Module 3

Procedural Generation

Design Considerations

- We use procedural generation to keep games fresh for the end user
- Letting the computer make things is great, but if we give it too much power, we can end up on the bad sides of an extreme
- Too random, and the user will become confused or the generated content will be nonsensical
- Too tight of restrictions, and the patterns become too obvious to spot
- We want to land in that sweet spot of randomness where the user is having a new experience, but in a way that feels familiar and natural

How do we hit the sweet spot?

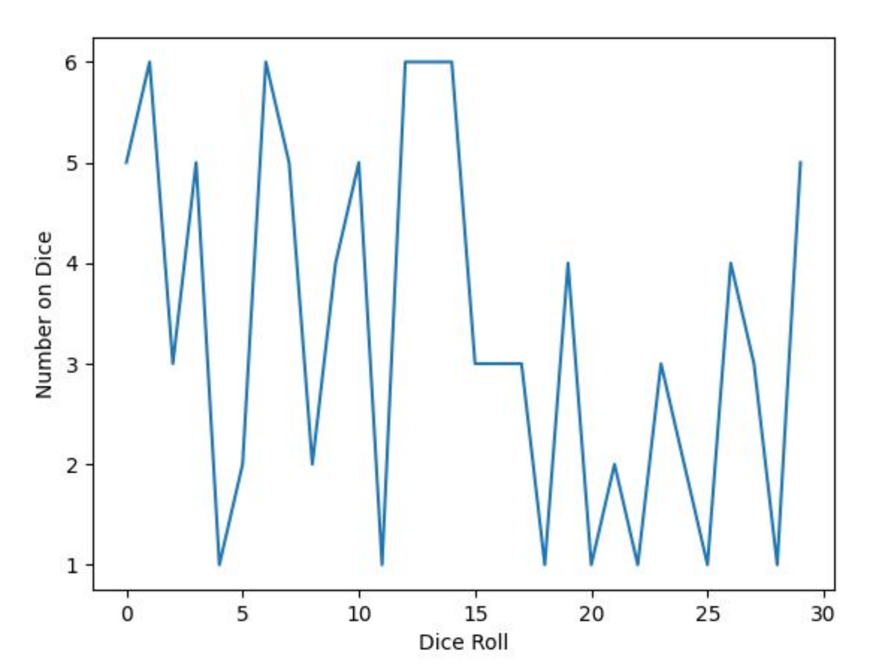
- Play areas: Where players spend their time
- Connectors: the links between areas. Players spend less time here
- How do we randomly generate fun play areas?
- The overwhelming paradigm is to hand make play areas
  - This lets us as developers control how the space operates, and allows the player to recognize familiar patterns
- Where do we use the randomness then?
  - Placement of obstacles/benefits in play areas
  - Terrain, look, feel of play areas
  - What kinds of obstacles/benefits are in play areas
- The bottom line: making new things is hard to do procedurally. It usually better to focus on varying things using some heuristic

Procedural Generation

Heuristics: Noise

What is Noise?

- Random Values
  - A roll of a dice
  - Math.random();
  - Random number from 1 to 10
- Typically quite jagged
- Not very useful on its own



White Noise

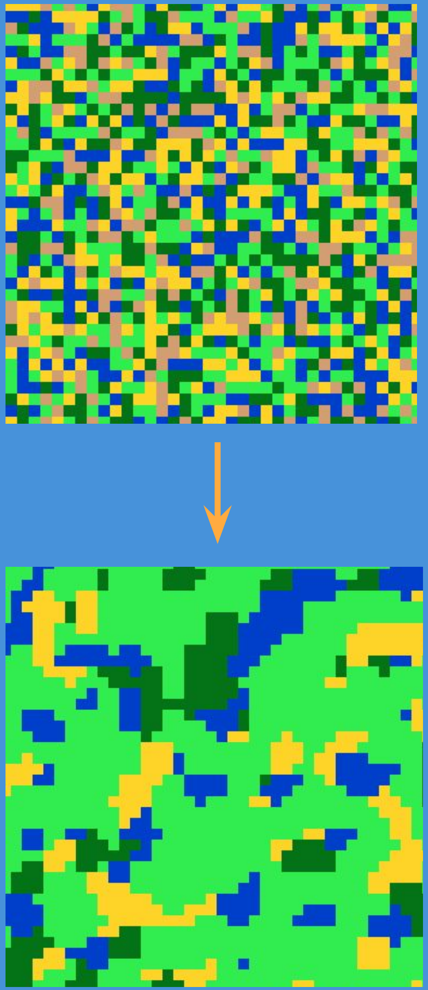
```
// returns a pseudorandom noise value for a given position

float noise(Vec2i vec) {
	Random r = new Random();
	r.setSeed(vec.hashCode());
	return r.nextFloat();
}
```

