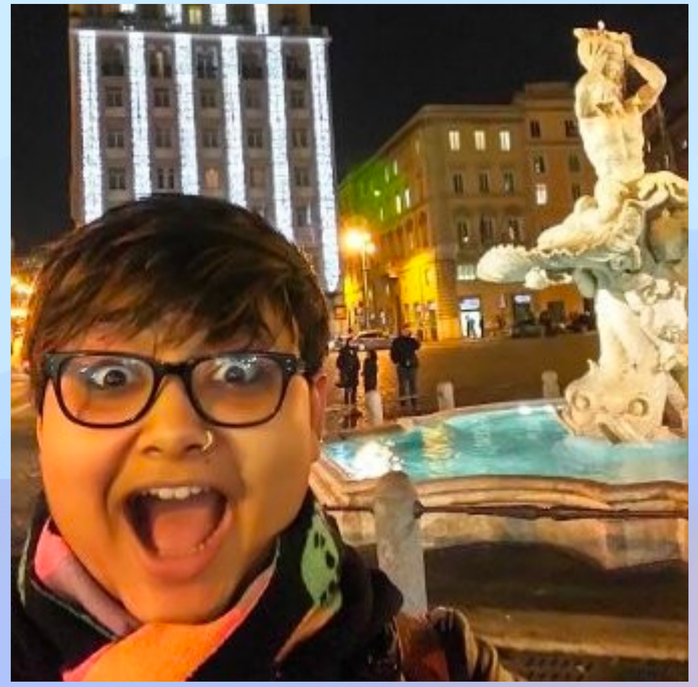
// Creative Coding with Compose

// tasha ramesh

Sep 2, 2022



@TashaRamesh

Co-speaker Spooky

Coding Challenges

PLAY ALL

Is it the gravy train or a train wreck? Watch me take on some viewer submitted coding projects!

Coding Challenge #1: Starfield in Processing

The Coding Train ✓
1.1M views • 6 years ago
CC

Coding Challenge #2: Menger Sponge Fractal

The Coding Train ✓
280K views • 6 years ago

Coding Challenge #3: The Snake Game

The Coding Train ✓
3.2M views • 6 years ago
CC

Coding Challenge #4: Purple Rain in Processing

The Coding Train ✓
1.8M views • 6 years ago
CC

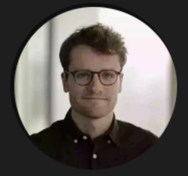
Coding Challenge #5: Space Invaders in JavaScript with...

The Coding Train ✓
542K views • 6 years ago
CC

Frontend Masters

Creative Coding with Canvas & WebGL

Learning Paths: Design to Code • Data Visualization with D3.js Topics: Creative Coding • 3D



Matt DesLauriers
Freelancer

4 hours, 45 minutes CC

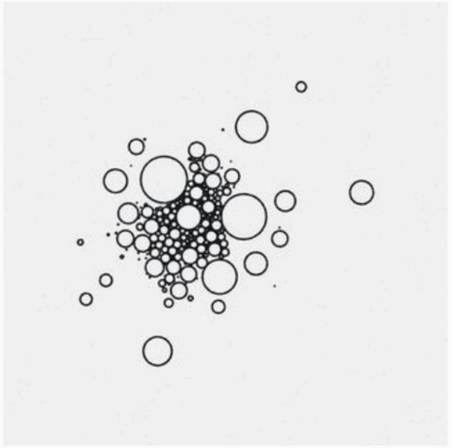
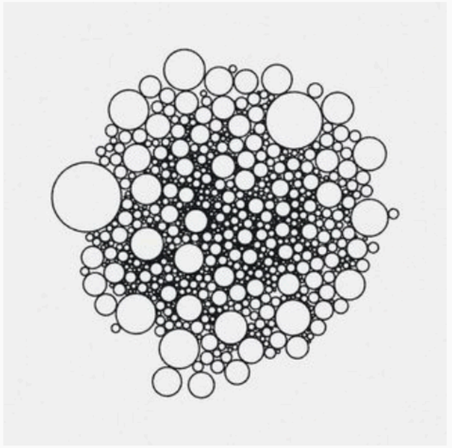
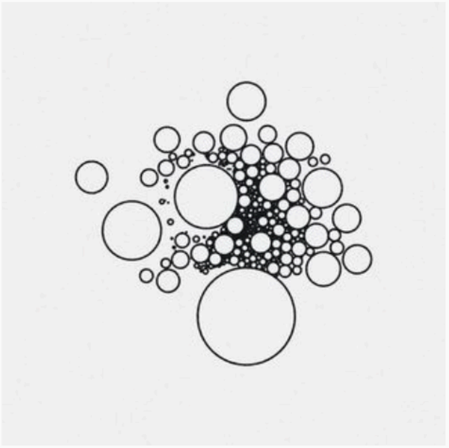
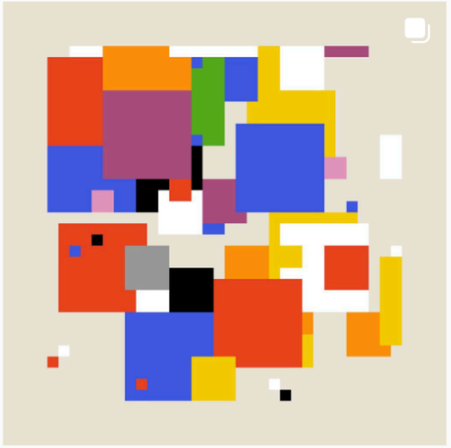
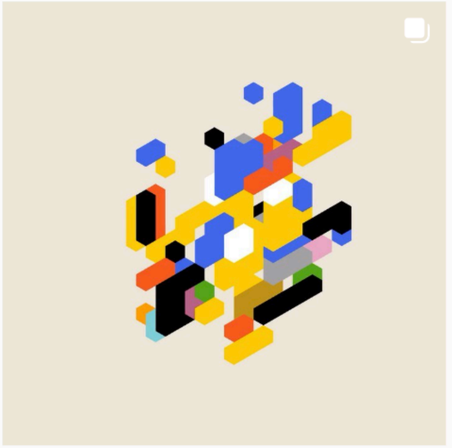
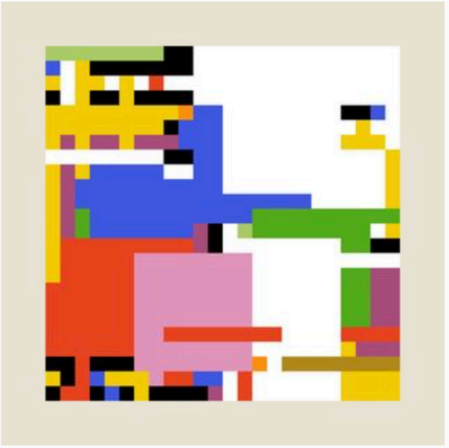
mattdesl_art

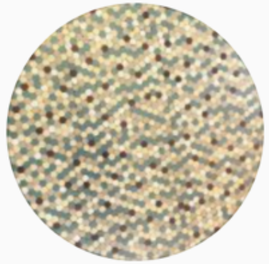
Matt DesLauriers he/him

Artist

Creative coder & generative artist. Works with code and algorithms to create artwork for print, web and installation.

www.vetroeditions.com/products/meridian

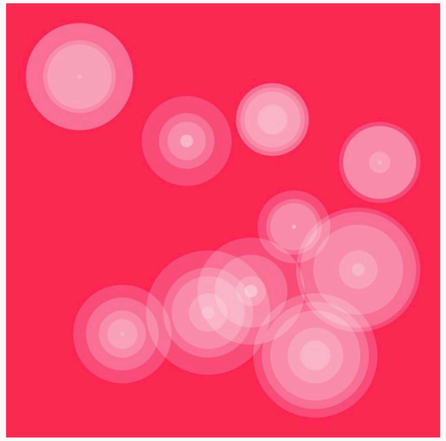







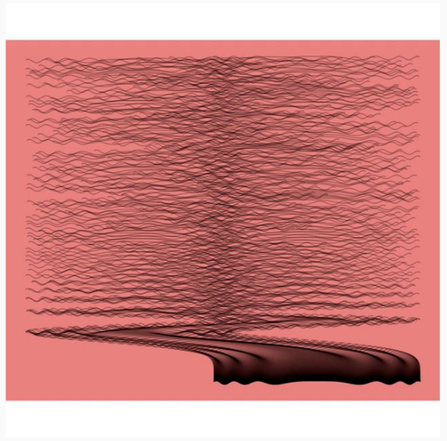


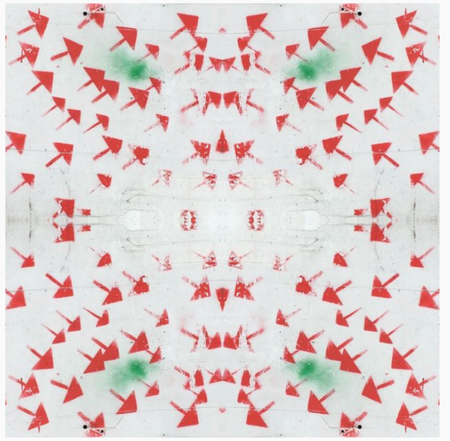
cutterkom

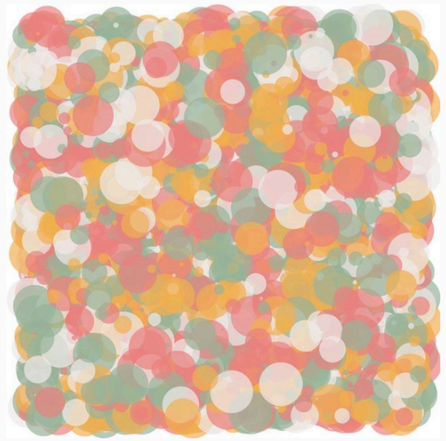
I play with data. #generativeart with #rstats. Create your own with github.com/cutterkom/generativeart

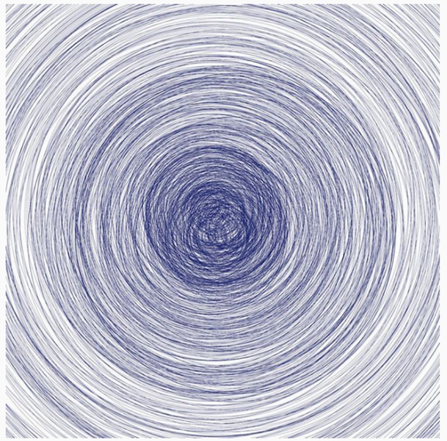
katharinabrunner.de











Generative Art

-using code to draw forms

-rules governing when and how forms change

-artgorithms!

