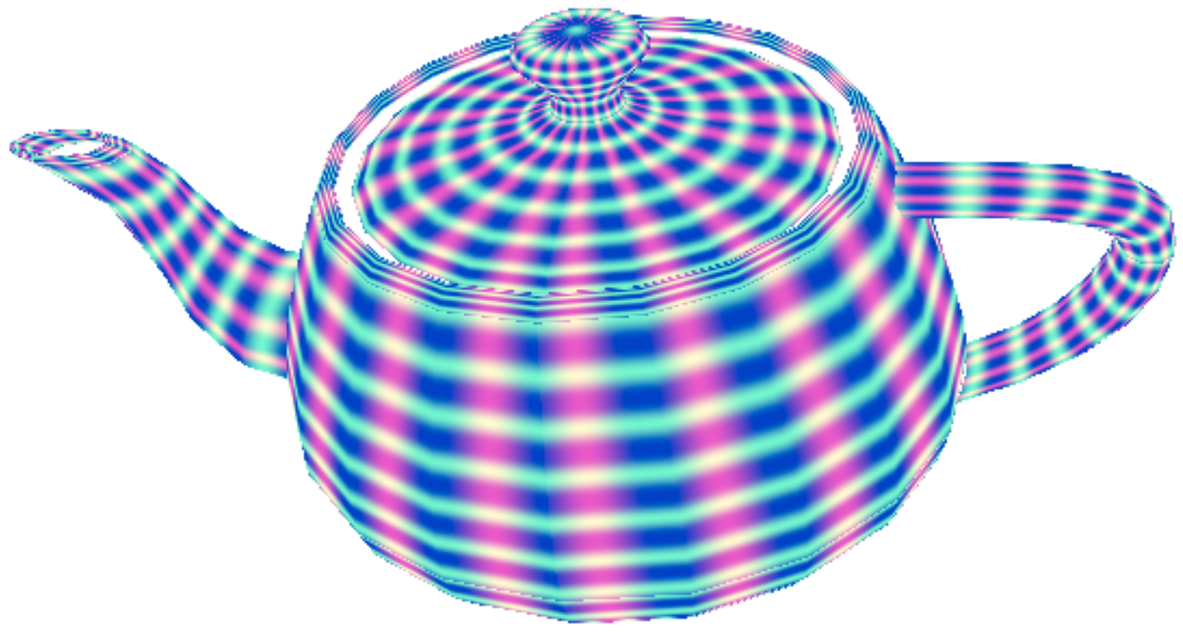
Information Coding / Computer Graphics, ISY, LiTH

TNM084
Procedural images

Ingemar Ragnemalm, ISY



Lecture 2

- Random numbers
- Splines
- Noise
- Filtering noise
- Perlin noise
- Simplex noise

Random numbers

Important source of interesting patterns

Randomness is chaos?

Randomness with structure

What is random?

Pick a "random" number - never truly random

People never pick the same number twice. Randomness does!

Roll a die

Is the die fair?

But how can we make a computer "roll a die"?

Pseudo-random numbers

Computers are intrinsically deterministic

How can we make randomness?

- Heat + A/D conversion
- Pseudo-random number generation

Analog randomness

Noise in analog inputs are also random numbers!

Input from noisy sensor, A/D-conversion.

Drawback: The result is not repeatable and dependent of momentary situation, like heat level.

The system may stop working a cold day!

PRBS

Pseudo-random binary sequence

Shift bits and do XOR with some bits

Creates a chaotic sequence

Eventually repeats itself

PRBS

Needs a "seed", a start value

Needs to pick target bits, a "mask"

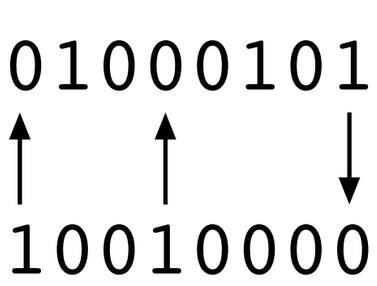
Find ones that create long sequences

Special cases with short sequence may occur

PRBS, example

8 bits

Sequence of 254 numbers



1 at end -> XOR with mask

106 53 82 41 92 46 23 67 105 124 62 31 71 107 125 118 59 85 98 49 80 40
20 10 5 74 37 90 45 94 47 95 103 123 117 114 57 84 42 21 66 33 88 44 22
11 77 110 55 83 97 120 60 30 15 79 111 127 119 115 113 112 56 28 14 7
75 109 126 63 87 99 121 116 58 29 70 35 89 100 50 25 68 34 17 64 32 16
8 4 2 1 72 36 18 9 76 38 19 65 104 52 26 13 78 39 91 101 122 61 86 43
93 102 51 81 96 48 24 12 6 3 73 108 54 27 69

Random library functions

Pseudo-random number functions are in most run-time libraries

High quality random functions, long sequencies

rand() and srand() in the standard libraries

random() and srandom()

Not cryptographically secure; irrelevant for us.

But...

The random number libraries are generally *sequential*!

Not useable in a shader.

Also questioable in the parallel image generation model.

In shaders, we need another solution!

Random numbers in GLSL

GLSL runs on the GPU

Built-in random functions never worked (!)

Typical ways to get noise in GLSL:

- Noise texture, pre-generated noise
- Feed shader continuously with numbers from host
- Truncated trigonometric numbers

Noise texture

Really pre-generated number in any kind of buffer

Easy to do

Static

Can be used for both static things and some animations (example: snow)

Continous feeding

Run random generator on host (CPU)

Feed new random numbers every frame

Much data traffic

Much load on CPU

Truncated trigonometric numbers

Smart trick with built-in functions

M is a number >> 1

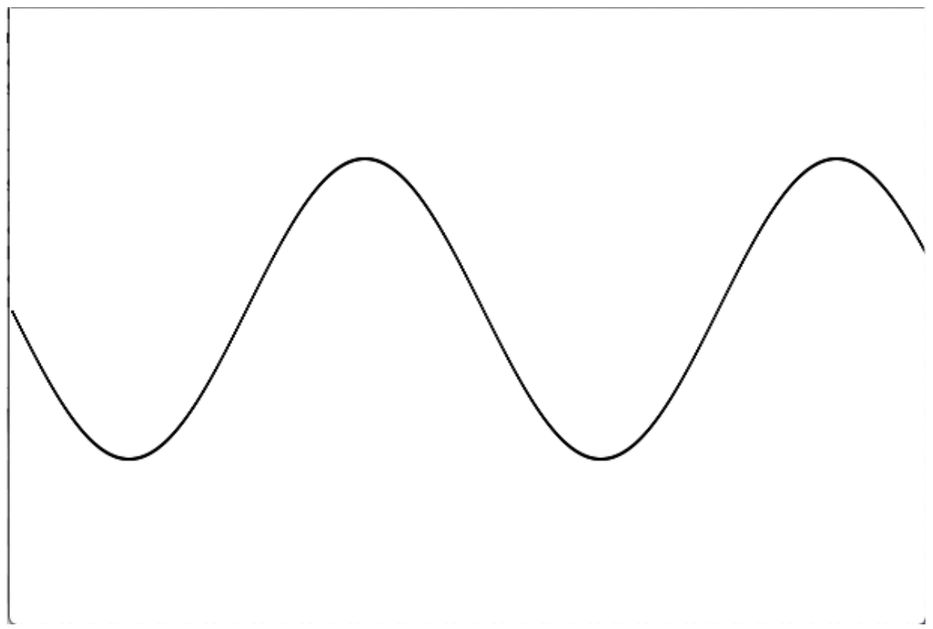
n = sin(x) * M

r = n - (int)n

Plain sin function

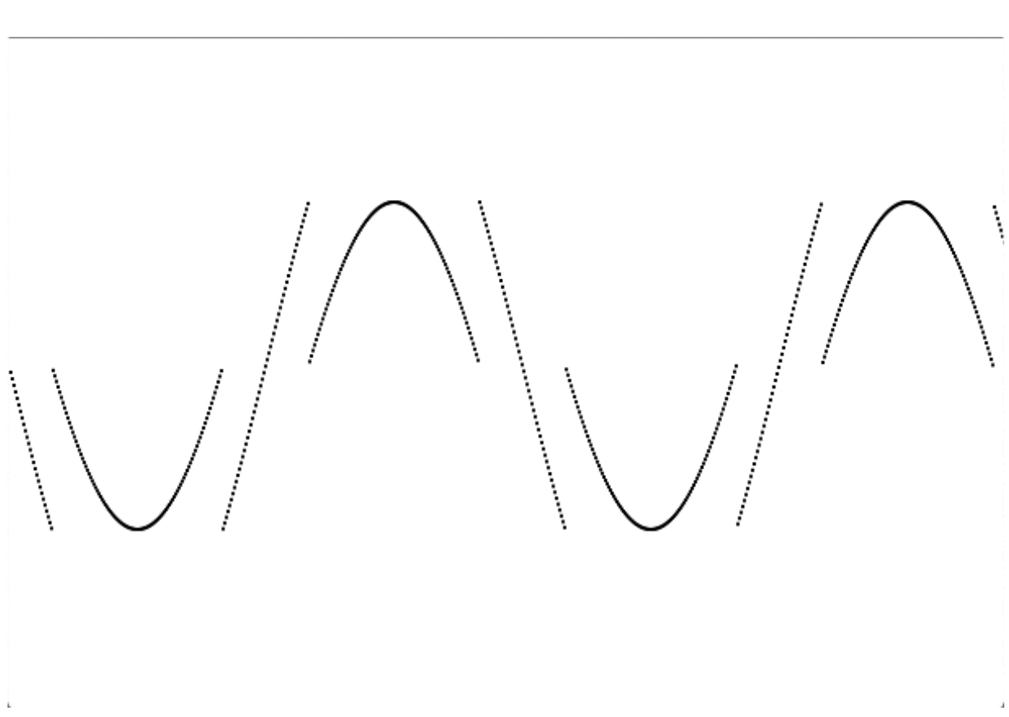
f := sin(x);

Nothing random about it



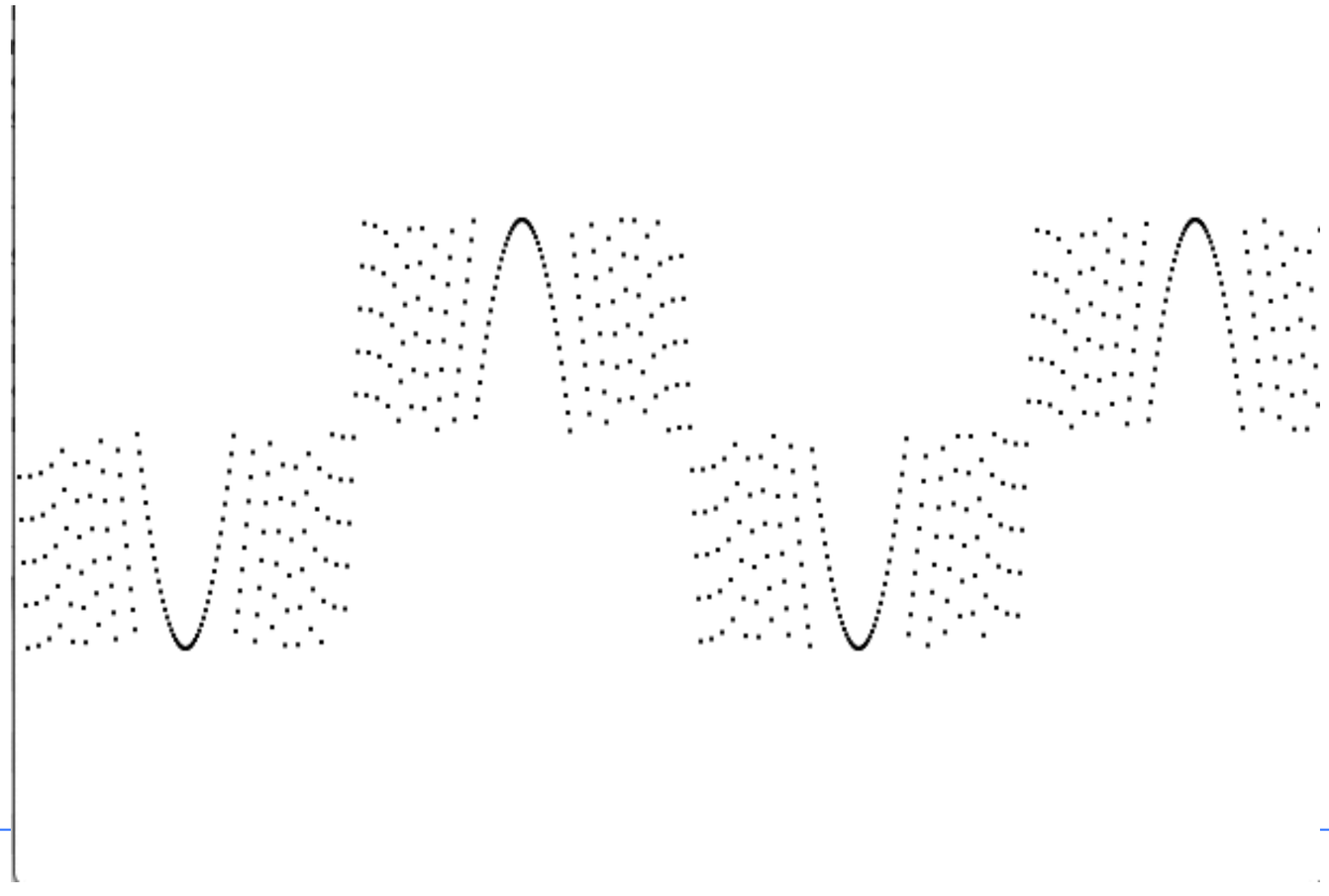
Multiply by 2, take the fraction part

f := frac(sin(x) * 2);



