CS195V Week 4

Noise Functions

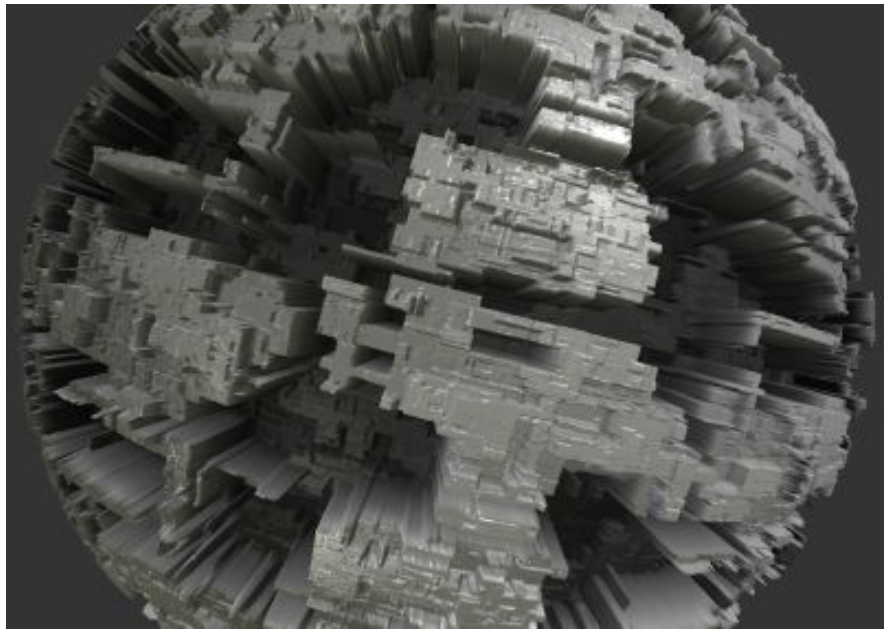
Now What?

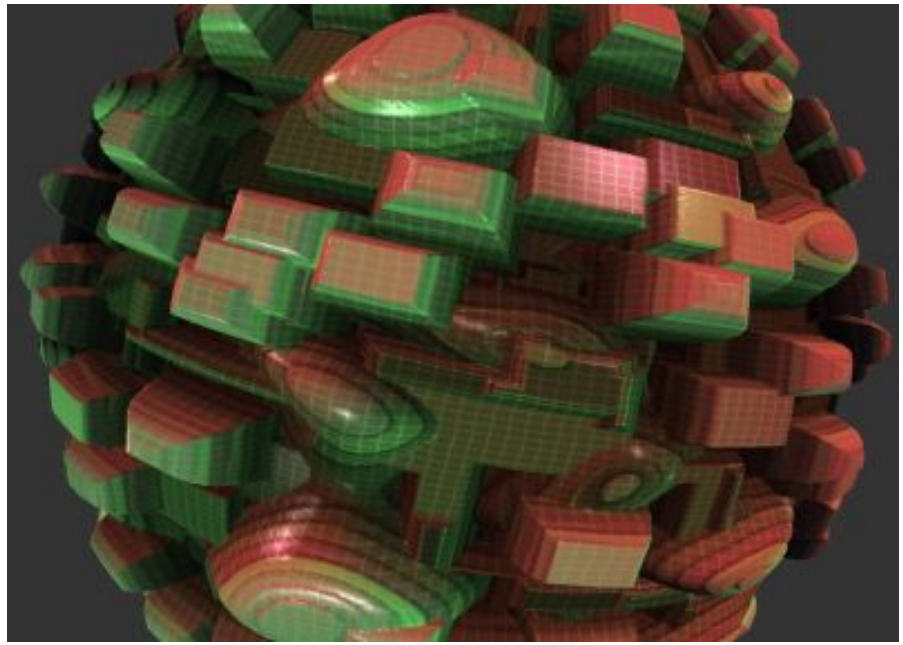
- We're done catching you up on GLSL / OpenGL
  - Everything else is really similar / you should be able to learn by yourself
  - The only way to really learn is to go out and code stuff yourself
  - That's why we have projects...
- So now what?
  - We're going to start covering random topics
  - Also, you're going to start covering random topics with us
  - Essentially for this part of the semester, we're going to be looking at applications of real time graphics
  - We suggest looking at stuff from GDC, SIGGRAPH, etc.

Before We Begin Talking About Other Things

- A word on the next project
- Next project: Warp
  - Doing some cool domain warping stuff with noise functions
  - 3D visualization must have some sort of tessellation
    - Icosahedron, plane, etc.
  - Otherwise up to you
  - Some post processing shaders (see handout)
- Purpose of this assignment is to become more familiar with the implementation and creative use of tessellation shaders, and some more OpenGL review for good measure
- This will be going out within the next day

Some Ideas...

