Codice dei Brush Shading Language

Questo documento riporta il codice di tutti (55) gli shader predefiniti in BlackInk 0.357.

Contents

Nothing

airbrush wisp

```
cfg{
  name="airbrush wisp";
  renderingTime = 20 ;
}

perPrim {
  float hardness = 0.5 ;
  {
    id = -1 ;
    uiMin = 0      ;
    uiMax = 1      ;
    uiTab = "shape"  ;
    uiFormat = percent  ;
    uiName = "hardness"  ;
  }
  float symetry = 0 ;
  {
    id = -1 ;
    uiMin = 0      ;
    uiMax = 1      ;
    uiTab = "shape"  ;
    uiFormat = percent  ;
    uiName = "symetry"  ;
  }
  float noisePower=0.1;
  {
    uiMax = 1 ;
    uiName = "Power";
    uiTab = "Dithering" ;
    uiFormat = percent ;
  }
}

float cptPrimAtt( idatas i, float2 dn, float attp, float sym )
{
  matrix2 tr = i.primBox.getToCenterTransfo();
  float2 p = tr.transform( i.pos+dn );
  float2 s = i.primBox.size ;
  float symf = lerp( 200, 1, pow(sym,0.01) );
  float y = (p.y < 0 ? -p.y*symf : p.y) / (s.y*0.5) ;
  float x = saturate( (abs(p.x)-s.x*0.05 ) / (s.x*0.5) ) ;

  return (1-x)*(1-pow(y,attp*0.5));
}

float2 hash( float3 p )
{
  float3 hscale = float3(.1031,-.1029,.1032) ;

  float3 p3 = frac( p * hscale ) ;
  p3 += dot(p3, p3.yzx + 209.191349);
  return frac( float2( (p3.x + p3.y) * p3.z, (p3.x + p3.z) * p3.y ) ) ;
}

// simple Value Noise
// the returned value is between [-1,1]
float2 noise( float3 x )
{
  float3 f = frac(x);
  float3 p = decompose( x, f );
  float3 u = f*f*(3.0-2.0*f);
  float2 v1 = bilinearLerp(  hash(p+float3(0,0,0)), hash(p+float3(1,0,0)),
              hash(p+float3(0,1,0)), hash(p+float3(1,1,0)), u.xy ) ;
  float2 v2 = bilinearLerp(  hash(p+float3(0,0,1)), hash(p+float3(1,0,1)),
              hash(p+float3(0,1,1)), hash(p+float3(1,1,1)), u.xy ) ;

  return 2*lerp( v1, v2, u.z )-1;
}

float4 main( idatas i )
{
  // compute dithering noise
  float2 dn = i.noisePower * 10 * noise( float3( i.pos,i.nbUserStroke+i.dist*0.01)
);
  float d = cptPrimAtt( i, dn, i.hardness, i.symetry );
  //d *= dn*0.5+0.5;
  float alpha = saturate(i.color.a*d) ;
  float3 col = i.color.xyz ;

  return float4( col, alpha ) ;
}
```

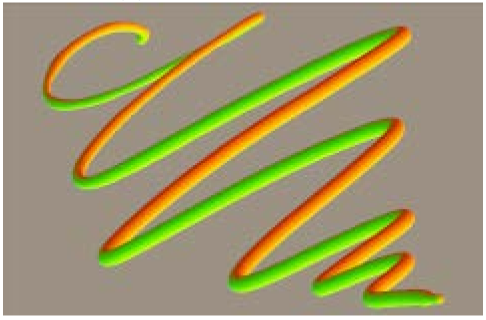
blur

```
cfg{
  name = "blur" ;
  blendEx = true ;
  blendDefault = replace ;
  samplingLayerMaxOffset = 7 ;
  renderingTime = 60 ;
}

perPrim{
  float blurStrength = 1 ;
  {
    uiFormat = percent ;
    uiMax = 1 ;
    uiTab = "Blur" ;
    uiName = "Strength"  ;
  }

  float alphaHardness = 1 ;
  {
    uiName = "Hardness";
    uiMax = 1 ;
    uiFormat = percent ;
    uiTab = "Shape" ;
  }

  float overlayFactor = 0.1 ;
  {
    uiMax = 1          ;
    uiTab = "Color"        ;
    uiFormat = percent      ;
    uiName = "Color overlay"  ;
  }

}

float hardnessCpt( idatas i, float h )
{
  float sm = vmin( i.primBox.getSize() )  ;
  float d = max( -i.primDistance, 0 )    ;
  float softness = pow( 1 - h,2)  ;
  d /= sm*softness + 0.001    ;
  d = 1 - exp( -d );

  return i.primDistanceValid ? d : 1 ;
}

float4 blurBottom( idatas i, float s )
{
  int nbsample = 5 ;
  float4 noBlur = bottomLayer.pointSample( i, i.pos );
  float maxdec = nbsample ;
  float radius = pow( s, 2 ) * fromDrawSpace( maxdec )  ;
  radius = min( radius, maxdec ) ;
  float tot = 0 ;
  float4 coltot = 0 ;
  for( int y=0; y<nbsample; y++ )
    for( int x = 0; x<nbsample; x++ )
    {
      float2 d2 = 2*( ( (float2( x, y ) + 0.5) / nbsample ) - 0.5 )  ;
      float d = length( d2 ) ;
      float att = exp( -d );
      tot += att ;
      coltot += att * bottomLayer.bilinearSample( i, i.pos + d2*radius );
    }
  coltot /= tot ;

  return smoothLerp( noBlur, coltot, saturate(radius) );
}

float4 main( idatas i )
{
  float4 noBlur = bottomLayer.pointSample( i, i.pos );
  float h = hardnessCpt( i, i.alphaHardness ) ;
  float bs = i.blurStrength * h  ;
  float4 blurred = blurBottom( i, bs );
  float4 col = blurred ;
  // apply a color overlay
  float4 tmp = blendOverlay( blurred,
i.color*float4(1,1,1,i.overlayFactor*pow(h,1.95)) );
  col.xyz = tmp.xyz ;
  float alpha = i.color.a ;
  col = blendNormalAlpha( noBlur, col, alpha );
  // take care of the eraser state
  col = i.eraser ? float4( noBlur.xyz, noBlur.w * (1-alpha) ) : col ;

  return col ;
}
```

blur Noise

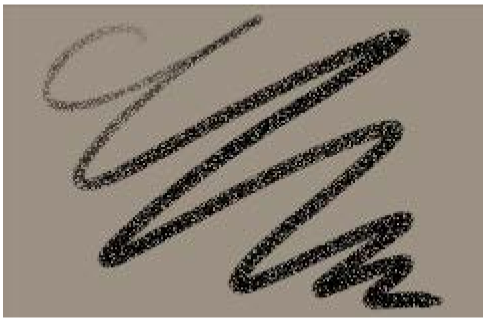
```
cfg{
  name = "blurNoise" ;
  blendEx = true ;
  blendDefault = replace ;
  samplingLayerMaxOffset = 7 ;
  renderingTime = 60 ;
}

globals{
  float noiseScale = 1 ;
  {
    uiTab = "Noise" ;
    uiName = "Scale"  ;
    uiFormat = percent ;
    uiMax = 1 ;
  }
}

perPrim{
  float blurStrength = 1 ;
  {
    uiFormat = percent ;
    uiMax = 1 ;
    uiTab = "Blur" ;
    uiName = "Strength"  ;
  }

  float alphaHardness = 1 ;
  {
    uiName = "Hardness";
    uiMax = 1 ;
    uiFormat = percent ;
    uiTab = "Shape" ;
  }

  float overlayFactor = 0.1 ;
  {
    uiMax = 1          ;
    uiTab = "blur"       ;
    uiFormat = percent      ;
    uiName = "color overlay"  ;
  }

  float ndensity = 0.5 ;
  {
    uiTab = "Noise" ;
    uiName = "Density"  ;
    uiFormat = percent ;
    uiMax = 1 ;
  }

  float densityHardness = 0.15 ;
  {
    uiName = "Hardness";
    uiMax = 1 ;
    uiTab = "Noise" ;
  }
}

float hardnessCpt( idatas i, float h )
{
  float sm = vmin( i.primBox.getSize() )  ;
  float d = max( -i.primDistance, 0 )    ;
  float softness = pow( 1 - h,2)  ;
  d /= sm*softness + 0.001    ;
  d = 1 - exp( -d );

  return i.primDistanceValid ? d : 1 ;
}

float hash( float3 p )
{
  float hscale = .1031 ;
  float3 p3 = frac( p * hscale ) ;
  p3 += dot(p3, p3.yzx + 19.19);

  return frac( (p3.x + p3.y) * p3.z );
}

float adjustDensity( float r, float d )
{
  d = 1-d ;
  float a = clamp( (d-0.5)*2, 0, 1 );
  float b = clamp( d*2, 0, 1 );
  r = (r-a) / max( b-a, 0.001) ;
  r = saturate( r );

  return r ;
}

float hashD( float3 p, float d )
{
  return adjustDensity( hash(p), d );
}

float cptPrimNoise( idatas i, float time, float density )
{
  float2 p = i.primBox.toUpperLeftNorm( i.pos )*2 - 1 ;
  p = floor( p*i.primBox.getSize()*noiseScale );

  return hashD( float3( p, time ), density );
}

float4 blurBottom( idatas i, float s )
{
  int nbsample = 5 ;

  float4 noBlur = bottomLayer.pointSample( i, i.pos );
  float maxdec = nbsample ;
  float radius = pow( s, 2 ) * fromDrawSpace( maxdec )  ;
  radius = min( radius, maxdec ) ;

  float tot = 0 ;
  float4 coltot = 0 ;
  for( int y=0; y<nbsample; y++ )
    for( int x = 0; x<nbsample; x++ )
    {
      float2 d2 = 2*( ( (float2( x, y ) + 0.5) / nbsample ) - 0.5 )  ;
      float d = length( d2 ) ;
      float att = exp( -d );
      tot += att ;
      coltot += att * bottomLayer.bilinearSample( i, i.pos + d2*radius );
    }

  coltot /= tot ;

  return smoothLerp( noBlur, coltot, saturate(radius) );
}

float4 main( idatas i )
{
  float4 noBlur = bottomLayer.pointSample( i, i.pos );

  float density = i.ndensity * hardnessCpt( i, i.densityHardness );
  float n = cptPrimNoise( i, i.dist*0.001, density );
  float nover = cptPrimNoise( i, i.dist*0.0135, density );

  float h = hardnessCpt( i, i.alphaHardness ) ;
  float bs = i.blurStrength * h * n  ;
  float4 blurred = blurBottom( i, bs );

  float4 col = blurred ;

  // apply a color overlay
  float4 tmp = blendOverlay( blurred,
i.color*float4(1,1,1,nover*i.overlayFactor*pow(h,1.95)) );
  col.xyz = tmp.xyz ;

  float alpha = i.color.a ;

  col = blendNormalAlpha( noBlur, col, alpha );

  return col ;
}
```

box2.toUpperLeftNorm



```
cfg{name="box2.toUpperLeftNorm";}

float4 main( idatas i )
{
  float2 p = i.primBox.toUpperLeftNorm( i.pos ) ;
  return float4( p.x, p.y, 0, 1 ) ;
}
```

Canvas noise



```
cfg{
  name = "canvas noise";
  renderingTime = 5 ;
}

globals{

  float noiseScale = 1 ;
  {
    uiTab = "Noise" ;
    uiName = "Scale"  ;
    uiFormat = percent ;
    uiMax = 1 ;
  }

}

perPrim{

  float ndensity = 0.5 ;
  {
    uiTab = "Noise" ;
    uiName = "Density"  ;
    uiFormat = percent ;
    uiMax = 1 ;
  }

}

float hash( float3 p )
{
  float hscale = .1031 ;

  float3 p3 = frac( p * hscale ) ;
    p3 += dot(p3, p3.yzx + 209.191349);
    return frac( (p3.x + p3.y) * p3.z );
}

float adjustDensity( float r, float d )
{
  d = 1-d ;
  float a = clamp( (d-0.5)*2, 0, 1 );
  float b = clamp( d*2, 0, 1 );

  r = (r-a) / max( b-a, 0.001) ;
  r = saturate( r );

  return r ;
}

float hashD( float3 p, float d )
{
  return adjustDensity( hash(p), d );
}

float4 main( idatas i )
{
  float density = i.ndensity ;
  float2 p = floor( i.pos * noiseScale );
  float alpha = hashD( float3(p,i.nbUserStroke), density );
  alpha = (density>=1) ? 1 : alpha ;

  float4 col = i.color ;
  col.a *= alpha ;

  return col ;
}
```

canvasSize

```
cfg{name="canvasSize";}

float4 main( idatas i )
{
  float2 p = saturate( i.pos / canvasSize );
  return float4( p.x, p.y, 0, 1 ) ;
}
```

Cell Voronoi distruct gradient

```
cfg{
  name="Cell Voronoi destruct gradient";
  renderingTime = 30 ;
}

globals{
  colorGradient grad ;
  {
    uiTab = "color" ;
    id = 1 ;
  }
}

float hash1( float2 p)
{
  float hscale = .1031 ;
  float3 p3 = frac( float3(p.x,p.y,p.x) * hscale ) ;
    p3 += dot(p3, p3.yzx + 19.19);
    return frac( (p3.x + p3.y) * p3.z );
}

float2 hash2( float2 p )
{
  float3 hscale3 = float3( .1031, .1030, .0973 ) ;

  float3 p3 = frac( float3(p.x,p.y,p.x) * hscale3) ;
    p3 += dot(p3, p3.yzx+19.19) ;
    return frac( float2( (p3.x + p3.y)*p3.z, (p3.x+p3.z)*p3.y) );
}

float4 hash4( float2 p )
{
  float4 hscale4 = float4( .1031, .1030, .0973, .1099 ) ;

  float4 p4 = frac( float4(p.xyxy) * hscale4 );
  p4 += dot( p4, p4.wzxy+19.19) ;
  return frac( float4((p4.x + p4.y)*p4.z, (p4.x + p4.z)*p4.y, (p4.y + p4.z)*p4.w,
(p4.z + p4.w)*p4.x));
}

float4 voronoi( idatas i, float2 uv, float time )
{
  float2 f ;
  float2 n = decompose( uv, f ) ;

  float seed = i.nbUserStroke ;

    //---------------------------------
    // regular voronoi
    //----------------------------------
    float4 col = 0 ;
    float3 lastd = 10 ;
    for( int j=-1; j<=1; j++ )
    {
    for( int i=-1; i<=1; i++ )
    {
      float2 g = float2( i, j );

      float2 ipos = n + g  ;

      float2 o = hash2( ipos+seed );
      o = 0.5 + 0.5*sin( time + 6.2831*o );  // animate the position

      float2 r = g + o - f;
      float d = dot(r,r);

      // Get color and orderd 3 min distances
      if( d < lastd.x )
      {
        col = grad.sample( hash2( ipos+seed+5.4 ).x );
        lastd.yz = lastd.xy ;
        lastd.x = d ;
      }
      else if( d < lastd.y )
      {
        lastd.z = lastd.y ;
        lastd.y = d ;
      }
      else if( d < lastd.z )
      {
        lastd = d ;
      }
    }
  }

    return col ;
}

float4 main( idatas i )
{
  box2 b = box2FromCenterAxe( i.strokeStartPos, max( length(i.strokePos-
i.strokeStartPos), 2 ),  normalizeSafe(i.strokePos-i.strokeStartPos) );
  float2 p = 4*b.toCenter( i.pos ) / b.size ;
  float4 col = voronoi( i, p, i.dist*0.005 ) ;

  return col ;
}
```

Cell Voronoi Edge

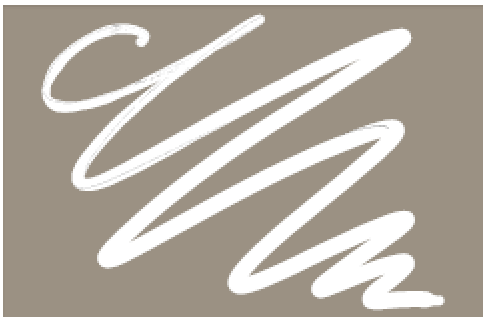
```
cfg{
  name="Cell Voronoi Edge";
  renderingTime = 30 ;
}

globals{
  float distFromEdge = 0.1 ;
  {
    uiMax = 1 ;
    uiName = "Size";
    uiTab = "Edge" ;
    uiFormat = percent ;
  }
}

float hash1( float2 p)
{
  float hscale = .1031 ;
  float3 p3 = frac( float3(p.x,p.y,p.x) * hscale ) ;
  p3 += dot(p3, p3.yzx + 19.19);

  return frac( (p3.x + p3.y) * p3.z );
}

float2 hash2( float2 p )
{
  float3 hscale3 = float3( .1031, .1030, .0973 ) ;
  float3 p3 = frac( float3(p.x,p.y,p.x) * hscale3) ;
  p3 += dot(p3, p3.yzx+19.19) ;

  return frac( float2( (p3.x + p3.y)*p3.z, (p3.x+p3.z)*p3.y) );
}

float4 hash4( float2 p )
{
  float4 hscale4 = float4( .1031, .1030, .0973, .1099 ) ;

  float4 p4 = frac( float4(p.xyxy) * hscale4 );
    p4 += dot( p4, p4.wzxy+19.19) ;
  return frac( float4((p4.x + p4.y)*p4.z, (p4.x + p4.z)*p4.y, (p4.y + p4.z)*p4.w,
(p4.z + p4.w)*p4.x));
}

float voronoiEdge( idatas i, float2 uv, float time )
{
  float2 f ;
  float2 n = decompose( uv, f ) ;

  float seed = i.nbUserStroke ;
    //----------------------------------
    // regular voronoi
    //----------------------------------
    float md = 10 ;
  float2 mg;
  float2 mr;
    for( int j=-1; j<=1; j++ )
    {
    for( int i=-1; i<=1; i++ )
    {
      float2 g = float2( i, j );

      float2 ipos = n + g  ;

      float2 o = hash2( ipos+seed );
      o = 0.5 + 0.5*sin( time + 6.2831*o );  // animate the position

      float2 r = g + o - f;
      float d = dot(r,r);

      if( d < md )
      {
        md = d  ;
        mr = r  ;
        mg = g  ;
      }

    }
  }

  //----------------------------------
    // second pass: distance to borders
    //----------------------------------
    md = 8.0;
    float2 mr2 ;
    for( int j=-2; j<=2; j++ )
  {
    for( int i=-2; i<=2; i++ )
    {
      float2 g = float2( i, j );

      float2 ipos = n + g  ;

      float2 o = hash2( ipos+seed );
      o = 0.5 + 0.5*sin( time + 6.2831*o );  // animate the position

      float2 r = g + o - f;

      if( dot(mr-r,mr-r)>0.00001 )
      {
        float dist = dot( 0.5*(mr+r), normalize(r-mr) ) ;
        if( dist < md )
        {
          md = dist  ;
          mr2 = r ;
        }
      }
    }
  }

    return 2*md ;
```

```
}

float4 main( idatas i )
{
  box2 b = box2FromCenterAxe( i.strokeStartPos, max( length(i.strokePos-
i.strokeStartPos), 2 ), normalizeSafe(i.strokePos-i.strokeStartPos) );
  float2 p = 4*b.toCenter( i.pos ) / b.size ;
  float edgeD = voronoiEdge( i, p, i.dist*0.005 ) ;

  float alpha = 1 ;
  alpha *= edgeD < distFromEdge ? 1 : 0 ;
  float4 col = i.color ;
  col.a *= alpha;

  return col ;
}
```

Cell Voronoi gradient

```
cfg{
  name="Cell Voronoi gradient";
  renderingTime = 30 ;
}

globals{
  colorGradient grad ;
  {
    uiTab = "color" ;
    id = 1 ;
  }
}

float hash1( float2 p)
{
  float hscale = .1031 ;

  float3 p3 = frac( float3(p.x,p.y,p.x) * hscale ) ;
    p3 += dot(p3, p3.yzx + 19.19);
    return frac( (p3.x + p3.y) * p3.z );
}

float2 hash2( float2 p )
{
  float3 hscale3 = float3( .1031, .1030, .0973 ) ;

  float3 p3 = frac( float3(p.x,p.y,p.x) * hscale3) ;
    p3 += dot(p3, p3.yzx+19.19) ;
    return frac( float2( (p3.x + p3.y)*p3.z, (p3.x+p3.z)*p3.y) );
}

float4 hash4( float2 p )
{
  float4 hscale4 = float4( .1031, .1030, .0973, .1099 ) ;

  float4 p4 = frac( float4(p.xyxy) * hscale4 );
    p4 += dot( p4, p4.wzxy+19.19) ;
  return frac( float4((p4.x + p4.y)*p4.z, (p4.x + p4.z)*p4.y, (p4.y + p4.z)*p4.w,
(p4.z + p4.w)*p4.x));
}

float4 voronoi( idatas i, float2 uv, float time )
{
  float2 f ;
  float2 n = decompose( uv, f ) ;

  float seed = i.nbUserStroke ;

    //----------------------------------
    // regular voronoi
    //----------------------------------
    float4 col = 0 ;
    float lastd = 10 ;
    for( int j=-1; j<=1; j++ )
    {
    for( int i=-1; i<=1; i++ )
    {
      float2 g = float2( i, j );

      float2 ipos = n + g  ;

      float2 o = hash2( ipos+seed );
      o = 0.5 + 0.5*sin( time + 6.2831*o );  // animate the position

      float2 r = g + o - f;
      float d = dot(r,r);

      if( d < lastd )
      {
        col = grad.sample( hash2( ipos+seed+5.4 ).x );
        lastd = d ;
      }

    }
  }

    return col ;
}

float4 main( idatas i )
{
  box2 b = box2FromCenterAxe( i.strokeStartPos, max( length(i.strokePos-
i.strokeStartPos), 2 ), normalizeSafe(i.strokePos-i.strokeStartPos) );
  float2 p = 4*b.toCenter( i.pos ) / b.size ;
  float4 col = voronoi( i, p, i.dist*0.005 ) ;

  return col ;
}
```

Cell Voronoi trabeculum bicolor



```
cfg{
  name="Cell Voronoi trabeculum bicolor";
  renderingTime = 30 ;
}

globals{
  float4 colA = 1 ;
  float4 colB = float4(0,0,0,1);
}

perPrim {
  float hardness = 0.5 ;
  {
    id = -1 ;
    uiMin = 0       ;
    uiMax = 1       ;
    uiTab = "shape"  ;
    uiFormat = percent  ;
    uiName = "hardness"  ;
  }
}

float hash1( float2 p)
{
  float hscale = .1031 ;

  float3 p3 = frac( float3(p.x,p.y,p.x) * hscale ) ;
    p3 += dot(p3, p3.yzx + 19.19);
    return frac( (p3.x + p3.y) * p3.z );
}

float2 hash2( float2 p )
{
  float3 hscale3 = float3( .1031, .1030, .0973 ) ;

  float3 p3 = frac( float3(p.x,p.y,p.x) * hscale3) ;
    p3 += dot(p3, p3.yzx+19.19) ;
    return frac( float2( (p3.x + p3.y)*p3.z, (p3.x+p3.z)*p3.y) );
}

float4 hash4( float2 p )
{
  float4 hscale4 = float4( .1031, .1030, .0973, .1099 ) ;

  float4 p4 = frac( float4(p.xyxy) * hscale4 );
    p4 += dot( p4, p4.wzxy+19.19) ;
  return frac( float4((p4.x + p4.y)*p4.z, (p4.x + p4.z)*p4.y, (p4.y + p4.z)*p4.w,
(p4.z + p4.w)*p4.x));
```