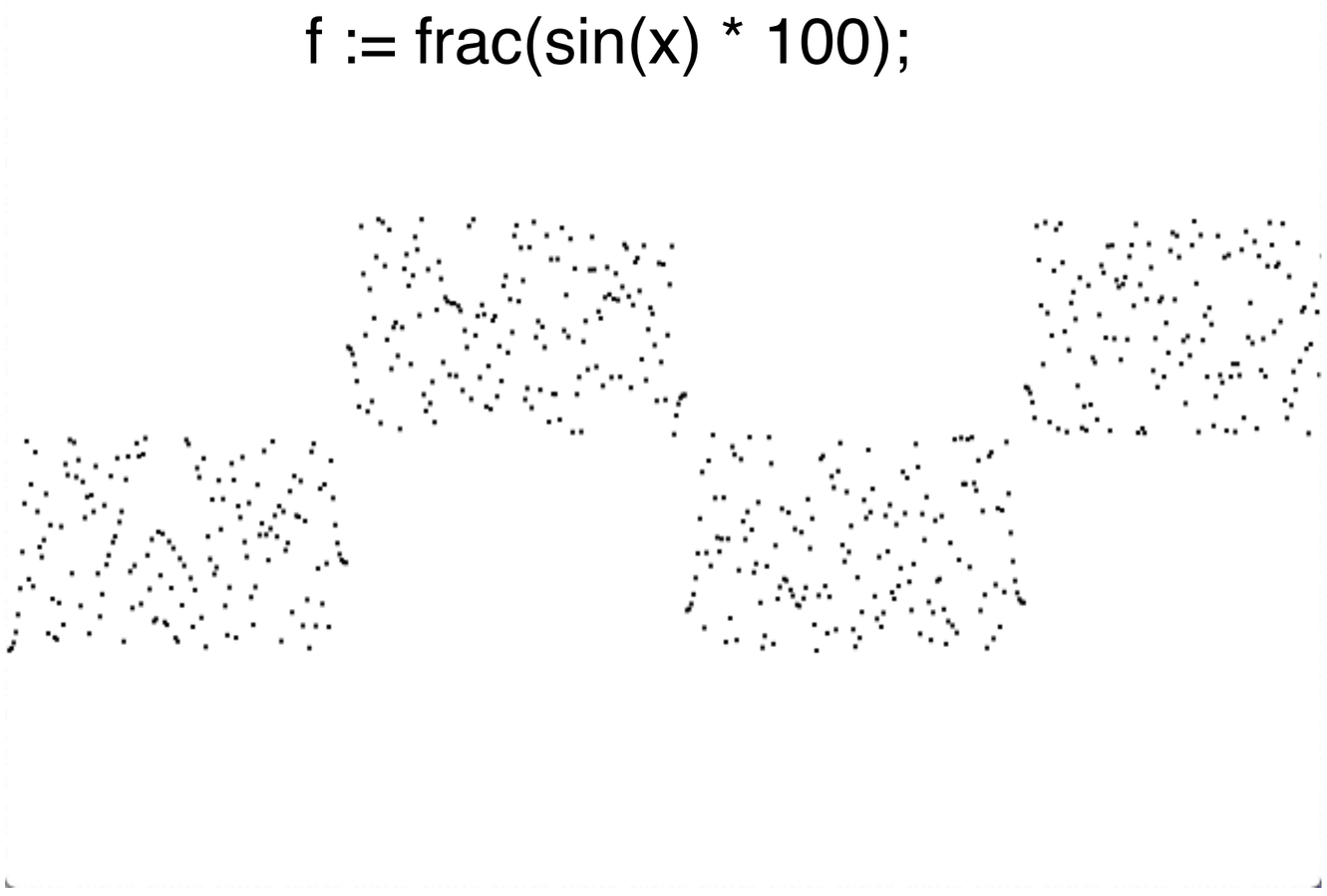
But if we multiply by more...

f := frac(sin(x) * 10);



...and even more...

f := frac(sin(x) * 100);



abs() to keep on one side and go high

f := frac(sin(x) * 100000);

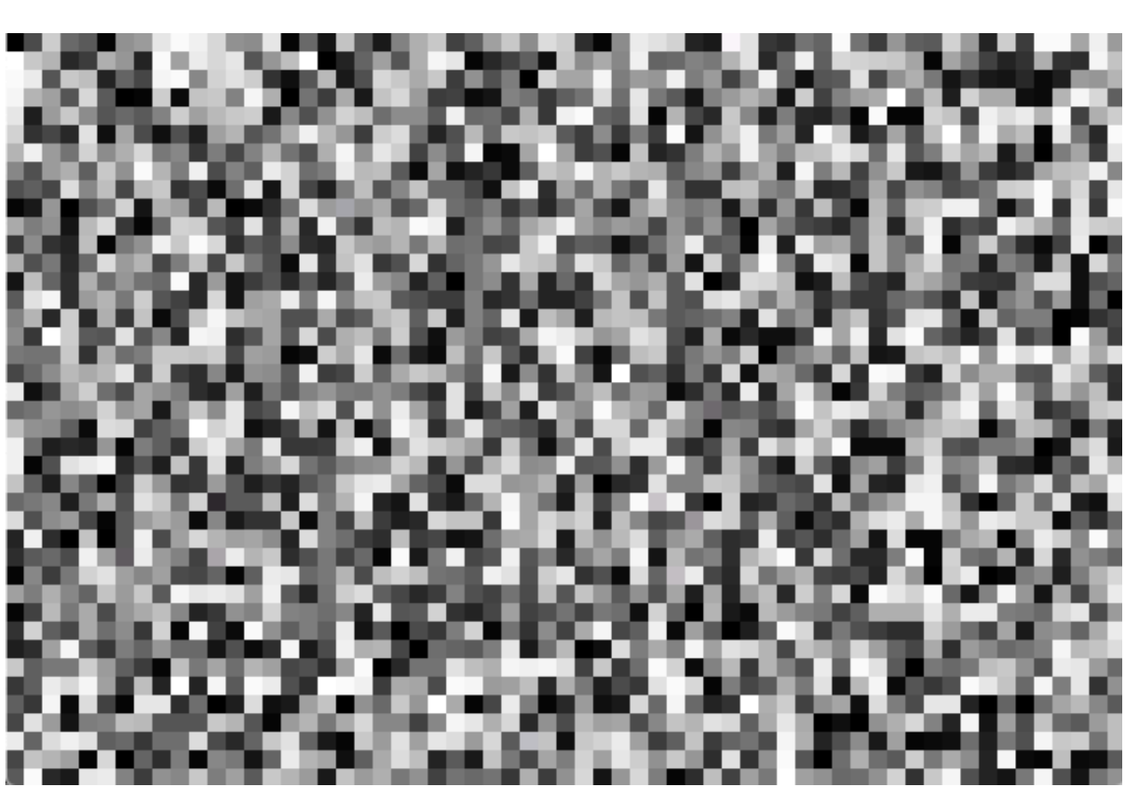
Truncated trigonometric numbers

We now have a non-sequential random number generator!
Only the pixel position is needed!

Just never take steps by π!

Random pixel values

Random intensity



Random numbers by permutation polynoms

Drawback with truncated harmonic functions: The result is implementation dependent!

Randomness by truncating an integer-based function using the modulo function.

Stefan suggests

$$hash = (34x^2 + 10x) \bmod 289$$

Creates same result on all machines!

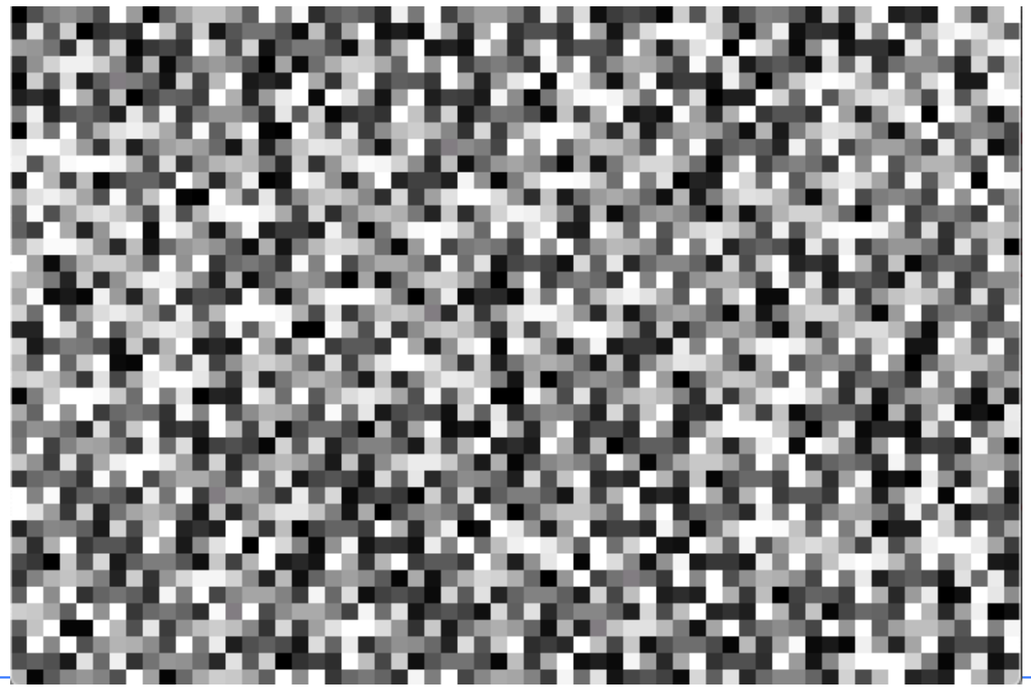
Permutation polynoms - usage

The hash function must be called *twice*!

$$hash = (34i^2 + 10i) \bmod 289$$

hash(hash(x)+y);

This creates a nice randomness:



Uses of randomness

Like above: Random patterns

Random geometrical patterns

Random movement

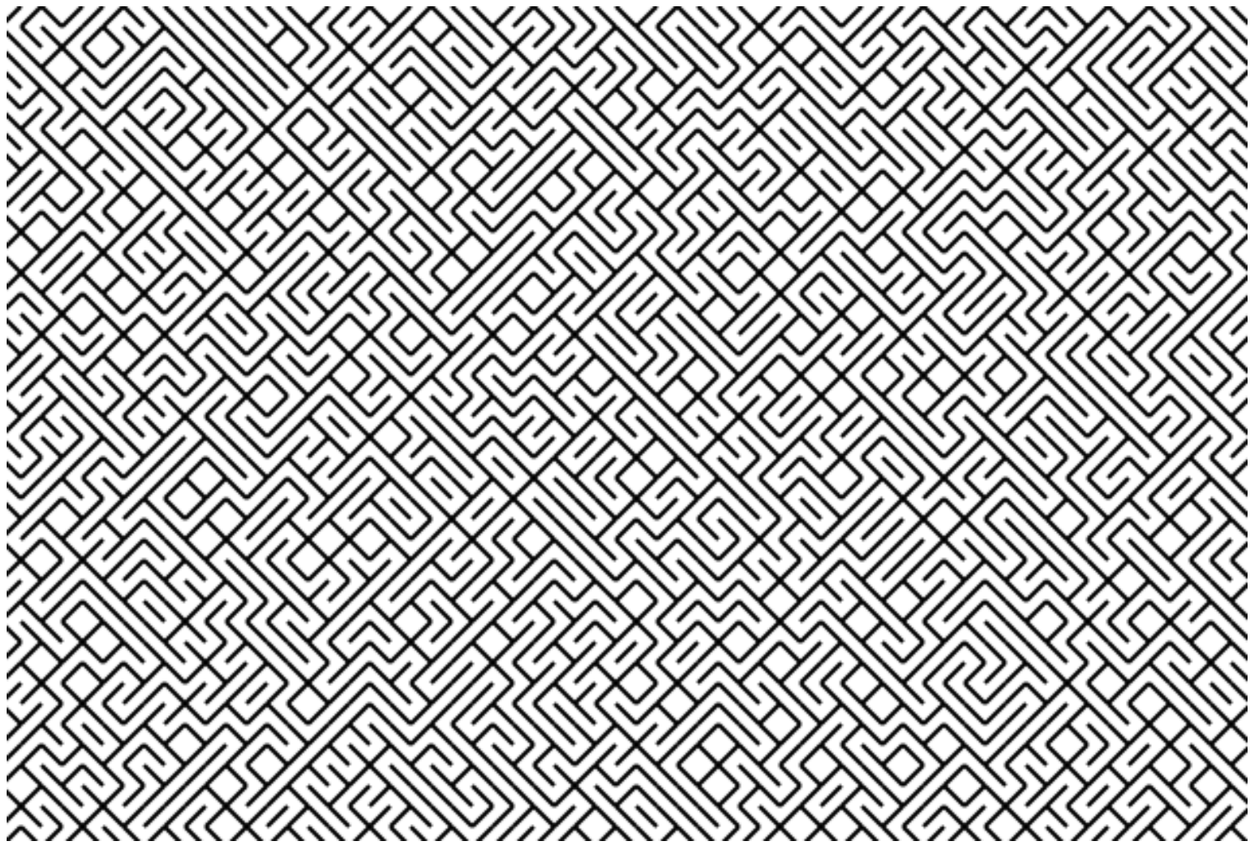
Random location

Random geometry

and more...

