Impact Lab 2025: Programming Fundamentals
Lecture 4: Randomness

Summer 2025

School of Computing and Data Science

Wentworth Institute of Technology

```
int getRandomNumber()
{
    return 4;  // chosen by fair dice roll.
               // guaranteed to be random.
}
```


Finishing up Last Time: Bubbles!

Let's make a Bubble class.

Each bubble will be a circle with an outline only.

They move by randomly jittering around on the screen.

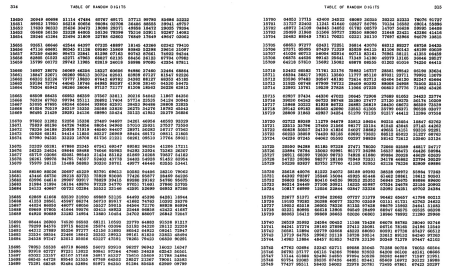
They'll have two methods, **`move()`** and **`show()`** that we will call from **`draw()`**.

Finishing up Last Time: Wrap up

- A class defines a complex variable type
  - Contains its own data fields and functions that are only for use with objects of that class
- There are many predefined classes in JavaScript.
- You can also define your own classes
  - Often done to represent an entity in your program that requires more than one variable
- This is just the beginning of object oriented (OO) software development

Too Important To Be Left To Chance

- Now that you have some of the fundamentals and we've played around with objects, let's talk randomness
- Today, I want to do some simple motion: Random Walk.
- Then, we'll talk about noise and how we can use randomness to generate interesting visuals.



Two pages from "A Million Random Numbers with 100,000 Normal Deviates" by the RAND corporation in 1947

Random Walk

Imagine you are on a balance beam.

Every 10 seconds you flip a coin.

If it's heads you take a step forward

If it's tails you take a step back

This is a random walk!

Now, imagine doing this in two dimensions:

Flip 1	Flip 2	Result
Heads	Heads	Forward
Heads	Tails	Right
Tails	Heads	Left
Tails	Tails	Back

Science

Random Walk

Imagine you are on a balance beam.

Every 10 seconds you flip a coin.

If it's heads you take a step forward

If it's tails you take a step back

This is a random walk!

Now, imagine doing this in two dimensions:

Seems a little unsophisticated, but you can use this model to simulate many things in the real world.

		Result
		Forward
		Right
Tails	Heads	Left
Tails	Tails	Back

Science

Random Walker

Let's create a walker object to represent the person (or thing) that moves in this way.

Two questions that I like to ask in this class are:

- "How can we define the rules that govern the behavior of an object?"
- "How can we implement these rules in our code?"

The random walk lets us see the answers to both of these for a simple, but very useful, simulation.

Wentworth
Computing & Data Science

Random Walker

```
class Walker {
  constructor(){
    this.x = width/2;
    this.y = height/2;
  }
```

Classes initialize their data in the constructor.

Our initial data is the position (**x**,**y**) of the walker.

I'm positioning the walker in the center of our canvas by default.

Remember: using **this** creates the data in the class that we can update whenever we want.

We'll use methods to move and draw the walker, similar to the ball that we had moving around the canvas.

Wentworth
Computing & Data Science

Random Walker: `show()`

`show()` is still within the brackets for the **`Walker`** class because it is a method of that class.

```
show(){
  stroke(0);
  point(this.x, this.y);
}
```

Remember to use **`this`** to access the data that was setup in the constructor.

- This structure is fairly common:
  - Numeric data that represents position/color/size/etc.
  - Methods that use that data to display the object.

Random Walker: `step()`

- To simulate the movement, we'll use a method called **`step()`**.
- This is where the real work happens.
- Remember: **`random(4)`** will give you a random number from 0-4 but not including 4, that's why we **`floor`** the value to get an integer from 0-3.

```
step(){
  let choose = floor(random(4));
  if(choice == 0){
    this.x++;
  } else if (choice == 1){
    this.x--;
  } else if (choice == 2){
    this.y++;
  } else {
    this.y--;
  }
}
```

Each time we run this (once per frame) it will change the position on the canvas by 1 pixel.