-experiment/explore variations

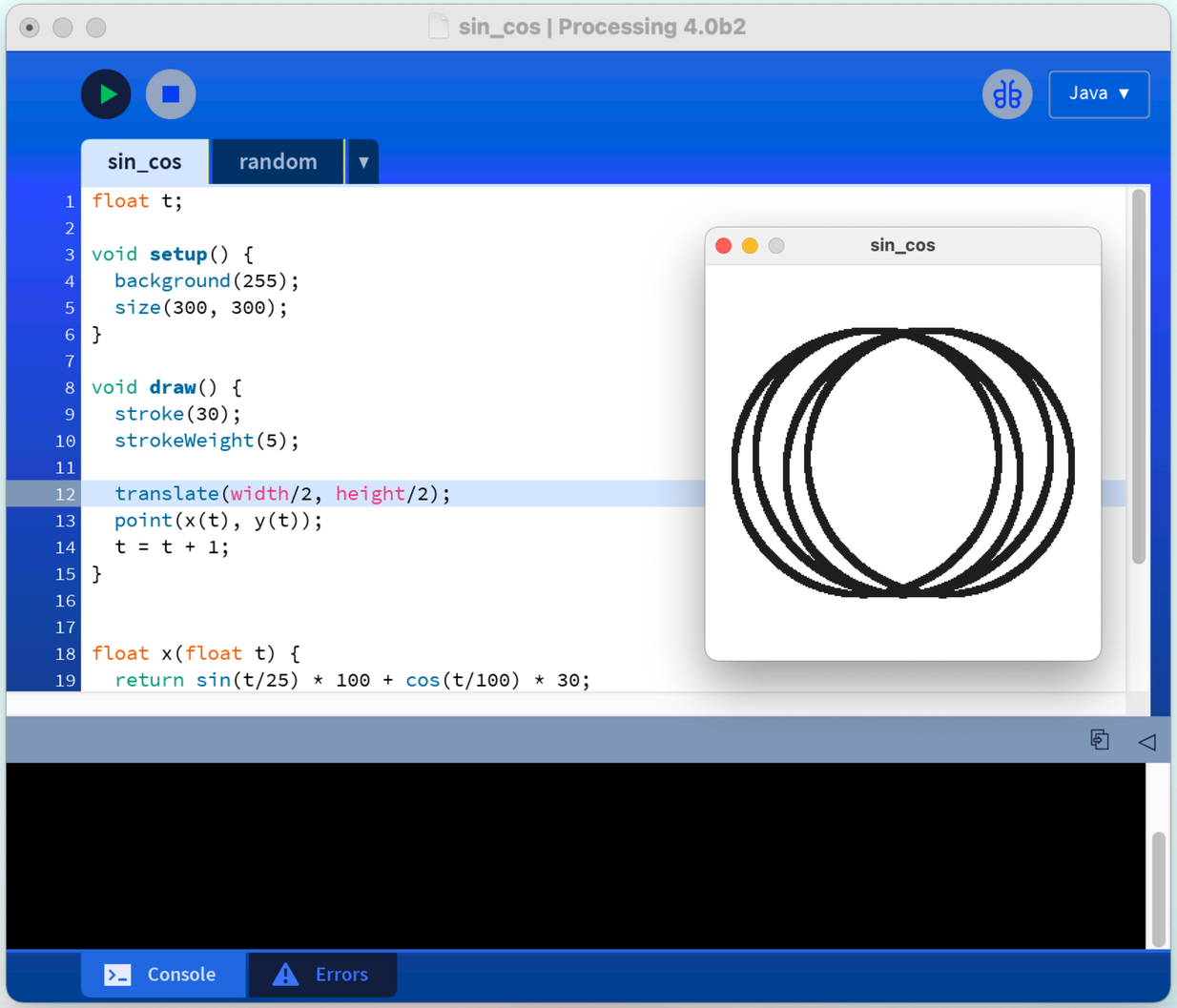
games/
graphics

big data
viz

audio
reactive
visuals

Common Tools

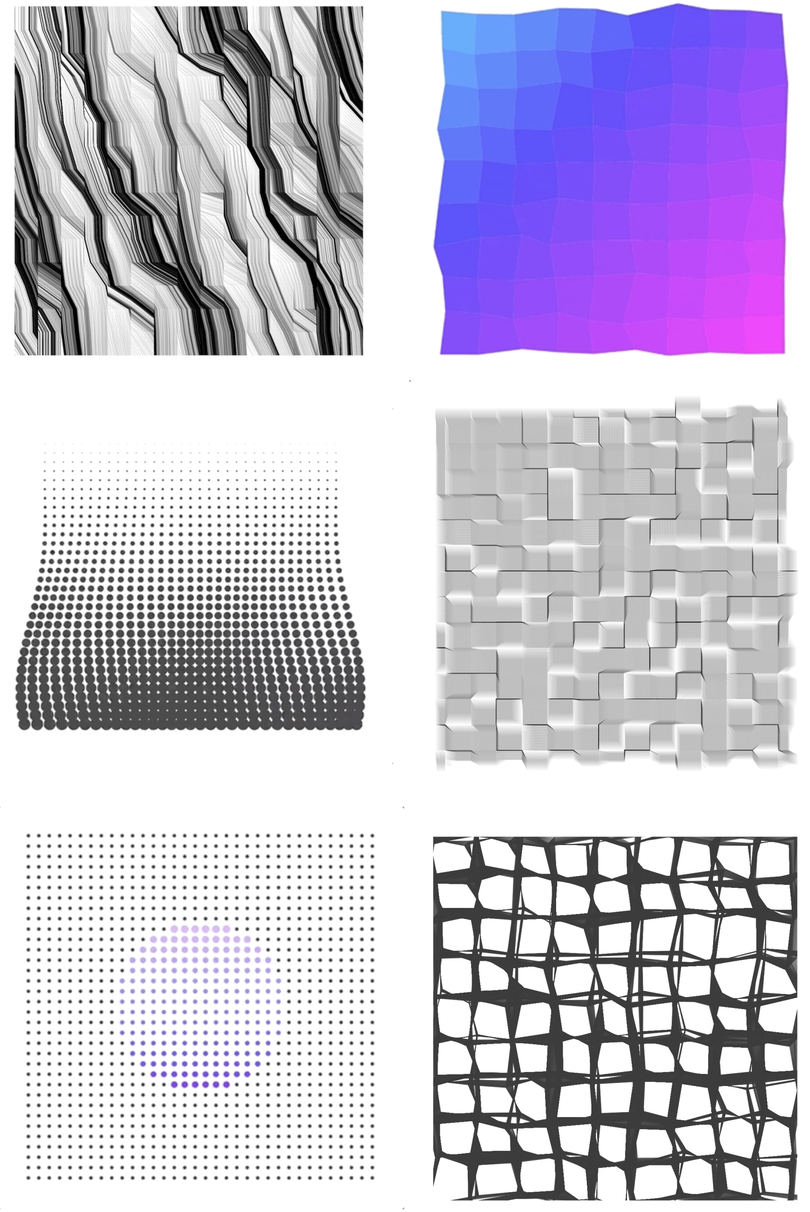
- Processing (Java, Android)
- OPENRNDR (Kotlin)
- p5.js, three.js
- glsl (openGL Shading Language) + shaders…



Processing 4 + Java

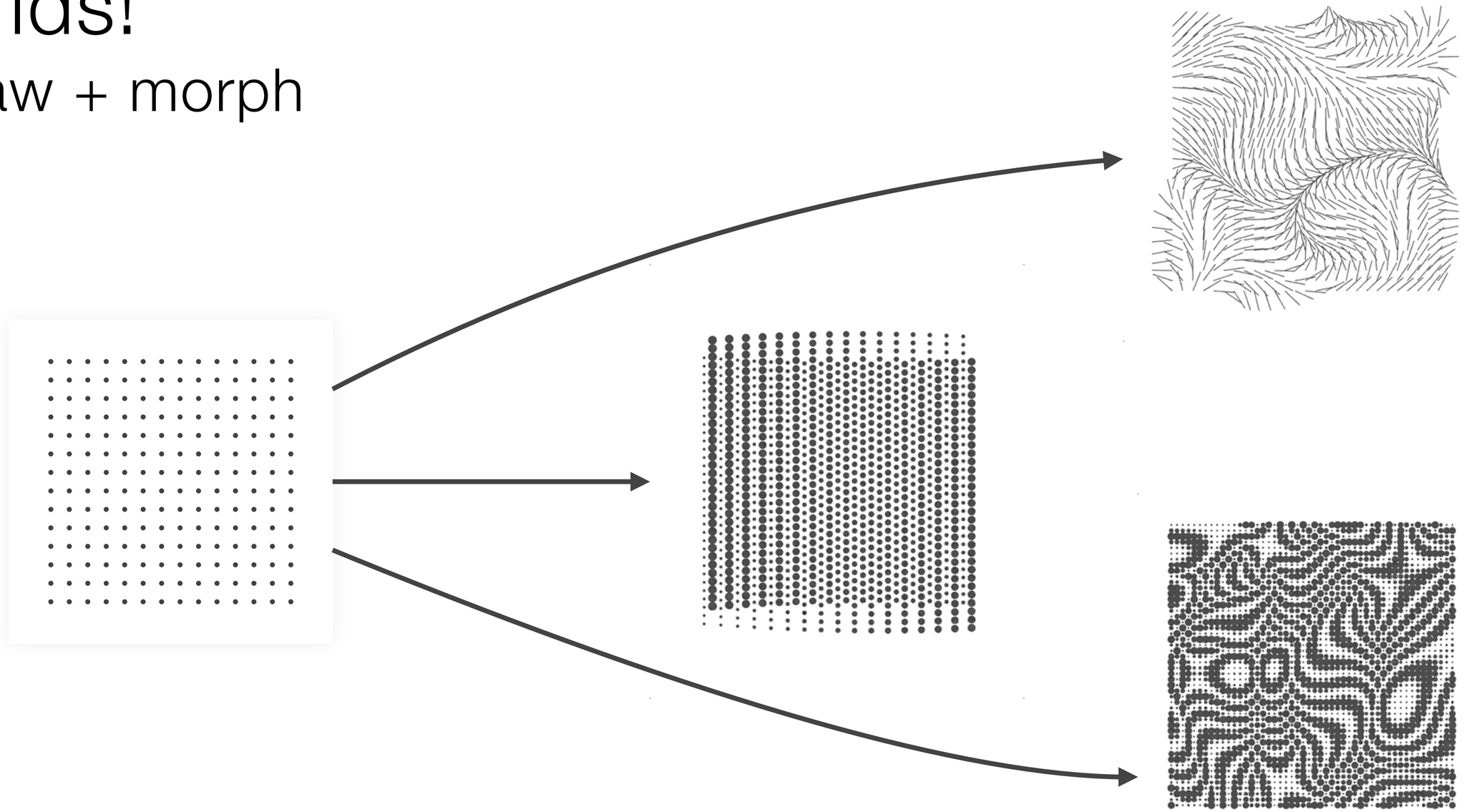
Using Compose

- compose makes graphics declarative, too
- less cumbersome, more efficient compared to Views
- using Compose —>

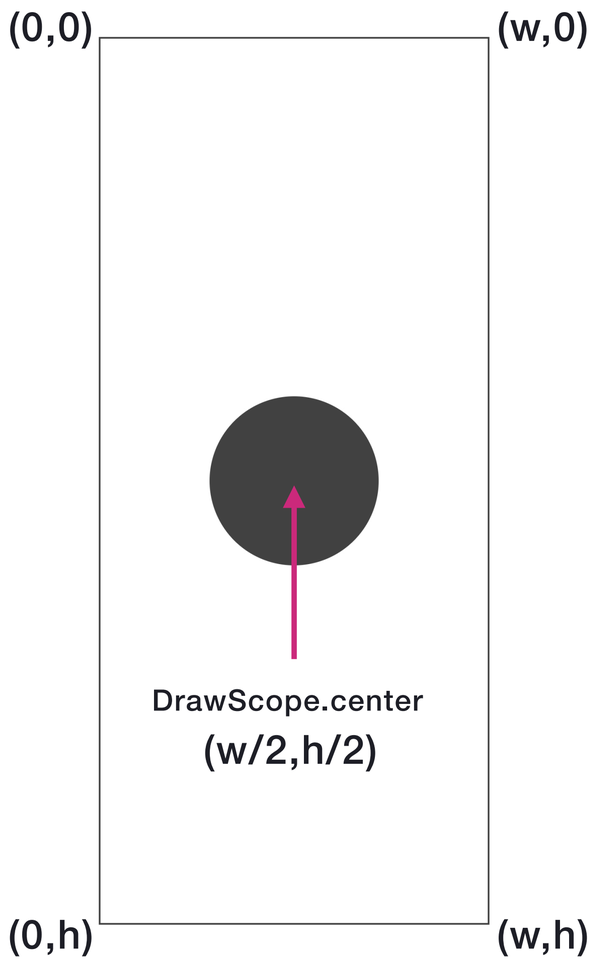


Grids!

draw + morph



Drawing

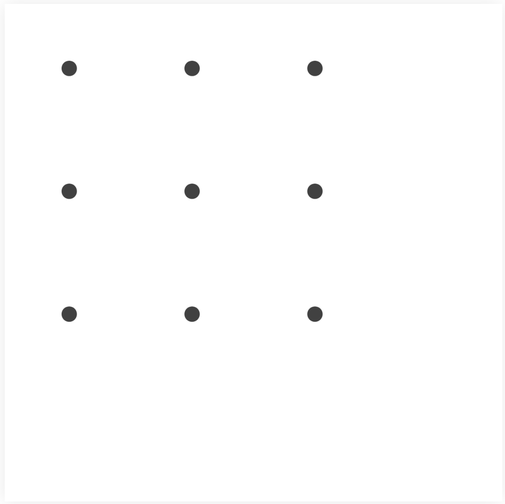



```
Canvas(
  modifier = modifier.fillMaxSize(0.7f)
    .border(1.dp, Color.DarkGray),
  onDraw = { // // this = DrawScope
    // Draws circle at this.center
    drawCircle(
      color = Color.DarkGray,
      radius = 200f
    )
  }
)
```

OR

```
Box(
  modifier = modifier.fillMaxSize(0.7f)
   .border(1.dp, Color.DarkGray)
    .drawBehind { // this = DrawScope
      // Draws circle at this.center
      drawCircle(
        color = Color.DarkGray,
        radius = 200f
      )
    }
)
```

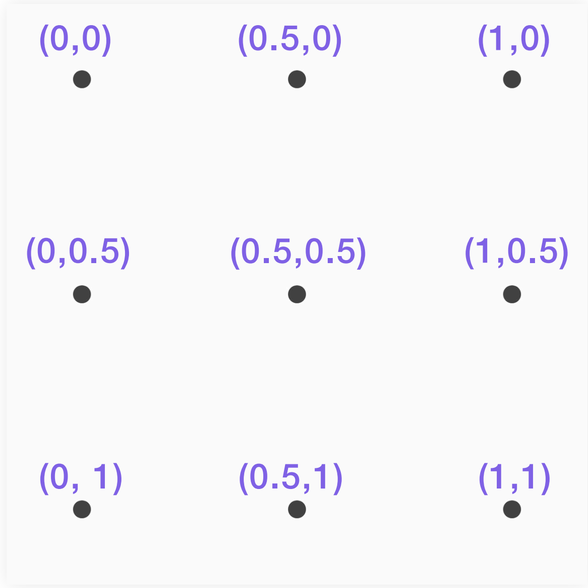
Static Grids



```
Canvas(modifier.aspectRatio(1)) {
  drawGrid(...)
}

private fun DrawScope.drawGrid(...) {
  (0 until dotCount).forEach { x →
    (0 until dotCount).forEach { y →
      val (width, height) = this.size
      drawCircle(
          color = DarkGray,
          radius = 20f,
          center = Offset(
              x * width/dotCount,
              y * height/dotCount
          )
      )
    }
  }
}
```

UV vs XY

Normalization

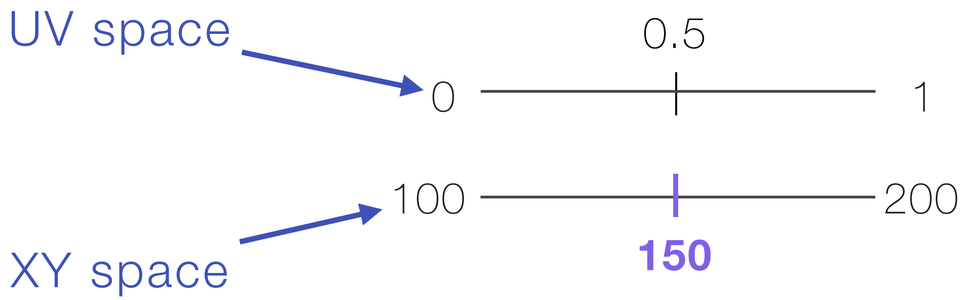



```
private fun DrawScope.drawGrid(...) {
  (0 until dotCount).forEach { x →
    (0 until dotCount).forEach { y →
      // get uv coordinates from 0 to 1
      val u = x / (dotCount - 1)
      val v = y / (dotCount - 1)

      // val xOffset = ??
      // val yOffset = ??

    drawCircle(
        color = DarkGray,
        radius = 20f,
        center = Offset(xOffset, yOffset)
      )
   }
  }
}
```