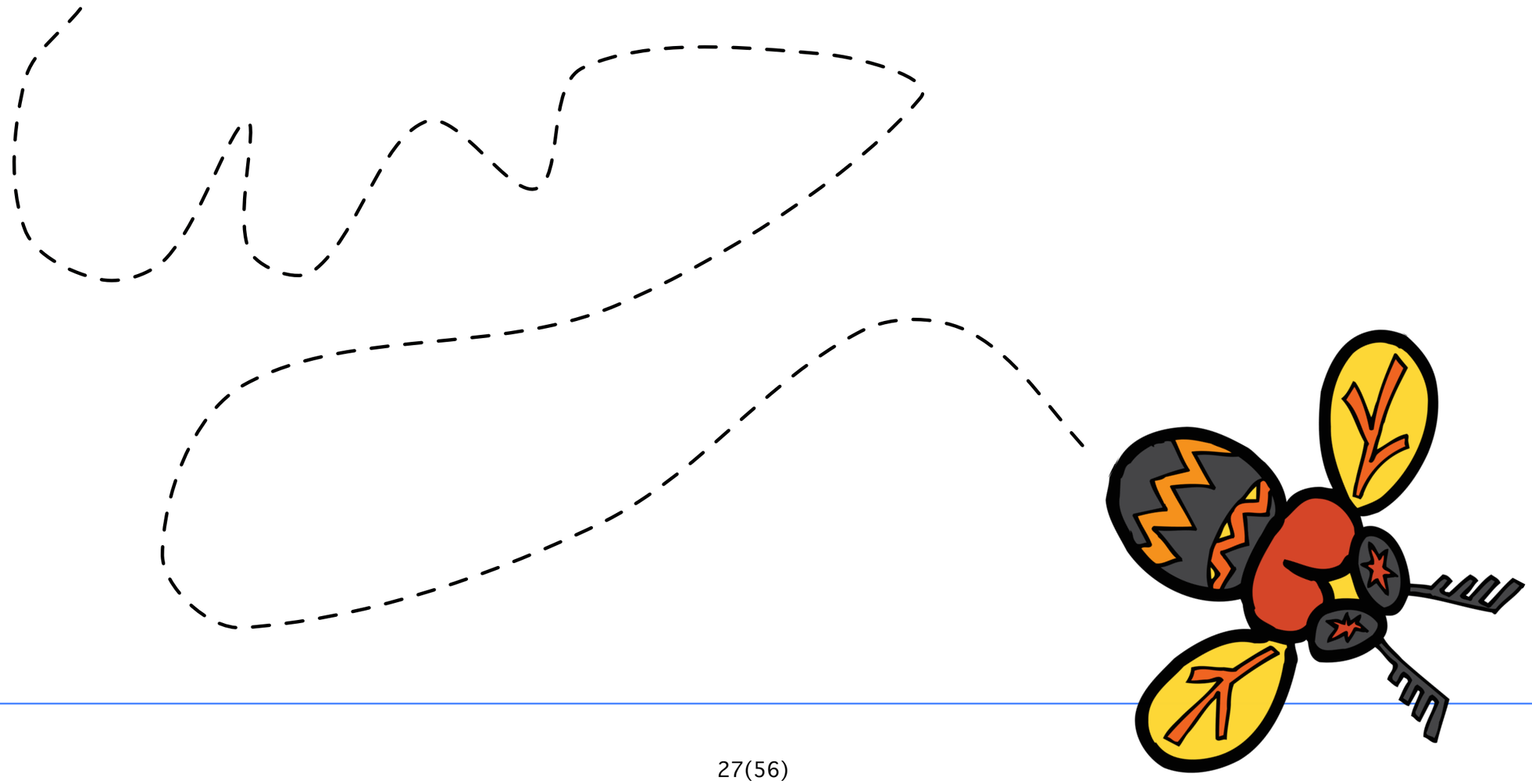
Random geometrical patterns

Vary the contents of areas



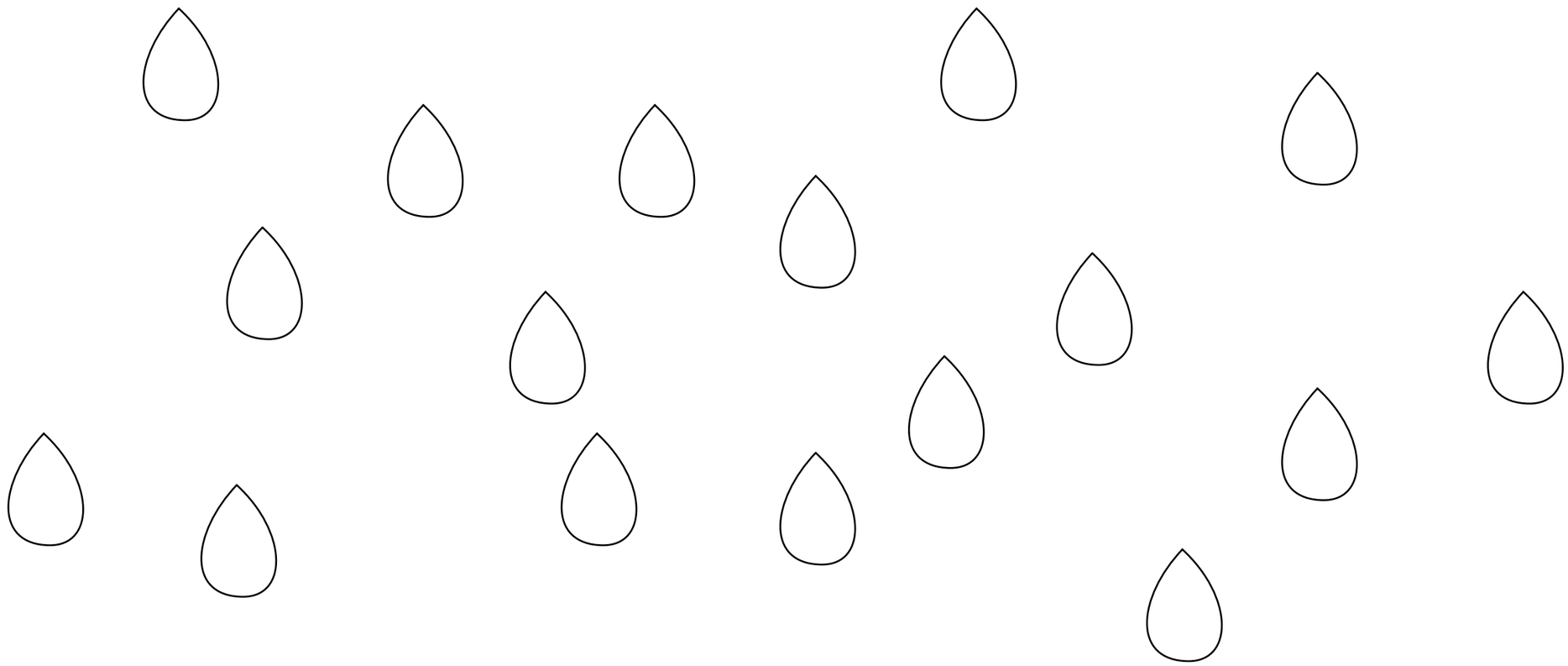
Random movement

Vary the speed or direction of moving objects



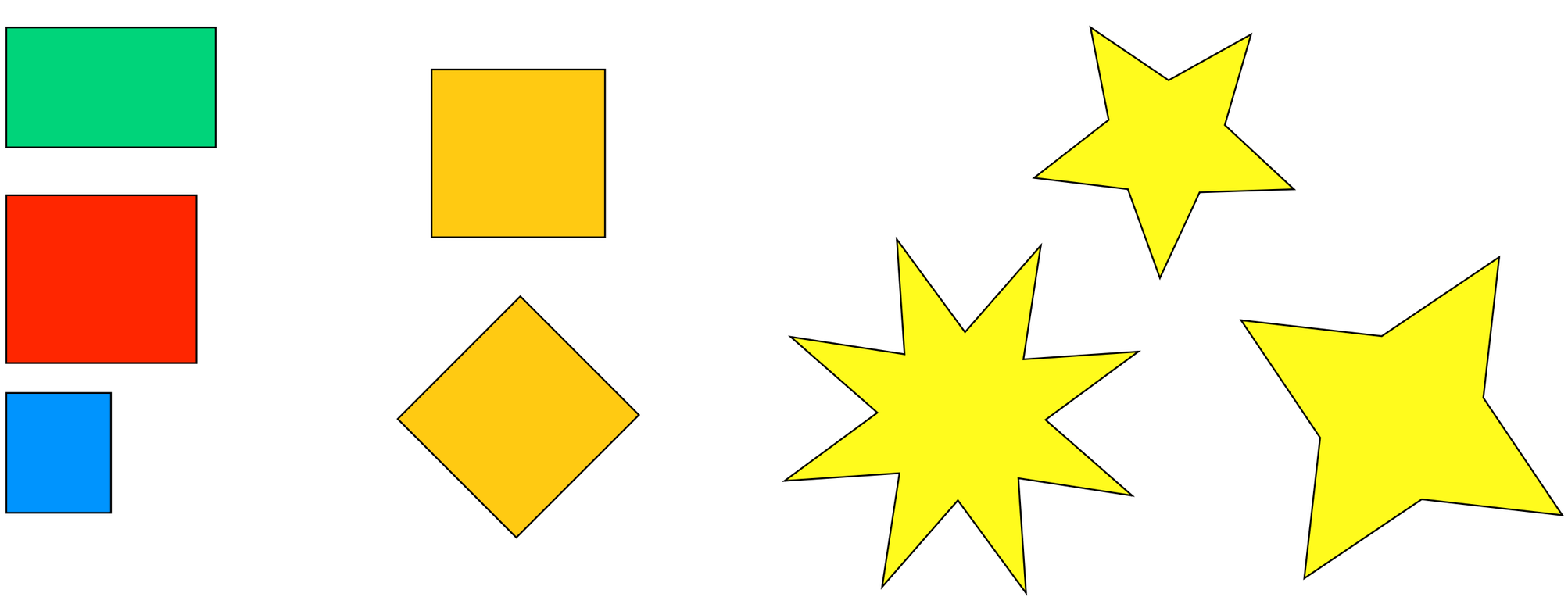
Random location

Vary the translation of objects



Random geometry

Vary parameters of objects



And more!

Noise functions!

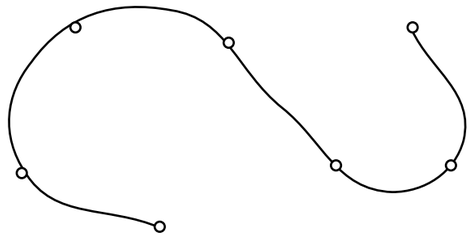
White noise

Colored noise

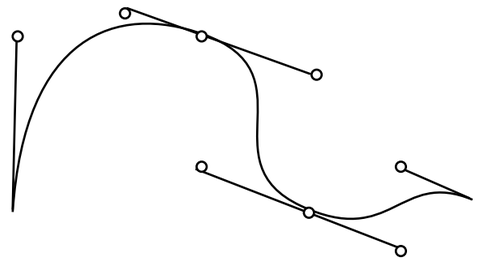
Perlin noise

Voronoi noise

But for the better ones we need splines:



Splines



Originally a drafting tool to create a smooth curve

In computer graphics: a curve built from sections, each described by a 2nd or 3rd degree polynomial.

Very common in non-real-time graphics, both 2D and 3D!

Useful also for real-time.

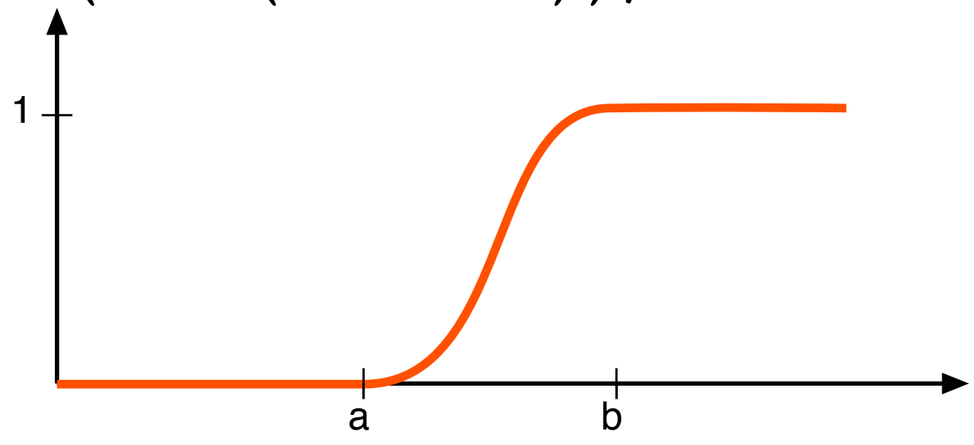
Applications of splines

- Designing smooth curves (common in 2D illustrations)
- Filter design
- Modelling smooth surfaces
- Representating of smooth surfaces (converted to polygons in real-time)
- Animation paths

Smoothstep

A smooth 0-1 transistion

```
float step(float a, float b, float x)
{
    if (x < a)
      return 0;
    if (x > b)
      return 1;
    x = (x - a) / (b - a);
    return (x*x*(3 - 2*x));
}
```



A spline defined by a parametric representation - just a function

But how can we modify this?

Parametric representation

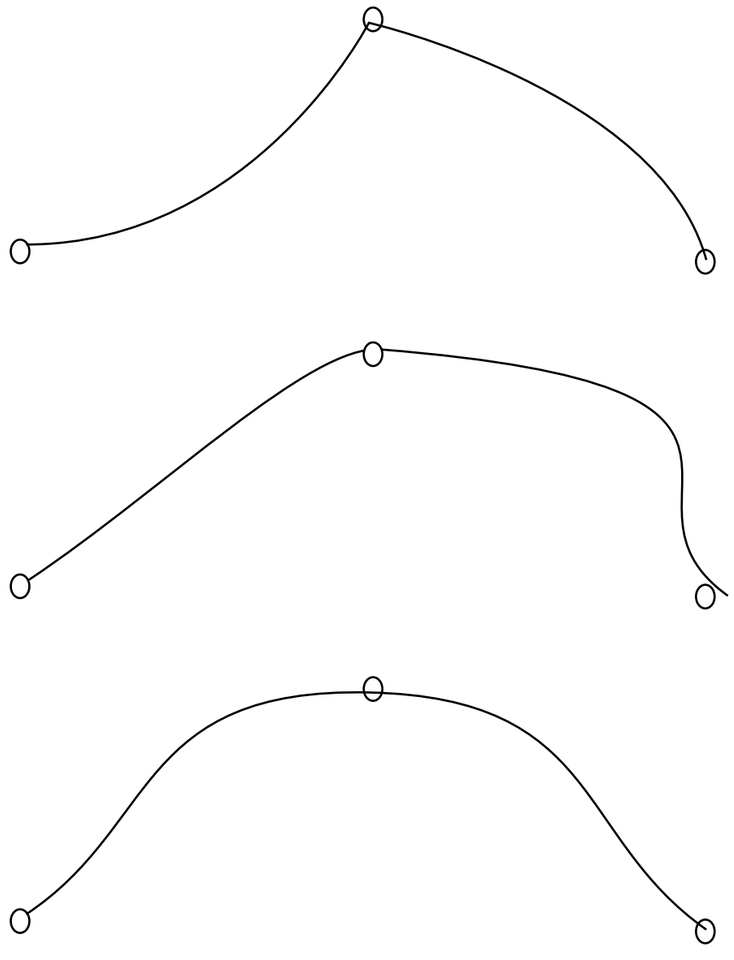
$$x = x(u)$$

$$y = y(u) \qquad u_1 \leq u \leq u_2$$

$$z = z(u)$$

A set of functions for each coordinate

Parametric continuity



C^0 = continuous position
= the curves meet

C^1 = continuous direction
= the curves meet at same angle
and same speed (first derivative)

C^2 = continuous curvature
= the curves meet at same bend and
2nd derivative

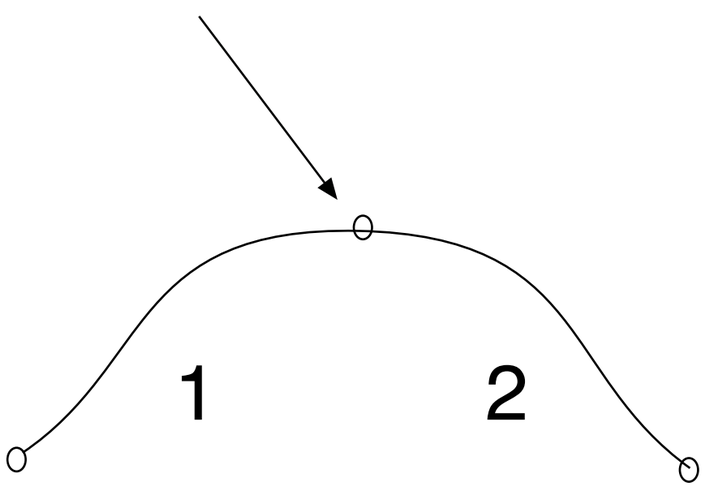
Specification of splines by polynomials in multiple sections

$$x_1(u) = a_{x1}u^3 + b_{x1}u^2 + c_{x1}u + d_{x1}$$

$$y_1(u) = a_{y1}u^3 + b_{y1}u^2 + c_{y1}u + d_{y1}$$

$$z_1(u) = a_{z1}u^3 + b_{z1}u^2 + c_{z1}u + d_{z1}$$

Continuity in this point?