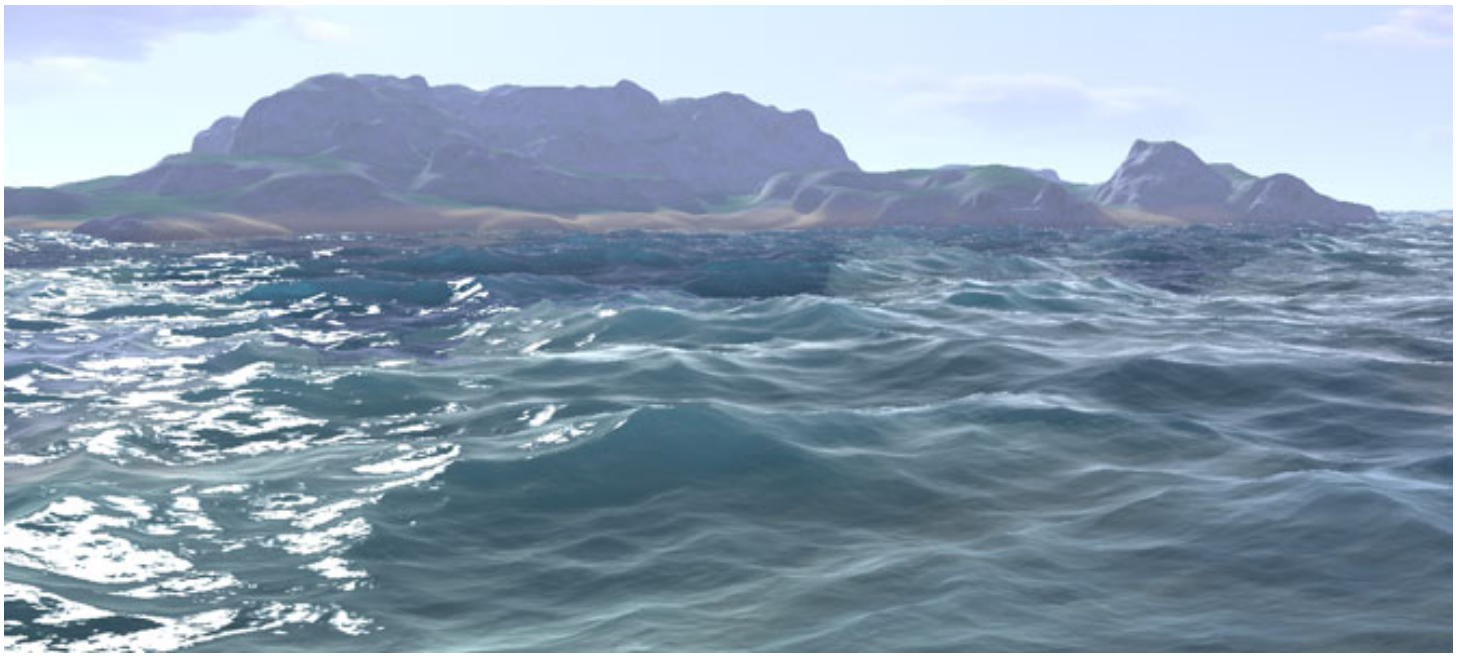


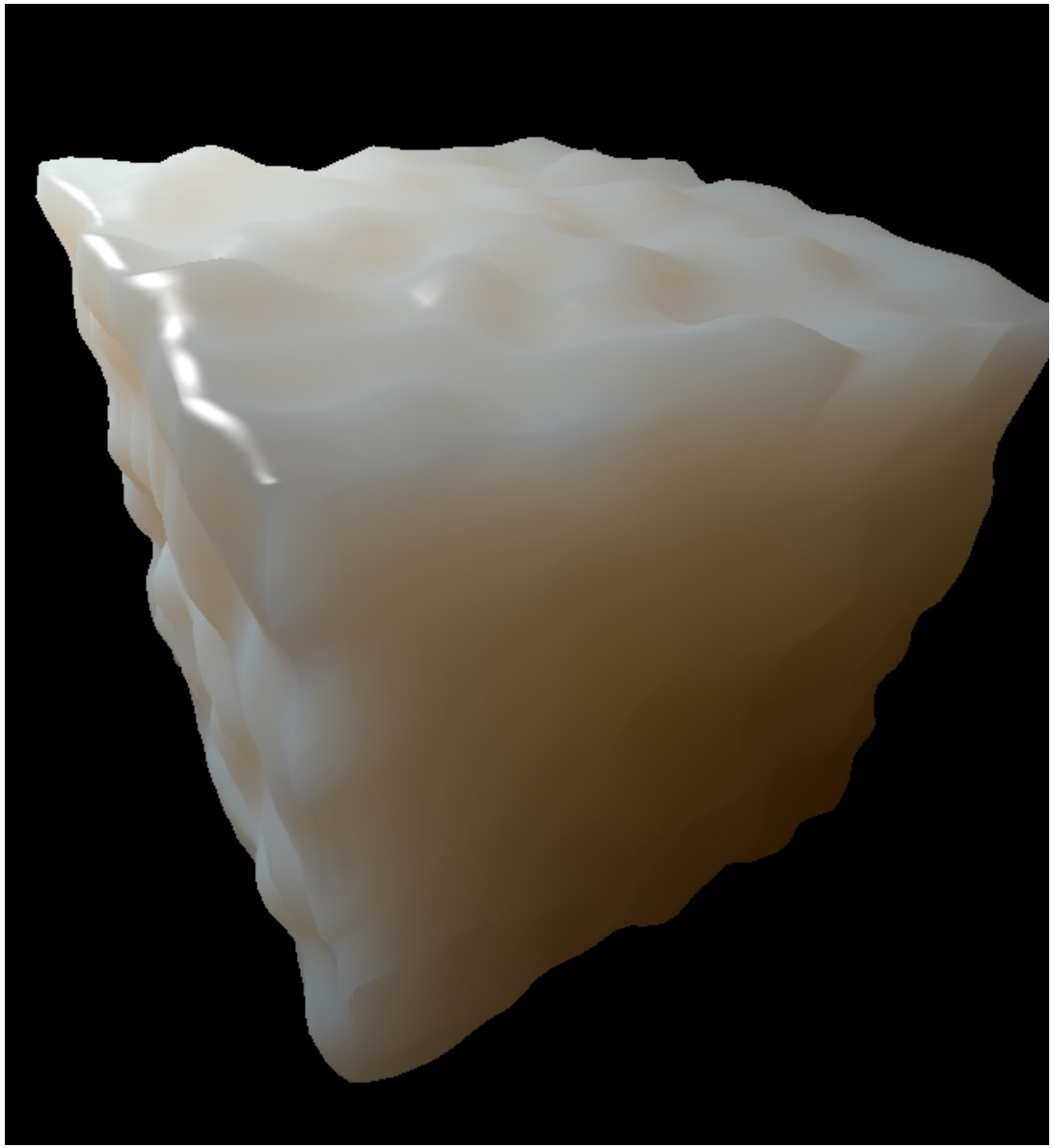


...or you could do this a get a million extra credit points

More Ideas

- Water
- Terrain
- Whole planet rendering

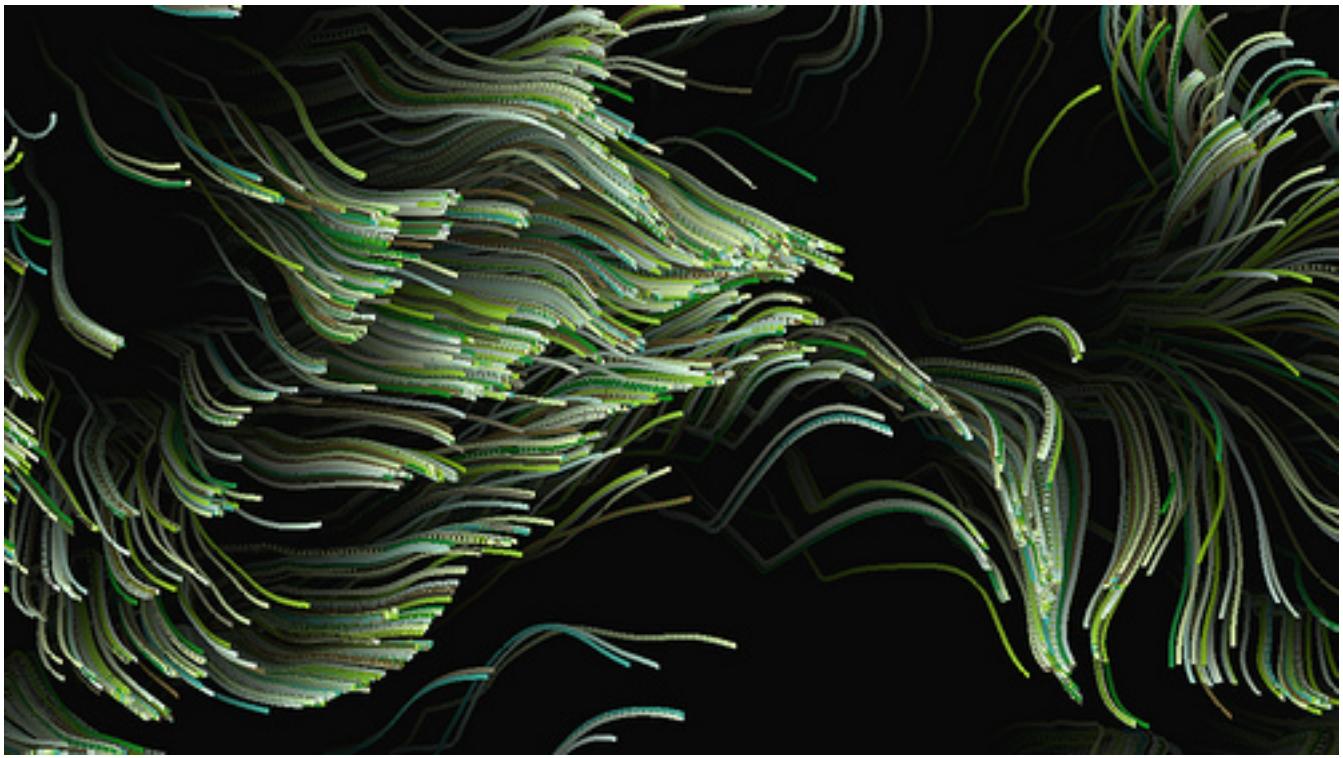




Noise Functions (on the GPU)

Noise Functions

- You've probably already heard about Perlin noise, Simplex noise, etc.
- Let's do a review of those, then talk about how to implement on the GPU



Noise Functions

- It used to be really hard to generate noise on the GPU - implementing a PRNG is kind of difficult
- Now, most GPUs have implemented the noise1 / noise2 / noise3 / noise4 GLSL functions
- These functions return values in the range of [-1.0, 1.0] and cover the range from [-0.6, 0.6] with a Gaussian
- They are repeatable the same input value in a noise function will produce the same output value
- C1 continuous
- But what are the other types of noise?

Noise

- Why do we like noise?
  - Add detail to geometry (ex. bump map)
  - Smoke / clouds
  - Terrain
  - Materials (marble, organic materials)
  - And more....

Noise Generation

- Noise can be generated from arbitrary basis functions - probably the most famous of which is Perlin's noise function
  - Perlin noise is a type of gradient noise...well go into this more
- Other functions are possible, but Perlin's algorithm achieves good results with acceptable efficiency
- Simplest noise is lattice noise, with generates a PRN at each lattice point in texture space and then interpolates values from lattice point to point
  - Choice of interpolant is important
  - Lattice convolution noise uses CM Rom filter to interpolate
  - Usually has some regularity due to PRN ...
- Sparse convolution noise generation can give good results
  - Essentially lattice noise, but uses random positions within each lattice cell (see Lewis89) - tends to produce good results

Perlin Noise

- Invented by Ken Perlin in 1985 for use in TRON
- Overview of algorithm
  - Define a grid of integer points (let's say 2D for simplicity)
  - Choose a floating point value within that grid
  - For each grid "corner" (4 for 2D), compute a psuedorandom gradient vector
  - Also for each corner, calculate the vector to the chosen floating point value
  - Take the dot product of the gradient and this vector for each corner
  - Take the weighted average of the corners (using an ease curve)
  - Now you have your noise value!

