NVIDIA

Real-Time Procedural Effects

John Spitzer
Director of European Developer Technology
NVIDIA Corporation

Overview

- What are procedural modeling and texturing?
- Advantages and disadvantages
- When to use procedural texturing
- What is noise?
- Ideal noise characteristics
- Noise implementations
- Procedural effects

What is Procedural Texturing?

- Code vs. tables
- Classic time vs. storage space trade-off

NVIDIA

Advantages of Procedural Texturing

- Compact
  - 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 variations on a theme
- Solid texturing (avoids 2D mapping problem)

Disadvantages of Procedural Texturing

- Computation time (big ouch!)
- Hard to code and debug
- Aliasing

When to use Procedural Textures

- Animating effects – fire, water, clouds, explosions, vegetation
- Large objects where a tiled texture would be obvious
- Lots of identical small objects where same texture would be obvious
- Models where you don't have well behaved texture coordinates

When NOT to use Procedural Textures

- Decal/diffuse textures for unique objects
  - marble vases, wood chess pieces
  - burn these instead, or have your artist paint them
- Procedural effects aren't free – don't just use them "for the hell of it"

The Basic Ingredient: Noise

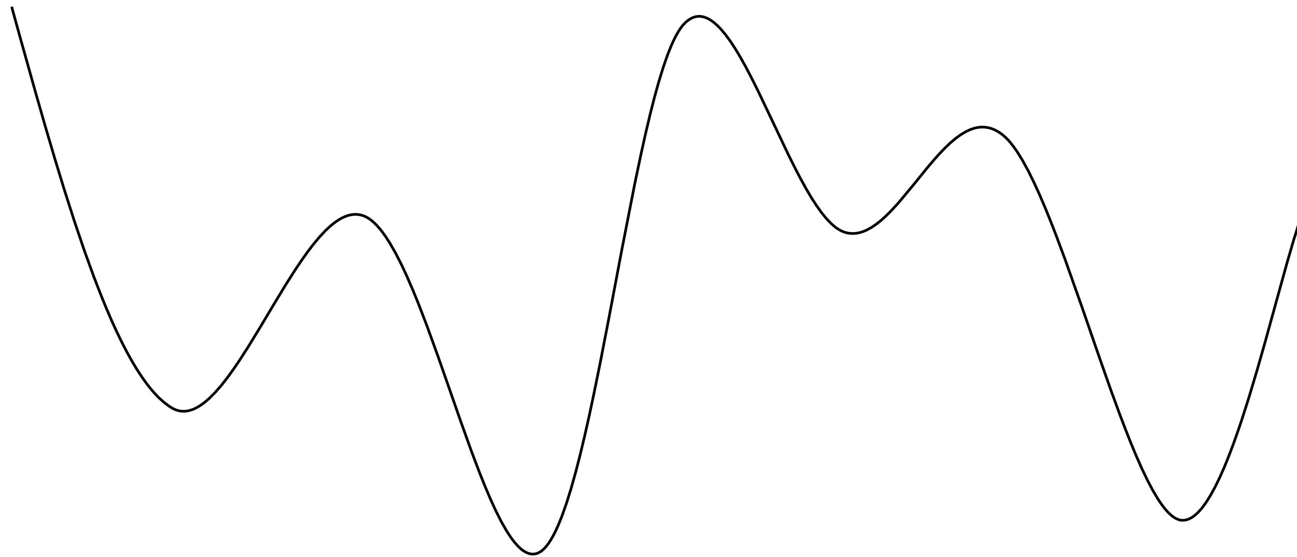
- Noise is an important part of many procedural textures
- Used *everywhere* in production rendering
- Procedural noise provides a controlled method of adding randomness to:
  - Diffuse color maps
  - Bump, displacement maps
  - Animation
  - Terrains
  - Just about anything else…

Ideal Noise Characteristics

- Can't just use rand()!
- An ideal noise function has:
  - *repeatable* pseudorandom values
  - specific range (typically [-1,1] or [0,1])
  - band-limited frequency in all directions
  - no obvious repeating patterns
  - invariance under rotation and translation
- "Random yet smooth"

What does Noise look like?

