GENERATING THINGS WITH CODE

PROCEDURAL CONTENT GENERATION (PCG)

generation is programmatic & often pseudo-random

design a *process* rather than an *object*

PCG ARE HEAVILY IS USED IN DESIGN, GAMES, ART, VFX, ARCHITECTURE...

NOT IMPORTANT

- coding platform
- coding language
- coding tools & toolchain
- capture device & data sources

IMPORTANT

1. data
2. space
3. process

WHAT PCGS ARE GOOD AT

MODELIZATION

create the model of a system

SIMULATION

use the model to process data

EXPLORATION

change the model's parameter

COROLLARY: VISUALISATION

rendering the model: 2D, 3D graphics, video, tangible objects, installation...


no man's sky

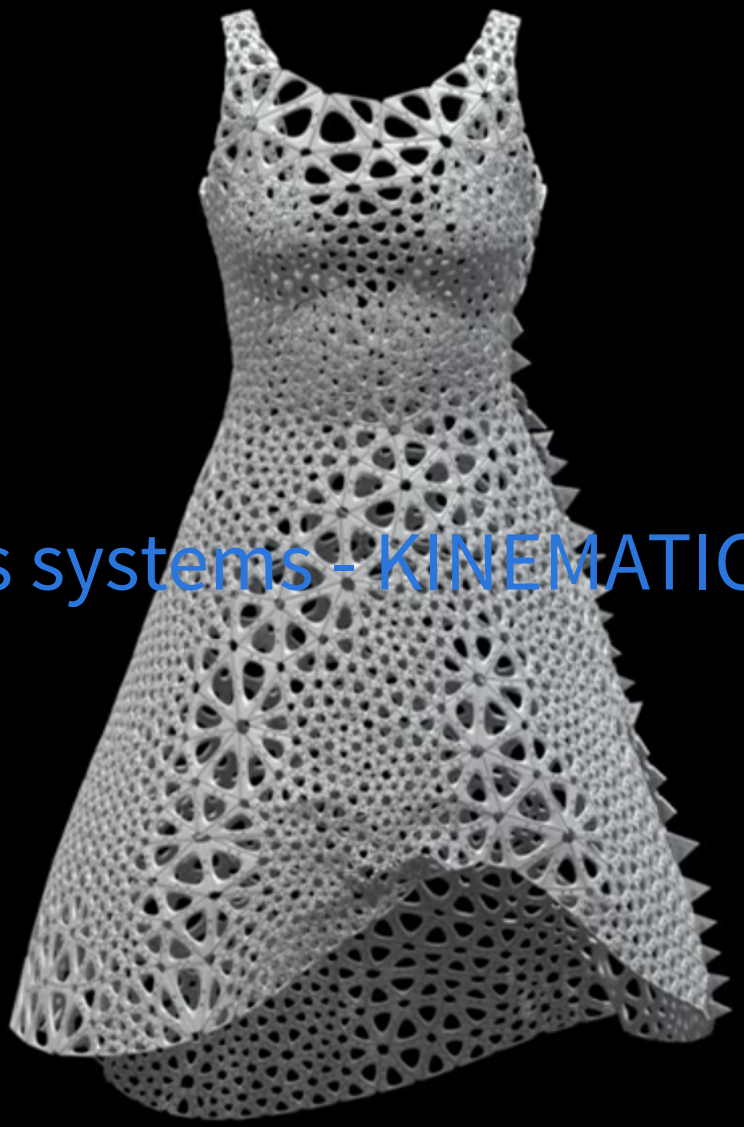
no man's sky

model: universe generator

simulation: update the model with game logic

exploration: move & interact with the model

visualisation: rendering a 3D scene



nervous systems - KINEMATICS FOLD

nervous systems - KINEMATICS FOLD

model: triangle based patch modeler

simulation: apply physics rules to the model

exploration: change the base model

visualisation: 3D renders & 3D print


unnamed sound sculpture

unnamed sound sculpture

system: use particles to represent a 3D object

simulation: playback of a recorded sequence

exploration: move a virtual camera through space

visualisation: render particles in 3D

unnamed sound sculpture

system rules:

- spawn particle at a 3D location
- make particle fall & disappear

the particles move & disappear *procedurally*

this piece is tightly related to the kinect data

UPSIDES

- the computer does the boring part
- few paramaters give many variations
- very good at serial content
- *emergent* behaviours can produce surprising or unexpected results

DOWNSIDES

- mathematical uniqueness is often not enough to create significant differences
- hard to maintain consistent variations while using numerous variables
- hard to assess a "good" settings human curation required

1. DATA

N dimensions

CONTINUOUS VS DISCRETE

the world is a *continuous* system
computers process *discrete* data

DISCRETIZING THE WORLD

continuous data are **measured**

discrete data are **counted**

CONTINUOUS ⇨ DISCRETE

time ⇨ interval

light ⇨ color palette

reaction ⇨ process

phenomenon ⇨ simulation

N-DIMENSIONAL OBJECTS

any given object can be described by
a variable number of *dimensions*

the values of each dimension of an object
can vary without affecting the others

LIGHT'S DIMENSIONS

direction, speed, wavelength, energy

COMPUTER REPRESENTATION OF A UNIDIMENSIONAL DATUM

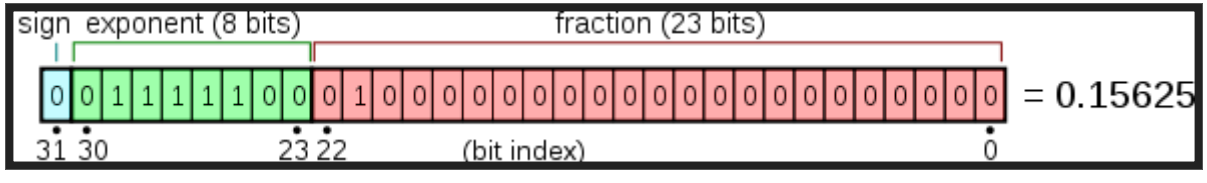
the BIT
0 / 1

limitation: it can only describe whether a given property of the object "is" or "isn't"

1D *EXTENSION PACK*

- bit (0/1)
- byte / octet: 8 bits & 256 integer values
- int, float, uint, double: 8, 16, 32, 64 bits
- single character: 7 bits (US ASCII), 32 bit (UTF-32)

in the computers's memory, a number
or a character is stored as an array of bits



so, technically, a number is itself
a N-dimensional object

2 DIMENSIONAL DATA

series of unidimesnsional data

- typed linear array: ["a", "b", "c"] / [0, 1, 2]
- string : "Nicolas Barradeau"
- an empty image: [width * height]

3 DIMENSIONAL DATA

combinatorics of unidimensional data

- color: { R, G, B }
- a vector: { X, Y, Z }
- binary pixel: { X, Y, value }

N-DIMENSIONAL DATA

complex *data structures*

- pixel, color: { X, Y, R, G, B }
- pixel, color & alpha: { X, Y, A, R, G, B }
- stereo sound: { left, right, time, sampling rate }
- video: pixel, color, sound, time
- 3D object: XYZ, indices, normals, colors, uvs...

?-DIMENSIONS

how many dimensions to a TWEET?


```
{
 "coordinates": null,
 "favorited": false,
 "created_at": "Wed Sep 05 00:37:15 +0000 2012",
 "truncated": false,
 "id_str": "243145735212777472",
 "entities": {
   "urls": [
   ],
   "hashtags": [
     {
       "text": "peterfalk",
       "indices": [
         35,
         45
```

a single tweet can potentially be represented in ~72 dimensions, often themselves N-dimensional

DATA ACQUISITION

YOUR MOBILE PHONE

photos, videos, microphone & GPS positions
as well as some "hidden" sensors:

- **TYPE_ACCELEROMETER** Motion detection (shake, tilt, etc.).
- **TYPE_AMBIENT_TEMPERATURE** Monitoring air temperatures.
- **TYPE_GRAVITY** Motion detection (shake, tilt, etc.).
- **TYPE_GYROSCOPE** Rotation detection (spin, turn, etc.).

OPEN DATA

- statistical analysis
- mostly "geo-social" data
- often geolocated

TRACKING DEVICES

- webcam
- microphone
- kinect
- leap motion
- VR devices: VIVE, Oculus

CUSTOM DEVICES

- Arduino boards + Processing
- all manners of cheap sensors: gesture detection, heat detector, 360° scanners, wind, rain, smoke...

SYNTHETIC DATA

- noise, fractals, turbulence...
- photogrammetry
  (build 3D models from 2D pictures)
- GANs (AI)

there are plenty of data around us

we use more and more complex data (feature vectors)

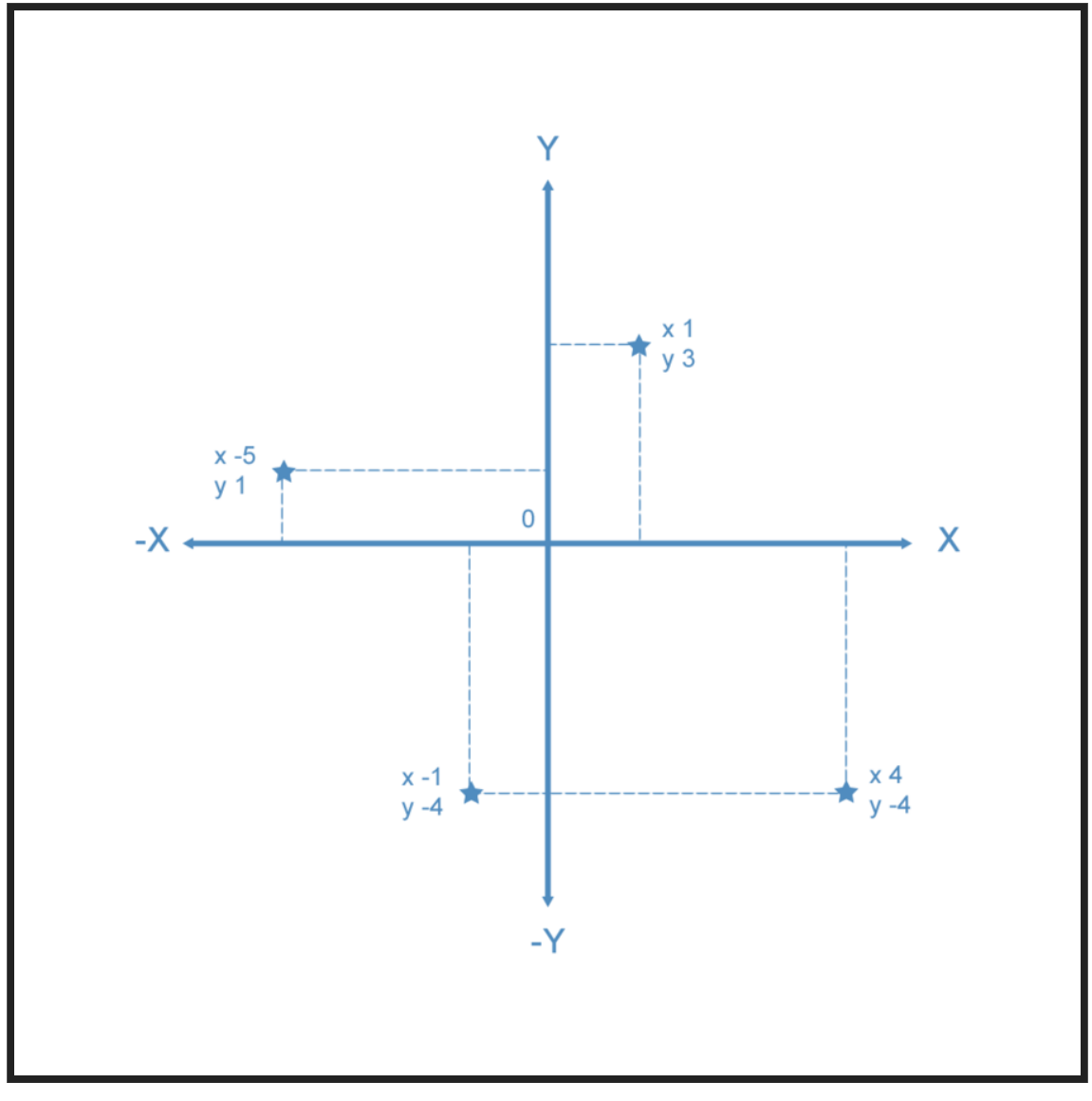
reduced to sets of *unidimensional* variables,
these data can *shift* between various *spaces*