Convert (0, 1) to an X,Y offset ? 🤔

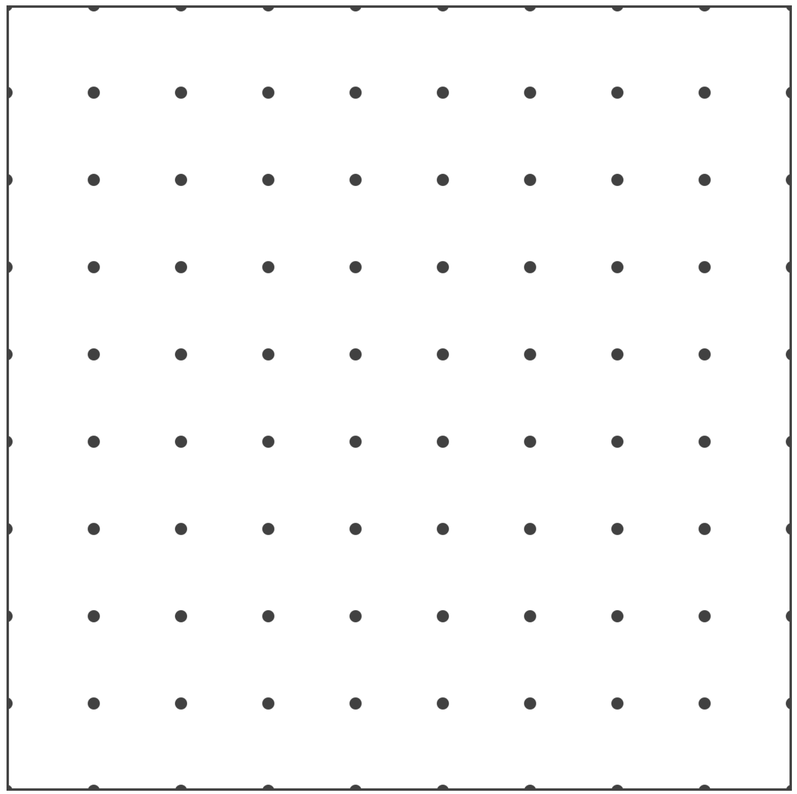
lerp()

Linear interpolation - 0-1 range to another range




```
(max - min) * inputValue + min
```

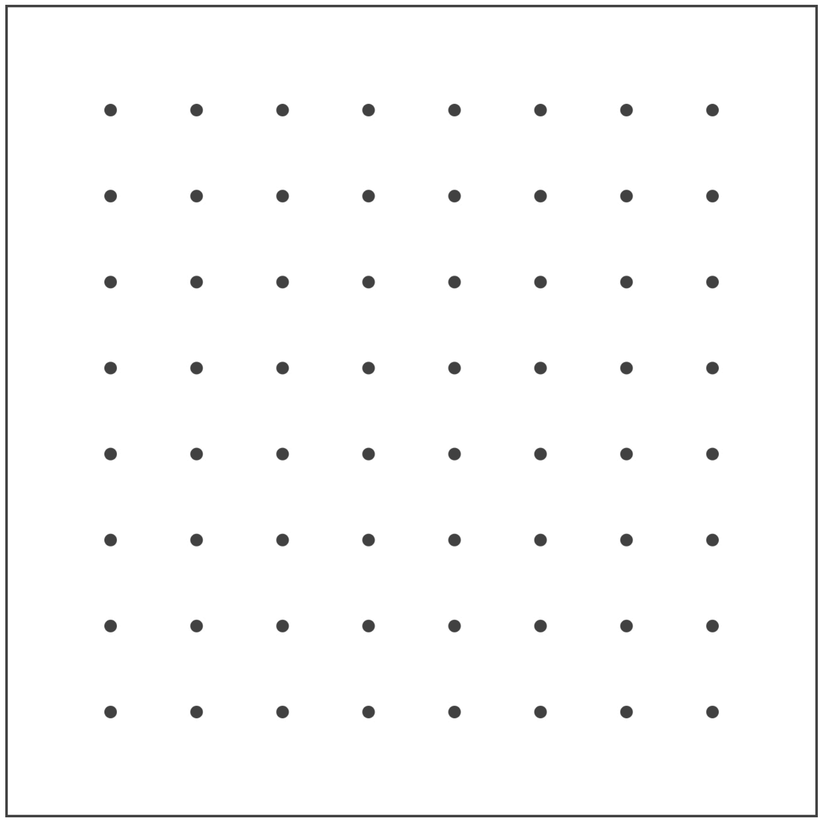
Grids



```
private fun DrawScope.drawGrid(…) {
  (0 until dotCount).forEach { x →
    (0 until dotCount).forEach { y →
      // get uv coordinates
      val u = x / (dotCount - 1)
      val v = y / (dotCount - 1)

      // lerp u/v between 0 & w/h
      val xOffset = lerp(u, 0f, width)
      val yOffset = lerp(v, 0f, height)

      drawCircle(
        center = Offset(xOffset, yOffset)
        ...
      )
    }
  }
}
```

Grids

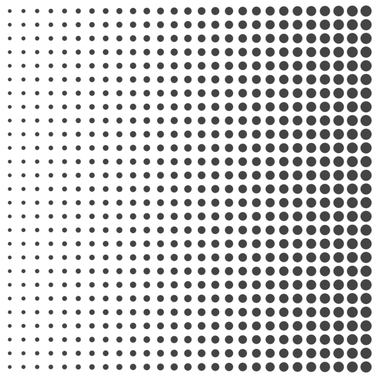
With padding



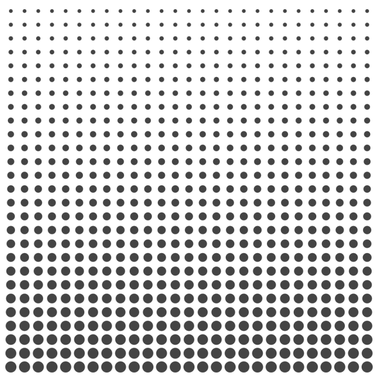
```
Canvas(
    modifier = modifier
        .fillMaxSize(0.9f)
        .aspectRatio(1f)
        .border(1.dp, Color.DarkGray)
        .padding(Padding)
) {
    val (width, height) = this.size
    drawGrid(. . .)
}
```

Dot Size Variations

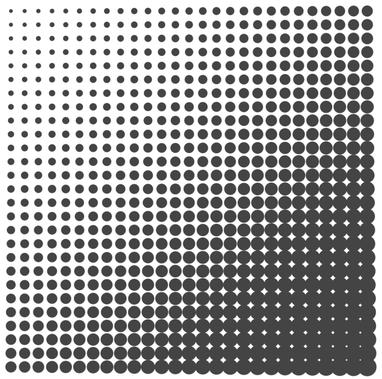
change **dot size** based on **position**



```
radius = lerp(
 value = u,
 min =  5f,
 max =  15
)
```

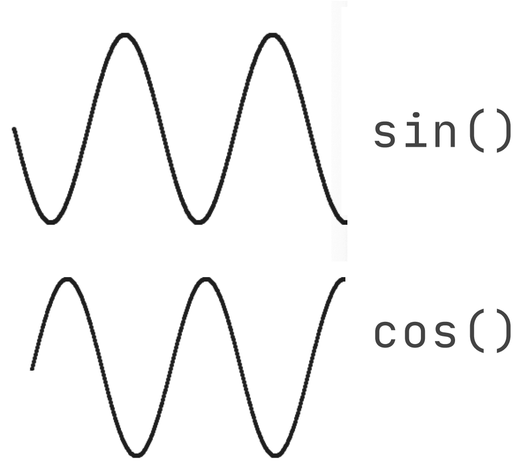


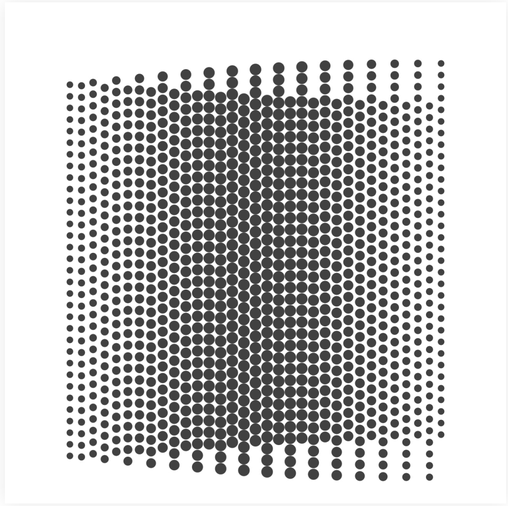
```
radius = lerp(
 value = v,
 min = 5f,
 max = 15f
)
```

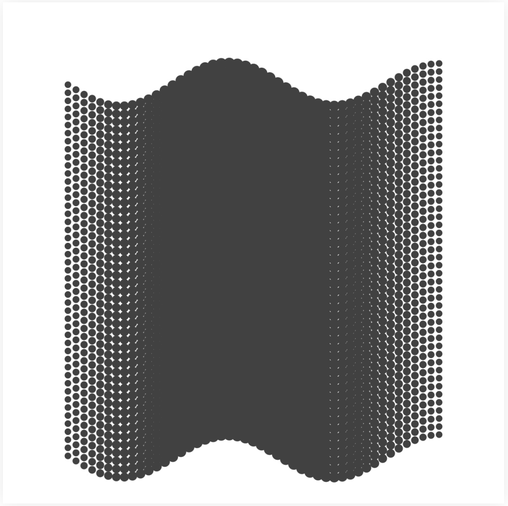


```
radius = lerp(
 value = u + v,
 min = 5f,
 max = 15f
)
```

- `sin()` and `cos()` repeats from `-1` to `1`
- perfect for wavy **positions**!
- need some way to “**map**” `-1` to `1` to x or y offsets 🤔

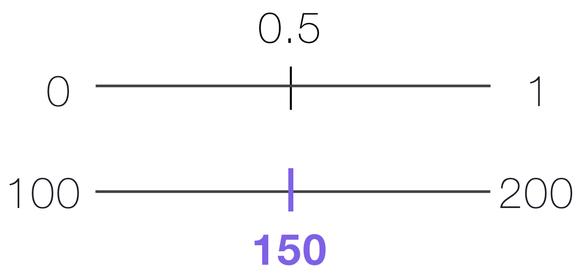






Linear interpolation

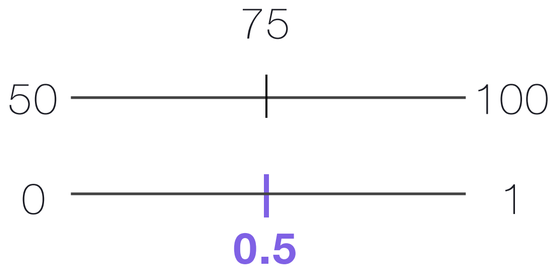
lerp()




```
(max - min) * inputValue + min
```

Normalization

norm()

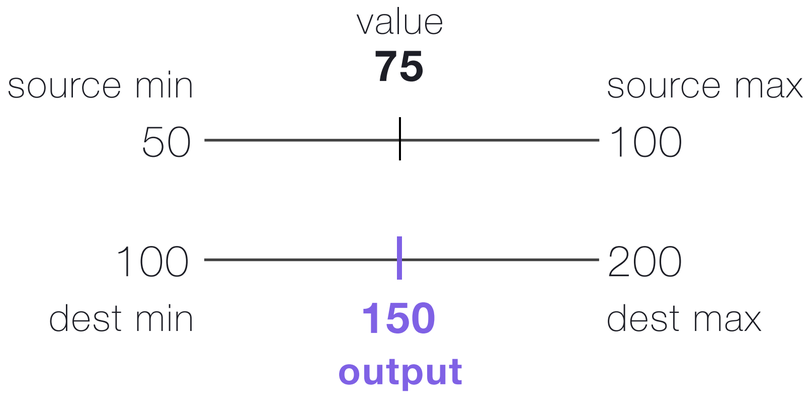



```
(inputValue - min) / (max - min)
```

map() = lerp(norm())

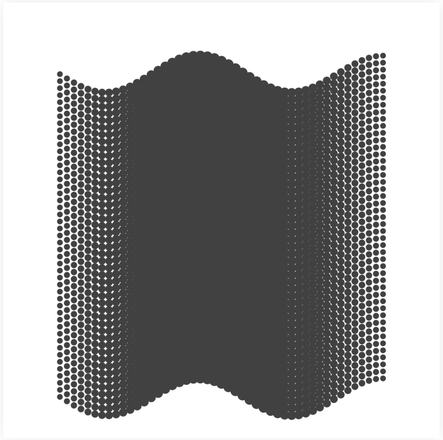
convert between ranges!

```
map() = lerp(
  inputValue = norm(
   inputValue,
   sourceMin,
   sourceMax
  ),
  destMin,
  destMax
)
```

