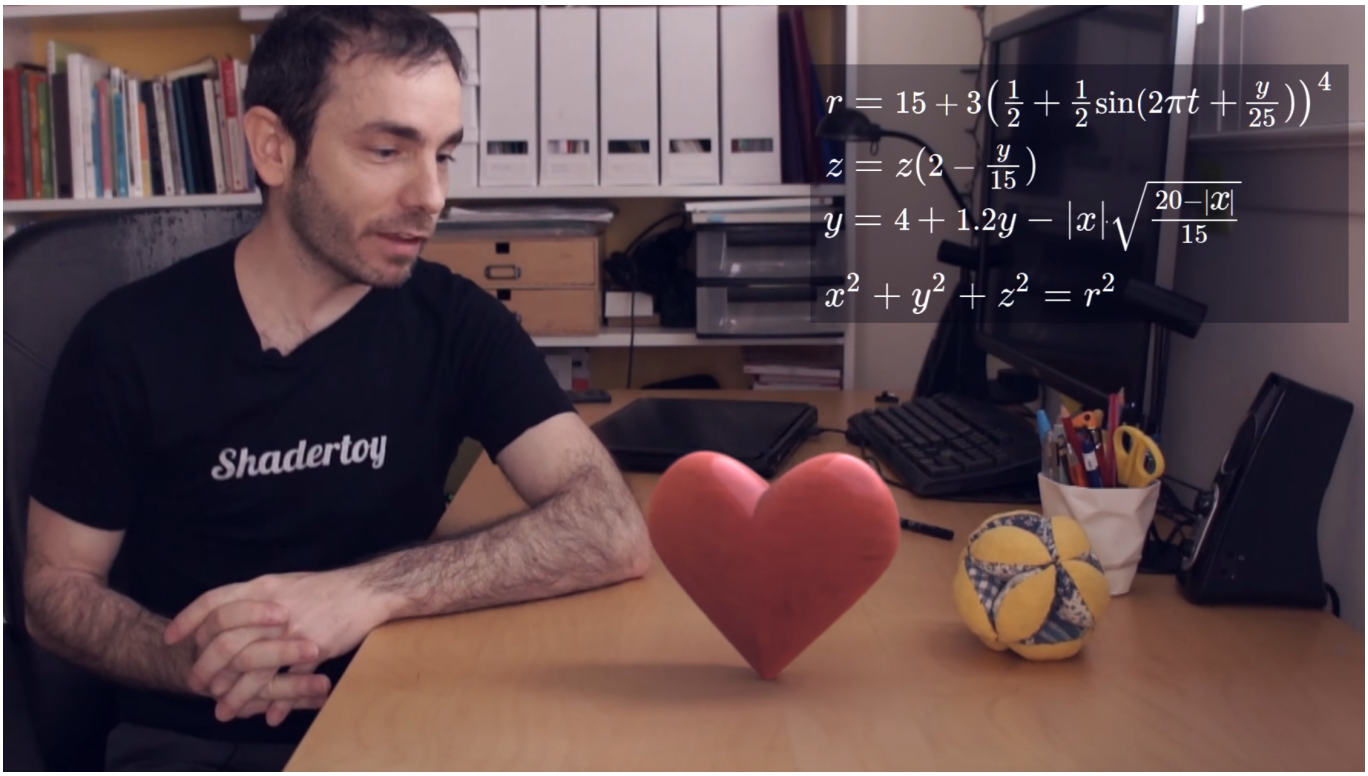
Procedural Generation and Simulation

Prof. Dr. Lena Gieseke | l.gieseke@filmuniversitaet.de | Film University Babelsberg KONRAD WOLF

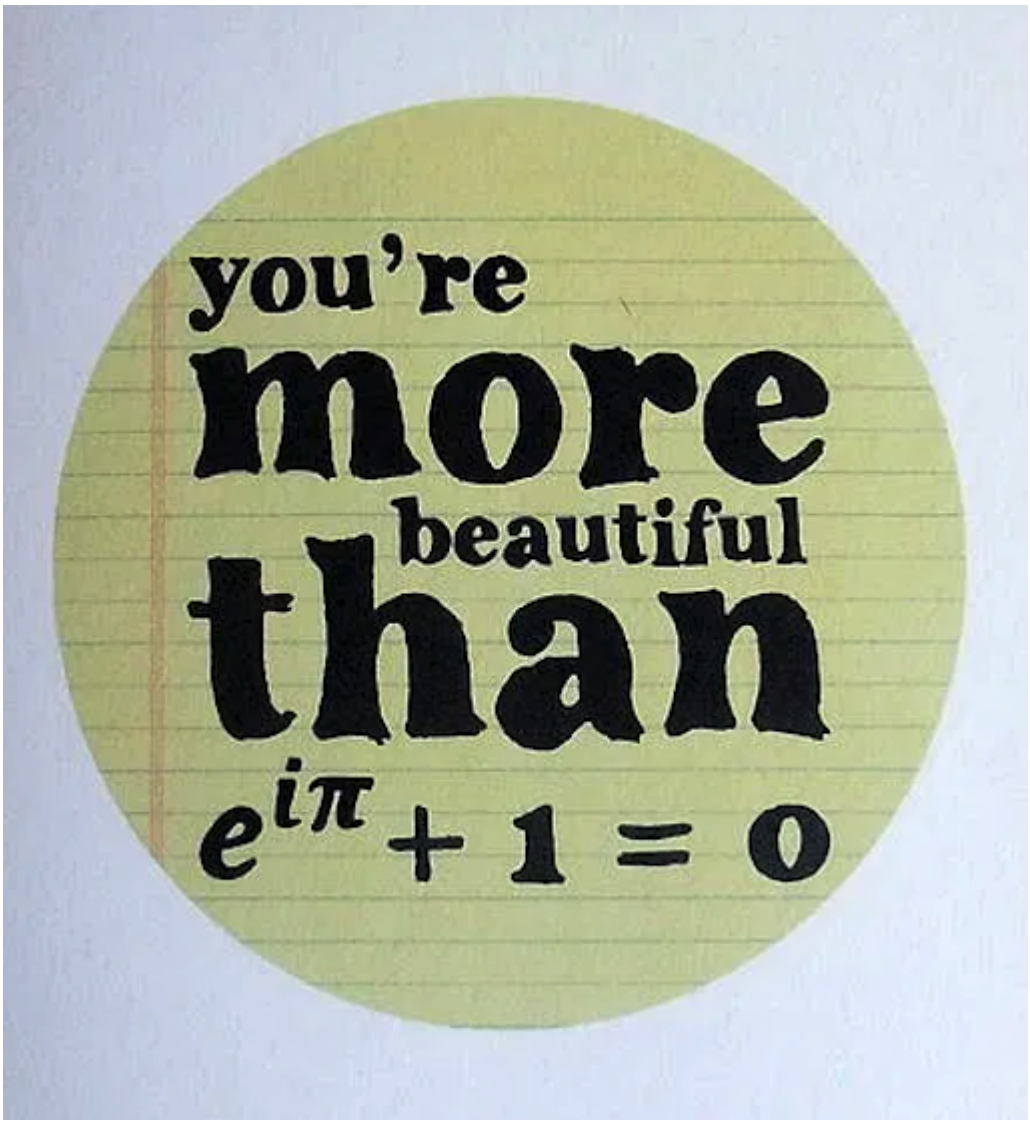
Chapter 04 - Function Design

In the last chapter we were looking at existing numbers and curve equations. Even though that was a lot of fun and beautiful those equations are also somewhat limited in their possible usage and designs. Hence, in this chapter we are going to have a look into how to combine functions to come up with individual designs.

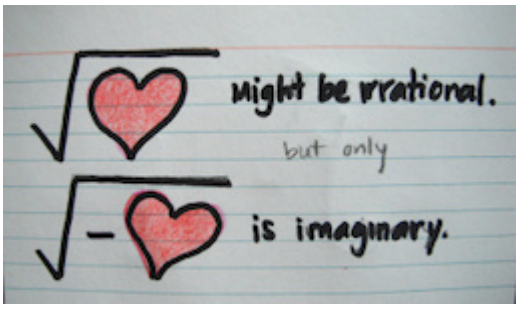
For function designs the ultimate rock star / god / person of incredible awesomeness is Inigo Quilez. His articles page is a resource of unmeasurable value. If interested, check out his explanations on Making a heart with mathematics (which is a somewhat advanced example though).



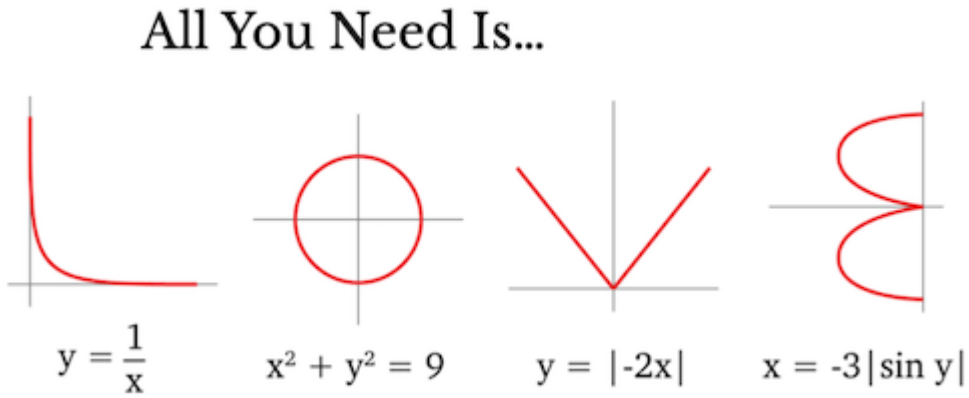
If you ever try to win the heart of a mathematician, here some inspiration:



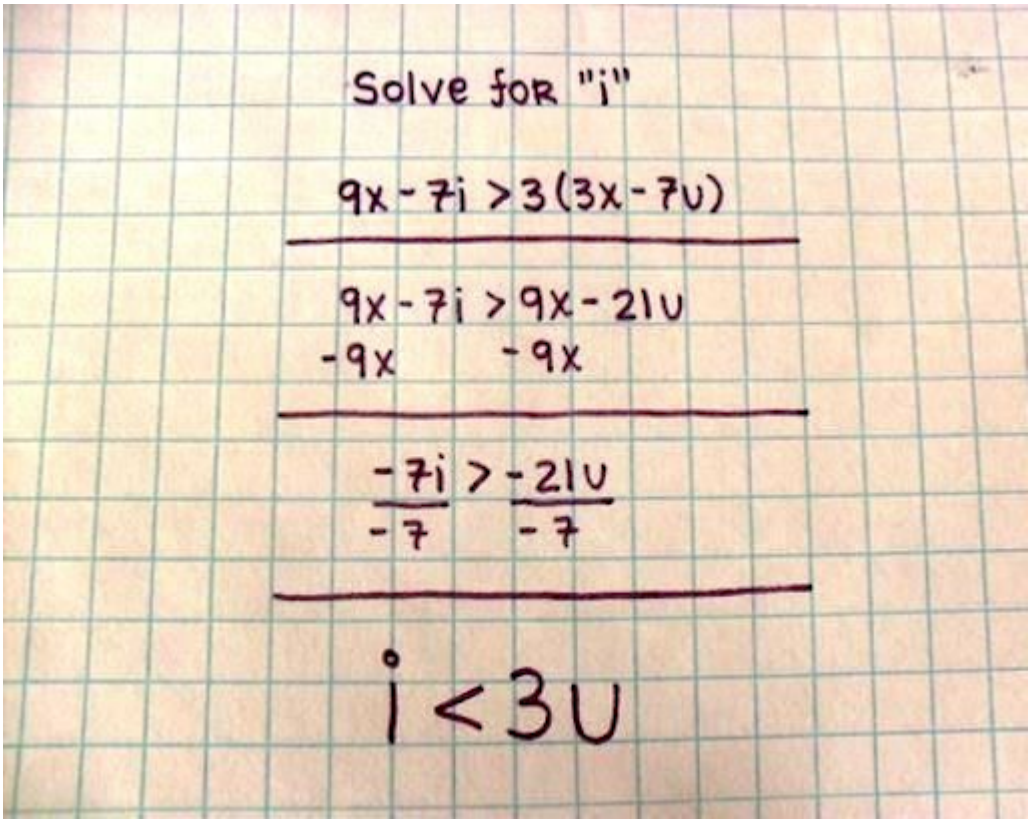
[notonthehighstreet]



[melaniedreams]



[mor.news]



[pinimg]

or TED Talk: The Mathematics of Love by Hannah Fry.

You are welcome.

Topics

- Transitions
- Primitive Components
- Periodicity

Learning Objectives

With this chapter, you will

- gain an intuitive understanding of the visual qualities of different operators and function components.
- gain some understanding of how to put the different function components together to create a specific design goal.

How does the following look like when plotted in a 2D cartesian coordinate system?

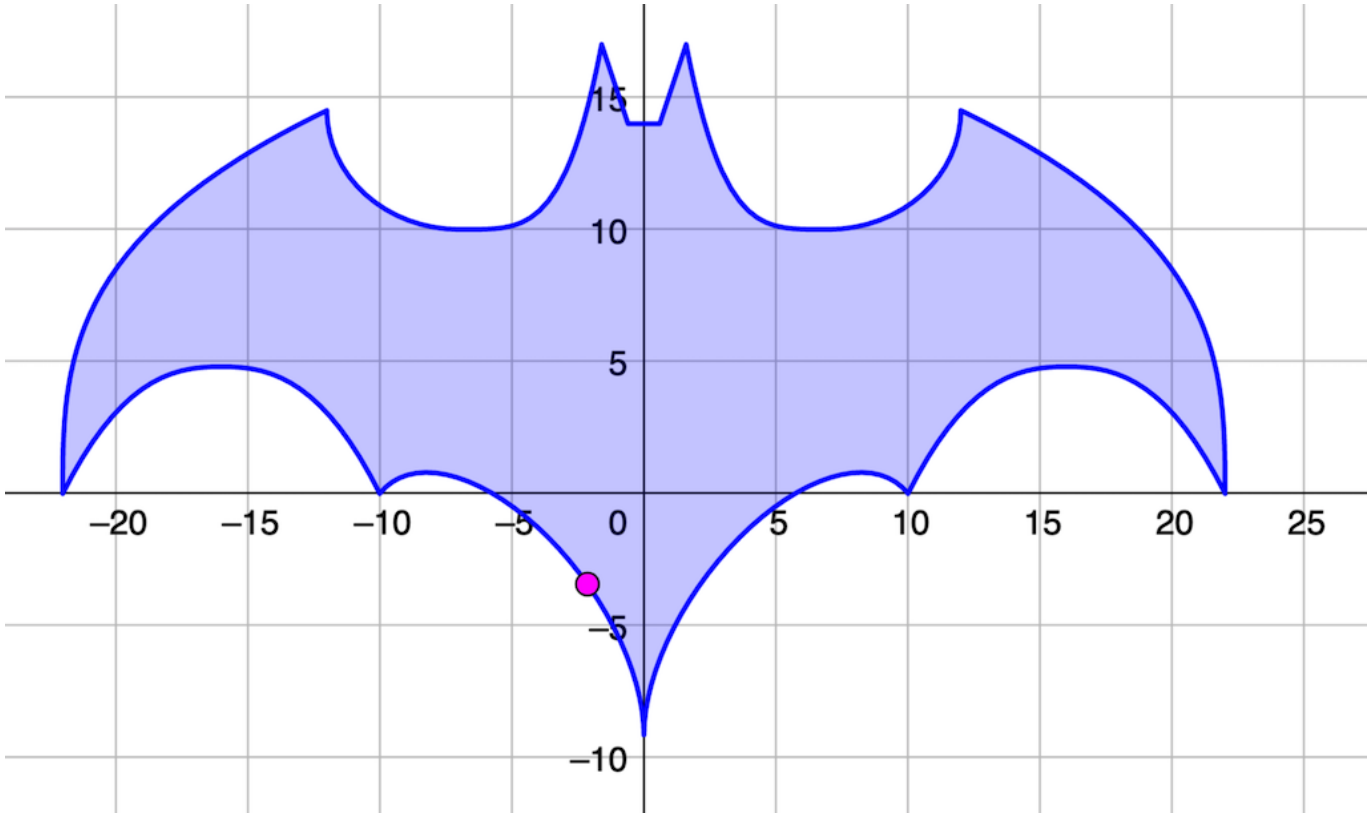
$$x(t) = \frac{|t|}{t}\left(.3|t| + .2||t| - 1| + 2.2||t| - 2| - 2.7||t| - 3| - 3||t| - 5| + 3||t| - 7| \right.$$
$$+ 5\sin\left(\frac{\pi}{4}(||t| - 3| - ||t| - 4| + 1)\right) + \frac{5}{4}(||t| - 4| - ||t| - 5| - 1)^3$$
$$\left. - 5.3\cos\left(\left(\frac{\pi}{2} + \sin^{-1}\frac{47}{53}\right)\left(\frac{||t| - 7| - ||t| - 8| - 1}{2}\right)\right) + 2.8\right)$$

$$y(t) = \frac{3}{2}||t| - 1| - \frac{3}{2}||t| - 2| - \frac{29}{4}||t| - 4| + \frac{29}{4}||t| - 5| + \frac{7}{16}(||t| - 2| - ||t| - 3| - 1)^4$$
$$+ 4.5\sin\left(\frac{\pi}{4}(||t| - 3| - ||t| - 4| - 1)\right) - \frac{3\sqrt{2}}{5}\left||t| - 5| - ||t| - 7|\right|^{\frac{5}{2}}$$
$$+ 6.4\sin\left(\left(\frac{\pi}{2} + \sin^{-1}\frac{47}{53}\right)\left(\frac{||t| - 7| - ||t| - 8| + 1}{2}\right) + \sin^{-1}\frac{56}{64}\right) + 4.95$$

$$-8 \le t \le 8$$

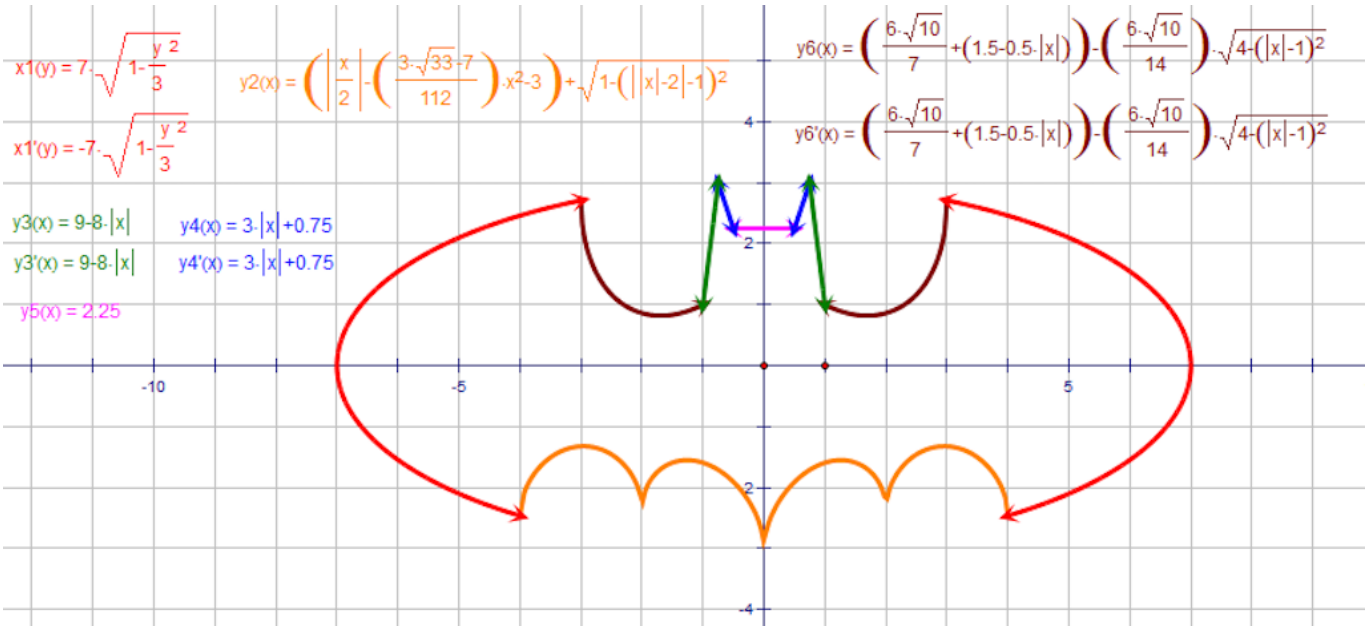
[geogebra]

You are right, it is Batman!



