

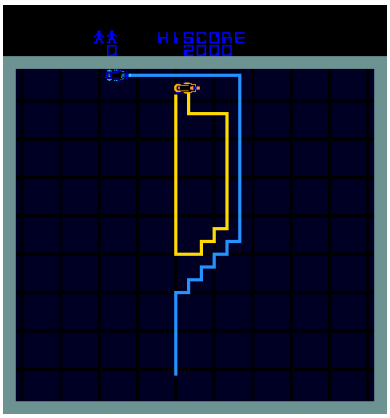
Playable link: https://editor.p5js.org/dberardi/full/awrxvRmUv
Sketch.js code: https://github.com/domberardi10/cmpm169/blob/master/experiment2/js/sketch.js

Step 1: Imitate

For this experiment, I wanted to try to imitate something that isn't necessarily vector art, nor is it something even made using processing. In our discussions about vector art, we looked over many different p5.js functions, yet one stood out as neglected in its use when compared to crazy art using ellipsis and curves- the line() function. Possibly the most simple of them all– how could I use this function to create animated, interactive vector art?

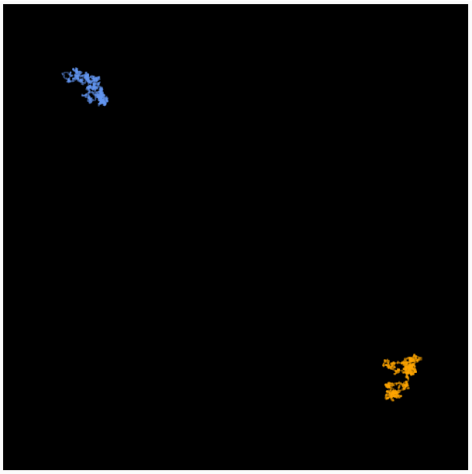
That's when I thought about an arcade game I played fairly recently at an arcade museum– based on a movie I love, too– Bally Midway's 1982 classic, *Tron*. When someone mentions *Tron*, what's the first thing they think of? Lightcycles! I've always been intrigued by the simplicity of the lightcycle game, and yet I can't help but watch it as the cycles make the most interesting shapes and paths to avoid hitting one another. Really, it's all just a bunch of lines. Perfect!





I whipped up some code (well, not exactly “whipped up” as I took an hour to study p5.js’s documentation as I kept running into the most simple errors) that… sort of tries to emulate two lightcycles moving along their paths. This code would not check for lightcycle collisions (“derezzed” as it’s called in the film) nor would it try to keep the cycles within the bounds. It just keeps drawing two lines, each changing in a random cardinal direction each draw() cycle– though, “lines” is a bit of a misnomer. Even though I am using the line() function, I kept x1 = x2 and y1 = y2, which essentially draws a single pixel at the given (x, y) coordinate.

```
function draw() {
  stroke("#6495ED");
  line(bx1, by1, bx2, by2);
  stroke("#FFA500");
  line(ox1, oy1, ox2, oy2);
  // BLUE NEXT PIXEL
  nextBlue = int(random(1, 5));
  // UP?
  if (nextBlue == 1) {
    by1 -= 1;
    by2 -= 1;
  }
  // DOWN?
  else if (nextBlue == 2) {
    by1 += 1;
    by2 += 1;
  }
  // LEFT?
  else if (nextBlue == 3) {
    bx1 -= 1;
    bx2 -= 1;
  }
  // RIGHT?
  else if (nextBlue == 4) {
    bx1 += 1;
    bx2 += 1;
  }
  // ORANGE NEXT PIXEL
  nextOrange = int(random(1, 5));
  // UP?
  if (nextOrange == 1) {
    oy1 -= 1;
    oy2 -= 1;
  }
```