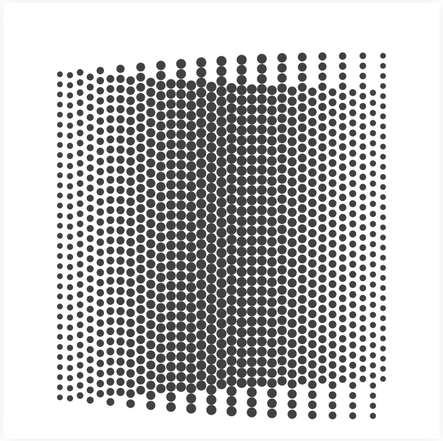


Grid Variation

From: sin/cos(θ) = [-1, 1] **To: [customMin, customMax]**



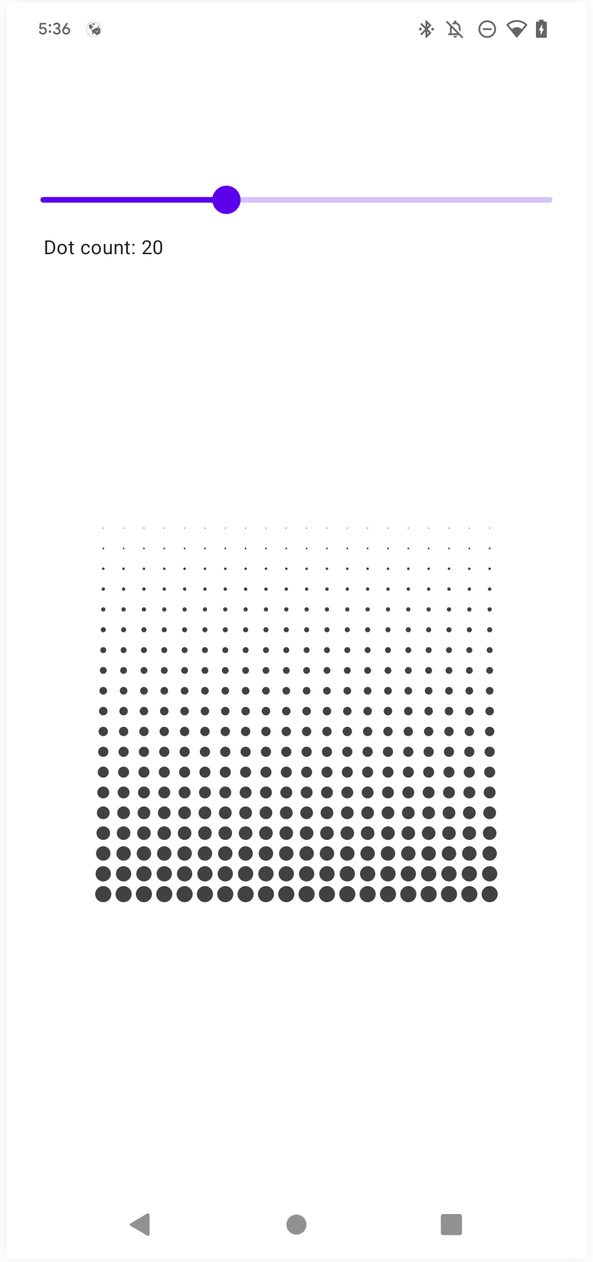
```
radius = map(sin(u * PI),-1f,1f,5f,15f)

center = Offset(
 x = xOffset,
 y = yOffset + map(sin(u * 300f),-1f,1f,-10f,100f)
)
```



```
radius = map(cos(u * 300f),-1f,1f,5f,15f)

center = Offset(
 x = xOffset,
 y = yOffset + map(sin(u * 300f),-1f,1f,-10f,100f)
)
```

Capturing Images

- can take screenshots and crop!
- too tedious; something quicker?
- capturing only the relevant composable?



Capturing

Capture specific composables

- Save Bitmaps from Composables?
- API 28: `PixelCopy!`
- < API 28: wrap View and use `View.drawToBitmap()`

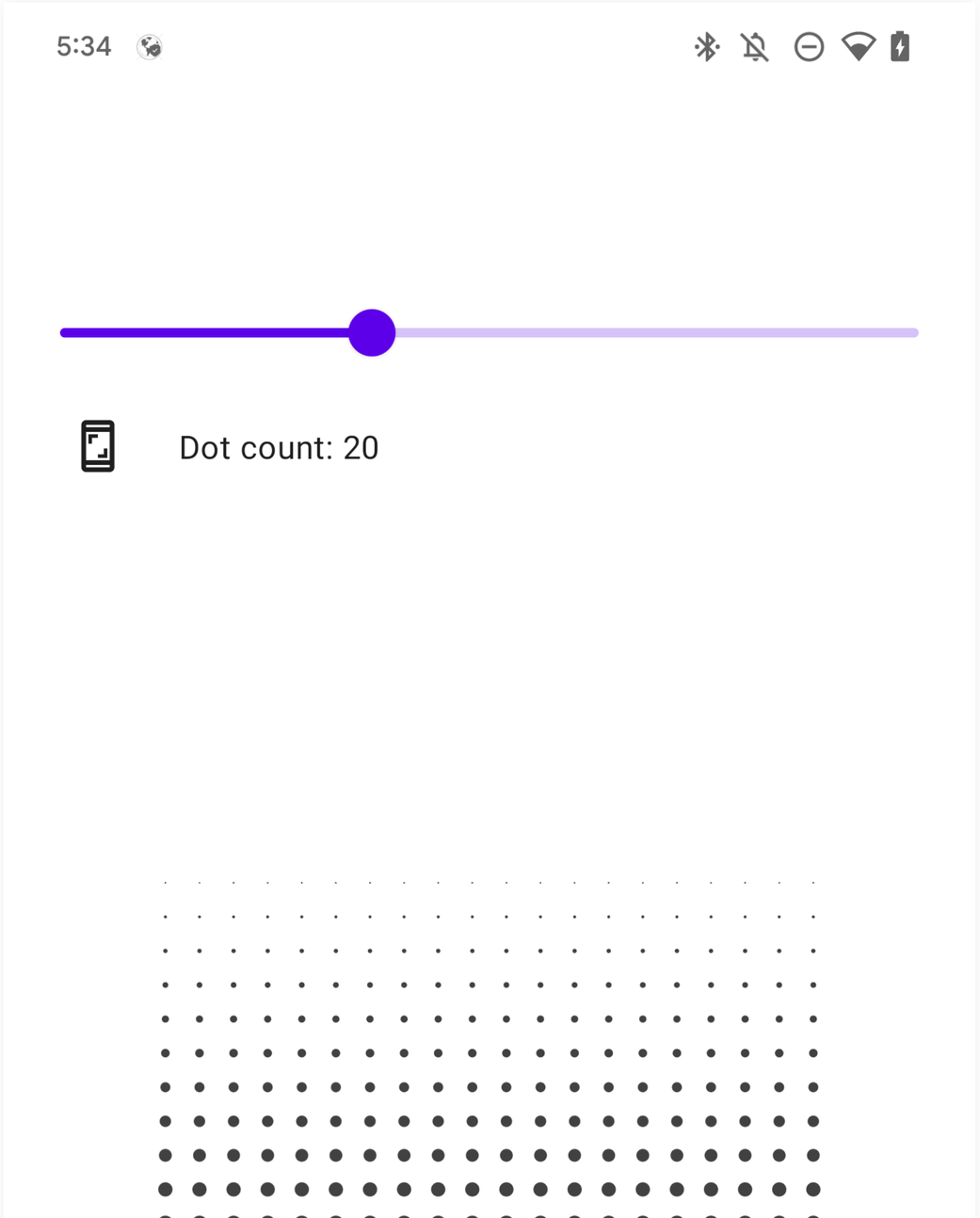
```
val bitmap = Bitmap.createBitmap(
  /* width = */ width,
  /* height = */ height,
  /* config = */ Bitmap.Config.ARGB_8888
)

// API Level 28
PixelCopy.request(
  /* source = */ context.getActivityWindow(),
  /* srcRect = */ Rect(rect),
  /* dest = */ bitmap,
  /* listener = */ { copyResult →
    if (copyResult == PixelCopy.SUCCESS) {
      // use the bitmap
      ...
    } else {
      // Handle failure
    }
  },
  /* listenerThread = */ Handler(getMainLooper())
)
```

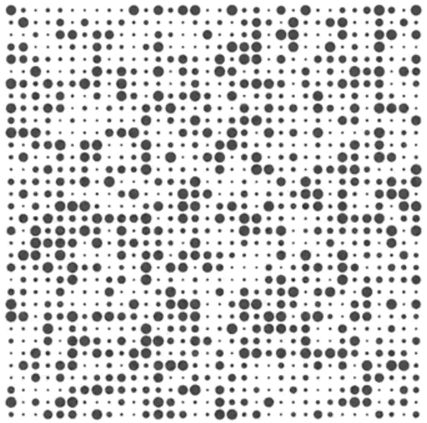
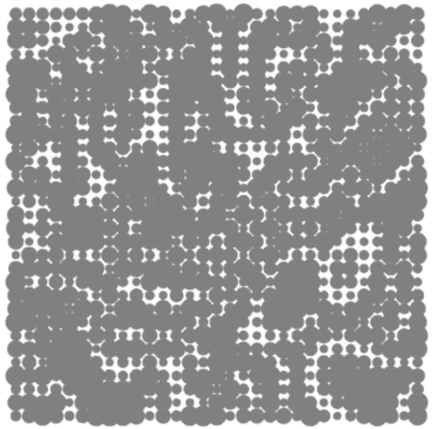
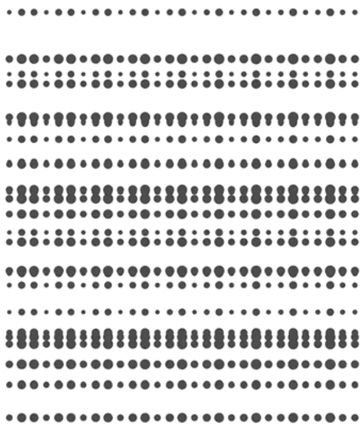
```
Canvas(
 Modifier
  .onGloballyPositioned { layoutCoords →
      // Get size
      layoutCoordinates.size

      // Get bounds in window for rect
      layoutCoordinates.boundsInWindow()
    }
)
```

```
val bitmap = Bitmap.createBitmap(
  /* width = */ width,
  /* height = */height,
  /* config = */Bitmap.Config.ARGB_8888
)

// API Level 28
PixelCopy.request(
  /* source = */ ...,
  /* srcRect = */ Rect(rect),
  /* dest = */ bitmap,
  /* listener = */ { copyResult →
    if (copyResult == PixelCopy.SUCCESS) {
      // use the bitmap
      ...
    } else {
      // Handle failure
    }
  },
  /* listenerThread = */ ...
)
```

Capturing Images

add a button to capture the grid composable only



Movement

Static Canvas

- right now `Canvas` is static
- thinking in terms of a draw loop in the canvas like Processing

```
// setup() runs once
void setup() {
  size(640, 360);
}

// draw() runs continuously
void draw() {
  translate(width/2, height/2);
  point(0, sin(t));
  // increment values to animate
  t++;
}
```

Draw Loops

Processing vs Compose

```
// Processing - Java
void draw() {
  translate(width/2, height/2);
  point(0, sin(t/10) * 50);
  t++;
}
```

```
// Compose!
+Sketch(
  modifier ..
) { t → // t increments somehwere?!
  translate(size.width/2f, size.height/2f) {
    drawCircle(
      center = Offset(0f, sin(t/10) * 50f),
      …
    )
  }
}
```

Sketch

Get a quick draw-loop using `AnimationState`

```
val time = remember { AnimationState(0f) }

LaunchedEffect(Unit) {
  while (isActive) {
    time.animateTo(
      targetValue = time.value + speed,
      animationSpec = animationSpec,
      sequentialAnimation = true
    )
  }
}
```

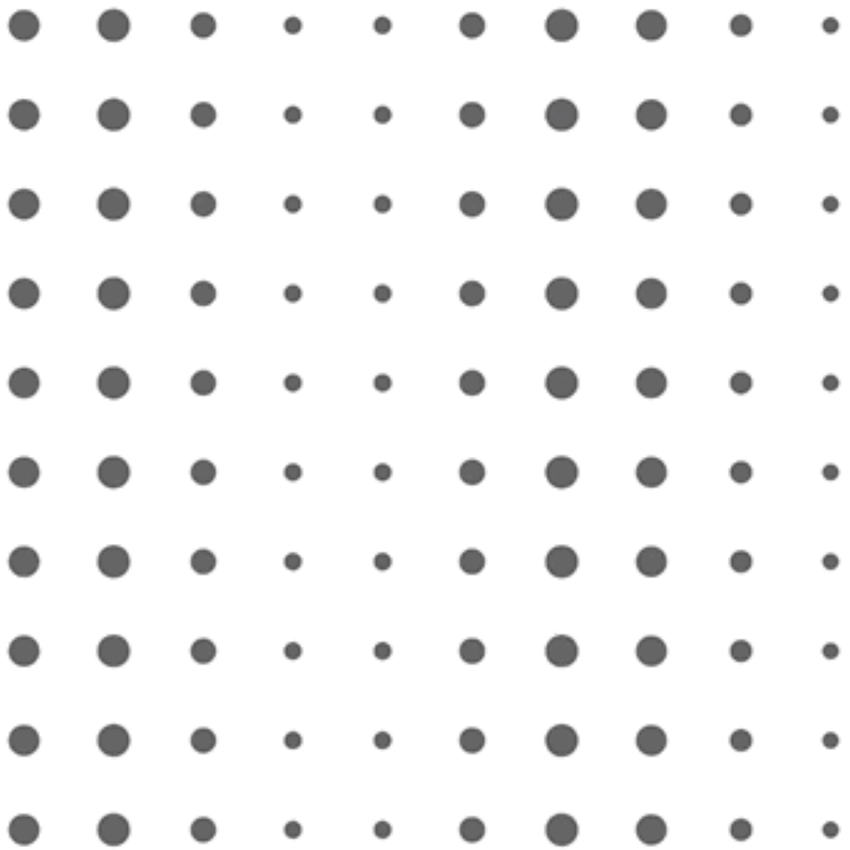
Sketch

Get a quick draw-loop using `AnimationState`

Wrap around a `Canvas`

```
val Spec = tween(
    5000, 50, easing = LinearEasing
)

@Composable
fun Sketch(
    speed: Float = 1f,
    animationSpec: AnimationSpec<Float> = Spec,
    onDraw: DrawScope.(Float) → Unit
) {
    val time = remember { AnimationState(0f) }

    LaunchedEffect(Unit) {
      while (isActive) {
        time.animateTo(
          targetValue = time.value + speed,
          animationSpec = animationSpec,
          sequentialAnimation = true
        )
      }
    }

    Canvas(...) {
      onDraw(time.value)
    }
}
```

