Bigger bang with fewer sprites

Tristan Lorach

Overview

- Current Explosion effects
- New type of explosions
- Back to procedural procedural noise & texturing
- Details on the example
- The demo

Actual explosion effects

- Most of the explosions are 2D billboards
  - easy to implement
  - Just a quad or a sprite for one element
  - Billboard contains an animation from a video
  - use a particle system to add complexity
- Drawbacks:
  - Billboards intersect badly with the 3D scene
  - Pay attention and you'll find out the same patterns
  - 2D billboard aren't volumes, evens through particles

Actual explosions

- Our purpose isn’t to replace typical technique
- Our purpose is to find out new technique for specific cases
- Billboards Explosions are good for small ones:
  - Small size help to hide artifacts
  - Fast explosion fooling the eye about details
- But what about big explosions (in space, atomic) ?

New type of explosion

- Some explosions may converge to a solid object
  - May interact with the scene
  - Must respect the floor and other surfaces (explosion into a corridor…)
  - The eye wants to see it as a volume
  - Can be the central topic of the scene
- Explosions can take various forms
  - Sphere, cone and complex mesh (mushroom)
- Explosions must tend to be unique in its details

New type of explosion

- New GPU's allows us to do so
  - At vertex level: displace the vertices and pre-compute some parameters
  - At fragment level: use procedural noise either from scratch or through 3D (and 1D/2D) textures
- Still, the CPU will keep the job of providing the basic mesh structure
  - Provide a simple growth (our example)
  - Provide a physical control of the mesh
- CPU for global behavior & GPU for near-surface behavior

Drawbacks of this new technique

- More expensive in computation : more triangles, complex vertex & fragment processing
- Difficult to fine-tune the parameters
  - Everything is almost arbitrary
- Need artists to realize good simulation
  - Any math won't be enough
  - We cannot use a video of fire
  - The evolution in time must behave correctly
  - Colors at fragment level are arbitrary. New ideas are welcome

Procedural noise for explosions

- Noise is the solution
  - Provide a near-unique result for each explosion
  - Fooling the eye thanks to complexity
- Noise can
  - displace the vertices
  - contribute to the color blending
- Noise can be real-time calculated (Perlin)
- Noise can be stored into textures (3D)
- Fractal sum of noise is good to approach nature phenomenon.

Primitives for an explosion

- Plasma disc
  - Illustrating the explosion’s shockwave
- The core of the explosion
  - A growing sphere or a more complex mesh
- Some secondary explosion sources
- Some material going out of the explosion
- Some optical distortion from the heat of the shockwave
- All can be using procedural texturing & and procedural noise

Disadvantages of Procedural Texturing & Procedural vertex displacement

- **Compact in memory**
  - code is small (compared to textures)
- **No fixed resolution**
  - "infinite" detail, limited only by precision
- **Unlimited extent**
  - can cover arbitrarily large areas, no repeating
- **Parameterized**
  - can easily generate a large no. of variations on a theme
- **Solid texturing (avoids 2D mapping problem)**
- **We can add a 4th dimension (time)**

Disadvantages of Procedural Texturing & Procedural vertex displacement

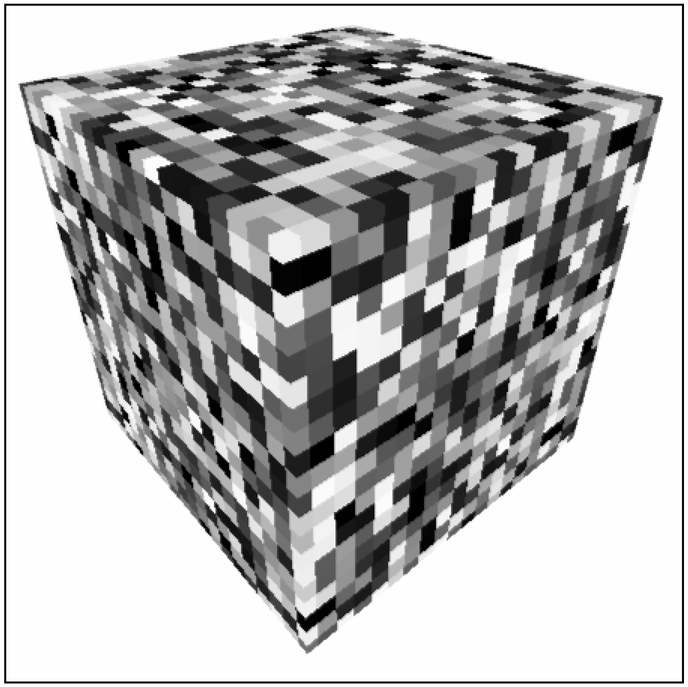
- Computation time
- Hard to code and debug
- vertices are Displaced in the rendering pipeline.
  - Resulting transformation cannot interact with the scene
  - implement the same procedure in the CPU to compute some values before the GPU

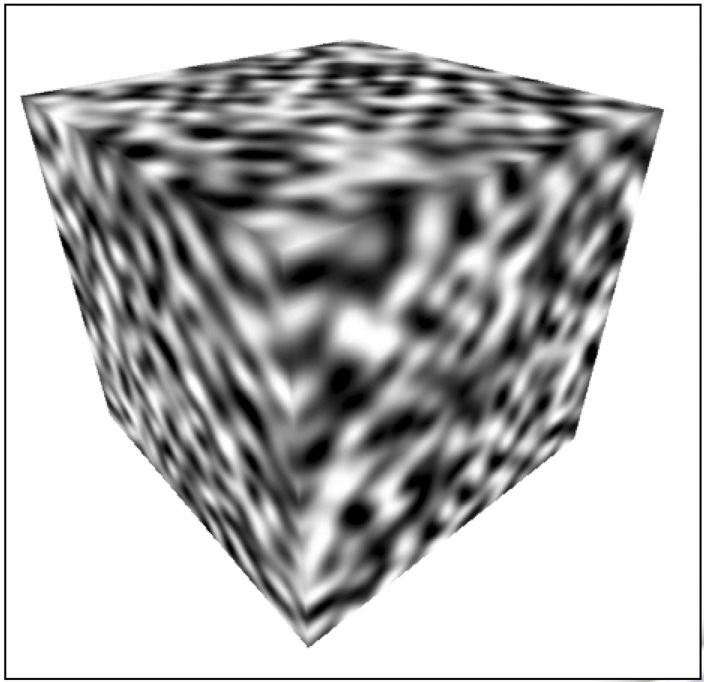
Ideal Noise Characteristics

- Can't just use rand()! An ideal noise function:
  - produces a *repeatable* pseudorandom value as a function of its input (same input -> same output)
  - has a known range (typically [-1,1] or [0,1])
  - doesn't show obvious repeating patterns (i.e. period is large)
  - is invariant under rotation and translation
- We want this noise to be smooth

What does Noise look like?