Procedural Generation

VALUE NOISE

Value Noise

- Smooth white noise by taking the average over neighbors
  - essentially blurring
- Turns white noise into something useful.



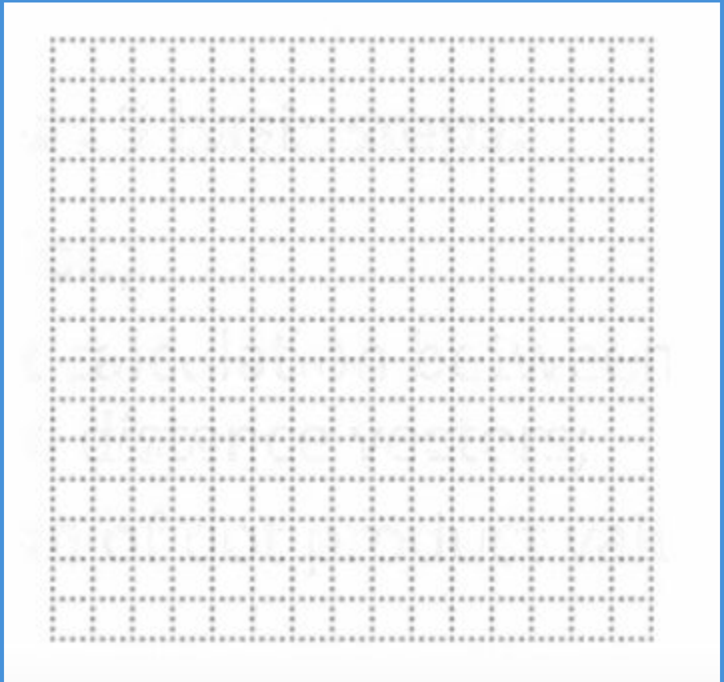
Helper: Grids on Grids

In order to generate noise,

we often layer a grid on top of our

existing coordinate system

like so



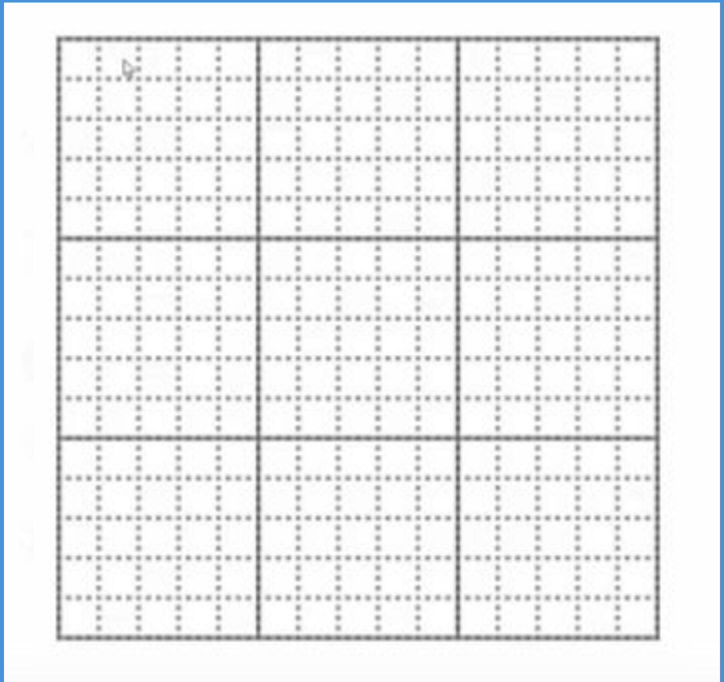
Helper: Grids on Grids

In order to generate noise,

we often layer a grid on top of our

existing coordinate system

like so



Value Noise

```
// returns a weighted average of the 9 points around the Vec2i v float
valueNoise(Vec2i vec){  // In this example, we are sampling with a grid size of 1
	// four corners, each multiplied by 1/16
	corners = ( noise(vec.x - 1, vec.y - 1) + noise(vec.x + 1, vec.y - 1)
