

# VEX

- [Camera Stuff](#)
- [Points](#)
- [If then statements](#)
- [Transforms and Junk](#)
- [Orientation](#)
- [Spiral](#)
- [Links](#)
- [Comparing array](#)
- [Looping](#)

# Camera Stuff

auto focus, get distance from object and camera:

```
vlength(vtorigin("/obj/geo1", "/obj/cam1"))
```

# Points

divide points into 3 equal parts:

```
i@part = floor(fit(rand(@ptnum+.258), 0, 1, 0, 2.9));
```

Nage replacement

```
@nage = fit(@age,0,@life,0,1);
```

Group from class:

```
string grpname = sprintf("class_%d", i@class);  
setprimgroup(0, grpname, @primnum, 1); // use setpointgroup() if running over points
```

Connect close points:

```
float max_dist = chf("connect_distance");  
  
int npts = @numpt;  
for (int i = 0; i < npts; i++) {  
    vector pi = point(0, "P", i);  
    for (int j = i+1; j < npts; j++) {  
        vector pj = point(0, "P", j);  
        if (distance(pi, pj) < max_dist) {  
            int prim = addprim(0, "polyline");  
            addvertex(0, prim, i);  
            addvertex(0, prim, j);  
            setprimattrib(0, "create_frame", prim, @Frame);  
        }  
    }  
}
```

Shift 0 -1 to 0 - 1 - 0:

```
float mask = f@mask;
```

```
float midpoint = 0.5;

float result = 1.0 - abs(mask - midpoint) * 2.0;
result = clamp(result, 0.0, 1.0);

f@newmask = result;
```

## Confine points to sphere:

```
vector center = prim(1, "P", 0);
float radius = ch("scale");

vector dir = @P - center;
float dist = length(dir);

if (dist > radius) {
    @P = center + normalize(dir) * radius;
}
```

## Connect points with lines by dist:

```
float max_dist = chf("connect_distance");

// Loop through all points in Group A
int npts = @numpt;
for (int i = 0; i < npts; i++) {
    if (!inpointgroup(0, "groupA", i)) continue;

    vector pi = point(0, "P", i);

    // Loop through all points in Group B
    for (int j = 0; j < npts; j++) {
        if (!inpointgroup(0, "groupB", j)) continue;

        // Avoid self-connection in case of overlapping group membership
        if (i == j) continue;

        vector pj = point(0, "P", j);
```

```
if (distance(pi, pj) < max_dist) {  
    int prim = addprim(0, "polyline");  
    addvertex(0, prim, i);  
    addvertex(0, prim, j);  
    @primid = @primnum;  
  
    }  
}  
}
```

# If then statements

If the pscale is greater than .4 then set it to .2, if not set it to its current pscale

```
@pscale = @pscale>.4?.2:@pscale
```

# Transforms and Junk

## 1. transforms to attribute matrix:

```
p@orient = quaternion(3@transform);  
v@scale = cracktransform(0,0,2,set(0.0.0). 3@transform);
```

## 2. rotate packed fracture based on point + distance:

Screenshot from 2023-06-21 11-48-58.png

```
vector p1= set(@P.x, @P.y, @P.z);  
  
vector crack1 = point(1, "P", 0);  
vector crack2 = point(2, "P", 0);  
vector p2 = crack1-p1;  
vector p3 = crack2-p1;  
float n = fit ( length ( p2 ), 0, ch("maxdist"), ch('mult'), 0 );  
float n2 = fit ( length ( p3 ), 0, ch("maxdist2"), ch('mult2'), 0 );  
  
vector4 q0 = quaternion ( 0 );  
vector4 q1 = sample_orientation_uniform ( rand ( @ptnum ) );  
vector4 q2 = slerp ( q0, q1, n+n2 );  
matrix3 xform = qconvert ( q2 );  
  
setprimintrinsic ( 0, "transform", @ptnum, xform );
```

## 3. Blending spiral (end beg):

Screenshot from 2023-06-21 15-48-58.png

```
vector target = point(1, "P", @ptnum);  
float blend = chramp("blendAlongSpiral", @curveu)*chf("multiplier");  
  
@P = lerp(@P, target, blend);
```

## 4. Position copy via uv:

Screenshot from 2023-06-21 15-51-53.png

```
v@P = uvsample(1, "P", "uv", @P);
```

## 5. move near points together:

```
int near = nearpoint(1, @P);  
vector target = point(1, "P", near);  
@P = target;
```

## 6. Affect the scale of packed prims:

```
//vector scale = fit01(vector(rand(@primnum)), 0,1.46) *@growth;  
vector scale = ch("scale");  
  
matrix3 trn = primintrinsic(0, "transform", @primnum);  
matrix scalem = maketransform(0, 0, {0,0,0}, {0,0,0}, scale, @P);  
trn *= matrix3(scalem);  
setprimintrinsic(0, "transform", @primnum, trn);
```