[geogebra]

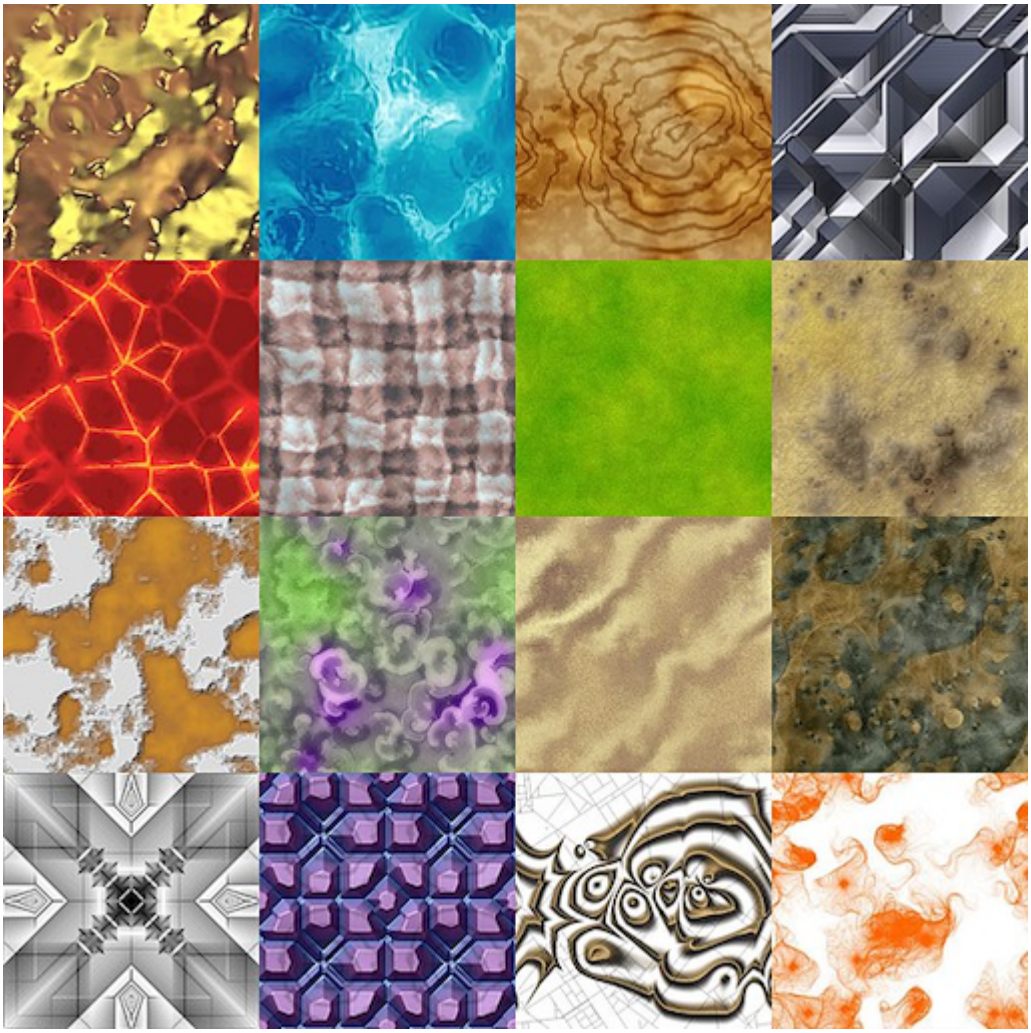
By slicing together different functions, you can archive many different curve designs.



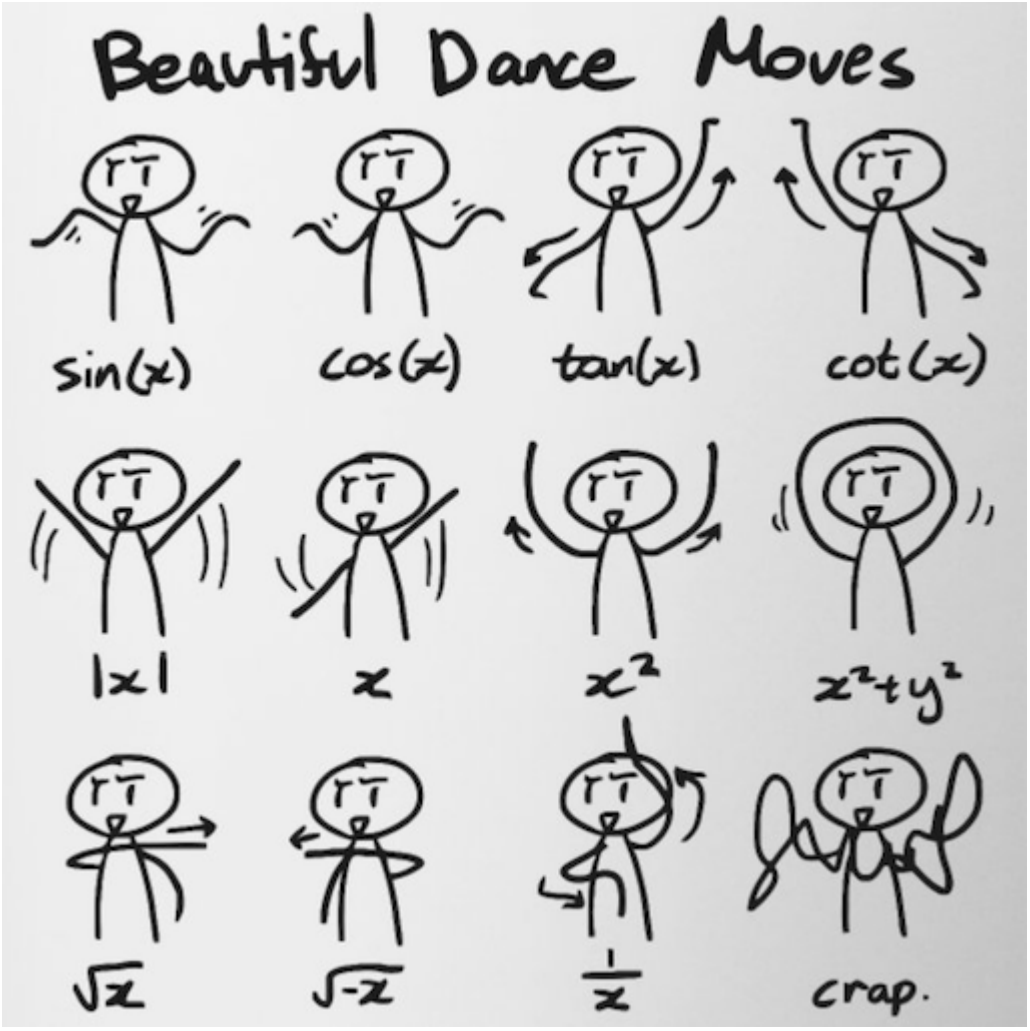
[math.stackexchange]

For now, we are having a look into somewhat simpler equations.

The underlying idea is to modify and shape different functions and by combining these we can almost draw anything. Such individually shaped functions can be used for a variety of applications, such as textures, shading, animation, geometry, dance, etc.



[wiki]



[webcomicms]

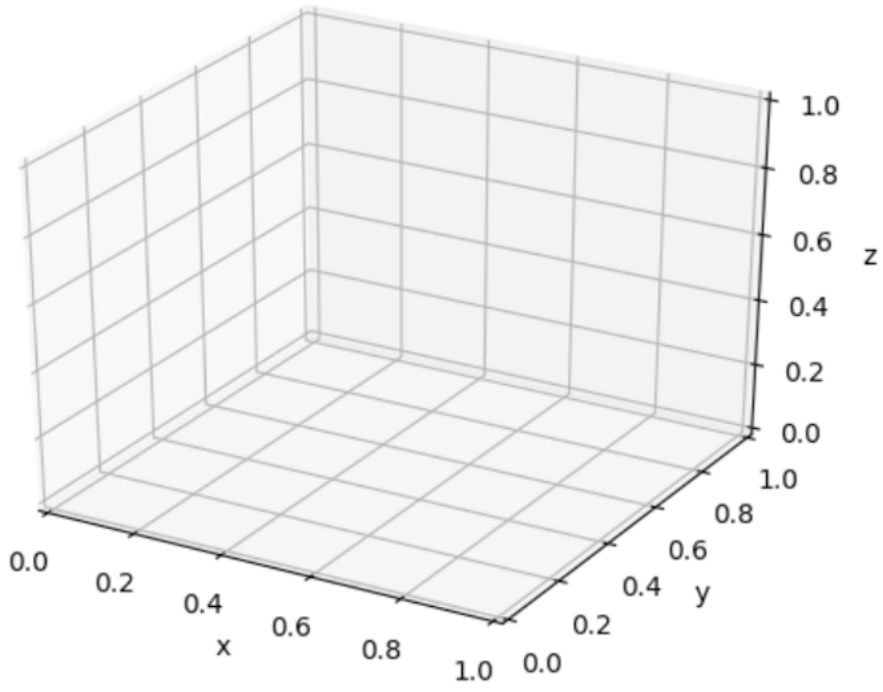
We will focus in this script on the generation of 2D graphics but please keep in mind that most functions are equally useful in different contexts and even dimensions.

2D Design

Let's assume we want to color a canvas, meaning giving each pixel a color value, e.g. when computing a texture.

To color this area, we can use a 2D function, meaning a function depending on two input parameters such as *x, y* in 3D space. Please note that in the following examples *z* is the up-axis of the coordinate system.

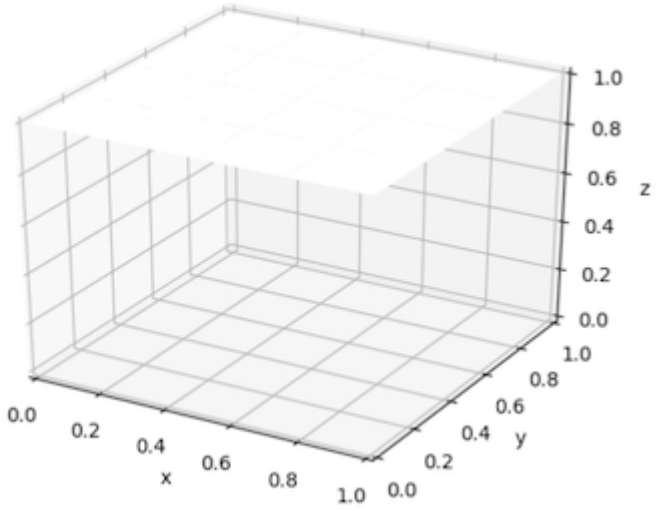
$$f(x,y) = z$$



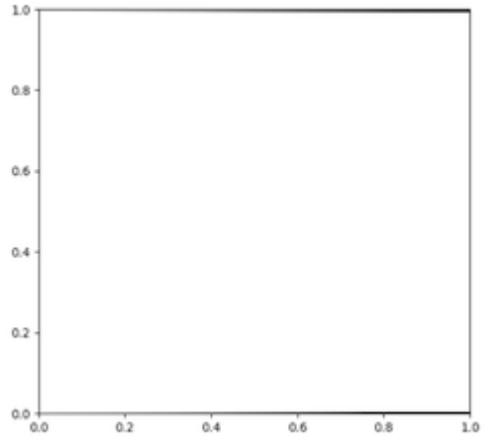
Then we interpret the value of *f(x,y) = z*, hence *z*, as color value.

For example,

$$f(x,y) = 1$$



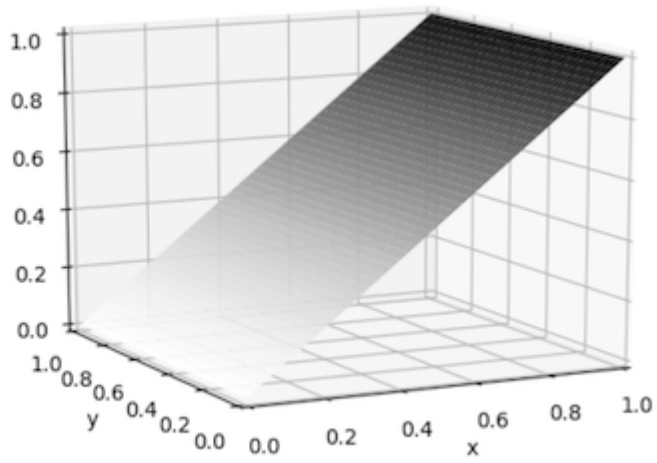
with *z* interpreted as color value between 0..1 on a 2D canvas:

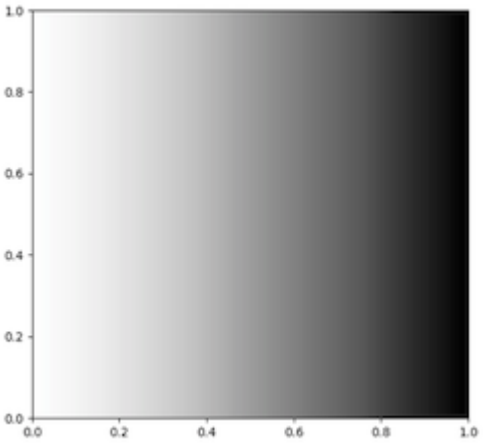


.

Or

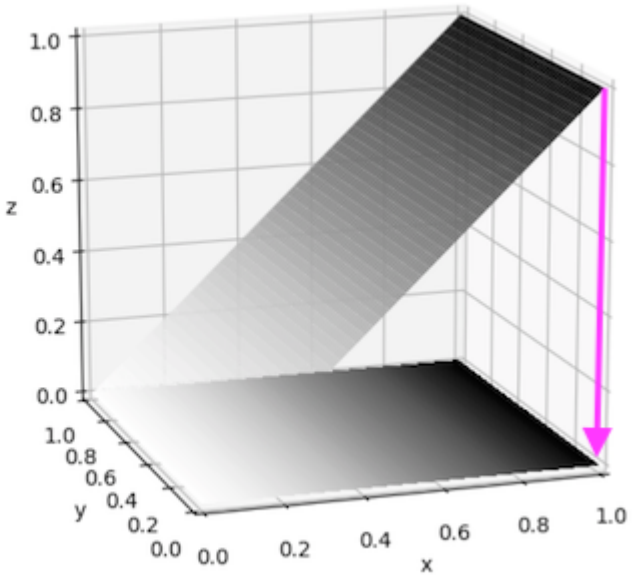
$$f(x,y) = x$$



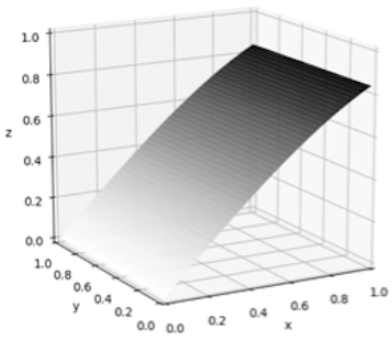


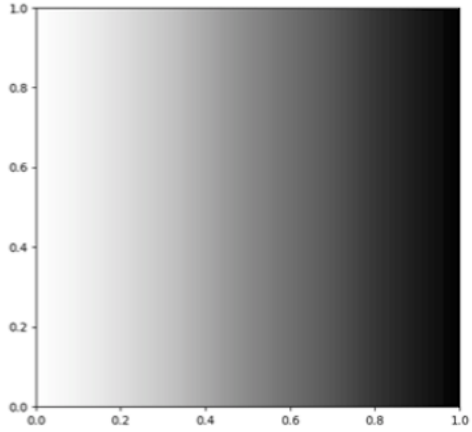
You can imagine this as if looking in -z direction onto the plotted gray value, or as if projecting the plot onto the xy-plane.

$f(x,y) = x$



$f(x,y) = sin(x)$

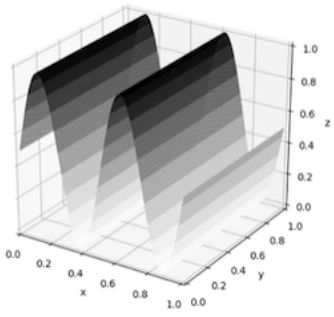


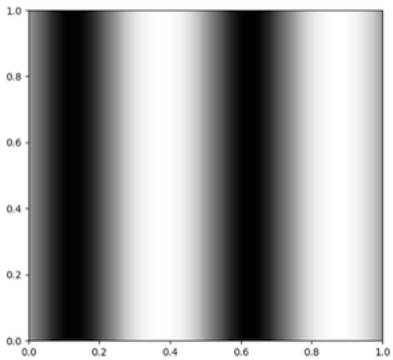


amplitude

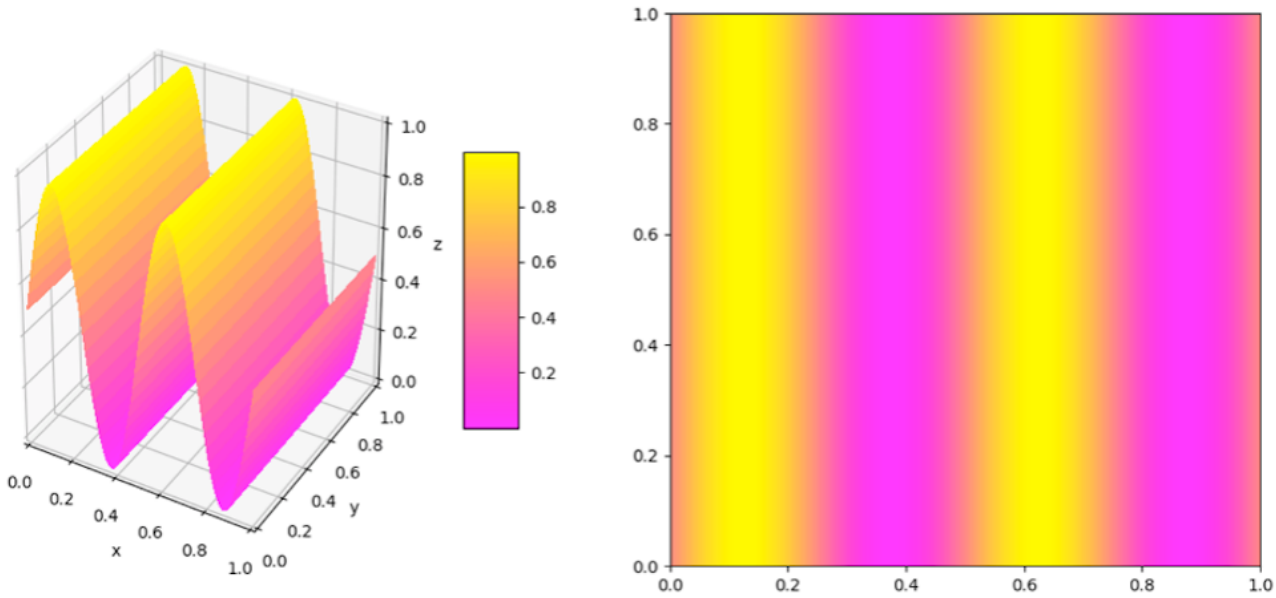
$f(x,y) = sin(4\pi \times x) \times 0.5 + 0.5$

frequency offset

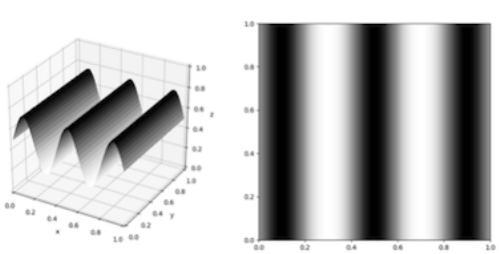




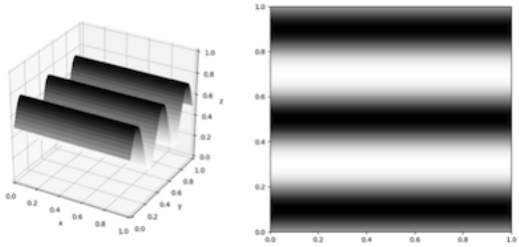
You could also map the function value to a color range such as