Some problems with Perlin Noise

- For a noise function of N dimensions, you need to consider 2^N grid points and do 2^(N-1) weighted sums, O(2^N)
- Gets very computationally intensive at higher dimensions
- Solution (that Perlin used in his first implementation): Lookup Tables
  - Generate an array of integers 0-255 in random order
  - Generate an array of 256 random gradients
  - To get your random gradient for point (i, j, k):
    - $g(i, j, k) = G[\ (\ i + P[\ (j + P[k]) \bmod 256\ ]\ ) \bmod 256\ ]$
  - Modulus 256 can actually be calculated with a bitwise AND
  - With this implementation, gradients repeat every 256 units but not a big deal

Perlin Noise

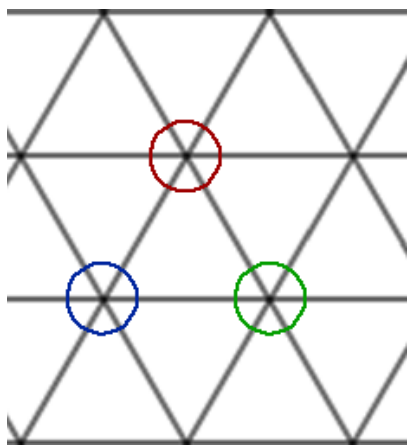
- Many optimizations can be made to Perlin generation, mostly involving smart usage of LUTs and sampling
- See GPU Gems 2 : Chapter 26 for more information
- http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter26.html

Simplex Noise

- Later developed to address some of the problems with Perlin noise
  - General computational complexity
  - Scaling to higher dimensions
  - Directional artifacts (parallel to axes)
  - Discontinuities in gradients/derivatives
  - Hardware implementation
- Most of this explanation is from http://staffwww.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf

Simplex Grids

- A simplex shape is the simplest shape that can be tessellated (oho) to fill an N-dimensional space
  - For 1D, a line segment; 2D, a triangle; 3D, a tetrahedron; etc.
- Why simplest shape?
  - It has the fewest corners possible and thus makes it easier to interpolate between values at the vertices
  - Scales better to higher dimensions (for an N dimensional space, the simplex has only N+1 vertices), compare this to other shapes



Interpolation

- In classic Perlin noise (2D), you interpolated between values at grid points by taking two averages (one in each dimension)
  - Scales badly with more dimensions and makes it difficult to calculate an analytic derivative
- In Simplex noise, instead we interpolate between vertices using an attenuation function (basically a filter kernel)
  - Imagine putting a round filter kernel around each vertex

Finding Your Point

- Now how do you find the noise at a particular point you select?
- Index into the simplex using your coordinates
  - Skew simplex into an axis-aligned space
  - Now your coordinates put you in an N dimensional hypercube
  - Then to find which simplex you are in, compare the magnitudes of your coordinate dimensions
  - Drawn example NAO
  - There are N! simplices in an N dimensional hypercube

Putting it Together

- You can find the simplex that your point is in using the methods described
- Given a gradient for each simplex vertex, you can interpolate them to get the noise value
- The original implementation of Simplex noise used the same permutation arrays to generate the gradients

Implementing on the GPU

- Calculating noise for points can be done in parallel
- You could store the permutation tables as textures and index into them using your shader
- But remember! Texture lookups are expensive!
- Also, if you are implementing this in a game or similar program, you will probably need to use your texture indexing for other tasks
- If you want to minimize texture lookups for noise generation, you can just compute the pseudorandom gradients on the fly, since you can do it in parallel for each vertex
- We will give you this code for the project

More Noise

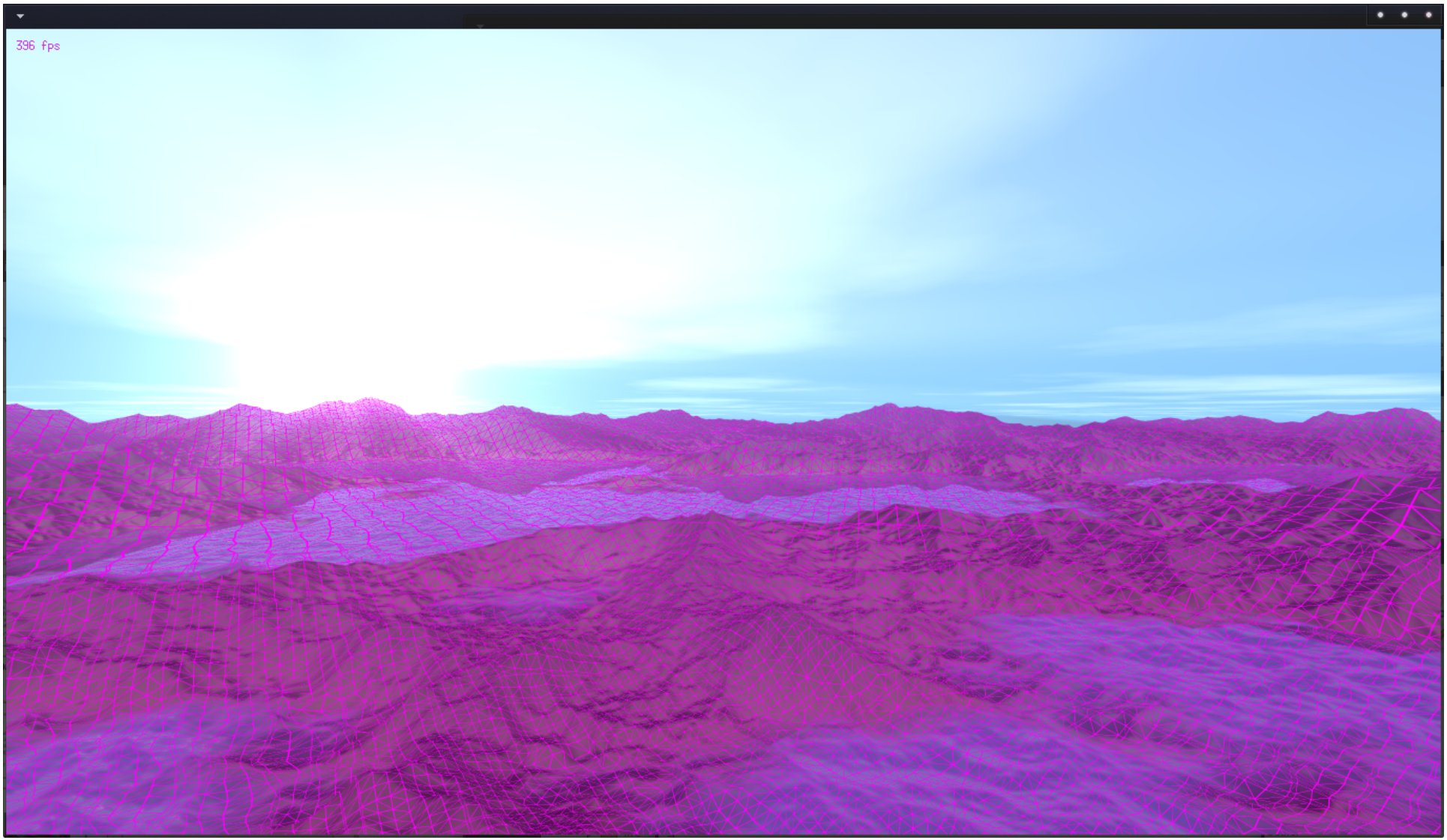
- Cellular/Worley noise
  - Similar to Perlin noise but for randomly distributed points
  - Useful for irregular patterns like rock/stone
- Flow noise
  - Used for flowing, animated objects
- Probably some others...