2. SPACE

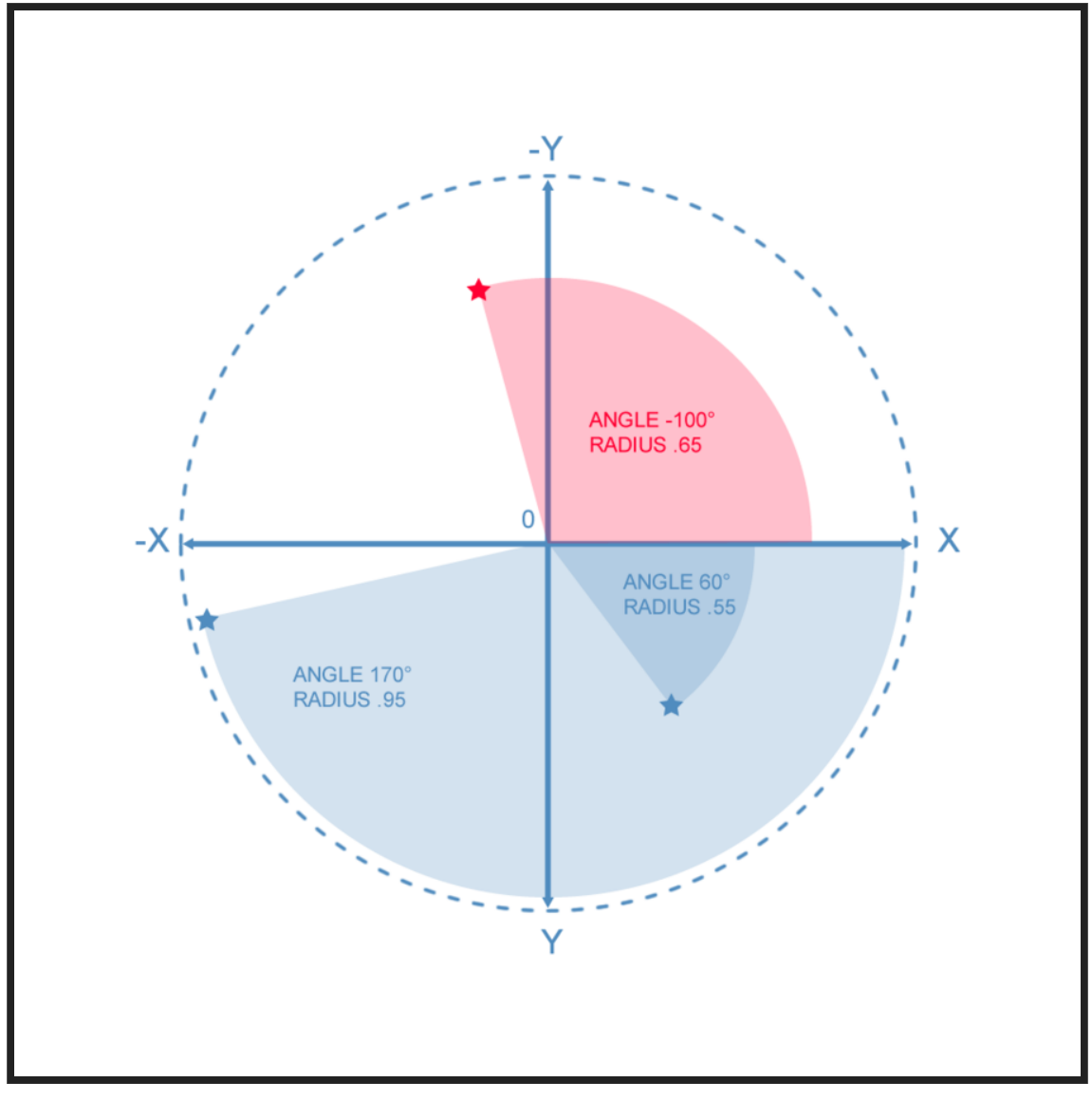
fields of representation

GEOMETRIC SPACES

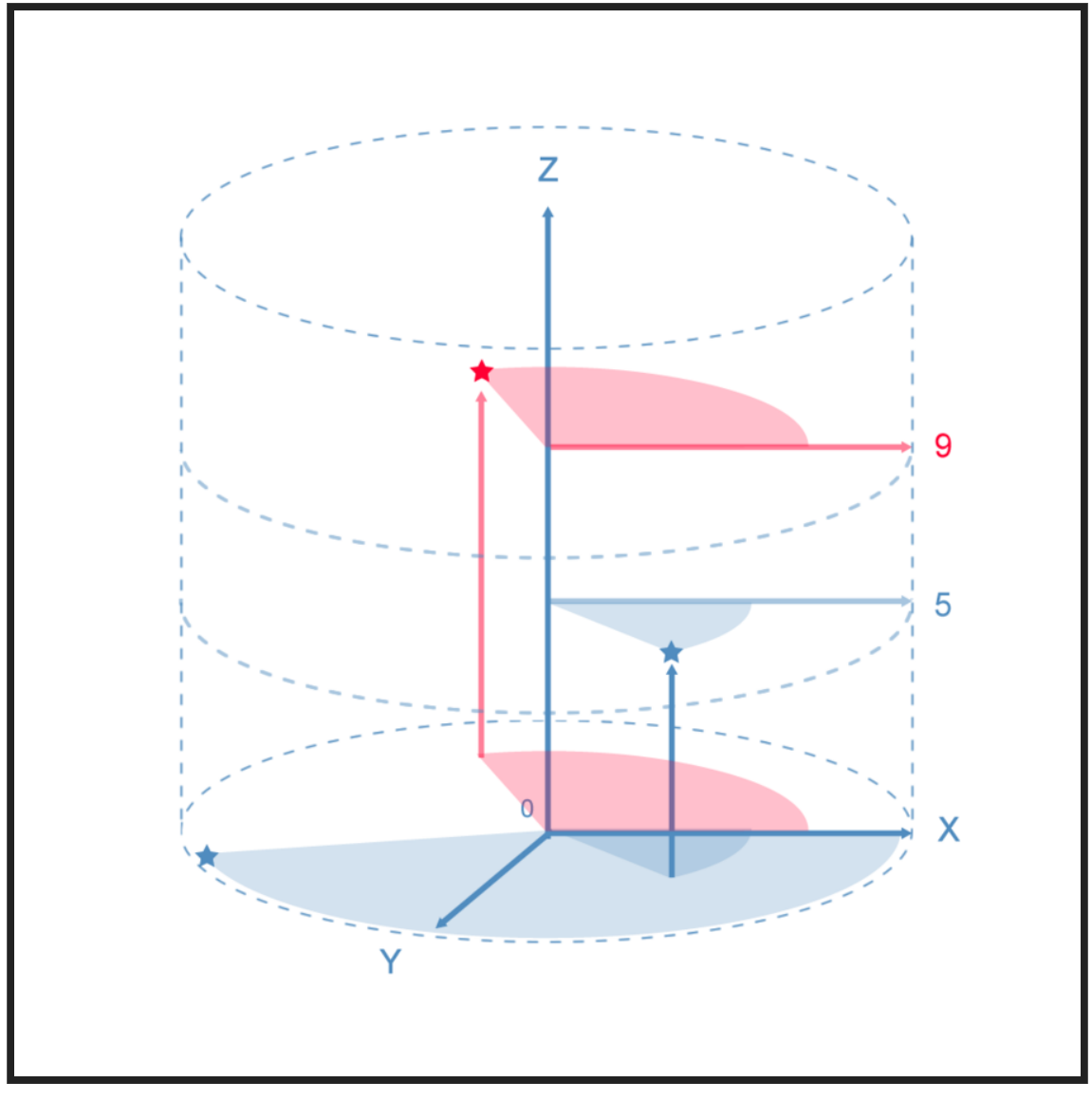
2D CARTESIAN SPACE



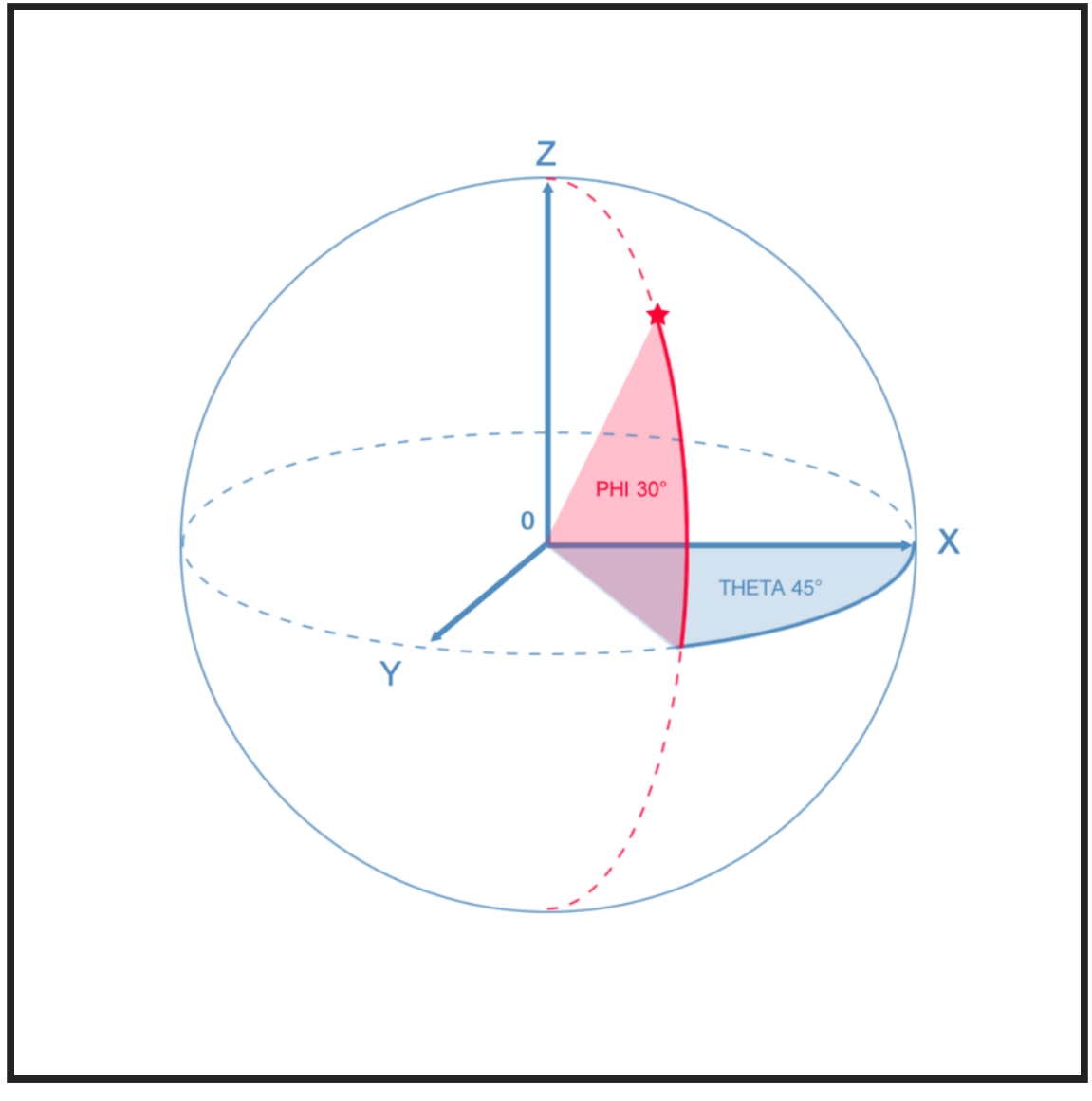
POLAR SPACE


-Y
ANGLE -100°
RADIUS .65
0
-X
X
ANGLE 60°
RADIUS .55
ANGLE 170°
RADIUS .95
Y