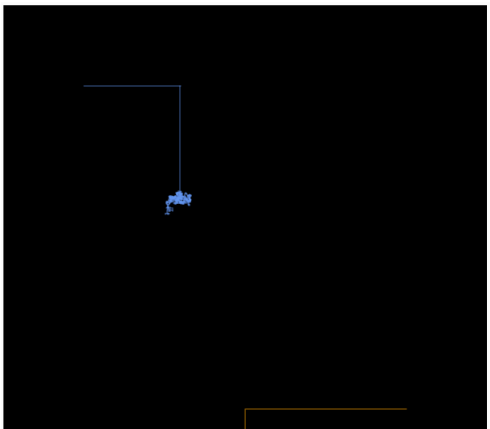


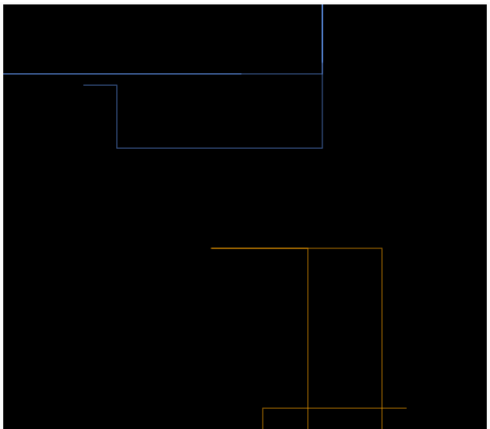
Well, that doesn’t look right.

The code works perfectly fine, but there’s a couple issues:

1.) Since the draw() function selects a new direction every single frame, there’s no way it will ever keep a consistent line, instead becoming a squiggly mess (it IS still a single line, though).

2.) Since there is nothing checking for “collisions”, or in other words, if a pixel has *already* been drawn onto, there are a bunch of overlapping pixels.



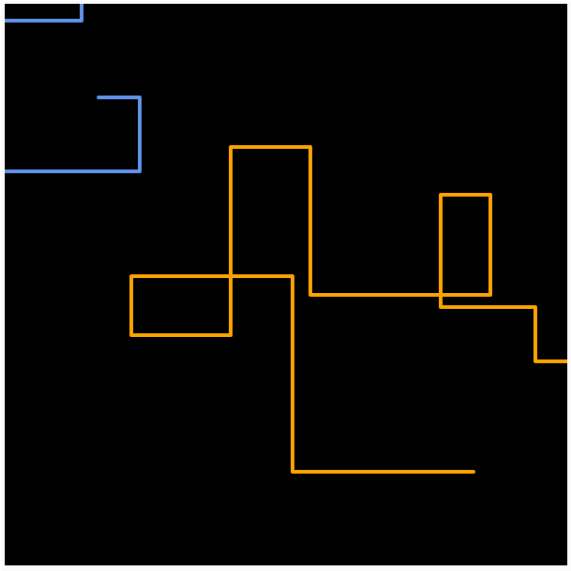


Some progress has been made, but still things to fix. The above left image was a simple, but funny mistake; forgot to reset a variable to 0 at the right time. Once that was fixed, the above right image is what occurred. Now the lightcycles draw actual lines, but there's still the issue of overlapping lines and moving offscreen. The former occurs because the code can still randomly choose the exact opposite direction the line was moving in, which cannot occur within *Tron*. The latter is simply because I have yet to come up with a way to account for it (will I even?!). Here's what the draw() function looks like after the changes to keep them as true lines. A random time is chosen between each direction change, and this is compared to the time elapsed since the last direction change, in which deltaTime comes quite in handy.

```
function draw() {
  blueDelta += deltaTime / 1000;
  stroke("■#6495ED");
  line(bx1, by1, bx2, by2);
    // UP?
    if (blueDir == 1) {
      by1 -= 1;
      by2 -= 1;
    }
    // DOWN?
    else if (blueDir == 2) {
      by1 += 1;
      by2 += 1;
    }
    // LEFT?
    else if (blueDir == 3) {
      bx1 -= 1;
      bx2 -= 1;
    }
    // RIGHT?
    else if (blueDir == 4) {
      bx1 += 1;
      bx2 += 1;
    }
  // BLUE NEXT DIR
  if (blueDelta > blueTime) {
    blueDir = int(random(1, 5));
    blueTime = random(0.5, 4);
    blueDelta = 0;
  }
```

```
function findNextDir(currDir){
  if (currDir == 1 || currDir == 2){
    return int(random(3, 5));
  }
  else if (currDir == 3 || currDir == 4){
    return int(random(1, 3));
  }
}
```