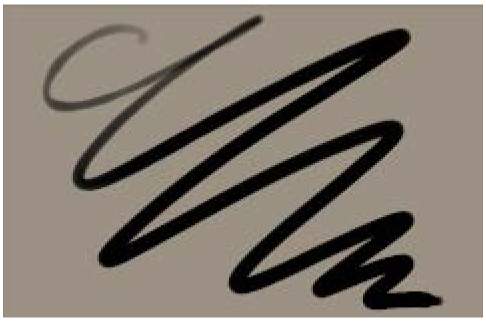
```
}

float4 voronoi( idatas i, float2 uv, float time )
{
  float2 f ;
  float2 n = decompose( uv, f ) ;

  float seed = i.nbUserStroke ;

    //----------------------------------
    // regular voronoi
    //----------------------------------
    float3 lastd = 10 ;
    for( int j=-1; j<=1; j++ )
    {
    for( int i=-1; i<=1; i++ )
    {
      float2 g = float2( i, j );

      float2 ipos = n + g  ;

      float2 o = hash2( ipos+seed );
      o = 0.5 + 0.5*sin( time + 6.2831*o );  // animate the position

      float2 r = g + o - f;
      float d = dot(r,r);

      // Get color and orderd 3 min distances
      if( d < lastd.x )
      {
        lastd.yz = lastd.xy ;
        lastd.x = d ;
      }
      else if( d < lastd.y )
      {
        lastd.z = lastd.y ;
        lastd.y = d ;
      }
      else if( d < lastd.z )
      {
        lastd.z = d ;
      }
    }
  }

  lastd = 5*sqrt(lastd);

  float alpha = 1./( 1./ (lastd.y-lastd.x)+1./(lastd.z-lastd.x) ) ; // Formula (c)
Fabrice NEYRET

  float hcut = lerp( 3, 0.05, i.hardness ) ;
  float4 col = alpha > hcut ? colA : colB  ;

    return col ;
}

float4 main( idatas i )
{
  box2 b = box2FromCenterAxe( i.strokeStartPos, max( length(i.strokePos-
i.strokeStartPos), 2 ), normalizeSafe(i.strokePos-i.strokeStartPos) );
  float2 p = 4*b.toCenter( i.pos ) / b.size ;
  float4 col = voronoi( i, p, i.dist*0.005 ) ;

  return col ;
}
```

Cell Voronoi trabeculum gradient

```
cfg{
  name="Cell Voronoi trabeculum gradient";
  renderingTime = 30 ;
}

globals{
  colorGradient grad ;
  {
    uiTab = "color" ;
    id = 1 ;
  }
}

perPrim {
  float hardness = 0.5 ;
  {
    id = -1 ;
    uiMin = 0      ;
    uiMax = 1      ;
    uiTab = "shape"  ;
    uiFormat = percent  ;
    uiName = "hardness"  ;
  }
}

float hash1( float2 p)
{
  float hscale = .1031 ;

  float3 p3 = frac( float3(p.x,p.y,p.x) * hscale ) ;
    p3 += dot(p3, p3.yzx + 19.19);
    return frac( (p3.x + p3.y) * p3.z );
}

float2 hash2( float2 p )
{
  float3 hscale3 = float3( .1031, .1030, .0973 ) ;

  float3 p3 = frac( float3(p.x,p.y,p.x) * hscale3) ;
    p3 += dot(p3, p3.yzx+19.19) ;
    return frac( float2( (p3.x + p3.y)*p3.z, (p3.x+p3.z)*p3.y) );
}

float4 hash4( float2 p )
{
  float4 hscale4 = float4( .1031, .1030, .0973, .1099 ) ;

  float4 p4 = frac( float4(p.xyxy) * hscale4 );
    p4 += dot( p4, p4.wzxy+19.19) ;
  return frac( float4((p4.x + p4.y)*p4.z, (p4.x + p4.z)*p4.y, (p4.y + p4.z)*p4.w,
(p4.z + p4.w)*p4.x));
}

float4 voronoi( idatas i, float2 uv, float time )
{
  float2 f ;
  float2 n = decompose( uv, f ) ;

  float seed = i.nbUserStroke ;

    //----------------------------------
    // regular voronoi
    //----------------------------------
    float4 col = 0 ;
    float3 lastd = 10 ;
    for( int j=-1; j<=1; j++ )
    {
    for( int i=-1; i<=1; i++ )
    {
      float2 g = float2( i, j );

      float2 ipos = n + g  ;

      float2 o = hash2( ipos+seed );
      o = 0.5 + 0.5*sin( time + 6.2831*o );  // animate the position

      float2 r = g + o - f;
      float d = dot(r,r);

      // Get color and orderd 3 min distances
      if( d < lastd.x )
      {
        col = grad.sample( hash2( ipos+seed+5.4 ).x );
        lastd.yz = lastd.xy ;
        lastd.x = d ;
      }
      else if( d < lastd.y )
      {
        lastd.z = lastd.y ;
        lastd.y = d ;
      }
      else if( d < lastd.z )
      {
        lastd.z = d ;
      }
    }
  }

  lastd = 5*sqrt(lastd);

  float alpha = 1./( 1./ (lastd.y-lastd.x)+1./(lastd.z-lastd.x) ) ; // Formula (c)
Fabrice NEYRET

  float hcut = lerp( 3, 0.05, i.hardness ) ;
  col.a *= alpha > hcut ? 1 : 0  ;

    return col ;
}

float4 main( idatas i )
```

```
{
  box2 b = box2FromCenterAxe( i.strokeStartPos, max( length(i.strokePos-i.strokeStartPos), 2 ), normalizeSafe(i.strokePos-i.strokeStartPos) );
  float2 p = 4*b.toCenter( i.pos ) / b.size ;
  float4 col = voronoi( i, p, i.dist*0.005 ) ;

  return col ;
}
```

Charcoal dual texture



```
cfg{
  name="Charcoal dual texture";
  renderingTime = 30 ;

  samplerDefault ;
  {
    adressU = clamp;
    adressV = clamp;
  }
}

globals{
  uiTab "Texture"  ;

  texture shape;
  {
    uiTab = "Texture" ;
  }
  texture splash;
  {
    uiTab = "Texture" ;
  }
}

float4 hash2( float2 p )
{
  float4 hscale = float4( .1031, 0.1059, -0.1087, -0.1029 ) ;

  float4 p4 = frac( p.xyxy * hscale ) ;
    p4 += dot(p4, p4.yzwx + 209.191349);
    return frac( float4( (p4.x + p4.y) * ( p4.z - p4.w ),
             (p4.y + p4.z) * ( p4.w - p4.x ),
             (p4.z + p4.w) * ( p4.x - p4.y ),
             (p4.w + p4.x) * ( p4.y - p4.z ) ) );
}

float4 cptSplash( float2 pos, float seed )
{
  float4 col = splash.sample( pos ) ;  // first sampling


  float4 noise ;
  matrix2 b ;
  float maxmove = 0.10 ;
  float minSize = 0.795 ;
  float maxSize = 1.1 ;
  float2 texp ;

  // other sampling
  noise = hash2( float2(seed,0) ) ;
  b = matrix2FromPRS( maxmove*(noise.xy*2-1),
            TWOPI*noise.w,
            lerp( minSize, maxSize, noise.z ) ) ;

  texp = b.transform( pos-0.5 )+0.5;
  col += splash.sample( texp ) ;


  // other sampling
  noise = hash2( float2(seed,1) ) ;
  b = matrix2FromPRS( maxmove*(noise.xy*2-1),
            TWOPI*noise.w,
            lerp( minSize, maxSize, noise.z ) ) ;

  texp = b.transform( pos-0.5 )+0.5;
  col += splash.sample( texp ) ;


  // other sampling
  noise = hash2( float2(seed,2) ) ;
  b = matrix2FromPRS( maxmove*(noise.xy*2-1),
            TWOPI*noise.w,
            lerp( minSize, maxSize, noise.z ) ) ;

  texp = b.transform( pos-0.5 )+0.5;
  col += splash.sample( texp ) ;


  return saturate( col );
}

float4 main( idatas i )
{
  float alpha = i.color.a ;

  // Get Shape
  {
    float2 pos = i.primBox.toUpperLeftNorm( i.pos ) ;
    float4 col = shape.sample( pos ) ;
    float a = col.a ;
    a = shape.isEmpty ? 1 : a ;
    alpha *= a ;
  }

  // Get Splash
  {
    float2 pos = i.primBox1.toUpperLeftNorm( i.pos ) ;
    float4 col = cptSplash( pos, i.primShapeId1+i.nbUserStroke ) ;

    float aover = i.primBox1Valid ? 1 : 0 ;
    float as = pow( i.color.a, 0.15 ) ;
    aover *= as ;

    // apply overlay blending
    alpha = blendOverlay( float4(alpha,alpha,alpha,1), float4(col.aaa,aover) ).x ;
  }

  return float4( i.color.xyz, alpha ) ;
}
```

Charcoal dual texture background overlay

```
cfg{
  name="Charcoal dual texture background overlay";
  renderingTime = 30 ;

  samplerDefault ;
  {
    adressU = clamp;
    adressV = clamp;
  }
}

globals{
  uiTab "Texture"  ;

  uiTab roughness  ;
  {
    row = 100 ;
  }

  uiTab stroke  ;
  {
    row = 99 ;
  }

  texture shape;
  {
    id = -2  ;
    uiTab = "shape" ;
    uiName = "texture shape";
  }
  texture splash;
  {
    id = -1 ;
    uiTab = "shape" ;
    uiName = "texture splash";
  }

  sampler smplmat ;

  texture mat ;
  {
    uiTab = "roughness" ;
    uiName = "pattern" ;
  }

  float scalePattern ;
  {
    id = 100;
    uiMin = 0.25;
    uiMax = 10    ;
    uiFormat = percent ;
    uiTab = "roughness" ;
  }
}

perPrim {

  // direction du tracé actuel
  float2 dir ;
  {
    uiEditor = angleDist ;
    raw = true ;
    uiTab = "stroke" ;
  }

  float pressure = 0.5 ;
  {
    uiMin = 0    ;
    uiMax = 1    ;
    uiFormat = percent ;
    uiName = "pressure" ;
    uiTab = "stroke" ;
  }

  float directionality = 0.85 ;
  {
    uiMin = 0  ;
    uiMax = 1  ;
    uiFormat = percent ;
    uiName = "directionality" ;
    uiTab = "stroke" ;
  }

}

float4 hash2( float2 p )
{
  float4 hscale = float4( .1031, 0.1059, -0.1087, -0.1029 ) ;

  float4 p4 = frac( p.xyxy * hscale ) ;
    p4 += dot(p4, p4.yzwx + 209.191349);
    return frac( float4( (p4.x + p4.y) * ( p4.z - p4.w ),
             (p4.y + p4.z) * ( p4.w - p4.x ),
             (p4.z + p4.w) * ( p4.x - p4.y ),
             (p4.w + p4.x) * ( p4.y - p4.z ) ) );
}

float4 cptSplash( float2 pos, float seed )
{
  float4 col = splash.sample( pos ) ;  // first sampling
//  col = 0 ;

  float4 noise ;
  matrix2 b ;
  float maxmove = 0.50 ;
  float minSize = 1. ;
  float maxSize = 4.5 ;
  float2 texp ;

  // other sampling
  noise = hash2( float2(seed,0) ) ;
  b = matrix2FromPRS( maxmove*(noise.xy*2-1),
            TWOPI*noise.w,
            lerp( minSize, maxSize, noise.z ) ) ;

  texp = b.transform( pos-0.5 )+0.5;
  col += splash.sample( texp ) ;


  // other sampling
  noise = hash2( float2(seed,1) ) ;
  b = matrix2FromPRS( maxmove*(noise.xy*2-1),
            TWOPI*noise.w,
            lerp( minSize, maxSize, noise.z ) ) ;

  texp = b.transform( pos-0.5 )+0.5;
  col += splash.sample( texp ) ;


  // other sampling
  noise = hash2( float2(seed,2) ) ;
  b = matrix2FromPRS( maxmove*(noise.xy*2-1),
            TWOPI*noise.w,
            lerp( minSize, maxSize, noise.z ) ) ;

  texp = b.transform( pos-0.5 )+0.5;
  col += splash.sample( texp ) ;


  // other sampling
  noise = hash2( float2(seed,3) ) ;
  b = matrix2FromPRS( maxmove*(noise.xy*2-1),
            TWOPI*noise.w,
            lerp( minSize, maxSize, noise.z ) ) ;

  texp = b.transform( pos-0.5 )+0.5;
  col += splash.sample( texp ) ;

  return saturate( col );
}

float surfaceHeight( float2 uv )
{
  return luminance( mat.sample( smplmat, uv/scalePattern ) )  ;
}

float3 surfaceNormal( float2 pos, float h )
{
  float2 opa = 1/mat.size ;
  float2 uv = pos / mat.size ;

  // compute normal with a Sobel filter
  float s00 = surfaceHeight( uv + opa*float2(-1,-1) );
  float s10 = surfaceHeight( uv + opa*float2( 0,-1) );
  float s20 = surfaceHeight( uv + opa*float2( 1,-1) );
  float s01 = surfaceHeight( uv + opa*float2(-1, 0) );
  float s21 = surfaceHeight( uv + opa*float2( 1, 0) );
  float s02 = surfaceHeight( uv + opa*float2(-1, 1) );
  float s12 = surfaceHeight( uv + opa*float2( 0, 1) );
  float s22 = surfaceHeight( uv + opa*float2( 1, 1) );

  // Compute dx using Sobel:
```

```
  //           -1 0 1
  //           -2 0 2
  //           -1 0 1
  float dX = -s00 + s20 -2*s01 + 2*s21 -s02 + s22  ;

    // Compute dy using Sobel:
    //           -1 -2 -1
    //            0  0  0
    //            1  2  1
    float dY = -s00 -2*s10 -s20 + s02 + 2*s12 + s22 ;

  return normalizeSafe( float3( h*dX, h*dY, 1 ) );
}

float remap( in float a, float minv, float maxv )
{
  return saturate( (a-minv) / (maxv-minv) );
}

float cptBackMaterial( idatas i )
{
  float heightscale = 2 ;

  float2 dir = normalizeSafe( i.dir );

  float3 bn = normalizeSafe( float3(heightscale*dir,-1) );
  float3 cn = surfaceNormal( i.pos, heightscale );

  float ah = surfaceHeight( i.pos / mat.size ) ;
  float pressure = i.pressure ;
  float pmin;
  float pmax ;

  pmin = pressure < 0.5 ? (1-2*pressure) : -(pressure-0.49) ;
  pmax = pressure < 0.5 ? 1.01 : (1-2*(pressure-0.5)) ;
  ah = remap( ah, pmin, pmax );

  float a = saturate( -dot( bn, cn ) );
  a = lerp( 1-i.directionality, 1, lerp(a,1,pressure*pressure*pressure) );

  a *= ah ;
  return a ;
}

float4 main( idatas i )
{
  float alpha = i.color.a ;


  // Get Shape
  {
    float2 pos = i.primBox.toUpperLeftNorm( i.pos ) ;
    float4 col = shape.sample( pos ) ;
    float a = col.a ;
    a = shape.isEmpty ? 1 : a ;
    alpha *= a ;
  }

  // Get background
  alpha *= cptBackMaterial( i );
  // Get Splash
  {
    float2 pos = i.primBox1.toUpperLeftNorm( i.pos ) ;
    float4 col = cptSplash( pos, i.primShapeId1+i.nbUserStroke ) ;

    float aover = i.primBox1Valid ? 1 : 0 ;

    // apply overlay blending
    alpha = blendOverlay( float4(alpha,alpha,alpha,1), float4(col.aaa,aover) ).x ;
  }


  return float4( i.color.xyz, alpha ) ;
}
```

Charcoal dual texture background smudge

```
cfg{
  name="Charcoal dual texture background smudge";
  renderingTime = 40 ;
  blendEx = true ;
  blendDefault = replace ;
  samplingLayerMaxOffset = 64 ;

  samplerDefault ;
  {
    adressU = clamp;
    adressV = clamp;
  }
}

globals{
  uiTab "Texture"  ;

  uiTab roughness  ;
  {
    row = 100 ;
  }

  uiTab stroke  ;
  {
    row = 99 ;
  }

  texture shape;
  {
    id = -2  ;
    uiTab = "shape" ;
    uiName = "texture shape";
  }
  texture splash;
  {
    id = -1 ;
    uiTab = "shape" ;
    uiName = "texture splash";
  }

  sampler smplmat ;

  texture mat ;
  {
    uiTab = "roughness" ;
    uiName = "pattern" ;
  }

  float scalePattern ;
  {
    id = 100;
    uiMin = 0.25;
    uiMax = 10    ;
    uiFormat = percent ;
    uiTab = "roughness" ;
  }
}

perPrim {

  // direction du tracé actuel
  float2 dir ;
  {
    uiEditor = angleDist ;
    raw = true ;
    uiTab = "stroke" ;
  }

  float pressure = 0.5 ;
  {
    uiMin = 0    ;
    uiMax = 1    ;
    uiFormat = percent ;
    uiName = "pressure" ;
    uiTab = "stroke" ;
  }

  float directionality = 0.85 ;
  {
    uiMin = 0  ;
    uiMax = 1  ;
    uiFormat = percent ;
    uiName = "directionality" ;
    uiTab = "stroke" ;
  }

  float load = 0.2 ;
  {
    uiMax = 1  ;
    uiFormat = percent ;
    uiTab = "color" ;
  }

}

float4 hash2( float2 p )
{
  float4 hscale = float4( .1031, 0.1059, -0.1087, -0.1029 ) ;

  float4 p4 = frac( p.xyxy * hscale ) ;
    p4 += dot(p4, p4.yzwx + 209.191349);
    return frac( float4( (p4.x + p4.y) * ( p4.z - p4.w ),
             (p4.y + p4.z) * ( p4.w - p4.x ),
             (p4.z + p4.w) * ( p4.x - p4.y ),
             (p4.w + p4.x) * ( p4.y - p4.z ) ) );
}

float4 cptSplash( float2 pos, float seed )
{
  float4 col = splash.sample( pos ) ;  // first sampling
//  col = 0 ;
```

```
  float4 noise ;
  matrix2 b ;
  float maxmove = 0.50 ;
  float minSize = 1. ;
  float maxSize = 4.5 ;
  float2 texp ;

  // other sampling
  noise = hash2( float2(seed,0) ) ;
  b = matrix2FromPRS( maxmove*(noise.xy*2-1),
            TWOPI*noise.w,
            lerp( minSize, maxSize, noise.z ) ) ;

  texp = b.transform( pos-0.5 )+0.5;
  col += splash.sample( texp ) ;


  // other sampling
  noise = hash2( float2(seed,1) ) ;
  b = matrix2FromPRS( maxmove*(noise.xy*2-1),
            TWOPI*noise.w,
            lerp( minSize, maxSize, noise.z ) ) ;

  texp = b.transform( pos-0.5 )+0.5;
  col += splash.sample( texp ) ;


  // other sampling
  noise = hash2( float2(seed,2) ) ;
  b = matrix2FromPRS( maxmove*(noise.xy*2-1),
            TWOPI*noise.w,
            lerp( minSize, maxSize, noise.z ) ) ;

  texp = b.transform( pos-0.5 )+0.5;
  col += splash.sample( texp ) ;


  // other sampling
  noise = hash2( float2(seed,3) ) ;
  b = matrix2FromPRS( maxmove*(noise.xy*2-1),
            TWOPI*noise.w,
            lerp( minSize, maxSize, noise.z ) ) ;

  texp = b.transform( pos-0.5 )+0.5;
  col += splash.sample( texp ) ;

  return saturate( col );
}

float surfaceHeight( float2 uv )
{
  return luminance( mat.sample( smplmat, uv/scalePattern ) )  ;
}

float3 surfaceNormal( float2 pos, float h )
{
  float2 opa = 1/mat.size ;
  float2 uv = pos / mat.size ;

  // compute normal with a Sobel filter
  float s00 = surfaceHeight( uv + opa*float2(-1,-1) );
  float s10 = surfaceHeight( uv + opa*float2( 0,-1) );
  float s20 = surfaceHeight( uv + opa*float2( 1,-1) );
  float s01 = surfaceHeight( uv + opa*float2(-1, 0) );
  float s21 = surfaceHeight( uv + opa*float2( 1, 0) );
  float s02 = surfaceHeight( uv + opa*float2(-1, 1) );
  float s12 = surfaceHeight( uv + opa*float2( 0, 1) );
  float s22 = surfaceHeight( uv + opa*float2( 1, 1) );

  // Compute dx using Sobel:
  //           -1 0 1
  //           -2 0 2
  //           -1 0 1
  float dX = -s00 + s20 -2*s01 + 2*s21 -s02 + s22  ;

    // Compute dy using Sobel:
    //           -1 -2 -1
    //            0  0  0
    //            1  2  1
    float dY = -s00 -2*s10 -s20 + s02 + 2*s12 + s22 ;

  return normalizeSafe( float3( h*dX, h*dY, 1 ) );
}

float remap( in float a, float minv, float maxv )
{
  return saturate( (a-minv) / (maxv-minv) );
}

float cptBackMaterial( idatas i )
{
  float heightscale = 2 ;

  float2 dir = normalizeSafe( i.dir );

  float3 bn = normalizeSafe( float3(heightscale*dir,-1) );
  float3 cn = surfaceNormal( i.pos, heightscale );

  float ah = surfaceHeight( i.pos / mat.size ) ;
  float pressure = i.pressure ;
  float pmin;
  float pmax ;

  pmin = pressure < 0.5 ? (1-2*pressure) : -(pressure-0.49) ;
  pmax = pressure < 0.5 ? 1.01 : (1-2*(pressure-0.5)) ;
  ah = remap( ah, pmin, pmax );

  float a = saturate( -dot( bn, cn ) );
  a = lerp( 1-i.directionality, 1, lerp(a,1,pressure*pressure*pressure) );

  a *= ah ;
  return a ;
}

float4 Smudge( idatas i, float power )
{

  // get the current background color
  float4 colBack = bottomLayer.pointSample( i, i.pos );

  // compute displacement from the current position
  float2 dir = i.strokePrecedPos - i.strokePos ;
  dir *= power;
  float mdisplace = length( dir ) ;

  // Get the displaced color
  float4 smpl = bottomLayer.pointSample( i, i.pos + dir );

  return smpl ;
}

float4 main( idatas i )
{
  float alpha = i.color.a ;


  // Get Shape
  {
    float2 pos = i.primBox.toUpperLeftNorm( i.pos ) ;
    float4 col = shape.sample( pos ) ;
    float a = col.a ;
    a = shape.isEmpty ? 1 : a ;
    alpha *= a ;
  }

  // Get background
  alpha *= cptBackMaterial( i );

  // Get Splash
  {
    float2 pos = i.primBox1.toUpperLeftNorm( i.pos ) ;
    float4 col = cptSplash( pos, i.primShapeId1+i.nbUserStroke ) ;

    float aover = i.primBox1Valid ? 1 : 0 ;

    // apply overlay blending
    alpha = blendOverlay( float4(alpha,alpha,alpha,1), float4(col.aaa,aover) ).x ;
  }

  // final color blending
  float4 colS = Smudge( i, alpha ) ;
  float4 colI = float4( i.color.xyz, alpha*i.load ) ;
  float4 col = blendNormal( colS, colI );

  // take care of the eraser state
  col = i.eraser ? float4( colS.xyz, colS.w * (1-colI.w) ) : col ;

  return col ;
  //return float4( i.color.xyz, alpha ) ;
}
```