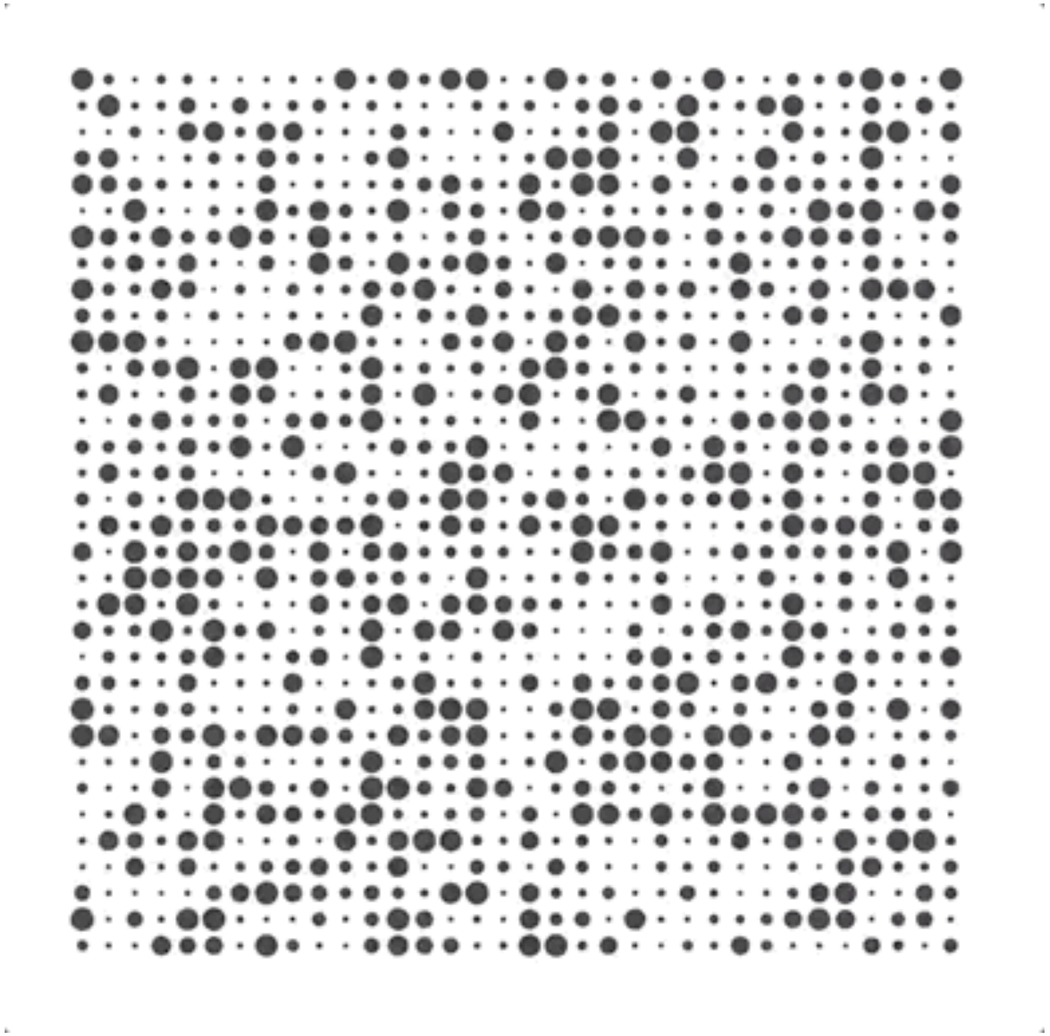
Animated Grid



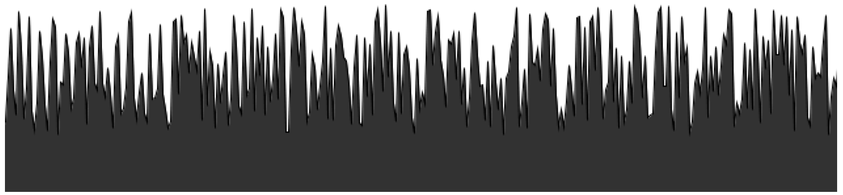
```
+Sketch(
-Canvas(
    modifier = . . .
) { time →
   drawGrid(. . .)
}
```

```
drawCircle(
  . . .
  radius = map(
    sin(u * 10f + time * 20f),
    -1f, 1f, // from
    10f, 20f // to
  )
)
```

Animated Randomness



```
// Remember some random values
val randoms by remember {
 List(gridSize) {
   Random.nextFloat()
 }
}
val random = randoms[dotIndex]

// Animate with time
val radius = map(
  sin(time * 20f + random * 10f),
  -1f, 1f,
  3f, 13f
)

drawCircle(
    radius = radius,
    ...
)
```



Random - too unpredictable

Trig - too predictable

Something more interesting?

Noise Functions

- algorithms that produce coherent randomness
- given some inputs, outputs a number between -1 & 1
- every value is similar to its surrounding value
- “smooth” vs spiky pseudo-random numbers
- *how to access these functions?*

random *noise*

kotlin-graphics/glm

- Kotlin port of OpenGL Mathematics (*GLM*)
- handy math library for graphics
- vectors, matrices, randomness & noise!

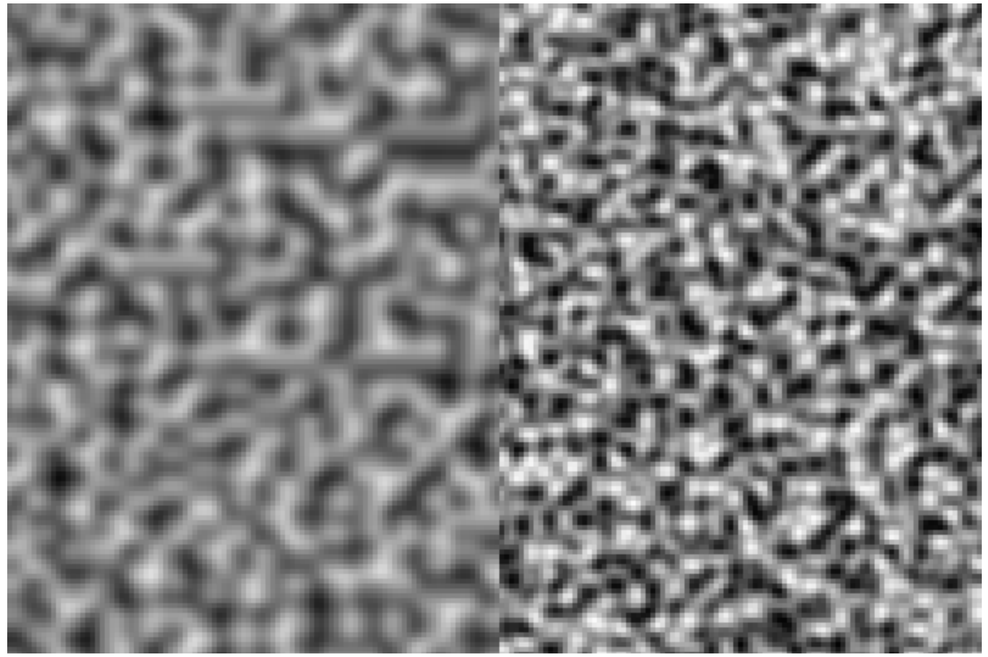
kotlin-graphics / glm Public

```
allprojects {
    repositories {
        maven("https://raw.githubusercontent.com/kotlin-graphics/mary/master")
    }
}

implementation("kotlin.graphics:glm:0.9.9.1-4")
```

Types of Noise

- two kinds of noise functions in glm
- Perlin noise - developed by Ken Perlin in 1983
- and simplex in 2001
- simplex has less computational overhead, looks more like “noise”

perlin simplex



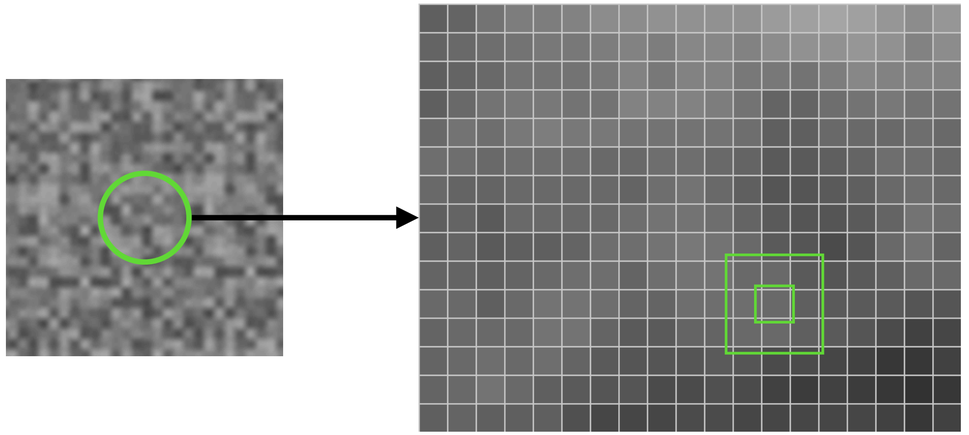
bit-101.com

glm + noise

```
perlin/simplex(Vec2(…))
```

2D Noise

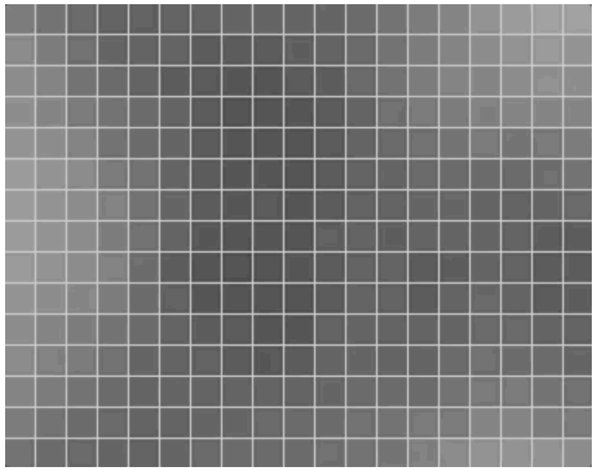
Each (x,y) value is similar to surrounding



```
perlin/simplex(Vec3(…))
perlin/simplex(Vec4(…))
```

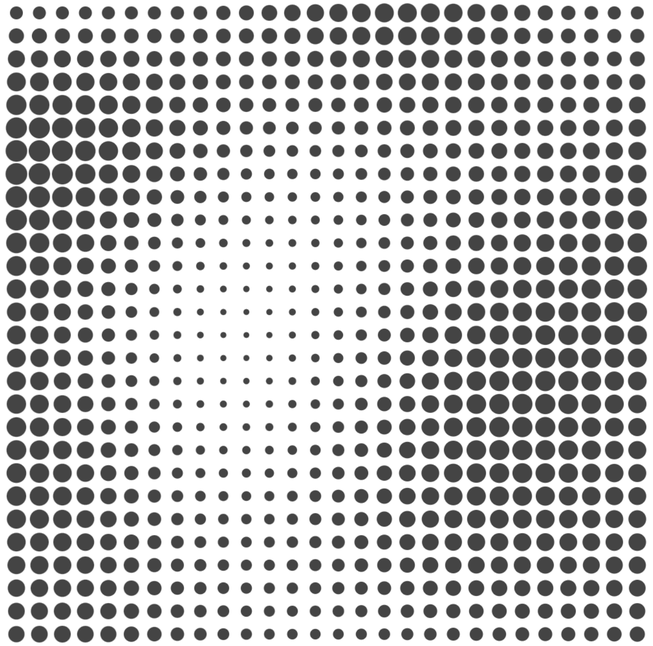
3D+ Noise

2D noise “slices” + 3rd/4th dimension of time!



Noisy Grids

- output range of noise = [-1, 1]
- changing radius with noise

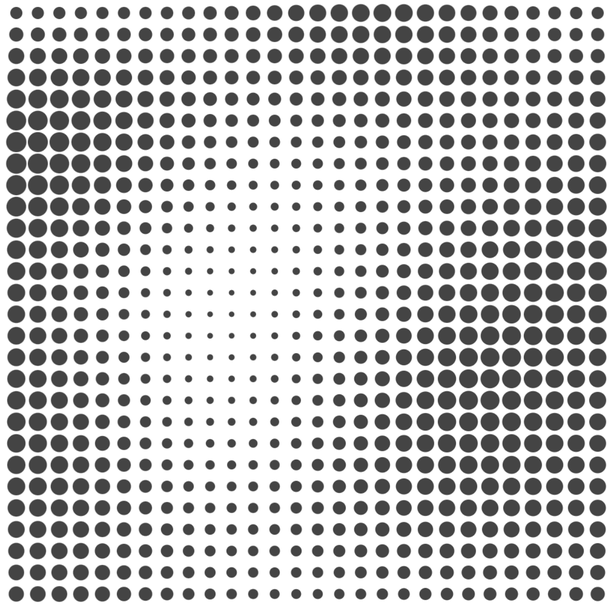


```
val noise2d = glm.simplex(
 Vec2(u, v)
)
```

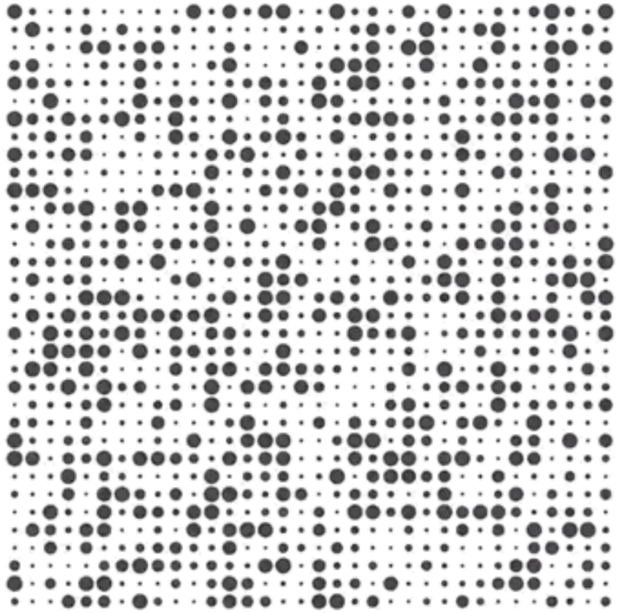
```
drawCircle(
  radius = map(
    noise2d,
    -1f, 1f,
    3f, 17f
  ),
  ...
)
```