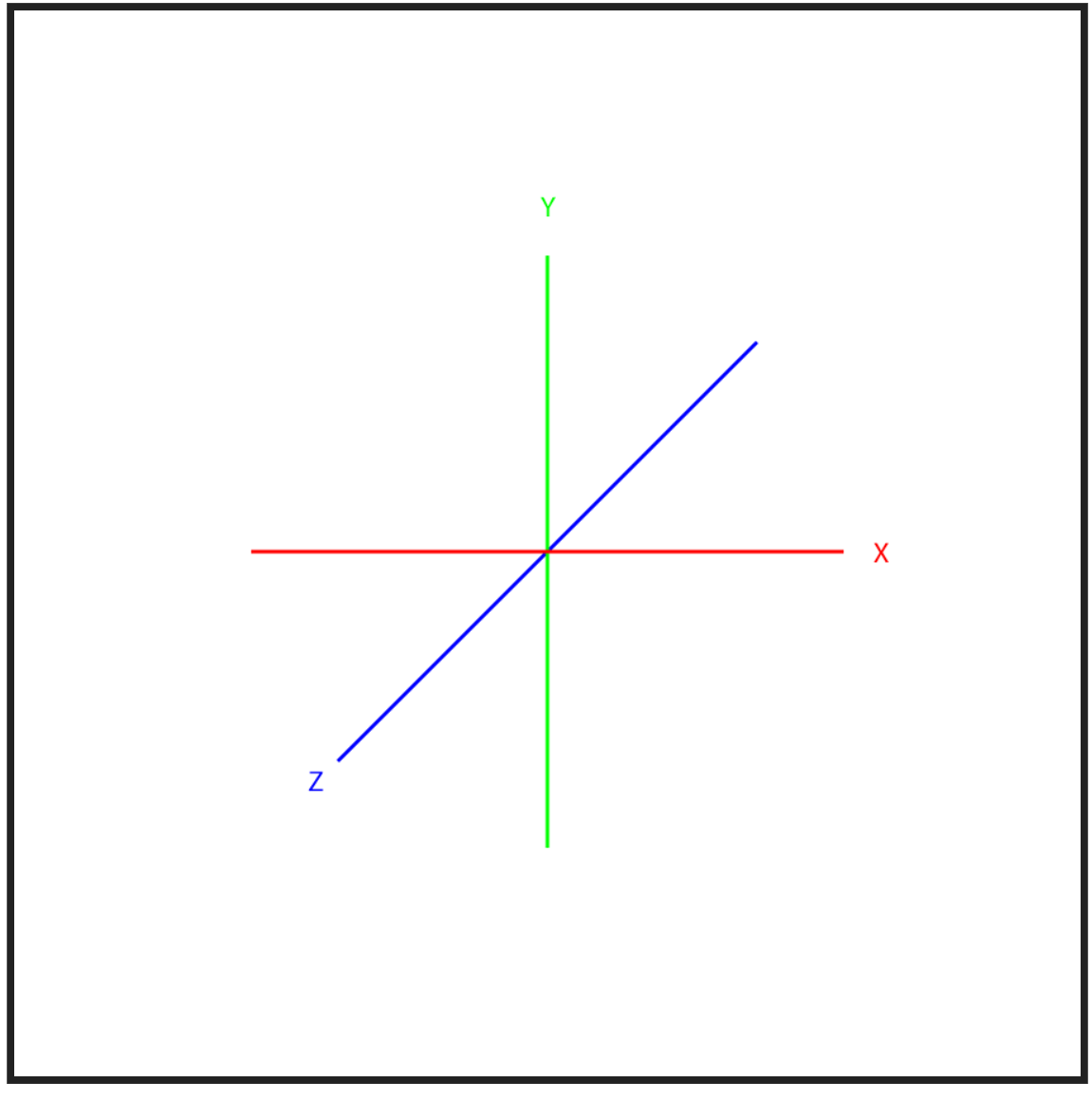
CYLINDRICAL SPACE



SPHERICAL SPACE

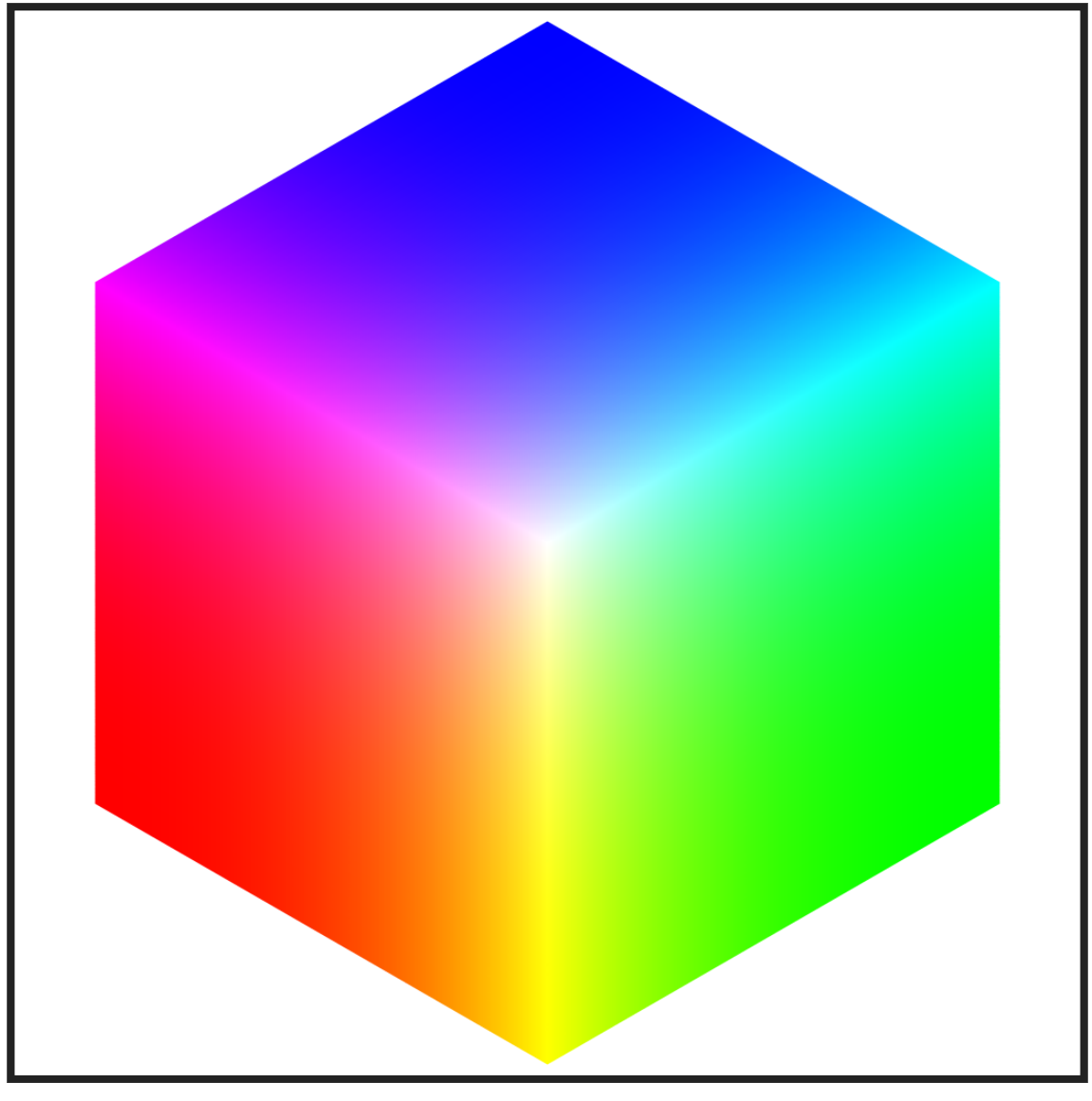


3D CARTESIAN SPACE

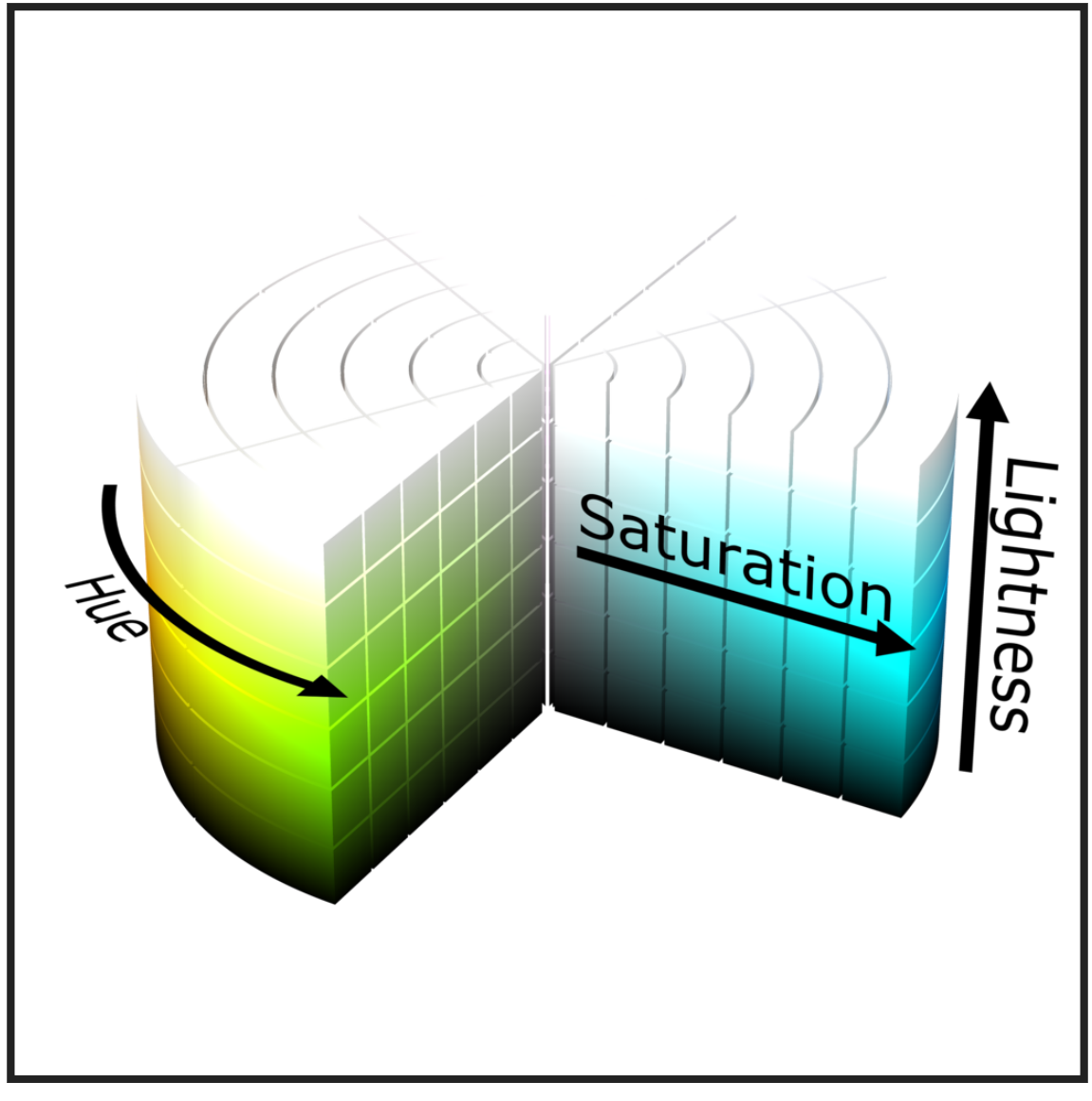


COLOR SPACES

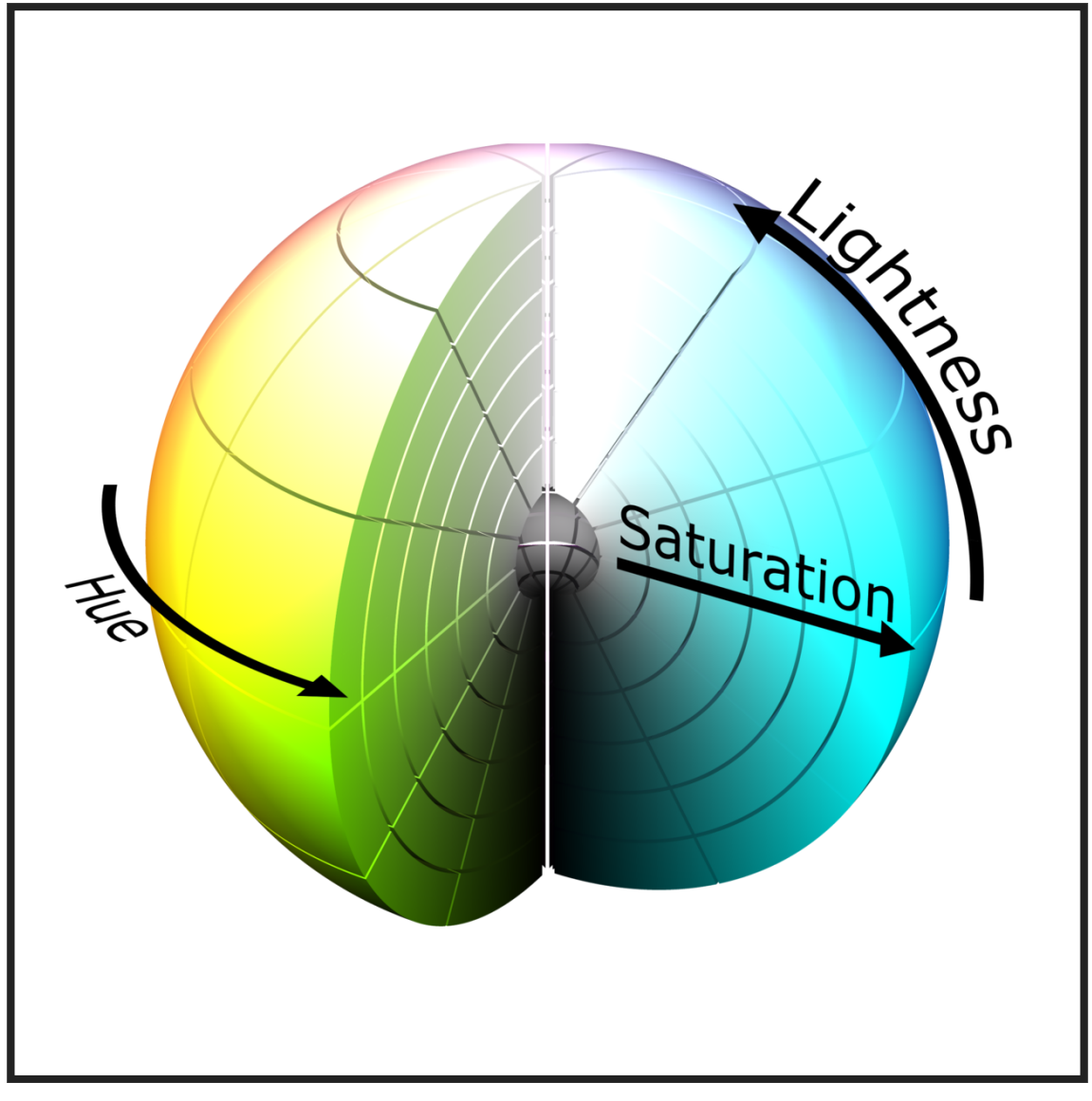
RGB COLOR SPACE



HSL COLOR SPACE

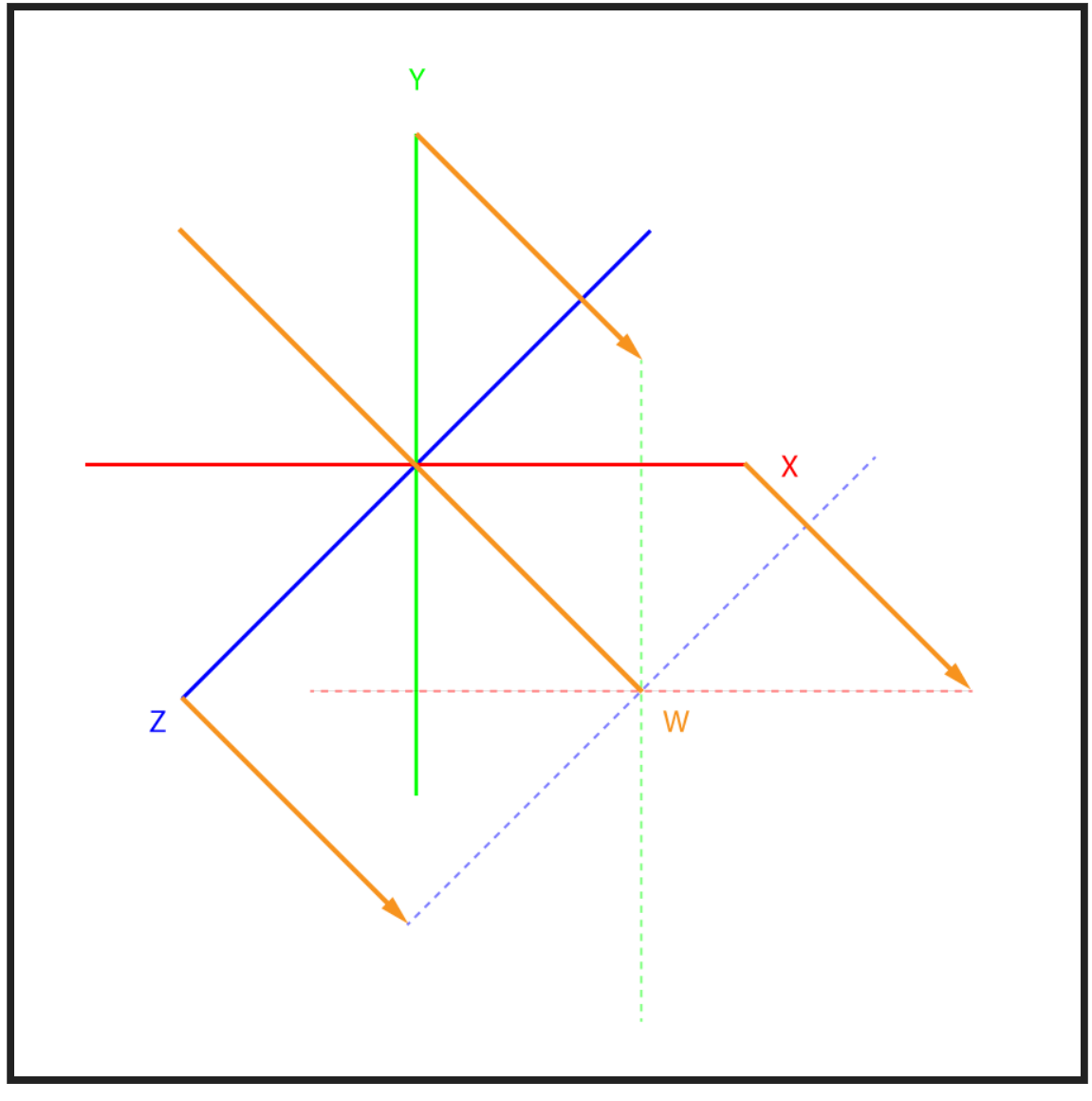