		+ noise(vec.x - 1, vec.y + 1) + noise(vec.x + 1, vec.y + 1) ) / 16;
	// four sides, each multiplied by 1/8
	sides = ( noise(vec.x - 1, vec.y) + noise(vec.x + 1, vec.y)
		+ noise(vec.x, vec.y - 1) + noise(vec.x, vec.y + 1) ) / 8;
	// center, multiplied by 1/4
	center = noise(vec.x, vec.y) / 4;
	return center + sides + corners;
}
```

Procedural Generation

INTERPOLATION

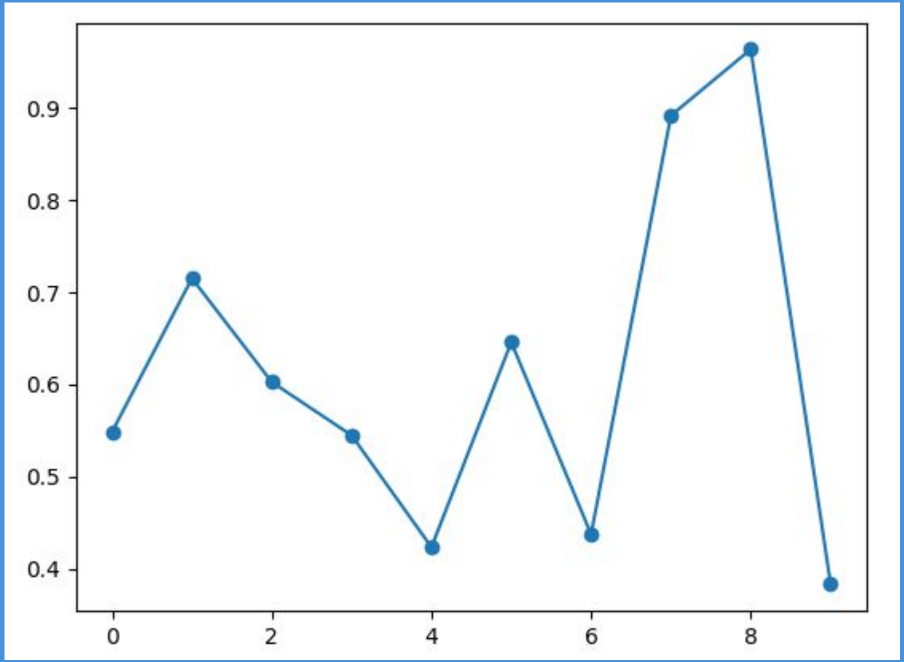
Interpolation

- Most interpolation functions take three arguments
  - A and B the values to interpolate between
  - t, a value between 0 and 1
- When t == 0 the function returns A
- When t == 1 the function returns B
- When 0 < t < 1 the function returns something between A and B

Short python script to see interpolation plots on following slides: [Interpolation.py]()