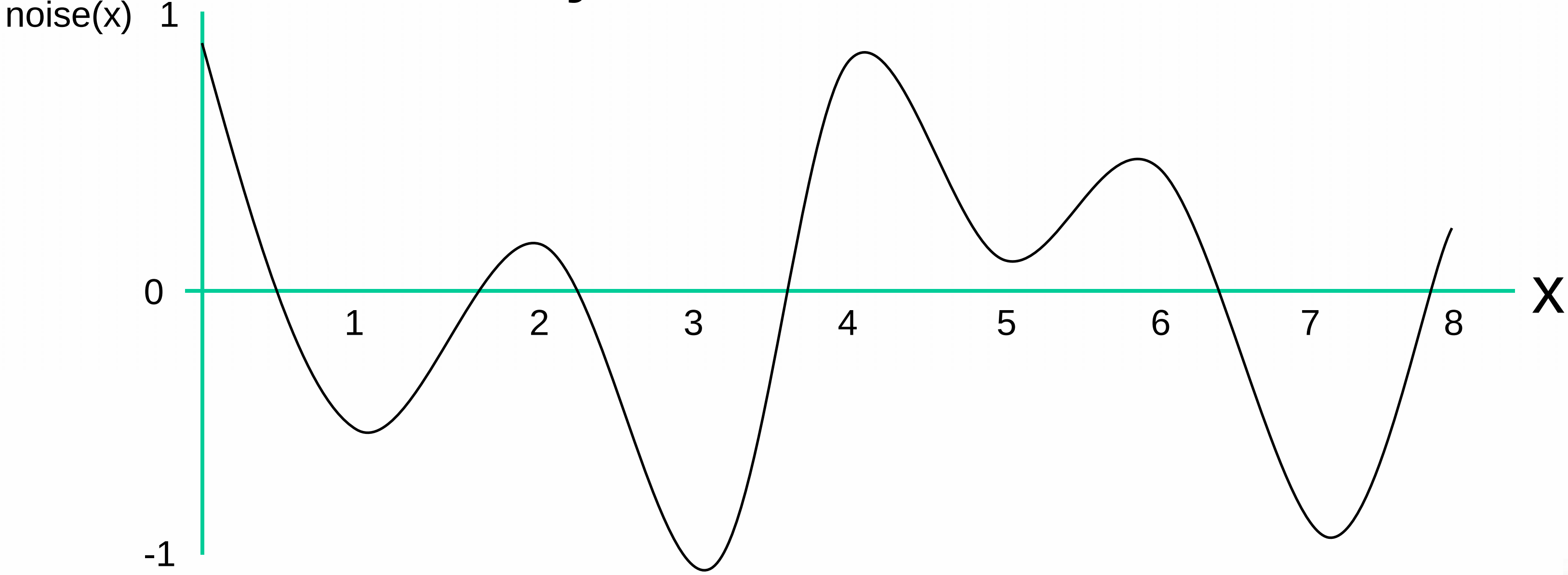
- Imagine creating a big block of random numbers and blurring them:





What does Noise look like?

- Random values at integer positions
- Varies smoothly in-between. In 1D:



This is value noise, gradient noise is zero at integer positions

Spectral Synthesis

- Noise by itself is not very exciting
- By summing together multiple noise signals at different frequencies we can produce more interesting patterns with detail at several scales
- This is like Fourier synthesis (summing sine waves)
- Each layer is known as an "octave" since the frequency typically doubles each time
- Increase in frequency known as "lacunarity" (gap)
- Change in amplitude/weight known as "gain"

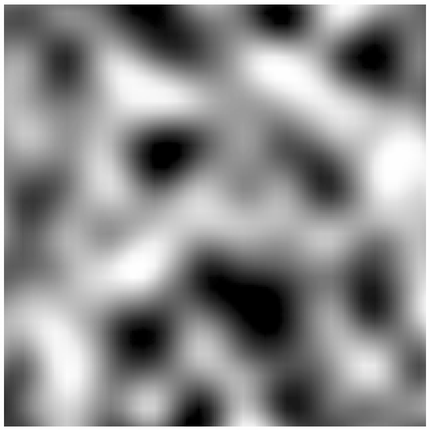
Fractal Sum

- Weighted sum of several layers of noise with increasing frequency and decreasing amplitude
- Fractal because of self-similarity at different scales
- Also known as "Fractional Brownian Motion" (fBm)
- Typically, octaves >= 4, lacunarity ~= 2.0, gain = 0.5

```
float fractalSum(float3 p, int octaves, float lacunarity, float gain)
{
  float sum = 0;
  float amp = 1;
  for(int i=0; i<octaves; i++) {
    sum += amp * noise(p);
    p *= lacunarity;
    amp *= gain;
  }
  return sum;
}
```