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



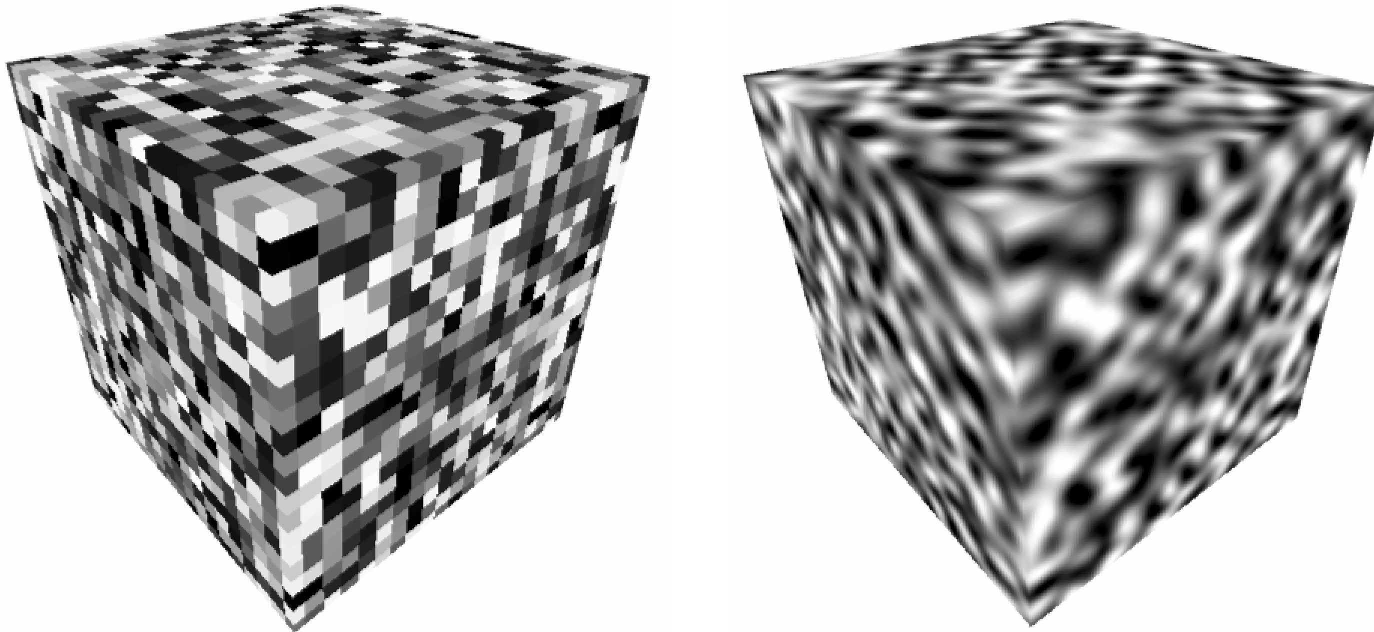
- **This is value noise, gradient noise is zero at integer positions**

Spectral Synthesis

- Narrow-band noise by itself is not very exciting
- Summations of multiple frequencies are!
- 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”