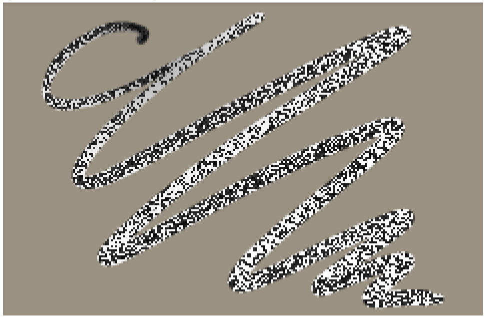
Charcoal dual texture multiply

```
cfg{
  name="Charcoal dual texture multiply";
  renderingTime = 30 ;

  samplerDefault ;
  {
    adressU = clamp;
    adressV = clamp;
  }
}

globals{
  uiTab "Texture"  ;

  uiTab roughness  ;
  {
    row = 100 ;
  }

  uiTab stroke  ;
  {
    row = 99 ;
  }

  texture shape;
  {
    id = -2  ;
    uiTab = "shape" ;
    uiName = "texture shape";
  }
  texture splash;
  {
    id = -1 ;
    uiTab = "shape" ;
    uiName = "texture splash";
  }

  sampler smplmat ;

  texture mat ;
  {
    uiTab = "roughness" ;
    uiName = "pattern" ;
  }

  float scalePattern ;
  {
    id = 100;
    uiMin = 0.25;
    uiMax = 10    ;
    uiFormat = percent ;
    uiTab = "roughness" ;
  }
}

perPrim {


  float pressure = 0.5 ;
  {
    uiMin = 0    ;
    uiMax = 1    ;
    uiFormat = percent ;
    uiName = "pressure" ;
    uiTab = "stroke" ;
  }

  float directionality = 0.85 ;
  {
    uiMin = 0  ;
    uiMax = 1  ;
    uiFormat = percent ;
    uiName = "directionality" ;
    uiTab = "stroke" ;
  }

  float2 dir ;
  {
    uiEditor = angleDist ;
    raw = true ;
    uiTab = "stroke" ;
  }

}

float4 hash2( float2 p )
{
  float4 hscale = float4( .1031, 0.1059, -0.1087, -0.1029 ) ;

  float4 p4 = frac( p.xyxy * hscale ) ;
    p4 += dot(p4, p4.yzwx + 209.191349);
    return frac( float4( (p4.x + p4.y) * ( p4.z - p4.w ),
             (p4.y + p4.z) * ( p4.w - p4.x ),
             (p4.z + p4.w) * ( p4.x - p4.y ),
             (p4.w + p4.x) * ( p4.y - p4.z ) ) );
}

float4 cptSplash( float2 pos, float seed )
{
  float4 col = splash.sample( pos ) ;  // first sampling

  float4 noise ;
  matrix2 b ;
  float maxmove = 0.40 ;
  float minSize = 0.5 ;
  float maxSize = 4.5 ;
  float2 texp ;

  // other sampling
  noise = hash2( float2(seed,0) ) ;
  b = matrix2FromPRS( maxmove*(noise.xy*2-1),
            TWOPI*noise.w,
            lerp( minSize, maxSize, noise.z ) ) ;

  texp = b.transform( pos-0.5 )+0.5;
  col += splash.sample( texp ) ;


  // other sampling
  noise = hash2( float2(seed,1) ) ;
  b = matrix2FromPRS( maxmove*(noise.xy*2-1),
            TWOPI*noise.w,
            lerp( minSize, maxSize, noise.z ) ) ;

  texp = b.transform( pos-0.5 )+0.5;
  col += splash.sample( texp ) ;


  // other sampling
  noise = hash2( float2(seed,2) ) ;
  b = matrix2FromPRS( maxmove*(noise.xy*2-1),
            TWOPI*noise.w,
            lerp( minSize, maxSize, noise.z ) ) ;

  texp = b.transform( pos-0.5 )+0.5;
  col += splash.sample( texp ) ;


  // other sampling
  noise = hash2( float2(seed,3) ) ;
  b = matrix2FromPRS( maxmove*(noise.xy*2-1),
            TWOPI*noise.w,
            lerp( minSize, maxSize, noise.z ) ) ;

  texp = b.transform( pos-0.5 )+0.5;
  col += splash.sample( texp ) ;

  // other sampling
  noise = hash2( float2(seed,4) ) ;
  b = matrix2FromPRS( maxmove*(noise.xy*2-1),
            TWOPI*noise.w,
            lerp( minSize, maxSize, noise.z ) ) ;

  texp = b.transform( pos-0.5 )+0.5;
  col += splash.sample( texp ) ;

  // other sampling
  noise = hash2( float2(seed,5) ) ;
  b = matrix2FromPRS( maxmove*(noise.xy*2-1),
            TWOPI*noise.w,
            lerp( minSize, maxSize, noise.z ) ) ;

  texp = b.transform( pos-0.5 )+0.5;
  col += splash.sample( texp ) ;

  return saturate( col );
}

float surfaceHeight( float2 uv )
{
  return luminance( mat.sample( smplmat, uv/scalePattern ) )  ;
}
```

```
float3 surfaceNormal( float2 pos, float h )
{
  float2 opa = 1/mat.size ;
  float2 uv = pos / mat.size ;

  // compute normal with a Sobel filter
  float s00 = surfaceHeight( uv + opa*float2(-1,-1) );
  float s10 = surfaceHeight( uv + opa*float2( 0,-1) );
  float s20 = surfaceHeight( uv + opa*float2( 1,-1) );
  float s01 = surfaceHeight( uv + opa*float2(-1, 0) );
  float s21 = surfaceHeight( uv + opa*float2( 1, 0) );
  float s02 = surfaceHeight( uv + opa*float2(-1, 1) );
  float s12 = surfaceHeight( uv + opa*float2( 0, 1) );
  float s22 = surfaceHeight( uv + opa*float2( 1, 1) );

  // Compute dx using Sobel:
  //           -1 0 1
  //           -2 0 2
  //           -1 0 1
  float dX = -s00 + s20 -2*s01 + 2*s21 -s02 + s22  ;

    // Compute dy using Sobel:
    //           -1 -2 -1
    //            0  0  0
    //            1  2  1
    float dY = -s00 -2*s10 -s20 + s02 + 2*s12 + s22 ;

  return normalizeSafe( float3( h*dX, h*dY, 1 ) );
}

float remap( in float a, float minv, float maxv )
{
  return saturate( (a-minv) / (maxv-minv) );
}

float cptBackMaterial( idatas i )
{
  float heightscale = 2 ;

  float2 dir = normalizeSafe( i.dir );

  float3 bn = normalizeSafe( float3(heightscale*dir,-1) );
  float3 cn = surfaceNormal( i.pos, heightscale );

  float ah = surfaceHeight( i.pos / mat.size ) ;
  float pressure = i.pressure ;
  float pmin;
  float pmax ;

  pmin = pressure < 0.5 ? (1-2*pressure) : -(pressure-0.49) ;
  pmax = pressure < 0.5 ? 1.01 : (1-2*(pressure-0.5)) ;
  ah = remap( ah, pmin, pmax );

  float a = saturate( -dot( bn, cn ) );
  a = lerp( 1-i.directionality, 1, lerp(a,1,pressure*pressure*pressure) );

  a *= ah ;
  return a ;
}
```

```
float4 main( idatas i )
{
  float alpha = i.color.a ;

  // Get Shape
  {
    float2 pos = i.primBox.toUpperLeftNorm( i.pos ) ;
    float4 col = shape.sample( pos ) ;
    float a = col.a ;
    a = shape.isEmpty ? 1 : a ;
    alpha *= a ;
  }

  // Get background
  alpha *= cptBackMaterial( i );

  // Get Splash
  {
    float2 pos = i.primBox1.toUpperLeftNorm( i.pos ) ;
    float4 col = cptSplash( pos, i.primShapeId1+i.nbUserStroke ) ;

    alpha *= col.a ;
  }

  return float4( i.color.xyz, alpha ) ;
}
```

gradient angular



```
cfg{
  name = "gradient angular" ;
}

globals{
  colorGradient grad ;
  {
    uiTab = "color" ;
    id = 1 ;
  }
  uiTab CenterHide;
  {
    row = 3 ;
  }
}

perPrim{
  float opacity=1;
  {
    uiMax = 1 ;
    uiTab = "color" ;
    uiFormat = percent ;
    id = 0 ;
  }
  float noisePower=0.1;
  {
    uiMax = 1 ;
    uiName = "Power";
    uiTab = "Dithering" ;
    uiFormat = percent ;
  }
  float noiseScale=1;
  {
    uiMax = 1 ;
    uiName = "Scale";
    uiTab = "Dithering" ;
    uiFormat = percent ;
  }
  float chide=0.06;
  {
    uiMax = 1 ;
    uiName = "Power";
    uiTab = "CenterHide" ;
    uiFormat = percent ;
  }
  float cpos=0;
  {
    uiMax = 1 ;
    uiName = "Pos";
    uiTab = "CenterHide" ;
    uiFormat = percent ;
  }
  float rotation=0;
  {
    uiMax = 1 ;
    uiName = "Rotation";
    uiTab = "Color" ;
    uiFormat = percent ;
  }
}

float hash( float3 p )
{
  float hscale = .1031 ;

  float3 p3 = frac( p * hscale ) ;
    p3 += dot(p3, p3.yzx + 209.191349);
    return frac( (p3.x + p3.y) * p3.z );
}

float noise( float3 x )
{
  return hash( floor(x) );
}

float4 main( idatas i )
{
  box2 b = box2FromCenterAxe( i.strokeStartPos, length(i.strokePos-
i.strokeStartPos), normalizeSafe(i.strokePos-i.strokeStartPos) );
  float2 p = b.toCenter( i.pos ) ;
  float spp = mod( toSpherical( p ), TWOPI ) / TWOPI;

  // compute noise value
  float2 npos = i.noiseScale*i.pos  ;
  float nv = 0.2*i.noisePower*(1-2*noise( float3(npos,i.nbUserStroke+i.dist*0.001)
));

  // hide centre disontinuity
  float l = length( i.strokeStartPos - i.pos ) ;
  float cphide = 1/(i.chide*500 + 1) ;
  float acenter = exp( -cphide*l );
  float4 ccenter = grad.sample( frac(saturate(i.cpos+nv)) );

  float4 col = grad.sample( frac(spp+nv+i.rotation) ) ;

  col = lerp( col, ccenter, acenter );

  col.a *= i.opacity ;

  return col ;
}
```

gradient linear

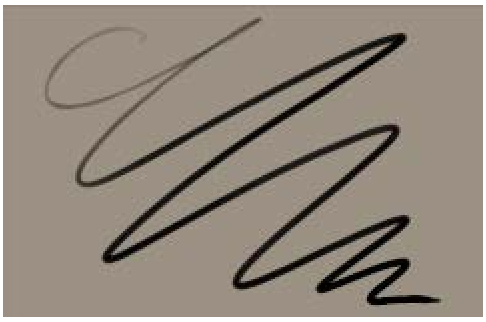
```
cfg{
  name = "gradient linear" ;
}

globals{
  colorGradient grad ;
  {
    uiTab = "color" ;
    id = 1 ;
  }
}

perPrim{
  float opacity=1;
  {
    uiMax = 1 ;
    uiTab = "color" ;
    uiFormat = percent ;
    id = 0 ;
  }
  float noisePower=0.1;
  {
    uiMax = 1 ;
    uiName = "Power";
    uiTab = "Dithering" ;
    uiFormat = percent ;
  }
  float noiseScale=1;
  {
    uiMax = 1 ;
    uiName = "Scale";
    uiTab = "Dithering" ;
    uiFormat = percent ;
  }
}

float hash( float3 p )
{
  float hscale = .1031 ;

  float3 p3 = frac( p * hscale ) ;
    p3 += dot(p3, p3.yzx + 209.191349);
    return frac( (p3.x + p3.y) * p3.z );
}

float noise( float3 x )
{
  return hash( floor(x) );
}
```

```
float4 main( idatas i )
{
  float2 dir = normalizeSafe( i.strokePos - i.strokeStartPos ) ;
  float l = length( i.strokeStartPos - i.strokePos );
  plane2 p = plane2FromNormPos( dir, i.strokeStartPos );
  plane2 p1 = plane2FromNormPos( perpendicular(dir), i.strokeStartPos );

  // compute noise value
  float2 npos = i.noiseScale*float2( p.getDistance(i.pos), p1.getDistance(i.pos) )
;
  float nv = i.noisePower*(1-2*noise( float3(npos,i.nbUserStroke+i.dist*0.001) ));

  float d = saturate( 0.2*nv + p.getDistance( i.pos ) / l );
  float4 col = grad.sample( d ) ;
  col.a *= i.opacity ;

  // output somehting only if with ahve more than 1 pixel to compute the plane
  col.a = (l>1) ? col.a : 0 ;

  return col ;
}
```

hardness ctrl

```
cfg{name="hardness ctrl";}

perPrim {
  float hardness = 0.5 ;
  {
    id = 1 ;
    uiMin = 0      ;
    uiMax = 1      ;
    uiTab = "shape"  ;
    uiFormat = percent  ;
    uiName = "hardness"  ;
  }
  float innerSize = 0.5 ;
  {
    id = 3 ;
    uiMin = 0      ;
    uiMax = 1      ;
    uiTab = "shape"  ;
    uiFormat = percent  ;
  }
}

// compute the normalized distance from the edge to the center of the primitive
float cptPrimNormDist( idatas i )
{
  float sm = 0.5*vmax( i.primBox.getSize() ) ;  // retreive maximum primitive axe
size
  float d = max( -i.primDistance, 0 ) / sm ;  // compute normalized current
distance to the primitive edge

  return i.primDistanceValid ? d : 0 ;
}

// compute Hardness
// d - distance from primitive edge [0,1]
// h - hardness parameter [0,1]
// i - distance to the inner "safe" size [0,1]
//
float cptHardness( float d, float h, float i )
{
  float attsize = (1-i)*2*(1-h) ;
  float att = saturate( (d - (1-i - attsize*0.5) ) / max( attsize,0.0001) ) ;
  return smoothLerp(0,1,att);
}

float4 main( idatas i )
{
  float d = cptPrimNormDist( i );
  float att = cptHardness( d, i.hardness, i.innerSize );
  att = i.primDistanceValid ? att : 1 ;

  float alpha = saturate(i.color.a*att) ;
  float3 col = i.color.xyz ;

  return float4( col, alpha ) ;
}
```

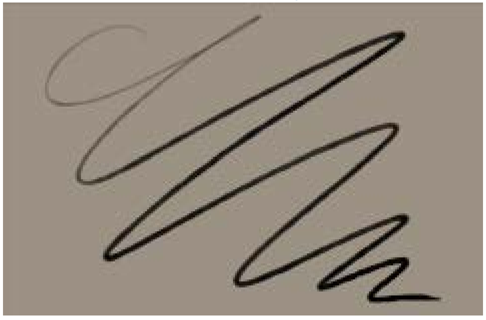
Hardness dither ctrl



```
cfg{
  name="Hardness dither ctrl";
  renderingTime = 20 ;
}

perPrim {
  float hardness = 0.5 ;
  {
    id = -1 ;
    uiMin = 0      ;
    uiMax = 1      ;
    uiTab = "shape"  ;
    uiFormat = percent  ;
    uiName = "hardness"  ;
  }
  float innerSize = 0.5 ;
  {
    id = 3 ;
    uiMin = 0      ;
    uiMax = 1      ;
    uiTab = "shape"  ;
    uiFormat = percent  ;
  }
  float noisePower=0.1;
  {
    uiMax = 1 ;
    uiName = "Power";
    uiTab = "Dithering" ;
    uiFormat = percent ;
  }
}

// compute the normalized distance from the edge to the center of the primitive
float cptPrimNormDist( idatas i )
{
  float sm = 0.5*vmax( i.primBox.getSize() ) ;  // retreive maximum primitive axe
size
  float d = max( -i.primDistance, 0 ) / sm ;    // compute normalized current
distance to the primitive edge

  return i.primDistanceValid ? d : 0 ;
}

// compute Hardness
// d - distance from primitive edge [0,1]
// h - hardness parameter [0,1]
// i - distance to the inner "safe" size [0,1]
//
float cptHardness( float d, float h, float i )
```

```
{
  float attsize = (1-i)*2*(1-h) ;
  float att = saturate( (d - (1-i - attsize*0.5) ) / max( attsize,0.0001) ) ;
  return smoothLerp(0,1,att);
}

float hash( float3 p )
{
  float3 hscale = float3(.1031,-.1029,.1032) ;

  float3 p3 = frac( p * hscale ) ;
    p3 += dot(p3, p3.yzx + 9.191349);
    return frac((p3.x + p3.y) * p3.z) ;
}

// return noise value [-1,1]
float noise( float3 x )
{
  return 2*hash( floor(x) )-1;
}

float4 main( idatas i )
{
  float d = cptPrimNormDist( i );

  // compute pure attenuation
  float att = cptHardness( d, i.hardness, i.innerSize );
  att = i.primDistanceValid ? att : 1 ;

  // compute noise value
  float2 npos = i.pos  ;
  float nv = (1-i.hardness)*(1-att)*i.noisePower*0.2*noise(
float3(npos,i.nbUserStroke+i.dist*0.0001) );
  float dn = saturate( d + nv );

  // compute perturbed attenuation
  float attn = cptHardness( dn, i.hardness, i.innerSize );
  attn = i.primDistanceValid ? attn : 1 ;


  float alpha = saturate(i.color.a*attn) ;
  float3 col = i.color.xyz ;

  return float4( col, alpha ) ;
}
```

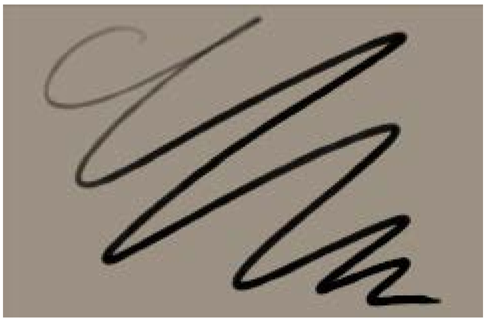
hardness noise blobby ctrl



```
cfg{
  name="hardness noise blobby ctrl";
  renderingTime = 30 ;
}

perPrim {
  float hardness = 0.5 ;
  {
    id = -1 ;
    uiMin = 0       ;
    uiMax = 1       ;
    uiTab = "shape"   ;
    uiFormat = percent   ;
    uiName = "hardness"   ;
  }
  float innerSize = 0.5 ;
  {
    id = 3 ;
    uiMin = 0       ;
    uiMax = 1       ;
    uiTab = "shape"   ;
    uiFormat = percent   ;
  }
  float noisePower=0.1;
  {
    uiMax = 1 ;
    uiName = "Power";
    uiTab = "Noise" ;
    uiFormat = percent ;
  }
  float noiseScale=1;
  {
    uiMin = 0.25 ;
    uiMax = 5 ;
    uiName = "Scale";
    uiTab = "Noise" ;
    uiFormat = percent ;
  }
  float ndensity = 0.5 ;
  {
    uiTab = "Noise" ;
    uiName = "Density"   ;
    uiFormat = percent ;
    uiMax = 1 ;
  }
}

// compute the normalized distance from the edge to the center of the primitive
float cptPrimNormDist( idatas i )
```

```
{
  float sm = 0.5*vmax( i.primBox.getSize() ) ;  // retreive maximum primitive axe
size
  float d = max( -i.primDistance, 0 ) / sm ;    // compute normalized current
distance to the primitive edge

  return i.primDistanceValid ? d : 0 ;
}

// compute Hardness
// d - distance from primitive edge [0,1]
// h - hardness parameter [0,1]
// i - distance to the inner "safe" size [0,1]
//
float cptHardness( float d, float h, float i )
{
  float attsize = (1-i)*2*(1-h) ;
  float att = saturate( (d - (1-i - attsize*0.5) ) / max( attsize,0.0001) ) ;
  return smoothLerp(0,1,att);
}

float3 hash( float3 p )
{
  float3 hscale = float3(.1031,-.1029,.1032) ;

  float3 p3 = frac( p * hscale ) ;
    p3 += dot(p3, p3.yzx + 209.191349);
    return -1+2*frac( float3(  (p3.x + p3.y) * p3.z,
                (p3.z + p3.y) * p3.x,
                (p3.x + p3.z) * p3.y ) );
}

float3 adjustDensity( float3 rs, float d )
{
  float3 r = abs(rs);
  d = 1-d ;
  float3 a = clamp( (d-0.5)*2, 0, 1 );
  float3 b = clamp( d*2, 0, 1 );

  r = (r-a) / max( b-a, 0.001) ;
  r = saturate( r );

  return rs < 0 ? -r : r ;
}

float3 hashD( float3 p, float d )
{
  return adjustDensity( hash(p), d );
}

// Gradient noise see https://www.shadertoy.com/view/XdXGW8
// the returned value is between [0,1]
float noise( float3 x, float d )
{
  float3 f = frac(x);
  float3 p = decompose( x, f );
    float3 u = f*f*(3.0-2.0*f);

    float v1 = bilinearLerp(  dot( hashD(p+float3(0,0,0),d), f-float3(0,0,0) ),
dot( hashD(p+float3(1,0,0),d), f-float3(1,0,0) ),
                dot( hashD(p+float3(0,1,0),d), f-float3(0,1,0) ), dot(
hashD(p+float3(1,1,0),d), f-float3(1,1,0) ), u.xy ) ;

    float v2 = bilinearLerp(  dot( hashD(p+float3(0,0,1),d), f-float3(0,0,1) ),
dot( hashD(p+float3(1,0,1),d), f-float3(1,0,1) ),
                dot( hashD(p+float3(0,1,1),d), f-float3(0,1,1) ), dot(
hashD(p+float3(1,1,1),d), f-float3(1,1,1) ), u.xy ) ;

  float ret = lerp( v1, v2, u.z )/0.707;  // normalize with the maxiumm slope
  return abs(ret);
}

// return a noise value between [-1,1]
float noiseFunction( float3 x, float d )
{
  matrix3 m  = matrix3FromPosAxes( float3(-0.8,0.63,1.03), float3(0.97,0.01,-
0.036), float3(-0.02,-.87,0.03), float3(0.04,0.024,1.01) );
  float ret = 0.5   * noise( x, d ) ; x = 2*m.transform(x);
  ret += 0.25 * noise(x, d);  x = 2*m.transform(x);
  ret += 0.125 * noise(x, d);  x = 2*m.transform(x);
  ret += 0.0625 * noise(x, d);  x = 2*m.transform(x);
  return ret*2-1 ;
}

float4 main( idatas i )
{
  float d = cptPrimNormDist( i );

  // compute noised distance
  float2 npos = toDrawSpace(i.pos)*.1/i.noiseScale  ;
  float dn = d + i.noisePower*noiseFunction(
float3(npos,i.nbUserStroke+i.dist*0.0001), i.ndensity )  ;

  // compute attenuation

  float att = cptHardness( dn, i.hardness, i.innerSize );
  att = i.primDistanceValid ? att : 1 ;

  float alpha = saturate(i.color.a*att) ;
  float3 col = i.color.xyz ;

  return float4( col, alpha ) ;
}
```

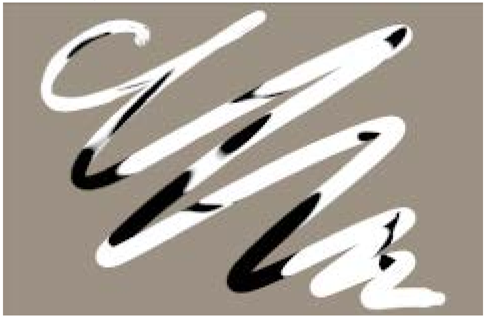
Hardness noise ctrl



```
cfg{
  name="Hardness noise ctrl";
  renderingTime = 30 ;
}

perPrim {
  float hardness = 0.5 ;
  {
    id = -1 ;
    uiMin = 0      ;
    uiMax = 1      ;
    uiTab = "shape"  ;
    uiFormat = percent  ;
    uiName = "hardness"  ;
  }
  float innerSize = 0.5 ;
  {
    id = 3 ;
    uiMin = 0      ;
    uiMax = 1      ;
    uiTab = "shape"  ;
    uiFormat = percent  ;
  }
  float noisePower=0.1;
  {
    uiMax = 1 ;
    uiName = "Power";
    uiTab = "Noise" ;
    uiFormat = percent ;
  }
  float noiseScale=1;
  {
    uiMin = 0.25 ;
    uiMax = 5 ;
    uiName = "Scale";
    uiTab = "Noise" ;
    uiFormat = percent ;
  }
  float ndensity = 0.5 ;
  {
    uiTab = "Noise" ;
    uiName = "Density"  ;
    uiFormat = percent ;
    uiMax = 1 ;
  }
}

// compute the normalized distance from the edge to the center of the primitive
float cptPrimNormDist( idatas i )
{
  float sm = 0.5*vmax( i.primBox.getSize() ) ;  // retreive maximum primitive axe
size
  float d = max( -i.primDistance, 0 ) / sm ;    // compute normalized current
distance to the primitive edge

  return i.primDistanceValid ? d : 0 ;
}

// compute Hardness
// d - distance from primitive edge [0,1]
// h - hardness parameter [0,1]
// i - distance to the inner "safe" size [0,1]
//
float cptHardness( float d, float h, float i )
{
  float attsize = (1-i)*2*(1-h) ;
  float att = saturate( (d - (1-i - attsize*0.5) ) / max( attsize,0.0001) ) ;
  return smoothLerp(0,1,att);
}

float3 hash( float3 p )
{
  float3 hscale = float3(.1031,-.1029,.1032) ;

  float3 p3 = frac( p * hscale ) ;
    p3 += dot(p3, p3.yzx + 209.191349);
    return -1+2*frac( float3(  (p3.x + p3.y) * p3.z,
                (p3.z + p3.y) * p3.x,
                (p3.x + p3.z) * p3.y ) );
}

float3 adjustDensity( float3 rs, float d )
{
  float3 r = abs(rs);
  d = 1-d ;
  float3 a = clamp( (d-0.5)*2, 0, 1 );
  float3 b = clamp( d*2, 0, 1 );

  r = (r-a) / max( b-a, 0.001) ;
  r = saturate( r );

  return rs < 0 ? -r : r ;
}

float3 hashD( float3 p, float d )
{
  return adjustDensity( hash(p), d );
}

// Gradient noise see https://www.shadertoy.com/view/XdXGW8
// the returned value is between [0,1]
float noise( float3 x, float d )
{
  float3 f = frac(x);
  float3 p = decompose( x, f );
    float3 u = f*f*(3.0-2.0*f);

    float v1 = bilinearLerp(  dot( hashD(p+float3(0,0,0),d), f-float3(0,0,0) ),
dot( hashD(p+float3(1,0,0),d), f-float3(1,0,0) ),
                dot( hashD(p+float3(0,1,0),d), f-float3(0,1,0) ), dot(
hashD(p+float3(1,1,0),d), f-float3(1,1,0) ), u.xy ) ;
```

```
    float v2 = bilinearLerp(  dot( hashD(p+float3(0,0,1),d), f-float3(0,0,1) ),
dot( hashD(p+float3(1,0,1),d), f-float3(1,0,1) ),
                dot( hashD(p+float3(0,1,1),d), f-float3(0,1,1) ), dot(
hashD(p+float3(1,1,1),d), f-float3(1,1,1) ), u.xy ) ;

  float ret = lerp( v1, v2, u.z )/0.707;  // normalize with the maxiumm slope
  return 0.5*ret+0.5;
}

// return a noise value between [-1,1]
float noiseFunction( float3 x, float d )
{
  matrix3 m  = matrix3FromPosAxes( float3(-0.8,0.63,1.03), float3(0.97,0.01,-
0.036), float3(-0.02,-.87,0.03), float3(0.04,0.024,1.01) );
  float ret = 0.5   * noise( x, d ) ; x = 2*m.transform(x);
  ret += 0.25 * noise(x, d);  x = 2*m.transform(x);
  ret += 0.125 * noise(x, d);  x = 2*m.transform(x);
  ret += 0.0625 * noise(x, d);  x = 2*m.transform(x);
  return ret*2-1 ;
}

float4 main( idatas i )
{
  float d = cptPrimNormDist( i );

  // compute noised distance
  float2 npos = toDrawSpace(i.pos)*.1/i.noiseScale  ;
  float dn = d + i.noisePower*noiseFunction(
float3(npos,i.nbUserStroke+i.dist*0.0001), i.ndensity )  ;

  // compute attenuation

  float att = cptHardness( dn, i.hardness, i.innerSize );
  att = i.primDistanceValid ? att : 1 ;

  float alpha = saturate(i.color.a*att) ;
  float3 col = i.color.xyz ;

  return float4( col, alpha ) ;
}
```