In order to resolve the overlapping issue, I needed a way to ensure that if the line is on the X plane, it could only move to the Y plane, and vice versa. Thus, I created a function findNextDir(), image above right. Though this solves when a line runs in one direction, and then reverses its direction, lines can still intersect (not explicitly overlap) each other. In addition, the lines can still go offscreen. I started to write a function to test what the (x, y) looked like in the direction the line is moving towards plus 5, to see if that point is either: 1. Offscreen, or 2. Has an existing color that is not the black background already on the point. However, I quickly realized that the function needs the current (x, y) data of the line, and there's no easy way to pass that information yet. Unless… I convert the code to OOP!



Things are looking pretty similar to what one might expect when thinking of lightcycles, issues aside. So I think it's time to move on to the next step: integrating OOP.

Step 2: Integrate

Luckily, converting my existing code to OOP shouldn't be too hard, as I already have copied variables for the blue and orange lines, respectively. I know what sorts of information the "lightcycle" object must contain. However, I am a bit unfamiliar with class syntax in Javascript– so I found a handy article that describes how to create a lightcycle game in JS. Note that I am not using any code from this article, as it is recreating the whole game whereas I am making an "automated" version; simply referencing it for how the below player class is set up, as well as insight into how the game version works. The below image is from the article.

https://javascript.plainenglish.io/building-a-classic-arcade-game-with-javascript-and-html5-canvas-18494e4201ba

```
class Player {
  constructor(x, y, color) {
    this.color = color || '#fff';
    this.dead = false;
    this.direction = '';
    this.key = '';
    this.x = x;
    this.y = y;
    this.startX = x;
    this.startY = y;

    this.constructor.counter = (this.constructor.counter || 0) + 1;
    this._id = this.constructor.counter;

    Player.allInstances.push(this);
  };
};

Player.allInstances = [];
```