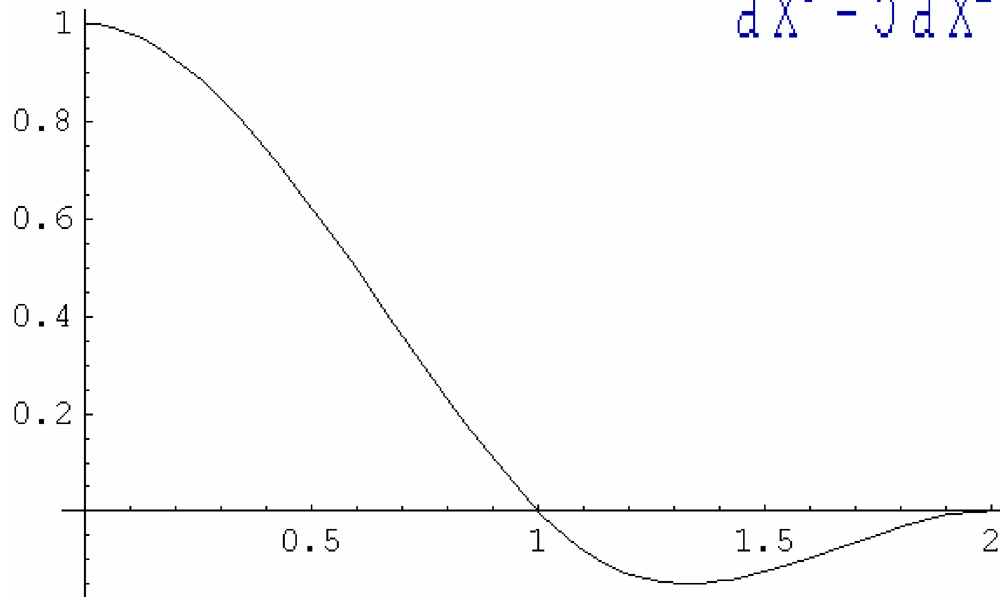
Value Noise

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



Blurring the 3D Noise Textures

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



Value Noise using 3D Textures

- Pre-compute 3D texture containing random values
- Pre-filtering with cubic filter helps avoid linear interpolation artifacts (wrap to other edges)
- 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

Noise using Fourier Synthesis

- Use GPU's built-in and efficient sin/cos instructions
- Available in both vertex and pixel shaders
- Can directly calculate derivative/gradient
- Having 1/f frequency spectrum very expensive
- Best for providing just low frequency elements