Noise vs Random

Noise

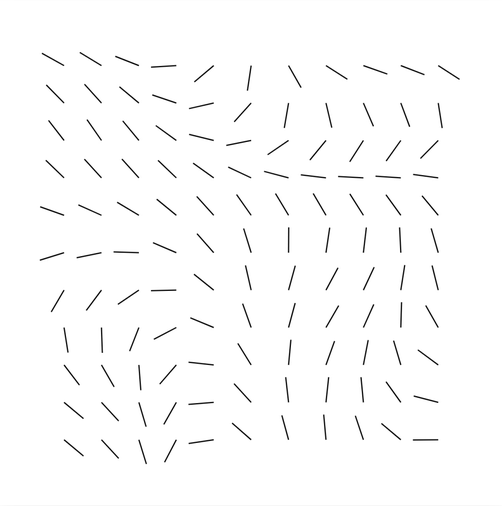


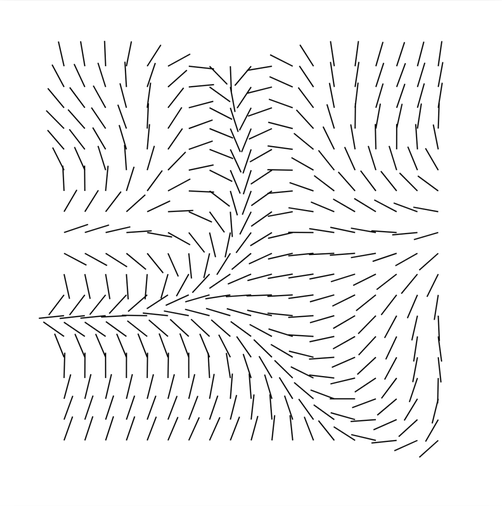
Random



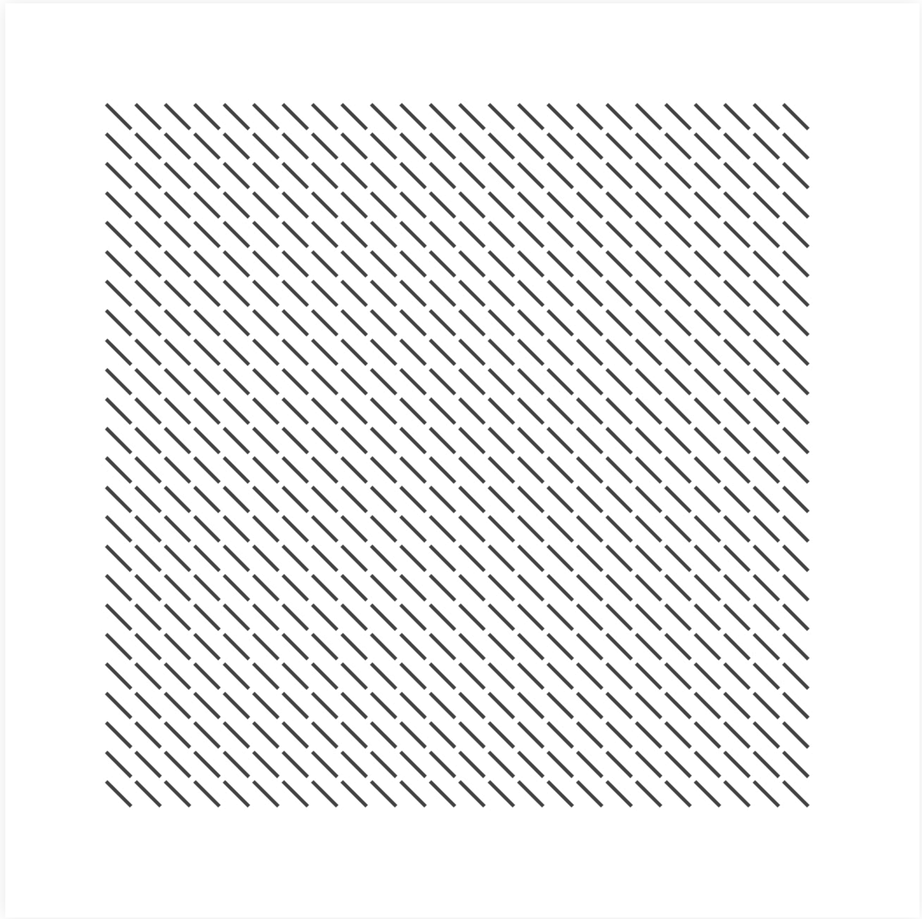
Noisy Angles

- draw lines instead of circles
- noise can also be visualized as angles aka flow fields
- flow fields lend an organic, cloth-like visual quality





Grid + Angles



lines & angles? should we rotate the canvas to draw things? 😨 (NO!)

```
// in drawGrid(...)

val r = 50f

val endX = ??????
val endY = ??????

+drawLine(
-drawCircle(
    start = Offset(startX, startY),
    end = Offset(endX, endY),
    strokeWidth = 5f,
    color = Color.DarkGray
)
```

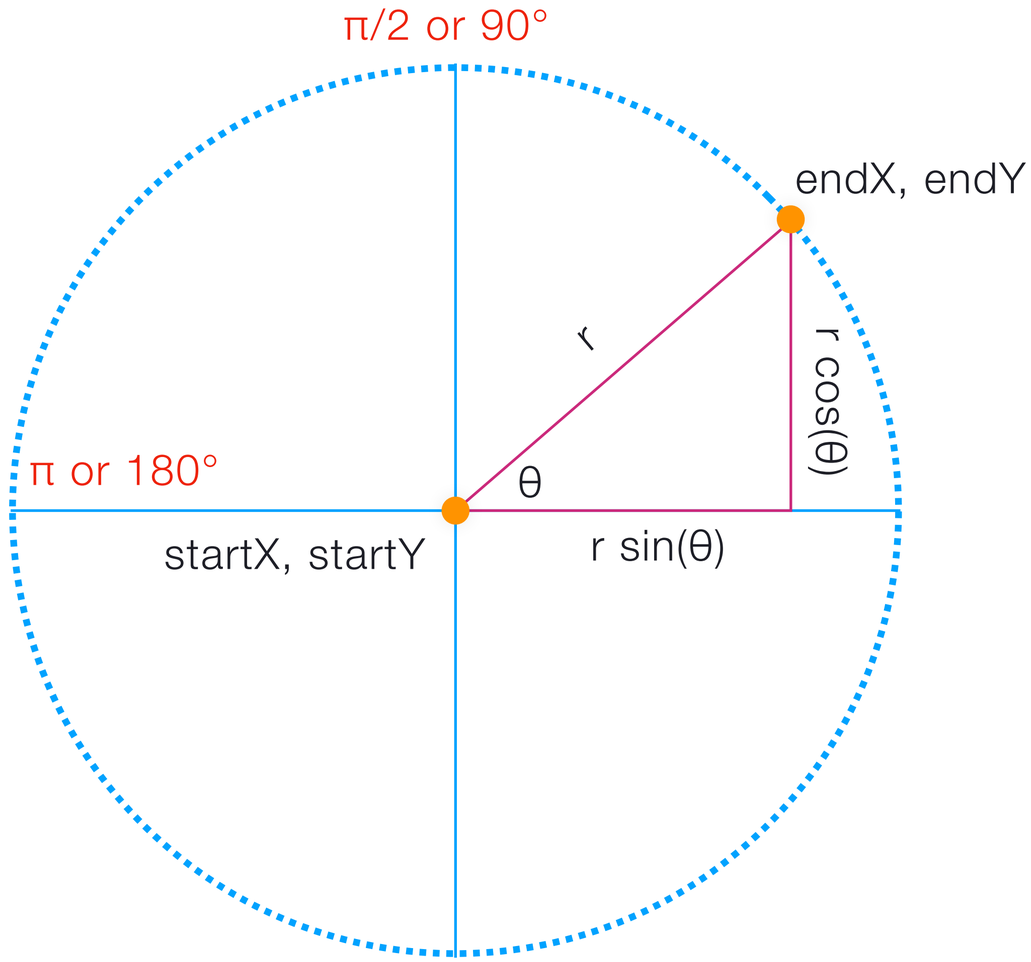
Polar Coords!

θ = radians radian = degree * π/180°

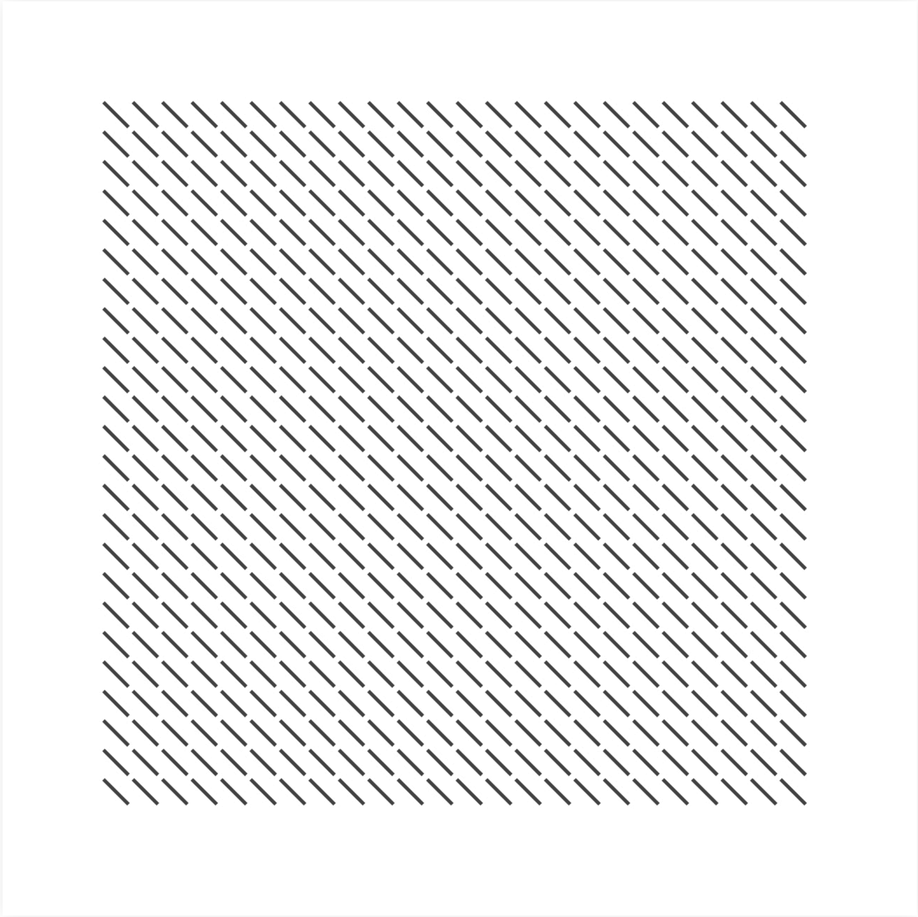
What we need for
line's end Offset

endX = startX + r*sin(θ)

endY = startY + r*cos(θ)

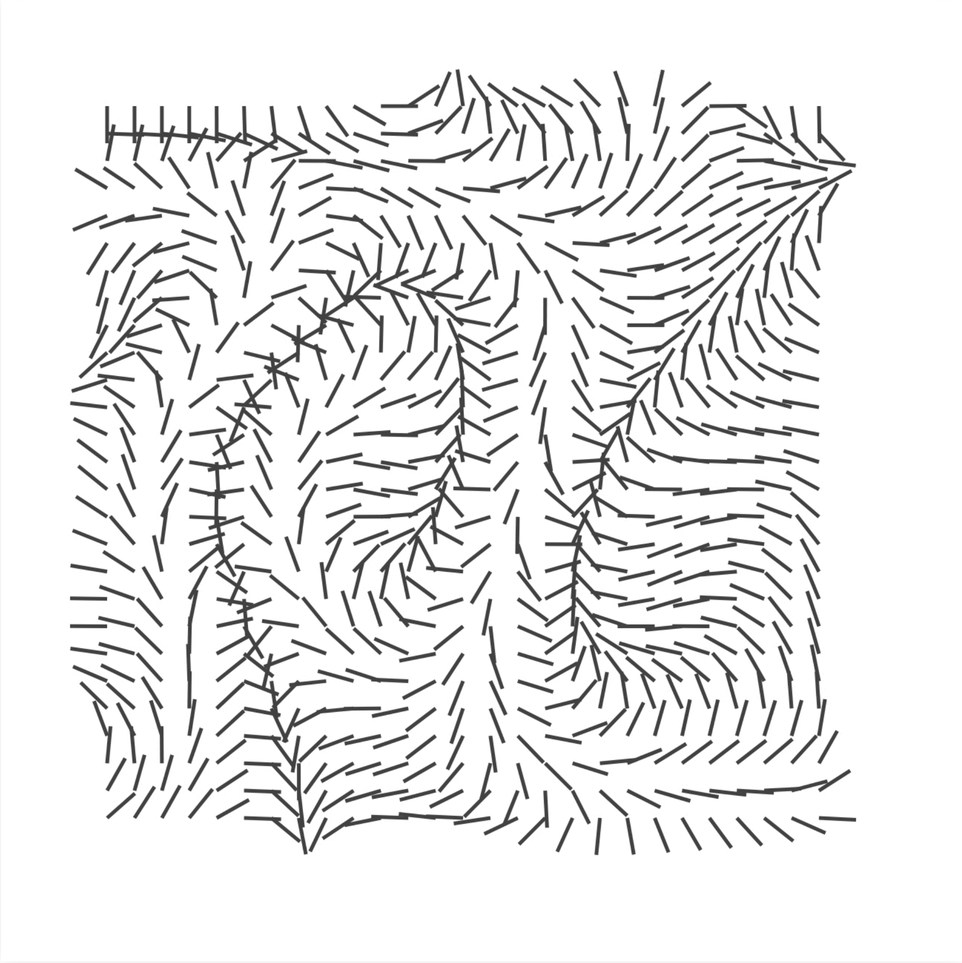


Grid + Angles



```
// in drawGrid(...)

val r = 50f

//π/4 = 45°
val endX = startX + (r * sin(PIf/4))
val endY = startY + (r * cos(PIf/4))

drawLine(
    start = Offset(startX, startY),
    end = Offset(endX, endY),
    ...
)
```

Angles + Noise



```
// in drawGrid(...)

val r = 50f

// multiply noise by 360° or 2π
val radians = glm.simplex(
  Vec2(u, v)
) * TWO_PI

val endX = startX + (r * sin(radians))
val endY = startY + (r * cos(radians))

drawLine(
    start = Offset(startX, startY),
    end = Offset(endX, endY),
    ...
)
```

Animated Flow Fields??