HSL COLOR SPACE

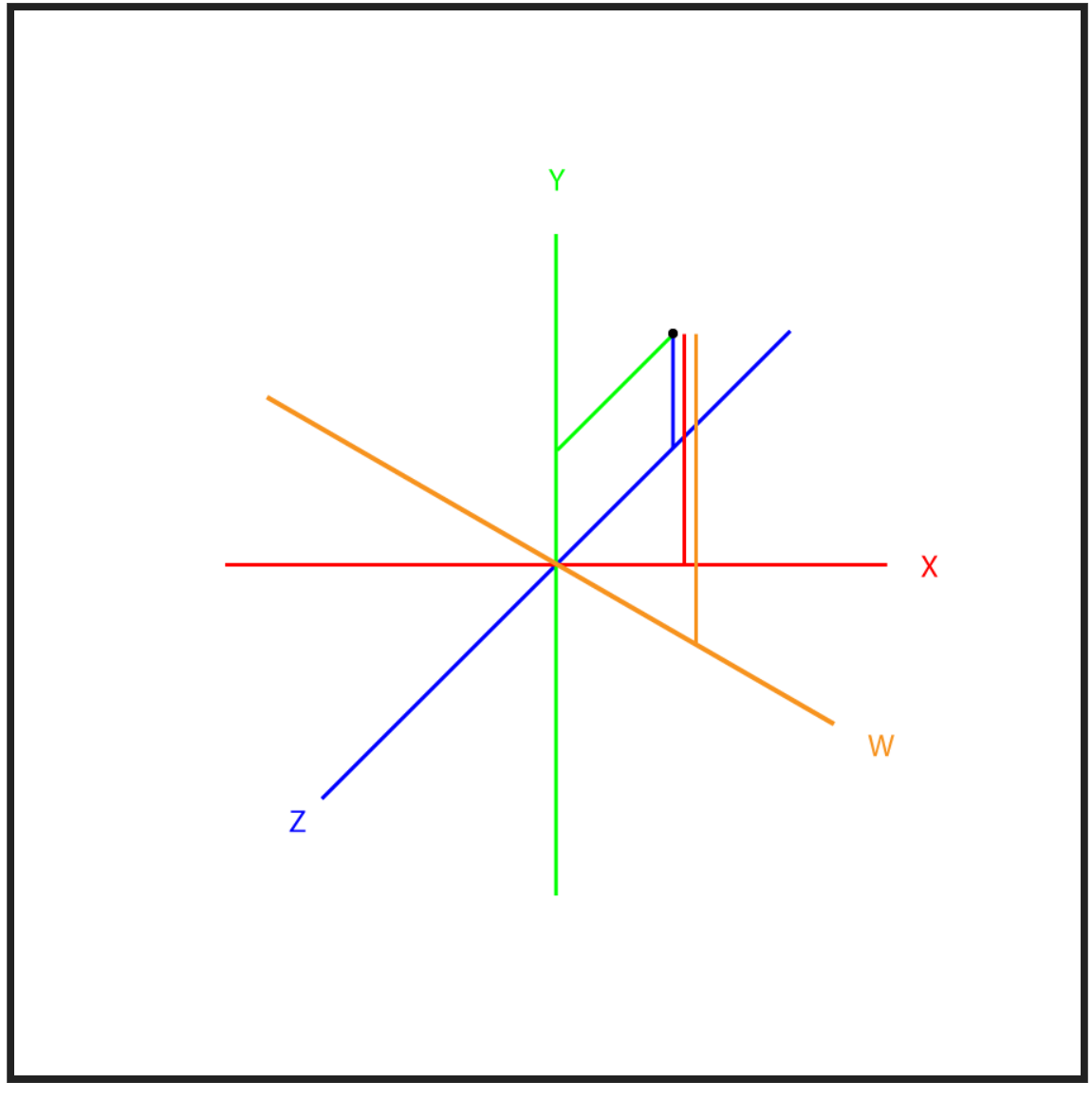


HIGHER DIMENSIONS

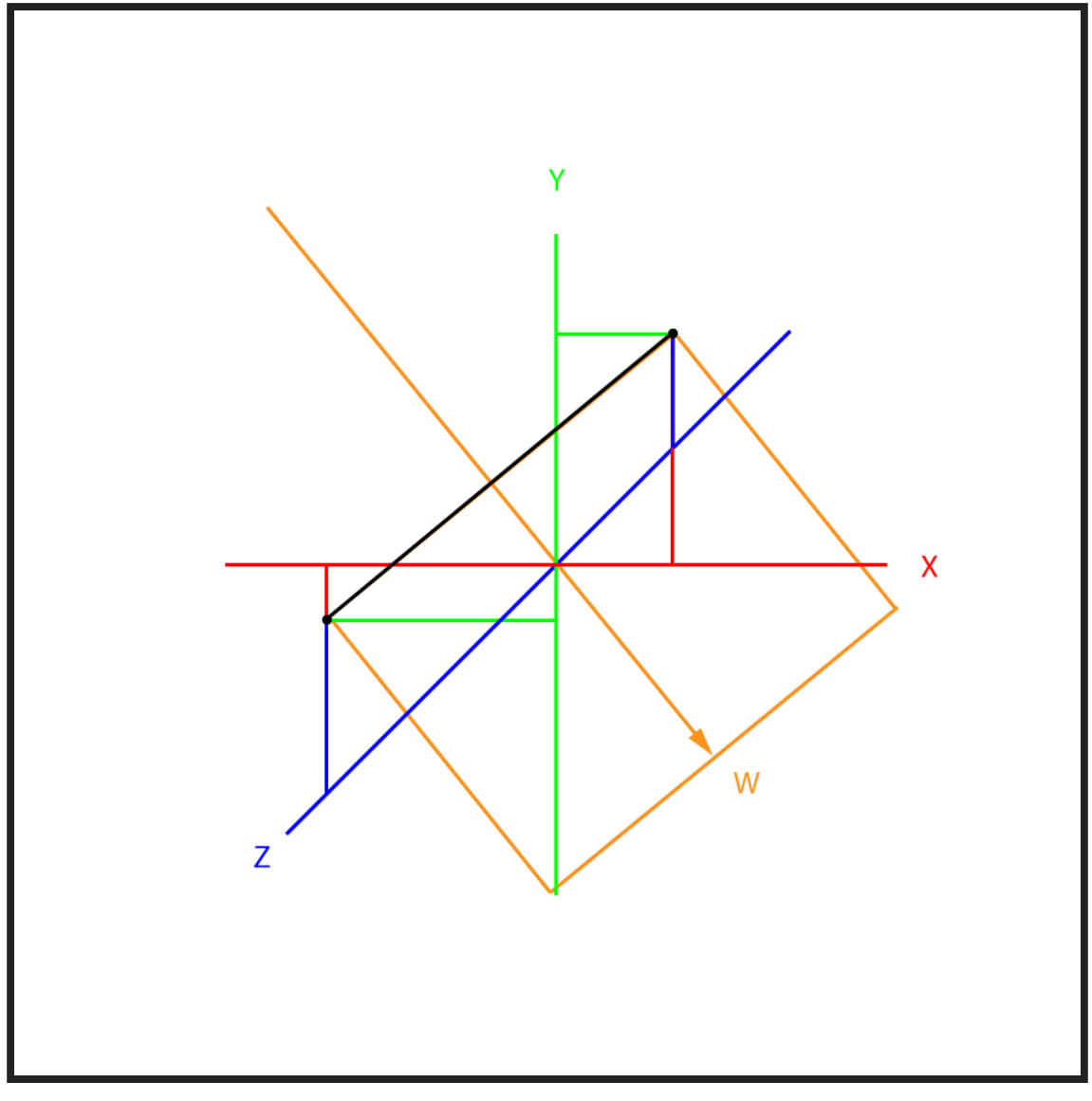
4D SPACE



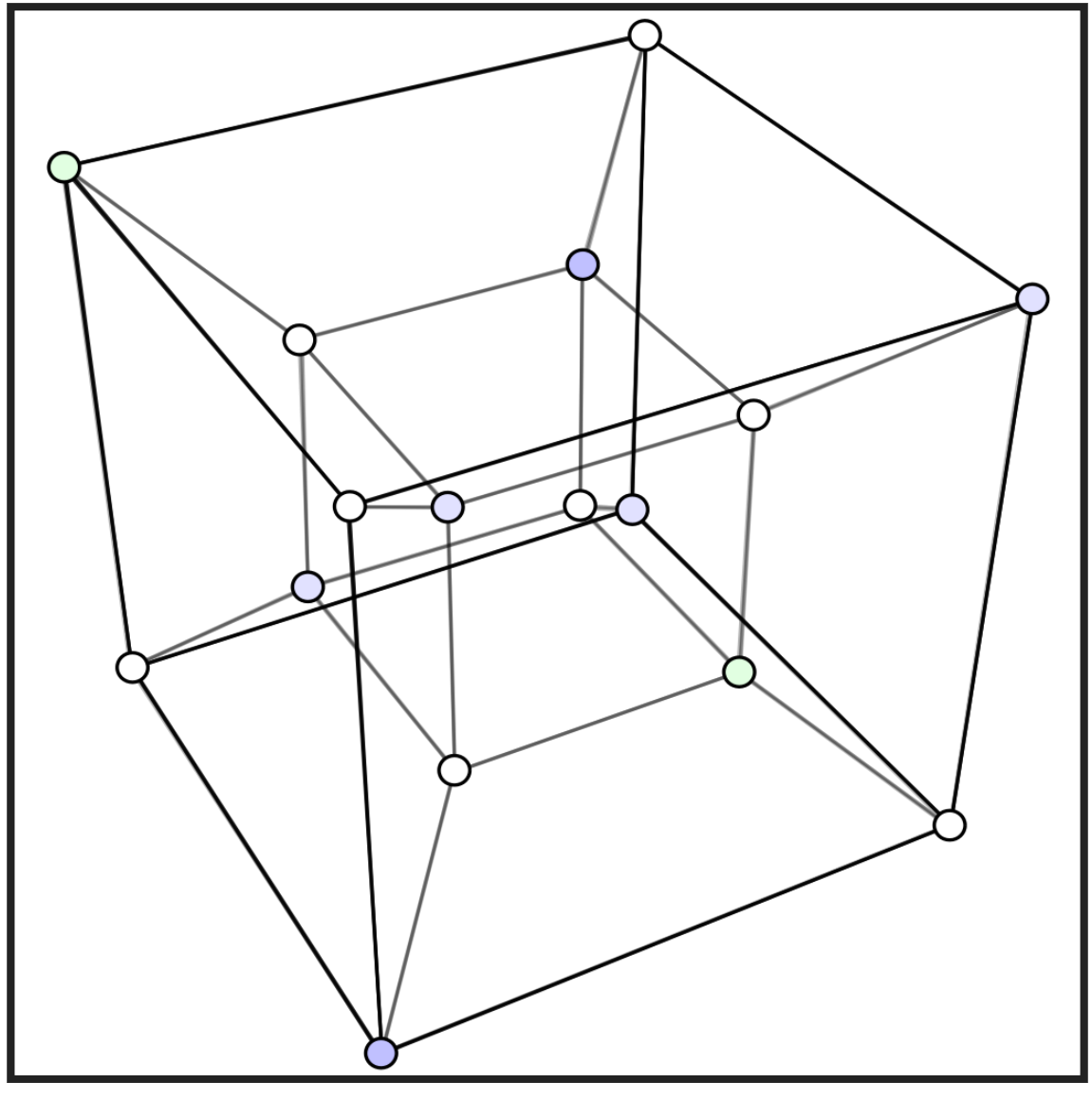
POINT IN A 4D SPACE



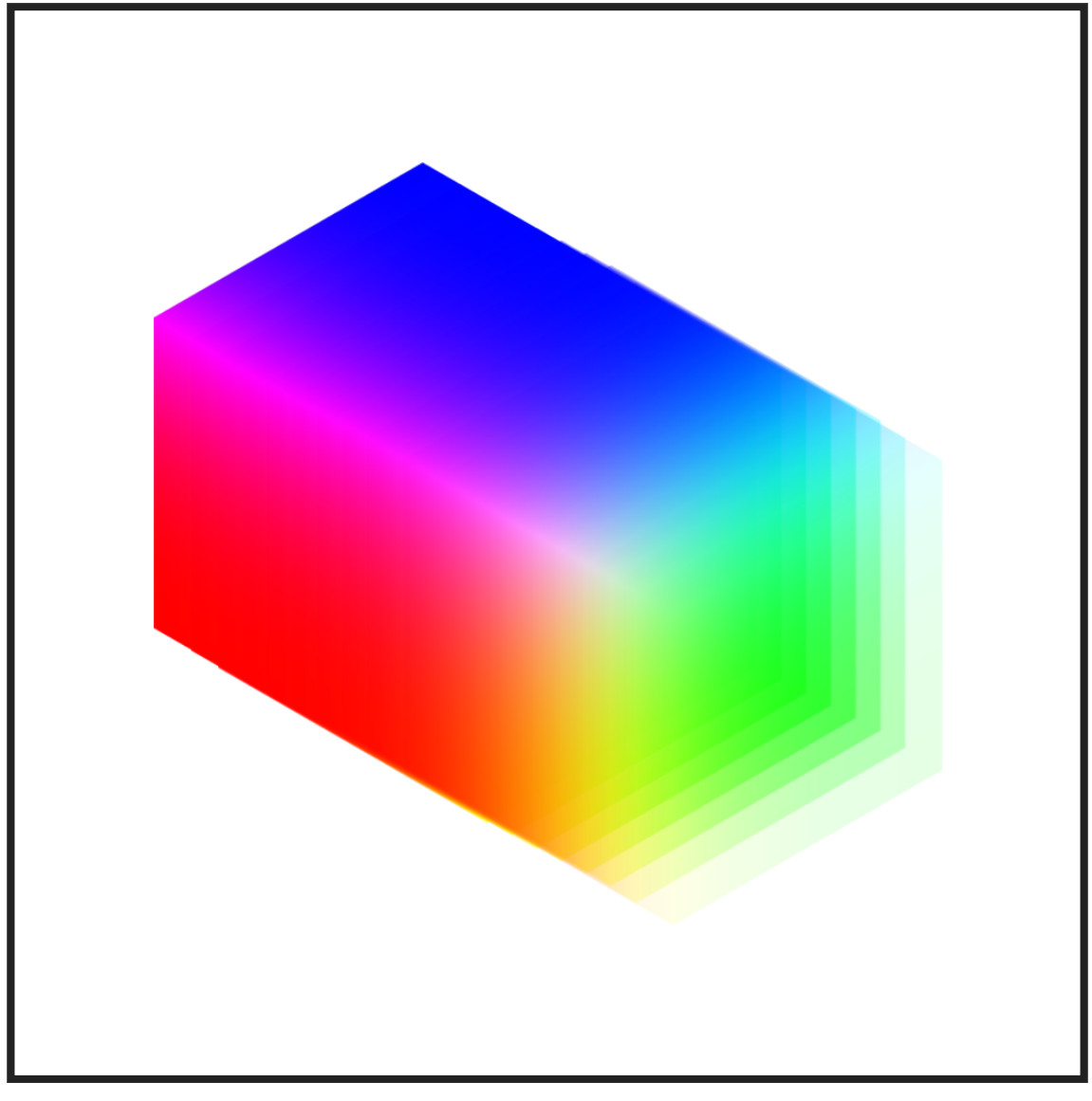
LINE IN A 4D SPACE