Worley Terrain

Fun Tricks

- Say you want to map onto a shape like a sphere
- If you just made a 2D map and tried to texture it like 123, you would get distortion and possibly discontinuities (seams)
- Instead, what if you used the 3D position to index into a 3D Perlin noise function?
- Increasing the dimensions of your noise function can make it easier to resolve these kinds of things

Octave Noise/Fractal Terrain



Octave Noise / Fractal Terrain

- Most these noise functions don't produce good terrain structure
- Some terminology :
  - Octaves - how many layers of noise you sum together (usually the more octaves, the more detail)
  - Frequency - How many random noise points fit into the space (usually increases at higher octaves)
  - Amplitude - Magnitude of the noise (usually decreases at higher octaves)
  - Lacunarity - controls how fast frequency grows per octave
  - Gain / Persistence - controls how fast amplitude grows per octave

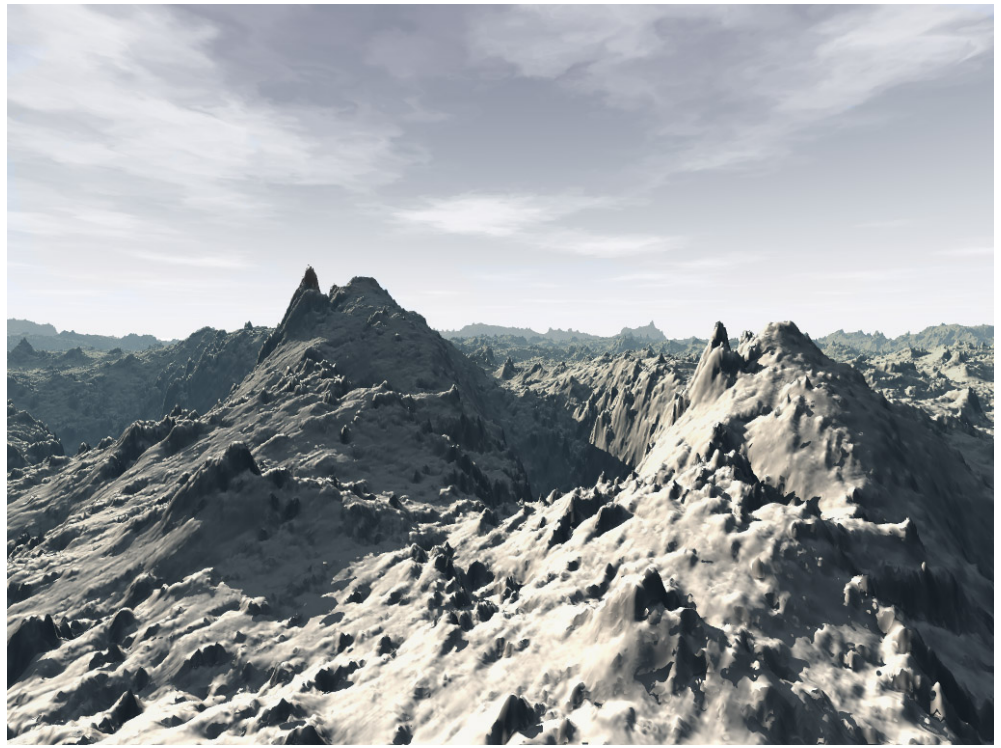
FBM

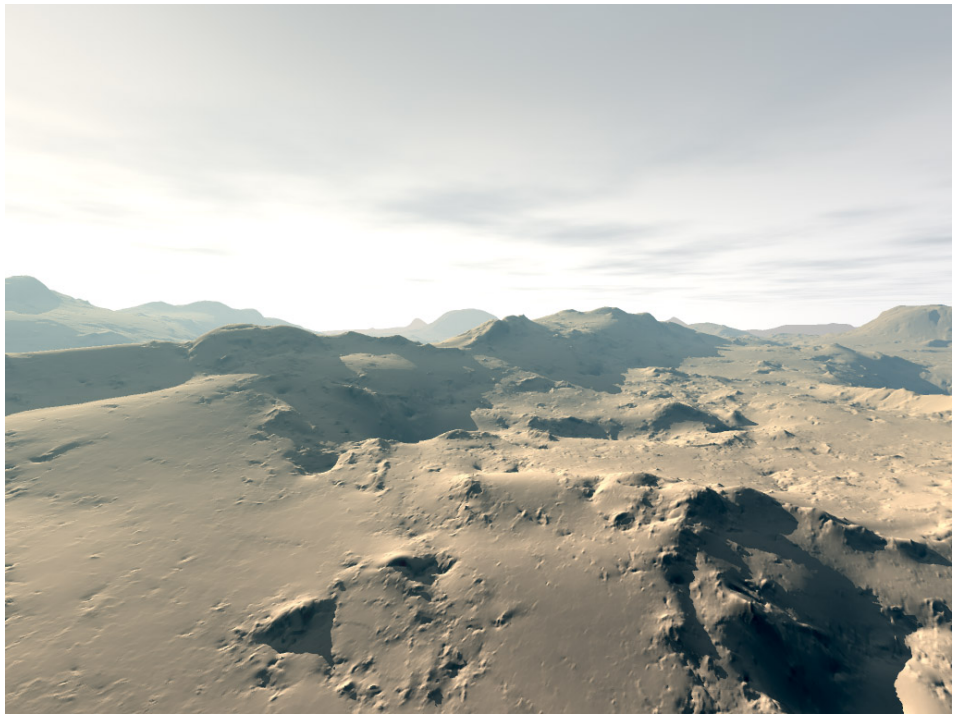
- Fractal Brownian Motion (fbm)
  - One of the earliest forms of noise generation - similar to random midpoint displacement
  - Sums up successive octaves of noise, each with higher freq and lower amp
  - Choose your noise function of choice - Perlin, noise1d, etc., and sum up several octaves of it

```
total = 0.0f;
frequency = 1.0f/(float)hgrid;
amplitude = gain;

for (i = 0; i < octaves; ++i)
{
        total += noise((float)x * frequency, (float)y * frequency) * amplitude;
        frequency *= lacunarity;
        amplitude *= gain;
}
```