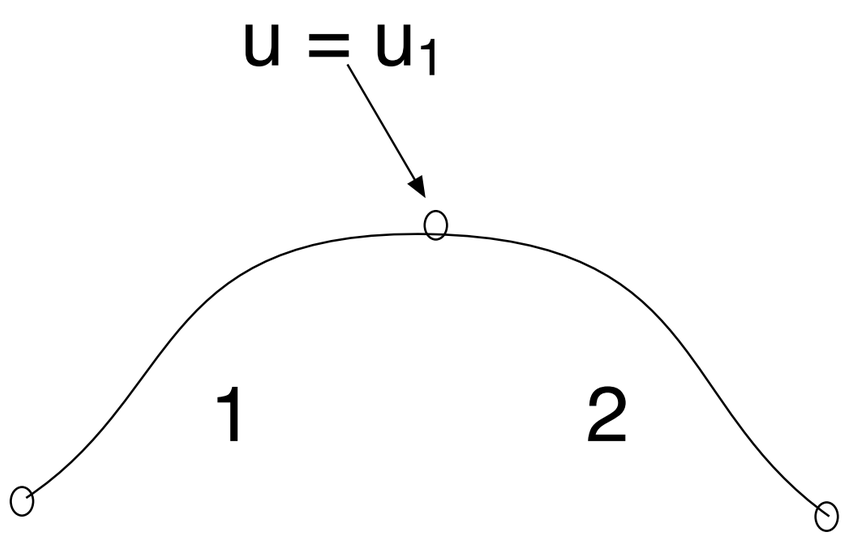


$$x_2(u) = a_{x2}u^3 + b_{x2}u^2 + c_{x2}u + d_{x2}$$

$$y_2(u) = a_{y2}u^3 + b_{y2}u^2 + c_{y2}u + d_{y2}$$

$$z_2(u) = a_{z2}u^3 + b_{z2}u^2 + c_{z2}u + d_{z2}$$

Parametric continuity



C^0:

$$x_1(u_1) = x_2(u_1)$$
$$y_1(u_1) = y_2(u_1)$$
$$z_1(u_1) = z_2(u_1)$$

C^1:

$$x'_1(u_1) = x'_2(u_1)$$
$$y'_1(u_1) = y'_2(u_1)$$
$$z'_1(u_1) = z'_2(u_1)$$

C1: 6 equations per vertex, 12 coefficients per section

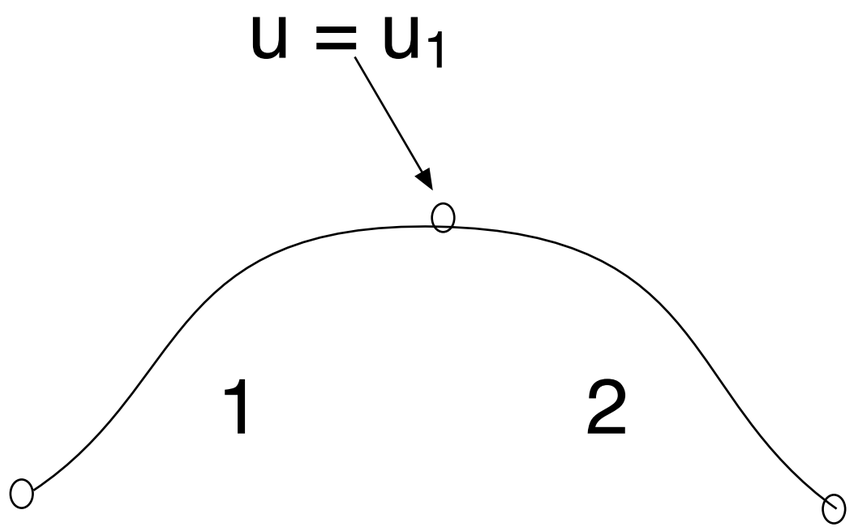
Geometric continuity

$G^0 = C^0$ = continuous position
= the curves meet

G^1 = proportional direction
= the curves meet at same angle
but not same velocity

G^2 = proportional curvature
= the curves meet at same bend but
not same velocity

Geometric continuity



G^0:

$x_1(u_1) = x_2(u_1)$
$y_1(u_1) = y_2(u_1)$
$z_1(u_1) = z_2(u_1)$

G^1:

$x'_1(u_1) = k*x'_2(u_1)$
$y'_1(u_1) = k*y'_2(u_1)$
$z'_1(u_1) = k*z'_2(u_1)'$
for some k

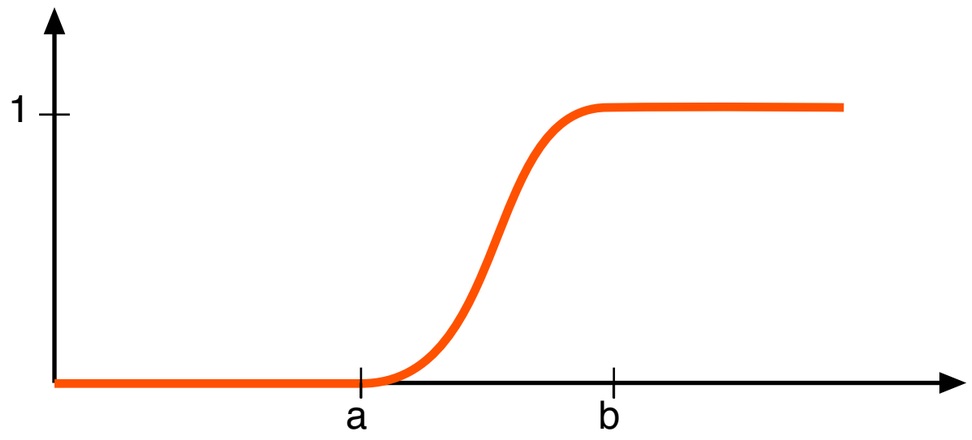
Essentially one less constraint

Smoothstep

A smooth 0-1 transistion

Consists of 3 sections, 2 straight ones.

Continuity in the points a and b desirable.



Can you test if this is G^1, C^1, G^2 or C^2?

$f(x) = 3x^2 - 2x^3$ $\quad f(0) = 0$ $\quad f(1) = 1$

$f'(x) = 6x - 6x^2$ $\quad f'(0) = 0$ $\quad f'(1) = 0$

$f''(x) = 6 - 12x$ $\quad f''(0) = 6$ $\quad f''(1) = -6$

$\Rightarrow C^1$ and G^1, but not C^2 or G^2!

Smoothstep is constructed by the constraints

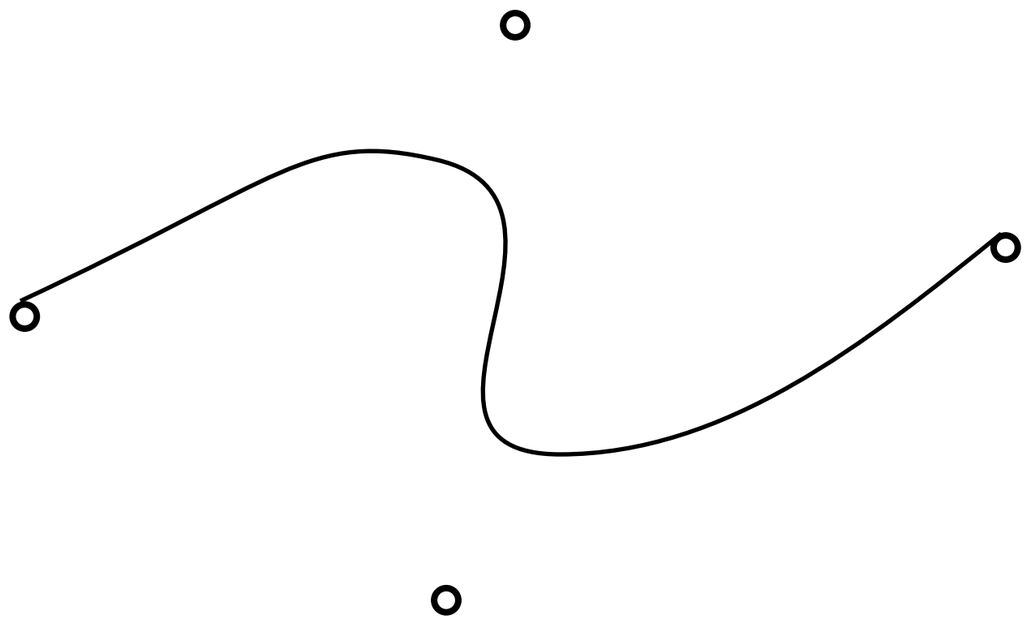
but a common alternative is:

Blending functions

Rewrite parametric form to a set of polynomials, one polynomial for each control point

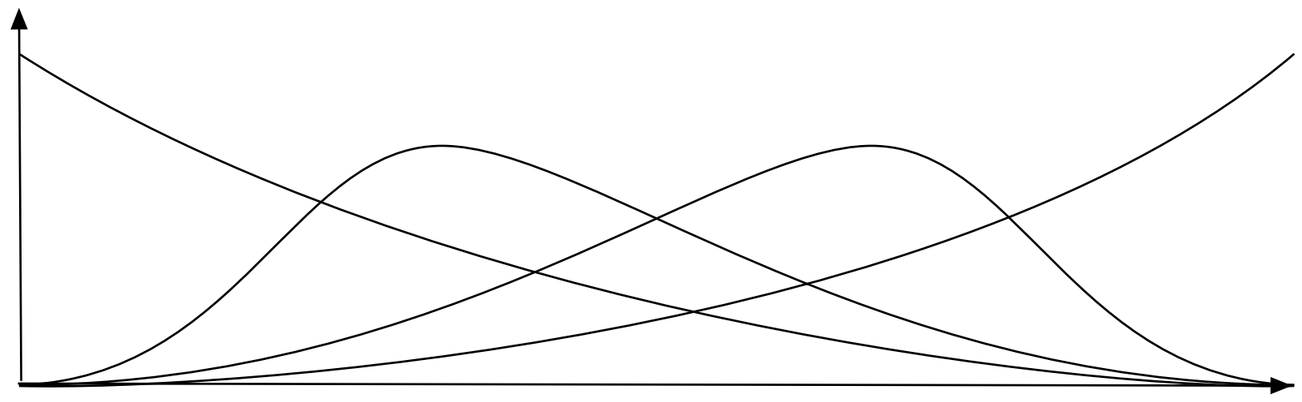
Bézier curves

Typically use 3 or 4 control points per section



Bézier curves

The 4 points are blended together using 4 blending functions



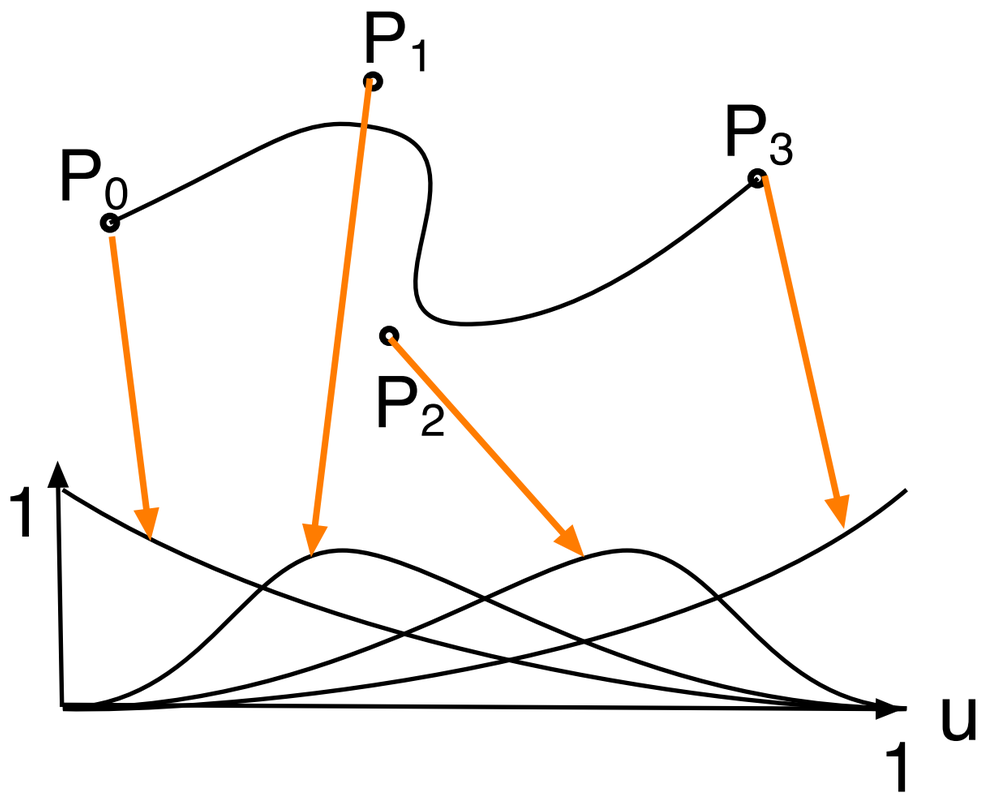
4 blending functions = Cubic Bézier

Bézier curves

Blending functions:
Bernstein polynomials

$$BEZ_{0,3} = (1-u)^3$$
$$BEZ_{1,3} = 3u(1-u)^2$$
$$BEZ_{2,3} = 3(1-u)u^2$$
$$BEZ_{3,3} = u^3$$