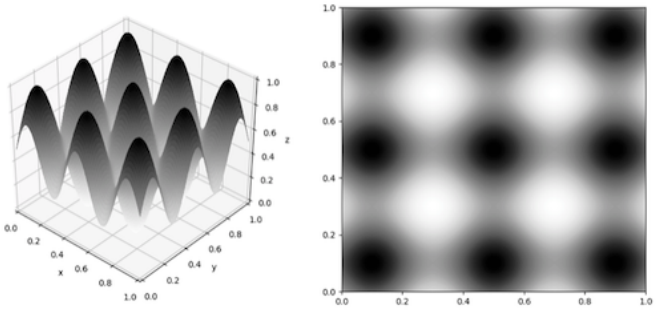
Also, the combination of simple functions can already lead to pleasing patterns



$f(x,y) = sin(x)$



$f(x,y) = sin(y)$



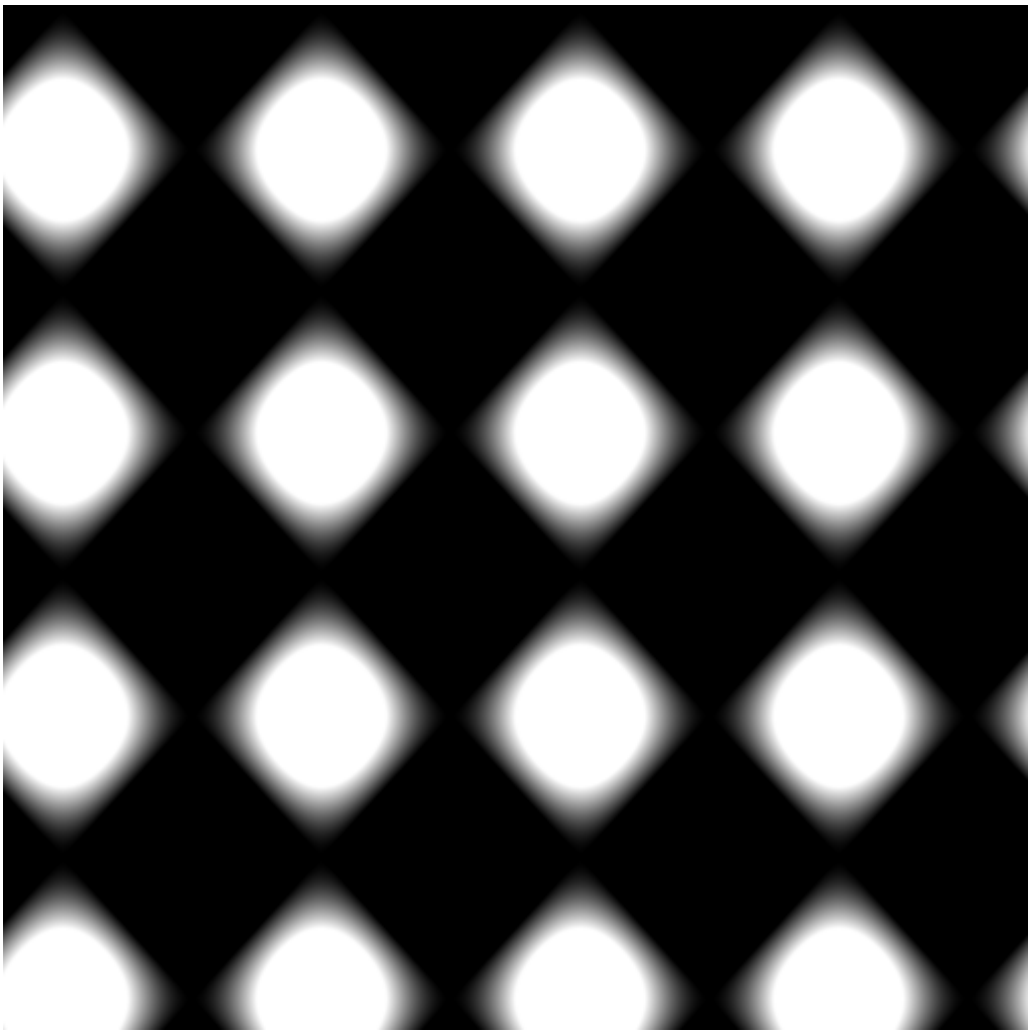
$f(x,y) = sin(y) + \sin(y)$

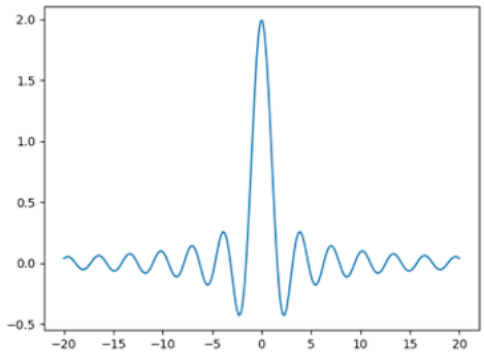
In the above plots frequency, amplitude and offset have been adjusted but left out in the equation for simplicity.

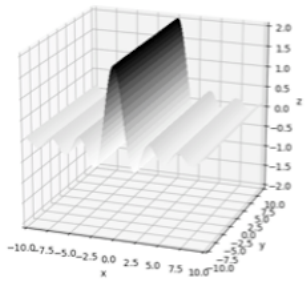
```
#ifdef GL_ES
precision mediump float;
#endif

#define PI 3.14159265359
```

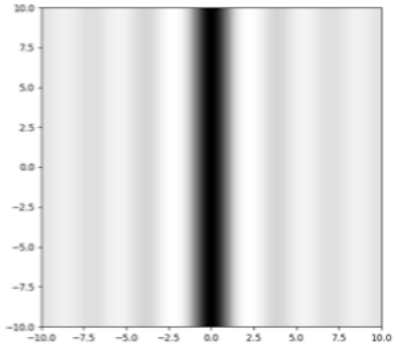
```
uniform vec2 u_resolution;

void main() {
    vec2 pt = gl_FragCoord.xy/u_resolution;

    // Increase frequency to fit more sin waves
    // between 0..1
    float verti = sin(8.0 * PI * pt.x);
    float hori = sin(8.0 * PI * pt.y);

    // Create vec3 from value
    vec3 color = vec3(hori + verti);

    // Assign frag color with alpha
    gl_FragColor = vec4(color,1.0);
}
```

Code rendering:









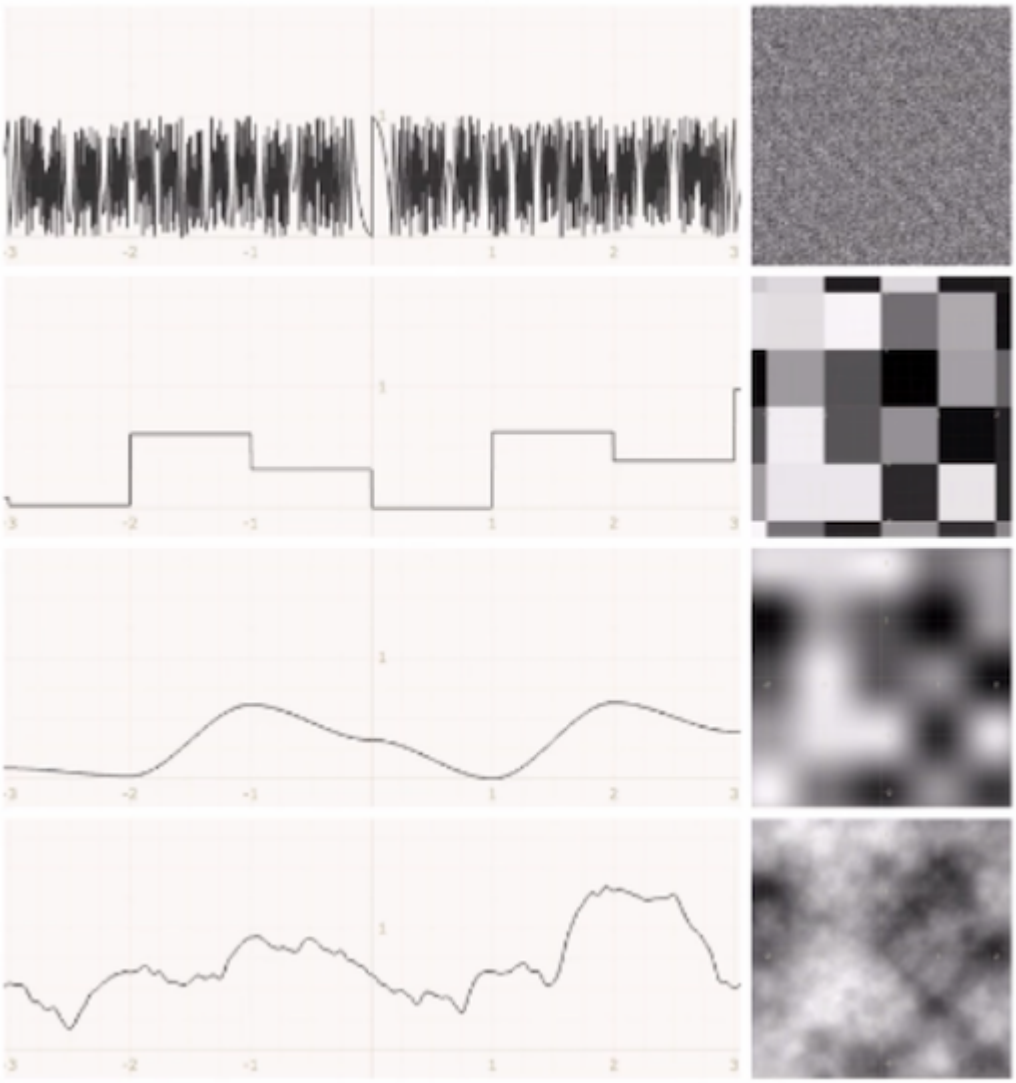
$$f(x,y) = sin(2x) \ / \ x$$

Different Dimensions

You will often find examples and explanations in lower dimensions, meaning in 1D, as the graphs for these functions are easier to visualize:

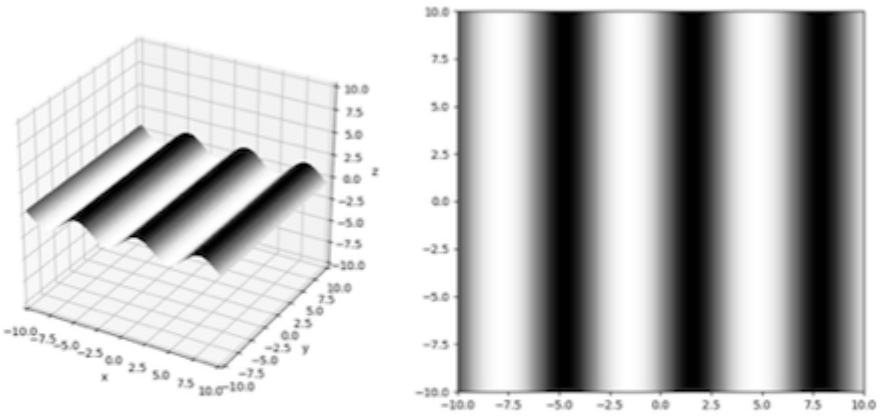
f(x) = c

with *c* as the gray or color value.



[tobyschachman]

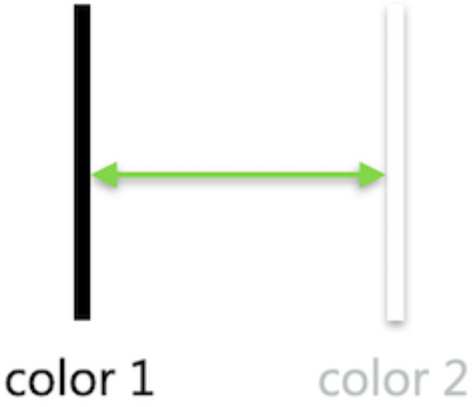
How to interpret in this scenarios the needed second dimension depends on the context. Often it is simply left out as an influencing parameter such as in our start example



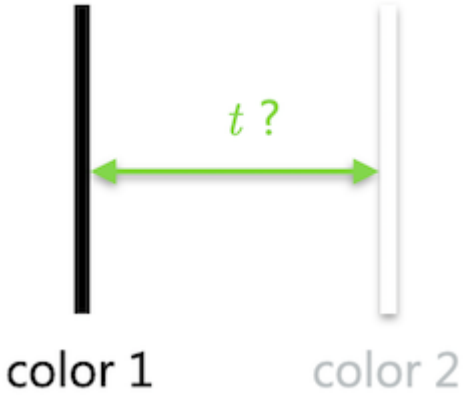
$$f(x,y) = sin(x)$$

Transitions

The above examples have already shown one overall principle, namely *how to get from one value to another value* or in our context *how to get from one color (e.g. white) to another color (e.g. black)*?

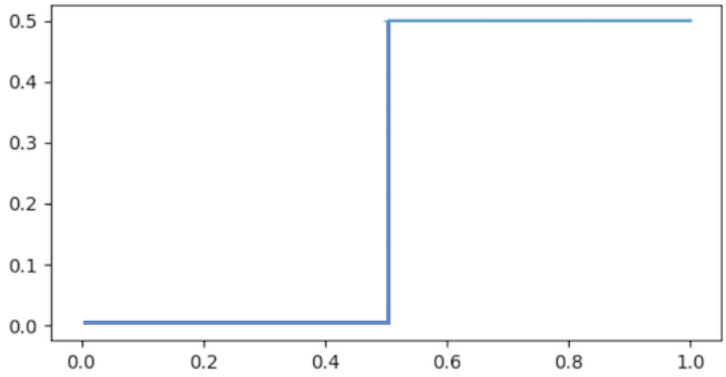


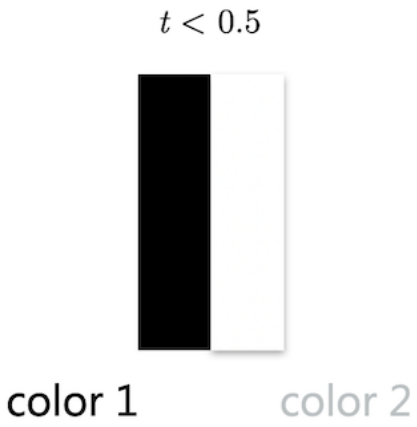
In more general terms we can understand this as defining a transition function *t*:



Step Function

The simplest of transitions is the step function, which switches between values based on a threshold, meaning with a fixed value that *t* is smaller or larger to.

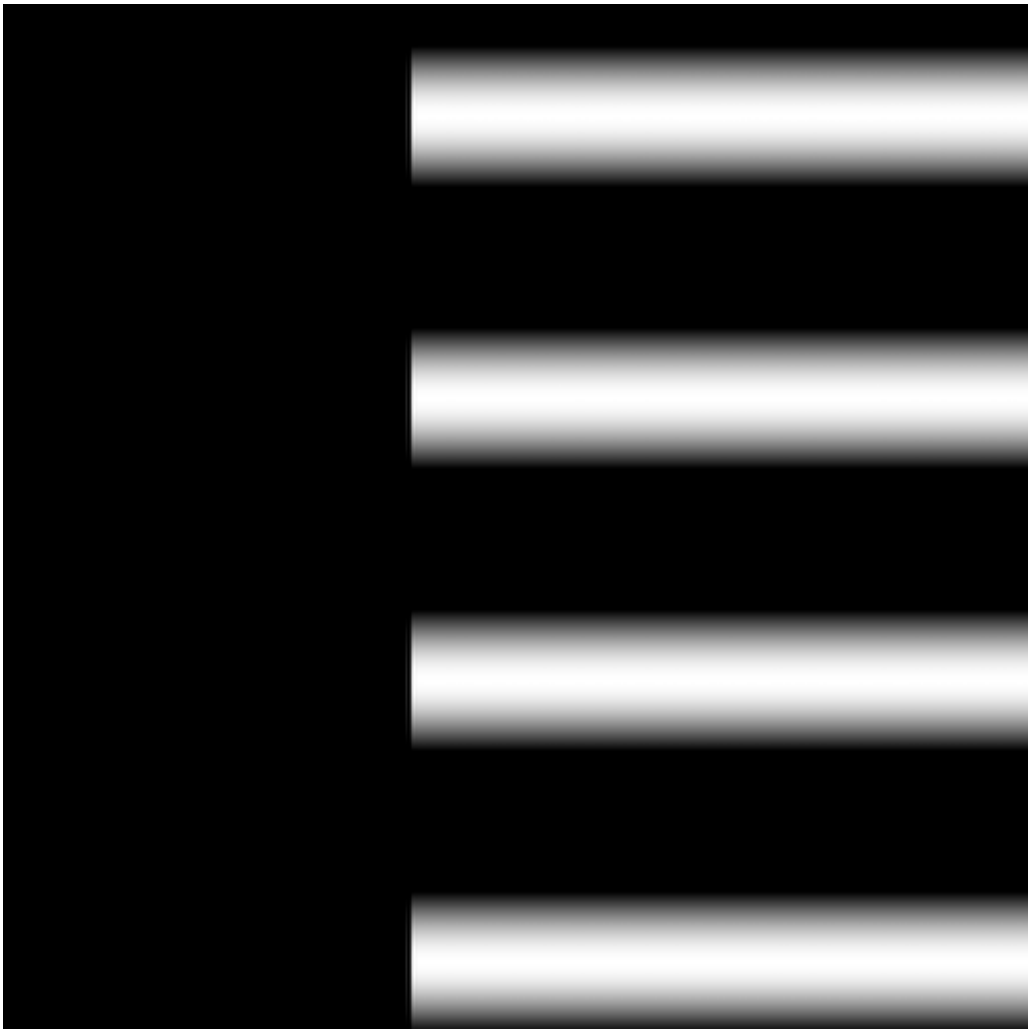




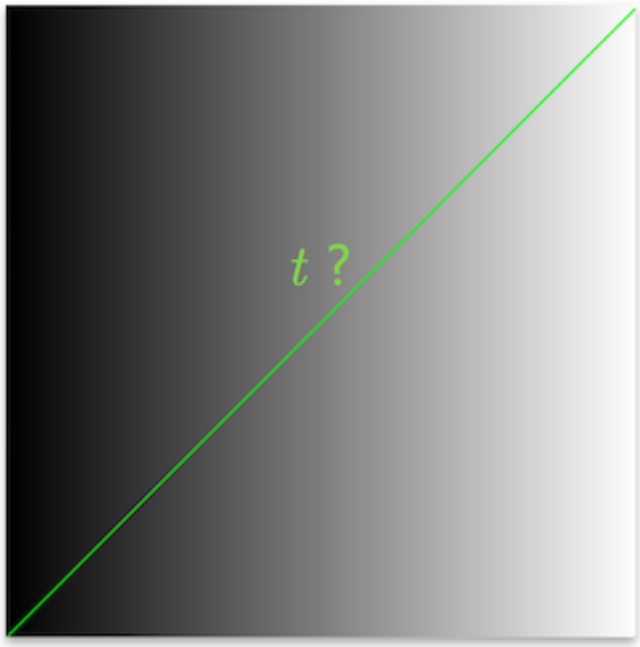

```
#ifdef GL_ES
precision mediump float;
#endif
```

```
#define PI 3.14159265359
uniform vec2 u_resolution;

void main()
{
    vec2 pt = gl_FragCoord.xy/u_resolution;

    // The following step function returns
    //  0.0 when x is smaller than 0.4 and
    //  1.0 otherwise
    float step_value = step(0.4, pt.x);

    // Our "pattern":
    float hori = sin(8.0 * PI * pt.y);

    // Multiplying 0 or 1 with the pattern
    // depending on the step function value
    vec3 color = vec3(step_value * hori);

    // Assign frag color with alpha
    gl_FragColor = vec4(color,1.0);
}
```

Code rendering:



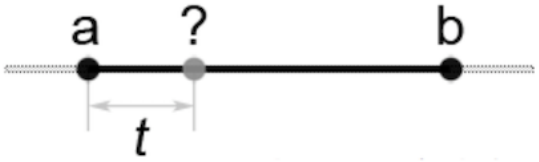
More often we want a smoother transition, e.g.



color 1 color 2

This is called an *interpolation*.