Simple enough. No longer will I have a mess of variables, nor two long blocks of very similar code for each lightcycle, as I can create a function that the class utilizes for each instance. Here's comparisons of the old version of the code with the new, shiny OOP version. **Left side is old, the right side is new.**

```
let bx1;
let bx2;
let by1;
let by2;
let blueDir;
let blueDelta;
let blueTime;

let ox1;
let ox2;
let oy1;
let oy2;
let orangeDir;
let orangeDelta;
let orangeTime;
```

```
class Lightcycle {
  constructor(x, y, color, dir) {
    this.x = x;
    this.y = y;
    this.color = color;
    this.dir = dir;
    this.delta = 0;
    this.time = random(0.5, 3);
  }
}
```

```
function setup() {
  createCanvas(600, 600);
  background("#000000");
  strokeWeight(4);
  bx1 = 100;
  bx2 = 100;
  by1 = 100;
  by2 = 100;
  blueDir = 4;
  blueTime = random(0.5, 3);
  blueDelta = 0;
  ox1 = 500;
  ox2 = 500;
  oy1 = 500;
  oy2 = 500;
  orangeDir = 3;
  orangeTime = random(0.5, 3);
  orangeDelta = 0;
}
```

```
function setup() {
  createCanvas(600, 600);
  background("■#000000");
  strokeWeight(4);
  lcBlue = new Lightcycle(100, 100, "■#6495ED", 4);
  lcOrange = new Lightcycle(500, 500, "■#FFA500", 3);
}
```

```
function draw() {
  blueDelta += deltaTime / 1000;
  stroke("#6495ED");
  line(bx1, by1, bx2, by2);
    // UP?
    if (blueDir == 1) {
      by1 -= 1;
      by2 -= 1;
    }
    // DOWN?
    else if (blueDir == 2) {
      by1 += 1;
      by2 += 1;
    }
    // LEFT?
    else if (blueDir == 3) {
      bx1 -= 1;
      bx2 -= 1;
    }
    // RIGHT?
    else if (blueDir == 4) {
      bx1 += 1;
      bx2 += 1;
    }
  // BLUE NEXT DIR
  if (blueDelta > blueTime) {
    blueDir = findNextDir(blueDir);
    blueTime = random(0.5, 3);
    blueDelta = 0;
  }

  orangeDelta += deltaTime / 1000;
  stroke("#FFA500");
  line(ox1, oy1, ox2, oy2);
    // UP?
    if (orangeDir == 1) {
      oy1 -= 1;
      oy2 -= 1;
    }
    // DOWN?
    else if (orangeDir == 2) {
      oy1 += 1;
```

```
function moveLightcycle(lc){
  lc.delta += deltaTime / 1000;
  stroke(lc.color);
  line(lc.x, lc.y, lc.x, lc.y);
    // UP
    if (lc.dir == 1) {
      lc.y -= 1;
      lc.y -= 1;
    }
    // DOWN
    else if (lc.dir == 2) {
      lc.y += 1;
      lc.y += 1;
    }
    // LEFT
    else if (lc.dir == 3) {
      lc.x -= 1;
      lc.x -= 1;
    }
    // RIGHT
    else if (lc.dir == 4) {
      lc.x += 1;
      lc.x += 1;
    }
  // NEXT DIR
  if (lc.delta > lc.time) {
    lc.dir = findNextDir(lc.dir);
    lc.time = random(0.5, 3);
    lc.delta = 0;
  }
}

function draw() {
  moveLightcycle(lcBlue);
  moveLightcycle(lcOrange);
}
```

Everything is working as it did before, with much cleaner code. However, the lines continue to drift off out of bounds and intersect with each other. I believe the fix to this needs some sort of "override" function, and thus I created overrideDir(). The out of bounds check is fairly simple, requiring a check whether the current X or Y values are within the bounds, and if they are about to go out of bounds, then forcibly change their direction.

```
function overrideDir(lc){
  if (lc.dir == 1){
    if (lc.y - 5 <= 15){
      lc.dir = findNextDir(lc.dir);
      lc.time = random(0.5, 3);
      lc.delta = 0;
    }
  }
  else if (lc.dir == 2){
    if (lc.y + 5 >= 585){
      lc.dir = findNextDir(lc.dir);
      lc.time = random(0.5, 3);
      lc.delta = 0;
    }
  }
  else if (lc.dir == 3){
    if (lc.x - 5 <= 15){
      lc.dir = findNextDir(lc.dir);
      lc.time = random(0.5, 3);
      lc.delta = 0;
    }
  }
  else if (lc.dir == 4){
    if (lc.x + 5 >= 585){
      lc.dir = findNextDir(lc.dir);
      lc.time = random(0.5, 3);
      lc.delta = 0;
    }
  }
}
```

This works great, keeping the lightcycles within bounds as they should, so I moved onto the issue of intersecting lines. The most straightforward way to accomplish this is to check what color the next pixels will be; if it's not pure black (#000000) like the background is then clearly a line has already been drawn on that pixel. The built-in function get() can grab the color of a given pixel, so I tried using that… only for it to not only not work properly, but lag the program extremely badly. P5.js documentation notes that get() is a pricey function, and calling it hundreds of times as I am is simply not feasible. The documentation also provides a faster way to access pixel colors using the built-in array pixels[], but alas, it lags as well, albeit not as badly (it was also returning undefined, but that's besides the point).

```
get(lc.x, lc.y - 10) != "#000000"
```

```
function pixelTest(x, y){
  let off = (y * width + x) * pixelDensity() * 4;
  let components = [
    pixels[off],
    pixels[off + 1],
    pixels[off + 2],
    pixels[off + 3]
  ];
  print(components);
}
```

I fiddled around with these and similar solutions for over an hour to no avail. There are other solutions to checking if a line has been drawn already, but these rely on a grid system, which I am not using. With my best options not working, I decided to cut my losses– this was the time to diverge from making art inspired by *Tron* and making something different.