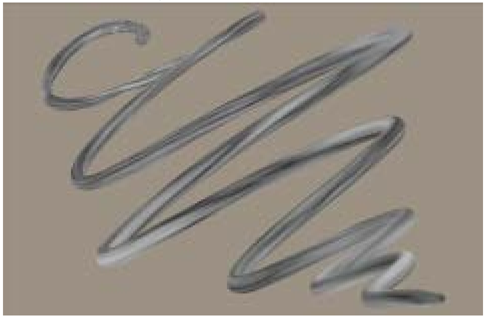
ink droplets



```
cfg{
  name="ink droplets";
  renderingTime = 30 ;
}

globals{
  float4 colA = 1 ;
  float4 colB = float4(0,0,0,1);
}

perPrim {
  float density = 0.5 ;
  {
    id = -1 ;
    uiMin = 0      ;
    uiMax = 1      ;
    uiTab = "shape"  ;
    uiFormat = percent  ;
    uiName = "density"  ;
  }
  float hardness = 0.5 ;
  {
    uiMin = 0      ;
    uiMax = 1      ;
    uiTab = "shape"  ;
    uiFormat = percent  ;
    uiName = "hardness"  ;
  }
}

float2 hash1( float3 p)
{
  float3 hscale = float3(.1031,-.1029,.1032) ;

  float3 p3 = frac( p * hscale ) ;
    p3 += dot(p3, p3.yzx + 209.191349);
    return frac( float2( (p3.x + p3.y) * p3.z,
              (p3.x + p3.z) * p3.x ) ) ;
}

float2 hash2( float2 p )
{
  float3 hscale3 = float3( .1031, .1030, .0973 ) ;

  float3 p3 = frac( float3(p.x,p.y,p.x) * hscale3) ;
    p3 += dot(p3, p3.yzx+19.19) ;
    return frac( float2( (p3.x + p3.y)*p3.z, (p3.x+p3.z)*p3.y) );
}

float voronoi( idatas i, float2 uv, float time )
```

```
{
  float2 f ;
  float2 n = decompose( uv, f ) ;

  float seed = i.nbUserStroke ;

    //----------------------------------
    // regular voronoi
    //----------------------------------
    float ret = 0 ;
    for( int j=-3; j<=3; j++ )
    {
    for( int i=-3; i<=3; i++ )
    {
      float2 g = float2( i, j );

      float2 ipos = n + g  ;

      float2 datas = hash1( float3(ipos,seed) );
      float att = lerp( 2.5, 10, pow(datas.x,1.5) );
      float move = lerp(0,1,datas.y );

      float2 o = hash2( ipos+seed );
      o = sin( time + 6.2831*o );  // animate the position


      float2 r = g + o - f;
      float d = dot(r,r);
      d = sqrt( d );

      ret += exp( -att*d ) ;  // accumulate density
    }
  }

    return ret ;
}

float4 main( idatas i )
{
  box2 b = box2FromCenterAxe( i.strokeStartPos, max( length(i.strokePos-
i.strokeStartPos), 2 ), normalizeSafe(i.strokePos-i.strokeStartPos) );
  float2 p = 4*b.toCenter( i.pos ) / b.size ;
  float density = voronoi( i, p, i.dist*0.02 ) ;

//  density = abs(density - 0.3) ;
//  density = frac( min( density*4, 40 ) );

  float cut = 1-i.density;

  float hscale = max( (1-cut) * (1-i.hardness), 0.001 );
  float smoothPos = density - cut ;
  smoothPos = saturate( smoothPos / hscale );

//  float4 col = density > (1-i.density) ? colA : colB ;
  float4 col = smoothLerp( colA, colB, smoothPos );

  return col ;
}
```

Perlin blobby noise



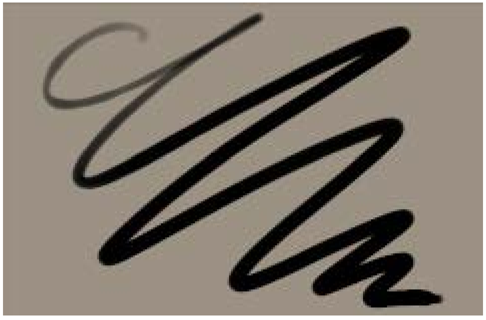
```
cfg{
  name = "Perlin blobby noise" ;
}

globals{
  colorGradient grad ;
  {
    uiTab = "color" ;
    id = 1 ;
  }
}

perPrim{
  float noiseScale=1;
  {
    uiMin = 0.5 ;
    uiMax = 10 ;
    uiName = "Scale";
    uiTab = "Noise" ;
    uiFormat = percent ;
  }
}

float3 hash( float3 p )
{
  float3 hscale = float3(.1031,-.1029,.1032) ;

  float3 p3 = frac( p * hscale ) ;
    p3 += dot(p3, p3.yzx + 209.191349);
    return -1+2*frac( float3(  (p3.x + p3.y) * p3.z,
                (p3.z + p3.y) * p3.x,
                (p3.x + p3.z) * p3.y ) );
}

// Gradient noise see https://www.shadertoy.com/view/XdXGW8
// the returned value is between [0,1]
float noise( float3 x )
{
  float3 f = frac(x);
  float3 p = decompose( x, f );
    float3 u = f*f*(3.0-2.0*f);

    float v1 = bilinearLerp(  dot( hash(p+float3(0,0,0)), f-float3(0,0,0) ), dot( 
hash(p+float3(1,0,0)), f-float3(1,0,0) ),
                dot( hash(p+float3(0,1,0)), f-float3(0,1,0) ), dot( 
hash(p+float3(1,1,0)), f-float3(1,1,0) ), u.xy ) ;

    float v2 = bilinearLerp(  dot( hash(p+float3(0,0,1)), f-float3(0,0,1) ), dot( 
hash(p+float3(1,0,1)), f-float3(1,0,1) ),
                dot( hash(p+float3(0,1,1)), f-float3(0,1,1) ), dot(
hash(p+float3(1,1,1)), f-float3(1,1,1) ), u.xy ) ;

  float ret = lerp( v1, v2, u.z )/0.707;  // normalize with the maxiumm slope
  return abs(ret);
}

// return a noise value between [0,1]
float noiseFunction( float3 x )
{
  matrix3 m  = matrix3FromPosAxes( float3(-0.8,0.63,1.03), float3(0.97,0.01,-
0.036), float3(-0.02,-.87,0.03), float3(0.04,0.024,1.01) );
  float ret = 0.5   * noise( x ) ; x = 2*m.transform(x);
  ret += 0.25 * noise(x);  x = 2*m.transform(x);
  ret += 0.125 * noise(x);  x = 2*m.transform(x);
  ret += 0.0625 * noise(x);  x = 2*m.transform(x);
  return ret ;
}

float2 cpt2DPosFromMouse( idatas i )
{
  float2 dir = normalizeSafe( i.strokePos - i.strokeStartPos ) ;
  float l = max( length( i.strokeStartPos - i.strokePos ), 1 );
  plane2 p = plane2FromNormPos( dir, i.strokeStartPos );
  plane2 p1 = plane2FromNormPos( perpendicular(dir), i.strokeStartPos );

  return 15*float2( p.getDistance(i.pos), p1.getDistance(i.pos) )/l  ;
}

float4 main( idatas i )
{
  // compute noise value
  float2 npos = 1/i.noiseScale*cpt2DPosFromMouse( i ) ;
  float nv = noiseFunction( float3(npos,i.nbUserStroke+i.dist*0.01) );

  float4 col = grad.sample(nv) ;

  return col ;
}
```

Perlin blobby noise_01

```
cfg{
  name = "Perlin blobby noise" ;
}

globals{
  colorGradient grad ;
  {
    uiTab = "color" ;
    id = 1 ;
  }
}

perPrim{
  float noiseScale=1;
  {
    uiMin = 0.5 ;
    uiMax = 10 ;
    uiName = "Scale";
    uiTab = "Noise" ;
    uiFormat = percent ;
  }
}

float3 hash( float3 p )
{
  float3 hscale = float3(.1031,-.1029,.1032) ;

  float3 p3 = frac( p * hscale ) ;
    p3 += dot(p3, p3.yzx + 209.191349);
    return -1+2*frac( float3(  (p3.x + p3.y) * p3.z,
                (p3.z + p3.y) * p3.x,
                (p3.x + p3.z) * p3.y ) );
}

// Gradient noise see https://www.shadertoy.com/view/XdXGW8
// the returned value is between [0,1]
float noise( float3 x )
{
  float3 f = frac(x);
  float3 p = decompose( x, f );
    float3 u = f*f*(3.0-2.0*f);

    float v1 = bilinearLerp(  dot( hash(p+float3(0,0,0)), f-float3(0,0,0) ), dot(
hash(p+float3(1,0,0)), f-float3(1,0,0) ),
                dot( hash(p+float3(0,1,0)), f-float3(0,1,0) ), dot(
hash(p+float3(1,1,0)), f-float3(1,1,0) ), u.xy ) ;
```

```
    float v2 = bilinearLerp(  dot( hash(p+float3(0,0,1)), f-float3(0,0,1) ), dot(
hash(p+float3(1,0,1)), f-float3(1,0,1) ),
                dot( hash(p+float3(0,1,1)), f-float3(0,1,1) ), dot(
hash(p+float3(1,1,1)), f-float3(1,1,1) ), u.xy ) ;

  float ret = lerp( v1, v2, u.z )/0.707;  // normalize with the maxiumm slope
  return abs(ret);
}

// return a noise value between [0,1]
float noiseFunction( float3 x )
{
  matrix3 m  = matrix3FromPosAxes( float3(-0.8,0.63,1.03), float3(0.97,0.01,-
0.036), float3(-0.02,-.87,0.03), float3(0.04,0.024,1.01) );
  float ret = 0.5   * noise( x ) ; x = 2*m.transform(x);
  ret += 0.25 * noise(x);  x = 2*m.transform(x);
  ret += 0.125 * noise(x);  x = 2*m.transform(x);
  ret += 0.0625 * noise(x);  x = 2*m.transform(x);
  return ret ;
}

float2 cpt2DPosFromMouse( idatas i )
{
  float2 dir = normalizeSafe( i.strokePos - i.strokeStartPos ) ;
  float l = max( length( i.strokeStartPos - i.strokePos ), 1 );
  plane2 p = plane2FromNormPos( dir, i.strokeStartPos );
  plane2 p1 = plane2FromNormPos( perpendicular(dir), i.strokeStartPos );

  return 15*float2( p.getDistance(i.pos), p1.getDistance(i.pos) )/l  ;
}

float4 main( idatas i )
{
  // compute noise value
  float2 npos = 1/i.noiseScale*cpt2DPosFromMouse( i ) ;
  float nv = noiseFunction( float3(npos,i.nbUserStroke+i.dist*0.01) );

  float4 col = grad.sample(nv) ;

  return col ;
}
```

Perlin noise (error)



```
cfg{
  name = "Perlin noise" ;
}

globals{
  colorGradient grad ;
  {
    uiTab = "color" ;
    id = 1 ;
  }
  uiTab Dithering ;
  {
    row = 2 ;
  }
}

perPrim{
  float opacity=1;
  {
    uiMax = 1 ;
    uiTab = "color" ;
    uiFormat = percent ;
    id = 0 ;
  }
  float noisePower=0.1;
  {
    uiMax = 1 ;
    uiName = "Power";
    uiTab = "Noise" ;
    uiFormat = percent ;
  }
  float noiseScale=1;
  {
    uiMax = 1 ;
    uiName = "Scale";
    uiTab = "Noise" ;
    uiFormat = percent ;
  }
  float ditherPower=0.1;
  {
    uiMax = 1 ;
    uiName = "Power";
    uiTab = "Dithering" ;
    uiFormat = percent ;
  }
  float ditherScale=1;
  {
    uiMax = 1 ;
    uiName = "Scale";
    uiTab = "Dithering" ;
    uiFormat = percent ;
  }
}

float hash( float3 p )
{
  float hscale = .1031 ;

  float3 p3 = frac( p * hscale ) ;
    p3 += dot(p3, p3.yzx + 209.191349);
    return frac( (p3.x + p3.y) * p3.z );
}

float noise( float3 x )
{
  float3 f ;
  float3 p = decompose( x, f );
    f = f*f*(3.0-2.0*f);

    float v1 = bilinearLerp(  hash(p+float3(0,0,0)), hash(p+float3(1,0,0)),
                hash(p+float3(0,1,0)), hash(p+float3(1,1,0)), f.xy ) ;

    float v2 = bilinearLerp(  hash(p+float3(0,0,1)), hash(p+float3(1,0,1)),
                hash(p+float3(0,1,1)), hash(p+float3(1,1,1)), f.xy ) ;

  return lerp( v1, v2, f.z );
}

float noiseFunction( float3 x )
{
  return  0.5   * noise( x ) +
      0.25  * noise( x*2.02 ) +
      0.125 * noise( x*4.509 ) +
      0.0625* noise( x*8.1580 ) ;
}

float4 main( idatas i )
{
  float2 dir = normalizeSafe( i.strokePos - i.strokeStartPos ) ;
  float l = length( i.strokeStartPos - i.strokePos );
  plane2 p = plane2FromNormPos( dir, i.strokeStartPos );
  plane2 p1 = plane2FromNormPos( perpendicular(dir), i.strokeStartPos );

  // compute noise value
  float2 npos = i.noiseScale*20*float2( p.getDistance(i.pos),
p1.getDistance(i.pos) )/l  ;
  float nv = i.noisePower*0.5*(1-2*noiseFunction(
float3(npos,i.nbUserStroke+i.dist*0.003) ));

  // compute dithering
  float2 nposDither = i.ditherScale*float2( p.getDistance(i.pos),
p1.getDistance(i.pos) )  ;
  float nvDither = i.ditherPower*(1-2*noiseDither(
float3(nposDither,i.nbUserStroke) ));

ERROR: Undeclared identifier ‘noiseDither’

  float d = saturate( nvDither + nv + p.getDistance( i.pos ) / l );
  float4 col = grad.sample( d ) ;
  col.a *= i.opacity ;
  // output somehting only if with ahve more than 1 pixel to compute the plane
  col.a = (l>1) ? col.a : 0 ;


  return col ;
}
```

Perlin noise scatter

```
cfg{
  name = "Perlin noise scatter" ;
  blendEx = true ;
  blendDefault = replace ;
  samplingLayerMaxOffset = 64 ;
  renderingTime = 30 ;
}

globals{
  uiTab Noise ;
  {
    row = 3;
    id = 1;
  }
}

perPrim{
  float noiseScale=1;
  {
    uiMin = 0.01 ;
    uiMax = 2 ;
    uiName = "Scale";
    uiTab = "Noise" ;
    uiFormat = percent ;
  }
  float scatterPower=0.5 ;
  {
    uiMin = 0 ;
    uiMax = 1 ;
    uiName = "Power";
    uiTab = "Noise" ;
    uiFormat = percent ;
  }
}

float3 hash( float3 p )
{
  float3 hscale = float3(.1031,-.1029,.1032) ;

  float3 p3 = frac( p * hscale ) ;
    p3 += dot(p3, p3.yzx + 209.191349);
    return -1+2*frac( float3(  (p3.x + p3.y) * p3.z,
                (p3.z + p3.y) * p3.x,
                (p3.x + p3.z) * p3.y ) );
}

// Gradient noise see https://www.shadertoy.com/view/XdXGW8
// the returned value is between [-1,1]
float noise( float3 x )
{
  float3 f = frac(x);
  float3 p = decompose( x, f );
    float3 u = f*f*(3.0-2.0*f);

    float v1 = bilinearLerp(  dot( hash(p+float3(0,0,0)), f-float3(0,0,0) ), dot(
hash(p+float3(1,0,0)), f-float3(1,0,0) ),
                dot( hash(p+float3(0,1,0)), f-float3(0,1,0) ), dot(
hash(p+float3(1,1,0)), f-float3(1,1,0) ), u.xy ) ;

    float v2 = bilinearLerp(  dot( hash(p+float3(0,0,1)), f-float3(0,0,1) ), dot(
hash(p+float3(1,0,1)), f-float3(1,0,1) ),
                dot( hash(p+float3(0,1,1)), f-float3(0,1,1) ), dot(
hash(p+float3(1,1,1)), f-float3(1,1,1) ), u.xy ) ;

  return lerp( v1, v2, u.z )/0.707;  // normalize with the maxiumm slope
}

// Gradient noise see https://www.shadertoy.com/view/XdXGW8
// the returned value is 2D vector between [-1,1]
float2 noise2( float3 x )
{
  return   float2( noise(x), noise(-x.yzx+float3(10.321,10.8798,11.9842)) ) ;
}

// return a noise value between [-1,1]
float2 noiseFunction( float3 x )
{
  matrix3 m  = matrix3FromPosAxes( float3(-0.8,0.63,1.03), float3(0.97,0.01,-
0.036), float3(-0.02,-.87,0.03), float3(0.04,0.024,1.01) );
  float2 ret = 0.5   * noise2( x ) ; x = 2*m.transform(x);
  ret += 0.25 * noise2(x);  x = 2*m.transform(x);
  ret += 0.125 * noise2(x);  x = 2*m.transform(x);
  ret += 0.0625 * noise2(x);  x = 2*m.transform(x);
  return ret ;
}

float2 cpt2DPosFromMouse( idatas i )
{
  float2 dir = normalizeSafe( i.strokePos - i.strokeStartPos ) ;
  float l = max( length( i.strokeStartPos - i.strokePos ), 1 );
  plane2 p = plane2FromNormPos( dir, i.strokeStartPos );
  plane2 p1 = plane2FromNormPos( perpendicular(dir), i.strokeStartPos );

  return 15*float2( p.getDistance(i.pos), p1.getDistance(i.pos) )/l  ;
}

float4 main( idatas i )
{
  // get the current background color
  float4 colBack = bottomLayer.pointSample( i, i.pos );

  // compute noise value
  float2 npos = 1/i.noiseScale*cpt2DPosFromMouse( i ) ;
  float2 nv = noiseFunction( float3(npos,i.nbUserStroke+i.dist*0.001) );

  // compute the current pixel displacement
  float mdisplace = i.scatterPower * samplingLayerMaxOffset;
  float2  offset = nv*mdisplace ;

  // Get the displaced color
  float4 smpl = bottomLayer.bilinearSample( i, i.pos + offset );
  smpl = smoothLerp( colBack, smpl, saturate( mdisplace ) );  // reset to a
pointSampling for little displacement

  float4 col = smpl ;
  col.xy = smpl ;

  return col ;
}
```

Perlin sin noise

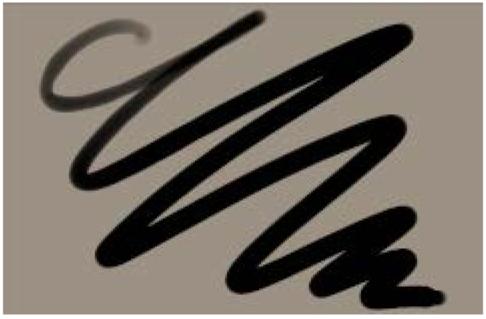
```
cfg{
  name = "Perlin sin noise" ;
}

globals{
  colorGradient grad ;
  {
    uiTab = "color" ;
    id = 1 ;
  }
}

perPrim{
  float noiseScale=1;
  {
    uiMin = 0.5 ;
    uiMax = 10 ;
    uiName = "Scale";
    uiTab = "Noise" ;
    uiFormat = percent ;
  }
  float turbScale=0.5;
  {
    uiMax = 1 ;
    uiName = "Turbulence";
    uiTab = "Noise" ;
    uiFormat = percent ;
  }
  float freq=0.5;
  {
    uiMax = 4 ;
    uiName = "Freq";
    uiTab = "Noise" ;
    uiFormat = percent ;
  }
}

float3 hash( float3 p )
{
  float3 hscale = float3(.1031,-.1029,.1032) ;

  float3 p3 = frac( p * hscale ) ;
    p3 += dot(p3, p3.yzx + 209.191349);
    return -1+2*frac( float3(  (p3.x + p3.y) * p3.z,
                (p3.z + p3.y) * p3.x,
                (p3.x + p3.z) * p3.y ) );
}

// Gradient noise see https://www.shadertoy.com/view/XdXGW8
// the returned value is between [0,1]
```

```
float noise( float3 x )
{
  float3 f = frac(x);
  float3 p = decompose( x, f );
    float3 u = f*f*(3.0-2.0*f);

    float v1 = bilinearLerp(  dot( hash(p+float3(0,0,0)), f-float3(0,0,0) ), dot(
hash(p+float3(1,0,0)), f-float3(1,0,0) ),
                dot( hash(p+float3(0,1,0)), f-float3(0,1,0) ), dot(
hash(p+float3(1,1,0)), f-float3(1,1,0) ), u.xy ) ;

    float v2 = bilinearLerp(  dot( hash(p+float3(0,0,1)), f-float3(0,0,1) ), dot(
hash(p+float3(1,0,1)), f-float3(1,0,1) ),
                dot( hash(p+float3(0,1,1)), f-float3(0,1,1) ), dot(
hash(p+float3(1,1,1)), f-float3(1,1,1) ), u.xy ) ;

  float ret = lerp( v1, v2, u.z )/0.707;  // normalize with the maxiumm slope
  return abs(ret);
}

// return a noise value between [0,1]
float noiseFunction( float3 x )
{
  matrix3 m  = matrix3FromPosAxes( float3(-0.8,0.63,1.03), float3(0.97,0.01,-
0.036), float3(-0.02,-.87,0.03), float3(0.04,0.024,1.01) );
  float ret = 0.5   * noise( x ) ; x = 2*m.transform(x);
  ret += 0.25 * noise(x);  x = 2*m.transform(x);
  ret += 0.125 * noise(x);  x = 2*m.transform(x);
  ret += 0.0625 * noise(x);  x = 2*m.transform(x);
  return ret ;
}

float2 cpt2DPosFromMouse( idatas i )
{
  float2 dir = normalizeSafe( i.strokePos - i.strokeStartPos ) ;
  float l = max( length( i.strokeStartPos - i.strokePos ), 1 );
  plane2 p = plane2FromNormPos( dir, i.strokeStartPos );
  plane2 p1 = plane2FromNormPos( perpendicular(dir), i.strokeStartPos );

  return 15*float2( p.getDistance(i.pos), p1.getDistance(i.pos) )/l  ;
}

float4 main( idatas i )
{
  // compute noise value
  float2 npos = 1/i.noiseScale*cpt2DPosFromMouse( i ) ;
  float nv = 0.5+0.5*sin( npos.x*i.freq + TWOPI*i.turbScale*noiseFunction(
float3(npos,i.nbUserStroke+i.dist*0.01) ) );

  float4 col = grad.sample(nv) ;

  return col ;
}
```