Linear Interpolation



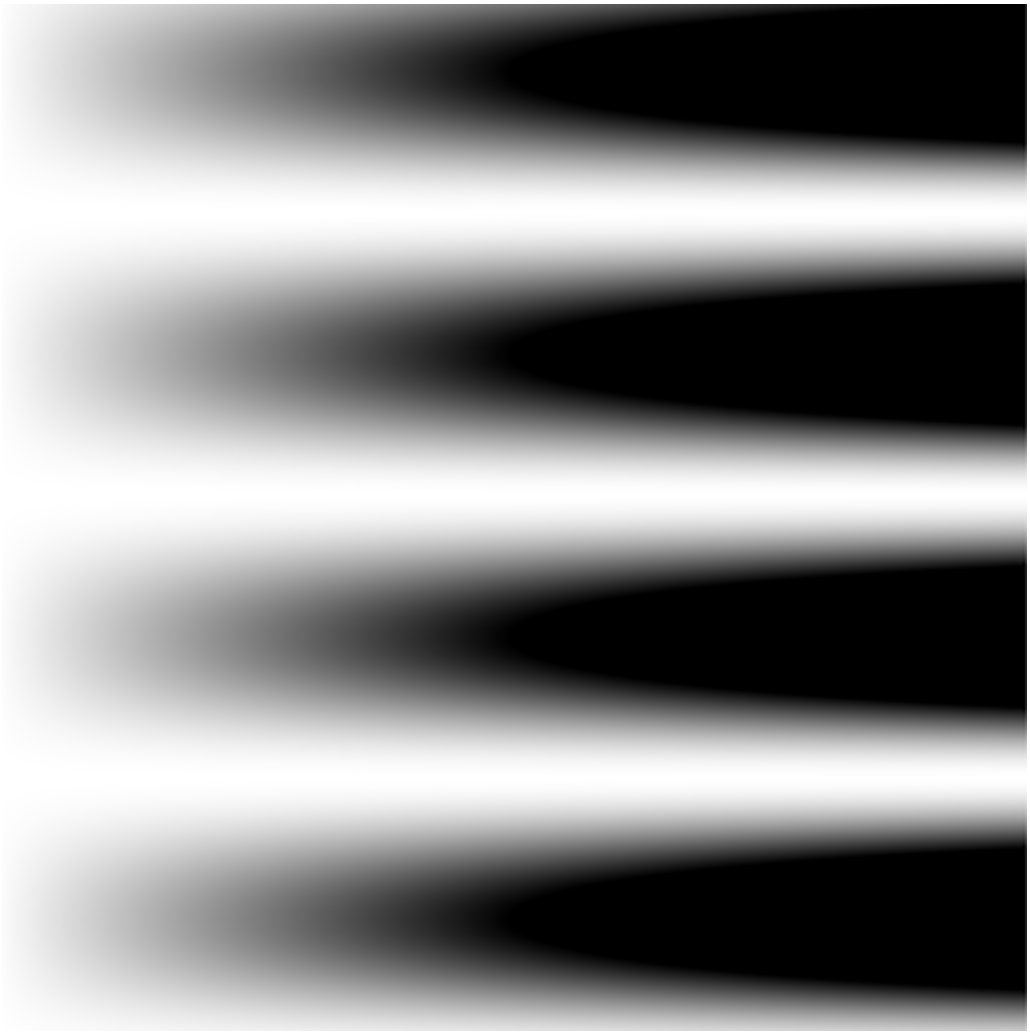
$$a(1-t)+bt \quad with \quad 0 \le t \le 1$$

[1]

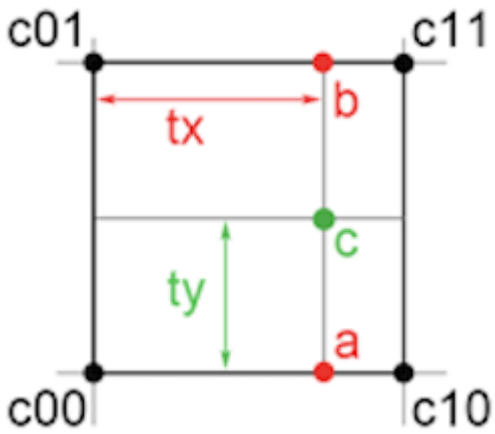
```
#ifdef GL_ES
precision mediump float;
#endif

#define PI 3.14159265359
uniform vec2 u_resolution;

void main()
{
    vec2 pt = gl_FragCoord.xy/u_resolution;

    // Our "pattern":
    float hori = sin(8.0 * PI * pt.y);
```

```
    // Interpolating between the pattern and 1.
    // depending on the x coordinate, meaning
    // with t = pt.x
    vec3 color = vec3(pt.x * hori + (1.0 - pt.x));

    // Assign frag color with alpha
    gl_FragColor = vec4(color,1.0);
}
```

Code rendering:



Bilinear Interpolation



[scratchapixel]

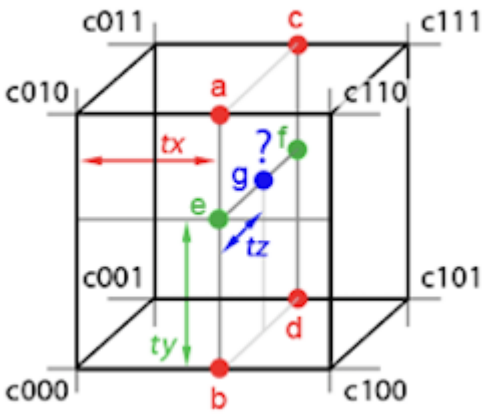
A bilinear interpolation is the linear interpolation of two linear interpolations, hence

- two linear interpolations to get `a` and `b` in one direction (here `tx`)
- one linear interpolation of `a` to `b` in the second direction (here `ty`)

```
a = c00 * (1 - tx) + c10 * tx;
b = c01 * (1 - tx) + c11 * tx;

c = a * (1) - ty) + b * ty;
```

Linear and bilinear interpolation is usually called `lerp()`, e.g. `lerp` in p5 or `lerp` in vex or `mix` in glsl..

Trilinear Interpolation



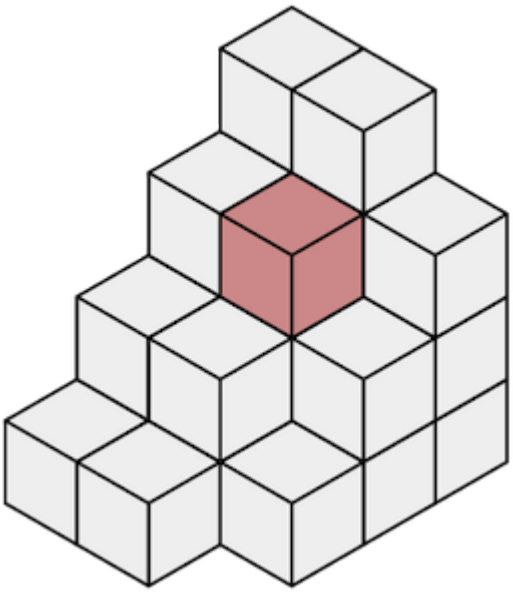
[scratchapixel]

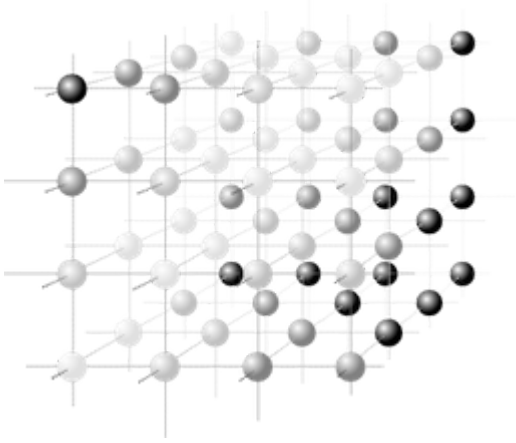
A trilinear interpolation is the linear interpolation of two bilinear interpolations.

```
e = bilinear(tx, ty, c000, c100, c010, c110);
f = bilinear(tx, ty, c001, c101, c011, c111);

g = e * ( 1 - tz) + f * tz;
```

By the way, *what is a voxel*?

A voxel is like a pixel but in 3D and represents a value on a regular 3D grid. We need voxels for volumes, for example.

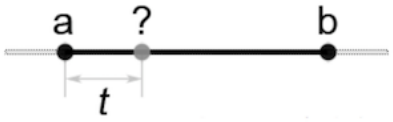




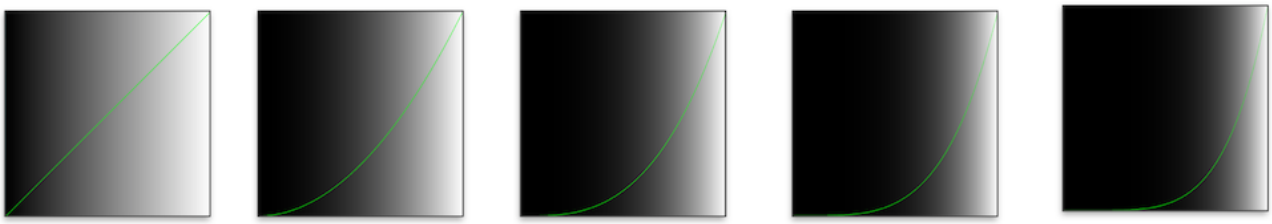
[wiki]

Interpolation Functions

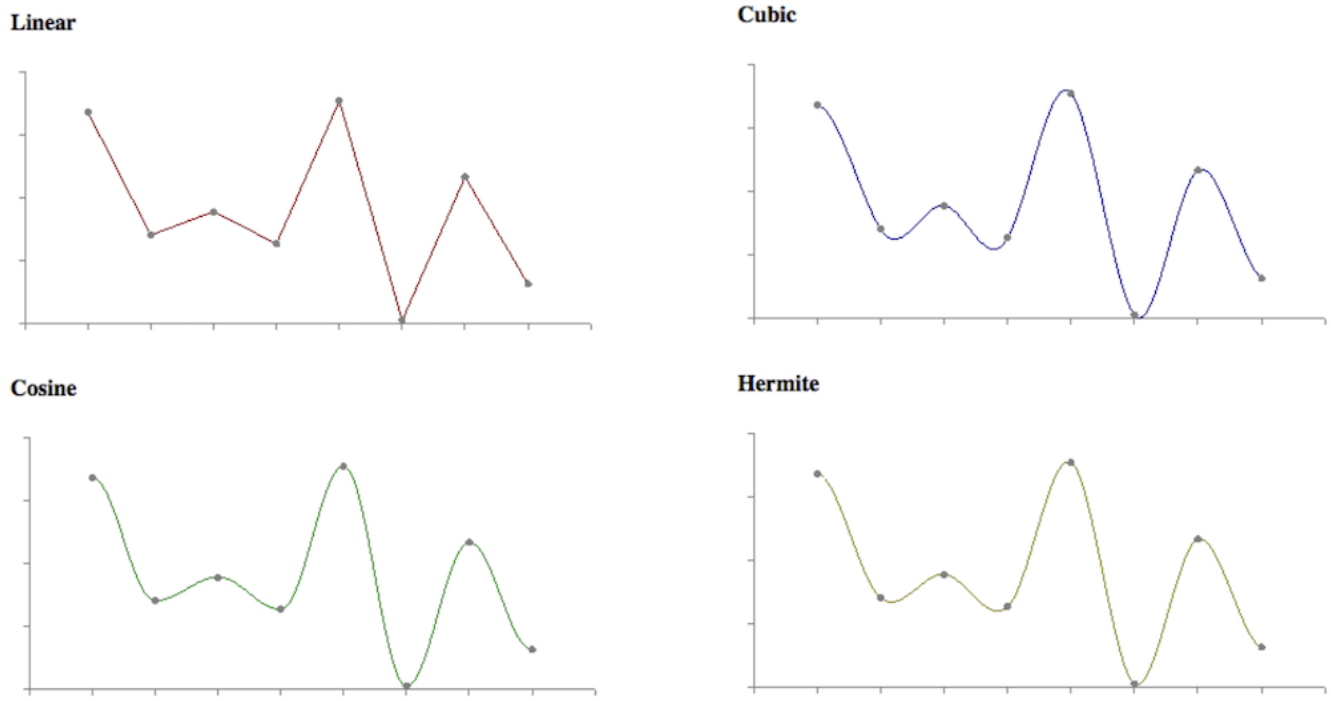
To move between values, we have many options. Simply taking different exponents for *t* in a linear interpolations changes the transition between the values notably.



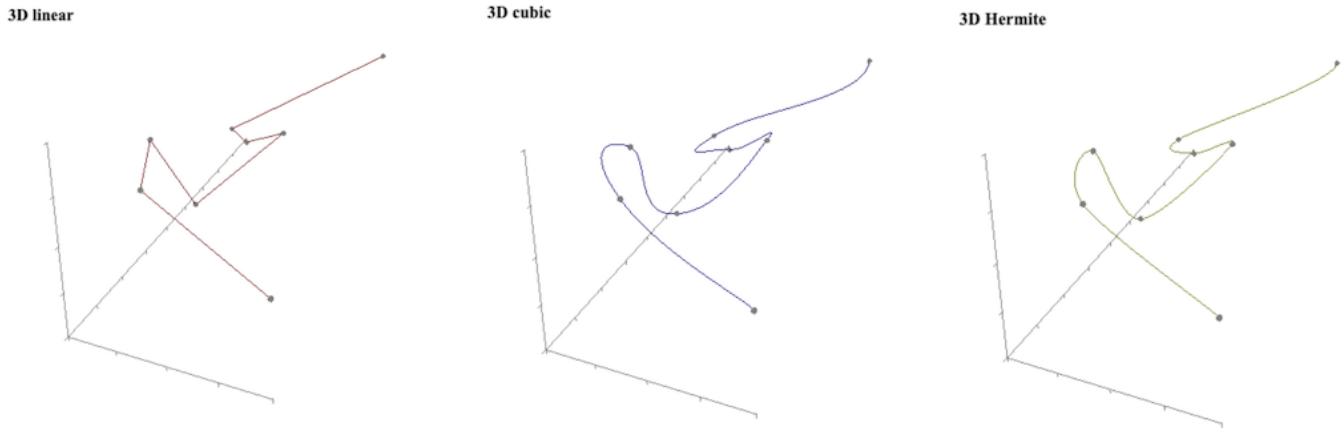
$$a(1-t)+bt \quad with \quad 0 \le t \le 1$$



For example, in 3D software such as Houdini, there are several interpolations functions to chose from. Here, some comparisons:

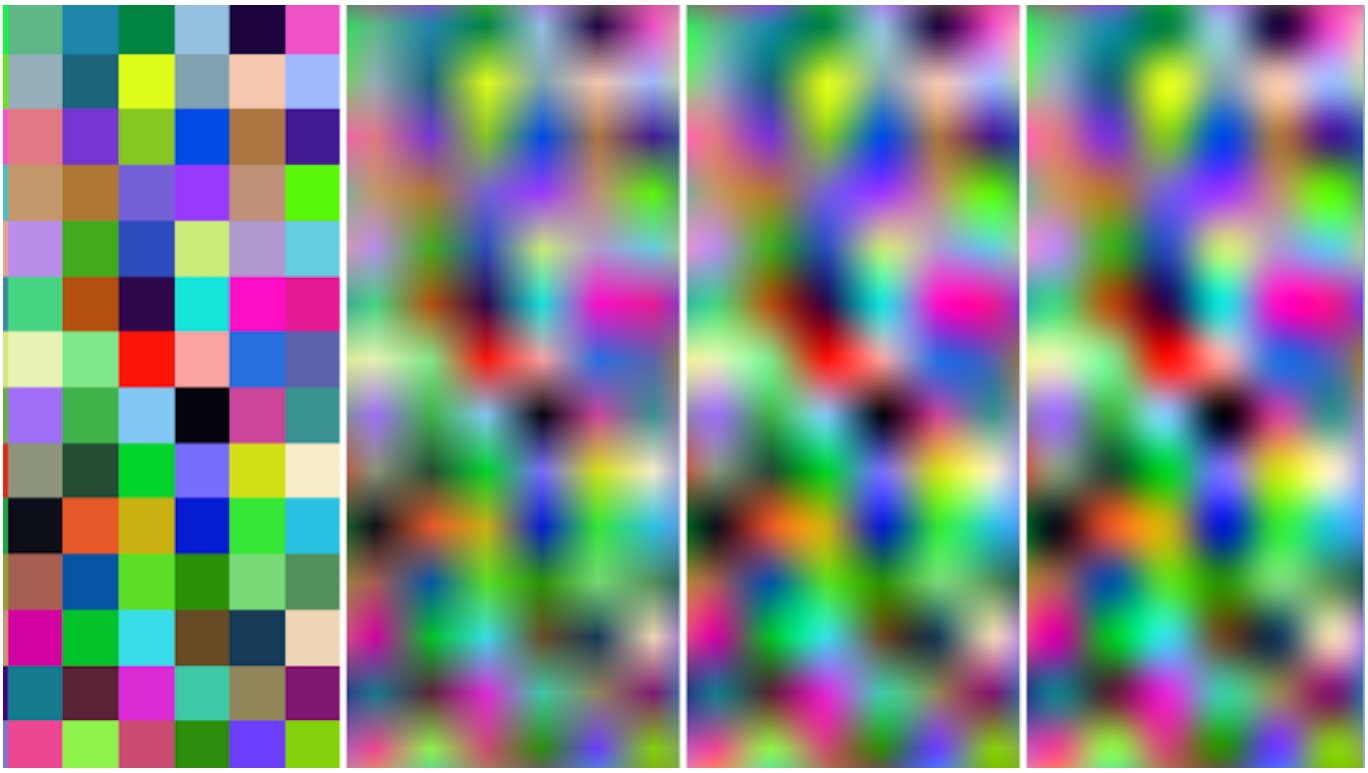


[paulbourke]

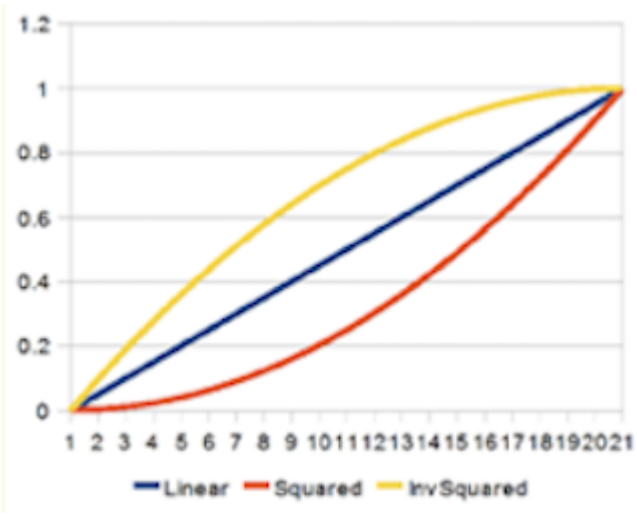


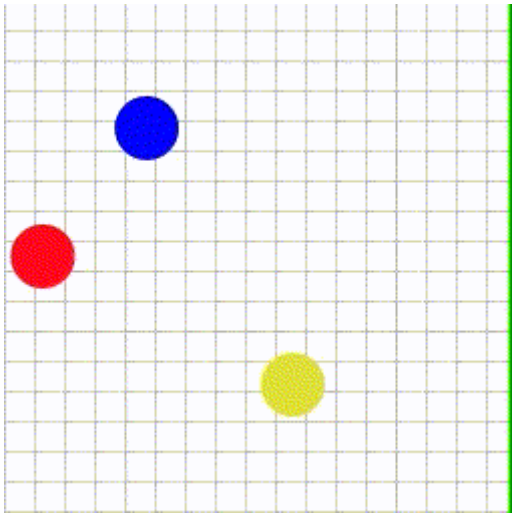
[paulbourke]

These different functions lead to different visual designs, depending on the context, e.g. for interpolating between colors for an image or positions for an animation. From left to right, Nearest Neighbor, Bilinear, Lagrange Bicubic interpolation (only interpolates values, not slopes), Hermite Bicubic interpolation:



[demofox]





[gfxile]

Smooth Step

Smoothstep is one of the most commonly used interpolation and clamping function in graphics and is often given as a build-in function from a framework.