Random Walker: The Sketch

```
let walker;

function setup(){
  createCanvas(640,240);
  walker = new Walker();
  background(255);
}

function draw(){
  walker.step();
  walker.show();
}
```

Declare a variable called **`walker`** (that is empty)

Setup the canvas, background, and a new **`Walker`** object stored in **`walker`**

Each frame, move the walker with **`step()`** and draw it with **`show()`**

Let's see what this looks like.

Random Walker: The Sketch

```
let walker;

function setup(){
  createCanvas(640,240);
  walker = new Walker();
  background(255);
}

function draw(){
  walker.step();
  walker.show();
}
```

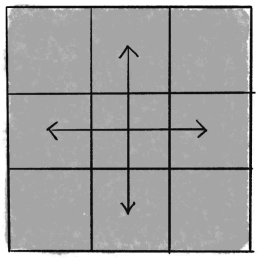
Declare a variable called **`walker`** (that is empty)

Setup the canvas, background, and a new **`Walker`** object stored in **`walker`**

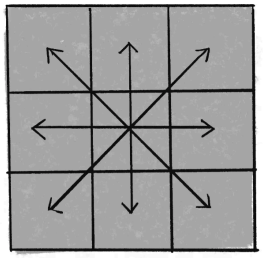
Since we only clear the background once in setup(), the "trail" of the walker will stay on the screen.

Let's see what this looks like.

Random Walker: More Directions



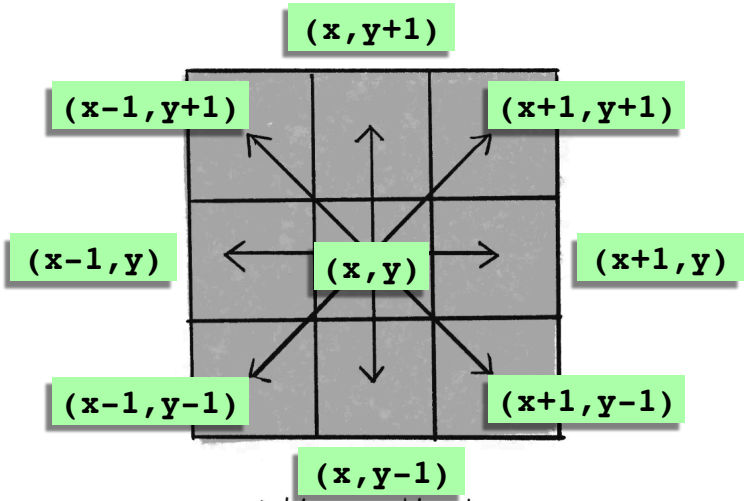
four possible steps



eight possible steps

We're only doing the cardinal directions. How might we do diagonals too?

Random Walker: More Directions



eight possible steps

Random Walker: More Directions

We could just do more if statements, but that becomes repetitive and cumbersome (and possibly error prone).

```
step(){
  let xstep=floor(random(3))-1;
  let ystep=floor(random(3))-1;

  this.x += xstep;
  this.y += ystep;
}
```

Instead, each possibility is just adding 0, 1, or -1 to x and y...

Remember: **`random(3)`** gives us values between 0-3 (not including 3), by flooring it, we get 0, 1, or 2. Then, subtracting 1 gives -1, 0, or 1.

Random Walker: More Directions

```
step(){
  let xstep=random(-1,1);
  let ystep=random(-1,1);

  this.x += xstep;
  this.y += ystep;
}
```

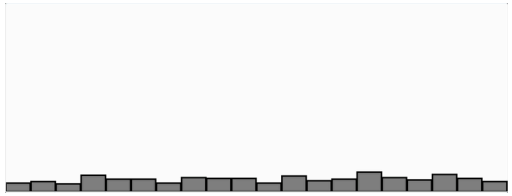
Taking this one step further, getting rid of the floor and just generating a random number between -1 to 1 give us a continuous step.