pointShadow

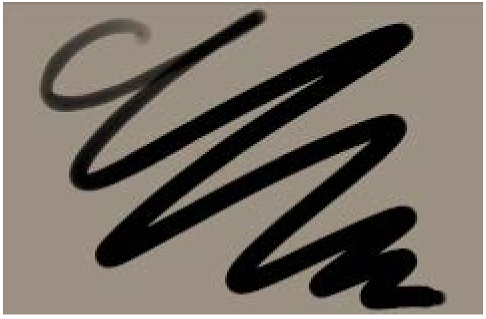
```
cfg{name="pointShadow";}

perPrim {

  float size = 0.5 ;
  {
    uiMin = 0           ;
    uiMax = 1           ;
    uiTab = "shape"         ;
    uiFormat = percent       ;
    uiName = "inner size"   ;
  }

  float hardness = 0.5 ;
  {
    uiMin = 0           ;
    uiMax = 1           ;
    uiTab = "shape"         ;
    uiFormat = percent       ;
  }

  float salpha = 0.275 ;
  {
    uiMin = 0       ;
    uiMax = 1       ;
    uiTab = "shadow"   ;
    uiFormat = percent   ;
  }

  float ssize = 0.39 ;
  {
    uiMin = 0       ;
    uiMax = 1       ;
    uiTab = "shadow"   ;
    uiFormat = percent   ;
  }

  float shardness = 0.45 ;
  {
    uiMin = 0       ;
    uiMax = 1       ;
    uiTab = "shadow"   ;
    uiFormat = percent   ;
    uiName = "hardness"   ;
  }

  float2 soffset = float2(-0.8,0.13) ;
  {
    uiTab = "shadow"     ;
    uiName = "offset 2D"   ;
    uiEditor = graph    ;
    uiYAxisInvert = true  ;
    power = 1        ;
  }


}

float cirlceGauss( float2 p, float r, float hardness )
{
  float d = max( -(length( p ) - r), 0 );
//  return d;

  float softness = pow( 1 - hardness,2)  ;  // permet de donner un feedback plus
lineaire sur ce paramètre
  d /= r*softness + 0.0001          ;  // normalise la distance en fct de la taile
de la primitive

  return 1 - exp( -d );  // courbe d'attenuation en fct de la distance
}

float4 main( idatas i )
{
  float2 uv = i.primBox.toUpperLeftNorm( i.pos )*2-1 ;

  float shadow = i.salpha * i.color.a * cirlceGauss( uv - i.soffset,
i.size*(1+i.ssize), i.hardness * i.shardness );
  float4 colShadow = float4(0,0,0,shadow );

  float calpha = cirlceGauss( uv, i.size, i.hardness );
  float4 col = i.color ;
  col *= calpha ;

  col = blendNormal( colShadow, col );

  return col ;
}
```

prim texture alpha invy



```
cfg{
  name="prim texture alpha invy";
  renderingTime = 20 ;
  samplerDefault ;
  {
    adressU = clamp;
    adressV = clamp;
  }
}

globals{
  texture mat;
  {
    uiTab = "Texture" ;
  }
}

float4 main( idatas i )
{
  float2 p = i.primBox.toUpperLeftNorm( i.pos ) ;

  // compute alpha from the Luminance of a Texture
  float4 cmat = mat.sample( float2( p.x, 1-p.y) )  ;
  cmat = mat.isEmpty ? 1 : cmat  ;

  float alphaTex = cmat.a ;

  float4 col = i.color ;
  col.a *= alphaTex ;

  return col ;
}
```

prim texture alpha Luminance

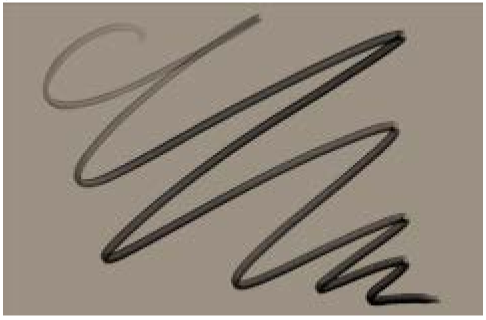


```
cfg{
  name="prim texture alpha Luminance";
}

globals{
  texture mat;
}

float4 main( idatas i )
{
  float2 p = i.primBox.toUpperLeftNorm( i.pos ) ;

  float4 cmat = mat.sample( p )  ;
  cmat = mat.isEmpty ? 1 : cmat  ;

  float4 col = i.color ;
  col.a *= luminance( cmat ) ;

  return  col;
}
```

prim texture alpha power

```
cfg{
  name="prim texture alpha power";
  renderingTime = 10 ;
}

globals{
  uiTab "Texture" ;

  texture mat;
  {
    uiTab = "Texture" ;
  }
}

perPrim {
  float hardness = 0.5 ;
  {
    uiMin = 0       ;
    uiMax = 1       ;
    uiTab = "Texture"  ;
    uiFormat = percent  ;
    uiName = "alphaPower"   ;
  }
}

float4 main( idatas i )
{
  float2 p = i.primBox.toUpperLeftNorm( i.pos ) ;

  // compute alpha from the Luminance of a Texture
  float4 cmat = mat.sample( p )  ;
  cmat = mat.isEmpty ? 1 : cmat  ;

  float alphaTex = cmat.a ;

  alphaTex = pow( alphaTex, lerp( 1, 8, i.hardness ) );


  float alpha = saturate(i.color.a*alphaTex) ;
  float3 col = i.color.xyz ;

  return float4( col, alpha ) ;
}
```

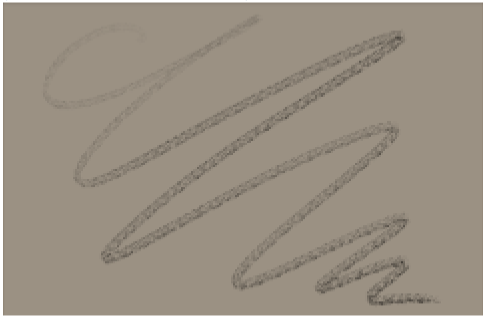
primBubbleNoise



```
cfg{name="primBubbleNoise";}

globals{

  float fnbStep0 = 4 ;
  {
    uiMin = 3         ;
    uiMax = 64         ;
    uiTab = "bubble 0"    ;
    uiFormat = integer    ;
    uiName = "angular step"   ;
    id = 2            ;
  }

  float fnbStep1 = 8 ;
  {
    uiMin = 3         ;
    uiMax = 64         ;
    uiTab = "bubble 1"    ;
    uiFormat = integer    ;
    uiName = "angular step"   ;
    id = 2            ;
  }

  float tscale0 = 1 ;
  {
    uiMin = 0         ;
    uiMax = 10         ;
    uiTab = "bubble 0"    ;
    uiName = "time scale"   ;
    id = 120          ;
  }

  float tscale1 = 1 ;
  {
    uiMin = 0         ;
    uiMax = 10         ;
    uiTab = "bubble 1"    ;
    uiName = "time scale"   ;
    id = 120          ;
  }
}

perPrim {

  float ampl0 = 0.5 ;
  {
    uiMin = 0       ;
    uiMax = 1       ;
    uiTab = "bubble 0"   ;
    uiFormat = percent  ;
    uiName = "noise amplitude"  ;
    id = 0        ;
  }

  float ampl1 = 0.25 ;
  {
    uiMin = 0      ;
    uiMax = 1      ;
    uiTab = "bubble 1"  ;
    uiFormat = percent  ;
    uiName = "noise amplitude"  ;
    id = 0        ;
  }

  float aoff0 = 0.5 ;
  {
    uiMin = 0      ;
    uiMax = 1      ;
    uiTab = "bubble 0"  ;
    uiFormat = percent  ;
    uiName = "radius"  ;
    id = 1        ;
  }

  float aoff1 = 0.25 ;
  {
    uiMin = 0      ;
    uiMax = 1      ;
    uiTab = "bubble 1"  ;
    uiFormat = percent  ;
    uiName = "radius"  ;
    id = 1        ;
  }

  float2 coff0 = 0.5 ;
  {
    uiTab = "bubble 0"    ;
    uiName = "offset 2D"  ;
    uiEditor = angleDist  ;
    power = 0 ;
  }

  float2 coff1 = -0.5 ;
  {
    uiTab = "bubble 1"    ;
    uiName = "offset 2D"  ;
    uiEditor = angleDist ;
    power = 0 ;
  }

}

float hash1( float2 p)
{
  float hscale = .1031 ;

  float3 p3 = frac( float3(p.x,p.y,p.x) * hscale ) ;
    p3 += dot(p3, p3.yzx + 19.19);
    return frac( (p3.x + p3.y) * p3.z );
```

```
}

float noise( float2 p, float t )
{
  float f ;
  float ip = decompose( p.x, f );
  float ip1 = mod( ip+1, p.y ) ;  // fait boucler en fct du maxi passé en y

  float ft ;
  float it = decompose( t, ft );
  float it1 = it+1 ;

  float a0 = smoothLerp( hash1(float2(ip,it)), hash1(float2(ip1,it)), f );
  float a1 = smoothLerp( hash1(float2(ip,it1)), hash1(float2(ip1,it1)), f );

  return smoothLerp( a0, a1, ft );
}

// return 1 if we are in the AngleCurve shape
float AngleCurveShape( float2 pos, float2 offset, float ampl, float aoffset, float
time, float fnbStep, float seed )
{
  float2 sph = toSpherical( pos-offset );

  float angle = mod( sph.x + seed , TWOPI );
  float nbStep = floor(fnbStep) ;
  angle = nbStep * angle / TWOPI    ;

  float displ = 0.5*noise(    float2(angle, nbStep), time )
        + 0.25*noise(  2*float2(angle, nbStep), time )
        + 0.125*noise(  4*float2(angle, nbStep), time )
        + 0.0625*noise( 8*float2(angle, nbStep), time )  ;

  float p  = aoffset + ampl*(2*displ-1) ;


  return sph.y > p ? 0 : 1 ;
}

float4 main( idatas i )
{
  float2 uv = i.primBox.toUpperLeftNorm( i.pos )*2-1 ;

  float seed = i.nbUserStroke * 2.23881;

  float shape0 = AngleCurveShape( uv, i.coff0, i.ampl0, i.aoff0, i.time*tscale0 +
seed, fnbStep0, seed );
  float shape1 = AngleCurveShape( uv, i.coff1, i.ampl1, i.aoff1, i.time*tscale1 +
seed*0.63798, fnbStep1, seed*1.205 );

  float gshape = shape0 + shape1 ;

  float alpha = gshape == 1 ? 1 : 0 ;

  float4 col = i.color ;
  col.a *= alpha ;

  return col ;
}
```

primBlubbleNoiseSpread



```
cfg{
  name="primBubbleNoiseSpread";
  renderingTime = 10 ;
}

globals{

  float noiseScale = 1 ;
  {
    uiTab = "Noise" ;
    uiName = "Scale"  ;
    uiFormat = percent ;
    uiMax = 1 ;
  }

  float fnbStep0 = 4 ;
  {
    uiMin = 3         ;
    uiMax = 64         ;
    uiTab = "bubble 0"    ;
    uiFormat = integer    ;
    uiName = "angular step"   ;
    id = 2            ;
  }

  float fnbStep1 = 8 ;
  {
    uiMin = 3         ;
    uiMax = 64         ;
    uiTab = "bubble 1"    ;
    uiFormat = integer    ;
    uiName = "angular step"   ;
    id = 2            ;
  }

  float tscale0 = 1 ;
  {
    uiMin = 0         ;
    uiMax = 10         ;
    uiTab = "bubble 0"    ;
    uiName = "time scale"  ;
    id = 120          ;
  }

  float tscale1 = 1 ;
  {
    uiMin = 0         ;
    uiMax = 10         ;
    uiTab = "bubble 1"    ;
    uiName = "time scale"  ;
    id = 120        ;
  }
}

perPrim
{
  float ndensity = 0.5 ;
  {
    uiTab = "Noise" ;
    uiName = "Density"  ;
    uiFormat = percent ;
    uiMax = 1 ;
  }

  float densityHardness = 0.15 ;
  {
    uiName = "Hardness";
    uiMax = 1 ;
    uiTab = "Noise" ;
  }
  float ampl0 = 0.5 ;
  {
    uiMin = 0      ;
    uiMax = 1      ;
    uiTab = "bubble 0"  ;
    uiFormat = percent  ;
    uiName = "noise amplitude"  ;
    id = 0        ;
  }

  float ampl1 = 0.25 ;
  {
    uiMin = 0      ;
    uiMax = 1      ;
    uiTab = "bubble 1"  ;
    uiFormat = percent  ;
    uiName = "noise amplitude"  ;
    id = 0        ;
  }

  float aoff0 = 0.5 ;
  {
    uiMin = 0      ;
    uiMax = 1      ;
    uiTab = "bubble 0"  ;
    uiFormat = percent  ;
    uiName = "radius"  ;
    id = 1        ;
  }

  float aoff1 = 0.25 ;
  {
    uiMin = 0      ;
    uiMax = 1      ;
    uiTab = "bubble 1"  ;
    uiFormat = percent  ;
    uiName = "radius"  ;
    id = 1        ;
  }

  float2 coff0 = 0.5 ;
  {
    uiTab = "bubble 0"    ;
    uiName = "offset 2D"  ;
    uiEditor = angleDist  ;
    power = 0 ;
  }

  float2 coff1 = -0.5 ;
  {
    uiTab = "bubble 1"    ;
    uiName = "offset 2D"  ;
    uiEditor = angleDist ;
    power = 0 ;
  }

}

float hash1( float2 p)
{
  float hscale = .1031 ;

  float3 p3 = frac( float3(p.x,p.y,p.x) * hscale ) ;
    p3 += dot(p3, p3.yzx + 19.19);
    return frac( (p3.x + p3.y) * p3.z );
}

float noise( float2 p, float t )
{
  float f ;
  float ip = decompose( p.x, f );
  float ip1 = mod( ip+1, p.y ) ;  // fait boucler en fct du maxi passé en y

  float ft ;
  float it = decompose( t, ft );
  float it1 = it+1 ;

  float a0 = smoothLerp( hash1(float2(ip,it)), hash1(float2(ip1,it)), f );
  float a1 = smoothLerp( hash1(float2(ip,it1)), hash1(float2(ip1,it1)), f );

  return smoothLerp( a0, a1, ft );
}

// return 1 if we are in the AngleCurve shape
float AngleCurveShape( float2 pos, float2 offset, float ampl, float aoffset, float
time, float fnbStep, float seed )
{
  float2 sph = toSpherical( pos-offset );

  float angle = mod( sph.x + seed , TWOPI );
  float nbStep = floor(fnbStep) ;
  angle = nbStep * angle / TWOPI    ;

  float displ = 0.5*noise(    float2(angle, nbStep), time )
        + 0.25*noise(  2*float2(angle, nbStep), time )
        + 0.125*noise(  4*float2(angle, nbStep), time )
        + 0.0625*noise( 8*float2(angle, nbStep), time )  ;

  float p  = aoffset + ampl*(2*displ-1) ;


  return sph.y > p ? 0 : 1 ;
```

```
}

float hash( float3 p )
{
  float hscale = .1031 ;

  float3 p3 = frac( p * hscale ) ;
    p3 += dot(p3, p3.yzx + 19.19);
    return frac( (p3.x + p3.y) * p3.z );
}

float adjustDensity( float r, float d )
{
  d = 1-d ;
  float a = clamp( (d-0.5)*2, 0, 1 );
  float b = clamp( d*2, 0, 1 );

  r = (r-a) / max( b-a, 0.001) ;
  r = saturate( r );

  return r ;
}

float hashD( float3 p, float d )
{
  return adjustDensity( hash(p), d );
}

float hardnessCpt( idatas i, float h )
{
  float sm = vmin( i.primBox.getSize() )  ;
  float d = max( -i.primDistance, 0 )    ;

  float softness = pow( 1 - h,2)  ;
  d /= sm*softness + 0.001    ;

  d = 1 - exp( -d );

  return i.primDistanceValid ? d : 1 ;
}

float cptPrimNoise( idatas i, float time, float density )
{
  float2 p = i.primBox.toUpperLeftNorm( i.pos )*2 - 1 ;
  p = floor( p*i.primBox.getSize()*noiseScale );

  return hashD( float3( p, time ), density );
}

float4 main( idatas i )
{
  float2 uv = i.primBox.toUpperLeftNorm( i.pos )*2-1 ;

  float seed = i.nbUserStroke * 2.23881;

  float shape0 = AngleCurveShape( uv, i.coff0, i.ampl0, i.aoff0, i.time*tscale0 +
seed, fnbStep0, seed );
  float shape1 = AngleCurveShape( uv, i.coff1, i.ampl1, i.aoff1, i.time*tscale1 +
seed*0.63798, fnbStep1, seed*1.205 );

  float gshape = shape0 + shape1 ;
  float alpha = gshape == 1 ? 1 : 0 ;

  float density = i.ndensity * hardnessCpt( i, i.densityHardness );
  alpha *= cptPrimNoise( i, i.dist*0.001, density );

  float4 col = i.color ;
  col.a *= alpha ;

  return col ;
}
```

primHardness

```
cfg{name = "primHardness";
}

perPrim{
  float densityHardness = 0.15 ;
  {
    uiName = "Hardness";
    uiMax = 1 ;
    uiTab = "Shape" ;
  }

}

float hardnessCpt( idatas i, float h )
{
  float sm = vmin( i.primBox.getSize() )  ;
  float d = max( -i.primDistance, 0 )    ;

  float softness = pow( 1 - h,2)  ;
  d /= sm*softness + 0.001    ;

  d = 1 - exp( -d );

  return i.primDistanceValid ? d : 1 ;
}

float4 main( idatas i )
{
  float alpha = hardnessCpt( i, i.densityHardness );

  float4 col = i.color ;
  col.a *= alpha ;

  return col ;
}
```

primsimplenoise (error)

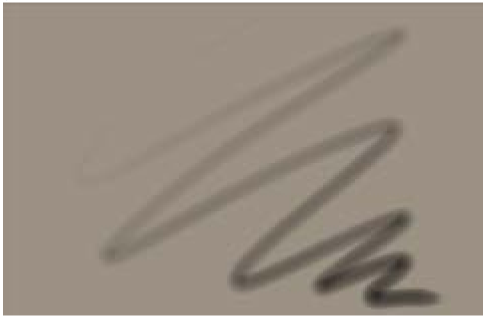
```
cfg{
  name = "primSimpleNoise";
  renderingTime = 5 ;
}

globals{
  float noiseScale = 1 ;
  {
    uiTab = "Noise" ;
    uiName = "Scale"  ;
    uiFormat = percent ;
    uiMax = 1 ;
  }
}

perPrim{
  float ndensity = 0.5 ;
  {
    uiTab = "Noise" ;
    uiName = "Density"  ;
    uiFormat = percent ;
    uiMax = 1 ;
  }

  float densityHardness = 0.15 ;
  {
    uiName = "Hardness";
    uiMax = 1 ;
    uiTab = "Noise" ;
  }

}

float hash( float3 p )
{
  float hscale = .1031 ;

  float3 p3 = frac( p * hscale ) ;
    p3 += dot(p3, p3.yzx + 19.19);
    return frac( (p3.x + p3.y) * p3.z );
}

float adjustDensity( float r, float d )
{
  d = 1-d ;
  float a = clamp( (d-0.5)*2, 0, 1 );
  float b = clamp( d*2, 0, 1 );

  r = (r-a) / max( b-a, 0.001) ;
  r = saturate( r );
  return r ;
}

float hashD( float3 p, float d )
{
  return adjustDensity( hash(p), d );
}

float hardnessCpt( idatas i, float h )
{
  float sm = vmin( i.primBox.getSize() )  ;
  float d = max( -i.primDistance, 0 )    ;

  float softness = pow( 1 - h,2)  ;
  d /= sm*softness + 0.001    ;

  d = 1 - exp( -d );

  return i.primDistanceValid ? d : 1 ;
}

float cptPrimNoise( idatas i, float time, float density )
{
  float2 p2 = i.primBox.toUpperLeftNorm( i.pos )*2 - 1 ;
  p2 = floor( p*i.primBox.getSize()*noiseScale );

ERROR: Undeclared identifier ‘p’

  float3 x = float3( p2, time );
  float3 f ;
  float3 p = decompose( x, f );
    f = f*f*(3.0-2.0*f);

    float v1 = bilinearLerp(  hashD(p+float3(0,0,0),d), hashD(p+float3(1,0,0),d),
                hashD(p+float3(0,1,0),d), hashD(p+float3(1,1,0),d), f.xy ) ;

    float v2 = bilinearLerp(  hashD(p+float3(0,0,1),d), hashD(p+float3(1,0,1),d),
                hashD(p+float3(0,1,1),d), hashD(p+float3(1,1,1),d), f.xy ) ;


  return lerp( v1, v2, f.z );
}

float4 main( idatas i )
{
  float density = i.ndensity * hardnessCpt( i, i.densityHardness );
  float alpha = cptPrimNoise( i, i.dist*0.001, density );

  float4 col = i.color ;
  col.a *= alpha ;

  return col ;
}
```

primSimpleNoise_01

```
cfg{
  name = "primSimpleNoise";
  renderingTime = 5 ;
}

globals{

  float noiseScale = 1 ;
  {
    uiTab = "Noise" ;
    uiName = "Scale"  ;
    uiFormat = percent ;
    uiMax = 1 ;
    id = 100 ;
  }

  texture mat ;
  {
    uiTab = "Noise" ;
    uiName = "Mat" ;
    id = -2 ;
  }

}

perPrim{
  float ndensity = 0.5 ;
  {
    uiTab = "Noise" ;
    uiName = "Density"  ;
    uiFormat = percent ;
    uiMax = 1 ;
    id = -1 ;
  }

  float densityHardness = 0.15 ;
  {
    uiName = "Hardness";
    uiMax = 1 ;
    uiTab = "Noise" ;
  }

}

float hash( float3 p )
{
  float hscale = .1031 ;

  float3 p3 = frac( p * hscale ) ;
```

```
    p3 += dot(p3, p3.yzx + 19.19);
    return frac( (p3.x + p3.y) * p3.z );
}

float adjustDensity( float r, float d )
{
  d = 1-d ;
  float a = clamp( (d-0.5)*2, 0, 1 );
  float b = clamp( d*2, 0, 1 );

  r = (r-a) / max( b-a, 0.001) ;
  r = saturate( r );

  return r ;
}

float hashD( float3 p, float d )
{
  return adjustDensity( hash(p), d );
}

float hardnessCpt( idatas i, float h )
{
  float sm = vmin( i.primBox.getSize() )  ;
  float d = max( -i.primDistance, 0 )    ;

  float softness = pow( 1 - h,2)  ;
  d /= sm*softness + 0.001    ;

  d = 1 - exp( -d );

  return i.primDistanceValid ? d : 1 ;
}

float cptPrimNoise( idatas i, float time, float density )
{
  float2 p = i.primBox.toUpperLeftNorm( i.pos )*2 - 1 ;
  p = floor( p*i.primBox.getSize()*noiseScale );

  return hashD( float3( p, time ), density );
}

float4 main( idatas i )
{
  float2 p = i.primBox.toUpperLeftNorm( i.pos ) ;

  float4 cmat = mat.sample( p )  ;
  cmat = mat.isEmpty ? 1 : cmat  ;
  float dmat = luminance( cmat )  ;

  float density = i.ndensity * dmat * hardnessCpt( i, i.densityHardness );
  float alpha = cptPrimNoise( i, i.dist*0.001, density );

  float4 col = i.color ;
  col.a *= alpha ;

  return col ;
}
```

primSmoothNoise



```
cfg{
  name = "primSmoothNoise";
  renderingTime = 5 ;
}

globals{
  float noiseScale = 2 ;
  {
    uiName = "Scale";
    uiMin = 0.25 ;
    uiMax = 100 ;
    uiTab = "Noise" ;
  }

  float evoSpeed = 0.05 ;
  {
    uiName = "EvoSpeed";
    uiMax = 1 ;
    uiTab = "Noise" ;
  }

}
```