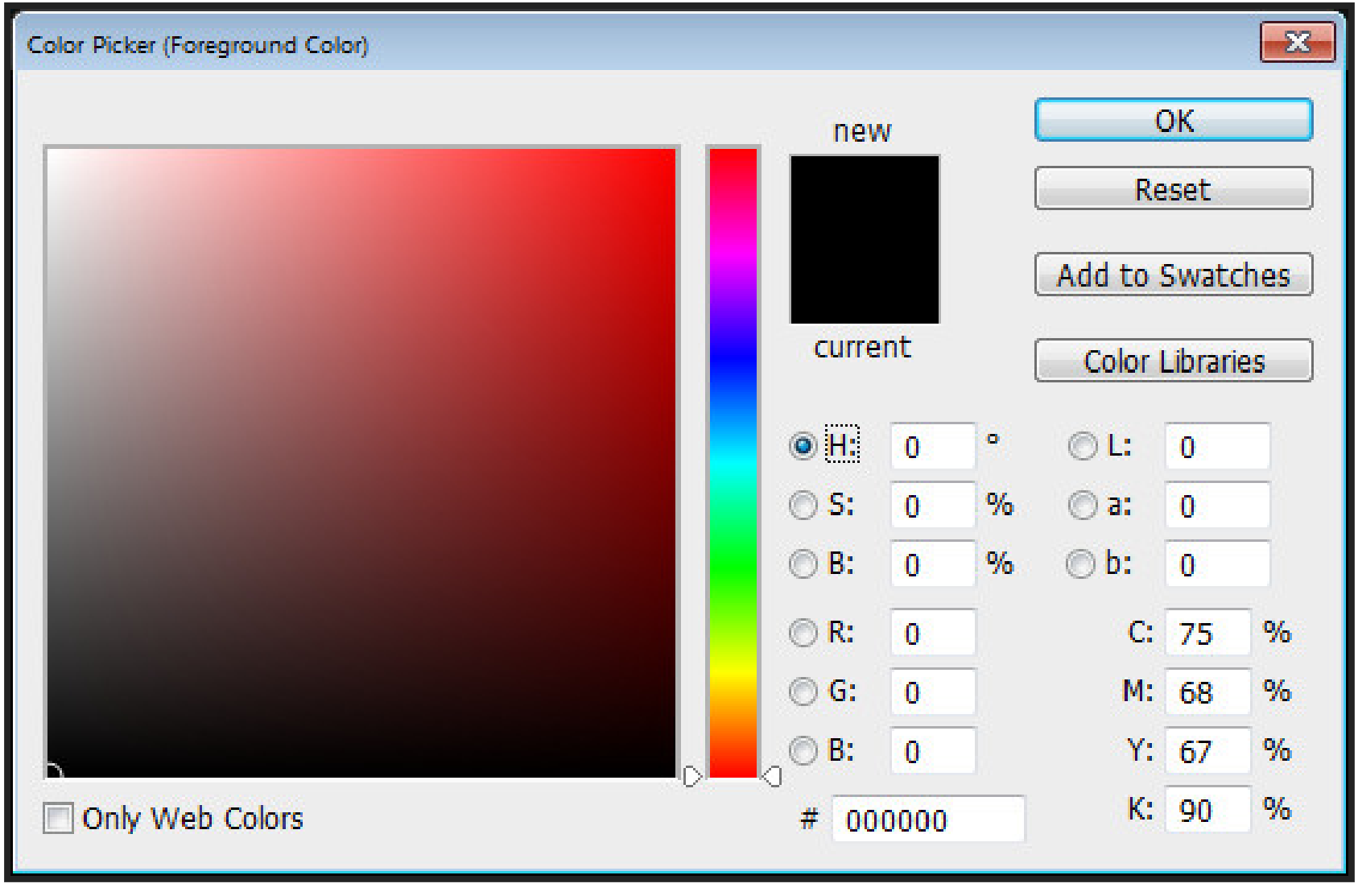
CUBE IN A 4D SPACE



RGBA (4D) COLOR SPACE

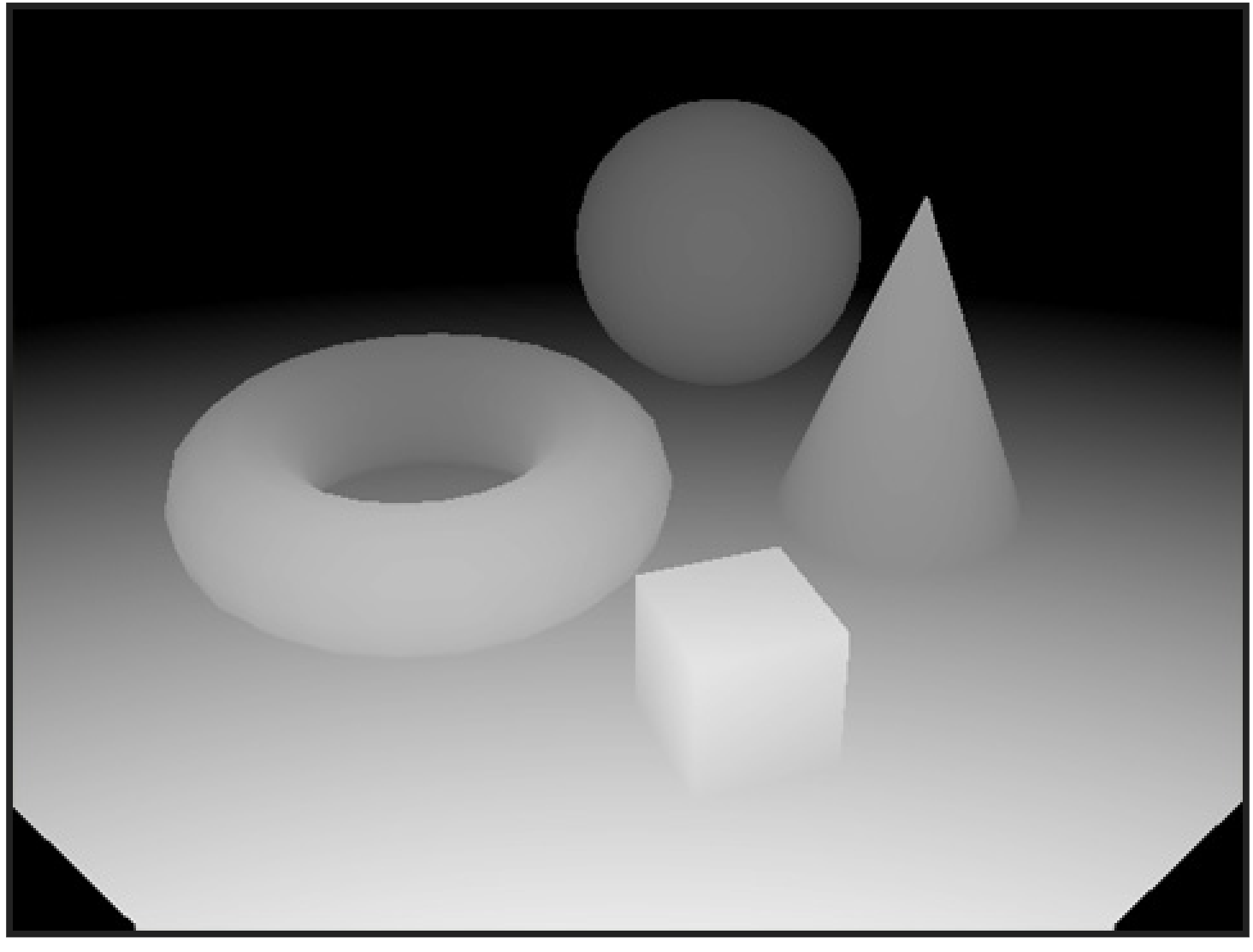


to visualise high dimensional space data, we'll often need to project them in a lower dimension and find a way to represent the parameters that "went missing" during the projection