On a side note: What is a *clamping* function? In computer graphics, clamping is the process of limiting a value to a range. Unlike wrapping, clamping merely moves the point to the nearest available value. [2]

$$\mathrm{S}_0(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } 0 \leq x \leq 1 \\ 1 & \text{if } 1 \leq x \end{cases}$$

S is a sigmoid function, which is bounded and often used in the context of mapping a potentially indefinite range of values to a range.

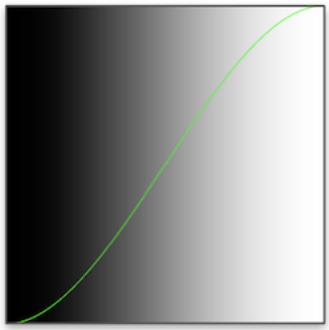
Often smoothstep implements a cubic Hermite interpolation after clamping:

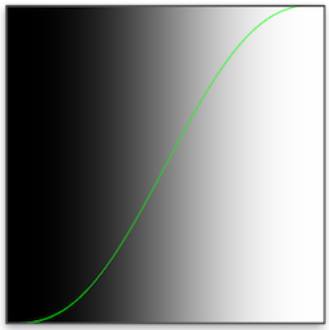
$$\text{smoothstep}(x) = S_1(x) = \begin{cases} 0 & x \leq 0 \\ 3x^2 - 2x^3 & 0 \leq x \leq 1 \\ 1 & 1 \leq x \end{cases}$$

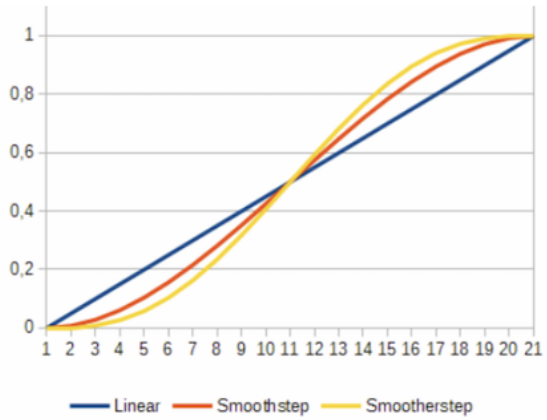
```
float smoothstep(float edge0, float edge1, float x)
{
    x = clamp((x, 0.0, 1.0);
    return x * x * (3 - 2 * x);
}

float clamp(float x, float lowerlimit, float upperlimit)
{
    if (x < lowerlimit)
        x = lowerlimit;
    if (x > upperlimit)
        x = upperlimit;
    return x;
}
```

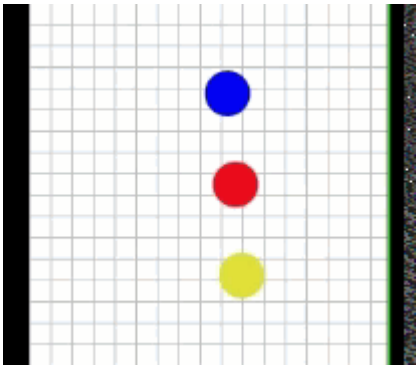
Smoother Steps is an improved version of the smoothstep function, created by Ken Perlin.

$$\text{smootherstep}(x) = S_2(x) = \begin{cases} 0 & x \leq 0 \\ 6x^5 - 15x^4 + 10x^3 & 0 \leq x \leq 1 \\ 1 & 1 \leq x \end{cases}$$









[gfxile]

[4]

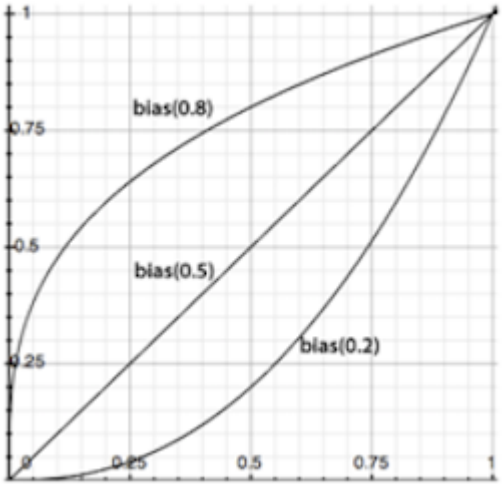
[3]

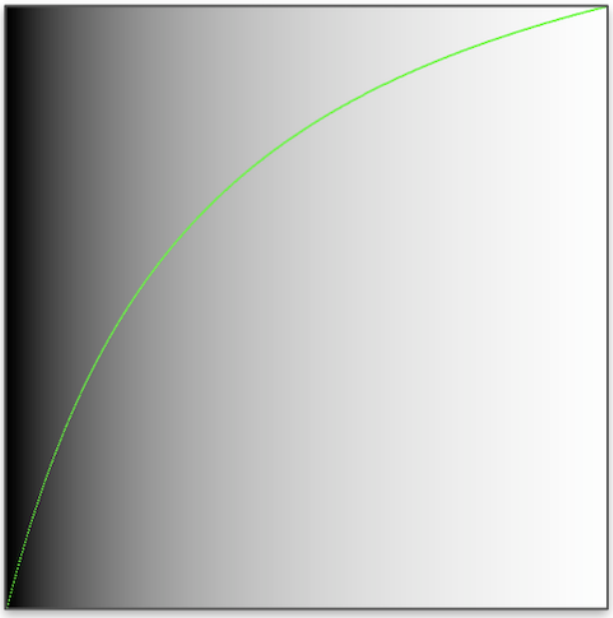
Bias and Gain

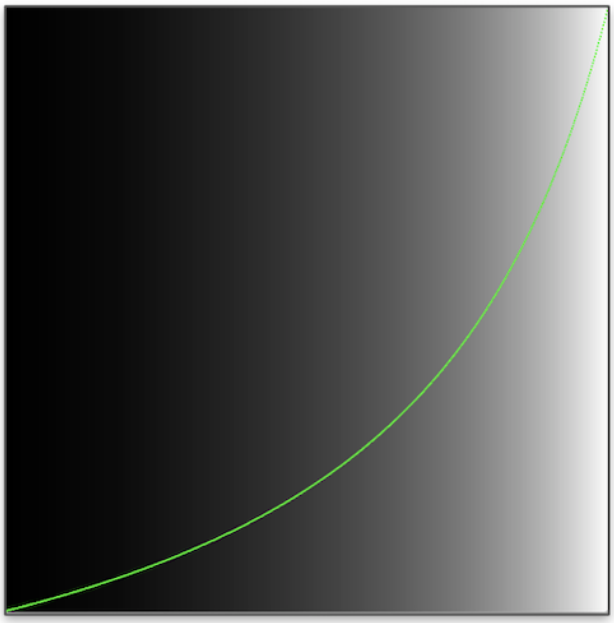
Bias and gain are parameters that give further control for the fine-tuning of a interpolation function curve. This has been, once again, Ken Perlin's idea.

Bias

Bias represents *how much time is spent at either end of the transition*? The larger the values, the faster grows the value at the beginning.



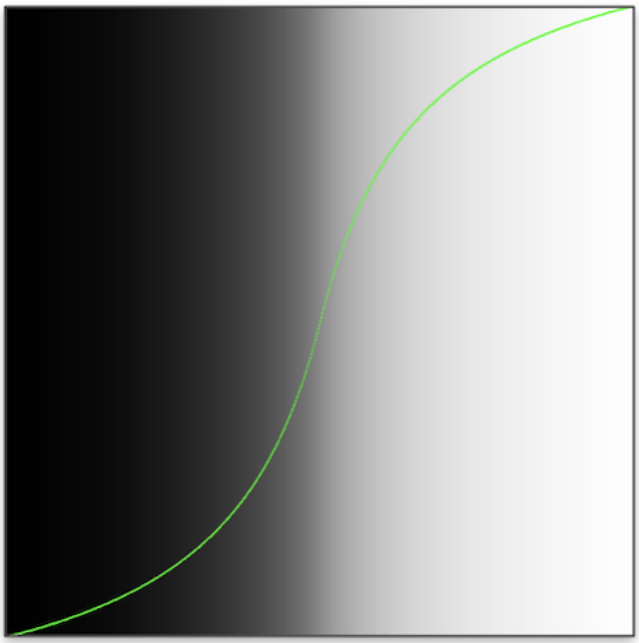




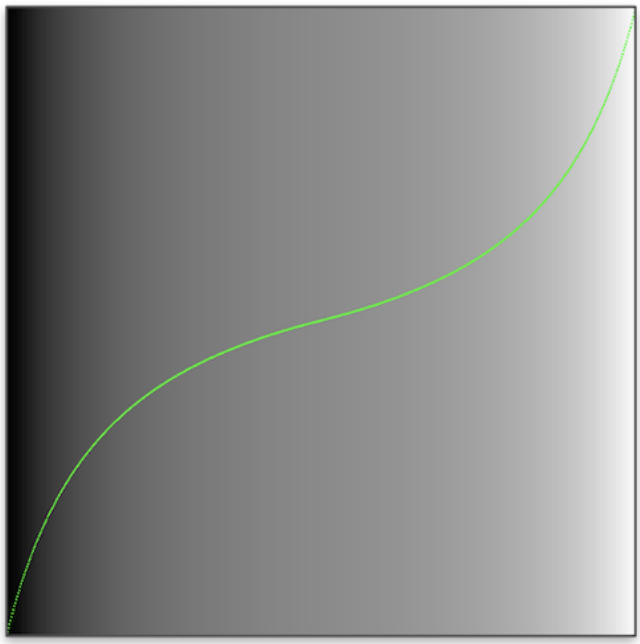
```
float get_bias(float t, float bias)
{
    return (t / (((1.0 / bias) - 2.0) * (1.0 - t)) + 1.0);
}
```

Gain

Gain represents *how much time is spent in the middle of the transition*? The larger the value, the slower changes the value around the middle.

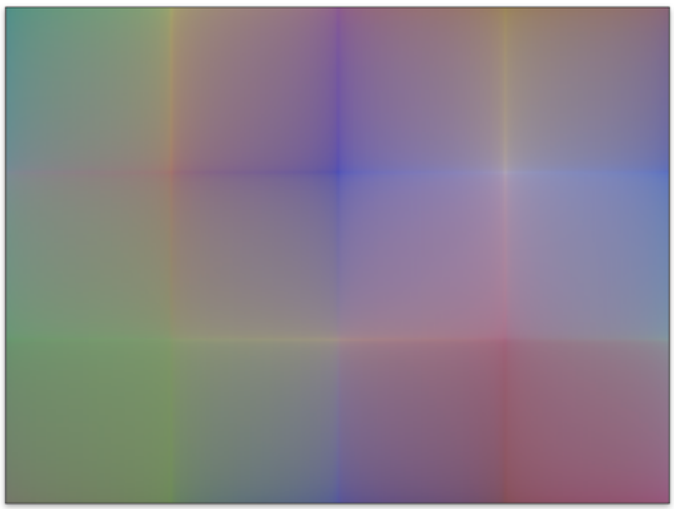


gain = 0.2

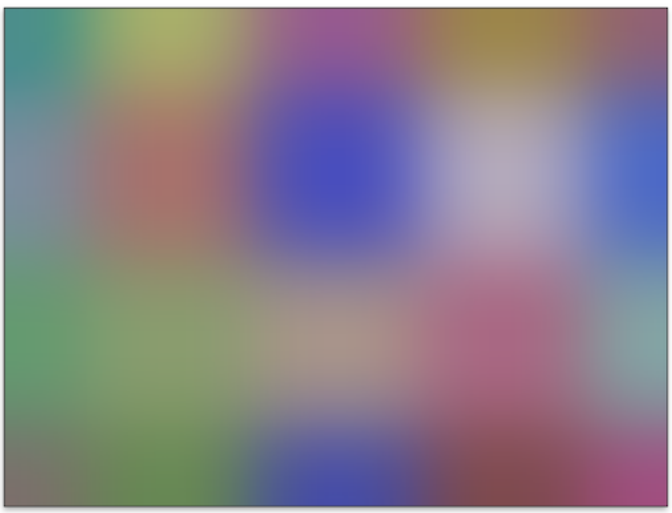


gain = 0.8

```
float get_gain(float t, float gain)
{
    if(t < 0.5)
        return get_bias(t * 2.0, gain) / 2.0;
    else
        return get_bias(t * 2.0 - 1.0, 1.0 - gain) / 2.0 + 0.5;
}
```



gain 0.25



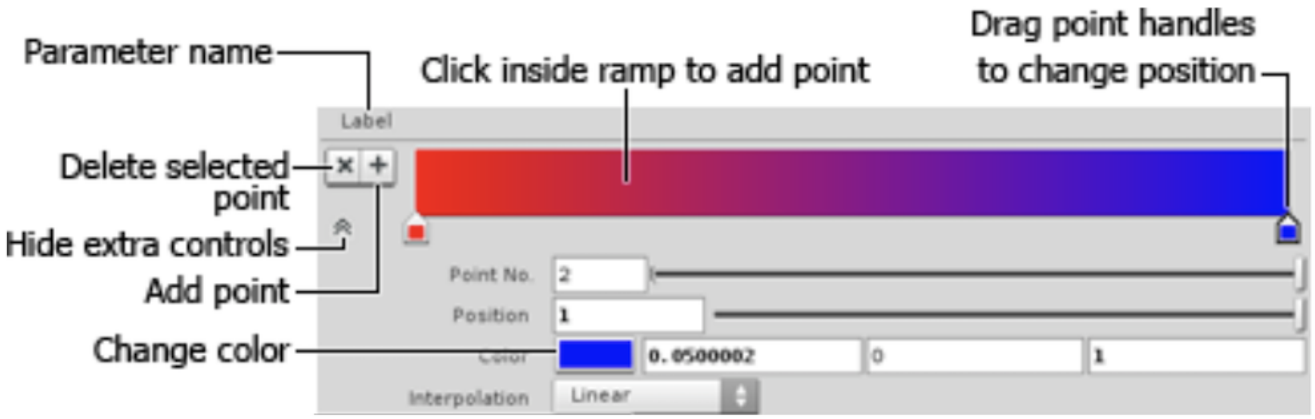
gain 0.75

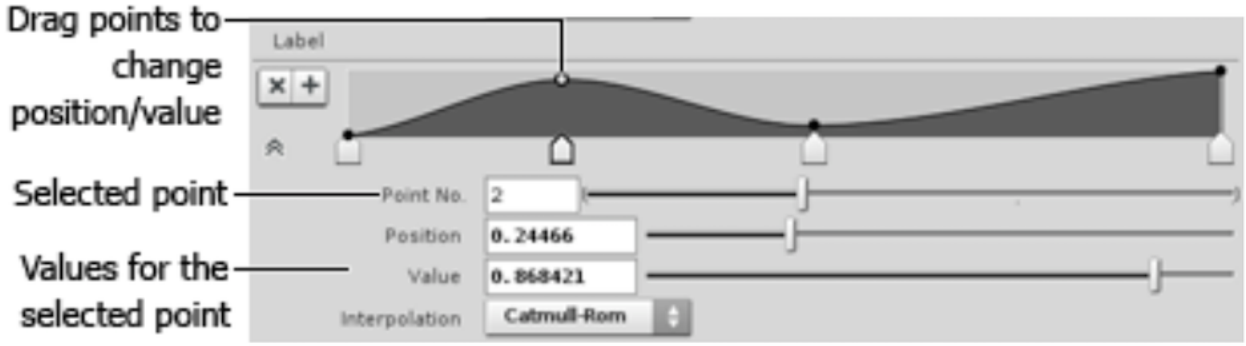
[4]

For further information, read Perlin's blog post Bias And Gain Are Your Friend.

Interpolation in Houdini

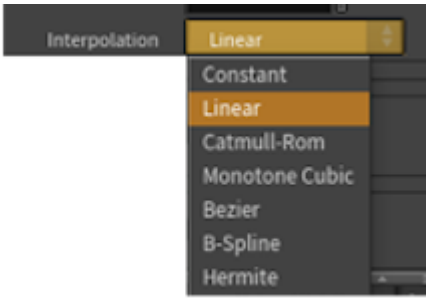
In Houdini interpolation is relevant in numerous places, e.g. for the ramp parameter control.