The sum is 1 for any u



$$BEZ_{0,3} = (1-u)^3$$
$$BEZ_{1,3} = 3u(1-u)^2u$$
$$BEZ_{2,3} = 3(1-u)u^2$$
$$BEZ_{3,3} = u^3$$

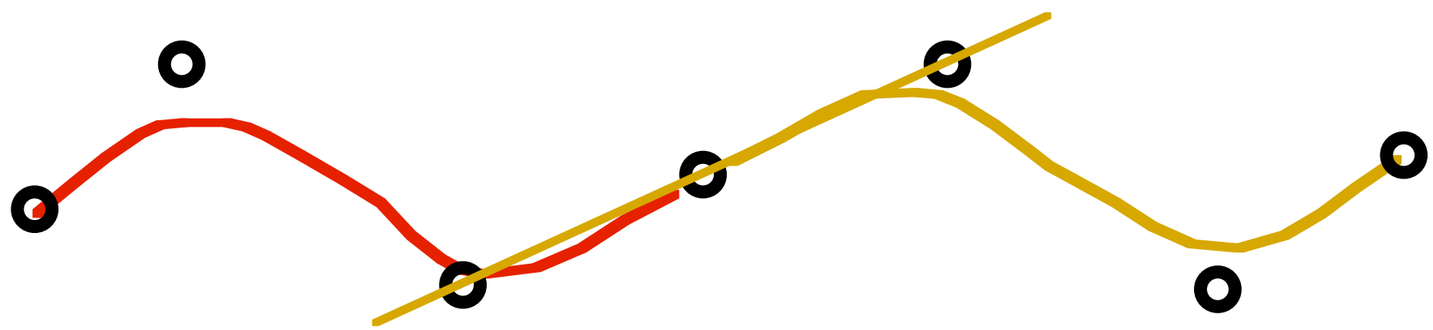
$$P(u) = P_0*(1-u)^3 + P_1*3u(1-u)^2 + P_2*3(1-u)u^2 + P_3*u^3$$

$$= \sum_{i=0}^{3} P_i * BEZ_{i,3}(u)$$

Fitting together sections

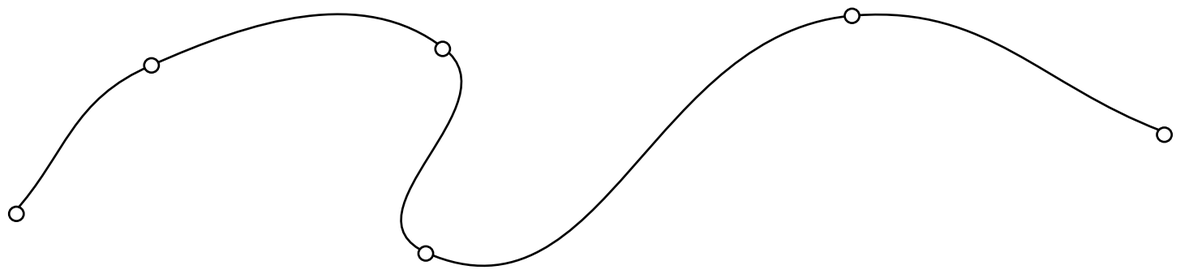
C^0/G^0 continuity: just fit the points

C^1 continuity: Tangents are equal along the edge.
G^1 continuity: Tangents have same direction along the edge.
Simple method: Put 3 points in a line, equidistant for C_1



Interpolation splines

Passes through all control points.

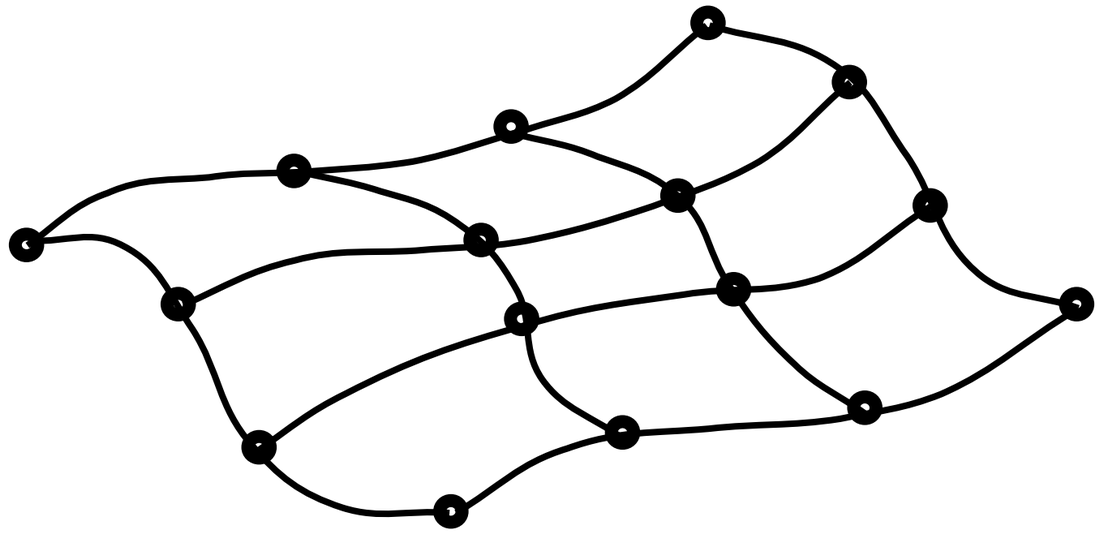


Control points on the curve.
e.g. Catmull-Rom
Not that relevant in this course.

Bézier surfaces

A surface is built from a set of Bézier patches

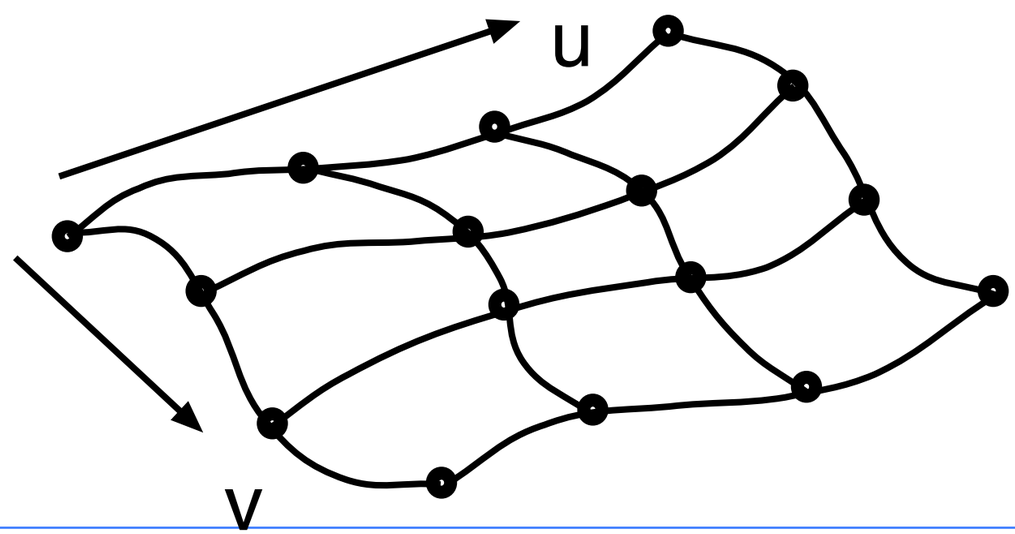
A Bézier patch consists of 16 control points in a 4x4 grid



Bézier surfaces

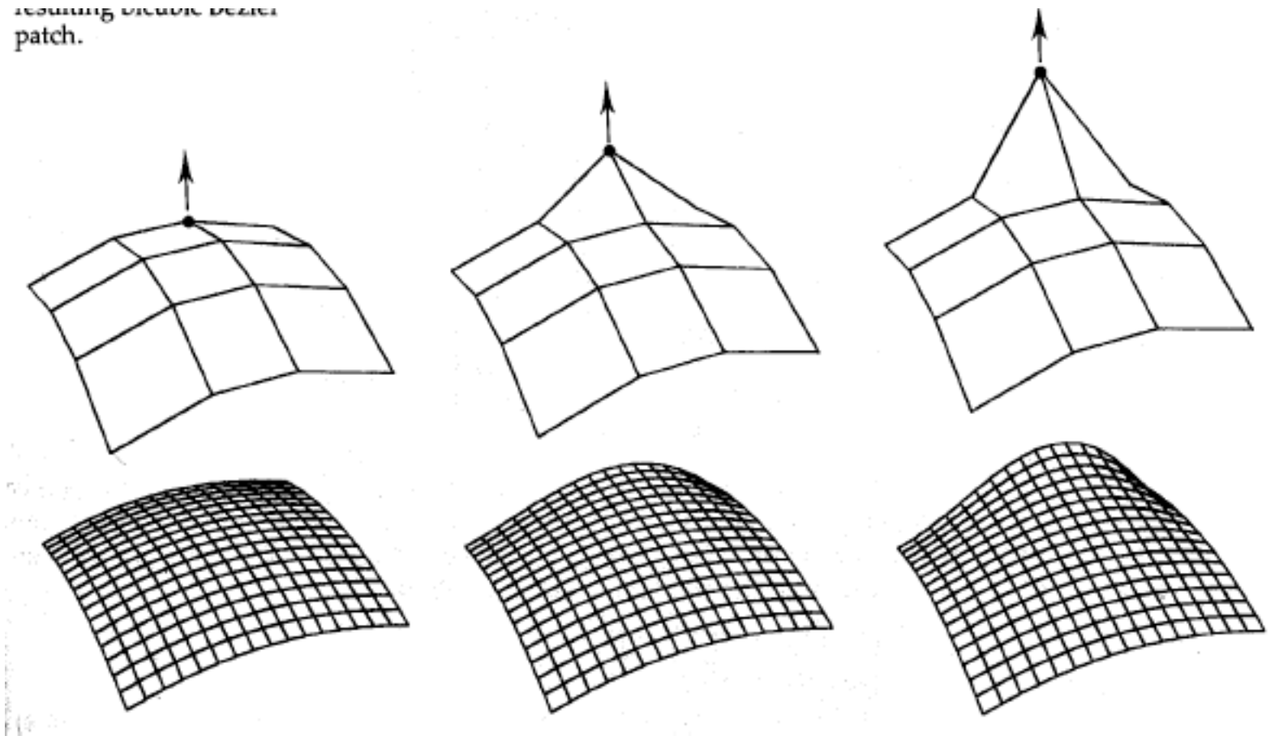
Blending of the 16 control points as a 2-dimensional sum

$$P(u,v) = \sum_{j=0}^{3} \sum_{k=0}^{3} p_{j,k}\, BEZ_{j,3}(v)\, BEZ_{k,3}(u)$$



Bézier surface example

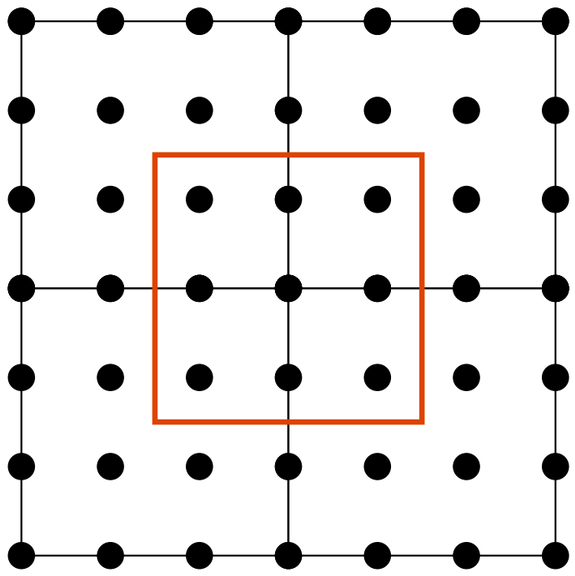
patch.