# Orientation

Get transform and orientation from camera:

```
string camera = "/obj/alembicarchive1/Camera2/CameraShape2"; // path to your camera
@P = ptransform(camera, "space:current", {0,0,0});
@N = ntransform(camera, "space:current", {0,0,-1});
```

Random orient on points:

```
float seed = float(@ptnum);
vector4 orient = quaternion(radians(rand(seed) * ch("add")), normalize(rand(seed + 1)));
@orient = orient;
```

Point normals at camera:

```
string cam = chs("cam");
matrix cam_xform = optransform(cam);
vector dirtocam = cracktransform(0,0,0,{0,0,0}, cam_xform);
@N = normalize(dirtocam - @P);
```

# Spiral

```
#include "math.h"
#include "voplib.h"

float easeOutCirc ( float t )
{
    return sqrt ( 1 - ( pow ( ( 1 - t ), 2 ) ) );
}

float index = @ptnum;
float numpts = @numpt;
float startAngle = radians ( ch("angle") );
float dir = 2 * ch("dir") - 1;
float steps = ( numpts - 1 ) / ch("turns");
float stepAngle = ( 2 * PI / steps ) * dir;

float inc = index / ( numpts - 1 );
int mirror = chi("spherical");
float linear = ( 1 + mirror ) * inc;
if ( mirror && index + 1 > numpts / 2 )
    linear = ( 1 + mirror ) * ( 1 - inc );

float circ = easeOutCirc ( linear );
float interp = linear + ( circ - linear ) * ch("roundness");
float r = ( ch("rx") + interp * ( ch("ry") - ch("rx") ) );

// Apply power to radius at the end (after curvature)
inc = ( ( numpts - 1 ) - index ) / ( numpts - 1 );
float theta = 2 * PI * inc;
if ( mirror && index + 1 > numpts / 2 )
    theta = 2 * PI * ( 1 - inc );
r *= pow ( ch("falloff"), theta );

float angle = index * stepAngle + startAngle;
float x = sin ( angle ) * r;
float z = cos ( angle ) * r;
```

```
float h = index / ( numpts - 1 );  
float y = vop_bias ( h, 0.5 * ch("bias") + 0.5 );  
y = vop_gain ( y, 0.5 * ch("gain") + 0.5 ) * ch("height");  
  
matrix3 xform = dihedral ( { 0, 1, 0 }, { 0, 0, -1 } ) * lookat ( 0, normalize ( chv("n") ) );  
@P = ch("scale") * set ( x, y, z ) * xform + chv("t");
```

# Links

Big resource:

<https://lex.ikoon.cz/vex-snippets/>

# Comparing array

Find the difference between input 1's ids and input 0's:

```
int delete_ids[] = array();

int numPrims = nprimitives(1);
for (int i = 0; i < numPrims; i++) {
    int leaf_id = prim(1, "leafid", i);
    append(delete_ids, leaf_id);
}

int my_id = i@leafid;

if (find(delete_ids, my_id) >= 0) {
    @group_keep=1;
}
```

# Looping

## Loop moving points on curve using CurveU:

```
int loop_frames = chi("loop_frame");
float fps = 29.97;
float loop_time = loop_frames / fps;

float t = @Time / loop_time;
t -= floor(t);

float ping = abs(2 * t - 1);
float speed_mult = chf("speed_mult");
ping *= speed_mult;

float exponent = chf("ease_exponent");
float slow_ping = pow(ping, exponent);

float u = f@curveu + slow_ping;
u -= floor(u);

int prim = i@class;
vector pos = primuv(1, "P", prim, set(u, 0, 0));
@P = pos;
```

Add a resample node with curveu, also to control amount of points

## Loop points in Y:

```
int loop_frames = chi("loop_frames");
float fps = 29.97;
float loop_time = loop_frames / fps;
```

```
float t = @Time / loop_time;
t -= floor(t);

float offset = frac(t + rand(@ptnum)); // unique phase per point

float min_y = chf("min_y");
float max_y = chf("max_y");
float range = max_y - min_y;

@P.y = min_y + offset * range;
```

## Loop on CurveU (nonPingPong):

1st attribute wrangle set to points after the point scatter:

```
vector uvw;
int prim;
float dist = xyzdist(1, @P, prim, uvw);
f@curveu = uvw.x;
i@class = prim;
```

2nd attrib wrangle set to points one the 1st input as the points and the 2nd to the curve with curevu from resample:

```
int loop_frames = chi("loop_frames");
float fps = 29.97;
float loop_time = loop_frames / fps;
//get time
float t = @Time / loop_time;
t -= floor(t);

float speed = chf("speed");
```

```
// offset
float base_u = f@curveu;
float offset_u = t * speed;

float u = base_u + offset_u;
u -= floor(u); // wrap it around

int prim = i@class;
vector pos = primuv(1, "P", prim, set(u, 0, 0));
@P = pos;
```