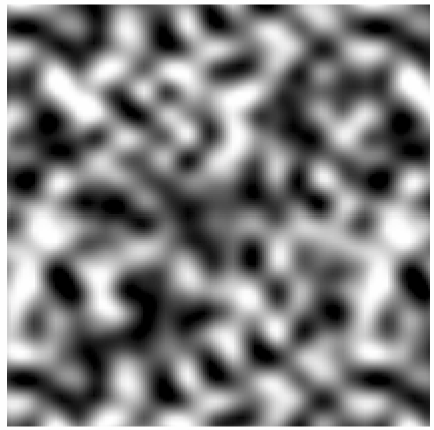
Fractal Sum

1/2



Freq = 1

+1/4



Freq = 2

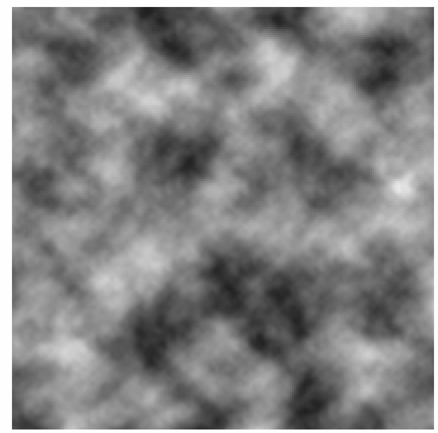
+1/8

Freq = 4

1/16

Freq = 8

=

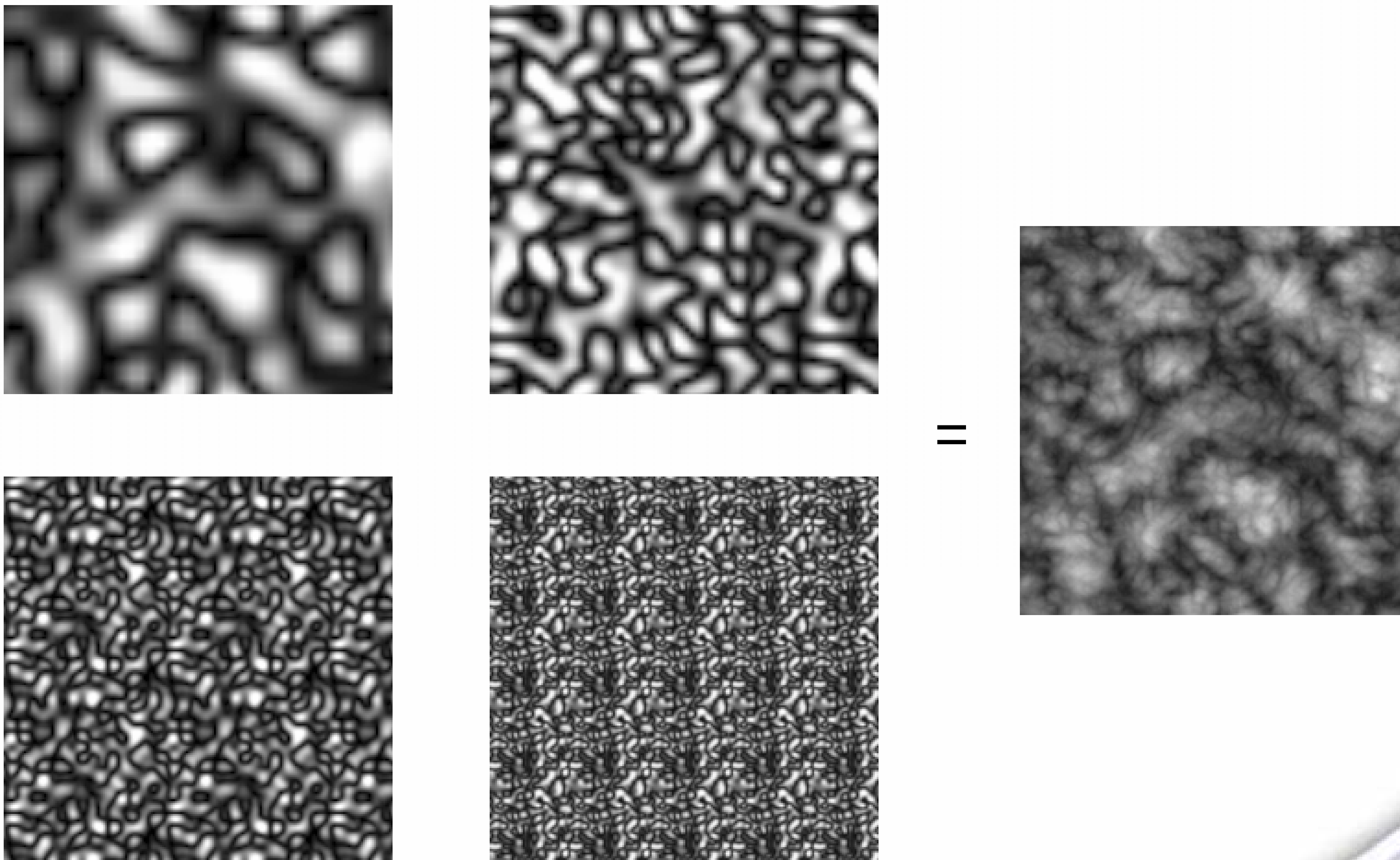


Turbulence

- **Ken Perlin's trick – assumes noise is signed [-1,1]**
- **Exactly like fBm, but take absolute value of noise**
- **Introduces discontinuities that make the image more "billowy"**

```
float turbulence(float3 p, int octaves, float lacunarity, float gain)
{
  float sum = 0;
  float amp = 1;
  for(int i=0; i<octaves; i++) {
    sum += amp * abs(noise(p));
    p *= lacunarity;
    amp *= gain;
  }
  return sum;
}
```

Turbulence



Vertex Shader Noise

- We don’t have texture lookups in the vertex shader
- Vertex noise used for procedural displacement of vertices
- Calculating perturbed normals isn’t obvious
  - We could calculate Normal at fragment level with DDX, DDY:

```
float3 dx = (ddx(IN.worldpos.xyz));
float3 dy = (ddy(IN.worldpos.xyz));
float3 N = normalize(cross(dx,dy));
```