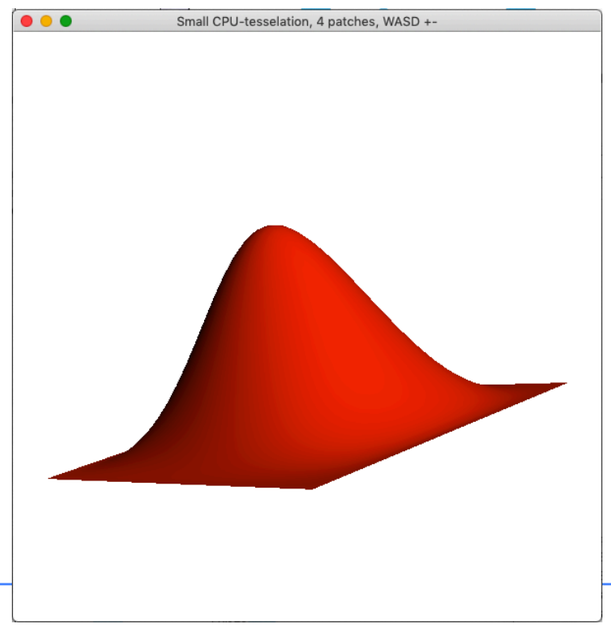


Fitting together patches

Fit in both u and v direction

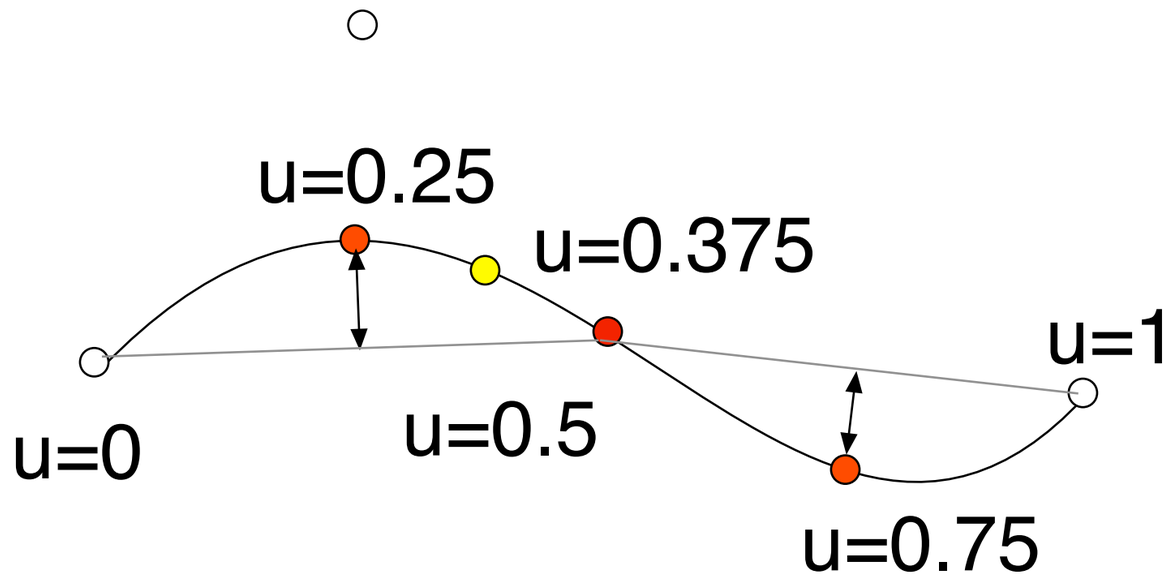
Make a 3x3 “joystick” at each corner





Drawing splines

Subdivide the spline until the error is small enough.



Splines and surfaces in OpenGL

Pre-generated shapes on CPU

Generate by multi-pass GPU processing

Old OpenGL: Evaluators (glMap)

3.2: Geometry shaders

4: Tesselation shaders

For this course, consider splines for:

Smooth edges, anti-aliasing, smoothing noise

Constructing and representing shapes

Building smooth surfaces procedurally by Bézier patches

and more

so let's smooth some noise!

Value noise

Random intensities

White noise

High frequency

Zoomed pixels = random rectanges

Not very useful BUT...

Interpolation of noise

Can we make smooth functions from noise?

Why do we want smooth functions?

- Curved shapes

- Narrow-banded - easy to control, no unwanted "spikes" in the frequency spectrum

Interpolation of white noise

- Nearest neighbor
- Linear
- Smoothstep
- Cubic spline

Interpolation is a reconstruction

If it is a sampling, what is the best reconstruction of the original signal?

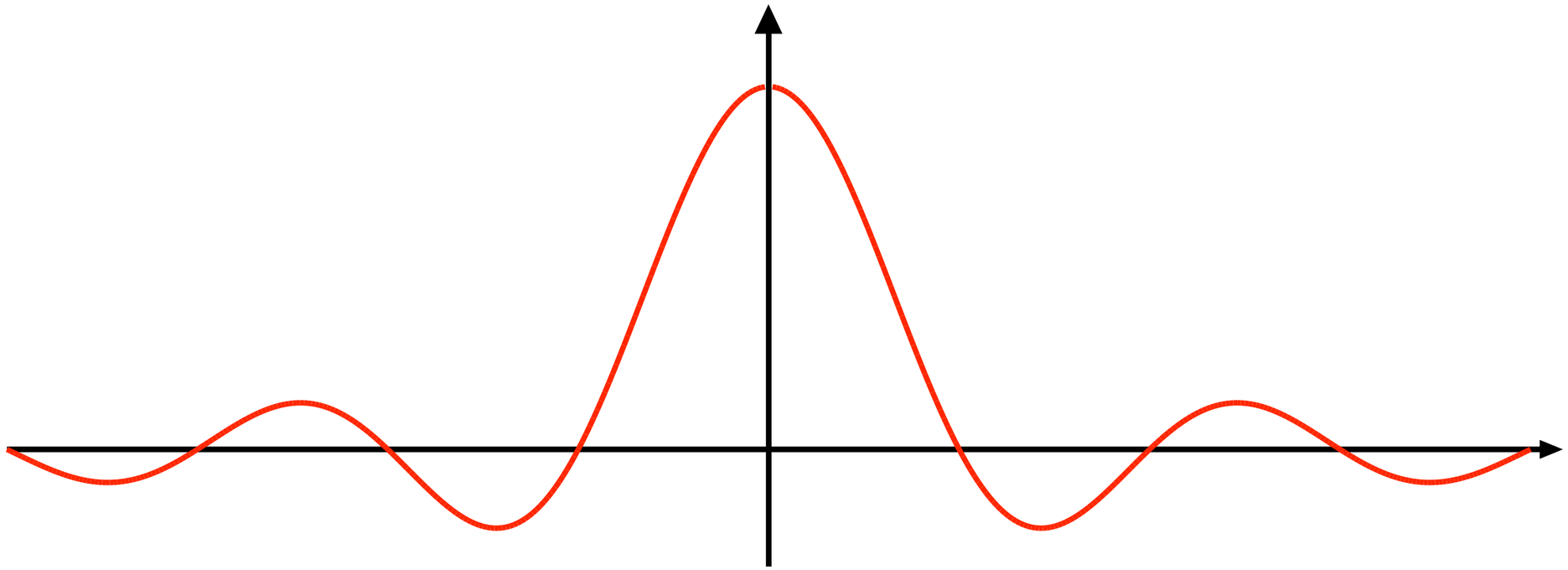
For a 1D signal, it can be proven that the ideal reconstruction is made by a *sinc* function

$$\mathrm{sinc}(x) = \sin(\pi x)/\pi x$$

Thus, the closer to sinc, the better reconstruction!

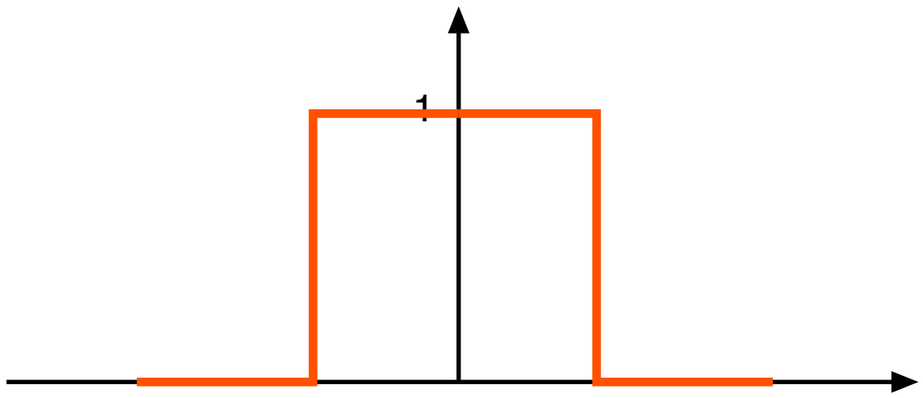
Sinc goes to infinity

So if we want to use it directly, we must cut off somewhere.

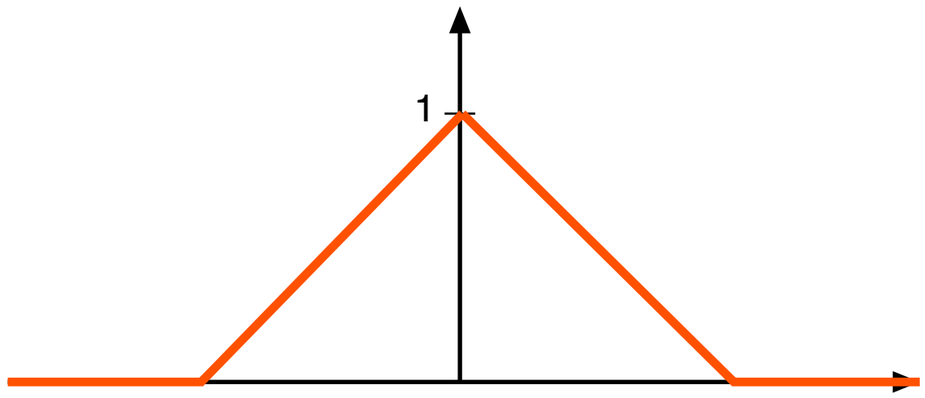


Sinc in the frequency plane

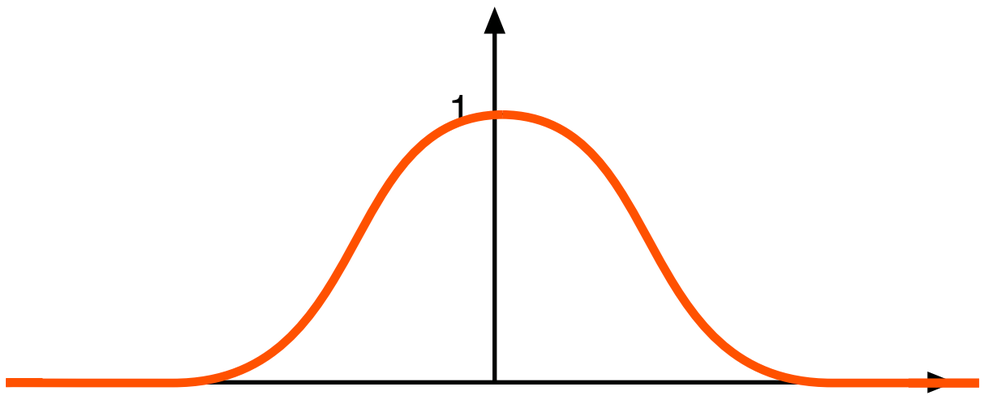
Sinc is a step function in the frequency plane = ideal low-pass filter = best possible smoothing function



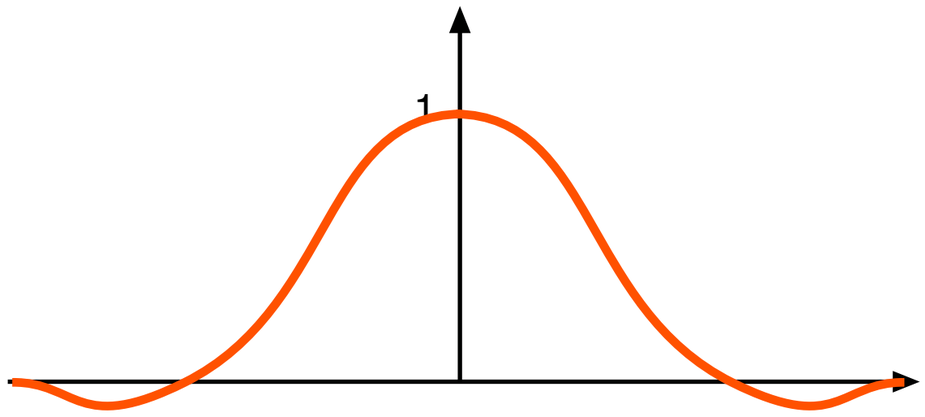
Nearest neighbor



(Bi)linear interpolation



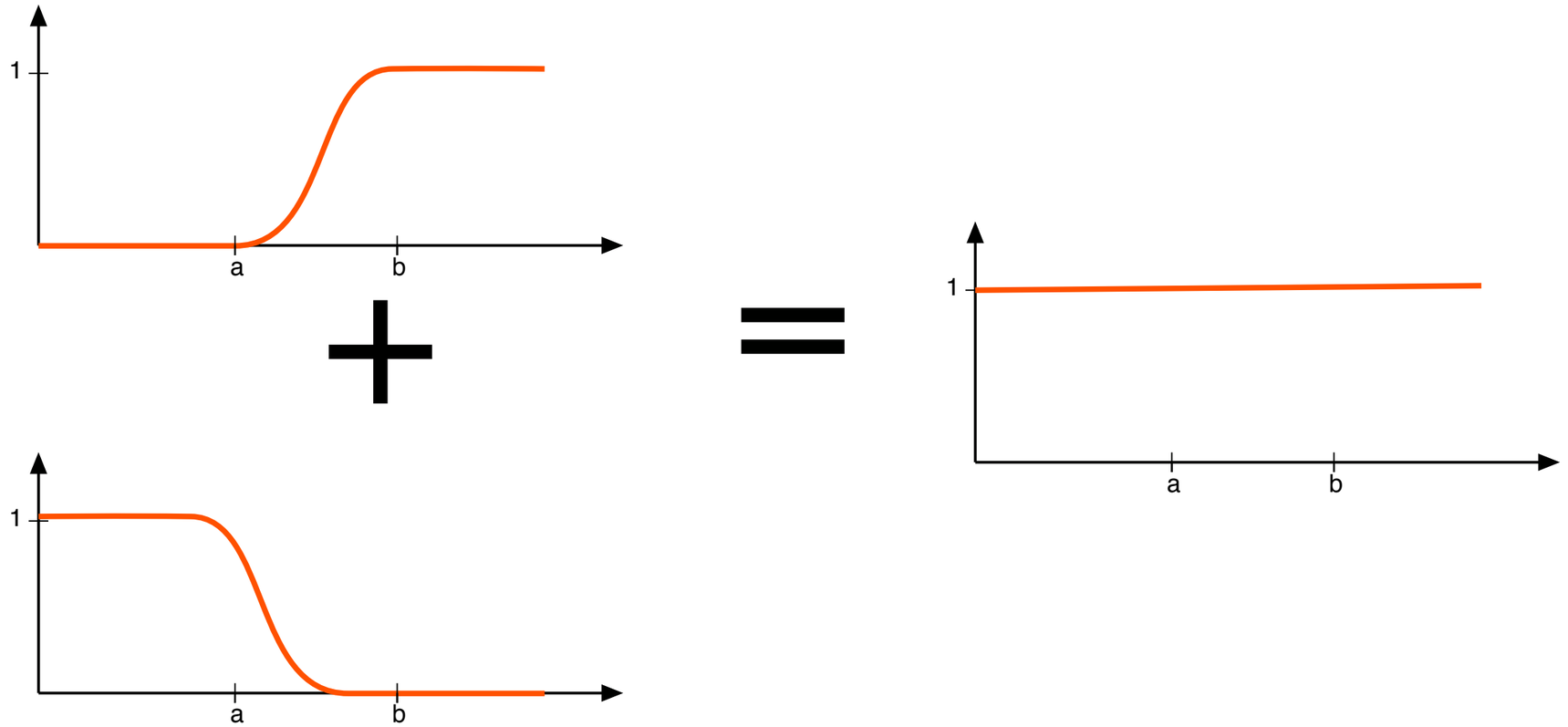
Hermite/smoothstep interpolation



Piecewise (bi)cubic interpolation
"Cubic spline"