Step 3: Innovate

Even though things weren't working as I had hoped for, the lightcycles still make interesting patterns and grids as they move along the canvas. But there were only two, each a set color, and no way for anyone to interact with it. So… what if clicking added random colored lightcycles at the mouse position? That could look interesting.

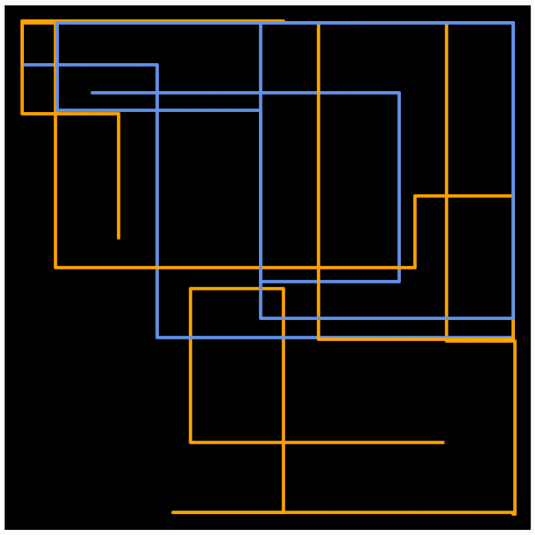
If I want to be able to generate new lightcycles, I'm going to have to make something to store and iterate through them to move each one. What better than an array?

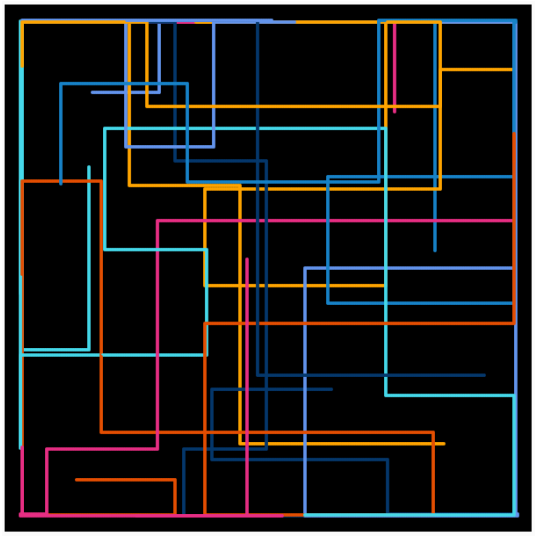
```
function draw() {
  overrideDir(lcBlue);
  moveLightcycle(lcBlue);
  overrideDir(lcOrange);
  moveLightcycle(lcOrange);
  for (let lc = 0; lc < lcArray.length; lc++){
    overrideDir(lcArray[lc]);
    moveLightcycle(lcArray[lc]);
  }
}
```