[houdini]

With the interpolation functions



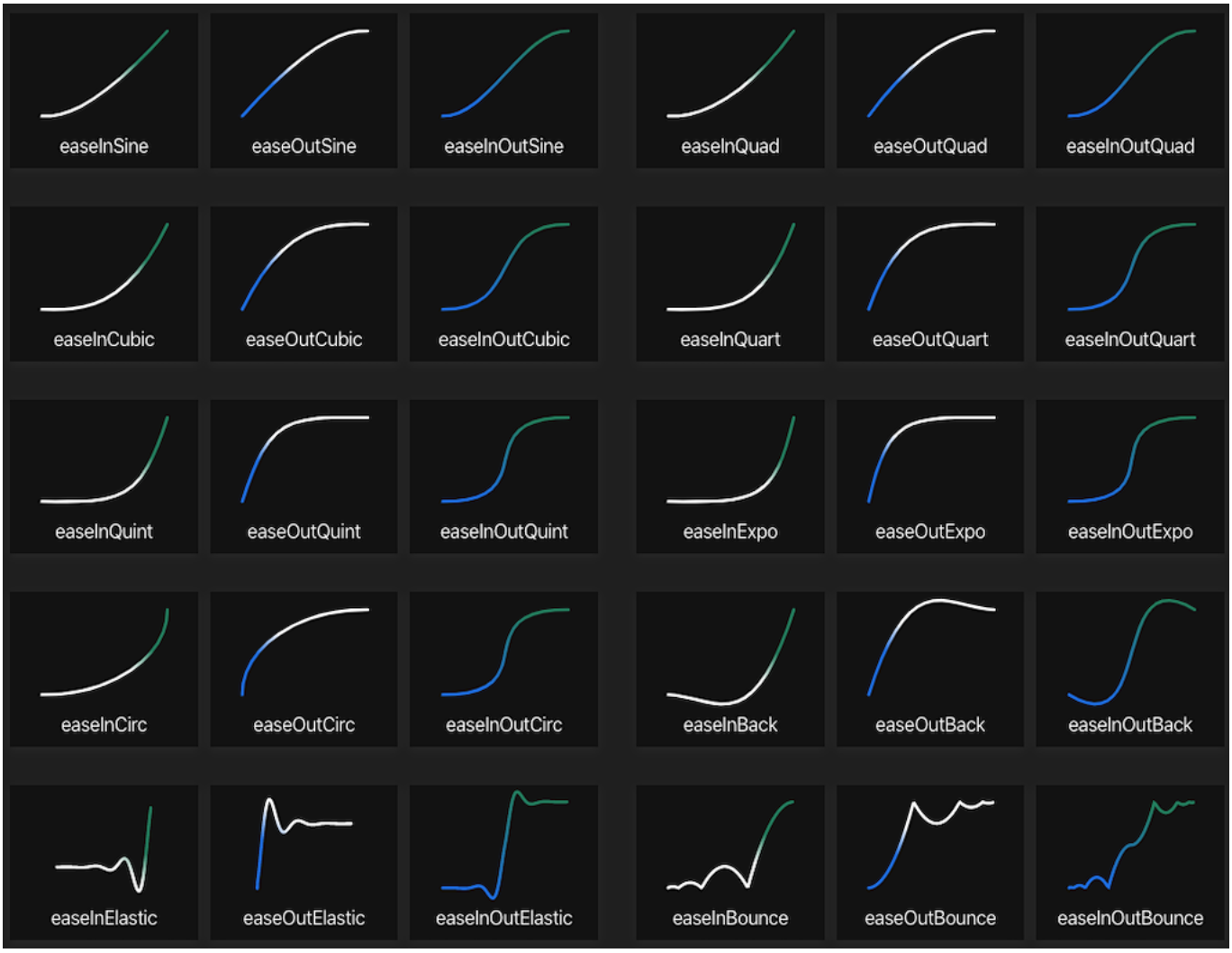
- Constant
  - Holds the value constant until the next key.
- Linear
  - Does a linear (straight line) interpolation between keys.
- Catmull-Rom
  - Interpolates smoothly between the keys. See Catmull-Rom_spline.
- Monotone Cubic
  - Another smooth interpolation that ensures that there is no overshoot. For example, if a key's value is smaller than the values in the adjacent keys, this type ensures that the interpolated value is never less than the key's value.
- Bezier

    - Cubic Bezier curve that interpolates every third control point and uses the other points to shape the curve. See Bezier curve.
- BSpline
    - Cubic curve where the control points influence the shape of the curve locally (that is, they influence only a section of the curve). See B-Spline.
- Hermite
    - Cubic Hermite curve that interpolates the odd control points, while even control points control the tangent at the previous interpolation point. See Hermite spline.

[5]

Interpolation in CSS

These discussed principles and properties of interpolation are the same as *easing functions* in CSS.



[easings]

Function Primitive Components

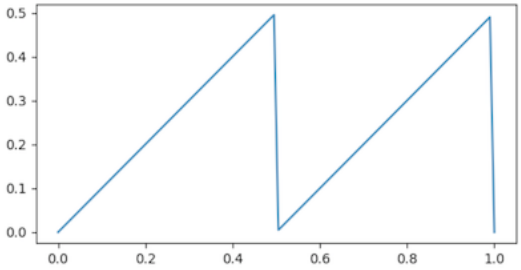
So far, we can only transition from one value or function exemplar to another one. That is a bit boring. The following presents a list of the most commonly used function components for putting together an individual design goal.

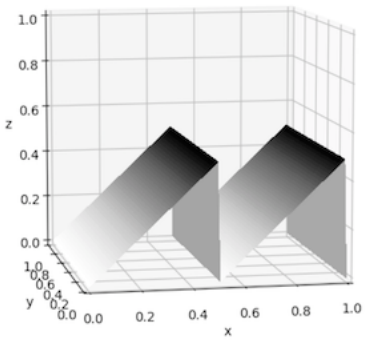
A great tool to work with function components and test how to put them together is the Graph Toy.

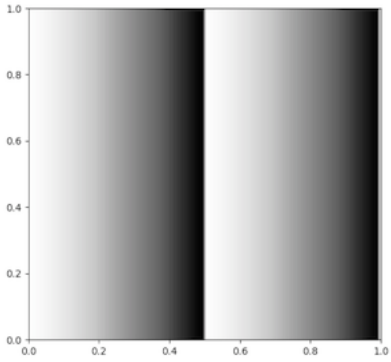
Modulo

With modulo you can easily iterate ranges and therefore loops, for example.

```
y = x % 0.5;
```



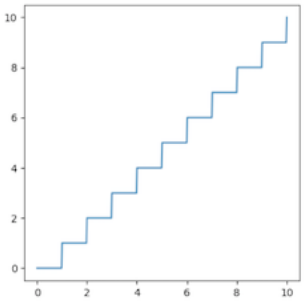


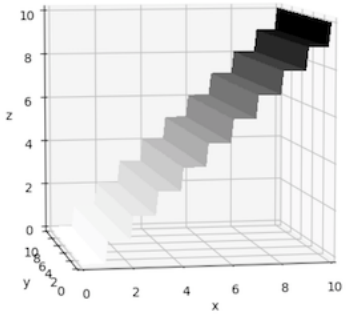


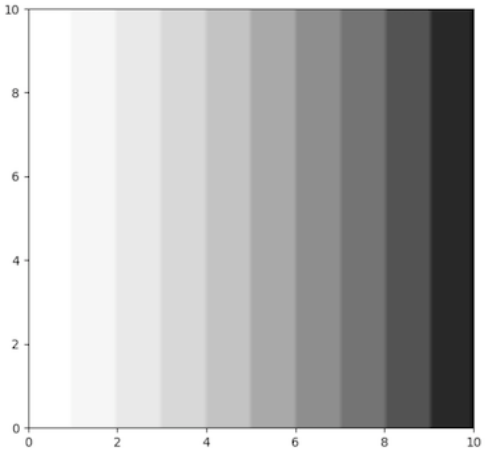
Floor

Floor ignores fraction and creates with that a continuous step function.

```
y = floor(x);
```



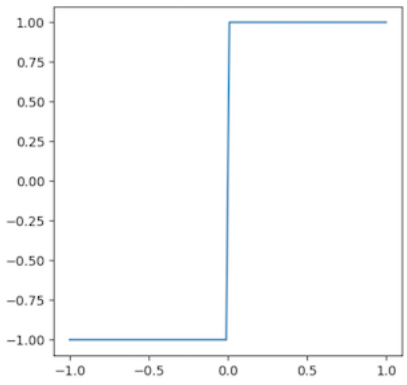
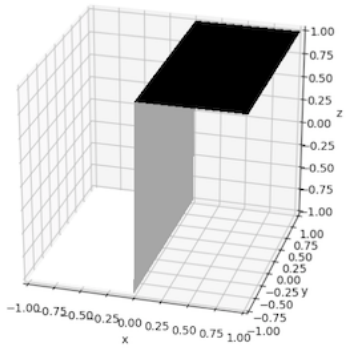
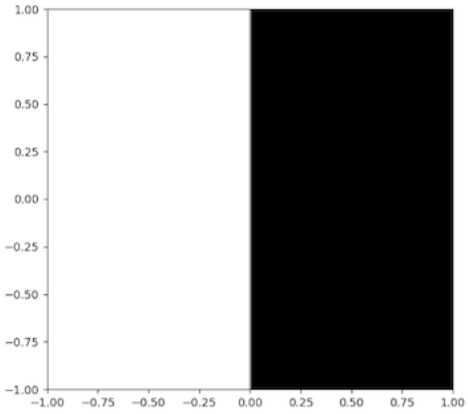




Sign

Sign extracts the sign of a real number and is therefore either 1 or −1.

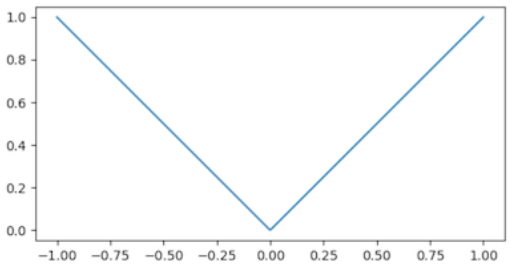
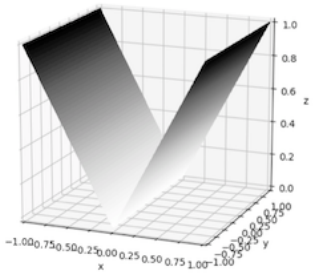
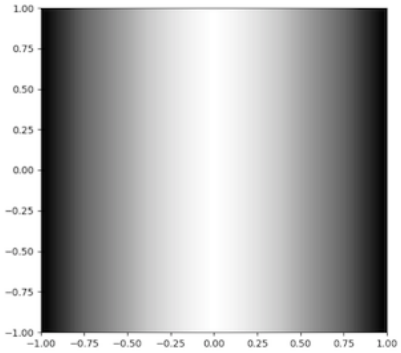
```
y = sign(x);

//-1 if x < 0, 0 if x==0, 1 if x > 0
```

Absolute

The absolute keeps values always positive.

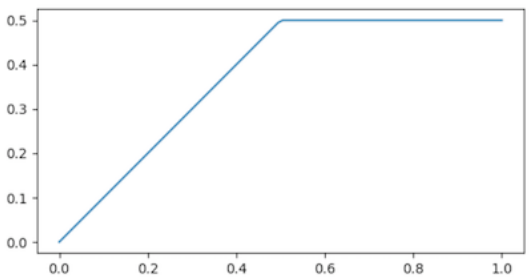
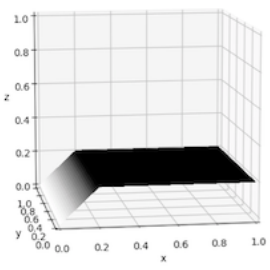
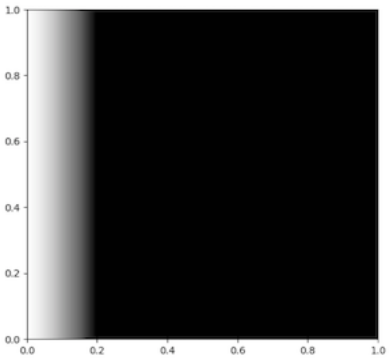
```
y = abs(x);
```

Min and Max

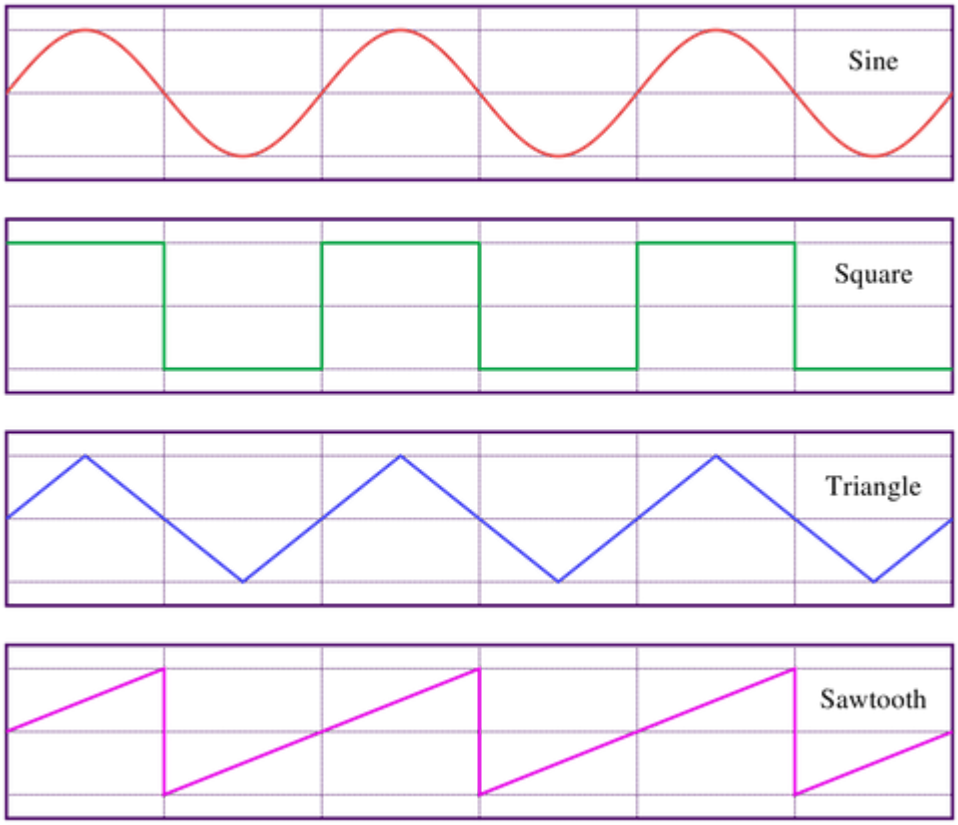
Min and Max are used to define lower and upper borders.

```
y = min(x, 0.5);

y = min(x, 0.2);
```

Periodicity

Often times we want to repeat certain visual features, which can be done in its simplest form e.g. with a `sin` function. However, there are several other design options. The following functions are also often called *wave functions*.



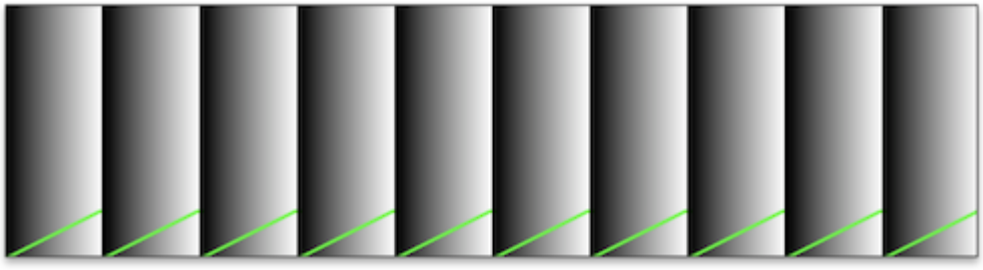
[wiki]

Wave functions have two common properties

- frequency ("*how often*"), and
- amplitude ("*how much*").

used as

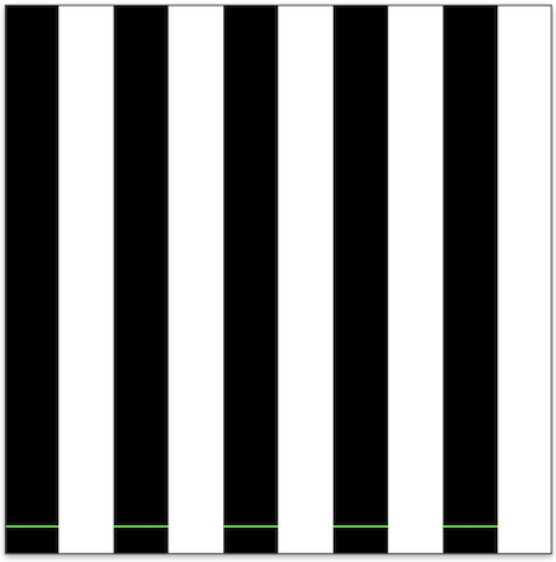
```
(t * frequency) % amplitude;
```



Square

The square wave enables a sharp oscillation between two values.

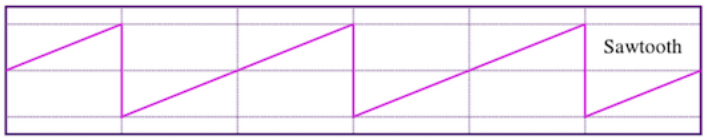
```
float wave_square(float t, float frequency, float amplitude)
{
  return floor(t* frequency) % 2 * amplitude;
}
```

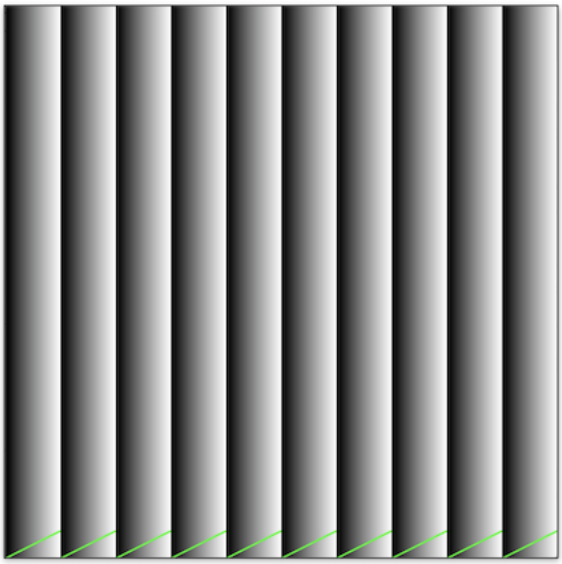


Sawtooth

The sawtooth wave enables a jagged oscillation — a value increases linearly and then resets.

```
float waveSawTooth(float t, float frequency, float amplitude)
{
  return (t * frequency - floor(t* frequency)) * amplitude;
}
```

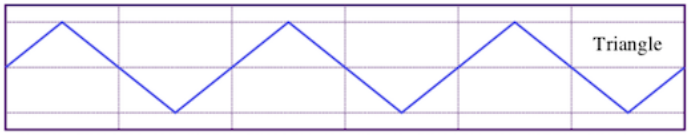


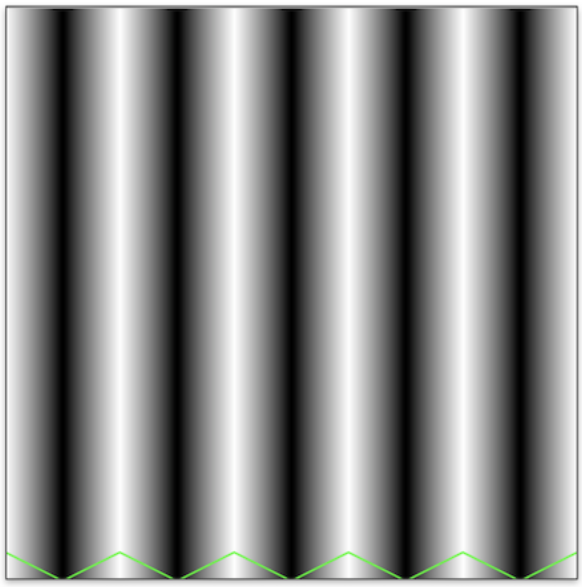


Triangle

The triangle wave enables a linear oscillation between two values.