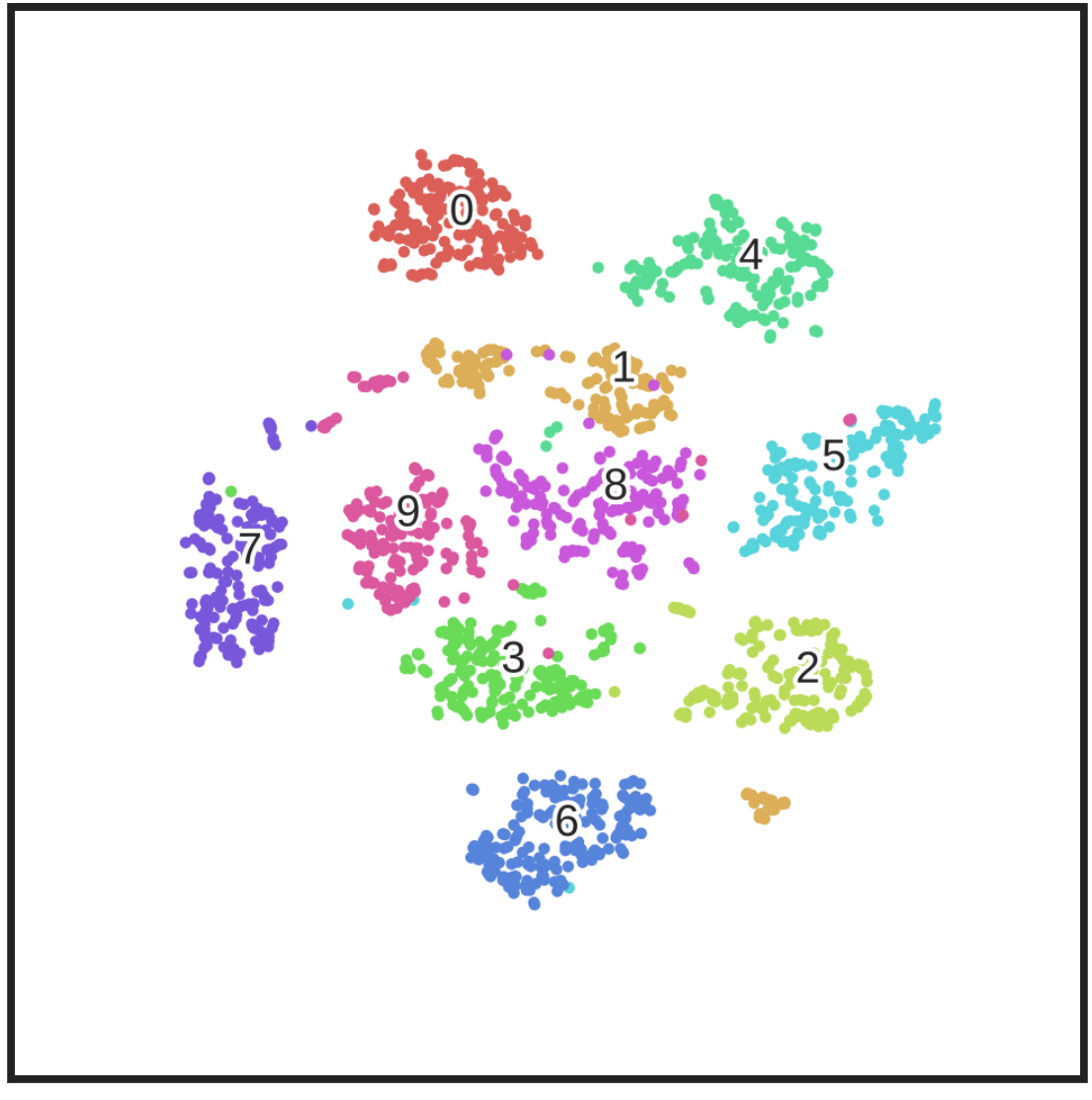


a color palette is a 2D projection of a 3D/4D object

here, the spectrum represents the "missing data"



in a 2D projection of a 3D mesh, we lose the *depth* data
we use a depth buffer to represent the "missing data"



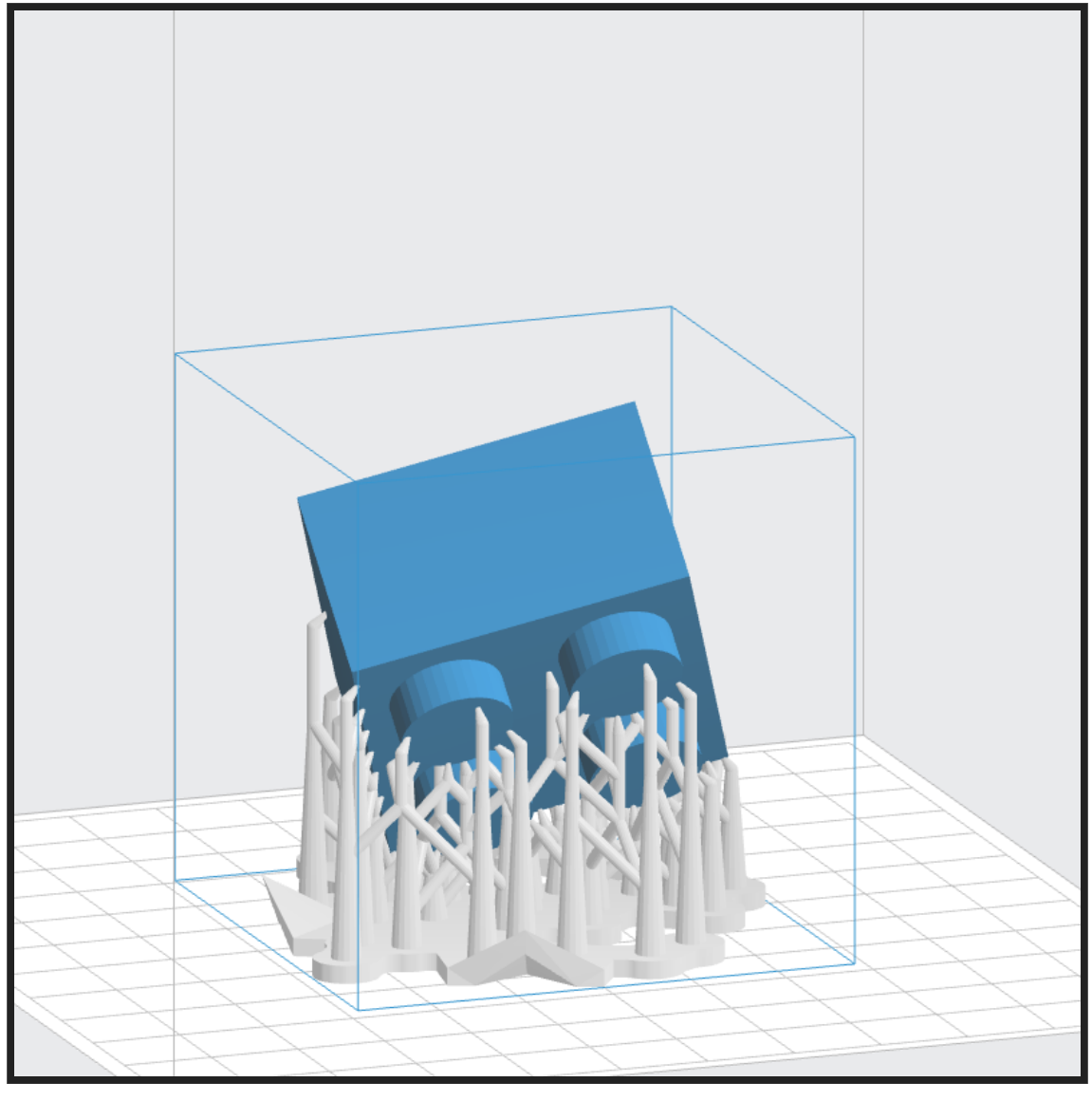
the t-SNE algorithm reduces data dimensionality and creates similarity clusters

making sense of high dimensional space data is hard

keep the parameter count as low as you can

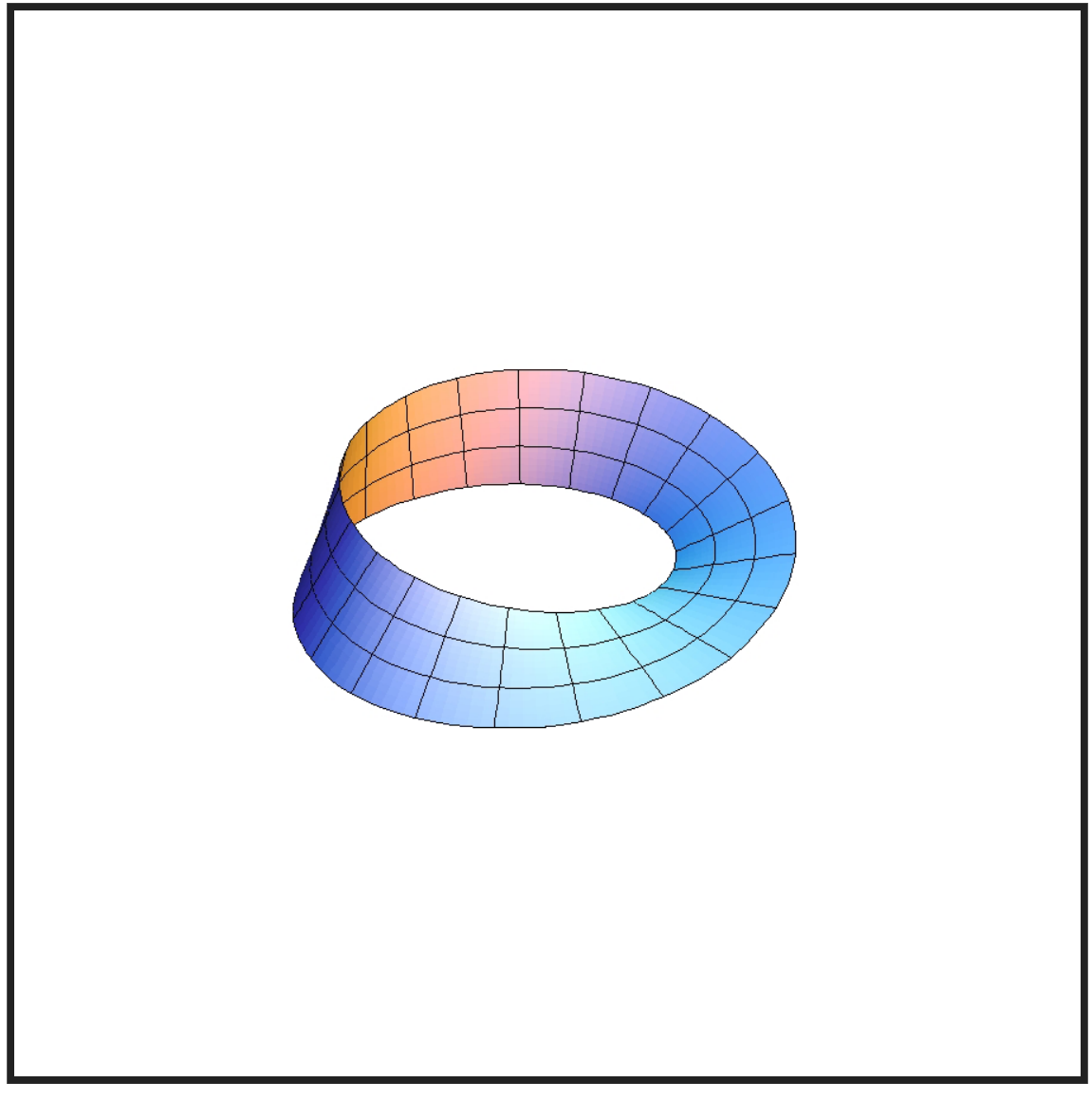
OTHER SPACES

PHYSICAL/TANGIBLE SPACE




SOUND SPACE
notes on blindness

NON EUCLIDEAN SPACE



3. PROCESS

DISTRIBUTION

DISTRIBUTION

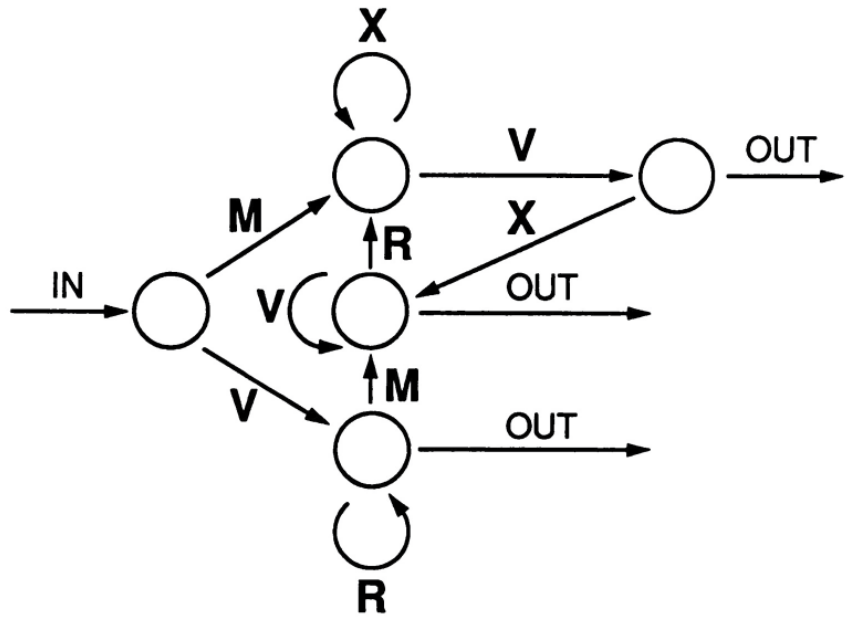
GRAMMAR

series of rules that defines all
valid sentences of a language

FORMAL GRAMMAR

a formal grammar is based on *production rules*, takes an *axiom* as an input and determines if it is valid