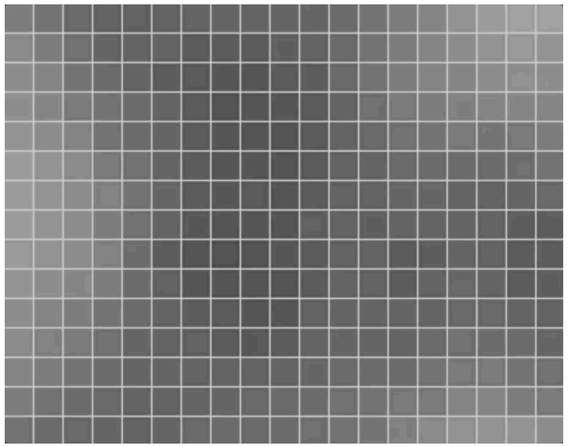
Where do we plug in time?

```
glm.simplex(Vec2(u, v))
```

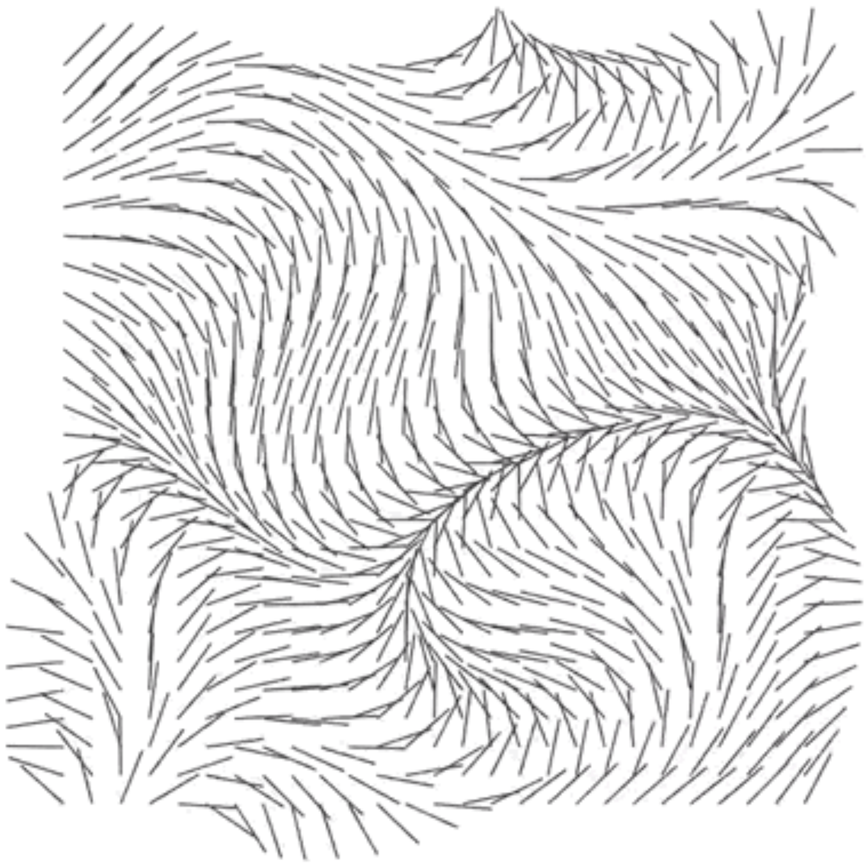
`perlin/simplex(Vec3(…))`
`perlin/simplex(Vec4(…))`

3D+ Noise

2D noise “slices” + 3rd/4th dimension of time!

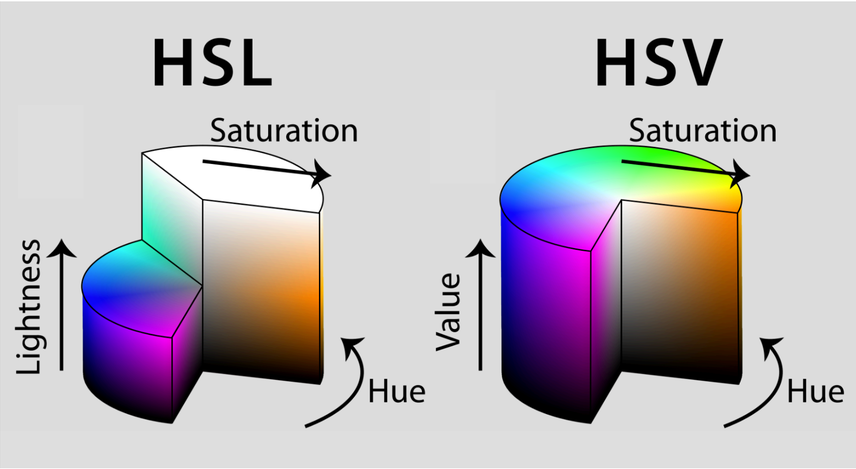


Animated Flow Fields!



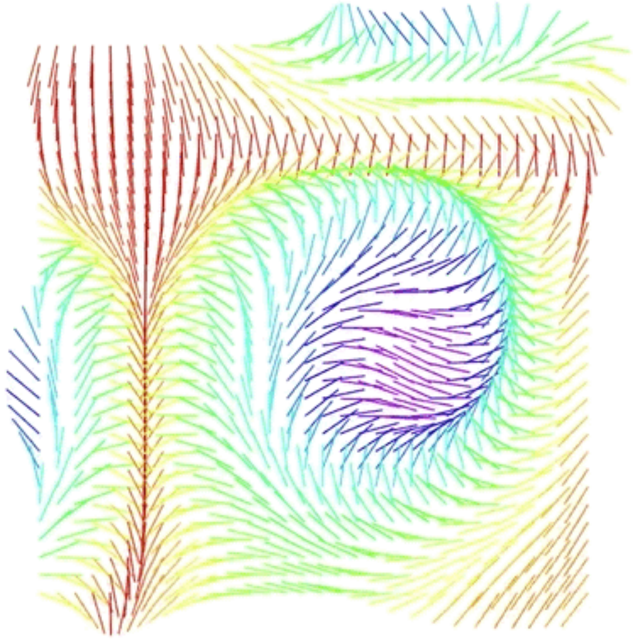
```
val radians = glm.simplex(
  Vec4(
    x = u,
    y = v,
    z = 5f * cos(TWO_PI * time / 20f),
    w = 5f * sin(TWO_PI * time / 20f)
  )
) * TWO_PI

val endX = startX + (r * sin(radians))
val endY = startY + (r * cos(radians))
drawLine(
  start = Offset(startX, startY),
  end = Offset(endX, endY),
  ...
)
```

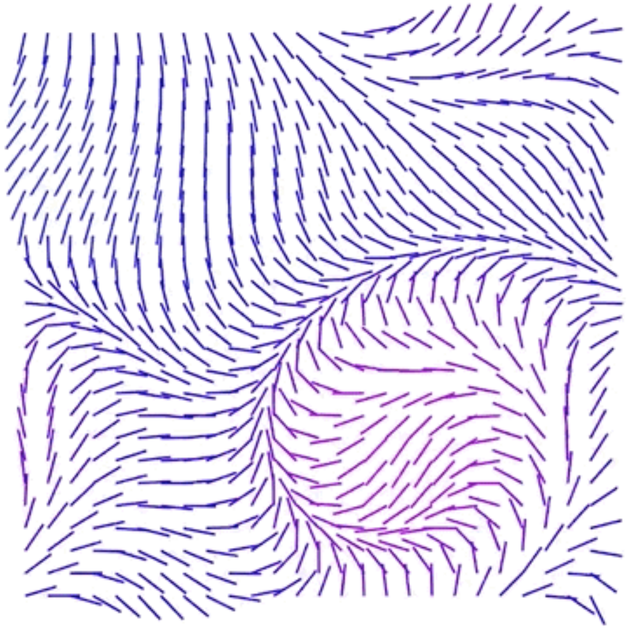
Color



wikipedia.org

- HSL, HSV use hue
- hue is in [0, 360] -> like angles?
- map noise to hue ranges 🌈

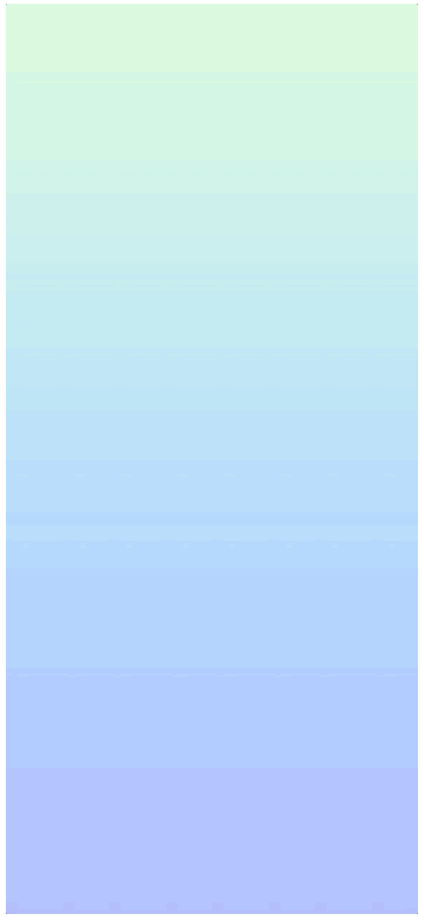




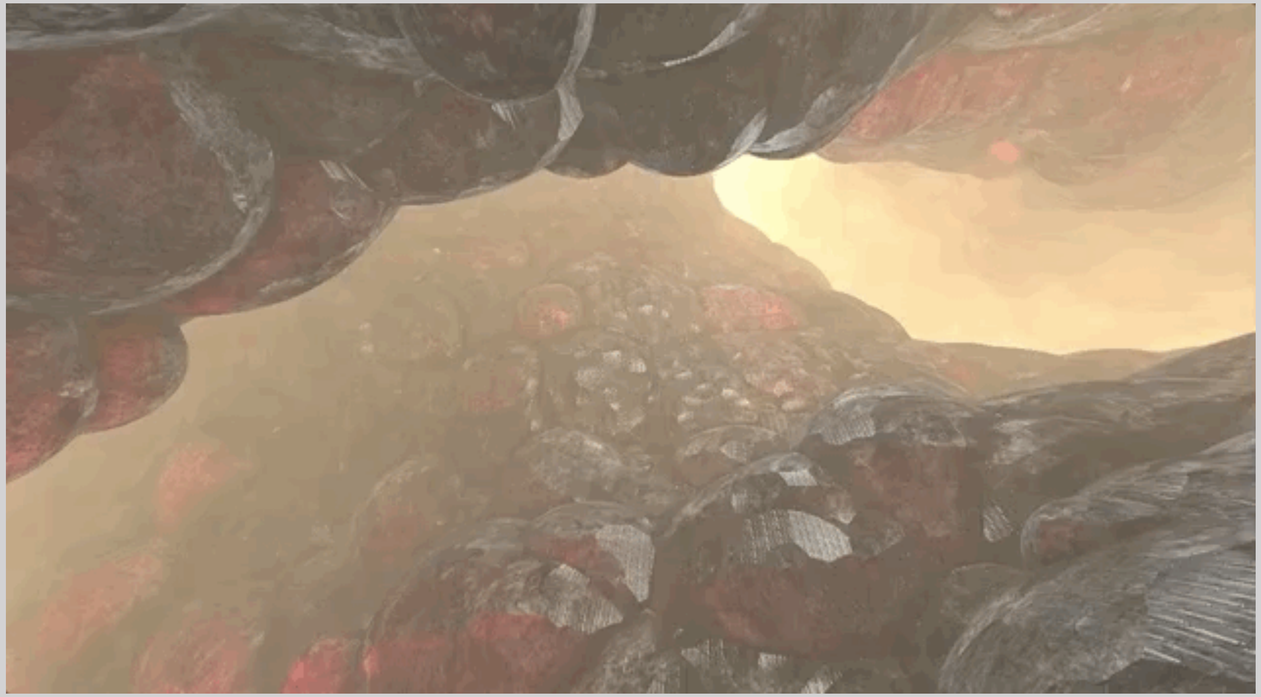
```
// Compute hue based on noise
val hue = (noise * 360f).absoluteValue
// OR
val hue = map(noise, -1f, 1f, 170f, 300f)

// Use hue in hsv
val color = Color.hsv(hue, saturation=1f, value=1f)
drawLine(color = color, ...)
```

// Shaders Android 13

What are they?