  - But this will make the triangles appear
  - In explosions, we’ll avoid this

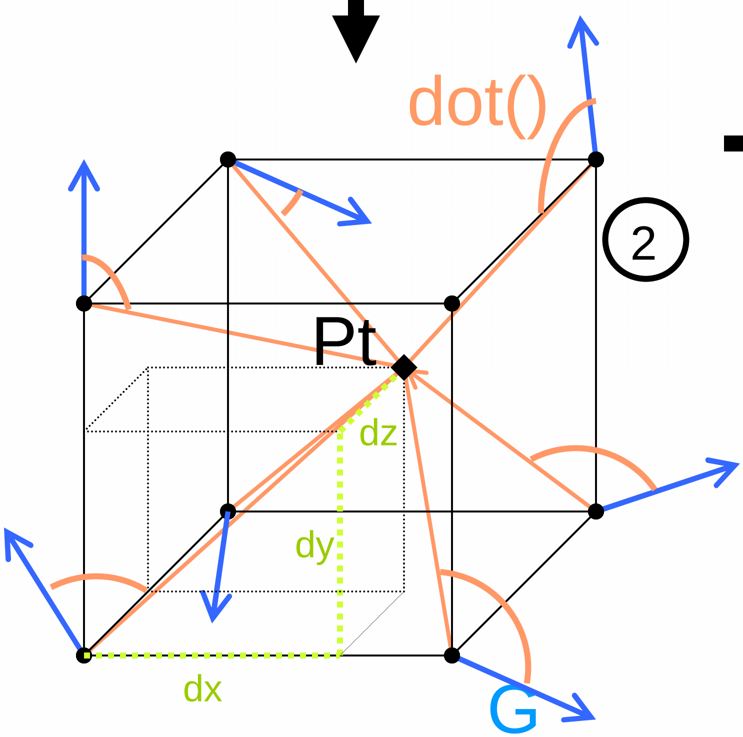
Vertex Shader Noise

- Uses permutation and gradient table stored in constant memory (rather than textures)
- Combines permutation and gradient tables into one table of float4s – (g[i].x, g[i].y, g[i].z, perm[i])
- Table is duplicated to avoid modulo operations in code
- Table size can be tailored to application
- Compiles to around 70 instructions for 3D noise

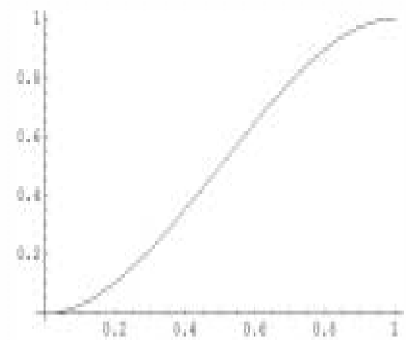
Vertex Shader Noise

p=(k+P[(j+P[i])%n])%n for 8 points around

1



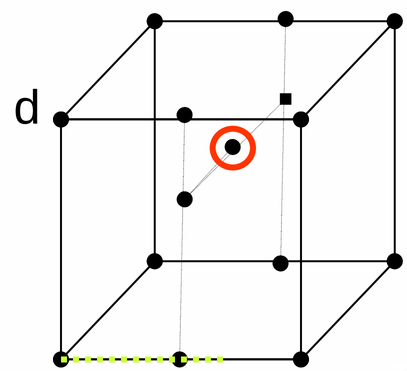
2



3

{dx’,dy’,dz’}= scurve({dx’,dy’,dz’})

noise = lerp dot products by {dx’,dy’,dz’}



4

Typical Vertex Shader Noise Cg Code

```
float noise(float3 v, const uniform float4 pg[B2])
{
  float3 i = frac(v * BR) * B;   // index between 0 and B-1
  float3 f = frac(v);            // fractional position
  // lookup in permutation table
  float2 p;
  p[0] = pg[ i[0]     ].w;
  p[1] = pg[ i[0] + 1 ].w;
  p = p + i[1];
  float4 b;                                                   1
  b[0] = pg[ p[0] ].w;
  b[1] = pg[ p[1] ].w;
  b[2] = pg[ p[0] + 1 ].w;
  b[3] = pg[ p[1] + 1 ].w;
  b = b + i[2];
  // compute dot products between gradients and vectors
  float4 r;
  r[0] = dot( pg[ b[0] ].xyz, f );
  r[1] = dot( pg[ b[1] ].xyz, f - float3(1.0f, 0.0f, 0.0f) );
  r[2] = dot( pg[ b[2] ].xyz, f - float3(0.0f, 1.0f, 0.0f) );
  r[3] = dot( pg[ b[3] ].xyz, f - float3(1.0f, 1.0f, 0.0f) );
  float4 r1;                                                                      2
  r1[0] = dot( pg[ b[0] + 1 ].xyz, f - float3(0.0f, 0.0f, 1.0f) );
  r1[1] = dot( pg[ b[1] + 1 ].xyz, f - float3(1.0f, 0.0f, 1.0f) );
  r1[2] = dot( pg[ b[2] + 1 ].xyz, f - float3(0.0f, 1.0f, 1.0f) );
  r1[3] = dot( pg[ b[3] + 1 ].xyz, f - float3(1.0f, 1.0f, 1.0f) );
  // interpolate
  f = s_curve(f);
  r = lerp( r, r1, f[2] );
  r = lerp( r.xyyy, r.zwww, f[1] );                   3   4
  return lerp( r.x, r.y, f[0] );
}
```

Pixel Shader Noise

- Almost the same as Vertex shader
- Gradient noise over R^3, scalar output
- Uses 2 1D textures as look-up tables:
  - Permutation texture – luminance alpha format, 256 entries, random shuffle of values from [0,255]. Holds p[i] and p[i+1] to avoid extra lookup.
  - Gradient texture – signed RGB format, 256 entries, random, uniformly distributed normalized vectors
- Compiles to around 50 instructions
- But here we won't use Such noise at fragment level
  - We'll prefere 3D texture noise instead (faster)