radial gradient

```
cfg{
  name = "radial gradient" ;
}

globals{
  colorGradient grad ;
  {
    uiTab = "color" ;
    id = 1 ;
  }
}

perPrim{
  float opacity=1;
  {
    uiMax = 1 ;
    uiTab = "color" ;
    uiFormat = percent ;
    id = 0 ;
  }
  float noisePower=0.1;
  {
    uiMax = 1 ;
    uiName = "Power";
    uiTab = "Dithering" ;
    uiFormat = percent ;
  }
  float noiseScale=1;
  {
    uiMax = 1 ;
    uiName = "Scale";
    uiTab = "Dithering" ;
    uiFormat = percent ;
  }
}

float hash( float3 p )
{
  float hscale = .1031 ;

  float3 p3 = frac( p * hscale ) ;
    p3 += dot(p3, p3.yzx + 209.191349);
    return frac( (p3.x + p3.y) * p3.z );
}

float noise( float3 x )
{
  return hash( floor(x) );
}
```

```
float4 main( idatas i )
{
  float l = max( length(i.strokePos - i.strokeStartPos ), EPSILON );
  float d = distance( i.pos, i.strokeStartPos ) / l;

  // compute noise value
  float2 npos = i.noiseScale*i.pos  ;
  float nv = i.noisePower*(1-2*noise( float3(npos,i.nbUserStroke+i.dist*0.001) ));

  float4 col = grad.sample( saturate(d+0.2*nv) ) ;
  col.a *= i.opacity ;


  return col ;
}
```

rakeNoise

```
cfg{name="rakeNoise";}

globals{

  float evoSpeed = 0.5 ;
  {
    uiName = "EvoSpeed";
    uiMax = 4 ;
    uiTab = "Noise" ;
  }
}

perPrim {

  float noiseSize = 1 ;
  {
    uiMin = 0.0      ;
    uiMax = 2      ;
    uiTab = "Noise"    ;
    uiFormat = percent  ;
  }

  float hardness = 0.5 ;
  {
    uiMin = 0      ;
    uiMax = 1      ;
    uiTab = "shape"  ;
    uiFormat = percent  ;
    uiName = "hardness"  ;
  }

  float hardRake = 0.5 ;
  {
    uiMin = 0      ;
    uiMax = 1      ;
    uiTab = "shape"  ;
    uiFormat = percent  ;
    uiName = "hardness rake"  ;
  }

  float ndensity = 0.5 ;
  {
    uiTab = "Noise" ;
    uiName = "Density"  ;
    uiFormat = percent ;
    uiMax = 1 ;
  }

}
```

```
float hash( float2 p )
{
  float hscale = .1031 ;

  float3 p3 = frac( float3(p.x,p.y,p.x) * hscale ) ;
    p3 += dot(p3, p3.yzx + 19.19);
    return frac( (p3.x + p3.y) * p3.z );
}

float adjustDensity( float r, float d )
{
  d = 1-d ;
  float a = clamp( (d-0.5)*2, 0, 1 );
  float b = clamp( d*2, 0, 1 );

  r = (r-a) / max( b-a, 0.001) ;
  r = saturate( r );

  return r ;
}

float hashD( float2 p, float d )
{
  return adjustDensity( hash(p), d );
}

float noiseEvo( idatas i )
{
  return i.nbUserStroke + toDrawSpace(i.dist)*evoSpeed*0.01;
}

float remap( float h, float a, float b )
{
  return saturate( (h-a) / (b-a+EPSILON) ) ;
}

float rakeNoise( idatas i, float density )
{
  float2 p = i.primBox.toCenter( i.pos );

  float rpf ;
  float rp = decompose( toDrawSpace(p.y) * i.noiseSize, rpf ) ;

  float f ;
  float dpos = decompose( noiseEvo(i), f );

  float n0 = smoothLerp( hashD(float2(rp,dpos),density),
hashD(float2(rp,dpos+1),density), f );
  float n1 = smoothLerp( hashD(float2(rp+1,dpos),density),
hashD(float2(rp+1,dpos+1),density), f );

  rpf = remap( rpf, 0.5*i.hardRake, 1-0.5*i.hardRake );

  return smoothLerp( n0, n1, rpf ) ;
}

float hardnessCpt( idatas i, float h )
{
  float sm = vmin( i.primBox.getSize() )  ;
  float d = max( -i.primDistance, 0 )    ;
  float softness = pow( 1 - h,2)  ;
  d /= sm*softness + 0.001    ;

  d = 1 - exp( -d );

  return i.primDistanceValid ? d : 1 ;
}

float4 main( idatas i )
{
  float rnoise = rakeNoise( i, i.ndensity );

  float ha = hardnessCpt( i, i.hardness );

  float4 col = i.color ;
  col.a *= rnoise ;
  col.a *= ha ;

  return col ;
}
```

Smooth Voronoi gradient

```
cfg{
  name="Smooth Voronoi Gradient";
  renderingTime = 60 ;
}

globals{
  colorGradient grad ;
  {
    uiTab = "color" ;
    id = 1 ;
  }
}

perPrim{
  float voroSmooth=0.1;
  {
    uiMax = 1 ;
    uiName = "Smoothness";
    uiTab = "Voronoi" ;
    uiFormat = percent ;
  }

  float ditherPower=0.1;
  {
    uiMax = 1 ;
    uiName = "Dithering power";
    uiTab = "Voronoi" ;
    uiFormat = percent ;
  }
}

float hash1( float2 p)
{
  float hscale = .1031 ;

  float3 p3 = frac( float3(p.x,p.y,p.x) * hscale ) ;
    p3 += dot(p3, p3.yzx + 19.19);
    return frac( (p3.x + p3.y) * p3.z );
}

float2 hash2( float2 p )
{
  float3 hscale3 = float3( .1031, .1030, .0973 ) ;

  float3 p3 = frac( float3(p.x,p.y,p.x) * hscale3) ;
    p3 += dot(p3, p3.yzx+19.19) ;
    return frac( float2( (p3.x + p3.y)*p3.z, (p3.x+p3.z)*p3.y) );
}

float4 hash4( float2 p )
```

```
{
  float4 hscale4 = float4( .1031, .1030, .0973, .1099 ) ;

  float4 p4 = frac( float4(p.xyxy) * hscale4 );
    p4 += dot( p4, p4.wzxy+19.19) ;
  return frac( float4((p4.x + p4.y)*p4.z, (p4.x + p4.z)*p4.y, (p4.y + p4.z)*p4.w,
(p4.z + p4.w)*p4.x));
}

float4 voronoi( idatas i, float2 uv, float time )
{
  float2 f ;
  float2 n = decompose( uv, f ) ;

  float seed = i.nbUserStroke ;
  float smoothness = lerp( 1, 50, pow(1-i.voroSmooth,4) );
  float2 ditherpos = i.pos ;
  float ditherPower = i.ditherPower ;

    //----------------------------------
    // regular voronoi
    //----------------------------------
    float tot = 0 ;
    float totAlph = 0 ;
    for( int j=-2; j<=2; j++ )
    {
    for( int i=-2; i<=2; i++ )
    {
      float2 g = float2( i, j );

      float2 ipos = n + g  ;

      float2 o = hash2( ipos+seed );
      o = 0.5 + 0.5*sin( time + 6.2831*o );  // animate the position

      float2 r = g + o - f;
      float d = dot(r,r);

      // ad dithering
      d += 0.2*ditherPower*(2*hash1( ditherpos+g+seed+1.05)-1) ;

      float alpha = saturate(exp(-smoothness*d));

      float col = hash2( ipos+seed+5.4 ).x ;

      tot += alpha*col ;
      totAlph += alpha ;
    }
  }

    return grad.sample( tot / totAlph ) ;
}

float4 main( idatas i )
{
  box2 b = box2FromCenterAxe( i.strokeStartPos, length(i.strokePos-
i.strokeStartPos), normalizeSafe(i.strokePos-i.strokeStartPos) );
  float2 p = 4*b.toCenter( i.pos ) / b.size ;
  float4 col = voronoi( i, p, i.dist*0.005 ) ;

  return col ;
}
```

Smudge

```
cfg{
  name = "Smudge" ;
  blendEx = true ;
  blendDefault = replace ;
  samplingLayerMaxOffset = 32 ;
  renderingTime = 30 ;
}

perPrim {

  float hardness = 0.5 ;
  {
    uiMin = 0      ;
    uiMax = 1      ;
    uiTab = "shape"    ;
    id = 0         ;
    uiFormat = percent  ;
    uiName = "Hardness"  ;
  }

  float maxDisplace = 1 ;
  {
    uiMin = 0        ;
    uiMax = 1        ;
    uiTab = "smudge"    ;
    uiFormat = percent    ;
    uiName = "Power"    ;
  }

  float overlayFactor = 0.1 ;
  {
    uiMax = 1          ;
    uiFormat = percent      ;
    uiName = "Color overlay"  ;
    uiTab = "color"         ;
  }

}

float smudgeFactor( idatas i )
{
  float sm = vmin( i.primBox.getSize() )  ;  // min axe size of the primitive
  float d = max( -i.primDistance, 0 )    ;  // distance to the primitive
  d = i.primDistanceValid ? d : 0      ;

  float softness = pow( 1 - i.hardness,2)  ;  // more linear feedback from the
parameter
  d /= sm*softness + 0.0001        ;  // normalize distance wit hthe primitive
size
```

```
  return 1 - exp( -d );  // attenuation curve
}

float4 main( idatas i )
{
  // get the current background color
  float4 colBack = bottomLayer.pointSample( i, i.pos );

  // compute smudge factor
  float sfact = smudgeFactor( i );

  // compute displacement from the current position
  float2 dir = i.strokePrecedPos - i.strokePos ;
  float mdisplace = length( dir ) * sfact * i.maxDisplace ;
  dir *= sfact * i.maxDisplace ;

  // Get the displaced color
  float4 smpl = bottomLayer.bilinearSample( i, i.pos + dir );
  smpl = smoothLerp( colBack, smpl, saturate( mdisplace ) );  // reset to a
pointSampling for little displacement

  // apply a color overlay
  float4 tmp = blendOverlay( smpl,
i.color*float4(1,1,1,0.5*i.overlayFactor*pow(sfact,1.95)) );

  // Eraser smudge effect
  tmp.a *= i.eraser ? 0.95 : 1 ;

  /// blend
  float alpha = i.color.a*pow(sfact,0.75);
  float4 col = blendNormalAlpha( colBack, tmp, alpha );

  return col ;
}
```

Smudge cloud noise

```
cfg{
  name = "Smudge cloud noise" ;
  blendEx = true ;
  blendDefault = replace ;
  samplingLayerMaxOffset = 32 ;
  renderingTime = 50 ;
}

perPrim {
  float hardness = 0.5 ;
  {
    uiMin = 0      ;
    uiMax = 1      ;
    uiTab = "shape"    ;
    id = 0        ;
    uiFormat = percent  ;
    uiName = "Hardness"  ;
  }
  float innerSize = 0.5 ;
  {
    id = 3 ;
    uiMin = 0      ;
    uiMax = 1      ;
    uiTab = "shape"  ;
    uiFormat = percent  ;
  }

  float maxDisplace = 1 ;
  {
    uiMin = 0        ;
    uiMax = 1        ;
    uiTab = "smudge"    ;
    uiFormat = percent    ;
    uiName = "Power"    ;
  }

  float colorAlpha = 0.1 ;
  {
    uiMax = 1          ;
    uiFormat = percent      ;
    uiName = "Color alpha"  ;
    uiTab = "color"        ;
  }

  float noisePower=0.1;
  {
    uiMax = 1 ;
    uiName = "Power";
    uiTab = "Noise" ;
    uiFormat = percent ;
  }
  float noiseScale=1;
  {
    uiMin = 0.25 ;
    uiMax = 5 ;
    uiName = "Scale";
    uiTab = "Noise" ;
    uiFormat = percent ;
  }
  float ndensity = 0.5 ;
  {
    uiTab = "Noise" ;
    uiName = "Density"  ;
    uiFormat = percent ;
    uiMax = 1 ;
  }

}

// compute the normalized distance from the edge to the center of the primitive
float cptPrimNormDist( idatas i )
{
  float sm = 0.5*vmax( i.primBox.getSize() ) ;  // retreive maximum primitive axe
size
  float d = max( -i.primDistance, 0 ) / sm ;    // compute normalized current
distance to the primitive edge

  return i.primDistanceValid ? d : 0 ;
}

// compute Hardness
// d - distance from primitive edge [0,1]
// h - hardness parameter [0,1]
// i - distance to the inner "safe" size [0,1]
//
float cptHardness( float d, float h, float i )
{
  float attsize = (1-i)*2*(1-h) ;
  float att = saturate( (d - (1-i - attsize*0.5) ) / max( attsize,0.0001) ) ;
  return smoothLerp(0,1,att);
}

float3 hash( float3 p )
{
  float3 hscale = float3(.1031,-.1029,.1032) ;

  float3 p3 = frac( p * hscale ) ;
    p3 += dot(p3, p3.yzx + 209.191349);
    return -1+2*frac( float3(  (p3.x + p3.y) * p3.z,
                (p3.z + p3.y) * p3.x,
                (p3.x + p3.z) * p3.y ) );
}

float3 adjustDensity( float3 rs, float d )
{
  float3 r = abs(rs);
  d = 1-d ;
  float3 a = clamp( (d-0.5)*2, 0, 1 );
  float3 b = clamp( d*2, 0, 1 );

  r = (r-a) / max( b-a, 0.001) ;
  r = saturate( r );

  return rs < 0 ? -r : r ;
}

float3 hashD( float3 p, float d )
{
  return adjustDensity( hash(p), d );
}

// Gradient noise see https://www.shadertoy.com/view/XdXGW8
// the returned value is between [0,1]
float noise( float3 x, float d )
{
  float3 f = frac(x);
  float3 p = decompose( x, f );
    float3 u = f*f*(3.0-2.0*f);

    float v1 = bilinearLerp(  dot( hashD(p+float3(0,0,0),d), f-float3(0,0,0) ),
dot( hashD(p+float3(1,0,0),d), f-float3(1,0,0) ),
                dot( hashD(p+float3(0,1,0),d), f-float3(0,1,0) ), dot(
hashD(p+float3(1,1,0),d), f-float3(1,1,0) ), u.xy ) ;

    float v2 = bilinearLerp(  dot( hashD(p+float3(0,0,1),d), f-float3(0,0,1) ),
dot( hashD(p+float3(1,0,1),d), f-float3(1,0,1) ),
                dot( hashD(p+float3(0,1,1),d), f-float3(0,1,1) ), dot(
hashD(p+float3(1,1,1),d), f-float3(1,1,1) ), u.xy ) ;

  float ret = lerp( v1, v2, u.z )/0.707;  // normalize with the maxiumm slope
  return abs(ret);
}

// return a noise value between [-1,1]
float noiseFunction( float3 x, float d )
{
  matrix3 m  = matrix3FromPosAxes( float3(-0.8,0.63,1.03), float3(0.97,0.01,-
0.036), float3(-0.02,-.87,0.03), float3(0.04,0.024,1.01) );
  float ret = 0.5   * noise( x, d ) ; x = 2*m.transform(x);
  ret += 0.25 * noise(x, d);  x = 2*m.transform(x);
  ret += 0.125 * noise(x, d);  x = 2*m.transform(x);
  ret += 0.0625 * noise(x, d);  x = 2*m.transform(x);
  return ret*2-1 ;
}

float4 main( idatas i )
{
  float d = cptPrimNormDist( i );

  // compute noised distance
  float2 npos = toDrawSpace(i.pos)*.1/i.noiseScale  ;
  float dn = d + i.noisePower*noiseFunction(
float3(npos,i.nbUserStroke+i.dist*0.0001), i.ndensity )  ;

  // get the current background color
  float4 colBack = bottomLayer.pointSample( i, i.pos );

  float att = cptHardness( dn, i.hardness, i.innerSize );
  att = i.primDistanceValid ? att : 1 ;


  // compute smudge factor
  float sfact = att;

  // compute angle noise
  float angle = (1-att)*HALFPI*noiseFunction(
float3(npos,i.nbUserStroke+i.dist*0.0001)+float3(-10.21,60.78,9780.7), i.ndensity
)  ;

  // compute displacement from the current position
  float2 dir = i.strokePrecedPos - i.strokePos ;
  float2 sphdir = toSpherical( dir+0.001 );
  sphdir.x += angle ;
  dir = sphdir.y*float2( cos(sphdir.x), sin(sphdir.x) );
  float mdisplace = length( dir ) * sfact * i.maxDisplace ;
  dir *= sfact * i.maxDisplace ;

  // Get the displaced color
  float4 smpl = bottomLayer.bilinearSample( i, i.pos + dir );
  smpl = smoothLerp( colBack, smpl, saturate( mdisplace ) );  // reset to a
pointSampling for little displacement

  // compute color noise
  float acolor = abs(noiseFunction(
float3(npos,i.nbUserStroke+i.dist*0.0001)+float3(7.11221,-97.78,0.7), i.ndensity
))  ;

  // apply a color on it
  float4 tmp = blendNormal( smpl, i.color*float4(1,1,1,0.5*i.colorAlpha*acolor) );

  /// blend
  float4 col = blendNormalAlpha( colBack, tmp, i.color.a*pow(sfact,0.75) );

  return col ;
}
```

Smudge cloud noise_01

```
cfg{
  name = "Smudge cloud noise" ;
  blendEx = true ;
  blendDefault = replace ;
  samplingLayerMaxOffset = 32 ;
  renderingTime = 40 ;
}

perPrim {
  float hardness = 0.5 ;
  {
    uiMin = 0      ;
    uiMax = 1      ;
    uiTab = "shape"    ;
    id = 0        ;
    uiFormat = percent  ;
    uiName = "Hardness"  ;
  }
  float innerSize = 0.5 ;
  {
    id = 3 ;
    uiMin = 0      ;
    uiMax = 1      ;
    uiTab = "shape"  ;
    uiFormat = percent  ;
  }

  float maxDisplace = 1 ;
  {
    uiMin = 0        ;
    uiMax = 1        ;
    uiTab = "smudge"    ;
    uiFormat = percent    ;
    uiName = "Power"    ;
  }

  float colorAlpha = 0.1 ;
  {
    uiMax = 1          ;
    uiFormat = percent      ;
    uiName = "Color alpha"  ;
    uiTab = "color"        ;
  }

  float noisePower=0.1;
  {
    uiMax = 1 ;
    uiName = "Power";
    uiTab = "Noise" ;
    uiFormat = percent ;
  }
  float noiseScale=1;
  {
    uiMin = 0.25 ;
    uiMax = 5 ;
    uiName = "Scale";
    uiTab = "Noise" ;
    uiFormat = percent ;
  }
  float ndensity = 0.5 ;
  {
    uiTab = "Noise" ;
    uiName = "Density"  ;
    uiFormat = percent ;
    uiMax = 1 ;
  }

}

// compute the normalized distance from the edge to the center of the primitive
float cptPrimNormDist( idatas i )
{
  float sm = 0.5*vmax( i.primBox.getSize() ) ;  // retreive maximum primitive axe
size
  float d = max( -i.primDistance, 0 ) / sm ;    // compute normalized current
distance to the primitive edge

  return i.primDistanceValid ? d : 0 ;
}

// compute Hardness
// d - distance from primitive edge [0,1]
// h - hardness parameter [0,1]
// i - distance to the inner "safe" size [0,1]
//
float cptHardness( float d, float h, float i )
{
  float attsize = (1-i)*2*(1-h) ;
  float att = saturate( (d - (1-i - attsize*0.5) ) / max( attsize,0.0001) ) ;
  return smoothLerp(0,1,att);
}

float3 hash( float3 p )
{
  float3 hscale = float3(.1031,-.1029,.1032) ;

  float3 p3 = frac( p * hscale ) ;
    p3 += dot(p3, p3.yzx + 209.191349);
    return -1+2*frac( float3(  (p3.x + p3.y) * p3.z,
                (p3.z + p3.y) * p3.x,
                (p3.x + p3.z) * p3.y ) );
}

float3 adjustDensity( float3 rs, float d )
{
  float3 r = abs(rs);
  d = 1-d ;
  float3 a = clamp( (d-0.5)*2, 0, 1 );
  float3 b = clamp( d*2, 0, 1 );

  r = (r-a) / max( b-a, 0.001) ;
  r = saturate( r );

  return rs < 0 ? -r : r ;
}

float3 hashD( float3 p, float d )
{
  return adjustDensity( hash(p), d );
}

// Gradient noise see https://www.shadertoy.com/view/XdXGW8
// the returned value is between [0,1]
float noise( float3 x, float d )
{
  float3 f = frac(x);
  float3 p = decompose( x, f );
    float3 u = f*f*(3.0-2.0*f);

    float v1 = bilinearLerp(  dot( hashD(p+float3(0,0,0),d), f-float3(0,0,0) ),
dot( hashD(p+float3(1,0,0),d), f-float3(1,0,0) ),
                dot( hashD(p+float3(0,1,0),d), f-float3(0,1,0) ), dot(
hashD(p+float3(1,1,0),d), f-float3(1,1,0) ), u.xy ) ;

    float v2 = bilinearLerp(  dot( hashD(p+float3(0,0,1),d), f-float3(0,0,1) ),
dot( hashD(p+float3(1,0,1),d), f-float3(1,0,1) ),
                dot( hashD(p+float3(0,1,1),d), f-float3(0,1,1) ), dot(
hashD(p+float3(1,1,1),d), f-float3(1,1,1) ), u.xy ) ;

  float ret = lerp( v1, v2, u.z )/0.707;  // normalize with the maxiumm slope
  return abs(ret);
}

// return a noise value between [-1,1]
float noiseFunction( float3 x, float d )
{
  matrix3 m  = matrix3FromPosAxes( float3(-0.8,0.63,1.03), float3(0.97,0.01,-
0.036), float3(-0.02,-.87,0.03), float3(0.04,0.024,1.01) );
  float ret = 0.5   * noise( x, d ) ; x = 2*m.transform(x);
  ret += 0.25 * noise(x, d);  x = 2*m.transform(x);
  ret += 0.125 * noise(x, d);  x = 2*m.transform(x);
  ret += 0.0625 * noise(x, d);  x = 2*m.transform(x);
  return ret*2-1 ;
}

float4 main( idatas i )
{
  float d = cptPrimNormDist( i );

  // compute noised distance
  float2 npos = toDrawSpace(i.pos)*.1/i.noiseScale  ;
  float dn = d + i.noisePower*noiseFunction(
float3(npos,i.nbUserStroke+i.dist*0.0001), i.ndensity )  ;

  // get the current background color
  float4 colBack = bottomLayer.pointSample( i, i.pos );

  float att = cptHardness( dn, i.hardness, i.innerSize );
  att = i.primDistanceValid ? att : 1 ;


  // compute smudge factor
  float sfact = att;

  // compute displacement from the current position
  float2 dir = i.strokePrecedPos - i.strokePos ;
  float mdisplace = length( dir ) * sfact * i.maxDisplace ;
  dir *= sfact * i.maxDisplace ;

  // Get the displaced color
  float4 smpl = bottomLayer.bilinearSample( i, i.pos + dir );
  smpl = smoothLerp( colBack, smpl, saturate( mdisplace ) );  // reset to a
pointSampling for little displacement

  // apply a color on it
  float4 tmp = blendNormal( smpl, i.color*float4(1,1,1,0.5*i.colorAlpha) );

  /// blend
  float alpha = i.color.a*pow(sfact,0.75);
  float4 col = blendNormalAlpha( colBack, tmp, alpha );

  // take care of the eraser state
  col = i.eraser ? float4( colBack.xyz, colBack.w * (1-alpha) ) : col ;

  return col ;
}
```