Don't get lost in these variations, they all have one thing in common:

At any moment in time, the walker has an equal probability to step in any of the possible directions.

`random()`

- The **`random()`** function in p5.js produces a uniform distribution of numbers.
- We can test this by generating many numbers, putting them in bins based on their value, and seeing what happens.

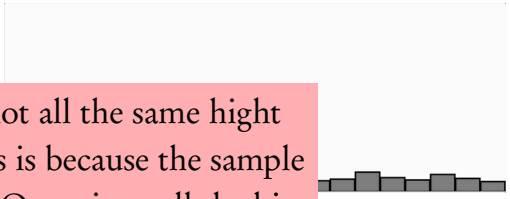


As the numbers are generated and binned, the hight of the bin goes up. Note that they all go up at about the same rate!

I won't make us code this up, but it is worth seeing.

`random()`

- The **`random()`** function in p5.js produces a uniform distribution
- We can test t
generating m
putting them
on their valu
what happen



They are not all the same hight though. This is because the sample size is small. Over time, all the bins would even out (if we use a good random number generator).

e generated
ight of the bin goes up. Note that they all go up at about the same rate!

I won't make us code this up, but it is worth seeing.

Nonuniform Distributions and Probability

- Uniform random numbers are useful in many situations, but don't always represent results that we may encounter in nature.
- Sometimes, we need to put our thumb on the scale!
- We'll start by thinking about single-event probability

Nonuniform random number generators will give some ranges of numbers more often than others.

By defining which numbers we want more often, we control the probability.

Single-Event Probability

Say we had a normal 52 card deck.

What is the probability of drawing an Ace?

$$\frac{\text{Number of Aces}}{\text{Number of Cards}}$$

$$\frac{4}{52} = 0.077 \approx 0.08 \text{ or } 8\%$$

Probability of drawing any diamond?

$$\frac{\text{Number of Diamonds}}{\text{Number of Cards}} = \frac{13}{52} = 0.25 \text{ or } 25\%$$

Multiple-Event Probability

We can also compute the probability of multiple events occurring in sequence...

Probability of getting 3 heads in a row by flipping a coin?

$$\frac{1}{2} \times \frac{1}{2} \times \frac{1}{2} = \frac{1}{8} = 0.125 \text{ or } 12\%$$

Each flip has a 50/50 chance to be heads, so we multiply each chance together to get the multiple-event probability.

Exercise

What is the probability of getting two aces in a row from a normal deck of 52 cards if:

- You reshuffle the first ace into the deck before drawing again?
- You didn't reshuffle the ace back into the deck before drawing again?

Nonuniform Walker

- Let's change our walker so that it tends to move to the right using these rules:
  - Up: 20% chance
  - Down: 20% chance
  - Left: 20% chance
  - Right: 40% chance

```
step(){
  let r = random();
  if(r < 0.4){
    this.x++;
  } else if(r < 0.6){
    this.x--;
  } else if(r < 0.8){
    this.y++;
  } else {
    this.y--;
  }
}
```

Note that the total % adds up to 100 (The walker must take a step, therefore 100% chance to move)

Nonuniform Walker

- Let's change our walker so th
- ri
- Right: 40% chance

We generate a random number between 0 and 1 then make the movement decision based on the value we get.

```
step(){
  let r = random();
  if(r < 0.4){
    this.x++;
  } else if(r < 0.6){
    this.x--;
  } else if(r < 0.8){
    this.y++;
  } else {
    this.y--;
  }
}
```

Note that the total % adds up to 100 (The walker must take a step, therefore 100% chance to move)

Nonuniform Walker

Another common use for this technique is to control the probability of an event that you want to occur sporadically in your code.

Say you want to spawn a new walker every 100 frames on average.

You could generate a random number, check if it is below 0.01. On average, you'll get a new walker every 100 frames!

Exercise

Create a random walker with dynamic probabilities:

Give it a 50% chance to move in the direction of the mouse. Use **`mouseX`** and **`mouseY`** to get the current mouse position.

This one is a little tricky. Moving one step in the direction of the mouse requires a change in both x and y and a little extra care...