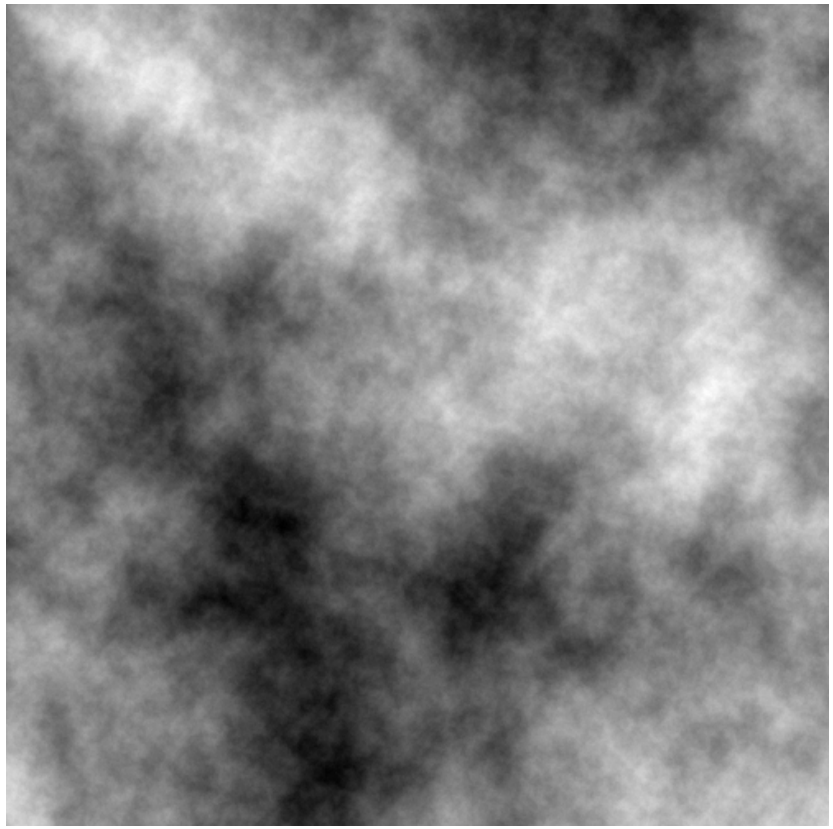
FBM





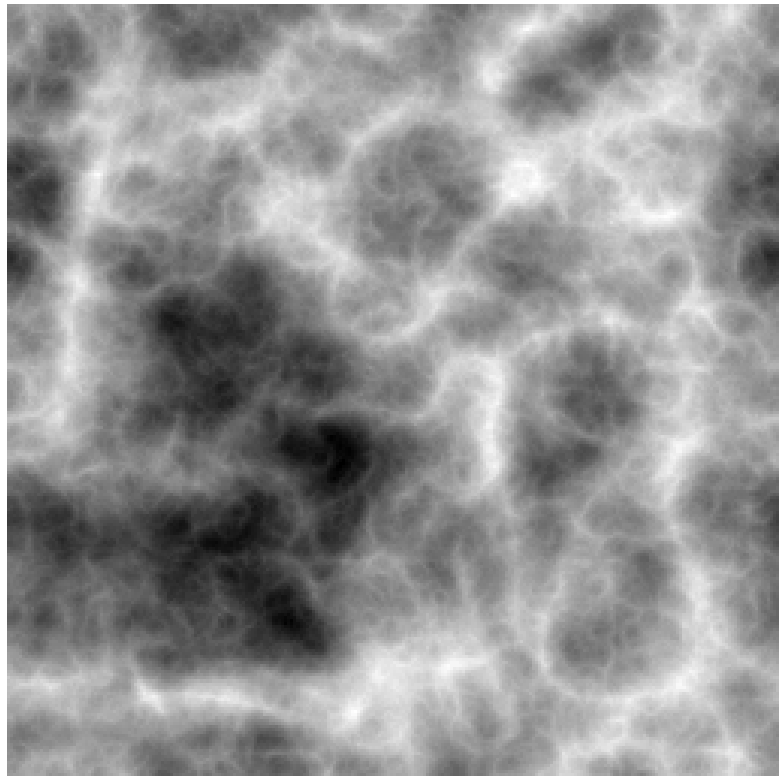
Terrain Noise

- Example of FBM output to the right
- Doesn't really create believable terrain / mountains
- Solution: Ridged multifractal
  - Very similar to Perlin noise, but the output of each octave is modified by an abs value function
- See Texturing & Modeling: A Procedural Approach (Musgrave et al, 1998)