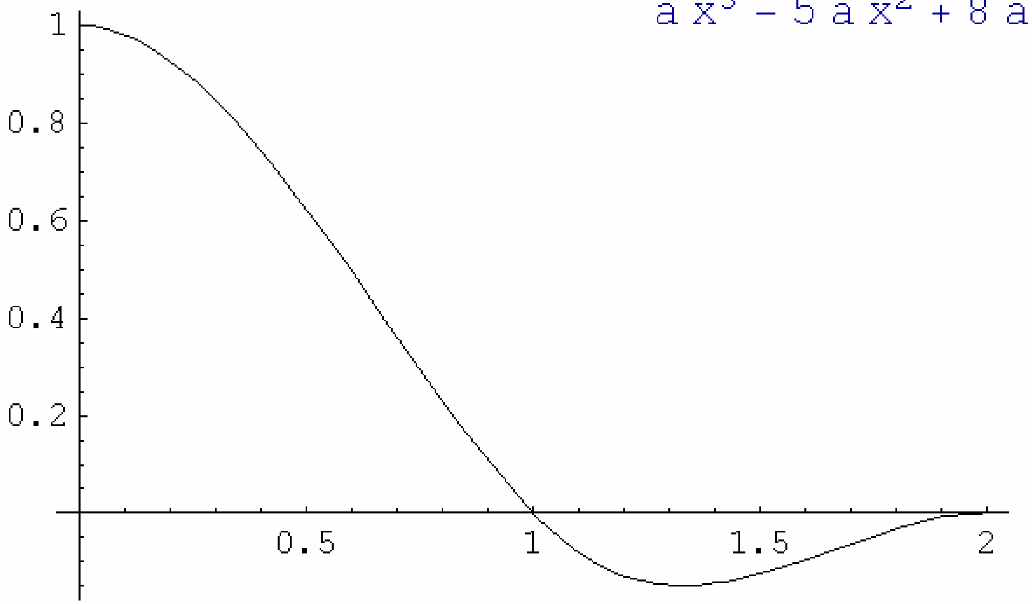
Pixel Shader Noise using 3D Textures

- Pre-compute 3D texture containing random values
- Pre-filtering with cubic filter helps avoid linear interpolation artifacts
- 4 lookups into a single 64x64x64 3D texture produces reasonable looking turbulence
- Uses texture filtering hardware
- Anti-aliasing comes for free via mip-mapping
- Period is low

Pixel Shader Noise using 3D Textures

- **Cubic filtering : f(x) =**

$$
\begin{cases} (a+2)\,x^3 - (a+3)\,x^2 + 1, & x \in [0, 1] \\ a\,x^3 - 5\,a\,x^2 + 8\,a\,x - 4\,a, & x \in [1, 2] \end{cases}
$$

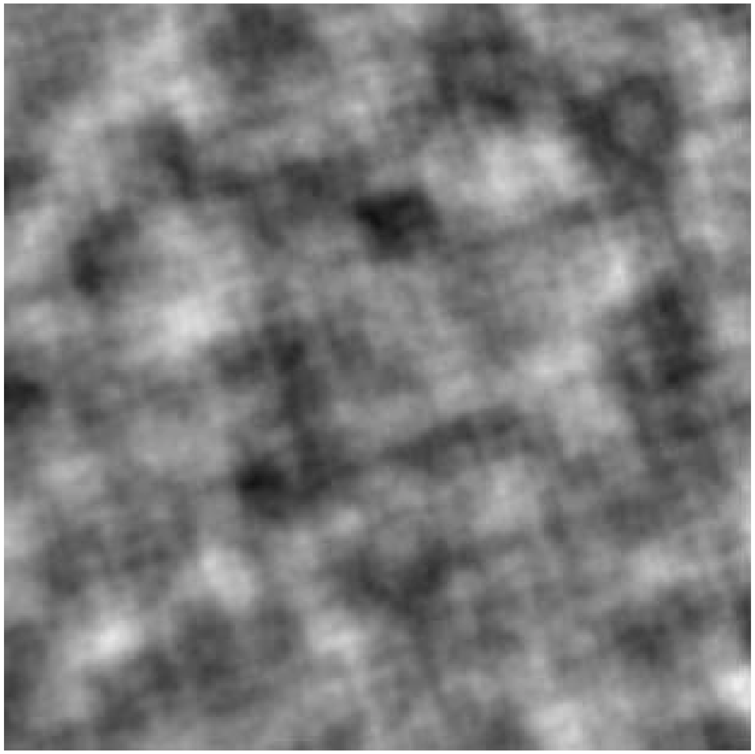


- **X = x0*f(1+x,a)+x1*f(x,a)+x2*f(1-x,a)+x3*f(2-x,a); (a=-0.75)**

Various 3D noise textures

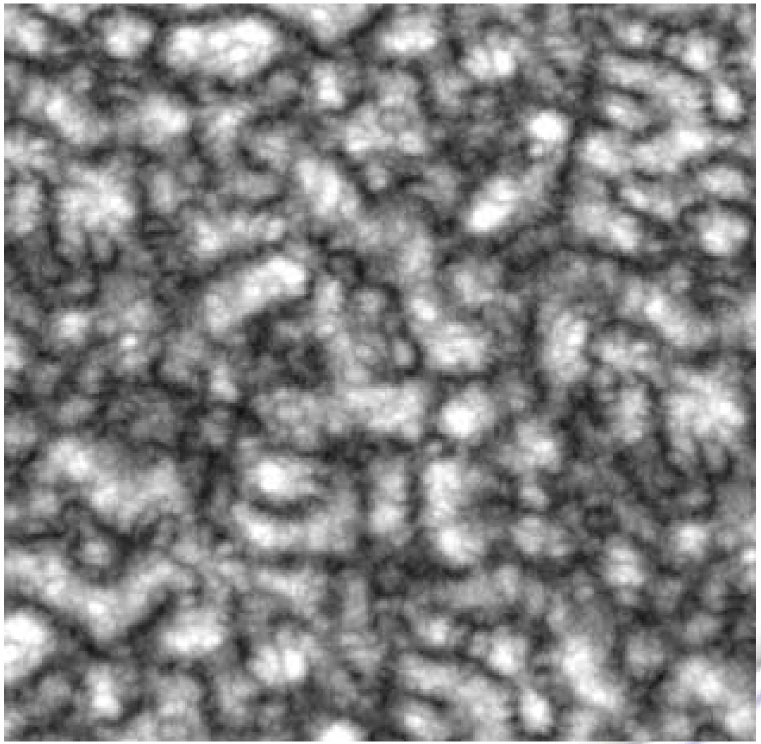
- **Noise**

(bicubicNoise3D(fx, fy, fz) + 1.0f)* 0.5f;