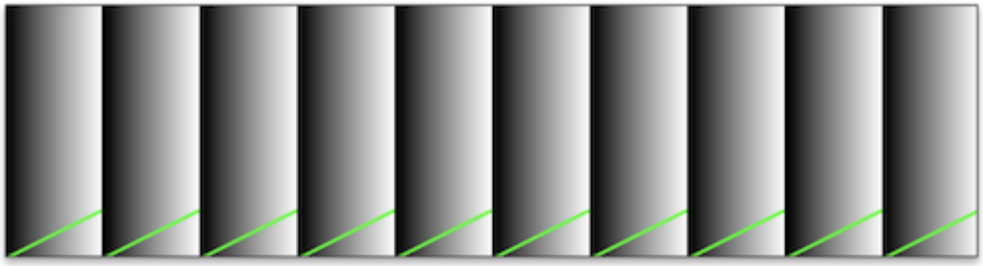
```
float waveTriangle(float t, float frequency, float amplitude)
{
  return abs((t * frequency) % amplitude - (0.5 * amplitude));
}
```





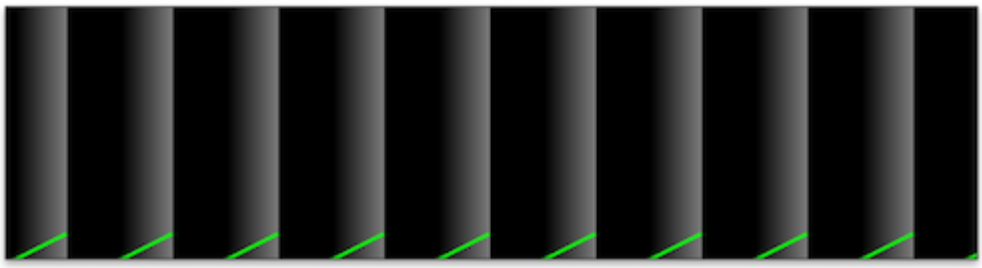
Again, the formula starts with:

```
float waveTriangle(float t, float frequency, float amplitude)
{
  return ((t * frequency) % amplitude);
}
```



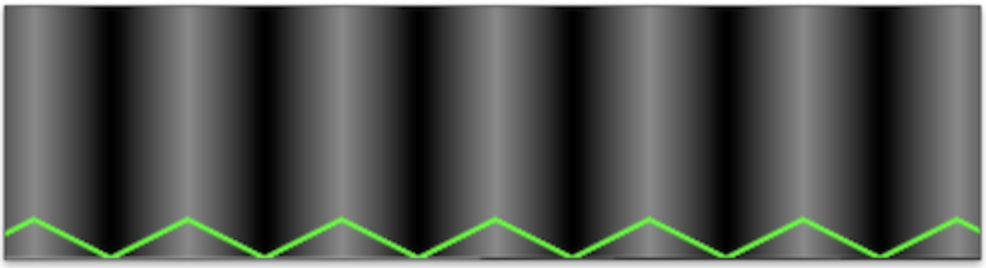
Then, we scale the triangle site by half in order to fit two of them :

```
float waveTriangle(float t, float frequency, float amplitude)
{
  return ((t * frequency) % amplitude - (0.5 * amplitude));
}
```



Lastly, we take the `abs` value in order to repeat the triangle site in the other direction:

```
float waveTriangle(float t, float frequency, float amplitude)
{
  return abs((t * frequency) % amplitude - (0.5 * amplitude));
}
```



Advanced Shapes

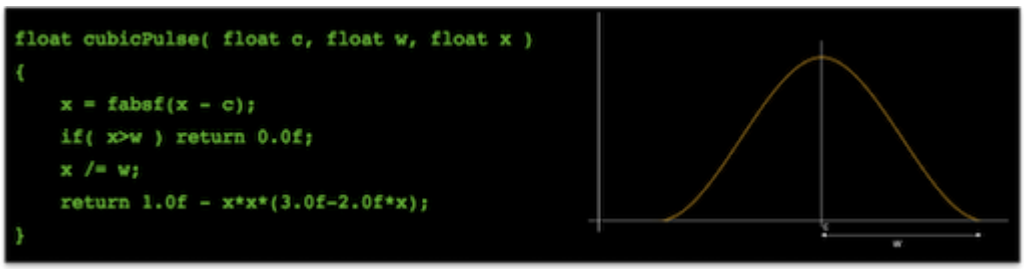
There are various other function shapes, which you could integrate into your design. The following examples are somewhat advanced and it is fine if you just skim through them and come back to them in case of need.

Pulse

The pulse function mimics a gaussian shape. Parameters:

- `c` controls centering
- `w` controls taper length

This function is useful for isolating a feature or for turning something on and off.



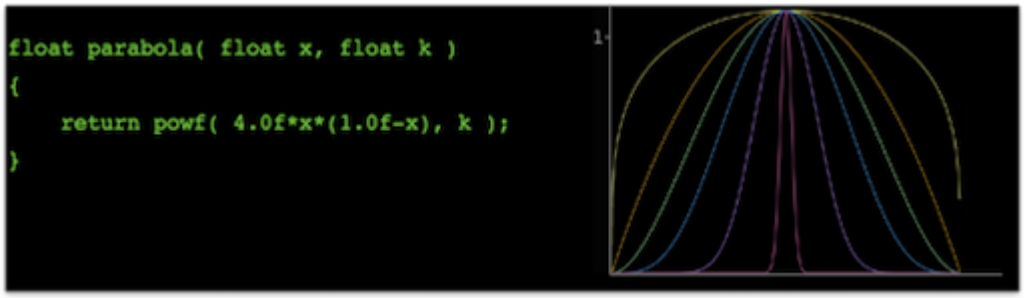
[iquilezles]

Parabola

The parabola function remaps the `[0, 1]` interval such that `f(0) = f(1) = 0` and `f(1/2) = 1`. Parameters:

- `k` controls steepness

Similar to pulse, use this function for symmetric shapes and for turning something on and off.

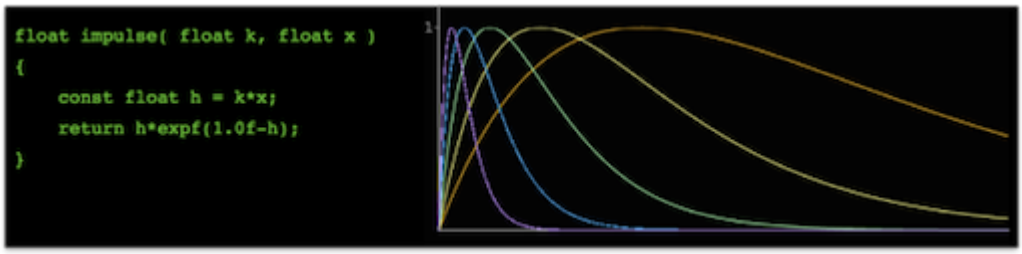


[iquilezles]

Impulse

The impulse function reaches a maximum value and then gradually decays. Parameters:

- `k` controls the rate of decay



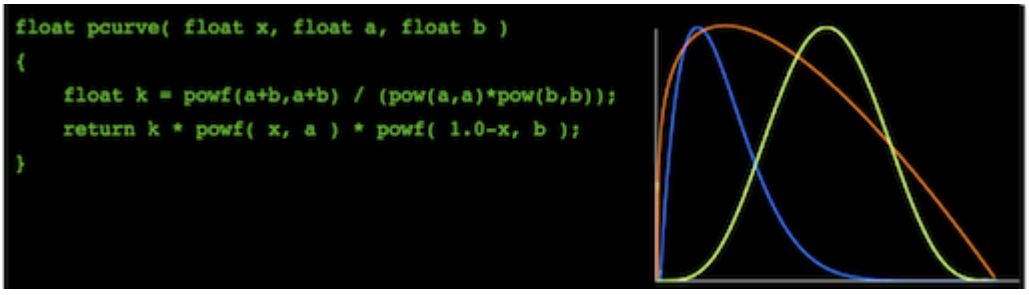
[iquilezles]

Power Curve

The power curve function skews a pulse-like curve to one side. Parameters:

- `a` and `b` control the bias toward either side

This function is good for creating shapes and for nudging signals off-center.



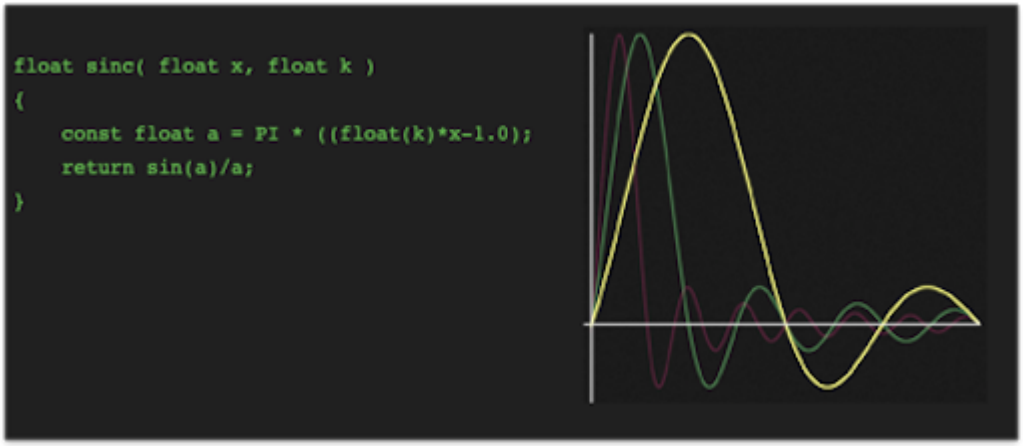
[iquilezles]

Sinc Curve

The sinc curve function is a phase shifter. Parameters:

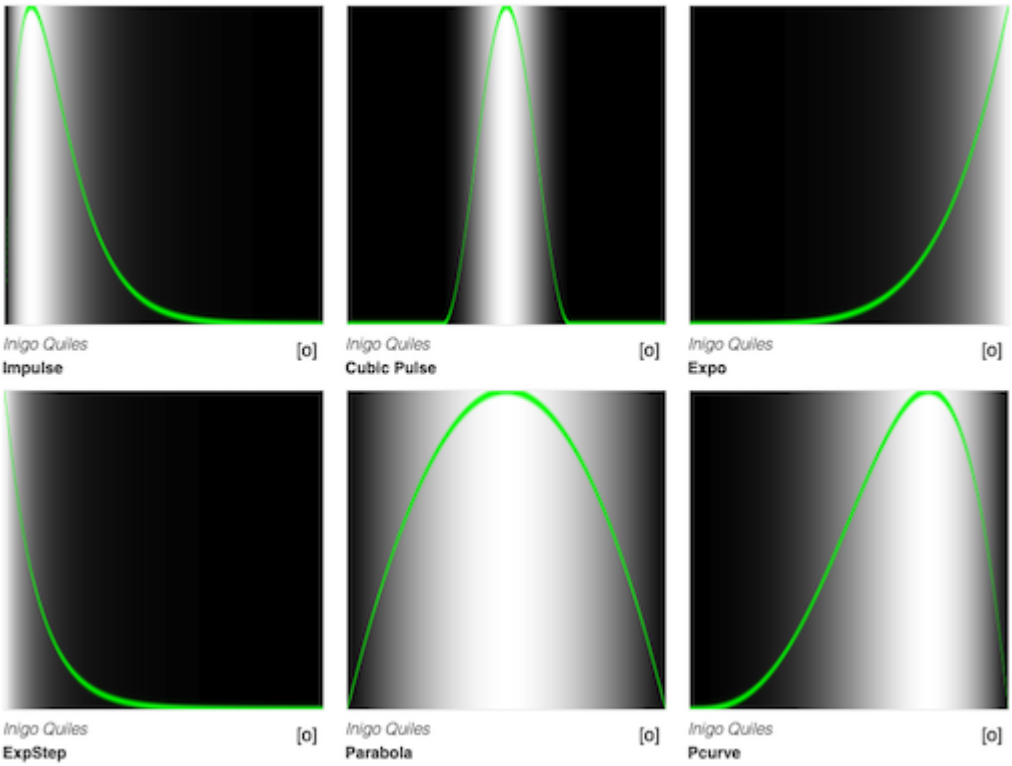
- k tweaks the amount of bounces

This function is useful for e.g. the creation of bounces.



[iquilezles]

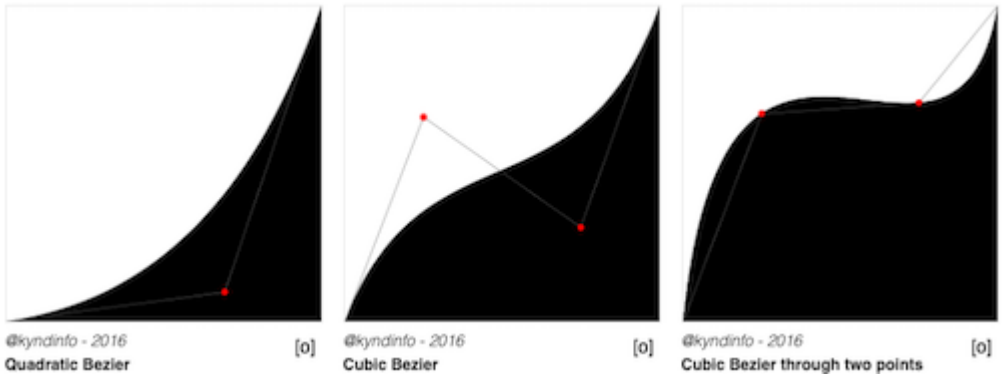
[6]



[thebookofshaders]

More functions

- Polynomial Shaping Functions
- Exponential Shaping Functions
- Circular & Elliptical Shaping Functions
- Bezier and Other Parametric Shaping Functions



[thebookofshaders]

[7]

Design Goals

To understand the above described different components is hopefully with some brain power manageable. But putting components together can be quite daunting. Also, don't be scared away by cryptic examples you will find on the web. Function design code is notoriously difficult to read as it is often optimized for performance.

The best side for finding shader inspirations is Shadertoy run by Inigo Quilez. ShaderToy is packed with very good examples (but also some bad ones...) and code to steal. Unfortunately, ShaderToy is slightly its own world with different variables namings and core functions. We will come back to the awesomeness that is ShaderToy in the Shader Programming workshop.

Whenever you find function designs that you would like to understand, you should try to find the overall *gist* of the design.

How to find the gist of a function design?

Divide and conquer:

- Test different values for constants and defines
- Separate functionalities, e.g. turn of animation, sound, interaction etc.
- Go line by line and display the result of each line separately
- Take out all scaling factors, offsets, etc.

Here, only practice and patience help.

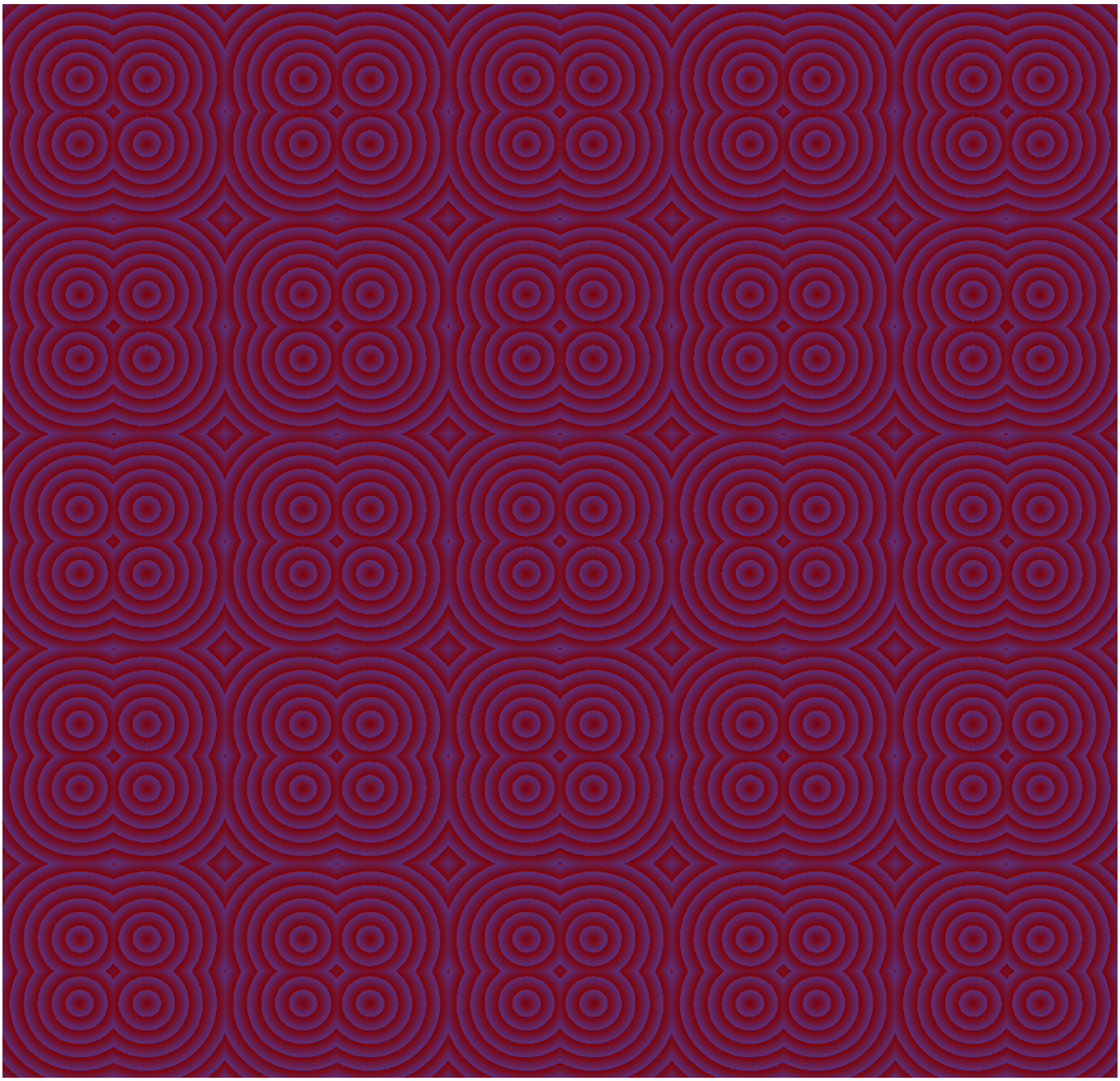
Then, at some point you will not only be as happy as this blobby creature but you might also be able to program this fully procedurally generated scene (including the renderer and such!), which is one of the masterpieces of Inigo Quilez.

[Happy Jumping by Iq]

If you are interested in how Inigo build this scene, there is a 6 hours (!) recorded live stream, deconstructing the Happy Jumping mathematical animation. I tried to watch the video several times but terribly failed each time. Inigo might be a shader mastermind, didactically he is not always.

Example

In this example, I am walking you through the steps to re-create this subtle pattern. It is a fairly easy design but include several of the most common approaches when putting functions together.



What do you see? What could be the steps to recreate this pattern?

One Cell

When working on repetitive patterns, one usually starts with one cell and repeats that cell in a second step.

Let's start with creating a circle by plotting the distance of each coordinate to the center point `0.5`, `0.5`.

- The `distance()` function calculates the distance between two points.
- The `mix()` function linearly interpolate between two values.