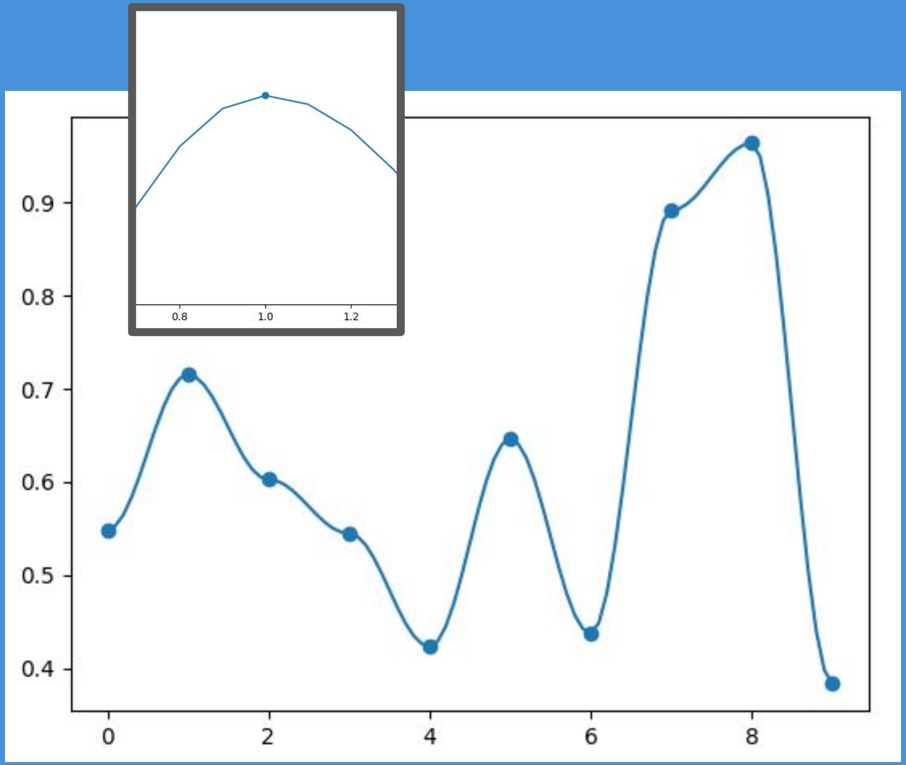
Linear Interpolation

- $f = a(1-t) + bt$
- Fast and jagged



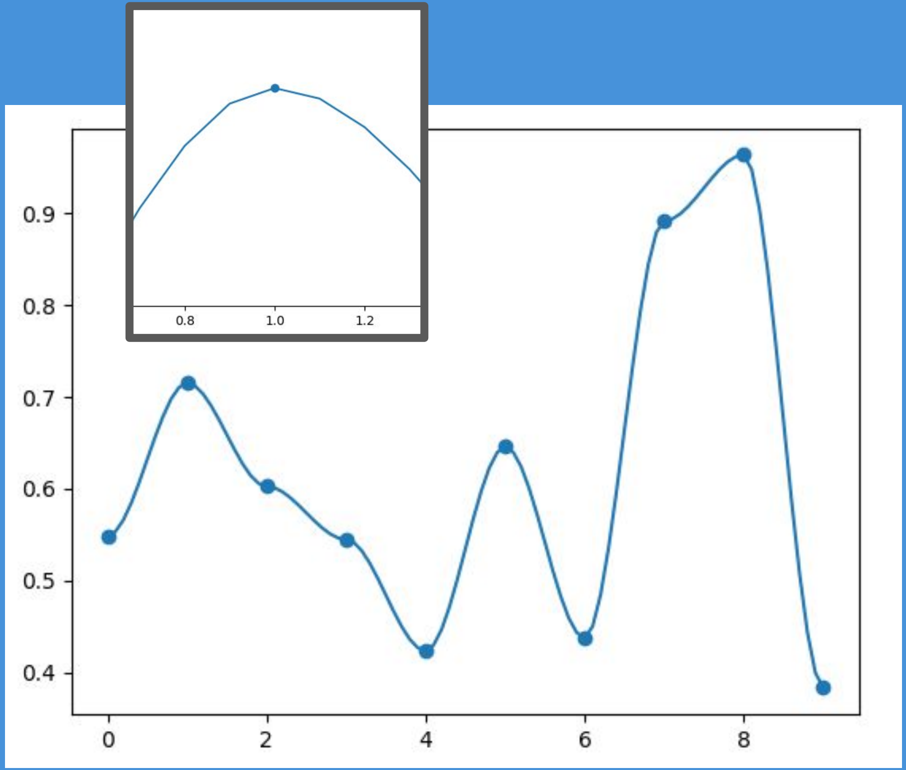
Cosine Interpolation

- t’ = (1 - cos(t * pi)) / 2
- f = a(1-t) + bt
- Slower but much smoother



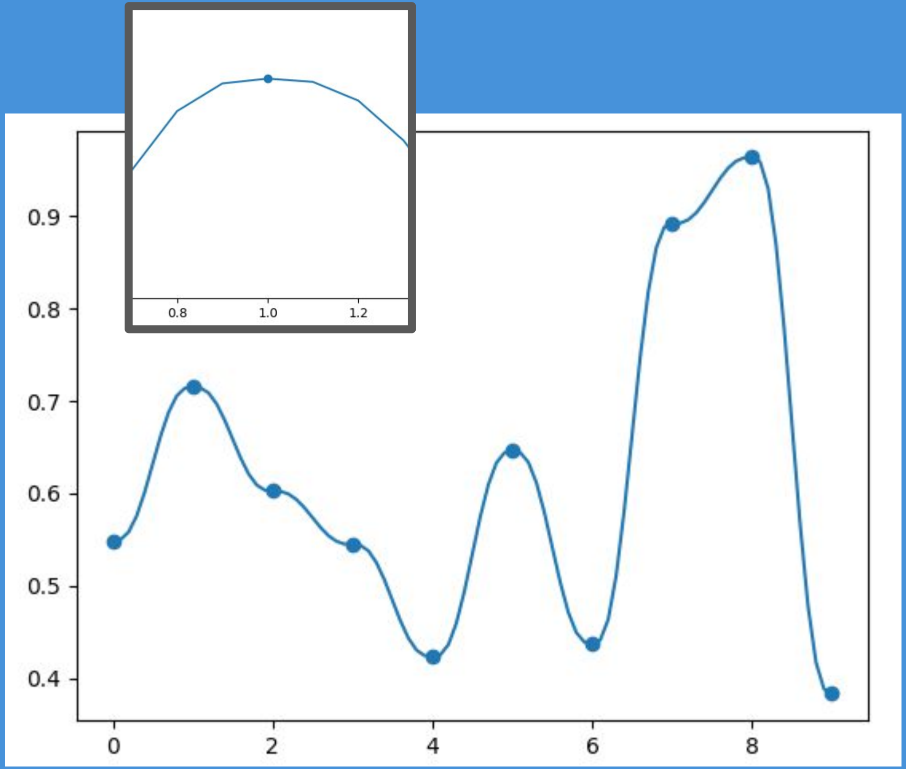
Cubic Interpolation

- $t' = 3t^2 - 2t^3$
- $f = a(1-t) + bt$
- Similar to cosine



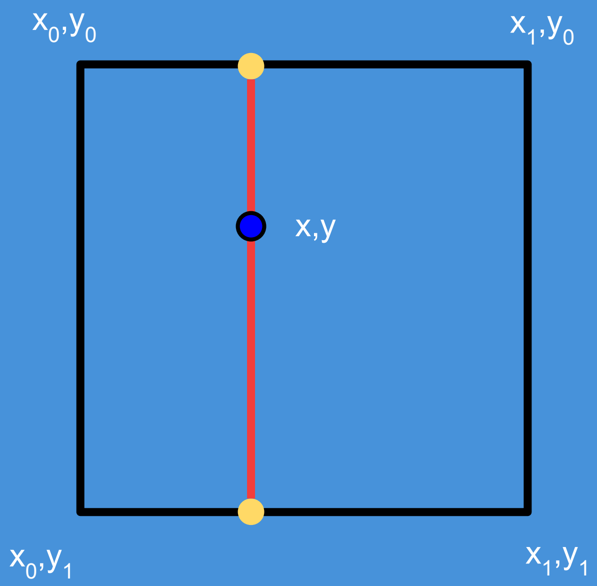
Perlin Interpolation

- $t' = 6t^5 - 15t^4 + 10t^3$
- $f = a(1-t) + bt$
- Slightly slower than cubic
- Super smooth



Interpolating Coordinates

- We want to be able to interpolate any point in the xy-plane
- From the grid of noise values find the square containing the point (x,y)
- Interpolate along opposite edges of the square and then interpolate between edges to get the value at the point.