- **Abs Noise**

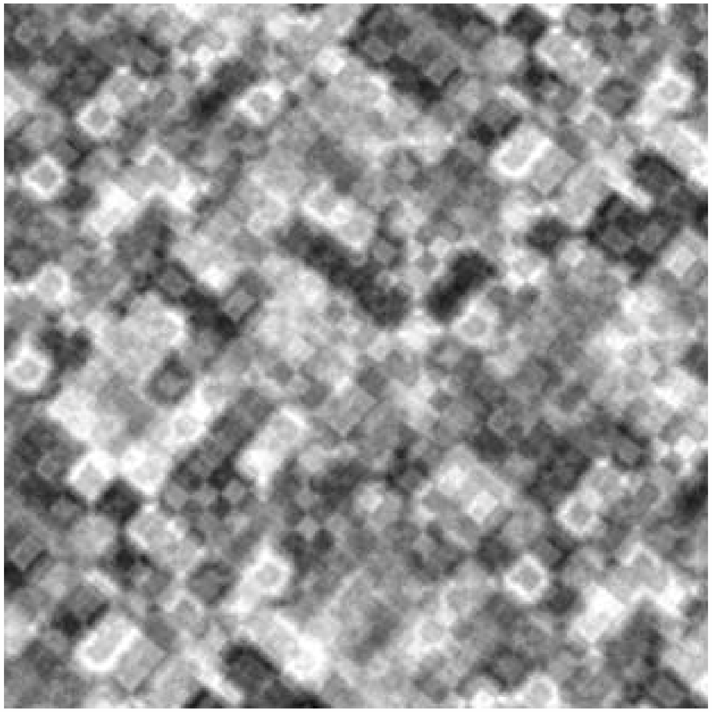
(fabs(bicubicNoise3D(fx, fy, fz)));



Various 3D noise textures

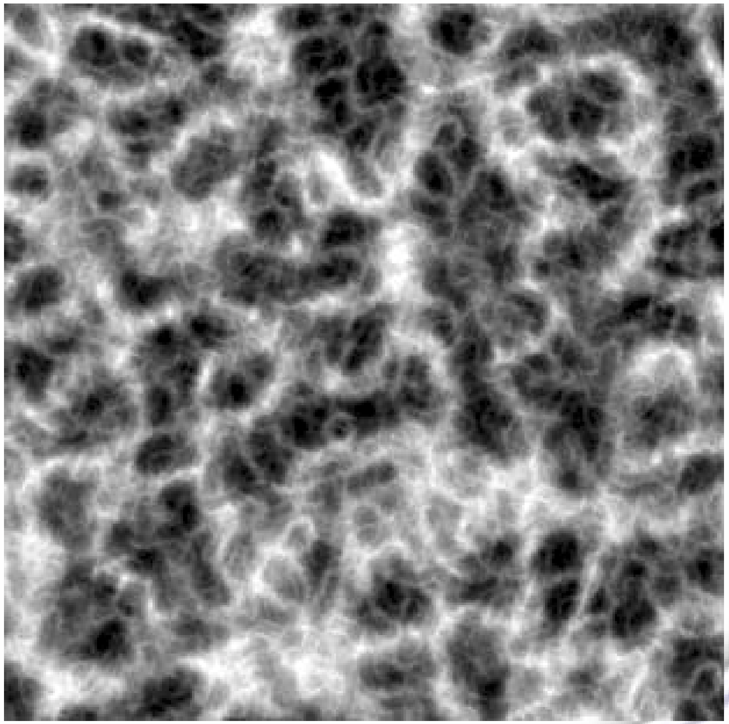
- **Raw Noise**

noise3D(fx, fy, fz) + 1.0f) * 0.5f



- **Veins**

1 - 2 *(fabs(bicubicNoise3D(fx, fy, fz)))



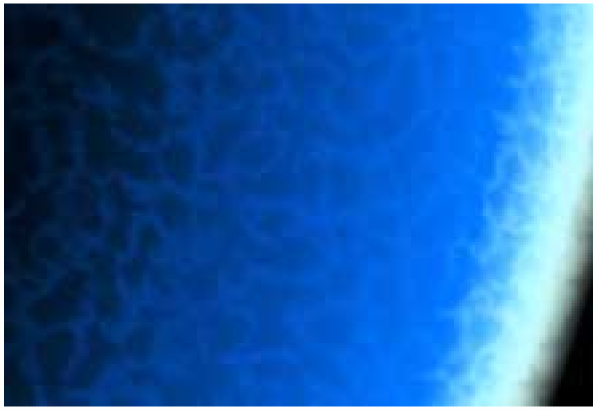
Applying color table for noise

- **Perturb the color table with the noise (create smoke/energy trail or natural spread effect)**
  - **Get a new perturbed texcoord**

```
s = clamp(IN.texCoord - (IN.texCoord * (noise*0.5+0.5)), 1/256.0,255.0/256.0);
```

  - **Lerp between perturbed & non-perturbed texcoords**

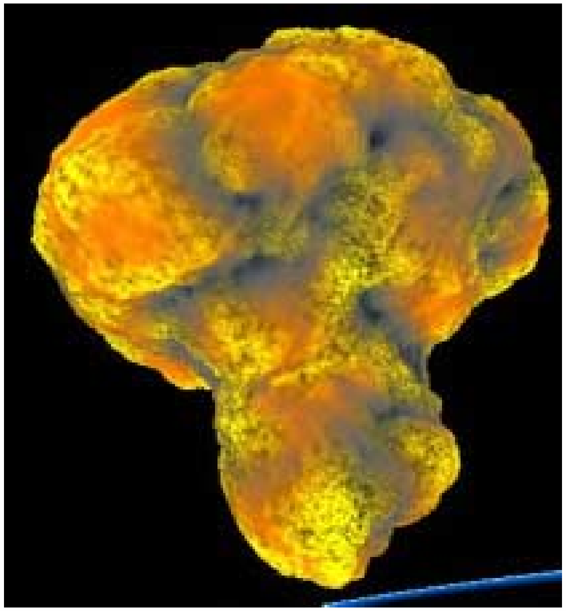
```
s = lerp(s, IN.texCoord, IN.texCoord);
texture = f4tex1D(BaseTexture, s) * NoiseAmp;
```