Normal Distribution

There are other ways to generate nonuniform distributions of random numbers.

But, real people's heights aren't uniformly distributed, there are many more people that are of "average" height than very tall or very short. A uniform distribution won't work here!

- The normal distribution is where numbers cluster around an average value.
- Let's say you want to draw random stick figures on the canvas, each with a hight of between 200 and 300 pixels:

```
let h = random(200, 300);
```

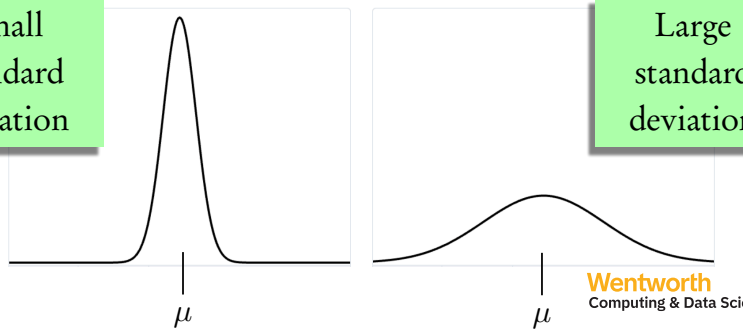

Normal Distribution

- The normal distribution or Gaussian distribution (or bell curve) generates random numbers around a mean, μ, with standard deviation, σ.

The area under the curve is alway 1, just like our probability always added to 100%

Small standard deviation

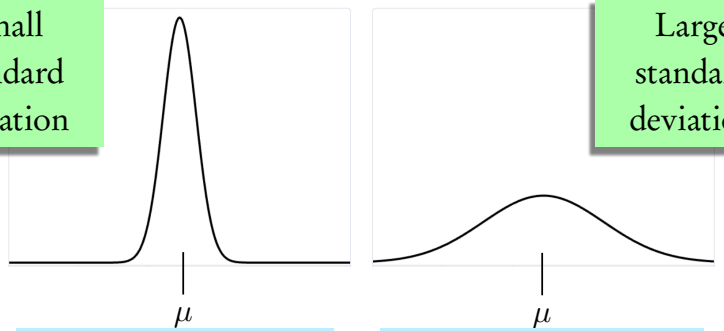
Large standard deviation



Normal Distribution

Small standard deviation

Large standard deviation



With a small standard deviation, all the values pile up around the mean, with few that are far away

With a large standard deviation, the values are more spread out (they deviate more from the mean)

Normal Distribution

- Luckily, we don't have to program a complicated equation to get this behavior.
- p5.js has a built in function to generate numbers in this way:

```
let num = randomGaussian();
```

- By default, this uses a mean of 0 and standard deviation of 1
- This is known as a standard gaussian.

To set our own mean and standard divisions, we provide 2 arguments.

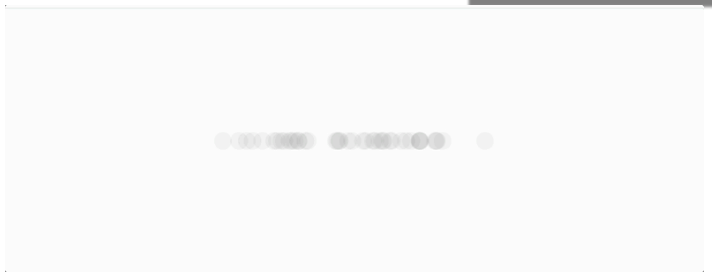
The first is the mean and the second is the standard deviation.

In our sketch, these value are in pixels

Normal Distribution

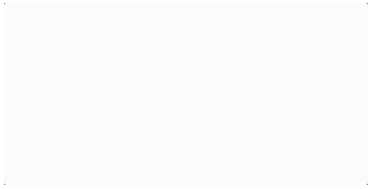
```
function draw(){
  let x = randomGaussian(320, 60);
  noStroke();
  fill(0,10);
  circle(x,120,16);
}
```

Draws a circle every frame in the middle of the sketch at an x position given by the normal distribution.