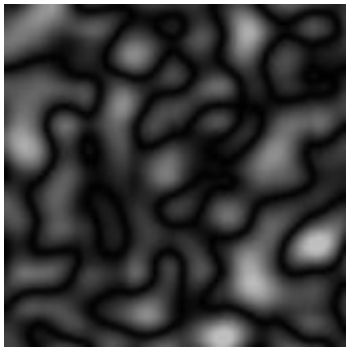
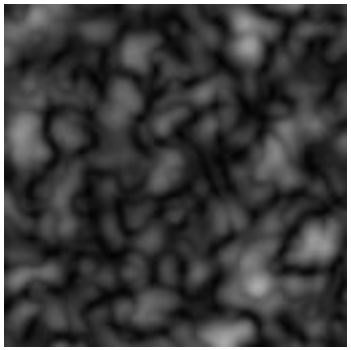
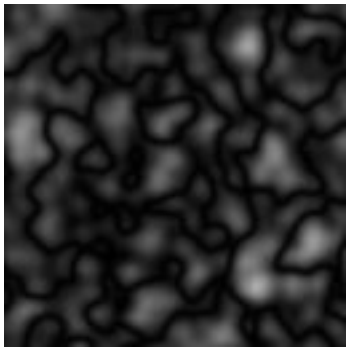


Terrain Noise

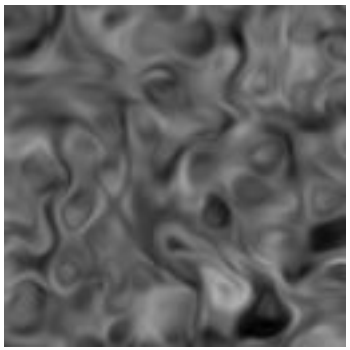
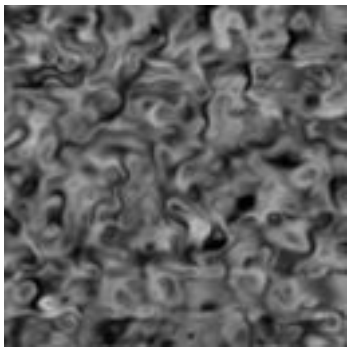
- Several other terrain noise generators (hybrid multifractals...)
- See Musgrave's work for more information - really the definitive work for this type of stuff (he designed the initial fractal based programs for Bryce)
- Interesting extensions: water / thermal erosion, river beds, domain distortion...
- Noise is complicated - but we're done talking about it for now



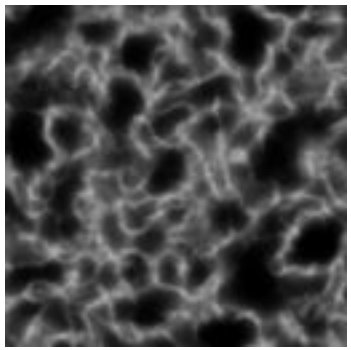
Noise Generation : Comparison

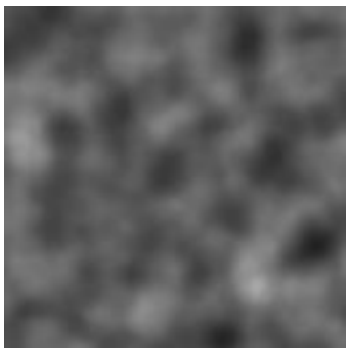
Perlin noise

FBM



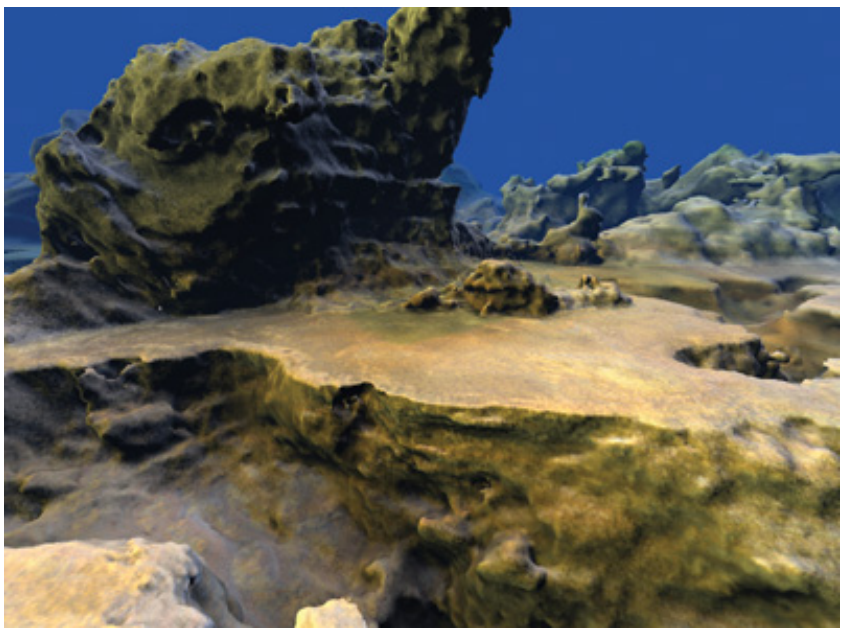
Ridged MF

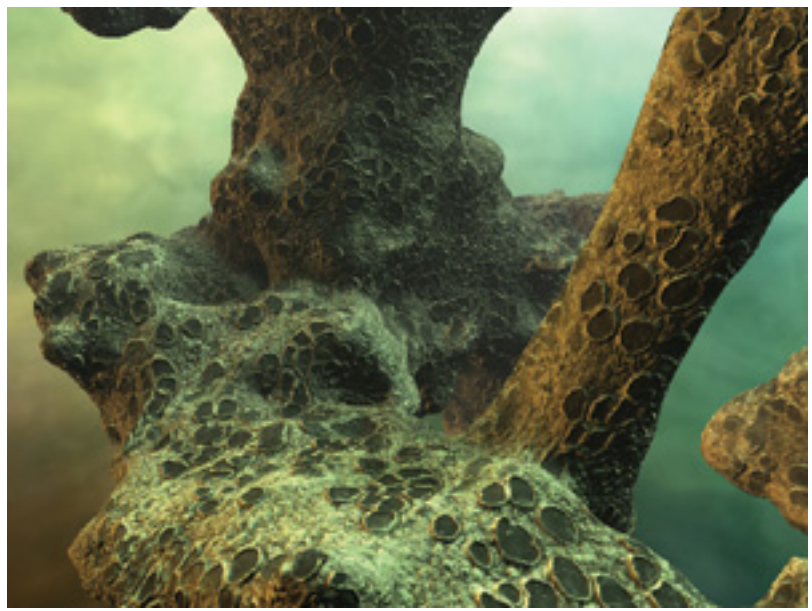


Hybrid MF

Terrain Noise

- What if we wanted caves, arches, and other 3D structures
- Its possible to extend these into 3D
  - Word of warning - meshing the noise becomes difficult
- With 2D noise, we can just use it as a displacement / heightmap
- If we have 3D noise, we need to use some kind of meshing algorithm (ex. marching cubes) to meshify the noise
- See GPU Gems Chapter 1