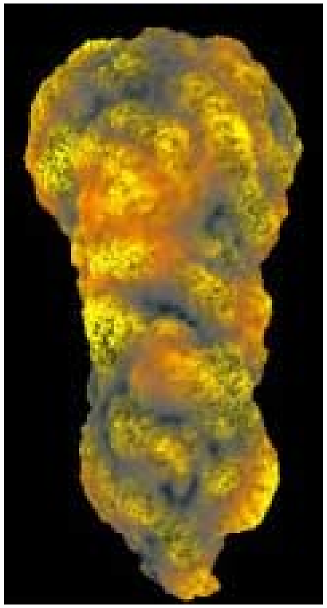


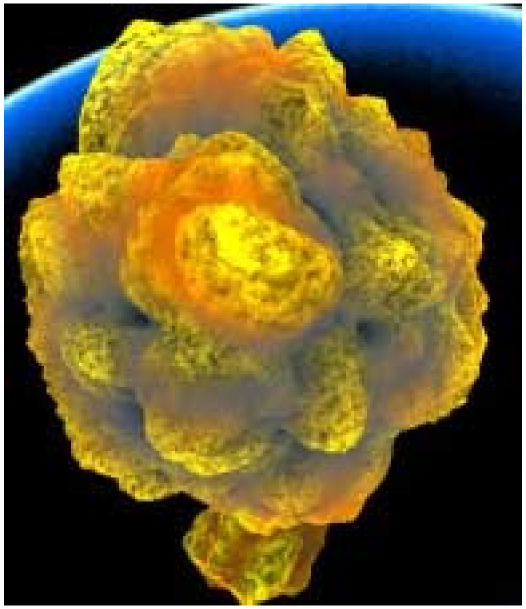
Our Example: Explosion core

- main part of the explosion
- Noise will be used to displace the vertices
- 3D noise textures will be used to represent various burning stages in the fireball.
- The idea is to play with various parameters to make the object grow like an explosion.
  - to represent the dilatation of the gas
  - to represent the rotational behavior of a gas
  - to differentiate hot part from warm parts
- Use either a sphere or any other shape

Our Example: Various shapes

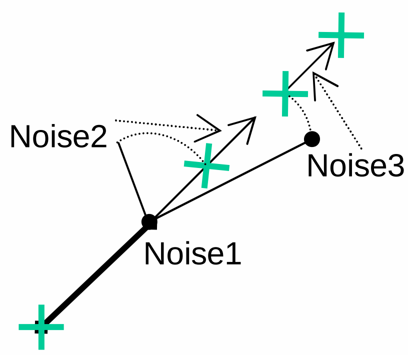






Demo: Explosion core at Vertex level

- **1st Displacement is done along the normal by fetching the noise value at the Vtx world pos**
- **2nd, 3rd and 4th displacement along the Normal**
  - **but we'll first rotate the noise space before fetching the value**
  - **Rotation center is the original vertex position**
- **This effect will create a rotational behaviour of the noise.**

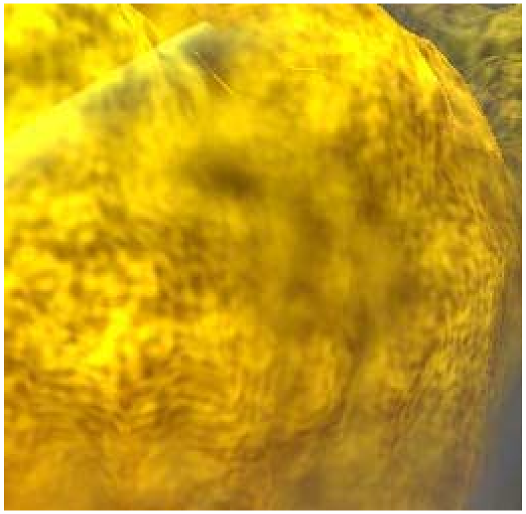


Demo: Explosion core at Vertex level

- **Computing new vertex and passing data to the fragment program**
- **Each 4 octave's noise values are passed by one interpolator : x, y, z and w for each**
- **The total normalized displacement (i.e. fractal sum) value is passed as a diffuse color component**
- **Passing the displaced vertex coordinate**

Demo: Explosion core at Fragment level

- **Using fractal sum of 3D texture noise, BUT:**
- **Instead of having a gain=0.5 at each octave, we'll use 4 octave's noise values from the Vertex program.**
- **This will emphasize some frequencies needed to create the burning parts.**
- **each octaves will appear/disappear like waves**



```
l = tex3D(NoiseTexture, IN.worldpos.xyz)*IN.noisescalars.x;
float3 scaledpos = IN.worldpos.xyz * 2.0;
l += tex3D(NoiseTexture, scaledpos) * IN.noisescalars.y;
scaledpos *= 2.0;
l += tex3D(NoiseTexture, scaledpos) * IN.noisescalars.z;
scaledpos *= 2.0;
l += tex3D(NoiseTexture, scaledpos) * IN.noisescalars.w;
```

Demo: Explosion core at Fragment level

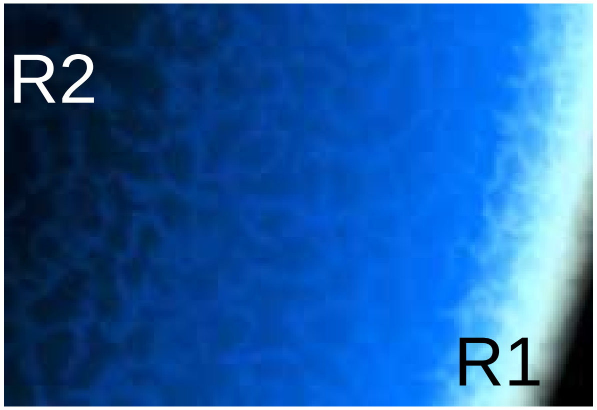
- The total normalized displacement interpolator used to fetch a 1D color table
  - 0 for the smoke color, 1 for a bright color (fire)
  - through animated scale-bias to change the look
- Use one of the 4 octave's noise values (the 2nd) to interpolate between the previous 3D noise and the color from 1D texture
- Make it sharp by X=X^3.
- Will create 2 parts
  - very hot (3D noise for the burning magma)
  - cold : the smoke from 1D color table