Smudge colored

```
cfg{
  name = "Smudge colored" ;
  blendEx = true ;
  blendDefault = replace ;
  samplingLayerMaxOffset = 32 ;
  renderingTime = 30 ;
}

perPrim {

  float hardness = 0.5 ;
  {
    uiMin = 0       ;
    uiMax = 1       ;
    uiTab = "shape"     ;
    id = 0         ;
    uiFormat = percent   ;
    uiName = "Hardness"   ;
  }
  float innerSize = 0.5 ;
  {
    id = 3 ;
    uiMin = 0       ;
    uiMax = 1       ;
    uiTab = "shape"   ;
    uiFormat = percent   ;
  }

  float maxDisplace = 1 ;
  {
    uiMin = 0         ;
    uiMax = 1         ;
    uiTab = "smudge"     ;
    uiFormat = percent     ;
    uiName = "Power"     ;
  }

  float colorAlpha = 0.1 ;
  {
    uiMax = 1             ;
    uiFormat = percent       ;
    uiName = "Color alpha"   ;
    uiTab = "color"         ;
  }

}

// compute the normalized distance from the edge to the center of the primitive
float cptPrimNormDist( idatas i )
{
  float sm = 0.5*vmax( i.primBox.getSize() ) ;  // retreive maximum primitive axe
size
  float d = max( -i.primDistance, 0 ) / sm ;    // compute normalized current
distance to the primitive edge

  return i.primDistanceValid ? d : 0 ;
}

// compute Hardness
// d - distance from primitive edge [0,1]
// h - hardness parameter [0,1]
// i - distance to the inner "safe" size [0,1]
//
float cptHardness( float d, float h, float i )
{
  float attsize = (1-i)*2*(1-h) ;
  float att = saturate( (d - (1-i - attsize*0.5) ) / max( attsize,0.0001) ) ;
  return smoothLerp(0,1,att);
}

float4 main( idatas i )
{
  float d = cptPrimNormDist( i );

  // get the current background color
  float4 colBack = bottomLayer.pointSample( i, i.pos );

  float att = cptHardness( d, i.hardness, i.innerSize );
  att = i.primDistanceValid ? att : 1 ;

  // compute smudge factor
  float sfact = att;

  // compute displacement from the current position
  float2 dir = i.strokePrecedPos - i.strokePos ;
  float mdisplace = length( dir ) * sfact * i.maxDisplace ;
  dir *= sfact * i.maxDisplace ;

  // Get the displaced color
  float4 smpl = bottomLayer.bilinearSample( i, i.pos + dir );
  smpl = smoothLerp( colBack, smpl, saturate( mdisplace ) );  // reset to a
pointSampling for little displacement


  // apply a color on it
  float4 acolor = i.color*float4(1,1,1,0.5*i.colorAlpha) ;
  acolor.a *= i.eraser ? 0 : 1 ;  // disable the color when the eraser is enabled
  float4 tmp = blendNormal( smpl, acolor );

  // Eraser smudge effect
  tmp.a *= i.eraser ? 0.95 : 1 ;

  /// blend
  float alpha =  i.color.a*pow(sfact,0.75);
  float4 col = blendNormalAlpha( colBack, tmp, alpha );

  return col ;
}
```

Smudge expand colored

```
cfg{
  name = "Smudge expand colored" ;
  blendEx = true ;
  blendDefault = replace ;
  samplingLayerMaxOffset = 32 ;
  renderingTime = 30 ;
}

perPrim {

  float hardness = 0.5 ;
  {
    uiMin = 0      ;
    uiMax = 1      ;
    uiTab = "shape"    ;
    id = 0        ;
    uiFormat = percent  ;
    uiName = "Hardness"  ;
  }
  float innerSize = 0.5 ;
  {
    id = 3 ;
    uiMin = 0      ;
    uiMax = 1      ;
    uiTab = "shape"  ;
    uiFormat = percent  ;
  }

  float maxDisplace = 1 ;
  {
    uiMin = 0        ;
    uiMax = 1        ;
    uiTab = "smudge"    ;
    uiFormat = percent    ;
    uiName = "Power"    ;
  }

  float colorAlpha = 0.1 ;
  {
    uiMax = 1          ;
    uiFormat = percent      ;
    uiName = "Color alpha"  ;
    uiTab = "color"        ;
  }

}

// compute the normalized distance from the edge to the center of the primitive
float cptPrimNormDist( idatas i )
{
  float sm = 0.5*vmax( i.primBox.getSize() ) ;  // retreive maximum primitive axe
size
  float d = max( -i.primDistance, 0 ) / sm ;    // compute normalized current
distance to the primitive edge

  return i.primDistanceValid ? d : 0 ;
}

// compute Hardness
// d - distance from primitive edge [0,1]
// h - hardness parameter [0,1]
// i - distance to the inner "safe" size [0,1]
//
float cptHardness( float d, float h, float i )
{
  float attsize = (1-i)*2*(1-h) ;
  float att = saturate( (d - (1-i - attsize*0.5) ) / max( attsize,0.0001) ) ;
  return smoothLerp(0,1,att);
}

float4 sampleFire( idatas i, float2 pos, float dirN )
{
  float4 smpl = 0 ;
  smpl +=  bottomLayer.bilinearSample( i, pos + dirN*-1 );
  smpl +=  bottomLayer.bilinearSample( i, pos + dirN*0 );
  smpl +=  bottomLayer.bilinearSample( i, pos + dirN*1 );
  smpl +=  bottomLayer.bilinearSample( i, pos + dirN*2 );

  float2 pdir = perpendicular( dirN );
  smpl +=  bottomLayer.bilinearSample( i, pos + pdir*1 );
  smpl +=  bottomLayer.bilinearSample( i, pos + pdir*-1 );

  return smpl / 6. ;
}

float4 main( idatas i )
{
  float d = cptPrimNormDist( i );

  float att = cptHardness( d, i.hardness, i.innerSize );
  att = i.primDistanceValid ? att : 1 ;

  // compute smudge factor
  float sfact = att;
  float alpha =  i.color.a*pow(sfact,0.75);

  float2 dir = i.primBox.center - i.pos ;
  float ldir = max( length( dir )-10, 0  ) ;
  dir = normalizeSafe( dir );

  float smplPower = 1-d ;
  smplPower = pow( smplPower, 1. );
  float2 posStart = i.pos + dir*ldir*smplPower ;

  float4 smpl = sampleFire( i, posStart, dir*smplPower*alpha*8 );

  // get the current background color
  float4 colBack = bottomLayer.pointSample( i, i.pos );


  // apply a color on it
  float4 acolor = i.color*float4(1,1,1,0.5*i.colorAlpha) ;
  acolor.a *= i.eraser ? 0 : 1 ;  // take into account eraser
  float4 tmp = blendNormal( smpl, acolor );

  // Eraser smudge effect
  tmp.a *= i.eraser ? 0.95 : 1 ;

  /// blend
  float4 col = blendNormalAlpha( colBack, tmp, alpha );

  return col ;
}
```

Smudge noise colored

```
cfg{
  name = "Smudge noise colored" ;
  blendEx = true ;
  blendDefault = replace ;
  samplingLayerMaxOffset = 64 ;
  renderingTime = 30 ;
}

globals{
  uiTab Noise ;
  {
    row = 3;
    id = 1;
  }
}

perPrim {

  float hardness = 0.5 ;
  {
    uiMin = 0       ;
    uiMax = 1       ;
    uiTab = "shape"     ;
    id = 0          ;
    uiFormat = percent  ;
    uiName = "Hardness"  ;
  }
  float innerSize = 0.5 ;
  {
    id = 3 ;
    uiMin = 0       ;
    uiMax = 1       ;
    uiTab = "shape"   ;
    uiFormat = percent  ;
  }

  float maxDisplace = 1 ;
  {
    uiMin = 0         ;
    uiMax = 1         ;
    uiTab = "smudge"     ;
    uiFormat = percent     ;
    uiName = "Power"     ;
  }

  float colorAlpha = 0.1 ;
  {
    uiMax = 1           ;
    uiFormat = percent       ;
    uiName = "Color alpha"   ;
    uiTab = "color"        ;
  }
  float noiseScale=1;
  {
    uiMin = 0.5 ;
    uiMax = 100 ;
    uiName = "Scale";
    uiTab = "Noise" ;
    uiFormat = percent ;
  }
  float scatterPower=0.5 ;
  {
    uiMin = 0 ;
    uiMax = 1 ;
    uiName = "Power";
    uiTab = "Noise" ;
    uiFormat = percent ;
  }

}

// compute the normalized distance from the edge to the center of the primitive
float cptPrimNormDist( idatas i )
{
  float sm = 0.5*vmax( i.primBox.getSize() ) ;  // retreive maximum primitive axe
size
  float d = max( -i.primDistance, 0 ) / sm ;    // compute normalized current
distance to the primitive edge

  return i.primDistanceValid ? d : 0 ;
}

// compute Hardness
// d - distance from primitive edge [0,1]
// h - hardness parameter [0,1]
// i - distance to the inner "safe" size [0,1]
//
float cptHardness( float d, float h, float i )
{
  float attsize = (1-i)*2*(1-h) ;
  float att = saturate( (d - (1-i - attsize*0.5) ) / max( attsize,0.0001) ) ;
  return smoothLerp(0,1,att);
}

float3 hash( float3 p )
{
  float3 hscale = float3(.1031,-.1029,.1032) ;

  float3 p3 = frac( p * hscale ) ;
    p3 += dot(p3, p3.yzx + 209.191349);
    return -1+2*frac( float3(  (p3.x + p3.y) * p3.z,
                (p3.z + p3.y) * p3.x,
                (p3.x + p3.z) * p3.y ) );
}

// Gradient noise see https://www.shadertoy.com/view/XdXGW8
// the returned value is between [-1,1]
float noise( float3 x )
{
  float3 f = frac(x);
  float3 p = decompose( x, f );
```

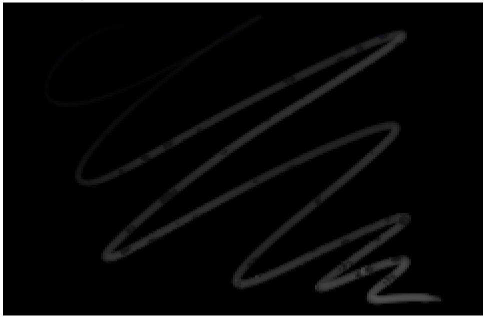
```
    float3 u = f*f*(3.0-2.0*f);

    float v1 = bilinearLerp(  dot( hash(p+float3(0,0,0)), f-float3(0,0,0) ), dot(
hash(p+float3(1,0,0)), f-float3(1,0,0) ),
                dot( hash(p+float3(0,1,0)), f-float3(0,1,0) ), dot(
hash(p+float3(1,1,0)), f-float3(1,1,0) ), u.xy ) ;

    float v2 = bilinearLerp(  dot( hash(p+float3(0,0,1)), f-float3(0,0,1) ), dot(
hash(p+float3(1,0,1)), f-float3(1,0,1) ),
                dot( hash(p+float3(0,1,1)), f-float3(0,1,1) ), dot(
hash(p+float3(1,1,1)), f-float3(1,1,1) ), u.xy ) ;

  return lerp( v1, v2, u.z )/0.707;  // normalize with the maxiumm slope
}

// return a noise value between [-1,1]
float noiseFunction( float3 x )
{
  matrix3 m  = matrix3FromPosAxes( float3(-0.8,0.63,1.03), float3(0.97,0.01,-
0.036), float3(-0.02,-.87,0.03), float3(0.04,0.024,1.01) );
  float2 ret = 0.5   * noise( x ) ; x = 2*m.transform(x);
  ret += 0.25 * noise(x);  x = 2*m.transform(x);
  ret += 0.125 * noise(x);  x = 2*m.transform(x);
  ret += 0.0625 * noise(x);  x = 2*m.transform(x);
  return ret ;
}

float2 noiseVecFunction( float3 x )
{
  return float2(  noiseFunction( x + float3(EPSILON,0,0) ) - noiseFunction( x +
float3(-EPSILON,0,0) ),
                  noiseFunction( x + float3(0,EPSILON,0) ) - noiseFunction( x +
float3(0,-EPSILON,0) ) ) / EPSILON;
}

float2 cptNoiseDisplace( idatas i )
{
  // compute noise value
  float2 npos = i.pos/i.noiseScale;
  float2 nv = noiseVecFunction( float3(npos,i.nbUserStroke+i.dist*0.1) );

  // compute the current pixel displacement
  float mdisplace = i.scatterPower * samplingLayerMaxOffset;
  float2  offset = nv*mdisplace ;

  return offset ;
}

float cptAttNoise( idatas i )
{
  float d = length( i.pos - i.primBox.center ) / vmax( i.primBox.size );
  return pow( clamp( d-0.005, 0, 1 ), 1.5 );
}

float4 main( idatas i )
{
  float d = cptPrimNormDist( i );

  // get the current background color
  float4 colBack = bottomLayer.pointSample( i, i.pos );
  float att = cptHardness( d, i.hardness, i.innerSize );
  att = i.primDistanceValid ? att : 1 ;

  // compute smudge factor
  float sfact = att;

  // compute displacement from the current position
  float2 dir = i.strokePrecedPos - i.strokePos ;
  float mdisplace = length( dir ) * sfact * i.maxDisplace ;
  dir *= sfact * i.maxDisplace ;

  // noise displacement
  float nfact = sfact * cptAttNoise( i );
  dir += cptNoiseDisplace( i )*nfact;

  // Get the displaced color
  float4 smpl = bottomLayer.bilinearSample( i, i.pos + dir );

  // apply a color on it
  float4 acolor = i.color*float4(1,1,1,0.5*i.colorAlpha) ;
  acolor.a *= i.eraser ? 0 : 1 ;
  float4 tmp = blendNormal( smpl, acolor );

  // eraser
  tmp.a *= i.eraser ? 0.95 : 1 ;

  /// blend
  float alpha =  i.color.a*pow(sfact,0.75);
  float4 col = blendNormalAlpha( colBack, tmp, alpha );

  return col ;
}
```

Smudge Scatter perlin noise

```
cfg{
  name = "Smudge Scatter perlin noise" ;
  blendEx = true ;
  blendDefault = replace ;
  samplingLayerMaxOffset = 64 ;
  renderingTime = 40 ;
}

perPrim {

  float hardness = 0.5 ;
  {
    uiMin = 0       ;
    uiMax = 1       ;
    uiTab = "shape"    ;
    id = 0         ;
    uiFormat = percent  ;
    uiName = "Hardness"  ;
  }
  float innerSize = 0.5 ;
  {
    id = 3 ;
    uiMin = 0       ;
    uiMax = 1       ;
    uiTab = "shape"  ;
    uiFormat = percent  ;
  }
  float maxDisplace = 1 ;
  {
    uiMin = 0         ;
    uiMax = 1         ;
    uiTab = "Noise"      ;
    uiFormat = percent    ;
    uiName = "Smudge Power"  ;
  }

  float colorAlpha = 0.1 ;
  {
    uiMax = 1           ;
    uiFormat = percent      ;
    uiName = "Color alpha"  ;
    uiTab = "color"         ;
  }
  float scatterPower=0.5 ;
  {
    uiMin = 0 ;
    uiMax = 1 ;
    uiName = "Power";
    uiTab = "Noise" ;
    uiFormat = percent ;
  }
  float noiseScale=1;
  {
    uiMin = 0.01 ;
    uiMax = 2 ;
    uiName = "Scale";
    uiTab = "Noise" ;
    uiFormat = percent ;
  }

}

// compute the normalized distance from the edge to the center of the primitive
float cptPrimNormDist( idatas i )
{
  float sm = 0.5*vmax( i.primBox.getSize() ) ;  // retreive maximum primitive axe
size
  float d = max( -i.primDistance, 0 ) / sm ;    // compute normalized current
distance to the primitive edge

  return i.primDistanceValid ? d : 0 ;
}

// compute Hardness
// d - distance from primitive edge [0,1]
// h - hardness parameter [0,1]
// i - distance to the inner "safe" size [0,1]
//
float cptHardness( float d, float h, float i )
{
  float attsize = (1-i)*2*(1-h) ;
  float att = saturate( (d - (1-i - attsize*0.5) ) / max( attsize,0.0001) ) ;
  return smoothLerp(0,1,att);
}

float3 hash( float3 p )
{
  float3 hscale = float3(.1031,-.1029,.1032) ;

  float3 p3 = frac( p * hscale ) ;
    p3 += dot(p3, p3.yzx + 209.191349);
    return -1+2*frac( float3(  (p3.x + p3.y) * p3.z,
                (p3.z + p3.y) * p3.x,
                (p3.x + p3.z) * p3.y ) );
}

// Gradient noise see https://www.shadertoy.com/view/XdXGW8
// the returned value is between [-1,1]
float noise( float3 x )
{
  float3 f = frac(x);
  float3 p = decompose( x, f );
    float3 u = f*f*(3.0-2.0*f);

    float v1 = bilinearLerp(  dot( hash(p+float3(0,0,0)), f-float3(0,0,0) ), dot(
hash(p+float3(1,0,0)), f-float3(1,0,0) ),
                dot( hash(p+float3(0,1,0)), f-float3(0,1,0) ), dot(
hash(p+float3(1,1,0)), f-float3(1,1,0) ), u.xy ) ;

    float v2 = bilinearLerp(  dot( hash(p+float3(0,0,1)), f-float3(0,0,1) ), dot(
hash(p+float3(1,0,1)), f-float3(1,0,1) ),
                dot( hash(p+float3(0,1,1)), f-float3(0,1,1) ), dot(
hash(p+float3(1,1,1)), f-float3(1,1,1) ), u.xy ) ;

  return lerp( v1, v2, u.z )/0.707;  // normalize with the maxiumm slope
}

// Gradient noise see https://www.shadertoy.com/view/XdXGW8
// the returned value is 2D vector between [-1,1]
float2 noise2( float3 x )
{
  return   float2( noise(x), noise(-x.yzx+float3(10.321,10.8798,11.9842)) ) ;
}

// return a noise value between [-1,1]
float2 noiseFunction( float3 x )
{
  matrix3 m  = matrix3FromPosAxes( float3(-0.8,0.63,1.03), float3(0.97,0.01,-
0.036), float3(-0.02,-.87,0.03), float3(0.04,0.024,1.01) );
  float2 ret = 0.5   * noise2( x ) ; x = 2*m.transform(x);
  ret += 0.25 * noise2(x);  x = 2*m.transform(x);
  ret += 0.125 * noise2(x);  x = 2*m.transform(x);
  ret += 0.0625 * noise2(x);  x = 2*m.transform(x);
  return ret ;
}

float4 main( idatas i )
{
  float d = cptPrimNormDist( i );

  // get the current background color
  float4 colBack = bottomLayer.pointSample( i, i.pos );

  // compute hardness attenuation
  float att = cptHardness( d, i.hardness, i.innerSize );
  att = i.primDistanceValid ? att : 1 ;


  // compute noise value
  float2 npos = i.pos*0.1/i.noiseScale ;
  float2 nv = noiseFunction( float3(npos,i.nbUserStroke+i.dist*0.001) );

  // compute the noise displacement
  float mdisplace = att * i.scatterPower * samplingLayerMaxOffset ;
  float2  noffset = nv*mdisplace ;

  // compute smudge factor
  float sfact = att*att ;

  // compute displacement from the current position
  float2 soffset = i.strokePrecedPos - i.strokePos ;
  float sdisplace = sfact * i.maxDisplace * min( length( soffset ),
samplingLayerMaxOffset ) ;
  soffset = normalizeSafe(soffset) * sdisplace ;

  // compute final offset
  float2 offset = noffset + soffset ;

  // Get the displaced color
  float4 smpl = bottomLayer.bilinearSample( i, i.pos + offset );
  smpl = smoothLerp( colBack, smpl, saturate( length(offset) ) );  // reset to a
pointSampling for little displacement
```

```
  // apply a color on it
  float4 acolor = i.color*float4(1,1,1,0.15*i.colorAlpha*att) ;
  acolor.a *= i.eraser ? 0 : 1 ;
  float4 tmp = blendNormal( smpl, acolor );

  // take care of the eraser state
  tmp.a *= i.eraser ? 0.95 : 1 ;

  /// blend
  float alpha = i.color.a*pow(sfact,0.75);
  float4 col = blendNormalAlpha( colBack, tmp, alpha );

  return col ;
}
```

Smudge scatter trabeculum



```
cfg{
  name = "Smudge Scatter trabeculum" ;
  blendEx = true ;
  blendDefault = replace ;
  samplingLayerMaxOffset = 64 ;
  renderingTime = 40 ;
}

perPrim {
  float hardness = 0.5 ;
  {
    uiMin = 0      ;
    uiMax = 1      ;
    uiTab = "shape"    ;
    id = 0        ;
    uiFormat = percent  ;
    uiName = "Hardness"  ;
  }
  float innerSize = 0.5 ;
  {
    id = 3 ;
    uiMin = 0      ;
    uiMax = 1      ;
    uiTab = "shape"  ;
    uiFormat = percent  ;
  }
  float maxDisplace = 1 ;
  {
    uiMin = 0        ;
    uiMax = 1        ;
    uiTab = "Noise"      ;
    uiFormat = percent    ;
    uiName = "Smudge Power"  ;
  }
  float colorAlpha = 0.1 ;
  {
    uiMax = 1          ;
    uiFormat = percent      ;
    uiName = "Color alpha"  ;
    uiTab = "color"        ;
  }
  float scatterPower=0.5 ;
  {
    uiMin = 0 ;
    uiMax = 1 ;
    uiName = "Power";
    uiTab = "Noise" ;
    uiFormat = percent ;
  }
  float noiseScale=1;
  {
    uiMin = 0.01 ;
    uiMax = 10 ;
    uiName = "Scale";
    uiTab = "Noise" ;
    uiFormat = percent ;
  }
    float TrabHardness = 0.5 ;
    {
        id = -1 ;
        uiMin = 0           ;
        uiMax = 1           ;
        uiTab = "shape" ;
        uiFormat = percent  ;
        uiName = "cell hardness" ;
    }
}

// compute the normalized distance from the edge to the center of the primitive
float cptPrimNormDist( idatas i )
{
  float sm = 0.5*vmax( i.primBox.getSize() ) ;  // retreive maximum primitive axe
size
  float d = max( -i.primDistance, 0 ) / sm ;    // compute normalized current
distance to the primitive edge

  return i.primDistanceValid ? d : 0 ;
}

// compute Hardness
// d - distance from primitive edge [0,1]
// h - hardness parameter [0,1]
// i - distance to the inner "safe" size [0,1]
//
float cptHardness( float d, float h, float i )
{
  float attsize = (1-i)*2*(1-h) ;
  float att = saturate( (d - (1-i - attsize*0.5) ) / max( attsize,0.0001) ) ;
  return smoothLerp(0,1,att);
}

float2 hash3( float3 p )
{
  float3 hscale = float3(.1031,-.1029,.1032) ;

  float3 p3 = frac( p * hscale ) ;
    p3 += dot(p3, p3.yzx + 209.191349);
    return -1+2*frac( float2(  (p3.x + p3.y) * p3.z,
                (p3.z + p3.y) * p3.x ) );
}

float2 hash2( float2 p )
{
    float3 hscale3 = float3( .1031, .1030, .0973 ) ;

    float3 p3 = frac( float3(p.x,p.y,p.x) * hscale3) ;
    p3 += dot(p3, p3.yzx+19.19) ;
    return frac( float2( (p3.x + p3.y)*p3.z, (p3.x+p3.z)*p3.y) );
}

float2 voronoi( idatas i, float2 uv, float time )
```

```
{
    float2 f ;
    float2 n = decompose( uv, f ) ;

    float seed = i.nbUserStroke ;

    //----------------------------------
    // regular voronoi
    //----------------------------------
    float3 lastd = 10 ;
  float2 off ;
    for( int j=-1; j<=1; j++ )
    {
        for( int i=-1; i<=1; i++ )
        {
            float2 g = float2( i, j );

            float2 ipos = n + g ;

            float2 o = hash2( ipos+seed );
            o = 0.5 + 0.5*sin( time + 6.2831*o );   // animate the position

            float2 r = g + o - f;
            float d = dot(r,r);


            // Get color and orderd 3 min distances
            if( d < lastd.x )
            {
        off = hash3( float3(ipos,time+seed) );
                lastd.yz = lastd.xy ;
                lastd.x = d ;
            }
            else if( d < lastd.y )
            {
                lastd.z = lastd.y ;
                lastd.y = d ;
            }
            else if( d < lastd.z )
            {
                lastd.z = d ;
            }
        }
    }

    lastd = 5*sqrt(lastd);

    float alpha = 1./( 1./ (lastd.y-lastd.x)+1./(lastd.z-lastd.x) ) ; // Formula
(c) Fabrice NEYRET

    float hcut = lerp( 3, 0.05, i.TrabHardness ) ;
    float2 ret = alpha > hcut ? off : 0  ;

    return ret ;
}

float4 main( idatas i )
{
  float d = cptPrimNormDist( i );

  // get the current background color
  float4 colBack = bottomLayer.pointSample( i, i.pos );

  // compute hardness attenuation
  float att = cptHardness( d, i.hardness, i.innerSize );
  att = i.primDistanceValid ? att : 1 ;


  // compute noise value
  float2 npos = i.pos*0.1/i.noiseScale ;
  float2 nv = voronoi( i, npos, i.dist*0.01 );

  // compute the noise displacement
  float mdisplace = att * i.scatterPower * samplingLayerMaxOffset ;
  float2  noffset = nv*mdisplace ;

  // compute smudge factor
  float sfact = att*att ;

  // compute displacement from the current position
  float2 soffset = i.strokePrecedPos - i.strokePos ;
  float sdisplace = sfact * i.maxDisplace * min( length( soffset ),
samplingLayerMaxOffset ) ;
  soffset = normalizeSafe(soffset) * sdisplace ;

  // compute final offset
  float2 offset = noffset + soffset ;

  // Get the displaced color
  float4 smpl = bottomLayer.bilinearSample( i, i.pos + offset );
  smpl = smoothLerp( colBack, smpl, saturate( length(offset) ) );  // reset to a
pointSampling for little displacement

  // apply a color on it
  float4 acolor = i.color*float4(1,1,1,0.15*i.colorAlpha*att) ;
  acolor.a *= i.eraser ? 0 : 1 ;
  float4 tmp = blendNormal( smpl, acolor );

  // take care of the eraser state
  tmp.a *= i.eraser ? 0.95 : 1 ;

  /// blend
  float alpha = i.color.a*pow(sfact,0.75);
  float4 col = blendNormalAlpha( colBack, tmp, alpha );

  return col ;
}
```