Exercise

- Create a simulation of paint splatter drawn as a collection of colored dots.
- Most of the paint lands around a central position, but some do get splattered further away.
- Use the normal distribution for multiple properties, like color, size, position.



You'll be generating multiple random numbers and applying them to different properties

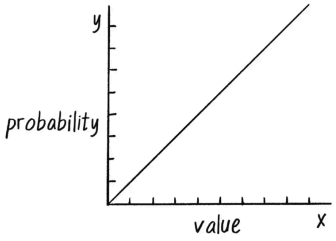
Exercise

- Create a gaussian random walker.
- Set the step size (how far the walker moves in one step) to be generated by the normal distribution.

You can still determine the direction is whatever way you want.

Custom Distributions

- There are ways to create your own custom distributions when you don't want uniform or normal.
- The process is a bit beyond what I want to do in Impact Lab.
- You can create any distribution you want for the specific simulation or purpose.



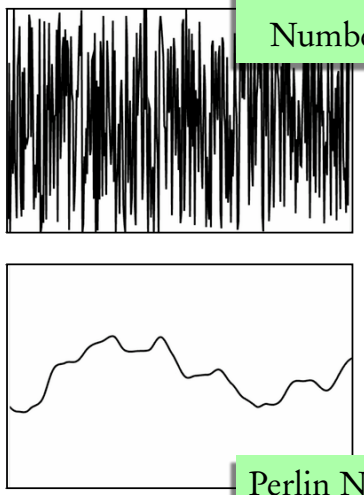
Let's say we wanted a distribution to look like this

Custom Distributions

The accept-reject algorithm can give us that distribution (it's a Monte Carlo algorithm).

Noise and a Smoother Approach

- Good random number generators give you numbers that have no relation to each other.
- But, we sometimes want our randomness to be a bit smoother.
- If we base the next random number on the previous one, we can get numbers that give us interesting effects.



Regular Random Numbers

Perlin Noise

Perlin Noise: History

- In the 1980's while working on the movi… thinking about smooth noise.
- He want to generate textures for CGI pr… computer algorithms instead of artists).
- In 1983, he developed an algorithm that is now named after him: Perlin noise.

He won an Academy Award in technical achievement for the algorithm.

Over the years, others have expanded its capabilities for different use cases.

Some notable variations are Worley noise and simplex noise (also developed by Perlin in 2001)

p5.js `noise()`

- p5.js has a function called **`noise()`** which generates smooth values.
- It is inspired by Perlin noice, but belongs to a class of noice called "Value Noise".
- It can take up to 3 arguments for generating noise in up to 3 dimensions

```
let x = random(0, width);

let x = noise(0, width);
circle(x, 180, 16);
```

It's tempting to just replace random with **`noise()`**, but that's not quite correct...

While the arguments to **`random()`** specify a range, **`noise()`** doesn't work this way, they always give a value from 0-1

Using `noise()`

- 1D Perlin noise can be thought of as a linear sequence of values over time (like the graph previously)

Time	Value
0	0.365
1	0.363
2	0.364
3	0.366

To get a value, we pick a "moment in time" and pass it to the **`noise()`** function.

```
let x = noise(3);
```

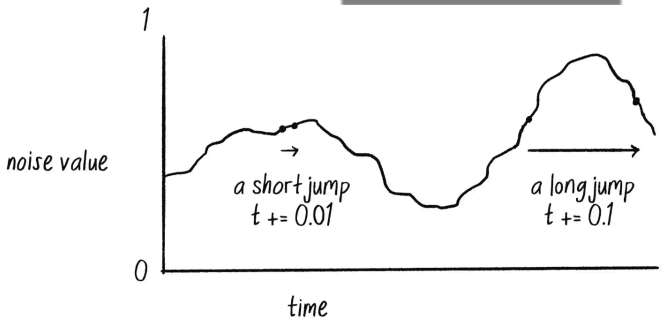
So, to get different values, the argument needs to change in some way (like every frame).