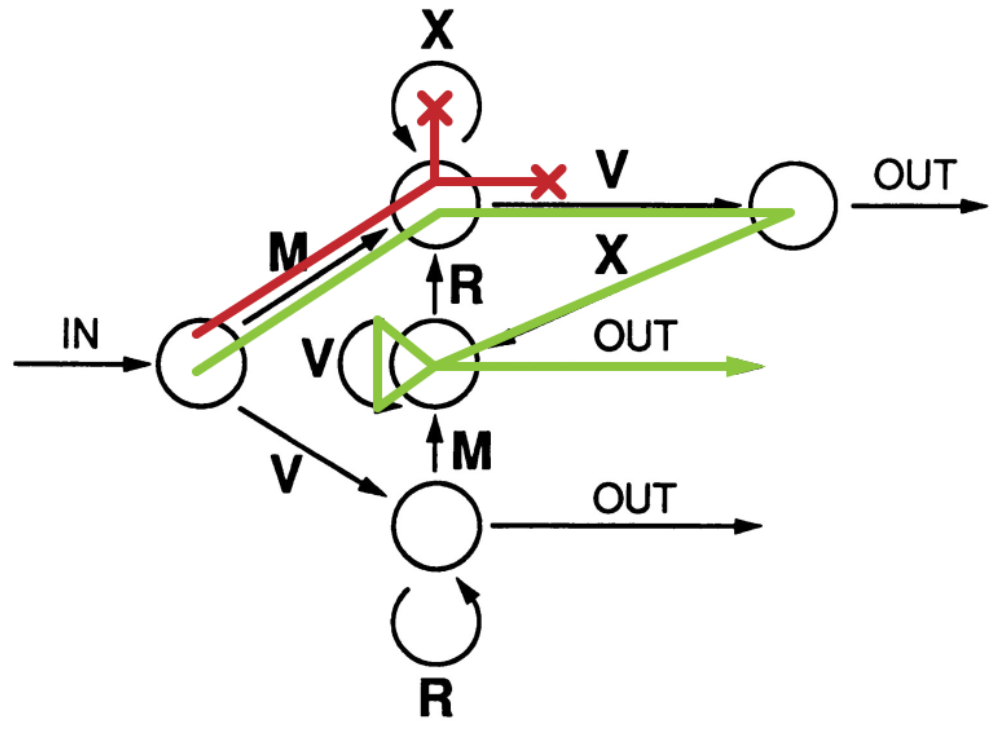
the next illustration is a "finite state machine"
it describes all the production rules relationships



Grammatical	Nongrammatical
MXV	VV
VMRV	MMX
MVXVV	MXR
VRRRM	XXXV

picture source

instead of writing a *dictionary* of all the valid *axioms*,
we use this FSM to ckeck if an *axiom* is valid



Grammatical	Nongrammatical
MXV	VV
VMRV	MMX
MVXVV	MXR
VRRRM	XXXV

L-SYSTEM

variation on formal grammar used to model plants' growth, it introduces the concepts of *rewriting, iteration, trees & recursion*

ruleset:

```
variables : A B
axiom  : A
rules  : (A → AB), (B → A)
```

production:

```
n=0:          A            start (axiom/initiator)
             / \
n=1:        A   B          (A→AB)
           /|    \
n=2:      A B     A         (A→AB), (B→A)
         /| |     |\
n=3:    A B A     A B       (A→AB), (B→A), (A→AB)
       /| | |\    |\ \
n=4: A B A A B   A B A     (A→AB), (B→A), (A→AB), (A→AB), (B→A)
```

if you don't see anything drawn, reload this page

length

angle

60

axiom

F

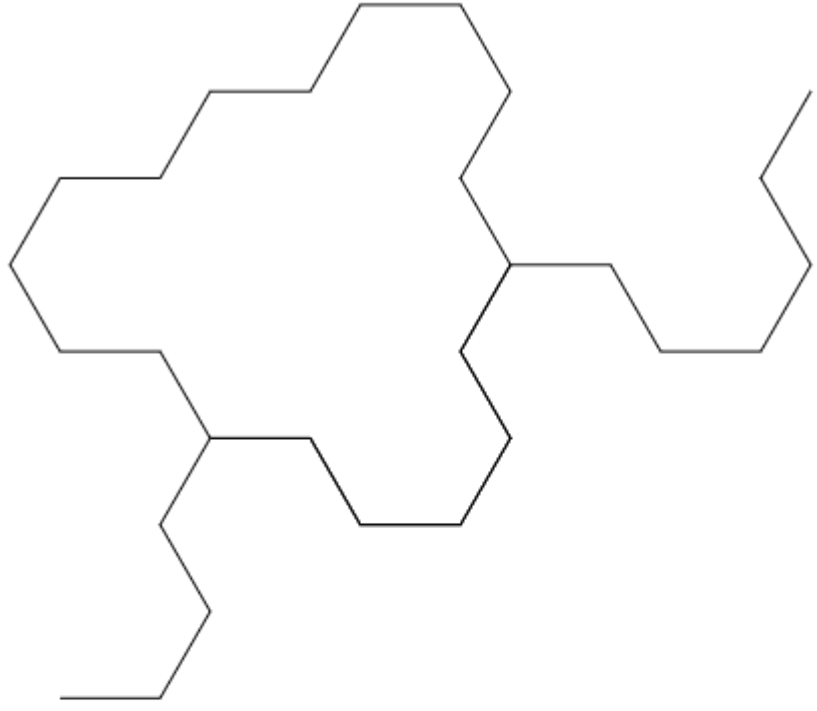
rule

f+f-f-f-f+f

generations

production

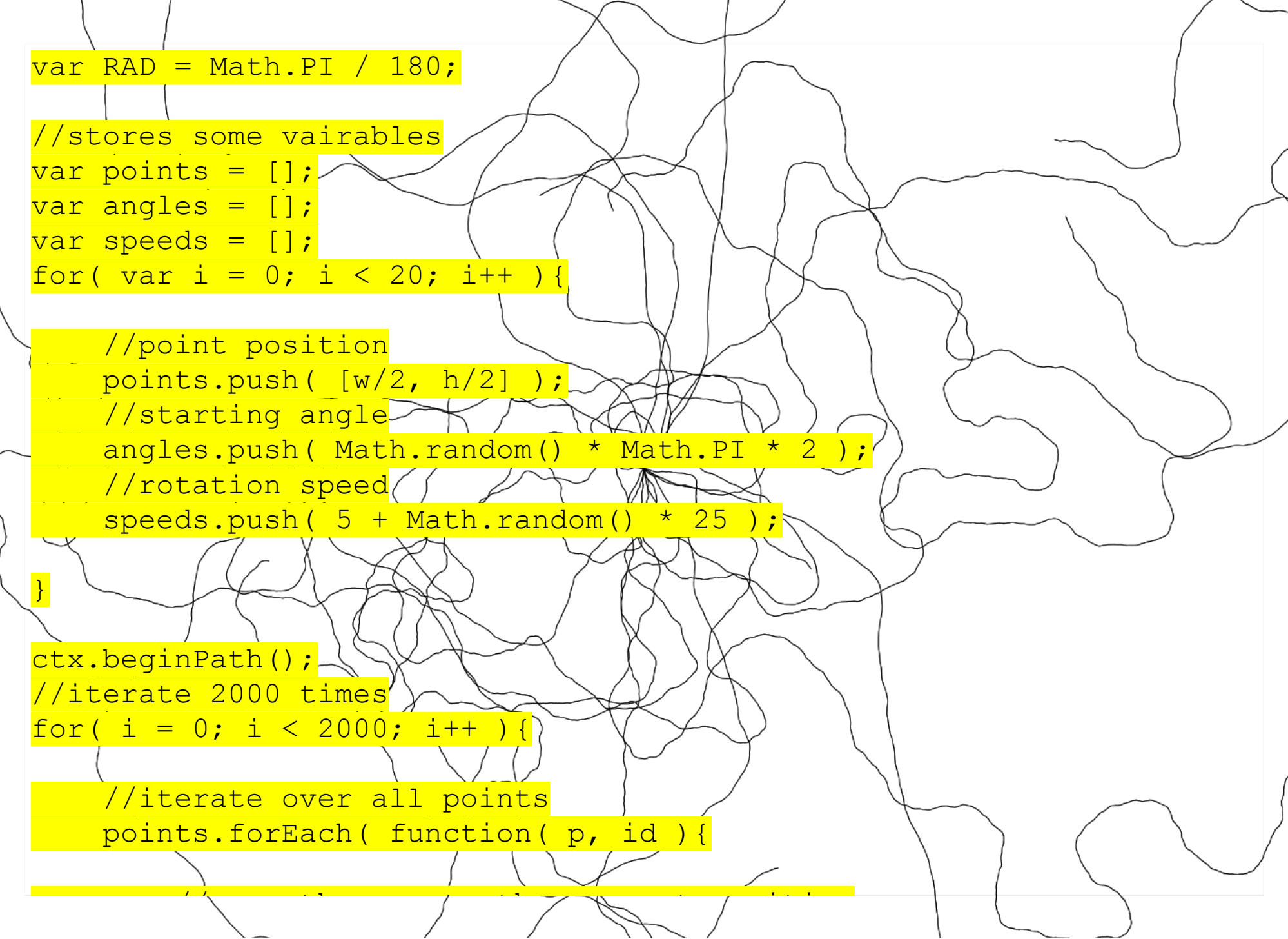
f+f-f-f-f+f+f+f-f-f-f+f-f+f-f-f-f+f-f+f-f-f-f+f-f+f-f-f-f+f+f+f+f-f-f-f+f



this is the essence of procedural generation:

describing the rules of production rather than describing the objects themeselves

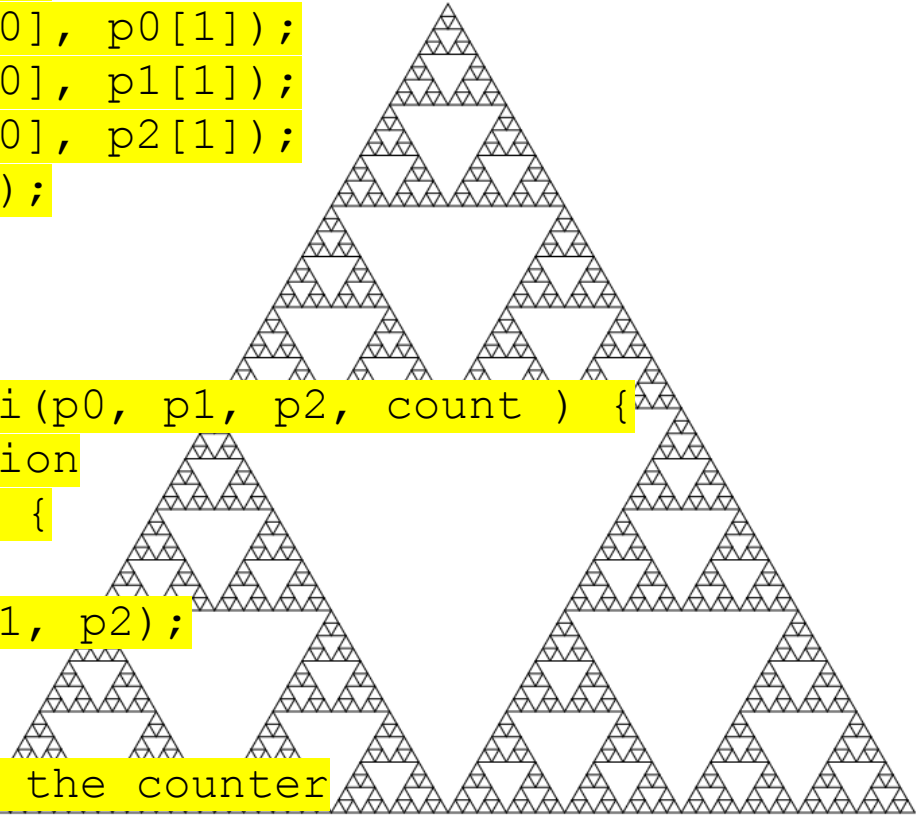
ITERATION



```
var RAD = Math.PI / 180;

//stores some vairables
var points = [];
var angles = [];
var speeds = [];
for( var i = 0; i < 20; i++ ){

    //point position
    points.push( [w/2, h/2] );
    //starting angle
    angles.push( Math.random() * Math.PI * 2 );
    //rotation speed
    speeds.push( 5 + Math.random() * 25 );

}

ctx.beginPath();
//iterate 2000 times
for( i = 0; i < 2000; i++ ){

    //iterate over all points
    points.forEach( function( p, id ){
```