Interpolating Coordinates

```
// returns the noise interpolated from the four nearest vertices
float interpolatedNoise(Vec2f  vec){
	Vec2i topLeft = Vec2i ((int) vec.x, (int) vec.y);
	Vec2i topRight = Vec2i ((int) vec.x + 1, (int) vec.y);
	Vec2i botLeft = Vec2i ((int) vec.x, (int) vec.y + 1);
	Vec2i botRight = Vec2i (int) vec.x + 1, (int) vec.y + 1);
	float dx = vec.x – ((int) vec.x);
	float dy = vec.y – ((int) vec.y);
	float topNoise = interpolate(valueNoise(topLeft), valueNoise(topRight), dx);
	float botNoise = interpolate(valueNoise(botLeft), valueNoise(botRight), dx);
	return interpolate(topNoise, botNoise, dy);
}
```

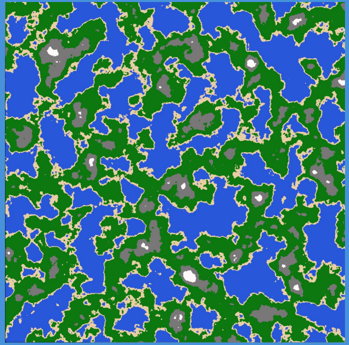
Procedural Generation

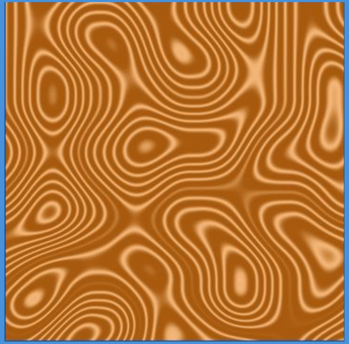
PERLIN NOISE

Perlin Noise

Named for its creator, Ken Perlin

It’s a great way to make smooth, natural noise which can be used to create terrain, cloud patterns, wood grain, and more!