Demo: Additional explosions

- The main object is displaced along its normals.
  - We would like some concavity.
  - Not easy and expensive to get the partial derivatives for the noise field to change the normals.
- Add some additional explosions like any particle systems
  - Lower Tessellation for smaller objects with shorter lifetime
- We must fit to the surface of the main explosion
  - We must implement the same algorithm as the Vertex program
  - get randomly a point onto the noisy surface when a particle is being born

Demo: Plasma Disc effect

- **Disc is a simple strip.**
- **Everything at the fragment level. Very few polygons**
- **Disc is 2D, so using a 2D+time noise function (x,t,z)**
- **Noise is Made of Absolute noise values**
- **A color Range with lerp() operation can create the bright border at R1 and the fadeout at R2**

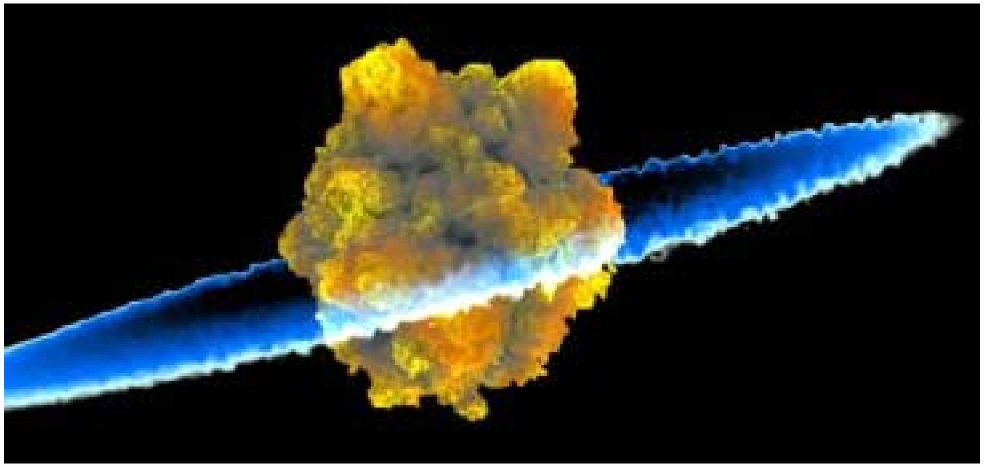


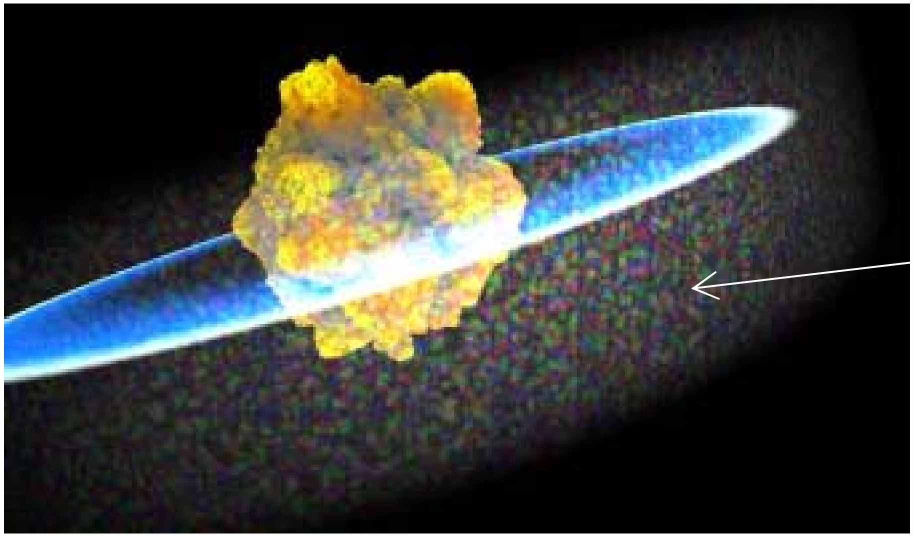

```
s = clamp(IN.texCoord - (IN.texCoord *
    (noise*0.5+0.5)),
1/256.0,255.0/256.0);

s = lerp(s, IN.texCoord, IN.texCoord);

texture = f4tex1D(Tex, s) * NoiseAmp;
```

Demo: Shockwave heat effect

- Compositing 2 P-buffers to the frame-buffer
  - First P-Buffer: the RGB scene
  - Second P-Buffer: the 2D offset map using R & G components made from invisible parts of objects
- invisible part of the disc : a cylinder around the disc
  - Fade out vertically with 1D texture
  - Fade out horizontally by lighting from the eye (dot product) and getting exponential value (*lit(1,eye_dot_n,5)*)
  - 2D Offset scale is depending on the perspective.
- Any object could contribute to this perturbation

Demo: Shockwave heat effect





Invisible cylinder of noise

To do next…

- Here : just a taste of what we can do…
- Add fourier for low frequencies
- add physical behavior for the explosion sphere
  - interact with the scene
- explosion spread with collisions of wall & floor
  - inside a corridor
  - along a landscape
- work on transparency, depending on the density
- glow, lighting effects
- …many optimizations to find

References

- “An Image Synthesizer”, Ken Perlin, Siggraph 1985
- “Improving Noise”, Ken Perlin, Siggraph 2002
- “Texturing & Modelling, A Procedural Approach” Ebert et al.
- “Advanced Renderman, Creating CGI for Motion Pictures”, Anthony A. Apodaca, Larry Gritz
- “Animating Suspended Particles Explosions”, Bryan E.Feldman, James F.O’Brien, Okan Arikan
- “Smoke Simulation For Large Scale Phenomena”, N. Rasmussen, D. Nguyen, W. Geiger, R. Fedkiw

?

tlorach@nvidia.com