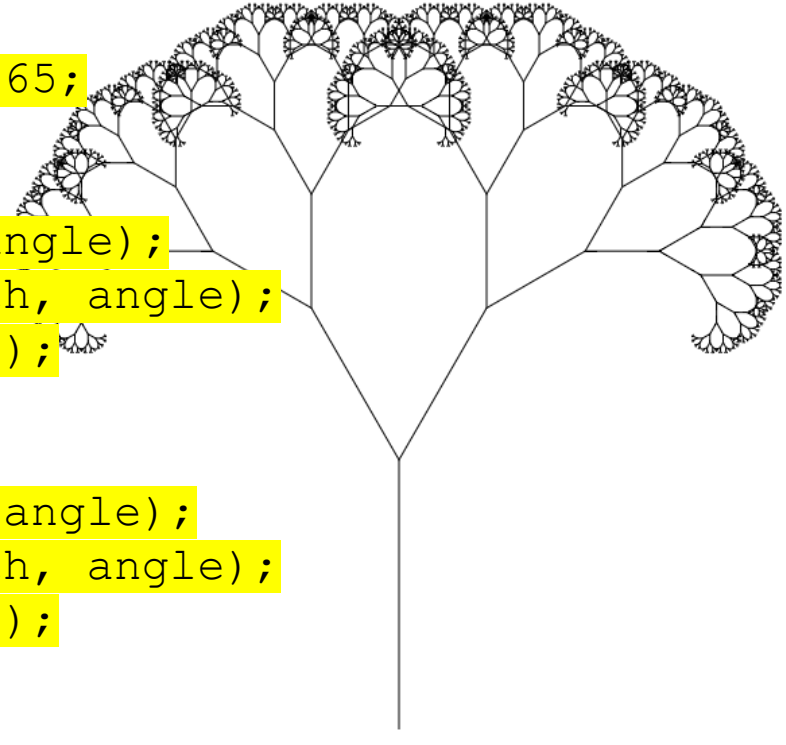
RECURSION

```
function draw(p0, p1, p2) {
    ctx.beginPath();
    ctx.moveTo(p0[0], p0[1]);
    ctx.lineTo(p1[0], p1[1]);
    ctx.lineTo(p2[0], p2[1]);
    ctx.closePath();
    ctx.stroke();
}

function sierpinski(p0, p1, p2, count ) {
    //break condition
    if(count <= 0) {
        //render
        draw(p0, p1, p2);
    }else{

        //decrease the counter
        count--;

        //find the edges' centers
        var a = [ ( p0[0] + p1[0] ) / 2, (p0[1] + p1[1]) / 2 ];
        var b = [ ( p1[0] + p2[0] ) / 2, (p1[1] + p2[1]) / 2 ];
        var c = [ ( p2[0] + p0[0] ) / 2, (p2[1] + p0[1]) / 2 ];
```



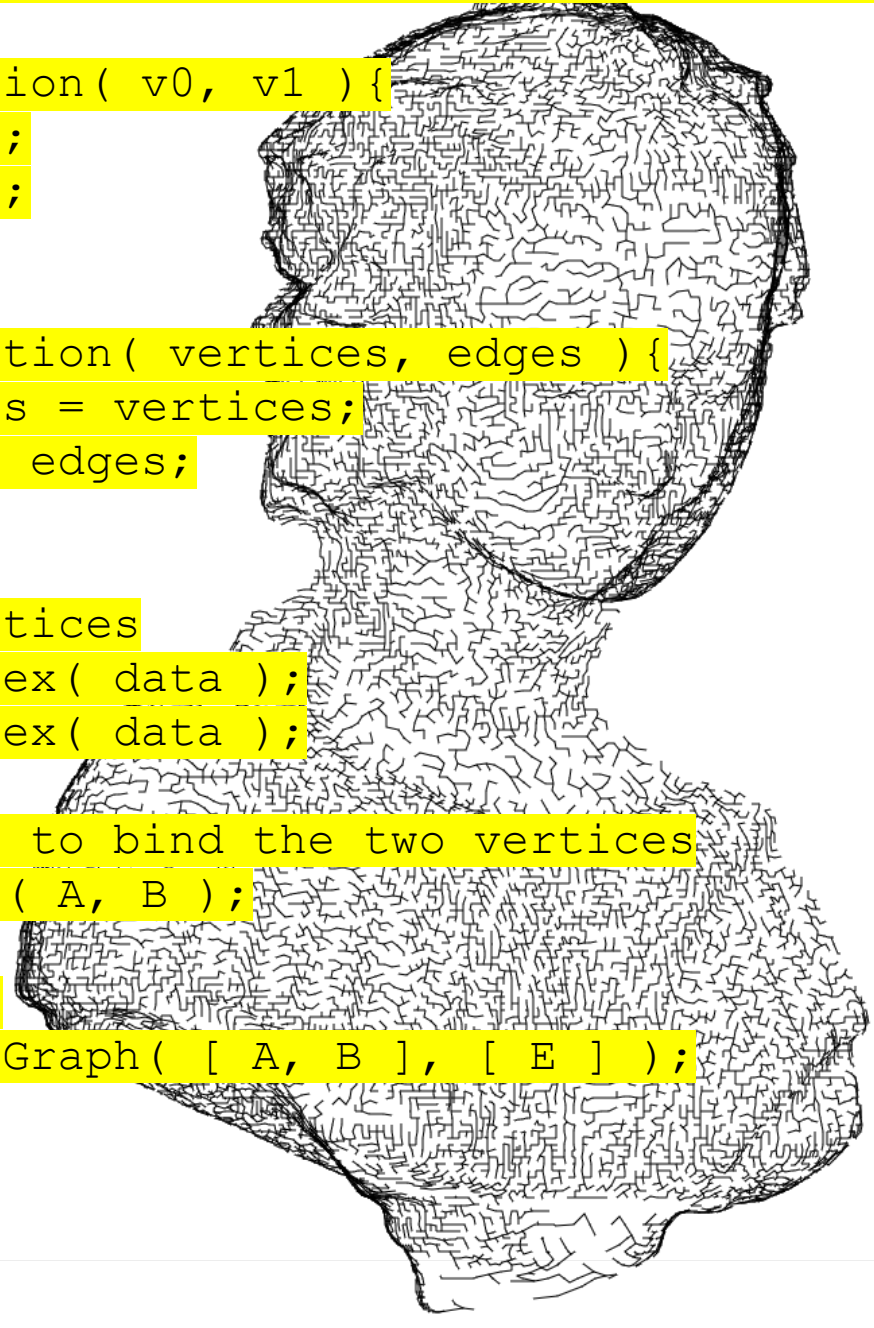
RECURSIVE TREE

```
function branch(length, angle) {

    line(0, 0, 0, -length);
    ctx.translate(0, -length);

    if (length > 2) {

        length *= 0.65;

        ctx.save();
        ctx.rotate(angle);
        branch(length, angle);
        ctx.restore();

        ctx.save();
        ctx.rotate(-angle);
        branch(length, angle);
        ctx.restore();
    }
}

function line(x0, y0, x1, y1) {
    ctx.beginPath();
    ctx.moveTo(x0, y0);
```



GRAPH

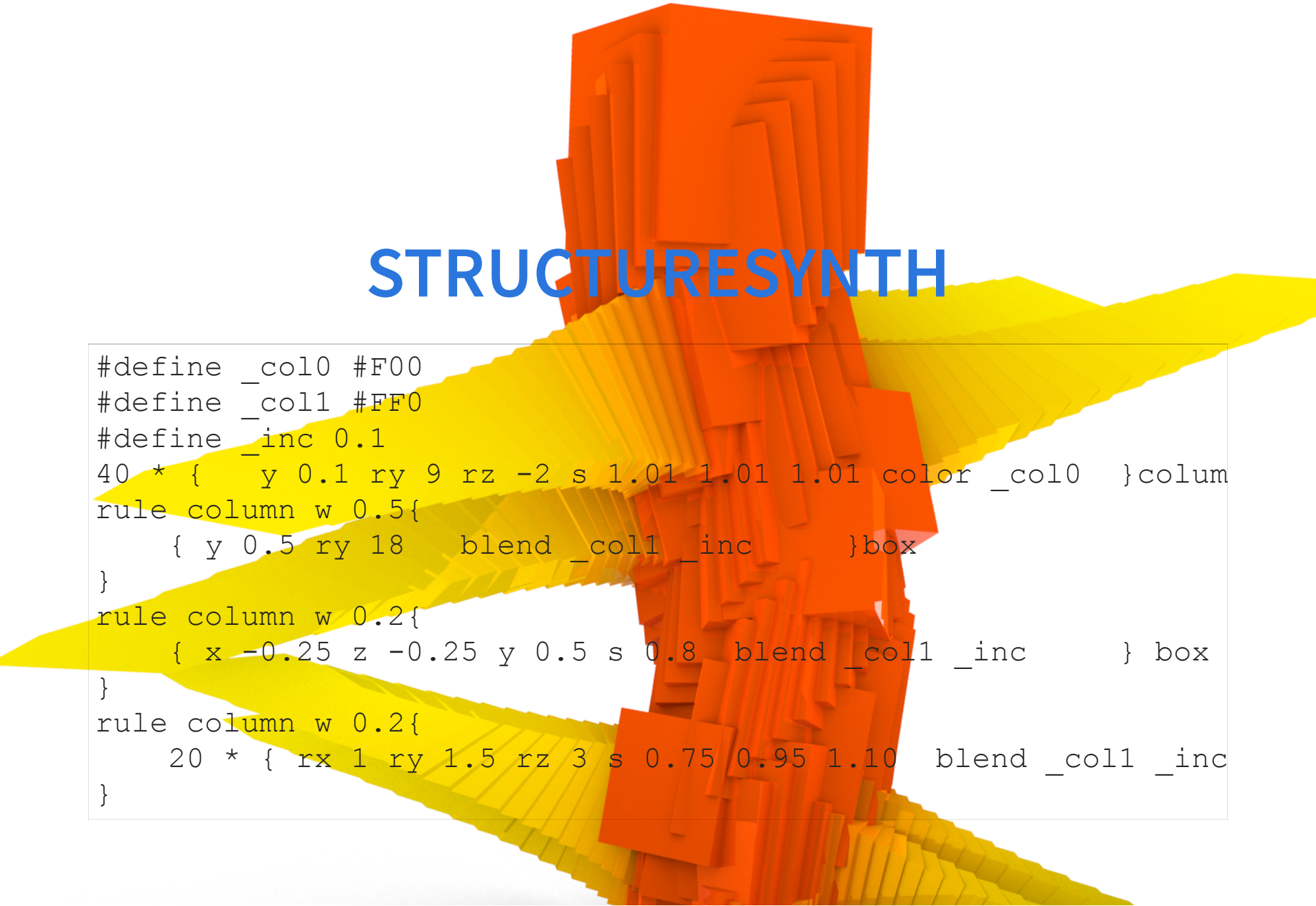
```
var Vertex = function( data ){ /*store the data*/ }

var Edge = function( v0, v1 ){
    this.v0 = v0;
    this.v1 = v1;
}

var Graph = function( vertices, edges ){
    this.vertices = vertices;
    this.edges = edges;
}

//create two vertices
var A = new Vertex( data );
var B = new Vertex( data );

//create an edge to bind the two vertices
var E = new Edge( A, B );

//create a graph
var graph = new Graph( [ A, B ], [ E ] );
```



CONTEXT FREE GRAMMAR

recursively applies *production rules* and stops when the *axiom* only contains terminal symbols (primitives)

STRUCTURESYNTH



```
#define _col0 #F00
#define _col1 #FF0
#define _inc 0.1
40 * {   y 0.1 ry 9 rz -2 s 1.01 1.01 1.01 color _col0  }colum
rule column w 0.5{
    { y 0.5 ry 18   blend _col1 _inc     }box
}
rule column w 0.2{
    { x -0.25 z -0.25 y 0.5 s 0.8  blend _col1 _inc     } box
}
rule column w 0.2{
    20 * { rx 1 ry 1.5 rz 3 s 0.75 0.95 1.10  blend _col1 _inc
}
```