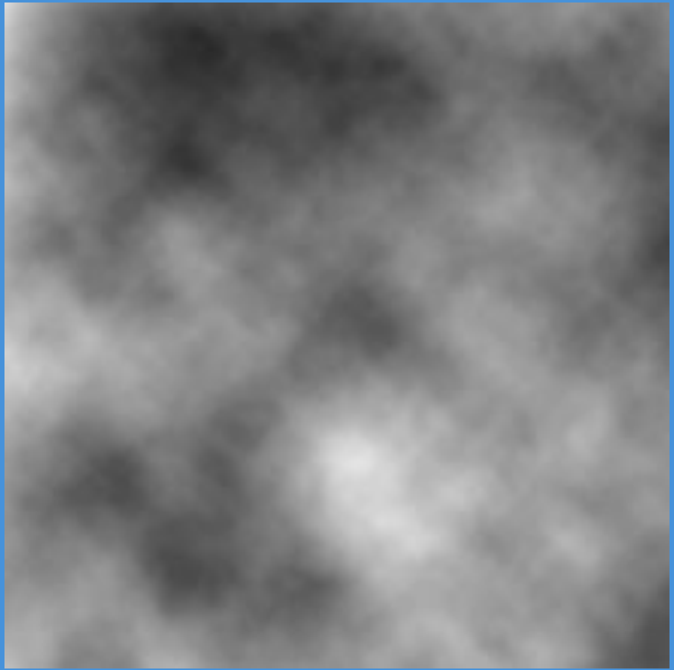




You will probably end up using it just for terrain

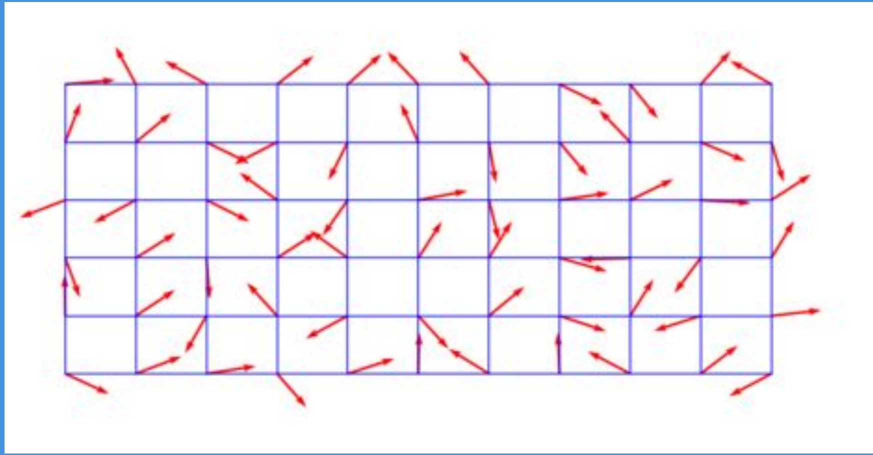
Recap: Value Noise

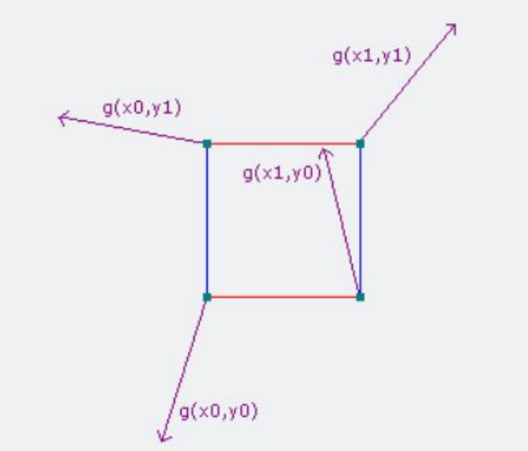
- Smooth white noise by taking an average of neighbors
- Turns white noise into something useful



Perlin Noise

- Assign each vertex a pseudorandom gradient

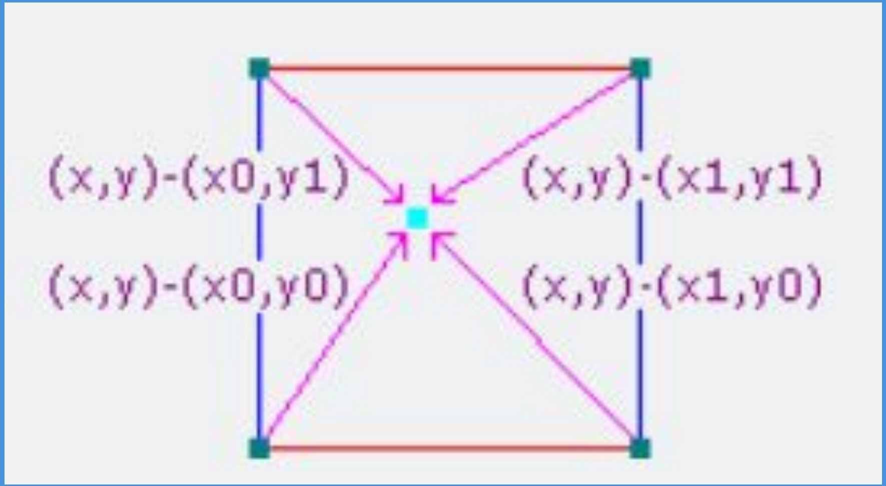





```
Vec2f gradient(Vec2i  vec){
	float theta = noise(vec) * 6.2832;
	return new Vec2f(cos(theta), sin(theta));
}
```