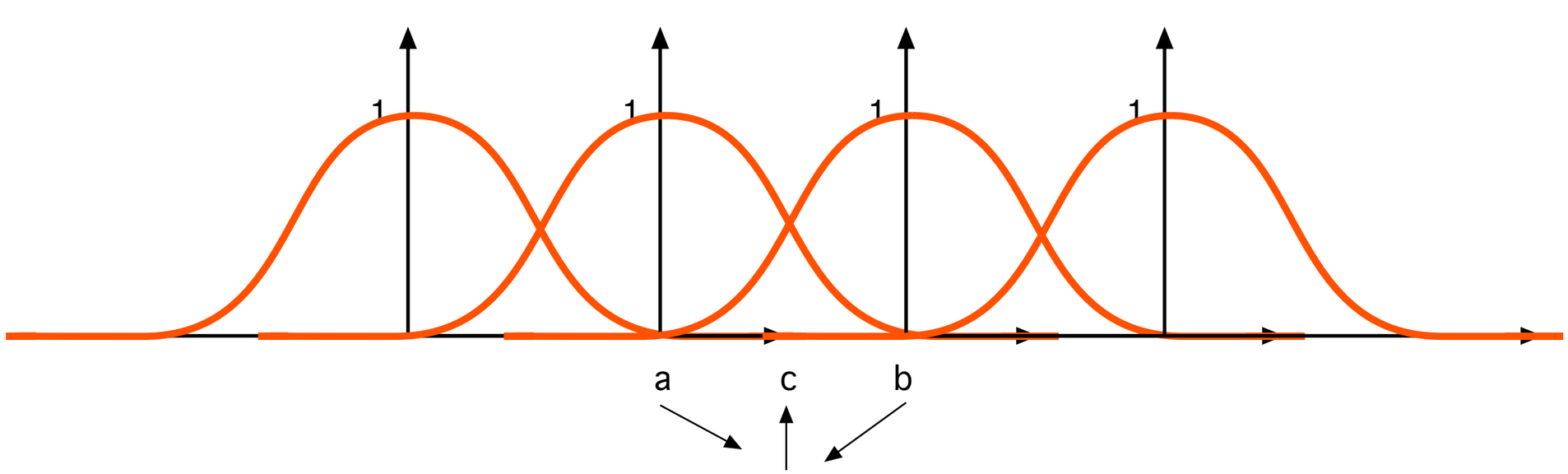
Sum to 1

An interpolation function should sum to 1



It really works like this

Every pixel affects a limited area.



To calculate a pixel c between a and b, take a and b, multiply by the weight for each given by the posotion of c and sum

For nearest neighbor and linear interpolation:

Trivial! (Right?)

For smoothstep:

$$\text{step}(x) + \text{step}(1-x) = x^2(3 - 2x) + (1-x)^2(3-2(1-x))$$

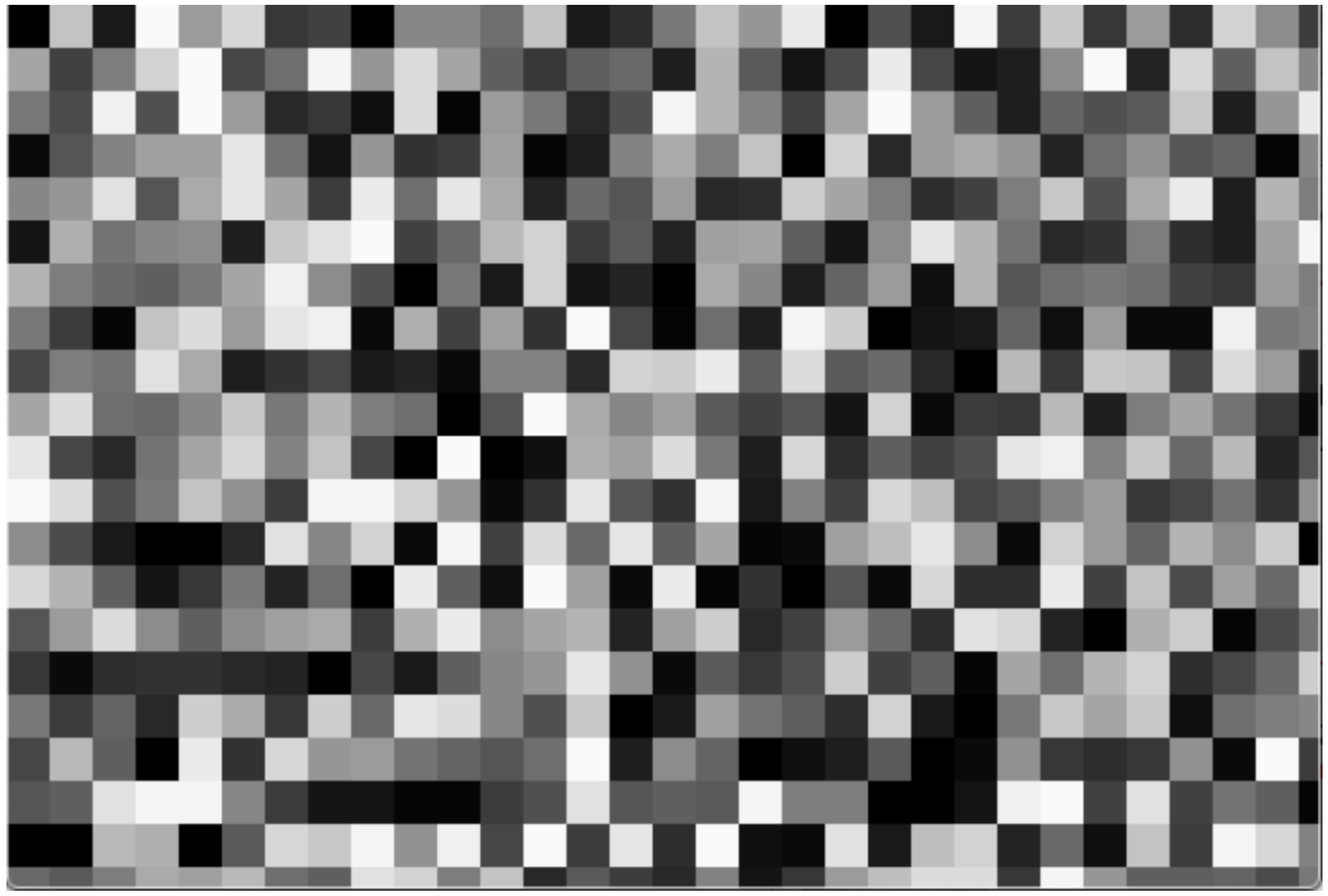
$$= 3x^2 - 2x^3 + 3 - 6x + 3x^2 - 2 + 6x - 6x^2 + 2x^3 = 1$$

For cubic spline:

Not right now please... (Sum over 4 samples)

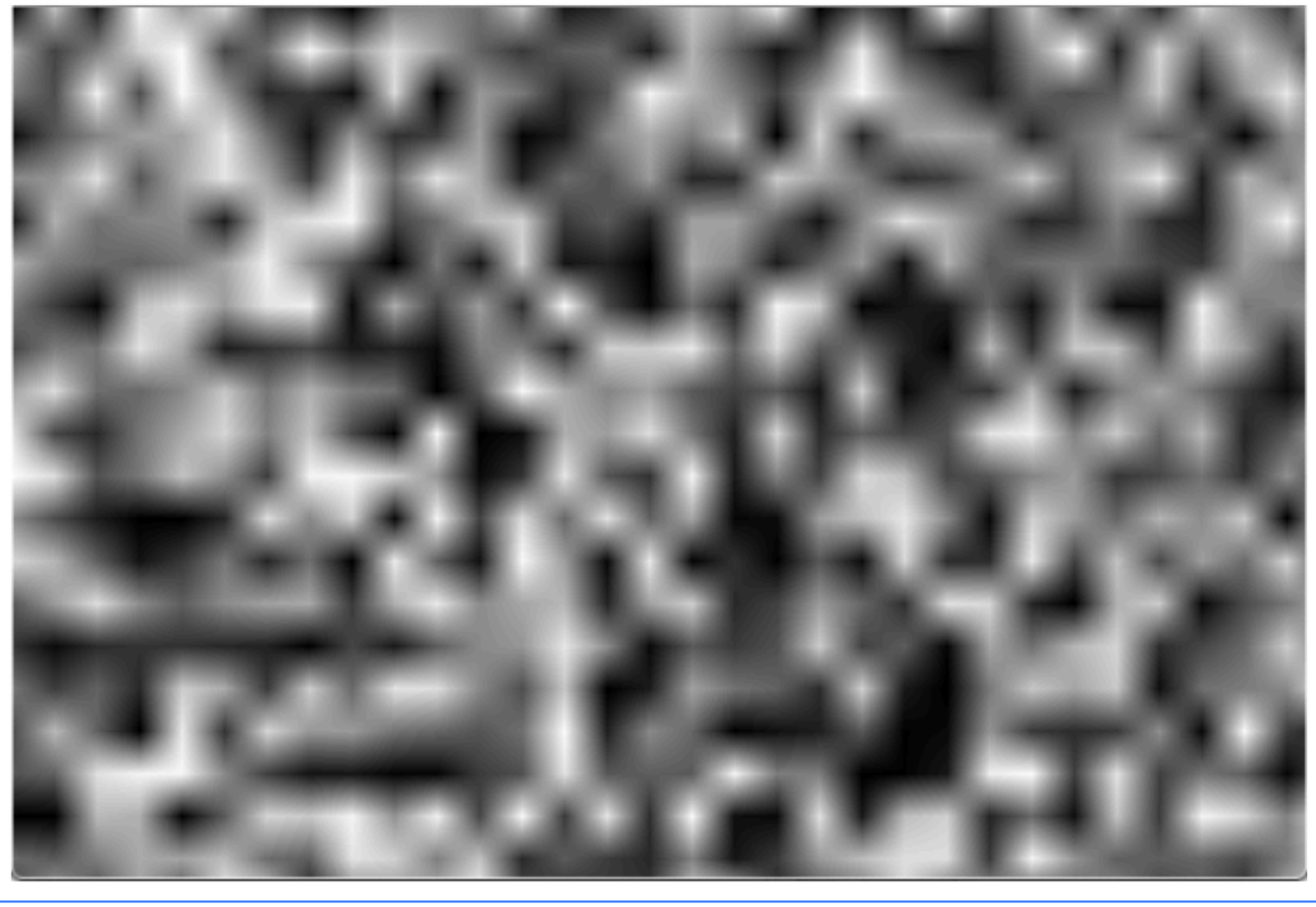
Nearest neighbor

Not smooth at all!



Bilinear interpolation

Better but with visible artifacts



Smoothstep

"Hermite filter". Smooth but a bit blocky

Piecewise bicubic interpolation ("cubic spline")

Good approximation of sinc. Close to optimal. Can "overshoot".

Gradient noise (Perlin noise)

Random numbers for pixel values is pretty good... but tends to be "blocky" even with interpolation. High quality interpolation is good but has some overshoot.

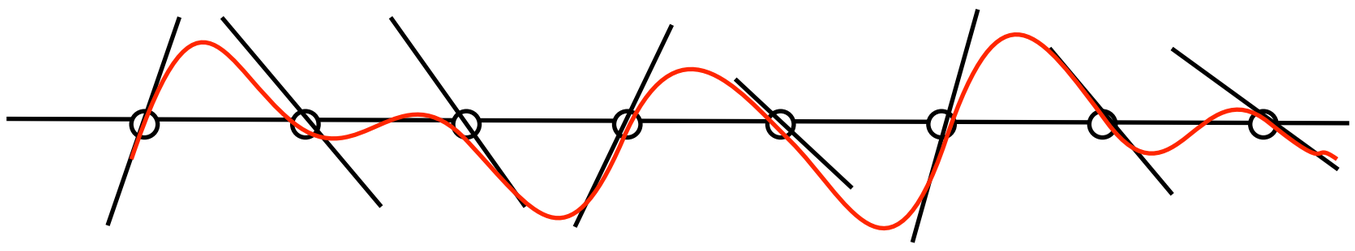
It is also dominated by high frequencies. We will return to that issue.

A more visually appealing noise: Random gradients.