Vertex Shader Noise

- We don't have texture lookups in the vertex shader (yet!), so must compute gradient noise
- 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));
```

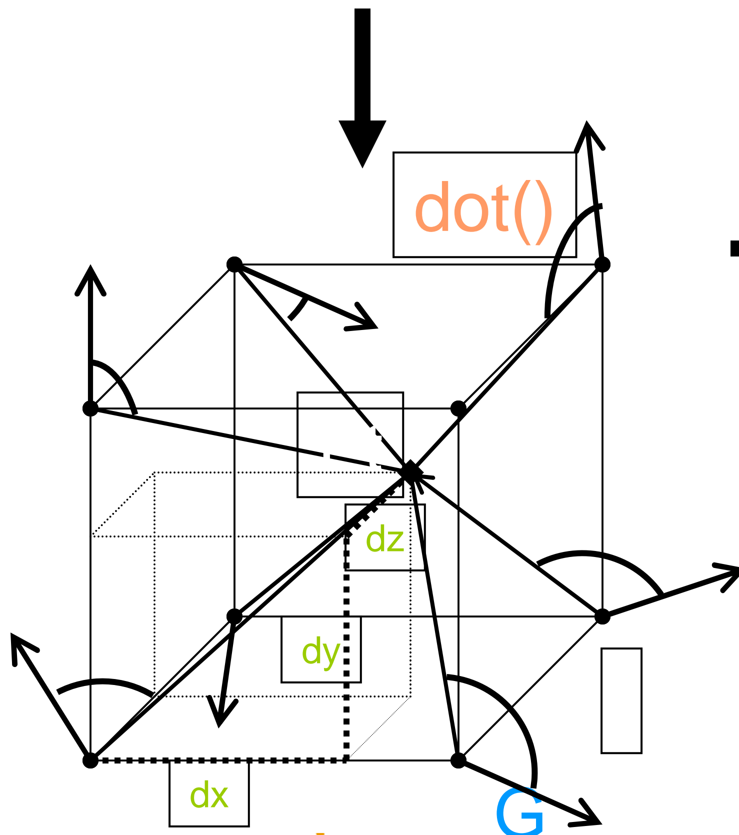
  - But gradient remains constant over triangle (faceted)

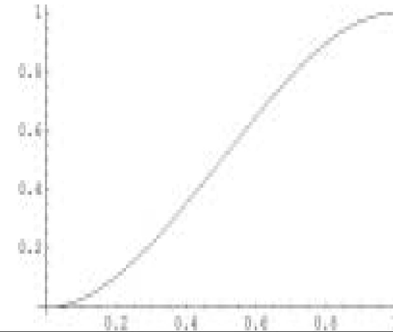
Vertex Shader Gradient Noise

- 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 Gradient Noise

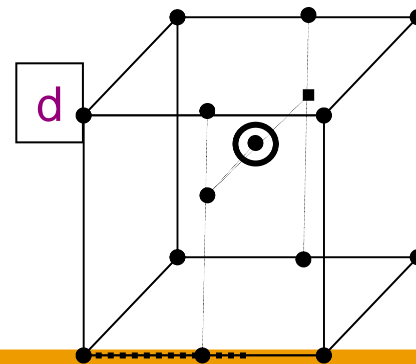
p=(k+P[(j+P[i])%n])%n for 8 points around



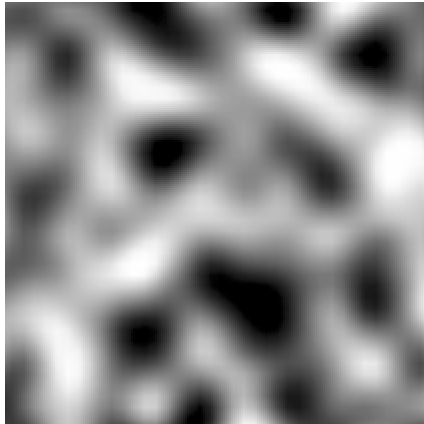


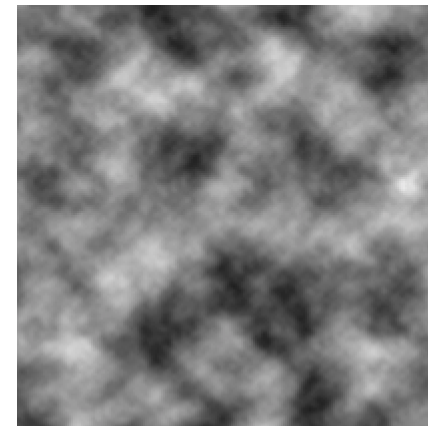
{dx',dy',dz'}= scurve({dx',dy',dz'})

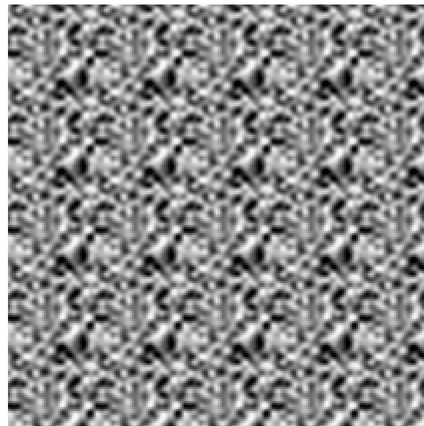
noise = lerp dot products by {dx',dy',dz'}



Fractal Sum





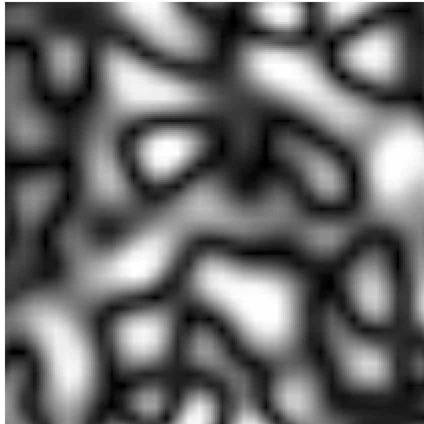
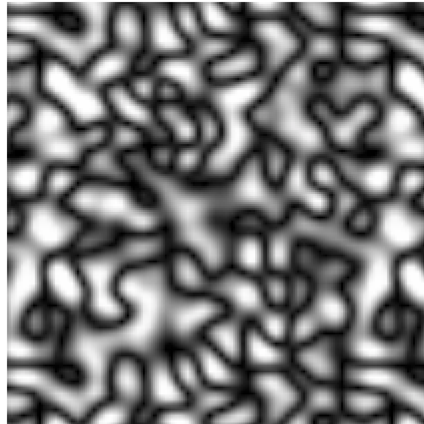
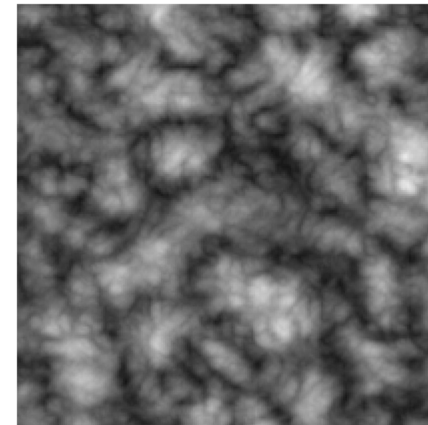
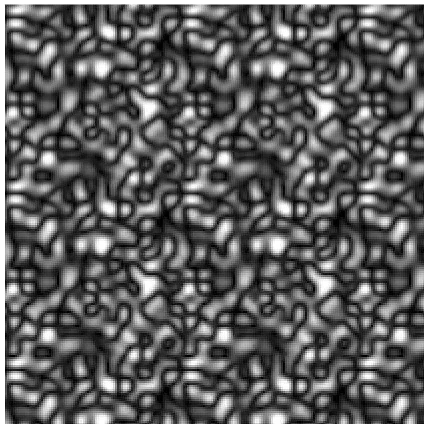
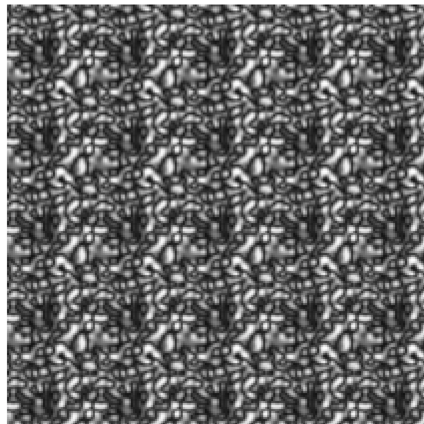


Turbulence

- 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

Pixel Shader Gradient 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

Noise is Expensive

- Perlin noise and 3D texture lookups are both relatively expensive
- Perlin is the only way to go for 4D input (like animated volumetric effects)
- Would be great if we could find a way to:
  - use 2D texture lookups only
  - use cheap math computations to combine them to form a satisfactory noise result