smudgeRake

```
cfg{
  name = "smudgeRake";
  blendEx = true ;
  blendDefault = replace ;
  samplingLayerMaxOffset = 16 ;
  renderingTime = 30 ;
}

globals{

  uiTab smudge ;
  {
    row = 2;
    id = 3 ;
  }

  float evoSpeed = 0.5 ;
  {
    uiName = "EvoSpeed";
    uiMax = 4 ;
    uiTab = "smudge" ;
    id = 100 ;
  }

}

perPrim {

  float noiseSize = 1 ;
  {
    uiMin = 0.0      ;
    uiMax = 2      ;
    uiTab = "smudge"  ;
    uiFormat = percent  ;
    uiName = "Filament" ;
  }

  float hardness = 0.5 ;
  {
    uiMin = 0      ;
    uiMax = 1      ;
    uiTab = "shape"  ;
    uiFormat = percent  ;
    uiName = "hardness"  ;
    id = -2 ;
  }

  float maxDisplace = 1 ;
  {
    uiMax = 1      ;
    uiTab = "smudge"  ;
    uiFormat = percent  ;
    uiName = "Power" ;
    id = -1 ;
  }

  float overlayFactor = 0.1 ;
  {
    uiMax = 1          ;
    uiTab = "color"      ;
    uiFormat = percent      ;
    uiName = "color overlay"  ;
  }

  float hardRake = 0.5 ;
  {
    uiMin = 0      ;
    uiMax = 1      ;
    uiTab = "shape"  ;
    uiFormat = percent  ;
    uiName = "hardness rake"  ;
    id = -1 ;
  }

  float ndensity = 0.5 ;
  {
    uiTab = "smudge" ;
    uiName = "Density"  ;
    uiFormat = percent ;
    uiMax = 1 ;
  }

}

float hash( float2 p )
{
  float hscale = .1031 ;

  float3 p3 = frac( float3(p.x,p.y,p.x) * hscale ) ;
    p3 += dot(p3, p3.yzx + 19.19);
    return frac( (p3.x + p3.y) * p3.z );
}

float adjustDensity( float r, float d )
{
  d = 1-d ;
  float a = clamp( (d-0.5)*2, 0, 1 );
  float b = clamp( d*2, 0, 1 );

  r = (r-a) / max( b-a, 0.001) ;
  r = saturate( r );

  return r ;
}

float hashD( float2 p, float d )
{
  return adjustDensity( hash(p), d );
}

float noiseEvo( idatas i )
```

```
{
  return i.nbUserStroke + toDrawSpace(i.dist)*evoSpeed*0.01;
}

float remap( float h, float a, float b )
{
  return saturate( (h-a) / (b-a+EPSILON) ) ;
}

float rakeNoise( idatas i, float density )
{
  float2 p = i.primBox.toCenter( i.pos );

  float rpf ;
  float rp = decompose( toDrawSpace(p.y) * i.noiseSize, rpf ) ;

  float f ;
  float dpos = decompose( noiseEvo(i), f );

  float n0 = smoothLerp( hashD(float2(rp,dpos),density),
hashD(float2(rp,dpos+1),density), f );
  float n1 = smoothLerp( hashD(float2(rp+1,dpos),density),
hashD(float2(rp+1,dpos+1),density), f );

  rpf = remap( rpf, 0.5*i.hardRake, 1-0.5*i.hardRake );

  return smoothLerp( n0, n1, rpf ) ;
}

float smudgeFactor( idatas i )
{
  float sm = vmin( i.primBox.getSize() )  ;
  float d = max( -i.primDistance, 0 )    ;
  d = i.primDistanceValid ? d : 0      ;

  float softness = pow( 1 - i.hardness,2)  ;
  d /= sm*softness + 0.0001        ;

  return 1 - exp( -d );  // attenuation curve
}

float4 main( idatas i )
{
  float dirN = rakeNoise( i, i.ndensity )  ;

  // get the current background color
  float4 colBack = bottomLayer.pointSample( i, i.pos );

  // compute smudge factor
  float sfact = smudgeFactor( i );
  sfact *= dirN  ;

  // compute displacement from the current position
  float2 dir = i.strokePrecedPos - i.strokePos ;
  float mdisplace = length( dir ) * sfact * i.maxDisplace ;
  dir *= sfact * i.maxDisplace ;

  // Get the displaced color
  float4 smpl = bottomLayer.bilinearSample( i, i.pos + dir );
  smpl = smoothLerp( colBack, smpl, saturate( mdisplace ) );  // reset to a
pointSampling for little displacement
```

```
  // apply a color overlay
  float4 acolor = i.color*float4(1,1,1,i.overlayFactor*pow(sfact,1.95)) ;
  acolor.a *= i.eraser ? 0 : 1 ;
  float4 tmp = blendOverlay( smpl, acolor );
  smpl.xyz = tmp.xyz ;

  // take care of the eraser state
  smpl.a *= i.eraser ? 0.95 : 1 ;

  /// blend
  float alpha = i.color.a*pow(sfact,0.95);
  float4 col = blendNormalAlpha( colBack, smpl, alpha );

  return col ;
}
```

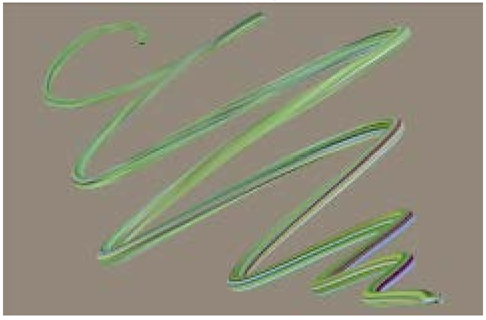
subtle color shift

```
cfg{
  name="subtle color shift";
  renderingTime = 30 ;
  blendEx = true ;
  blendDefault = replace ;

  samplerDefault ;
  {
    adressU = clamp;
    adressV = clamp;
  }
}

globals{
  uiTab "Texture"  ;

  texture shape;
  {
    id = -2  ;
    uiTab = "shape" ;
    uiName = "texture shape";
  }
  texture splash;
  {
    id = -1 ;
    uiTab = "shape" ;
    uiName = "texture splash";
  }
  uiTab stroke  ;
  {
    row = 99 ;
  }
  uiTab colorShifting  ;
  {
    row = 100 ;
  }
}

perPrim {

  float pressure = 0.5 ;
  {
    uiMin = 0    ;
    uiMax = 1    ;
    uiFormat = percent ;
    uiName = "pressure" ;
    uiTab = "stroke" ;
  }

  float hshift = 0.5 ;
  {
    uiMin = 0    ;
    uiMax = 1    ;
    uiFormat = percent ;
    uiName = "hue" ;
    uiTab = "colorShifting" ;
  }

  float sshift = 0.5 ;
  {
    uiMin = 0    ;
    uiMax = 1    ;
    uiFormat = percent ;
    uiName = "saturation" ;
    uiTab = "colorShifting" ;
  }

  float vshift = 0.5 ;
  {
    uiMin = 0    ;
    uiMax = 1    ;
    uiFormat = percent ;
    uiName = "brightness" ;
    uiTab = "colorShifting" ;
  }

}

float4 hash2( float2 p )
{
  float4 hscale = float4( .1031, 0.1059, -0.1087, -0.1029 ) ;

  float4 p4 = frac( p.xyxy * hscale ) ;
    p4 += dot(p4, p4.yzwx + 209.191349);
    return frac( float4( (p4.x + p4.y) * ( p4.z - p4.w ),
             (p4.y + p4.z) * ( p4.w - p4.x ),
             (p4.z + p4.w) * ( p4.x - p4.y ),
             (p4.w + p4.x) * ( p4.y - p4.z ) ) );
}

float4 cptSplash( float2 pos, float seed )
{
  float4 col = splash.sample( pos ) ;  // first sampling
  col = 0 ;

  float4 noise ;
  matrix2 b ;
  float maxmove = 0.50 ;
  float minSize = 0.5 ;
  float maxSize = 2.5 ;
  float2 texp ;



  // other sampling
  noise = hash2( float2(seed,3) ) ;
  b = matrix2FromPRS( maxmove*(noise.xy*2-1),
            TWOPI*noise.w,
            lerp( minSize, maxSize, noise.z ) ) ;

  texp = b.transform( pos-0.5 )+0.5;
  col += splash.sample( texp ) ;

  // other sampling
  noise = hash2( float2(seed,3) ) ;
  b = matrix2FromPRS( maxmove*(noise.xy*2-1),
            TWOPI*noise.w,
            lerp( minSize, maxSize, noise.z ) ) ;

  texp = b.transform( pos-0.5 )+0.5;
  col += splash.sample( texp ) ;

  return saturate( col );
}

float remap( in float a, float minv, float maxv )
{
  return saturate( (a-minv) / (maxv-minv) );
}

float cptFromPressure( float a, float pressure )
{
  float pmin;
  float pmax ;

  pmin = pressure < 0.5 ? (1-2*pressure) : -(pressure-0.49) ;
  pmax = pressure < 0.5 ? 1.01 : (1-2*(pressure-0.5)) ;
  return remap( a, pmin, pmax );
}

float3 hash3( float2 p )
{
  float hscale = float3(.1031,0.1029,-0.1027) ;

  float3 p3 = frac( p.xyx * hscale ) ;
    p3 += dot(p3, p3.yzx + 19.19);
    return float3(  frac( (p3.x + p3.y) * p3.z ),
          frac( (p3.z + p3.y) * p3.x ),
          frac( (p3.z + p3.x) * p3.y ) ) ;
}

float4 main( idatas i )
{
  float alpha = i.color.a ;

  // Get Shape
  {
    float2 pos = i.primBox.toUpperLeftNorm( i.pos ) ;
    float4 col = shape.sample( pos ) ;
    float a = col.a ;
    a = shape.isEmpty ? 1 : a ;
    alpha *= a ;
  }

  // Get Splash
  {
    float2 pos = i.primBox1.toUpperLeftNorm( i.pos ) ;
    float4 col = cptSplash( pos, i.primShapeId1+i.nbUserStroke ) ;

    float aover = i.primBox1Valid ? col.a : 1 ;

    // apply overlay blending
    alpha *= cptFromPressure( aover, i.pressure );
  }

  // Compute Color shifting
  float4 colBack = bottomLayer.pointSample( i, i.pos );

  float4 hsv = fromRGBtoHSV( colBack );

  float3 n = 2*hash3( float2(i.dist*0.1, i.nbUserStroke ) )-1;

  hsv.xyz += n*(10./100.)*float3(i.hshift,i.sshift,i.vshift);
  hsv.yz = saturate(hsv.yz);

  float4 rgb = fromHSVtoRGB( hsv );

  rgb.rgb = lerp( colBack.rgb, rgb.rgb, alpha ) ;

  float4 col = float4( rgb.rgb, colBack.a ) ;

  // take care of the eraser state
  col = i.eraser ? float4( col.xyz, col.w * (1-alpha) ) : col ;


  return col ;
}
```

Texture Background pattern

```
cfg{
  name="Texture Background pattern";
  renderingTime = 30 ;
}

globals{

  uiTab stroke  ;
  {
    row = 99 ;
  }

  uiTab roughness  ;
  {
    row = 100 ;
  }

  texture mat ;
  {
    uiTab = "roughness" ;
    uiName = "pattern" ;
  }

  float scalePattern ;
  {
    id = 100;
    uiMin = 0.25    ;
    uiMax = 10    ;
    uiFormat = percent ;
    uiTab = "roughness" ;
  }
}

perPrim {


  float pressure = 0.5 ;
  {
    uiMin = 0    ;
    uiMax = 1    ;
    uiFormat = percent ;
    uiName = "pressure" ;
    uiTab = "stroke" ;
  }

  float directionality = 0.85 ;
  {
    uiMin = 0  ;
    uiMax = 1  ;
    uiFormat = percent ;
    uiName = "directionality" ;
    uiTab = "stroke" ;
  }

  float hardness = 1 ;
  {
    uiTab = "Shape" ;
    id = -1 ;
    uiMax = 1 ;
    uiName = "Hardness" ;
  }

  // direction du tracé actuel
  float2 dir ;
  {
    uiEditor = angleDist ;
    raw = true ;
    uiTab = "stroke" ;
  }
}

float surfaceHeight( float2 uv )
{
  return luminance( mat.sample( uv/scalePattern ) )  ;
}

float3 surfaceNormal( float2 pos, float h )
{
  float2 opa = 1/mat.size ;
  float2 uv = pos / mat.size ;

  // compute normal with a Sobel filter
  float s00 = surfaceHeight( uv + opa*float2(-1,-1) );
  float s10 = surfaceHeight( uv + opa*float2( 0,-1) );
  float s20 = surfaceHeight( uv + opa*float2( 1,-1) );
  float s01 = surfaceHeight( uv + opa*float2(-1, 0) );
  float s21 = surfaceHeight( uv + opa*float2( 1, 0) );
  float s02 = surfaceHeight( uv + opa*float2(-1, 1) );
  float s12 = surfaceHeight( uv + opa*float2( 0, 1) );
  float s22 = surfaceHeight( uv + opa*float2( 1, 1) );

  // Compute dx using Sobel:
  //           -1 0 1
  //           -2 0 2
  //           -1 0 1
  float dX = -s00 + s20 -2*s01 + 2*s21 -s02 + s22  ;

    // Compute dy using Sobel:
    //           -1 -2 -1
    //            0  0  0
    //            1  2  1
    float dY = -s00 -2*s10 -s20 + s02 + 2*s12 + s22 ;

  return normalizeSafe( float3( h*dX, h*dY, 1 ) );
}

float hardnessCpt( idatas i, float h )
{
  float sm = vmin( i.primBox.getSize() )  ;
  float d = max( -i.primDistance, 0 )    ;
  float softness = pow( 1 - h,2)  ;
  d /= sm*softness + 0.001    ;
  d = 1 - exp( -d );

  d = i.primDistanceValid ? d : 1 ;
  return d;
}

float remap( in float a, float minv, float maxv )
{
  return saturate( (a-minv) / (maxv-minv) );
}

float4 main( idatas i )
{
  float heightscale = 2 ;

  float2 dir = normalizeSafe( i.dir );

  float3 bn = normalizeSafe( float3(heightscale*dir,-1) );
  float3 cn = surfaceNormal( i.pos, heightscale );

  float ah = surfaceHeight( i.pos / mat.size ) ;
  float pressure = i.pressure ;
  float pmin;
  float pmax ;

  pmin = pressure < 0.5 ? (1-2*pressure) : -(pressure-0.49) ;
  pmax = pressure < 0.5 ? 1.01 : (1-2*(pressure-0.5)) ;
  ah = remap( ah, pmin, pmax );

  float a = saturate( -dot( bn, cn ) );
  a = lerp( 1-i.directionality, 1, lerp(a,1,pressure*pressure*pressure) );

  a *= ah ;

  a*= hardnessCpt( i, i.hardness );

  float4 col = i.color ;
  col.a *= a ;

  return col ;
}
```

Texture Voronoi Tiling

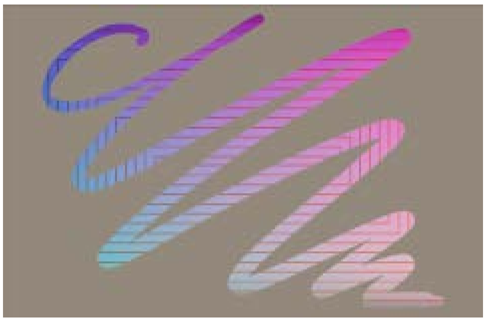


```
cfg{
  name="Texture Voronoi Tiling";
  renderingTime = 30 ;
}

globals{
  texture mat;
  {
    uiTab = "Texture" ;
  }
}

perPrim{
  float4 color1=1;
  {
    uiName = "Color";
    uiTab = "Color" ;
  }
  float overlay=0;
  {
    uiMax = 1 ;
    uiTab = "Color" ;
    uiFormat = percent ;
  }
  float matAlpha=1;
  {
    uiName = "alpha";
    uiTab = "Color" ;
    uiFormat = percent ;
    uiMax = 1 ;
  }
}

float hash1( float2 p)
{
  float hscale = .1031 ;

  float3 p3 = frac( float3(p.x,p.y,p.x) * hscale ) ;
    p3 += dot(p3, p3.yzx + 19.19);
    return frac( (p3.x + p3.y) * p3.z );
}

float2 hash2( float2 p )
{
  float3 hscale3 = float3( .1031, .1030, .0973 ) ;

  float3 p3 = frac( float3(p.x,p.y,p.x) * hscale3) ;
    p3 += dot(p3, p3.yzx+19.19) ;
    return frac( float2( (p3.x + p3.y)*p3.z, (p3.x+p3.z)*p3.y) );
```

```
}

float4 hash4( float2 p )
{
  float4 hscale4 = float4( .1031, .1030, .0973, .1099 ) ;

  float4 p4 = frac( float4(p.xyxy) * hscale4 );
    p4 += dot( p4, p4.wzxy+19.19) ;
  return frac( float4((p4.x + p4.y)*p4.z, (p4.x + p4.z)*p4.y, (p4.y + p4.z)*p4.w,
(p4.z + p4.w)*p4.x));
}

float4 voronoiTex( texture tex, float2 uv, float seed, float time )
{
  float2 f ;
  float2 n = decompose( uv, f ) ;

  // derivatives ( for correct mipmapping )
  float2 Dx = ddx(uv) ;
  float2 Dy = ddy(uv) ;

    //---------------------------------
    // regular voronoi
    //---------------------------------
    float4 tot = 0 ;
    float totAlph = 0 ;
  float totAlph2 = 0;
    for( int j=-1; j<=1; j++ )
    {
    for( int i=-1; i<=1; i++ )
    {
      float2 g = float2( i, j );

      float2 ipos = n + g  ;

      float2 o = hash2( ipos+seed );
      o = 0.5 + 0.5*sin( time + 6.2831*o );  // animate the position

      float2 r = g + o - f;
      float d = dot(r,r);

      float alpha = exp(-5*d);

      // compute transform parameter for the texture
      float4  trans = hash4( ipos+seed );
      float2  dir = sign( trans.xy-0.5 );
      dir = dir==0 ? 1 : dir ;  // fix the zero case

      float4 col = tex.sampleGrad( uv*dir + trans.zw, Dx, Dy );

      tot += alpha*col ;
      totAlph += alpha ;
      totAlph2 += alpha*alpha ;
    }
  }

  float4 mean = tex.sampleGrad( 0, 1000, 1000 );

  return mean + (tot-totAlph*mean) / sqrt(totAlph2) ;  // contrast preserving
blending
```

```
    return tot / totAlph ; // normal blending
}

float4 main( idatas i )
{
  float2 ratio = mat.size / mat.size.x ;
  box2 b = box2FromCenterAxe( i.strokeStartPos, length(i.strokePos-
i.strokeStartPos)*ratio, normalizeSafe(i.strokePos-i.strokeStartPos) );
  float2 p = b.toCenter( i.pos ) / b.size ;
  float4 col = voronoiTex( mat, p, i.nbUserStroke, i.dist*0.005 ) ;
  col = mat.isEmpty ? 1 : col ;
  col.a *= i.matAlpha ;

  col.xyz = lerp( col.xyz, i.color1.xyz, i.overlay );

  return col ;
}
```

viewBox



```
cfg{
  name="Texture Voronoi Tiling";
  renderingTime = 30 ;
}

globals{
  texture mat;
  {
    uiTab = "Texture" ;
  }
}

perPrim{
  float4 color1=1;
  {
    uiName = "Color";
    uiTab = "Color" ;
  }
  float overlay=0;
  {
    uiMax = 1 ;
    uiTab = "Color" ;
    uiFormat = percent ;
  }
  float matAlpha=1;
  {
    uiName = "alpha";
    uiTab = "Color" ;
    uiFormat = percent ;
    uiMax = 1 ;
  }
}

float hash1( float2 p)
{
  float hscale = .1031 ;

  float3 p3 = frac( float3(p.x,p.y,p.x) * hscale ) ;
    p3 += dot(p3, p3.yzx + 19.19);
    return frac( (p3.x + p3.y) * p3.z );
}

float2 hash2( float2 p )
{
  float3 hscale3 = float3( .1031, .1030, .0973 ) ;

  float3 p3 = frac( float3(p.x,p.y,p.x) * hscale3) ;
    p3 += dot(p3, p3.yzx+19.19) ;
    return frac( float2( (p3.x + p3.y)*p3.z, (p3.x+p3.z)*p3.y) );
```

```
}

float4 hash4( float2 p )
{
  float4 hscale4 = float4( .1031, .1030, .0973, .1099 ) ;

  float4 p4 = frac( float4(p.xyxy) * hscale4 );
    p4 += dot( p4, p4.wzxy+19.19) ;
  return frac( float4((p4.x + p4.y)*p4.z, (p4.x + p4.z)*p4.y, (p4.y + p4.z)*p4.w,
(p4.z + p4.w)*p4.x));
}

float4 voronoiTex( texture tex, float2 uv, float seed, float time )
{
  float2 f ;
  float2 n = decompose( uv, f ) ;

  // derivatives ( for correct mipmapping )
  float2 Dx = ddx(uv) ;
  float2 Dy = ddy(uv) ;

    //----------------------------------
    // regular voronoi
    //----------------------------------
    float4 tot = 0 ;
    float totAlph = 0 ;
  float totAlph2 = 0;
    for( int j=-1; j<=1; j++ )
    {
    for( int i=-1; i<=1; i++ )
    {
      float2 g = float2( i, j );

      float2 ipos = n + g  ;

      float2 o = hash2( ipos+seed );
      o = 0.5 + 0.5*sin( time + 6.2831*o );  // animate the position

      float2 r = g + o - f;
      float d = dot(r,r);

      float alpha = exp(-5*d);

      // compute transform parameter for the texture
      float4  trans = hash4( ipos+seed );
      float2  dir = sign( trans.xy-0.5 );
      dir = dir==0 ? 1 : dir ;  // fix the zero case

      float4 col = tex.sampleGrad( uv*dir + trans.zw, Dx, Dy );

      tot += alpha*col ;
      totAlph += alpha ;
      totAlph2 += alpha*alpha ;
    }
  }

  float4 mean = tex.sampleGrad( 0, 1000, 1000 );

  return mean + (tot-totAlph*mean) / sqrt(totAlph2) ;  // contrast preserving
blending
```

```
    return tot / totAlph ; // normal blending
}

float4 main( idatas i )
{
  float2 ratio = mat.size / mat.size.x ;
  box2 b = box2FromCenterAxe( i.strokeStartPos, length(i.strokePos-
i.strokeStartPos)*ratio, normalizeSafe(i.strokePos-i.strokeStartPos) );
  float2 p = b.toCenter( i.pos ) / b.size ;
  float4 col = voronoiTex( mat, p, i.nbUserStroke, i.dist*0.005 ) ;
  col = mat.isEmpty ? 1 : col ;
  col.a *= i.matAlpha ;

  col.xyz = lerp( col.xyz, i.color1.xyz, i.overlay );

  return col ;
}
```

viewZoom

```
cfg{name="viewZoom";}

// the brush color will change according the zoom value
float4 main( idatas i )
{
  float t = saturate( viewZoom )  ;
  float3 col = lerp( i.color.xyz, float3(1,0,0), t );
  return float4( col, i.color.a );
}
```