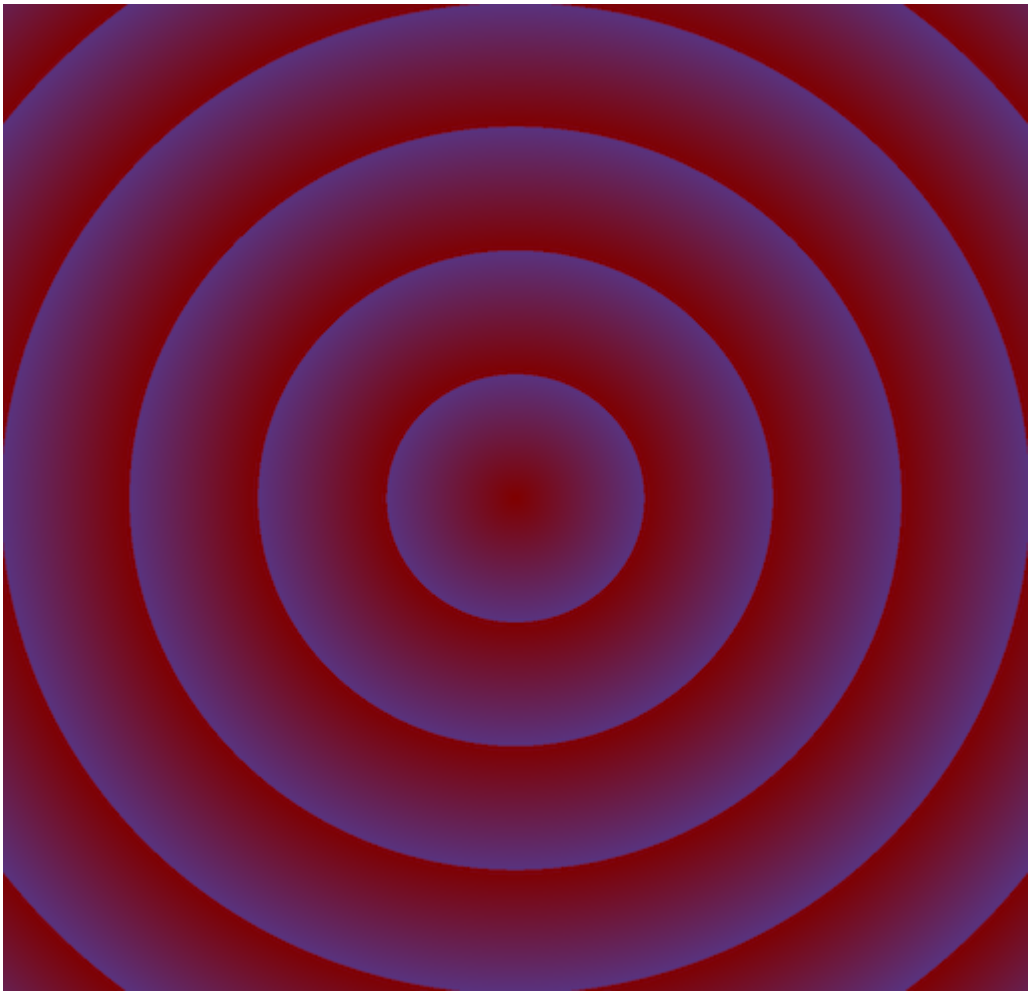
```
#ifdef GL_ES
precision mediump float;
#endif

uniform vec2 u_resolution;
uniform float u_time;
```

```
float CELLSIZE = 0.2; //relative, hence 0..1
vec2 OFFSET = vec2(0.3);

void main()
{

    vec2 coord = gl_FragCoord.xy/u_resolution;

    // 1. One Cell, distance to center point
    float d = distance(coord, vec2(0.5));

    vec3 color = mix(vec3(0.5, 0.0, 0.0), vec3(0.35, 0.2, 0.5), d);
    gl_FragColor = vec4(color, 1.0);
}
```

Next, let's create ridges with the `floor` function.

```
#ifdef GL_ES
precision mediump float;
#endif

uniform vec2 u_resolution;
uniform float u_time;
```

```
float CELLSIZE = 0.2; //relative, hence 0..1
vec2 OFFSET = vec2(0.3);

void main()
{

    vec2 coord = gl_FragCoord.xy/u_resolution;

    // 2. Ridges
    float d = distance(coord, vec2(0.5));
    d *= 8.0;
    d -= floor(d);

    vec3 color = mix(vec3(0.5, 0.0, 0.0), vec3(0.35, 0.2, 0.5), d);
    gl_FragColor = vec4(color, 1.0);
}
```

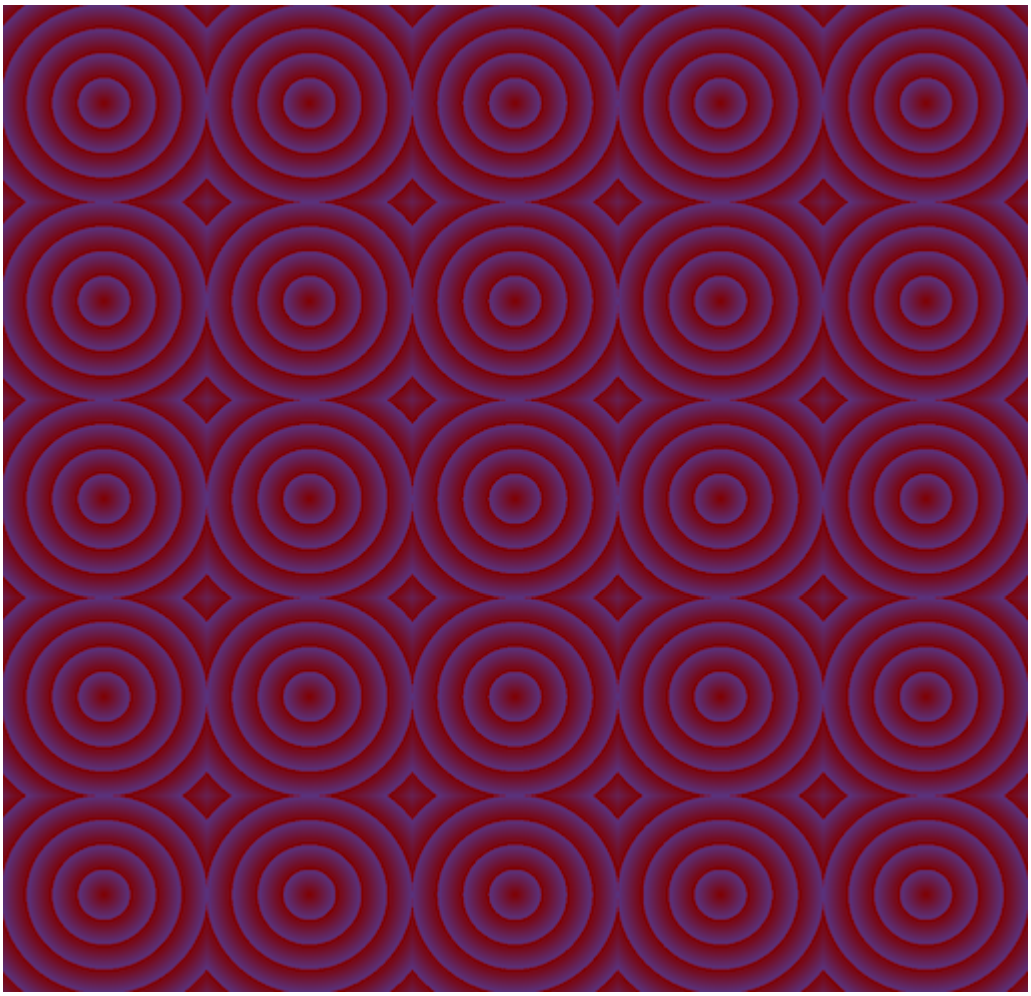


Repetitive Cells

Next, let's create the cells by dividing the 0..1 original x,y coordinate by the cell size.

```
#ifdef GL_ES
precision mediump float;
```

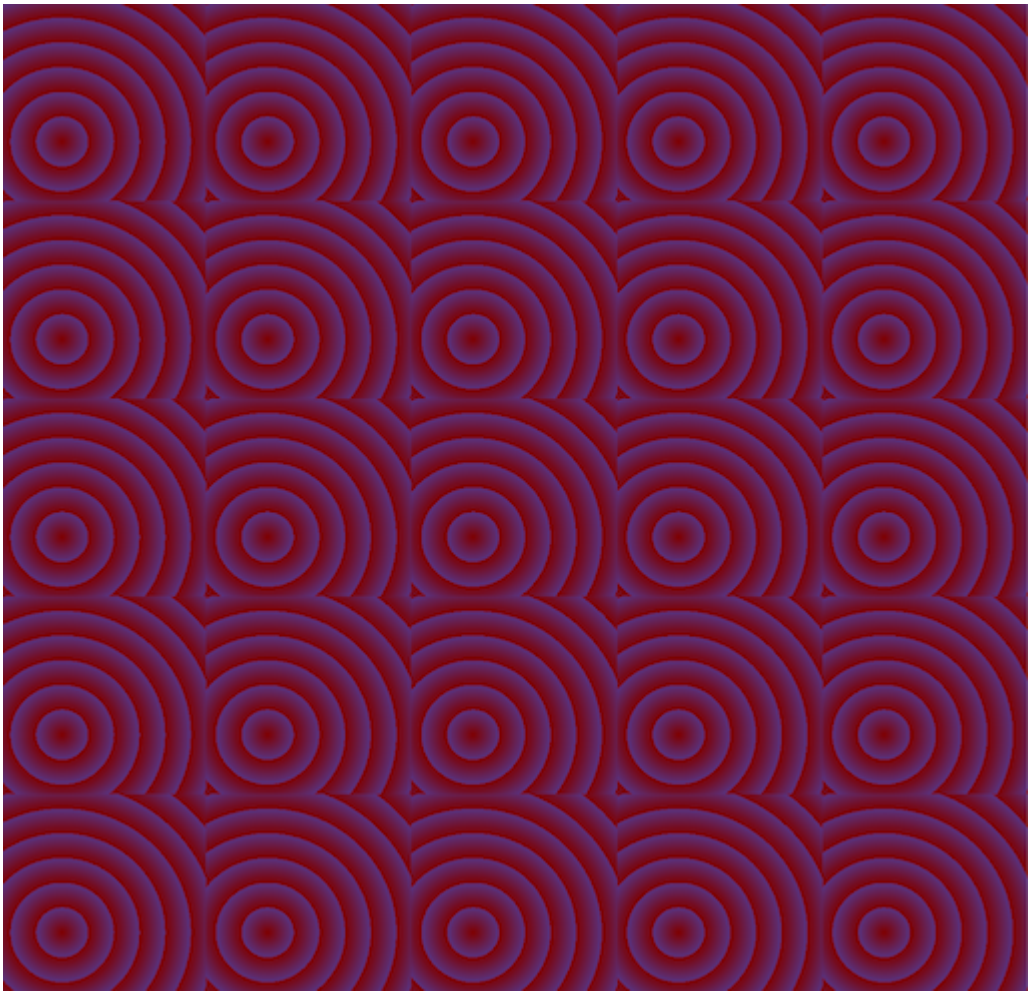
```
#endif

uniform vec2 u_resolution;
uniform float u_time;

float CELLSIZE = 0.2; //relative, hence 0..1
vec2 OFFSET = vec2(0.3);

void main()
{

    vec2 coord = gl_FragCoord.xy/u_resolution;

    // 3. Create Cells
    // Get into one cell
    float x = coord.x / CELLSIZE;
    float y = coord.y / CELLSIZE;
    x -= floor(x);
    y -= floor(y);

    float d = distance(vec2(x, y), vec2(0.5));
    d *= 8.0;
    d -= floor(d);

    vec3 color = mix(vec3(0.5, 0.0, 0.0), vec3(0.35, 0.2, 0.5), d);
    gl_FragColor = vec4(color, 1.0);
}
```



Now, we simply move the center point (the one that we are computing the distance to for the circles).

```
#ifdef GL_ES
precision mediump float;
#endif

uniform vec2 u_resolution;
uniform float u_time;

float CELLSIZE = 0.2; //relative, hence 0..1
vec2 OFFSET = vec2(0.3);

void main()
{

    vec2 coord = gl_FragCoord.xy/u_resolution;

    // 4. Move center point by OFFSET to compute distance to
    float x = coord.x / CELLSIZE;
    float y = coord.y / CELLSIZE;
    x -= floor(x);
    y -= floor(y);

    float d = distance(vec2(x, y), OFFSET);
    d *= 8.0;
    d -= floor(d);
```

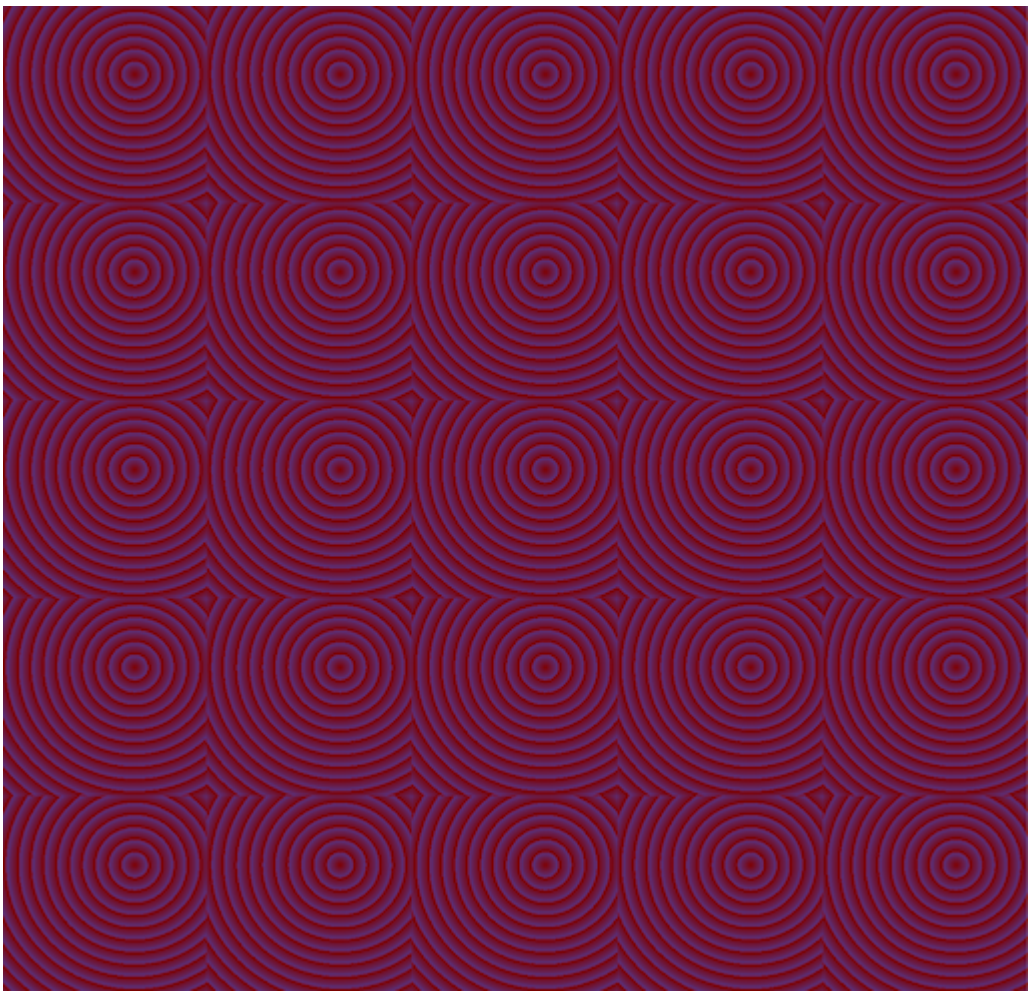
```
    vec3 color = mix(vec3(0.5, 0.0, 0.0), vec3(0.35, 0.2, 0.5), d);
    gl_FragColor = vec4(color, 1.0);
}
```



Lastly, we want to repeat the pattern within the cell as well and also flip it. For this we remap the original value range from 0..1 to -1..1 and take the absolute of those values.

```
#ifdef GL_ES
precision mediump float;
#endif

uniform vec2 u_resolution;
uniform float u_time;

float CELLSIZE = 0.2; //relative, hence 0..1
vec2 OFFSET = vec2(0.3);

void main()
{

    vec2 coord = gl_FragCoord.xy/u_resolution;
```

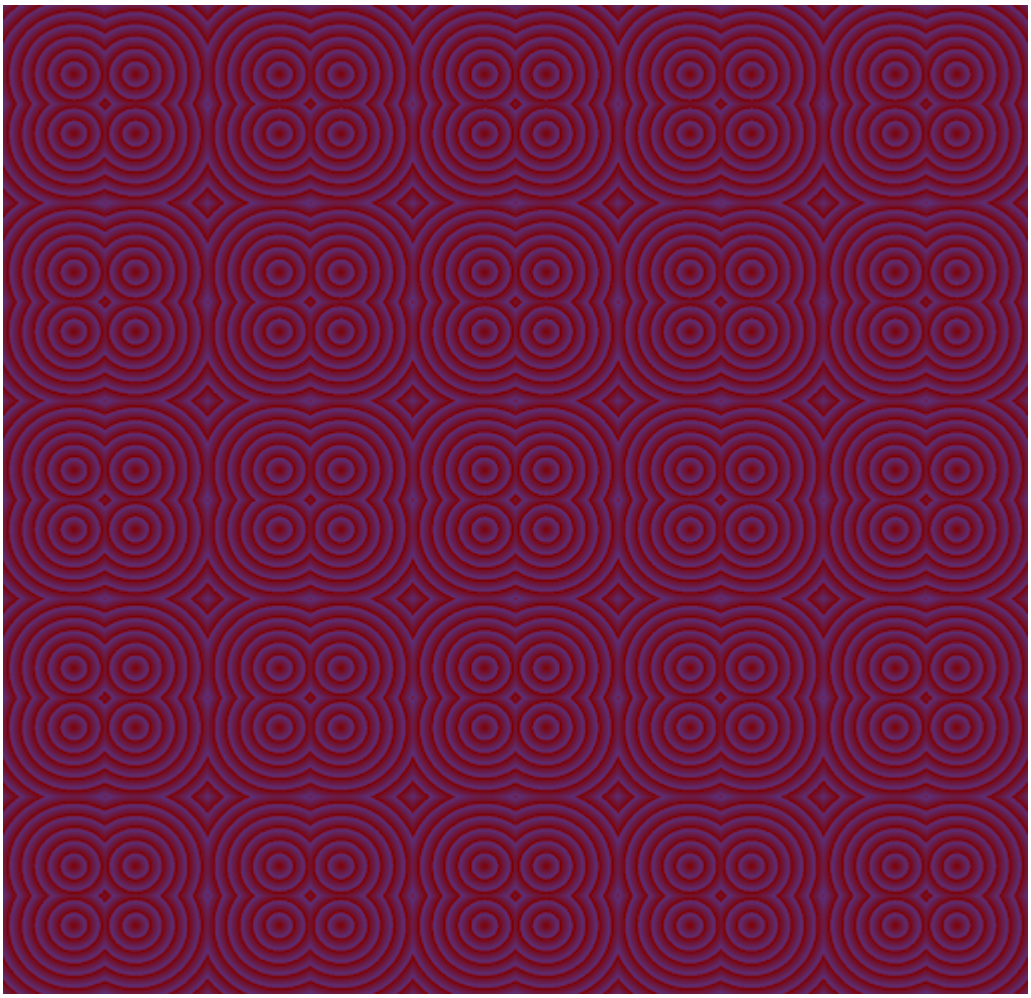
```
    // 5a. Remapping the range

    float x = coord.x / CELLSIZE;
    float y = coord.y / CELLSIZE;
    x -= floor(x);
    y -= floor(y);

    // Modify value range from 0..1 to -1..1
    float x_remap = (x - 0.5) * 2.0;
    float y_remap = (y - 0.5) * 2.0;

    float d = distance(vec2((x_remap), (y_remap)), OFFSET);
    d *= 8.0;
    d -= floor(d);

    vec3 color = mix(vec3(0.5, 0.0, 0.0), vec3(0.35, 0.2, 0.5), d);
    gl_FragColor = vec4(color, 1.0);
}
```



... and taking the absolute of the new value range

```
#ifdef GL_ES
precision mediump float;
#endif
```

```
uniform vec2 u_resolution;
uniform float u_time;

float CELLSIZE = 0.2; //relative, hence 0..1
vec2 OFFSET = vec2(0.3);

void main()
{

    vec2 coord = gl_FragCoord.xy/u_resolution;


    // 5a. Remapping the range to -1..1
    // 5b. and taking the absolute values

    float x = coord.x / CELLSIZE;
    float y = coord.y / CELLSIZE;
    x -= floor(x);
    y -= floor(y);

    // Modify value range from 0..1 to -1..1
    float x_remap = (x - 0.5) * 2.0;
    float y_remap = (y - 0.5) * 2.0;

    float d = distance(vec2(abs(x_remap), abs(y_remap)), OFFSET);
    d *= 8.0;
    d -= floor(d);

    vec3 color = mix(vec3(0.5, 0.0, 0.0), vec3(0.35, 0.2, 0.5), d);
    gl_FragColor = vec4(color, 1.0);
}
```



I hope you didn't go blind by this example... sorry.

Next

Next we are going to add one more brush to our tool box: noise functions!

.center[

[thebookofshaders]]

References

[1] Scratch a Pixel
[2] Wiki - Clamping
[3] Wiki - Smooth Step
[4] CIS 700 - Special Topics in Computer Graphics, University of Pennsylvania, Rachel Hwang
[5] Houdini Docs - Ramp Basics
[6] Inigo Quilez - Functions
[7] Vivo, Patricio Gonzalez, and Lowe, Jen (2015). The Book of Shaders.
[8] Floersch, Brian (2019). Intro to 2D signed distance functions and drawing touch controls with the GPU
[9] Böhringer, Ronja (2018). 2D Signed Distance Field Basics

The End