Separable Noise

- Project unique 2D textures down each axis
- Combine those results to form a single value
- Question is:
  - How do you select your input textures?
  - How do you select your combining function?
- Inputs could be scalars or vectors (or quats)
- Combining function could be anything, but needs to use all inputs, and not allow “zeroing” out any axis by the other two
- Work still needs to be done here…

Procedural Effects

- Detail and variation
- Terrain
- Water
- Fire
- Vegetation
- Explosions

Detail and Variation

- Used everywhere in offline rendering
- Replace/modulate repeating texture with procedural one
- Since these usually take a lot of pixels on screen (terrain, walls, etc.), it's best to:
  - have texture coordinates with uniform spacing
  - use multiple octaves of 2D noise texture, if possible
  - consider more expensive method only for lowest frequency octave (Perlin noise or Fourier synthesis)

NVIDIA

Terrain

- Use 2 or more octaves of Perlin noise in vertex shader
- Use 3D volume texture for diffuse texture
- Evaluate noise function on host for collision detection against 2.5D grid

Water

- Ocean waves
  - Use Fourier synthesis to mimic ocean waves
  - Directly calculate closed form derivative for normal for lighting
- Dynamic water
  - Interacts with the character or other phenomena
  - Implement through cellular automata
  - Calculate forces from neighboring pixels using the pixel shader and texture lookups
  - Solve for water height, then generate normal map

Vegetation

- Use Fourier synthesis to mimic period wind motion for trees and grass
- Wind gusts can propogate through grass and trees
- Sin/cos inexpensive in vertex/pixel shaders
- Summation of 2-4 frequencies usually sufficient

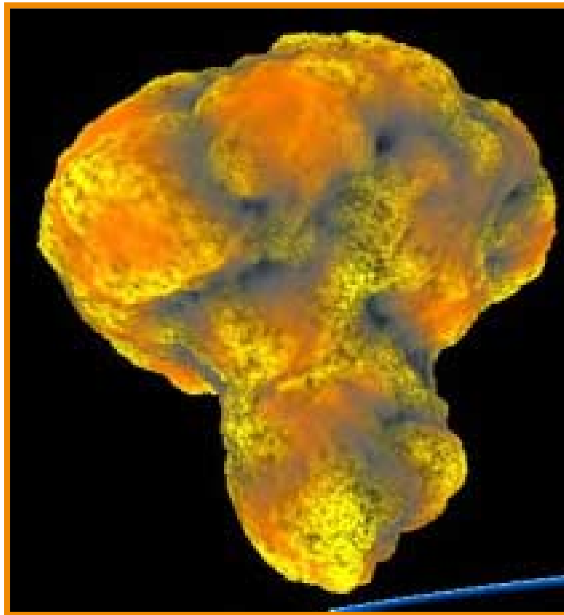
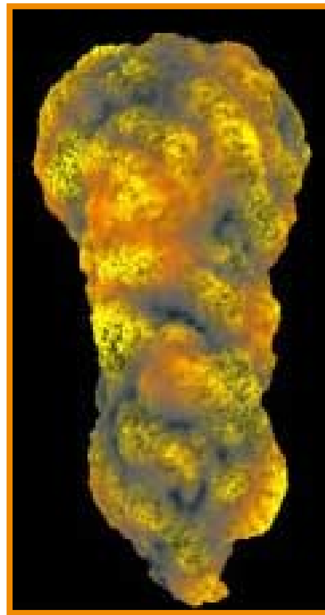
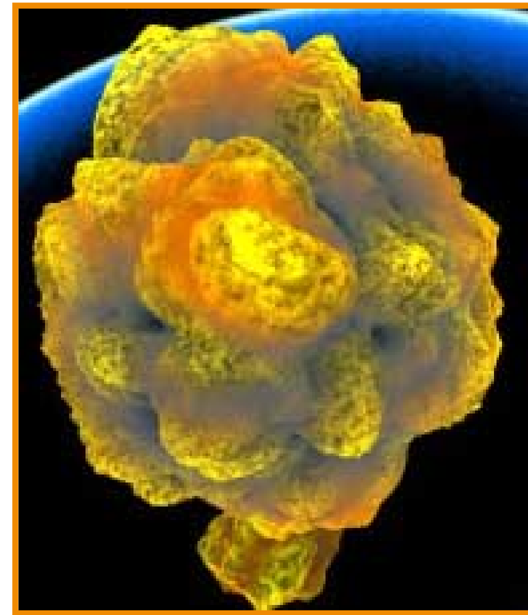
Fire

- Perlin noise (R4 input) looks great, but too expensive to real-time at this point
- Might want to experiment with cheaper noise methods
- Image space effects from cellular automata produce somewhat convincing results
- Particle/sprite effects still the best at this point
- But placement of sprites can be done procedurally with GPU

Explosions

- Noise will be used to displace the vertices of core
- 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 expansion of the gas
  - to represent the rotational behavior of a gas
  - to differentiate hot part from warm parts
- Use a sphere or any other shape

Our Example: Various shapes

Demo: Explosion core at Vertex level

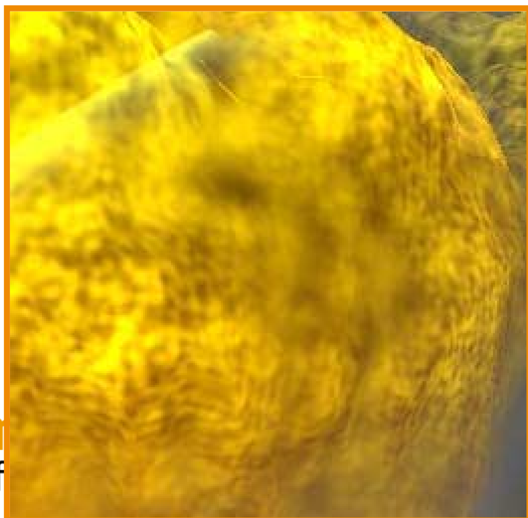
- 1st Displacement is done along the normal by fetching the noise value at vertex's world pos
- 2nd, 3rd and 4th displacement along the Normal
  - but we'll first rotate the noise space before fetching the value