- programs mapping a pixel’s position to a color
- they run per pixel on the screen, in parallel
- they only have info about “the current” pixel and its position
- cant access neighboring pixels



3D Cellular Tiling *Created by* ***Shane*** *in 2016-04-17*

Shader Inputs

```
uniform vec3      iResolution;           // viewport resolution (in pixels)
uniform float     iTime;                 // shader playback time (in seconds)
uniform float     iTimeDelta;            // render time (in seconds)
uniform int       iFrame;                // shader playback frame
uniform float     iChannelTime[4];       // channel playback time (in seconds)
uniform vec3      iChannelResolution[4]; // channel resolution (in pixels)
uniform vec4      iMouse;                // mouse pixel coords. xy: current (if MLB down), zw: click
uniform samplerXX iChannel0..3;          // input channel. XX = 2D/Cube
uniform vec4      iDate;                 // (year, month, day, time in seconds)
uniform float     iSampleRate;           // sound sample rate (i.e., 44100)
```

```
586         float tanHi = abs(mod(per*.5 + t + iTime, per) - per*.5);
587         vec3 tanHiCol = vec3(0, .2, 1)*(1./tanHi*.2);
588         sceneCol += tanHiCol;
589         */
590
591
592         //vec3 refCol = vec3(.5, .7, 1)*smoothstep(.2, 1., noise3D((sp + ref*2.)*2.)*.66 + nois
593         //sceneCol += refCol*.5;
594
595
596         // Shading.
597         sceneCol *= atten*shading*ao;
598
599         //sceneCol = vec3(ao);
600
601
602     }
603
604     // Blend the scene and the background with some very basic, 4-layered fog.
605     float mist = getMist(camPos, rd, light_pos, t);
606     vec3 sky = vec3(2.5, 1.75, .875)* mix(1., .72, mist)*(rd.y*.25 + 1.);
607     sceneCol = mix(sceneCol, sky, min(pow(t, 1.5)*.25/FAR, 1.));
608
609     // Clamp, perform rough gamma correction, then present the pixel to the screen.
610     fragColor = vec4(sqrt(clamp(sceneCol, 0., 1.)), 1.0);
611
612 }
```

On Android

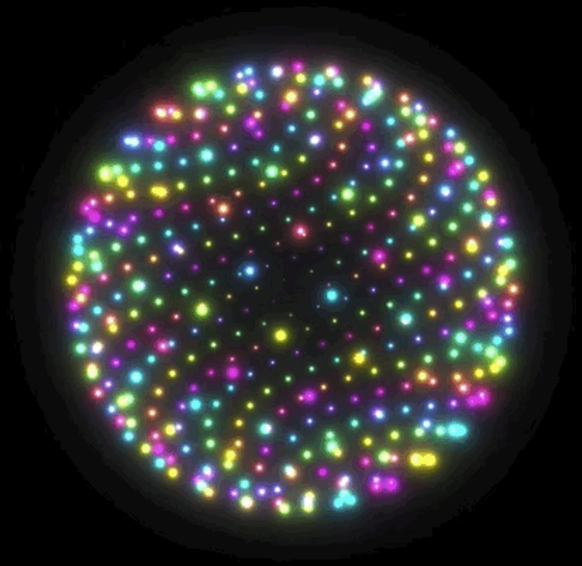
- < Android 13 can use only pre-built shaders:
  BitmapShader, **LinearGradient**, etc

- with Android 13 can use
  programmable **RuntimeShader**s written in AGSL

- GPU level effects without direct OpenGL

- pass shaders as ***string*** into a **RuntimeShader**
  object!

```
val shader = RuntimeShader("""
  // Shader code here in AGSL
 """
)

// Make a Brush
val brush = ShaderBrush(shader)

// Canvas() / DrawScope
onDraw = { value →
 // Use it to paint anything!
 drawRect(brush)
}
```



GLSL - shadertoy.com

Shader Inputs

```
uniform vec3      iResolution;           // viewport resolution (in pixels)
uniform float     iTime;                 // shader playback time (in seconds)
uniform float     iTimeDelta;            // render time (in seconds)
uniform int       iFrame;                // shader playback frame
uniform float     iChannelTime[4];       // channel playback time (in seconds)
uniform vec3      iChannelResolution[4]; // channel resolution (in pixels)
uniform vec4      iMouse;                // mouse pixel coords. xy: current (if MLB do
uniform samplerXX iChannel0..3;          // input channel. XX = 2D/Cube
uniform vec4      iDate;                 // (year, month, day, time in seconds)
uniform float     iSampleRate;           // sound sample rate (i.e., 44100)
```

```
//https://twitter.com/XorDev/status/1475524322785640455
void mainImage( out vec4 fragColor, in vec2 fragCoord ) {
  vec4 o = vec4(0.0);
  vec2 p = vec2(0.0), c=p, u=fragCoord.xy*2.-iResolution.xy;
  float a;
  for (float i=0.0; i<4e2; i++) {
    a = i/2e2-1.;
    p = cos(i*2.4+iTime+vec2(0.0,11.0))*sqrt(1.-a*a);
    c = u/iResolution.y+vec2(p.x,a)/(p.y+2.);
    o += (cos(i+vec4(0.0,2.0,4.0,0.0))+1.)/dot(c,c)*(1.-p.y)/3e4;
  }
  fragColor = o;
}

```

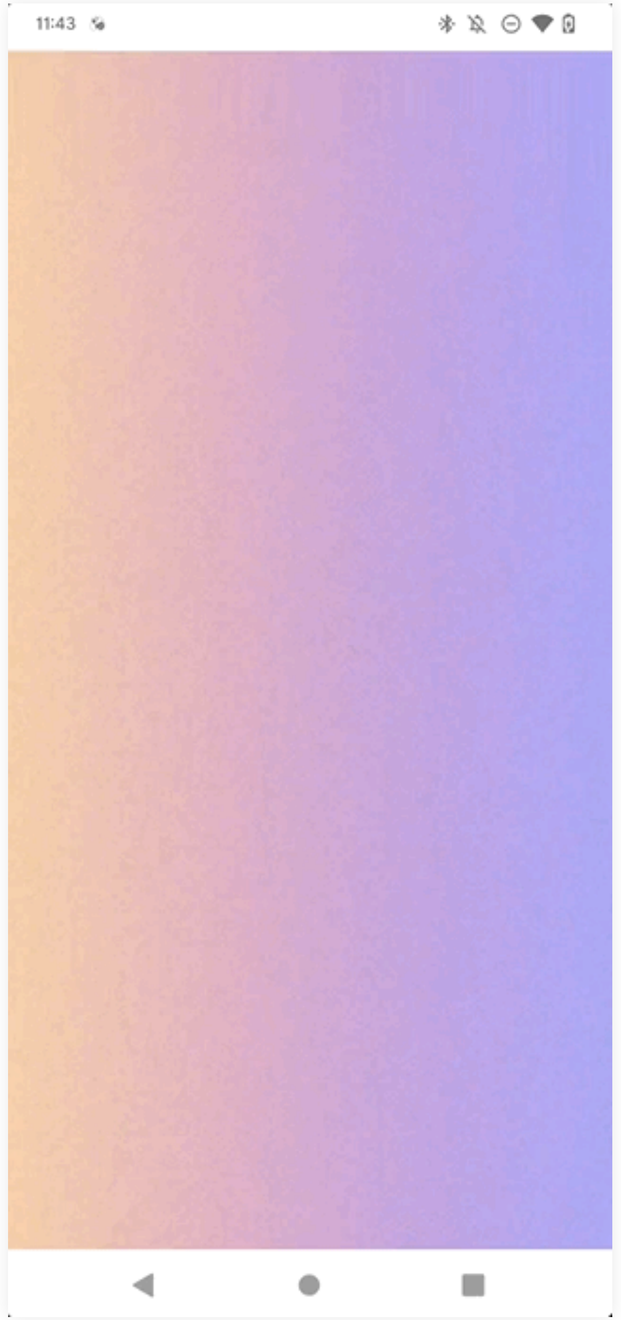
SKSL (~AGSL) - shaders.skia.org

Shader Inputs

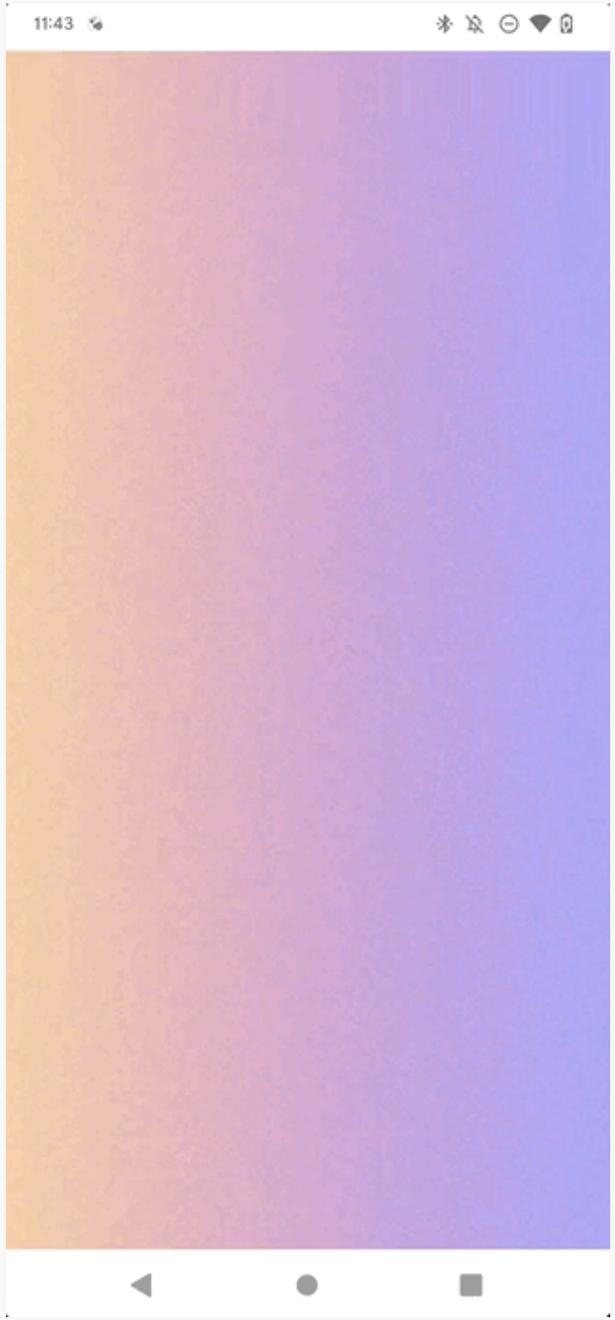
```
uniform float3 iResolution;     // Viewport resolution (pixels)
uniform float  iTime;           // Shader playback time (s)
uniform float4 iMouse;          // Mouse drag pos=.xy Click pos=.zw (pixels)
uniform float3 iImageResolution; // iImage1 resolution (pixels)
uniform shader iImage1;         // An input image.
```

```
// Source: @XorDev https://twitter.com/XorDev/status/1475524322785
vec4 main(vec2 fragCoord) {
  vec4 o = vec4(0.0);
  vec2 p = vec2(0.0), c=p, u=fragCoord.xy*2.-iResolution.xy;
  float a;
  for (float i=0; i<4e2; i++) {
    a = i/2e2-1.;
    p = cos(i*2.4+iTime+vec2(0.0,11.0))*sqrt(1.-a*a);
    c = u/iResolution.y+vec2(p.x,a)/(p.y+2.);
    o += (cos(i+vec4(0.0,2.0,4.0,0.0))+1.)/dot(c,c)*(1.-p.y)/3e4;
  }
  return o;
}

```

```
// AGSL ⬇️

uniform float2 iResolution; // Viewport resolution (px)
uniform float iTime; // Shader playback time (s)

vec4 main(in float2 fragCoord) {

  // Normalized pixel coordinates (from 0 to 1)
  vec2 uv = fragCoord/iResolution.xy;

  // Time varying pixel color
  vec3 col = 0.8
  + 0.2*cos(iTime*2.0+uv.xxx*2.0+vec3(1,2,4));

  // Output to screen
  return vec4(col,1.0);
}
```



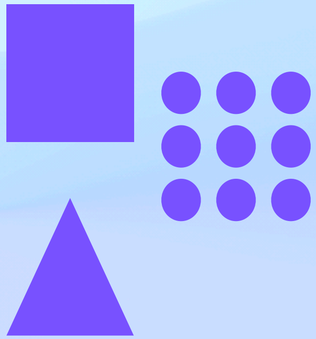
```
val shader = RuntimeShader("...shader code ...")
val brush = ShaderBrush(shader)

Sketch(
    onDraw = { time →
      // Get dimensions from DrawScope.size
      shader.setFloatUniform(
        "iResolution",
        size.width, size.height
      )

      // From Sketch!
      shader.setFloatUniform("iTime", time)

    drawRect(brush)
    }
)
```



Shaders in 5 Steps 🤯

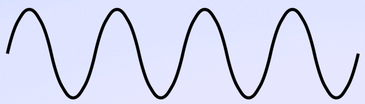
1. **thebookofshaders.com** !
2. use **shadertoy.com** & **shaders.skia.org** as a playground
3. GLSL - convert to AGSL; SKSL - use as is
4. plug into **RuntimeShader**
5. use size for **iResolution**, Sketch, or any animating time value for **iTime**

Generative Art Pipeline



+

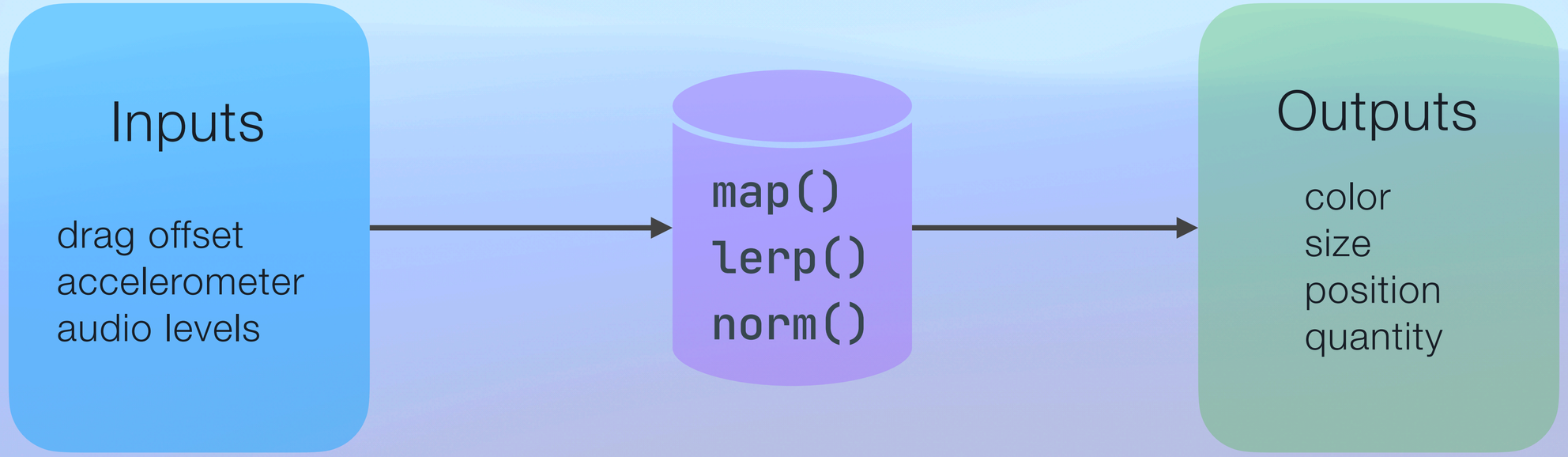
noise(2d)
noise(4d)
random()



sin() cos()

+

Going Further



Resources & Examples

@TashaRamesh