Using `noise()`

```
let t = 0;
function draw(){
  let n = noise(t);
  print(n);
  t += 0.01;
}
```

It's conventional to start with **`t=0`**, but you can start where ever you want.

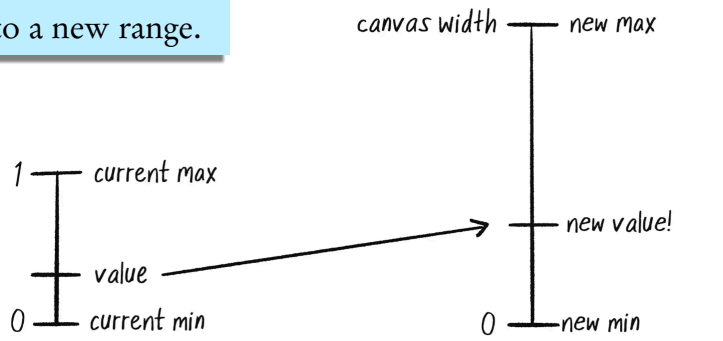
`t` is incremented a little bit each frame, that is our step.



Noise Ranges

Since we only get values from 0-1, p5.js provides us with a way to map the values to a new range.

Say we wanted to map 0-1 to our canvas width:



new value = map(value, current min, current max, new min, new max)

Noise Ranges

Using **map()** to customize the range of the **noise()** output.

```
let t = 0;
function draw(){
  let n = noise(t);
  let x = map(n, 0, 1, 0, width);
  ellipse(x, 180, 16, 16);
  t += 0.01;
}
```

map() takes five arguments:

- The value you want to map
- The value's current range (min and max)
- The new range you want (min and max)

Perlin Walker in X and Y

```
class Walker{
  constructor(){
    this.tx = 0;
    this.ty = 10000;
  }
  step(){
    let nx = noise(tx);
    let ny = noise(ty);
    let x = map(nx, 0, 1, 0, width);
    let y = map(ny, 0, 1, 0, height);
    tx += 0.01;
    ty += 0.01;
  }
```

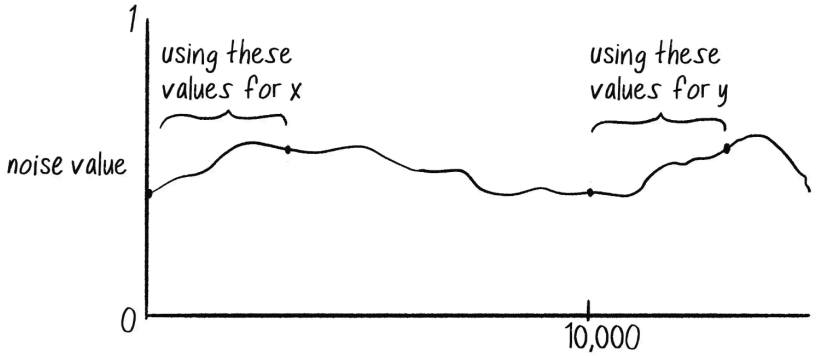
If we let **tx** and **ty** both start at 0, the object will only move diagonally, so we set **ty** to some large value far away from **tx**. See the next slide!

Perlin Walker in X and Y

```
class Walker{
  constructor(){
    this.tx = 0;
    this.ty = 10000;
  }
  step(){
    let nx = noise(tx);
    let ny
    let x
    let y
    tx +=
    ty +=
  }
```

If we let **tx** and **ty** both start at 0, the object will only move diagonally, so we set **ty** to some large value far away from **tx**. See the next slide!

Perlin Walker in X and Y



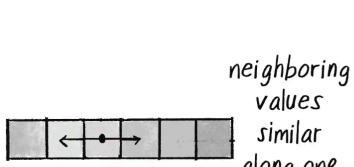
Starting **`ty`** at 10000 ensures that we won't overlap with **`tx`**.

Exercise

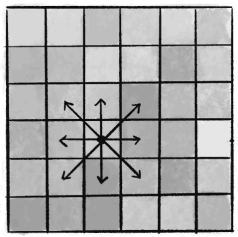
- Take the Perlin Walker and map the result of noise() to the walker's step size instead of its position.