  - Rotation center is the original vertex position
- This effect will create a rotational 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 octave will appear/disappear like waves



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

Demo: Additional explosions

- The main object is displaced along its normal
  - We would like some concavity
  - Difficult to get the partial derivatives for the noise field to change the normals
- Add some additional explosions like any particle systems
- We must fit to the surface of the main explosion
  - implement the same algorithm as vertex program
  - get random point on the noisy surface when a particle is born

Demo: Plasma Disc effect

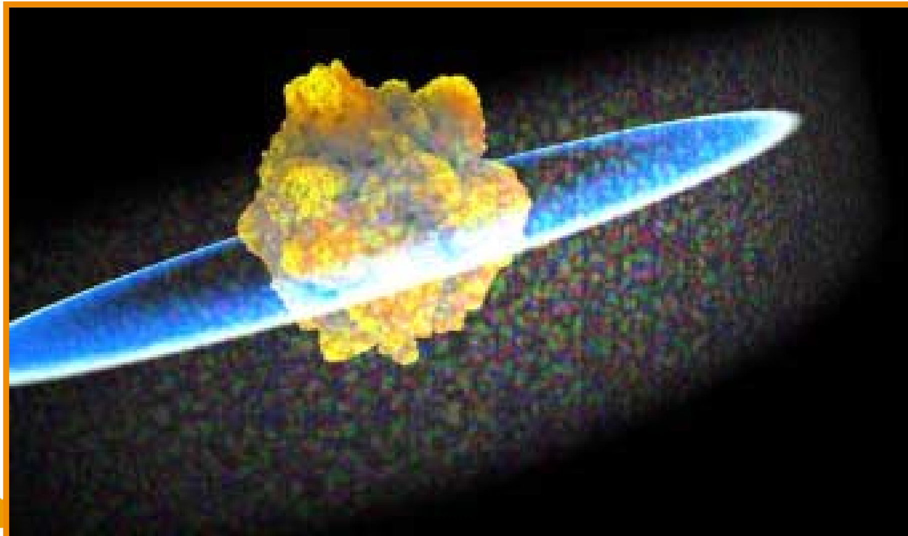
- Disc is a simple strip.
- Everything at the fragment level. Very few polygons
- Disc is 2D, so using a 2D+time noise function (x,t,z)
- Noise is Made of Absolute noise values
- A color Range with lerp() operation can create the bright border at R1 and the fadeout at R2

```
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



Conclusion

- Procedural techniques are useful today for a broad range of real-time effects
- Their spectrum of application will become only broader as GPUs become faster
- Noise implementations will continue to evolve

Questions, comments, feedback?

- John Spitzer, jspitzer@nvidia.com
- Tristan Lorach, tlorach@nvidia.com

- developer.nvidia.com