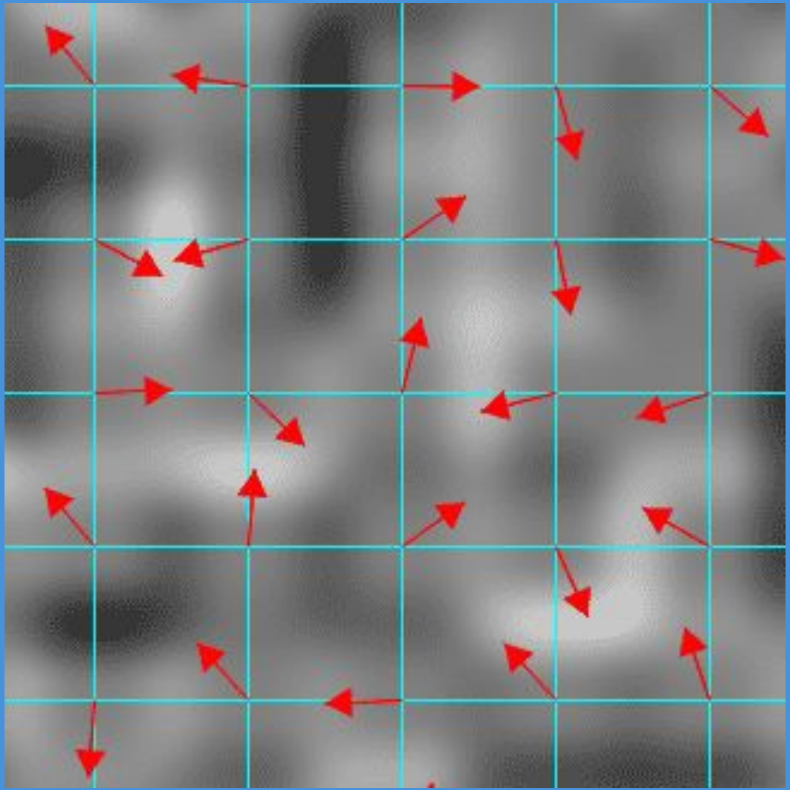
Perlin Noise

- For a point, p=(x,y), we find the noise values of each vertex of the surrounding square for that point.
- The Noise value of each vertex is the dot product of its gradient and the vector from that vertex to the point p
- What do we do with noise value… Interpolate!



Perlin Noise

- Interpolate between the noise values of the four vertices (just like we did with value noise)
- Using linear interpolation is not recommended, it will leave hard edges in your noise. The actual Perlin algorithm uses a special fade function to get the interpolation just right.

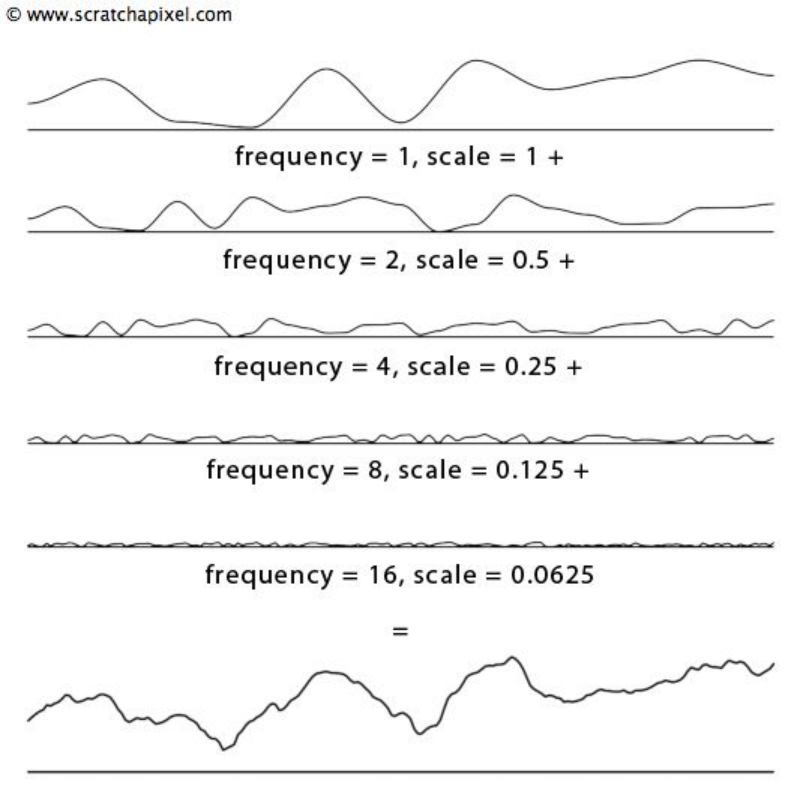


Procedural Generation

ADDING NOISE

Adding Noise Functions

Adding Noise Functions



Actually Using Noise

- Sometimes it can hard to envision what using this noise might look like
- Two basic approaches:
- Create a random structure, then clean it up using rules
    - Noise is great for creating a random structure
- Start with something handmade, and add randomness to it

Extras

- We haven't really talked about how noise layering works
  - Persistence, octaves, ...
  - Here is a website that talks about some of those things: https://pvigier.github.io/2018/06/13/perlin-noise-numpy.html
- Perlin Noise implementation reference: http://adrianb.io/2014/08/09/perlinnoise.html
- Some cool island generation with perlin noise: https://medium.com/@yvanscher/playing-with-perlin-noise-generating-realistic-archipelagos-b59f004d8401