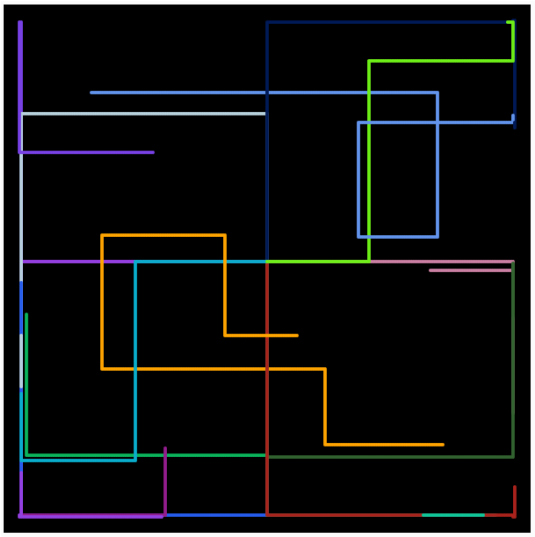
And when the mouse is clicked (to which there exists a mouseClicked() function) a new lightcycle object is created and pushed into the array. But I found myself having an issue– the color attribute of the Lightcycle class. When I created the blue and orange lightcycles, they simply used a hex code for each; but how was I to randomly generate a hex code? I looked at the documentation on stroke() to see what are other ways of inputting a color, and found the string 'rgb(r,g,b)' works. Generate each color value, then splice together the parts into the full string, and boom. Random colors!

```
function mouseClicked(){
  let rStr = str(int(random(0, 255)));
  let gStr = str(int(random(0, 255)));
  let bStr = str(int(random(0, 255)));
  let rgbStr = "rgb(" + rStr + "," + gStr + "," + bStr + ")";
  lcArray.push(new Lightcycle(mouseX, mouseY, rgbStr, int(random(1, 5))));
}
```

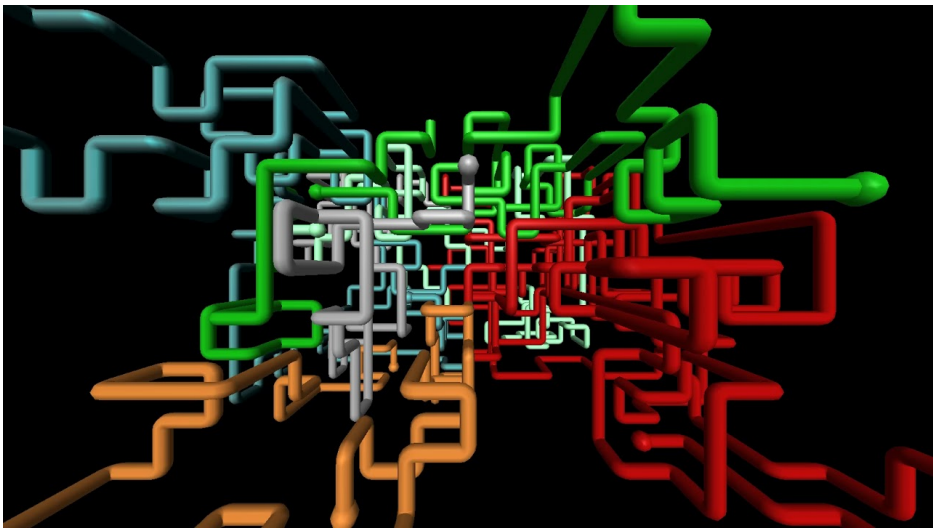
After some debugging of my iterator through the lightcycles, it was complete. Moment of truth– will it look interesting?







Good enough! I've spent way too much time on this without even having a lot to show for, but oh well. The end result reminds me of the Windows pipe screensaver, funnily enough. (Image below not mine.)



Reflect

After everything is said and done, I think what I made is visually interesting to a point, but the novelty wears off quickly. I'm disappointed that I couldn't follow the rules of *Tron*'s lightcycles exactly, but this piece did evolve over the course of working on it into something inspired, but different. I was the sole person working on this experiment, and I think I'll find a partner for future ones, as progress could have been made much quicker than I alone, and the end result would likely be much better. The high of this experiment was how I converted my code into OOP, I believe it translated very well. The low of the experiment would be not getting the intersection of lines working in time. Altogether, I put in good effort into the experiment, working about 8 hours over the course of 2 days.

Self-Evaluation

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?** **I think my writeup and code are formatted well, though I didn't try to perfect either of them.**
		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?** **I continued to search for ways to fix issues that were not necessary, but ultimately gave up after the time sink.**
	☐		X		☐	
Reflection in section above.						