2D Noise

- In 1D noise, the sequence of values are similar to their 2 neighbors.
- 2D noise has more neighbors, but the same holds true in both dimensions:

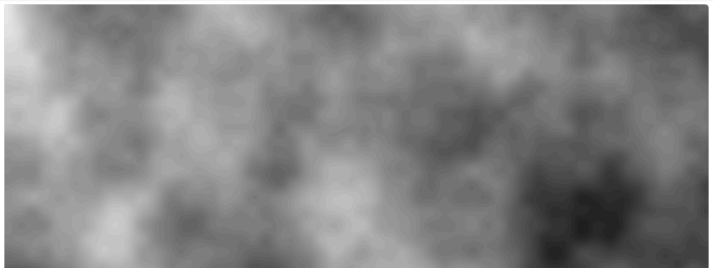


neighboring values similar along one dimension



neighboring values similar along two dimensions

2D Noise



2D noise essentially creates an image, it fills a grid of cells with noise values.

To make this visualization, we can just display a color depending on the value of the noise.

2D Noise: Visualization

```
loadPixels();
for (let x = 0; x < width; x++) {
  for (let y = 0; y < height; y++) {
    let index = (x + y * width) * 4;
    // A random brightness!
    let bright = random(255);
    // Set the red, green, blue, and alpha values.
    pixels[index] = bright;
    pixels[index + 1] = bright;
    pixels[index + 2] = bright;
    pixels[index + 3] = 255;
  }
}
updatePixels();
```

We loop over all pixels, and give them a brightness (here, it's just the regular **`random()`** call).

2D Noise: Visualization

```
loadPixels();
for (let x = 0; x < width; x++) {
  for (let y = 0; y < height; y++) {
    let index = (x + y * width) * 4;
    // A random brightness!
    let bright = random(255);
    // Set the red, green, blue, and alpha values.
    pixels[index] = bright;
    p
    p
    p
  }
}
updatePixels();
```

If you want Perlin brightness instead:

```
let r = noise(x,y);
let bright = map(r, 0, 1, 0, 255);
```

s, and give them a brightness (here, it's just the regular **`random()`** call).

2D Noise: Visualization

```
loadPixels();
for
  fo
    pixels[index] = bright;
    p
    p
    p
  }
}
updatePixels();
```

The problem with this version is that the (**x**,**y**) are incremented by 1, which is too large for the step. Let's try and fix it.

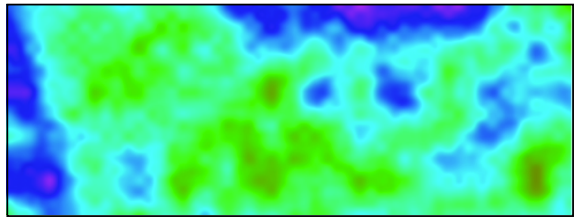
If you want Perlin brightness instead:

```
let r = noise(x,y);
let bright = map(r, 0, 1, 0, 255);
```

s, and give them a brightness (here, it's just the regular **`random()`** call).

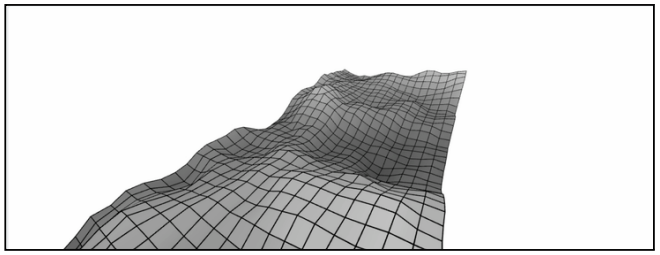
Exercise

- Play with color to see if you can get an interesting look.
- Add a third argument to you **`noise()`** calls to animate the noise effect.



Example

- Use the noise values as an elevation of a landscape.



Perlin noise has been used to create landscapes for many movies and video games!

Next Week

- Arrays and more