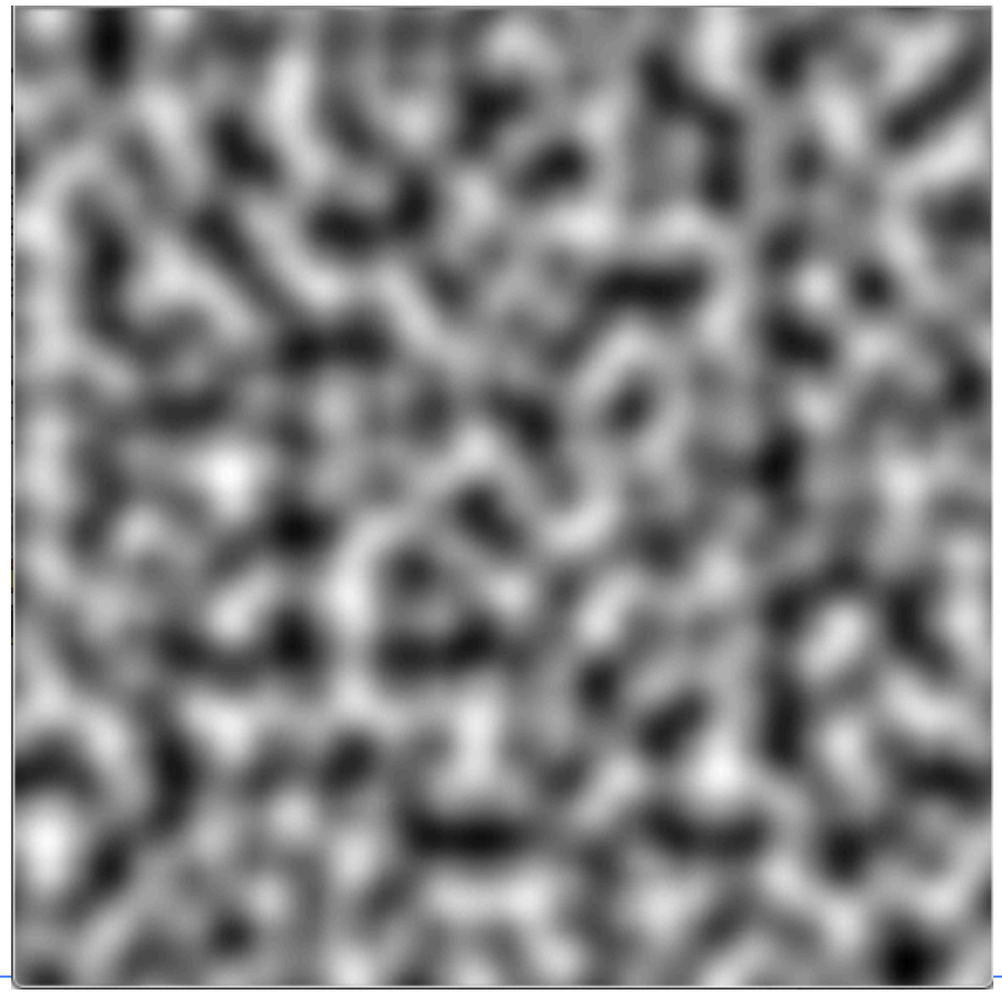
Gradient noise

If the values are used for gradients instead of height, we get gradient noise. (Perlin noise.)

The function is interpolated to match the gradients.



Gradient/Perlin noise



Implementation

Not Perlin's but still gradient noise. Brief version from the lab

```
vec2 random2(vec2 st)
{
    st = vec2( dot(st,vec2(127.1,311.7)),
               dot(st,vec2(269.5,183.3)) );
    return -1.0 + 2.0*fract(sin(st)*43758.5453123);
}

// Gradient Noise by Inigo Quilez - iq/2013
// https://www.shadertoy.com/view/XdXGW8
float noise(vec2 st)
{
    vec2 i = floor(st);
    vec2 f = fract(st);

    vec2 u = f*f*(3.0-2.0*f);

    return mix( mix( dot( random2(i + vec2(0.0,0.0) ), f - vec2(0.0,0.0) ),
                     dot( random2(i + vec2(1.0,0.0) ), f - vec2(1.0,0.0) ), u.x),
                mix( dot( random2(i + vec2(0.0,1.0) ), f - vec2(0.0,1.0) ),
                     dot( random2(i + vec2(1.0,1.0) ), f - vec2(1.0,1.0) ), u.x), u.y);
}
```

fract-based random numbers

Integer and fractional parts

Spline/smoothstep

Interpolate 4 nearest

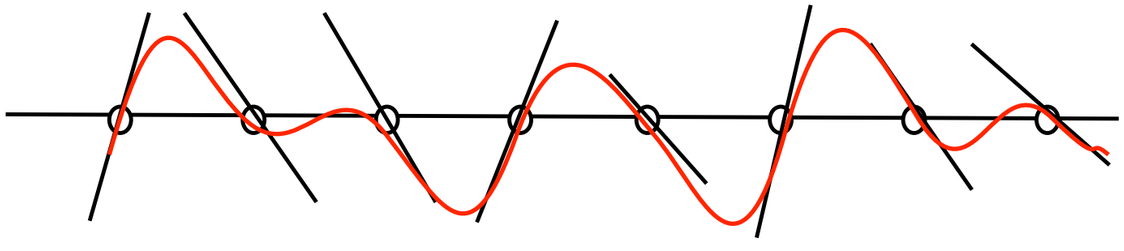
Artifacts

Perlin noise is incomplete! It "locks" in certain points, only producing certain phases of the signal.

I.e. produce only the cosine part of a signal and skipping the sin!

This can be corrected by generating two sets of the signal, with a proper offset!

Usually ignored.



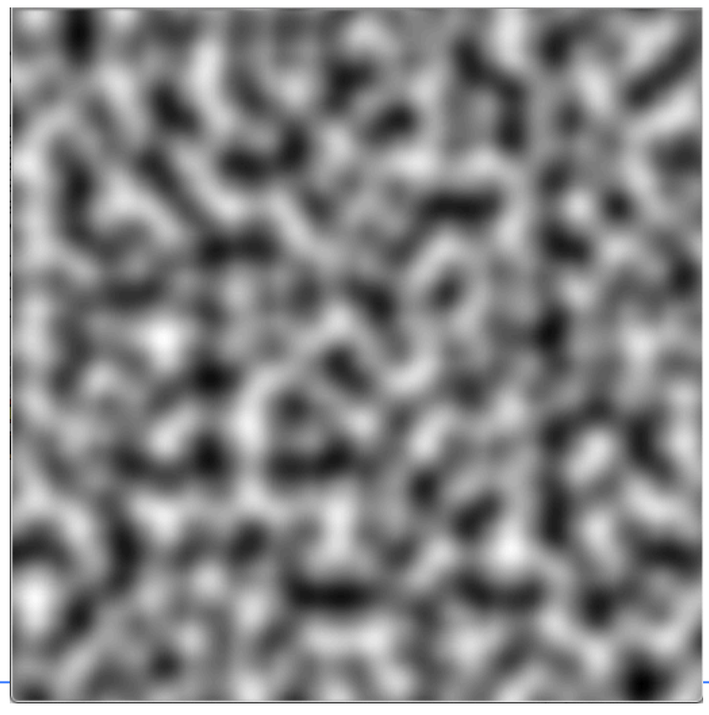
Did this really help?

Perlin vs cubic spline

Cubic splines can overshoot sometimes

Pelin noise rather "undershoots", the function tends to be low.

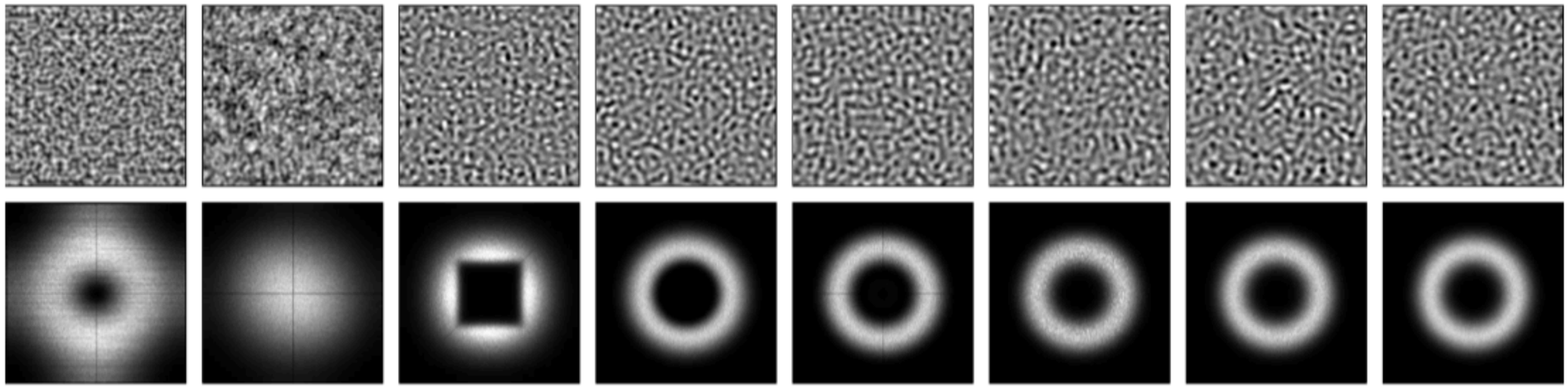
Perlin clearly better at diagonal shapes



Smooth noise = limited bandwidth

Ideally a smooth circle in frequency space

Limited information but can be combined with more.
More about that later.



(a) Perlin noise. (b) Lewis' noise. (c) Wavelet noise. (d) Goldberg's. (e) Kensler's. (f) Ours, CPU. (g) Ours, GPU. (h) Ours, valid.

from **Procedural Noise using Sparse Gabor Convolution,** Ares Lagae et al

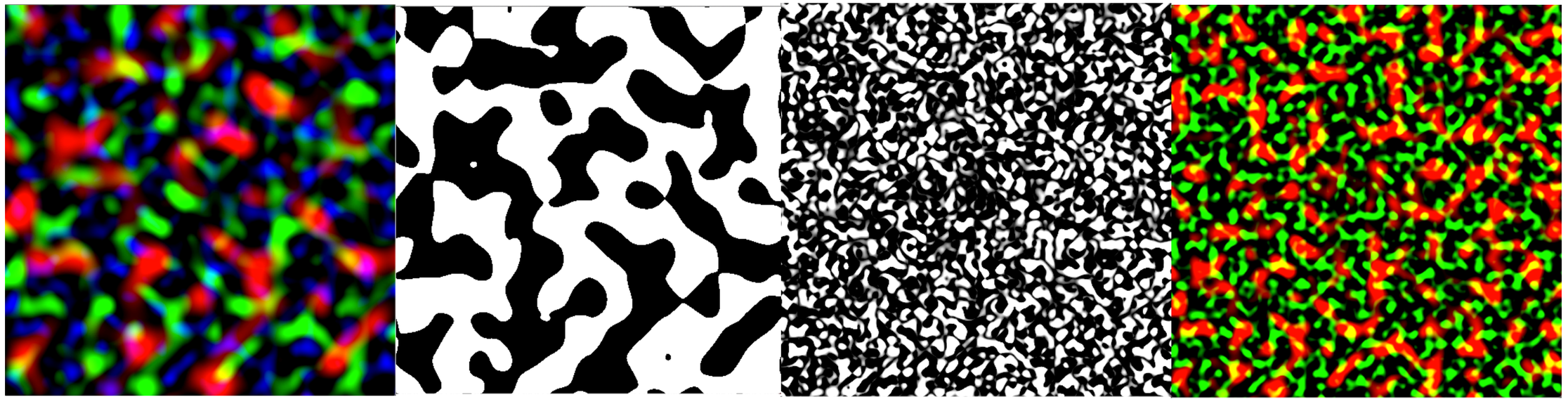
Usages of smooth noise

Treshold for interesting shapes

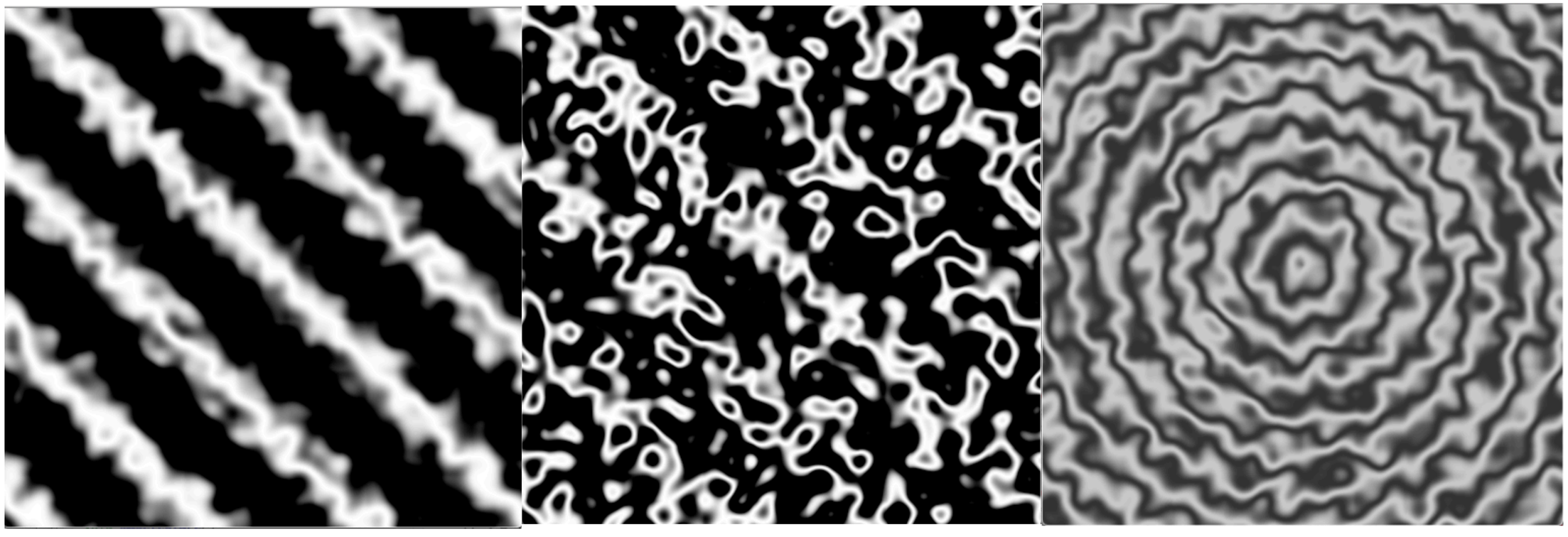
Shape modulation, use an existing shape and modulate the positions

Combine with other data (including other noise) for interesting effects

Just messing about...



Mix with other functions, here sin:



Explore the possibilities on lab 1!

- Simple patterns
- Noise-based patterns
- GPU implementation

Impress me!

But we can do even better:

Simplex noise

Gradient noise based on triangles/tetrahedrons. Ken Perlin's replacement for "Perlin noise" (gradient noise on quads).

Advantage: Fewer points affect each output.

Also applied other computing strategies for higher performance.