Water Noise

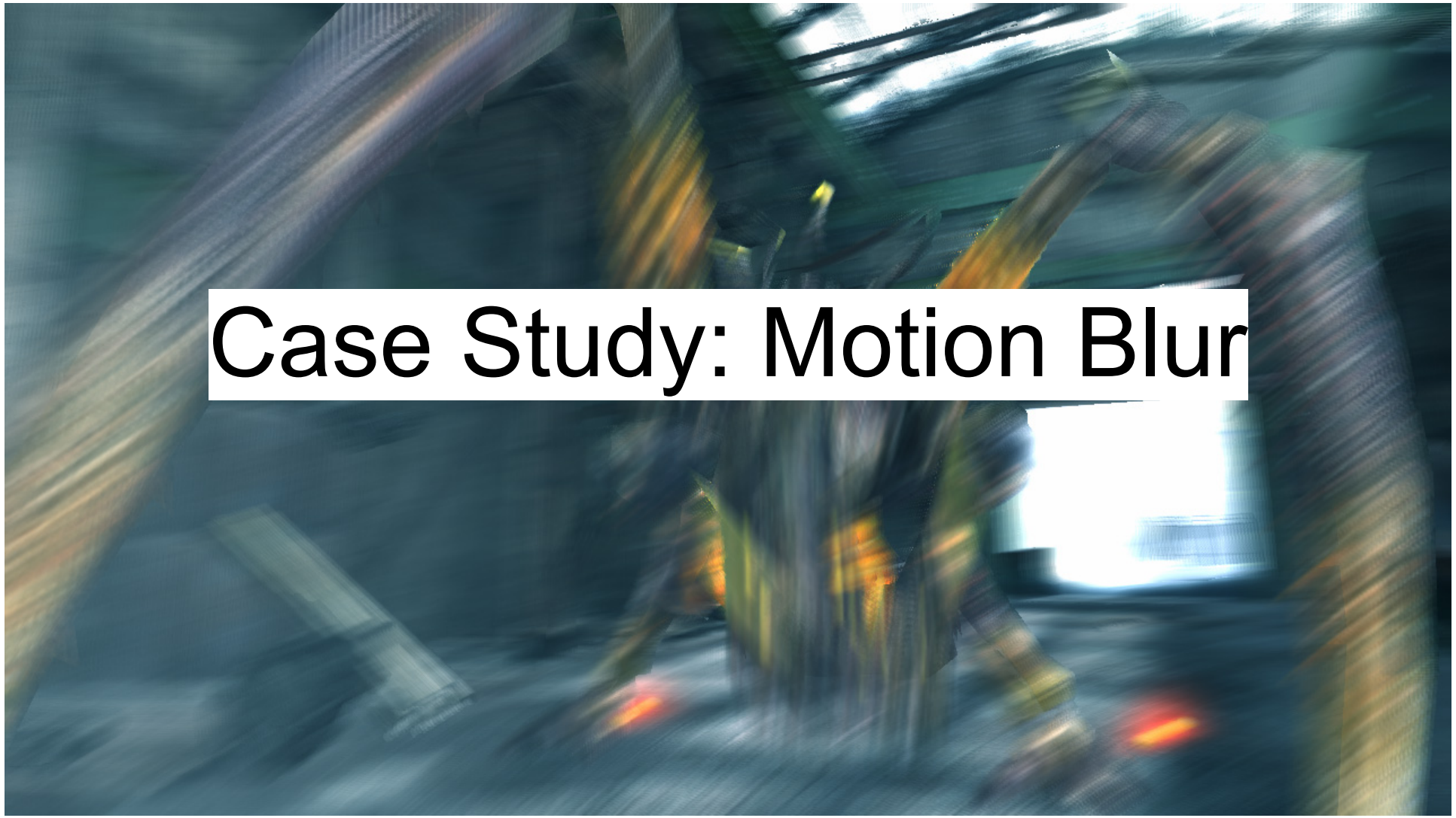
- Its possible to create a somewhat believable ocean using noise...
- See GPU Gems Chapter 18, http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter18.html

A Quick Note On Water Simulation

- Today, noise generation to simulate ocean water is kind of lame
- Modern games base ocean sims off of Tessendorf's work, Simulating Ocean Water (2001) - probably will talk about this

"Our water rendering techniques are quite similar to the solution we used in Crysis 1," explains Tiago Sousa [CryEngine 3]. "They are still based on Jerry Tessendorf's seminal work.

The biggest difference in the latest version, besides performance optimizations that allowed us to maintain most of its look on consoles, is that dynamic interaction is efficiently handled on all platforms."


Case Study: Motion Blur

Motion Blur

- In film and photography, happens because an object is in motion during the exposure, smearing photons across the film/CCD
- Similarly, we as human notice blurs in fast moving objects because photons are smeared across the retina
- How to emulate such blur in graphics?
  - The closest physical analogue would be an accumulation buffer, but we don't like those right
  - As always we will use H4X to simulate a motion blur effect cheaply
  - There are multiple ways to implement motion blur

2.5D Motion Blur

1. Render the scene into a texture to use later
2. Render again, using shaders to calculate velocity at each point (comparing previous and current vertex positions using transformations and interpolating between vertices)
3. Using this velocity, sample the texture multiple times to get the final pixel
    - You can count them all the same (box filter) or use a ramp or whatever

see http://origin-developer.nvidia.com/docs/IO/8230/GDC2003_OpenGLShaderTricks.pdf?q=docs/IO/8230/GDC2003_OpenGLShaderTricks.pdf for more details

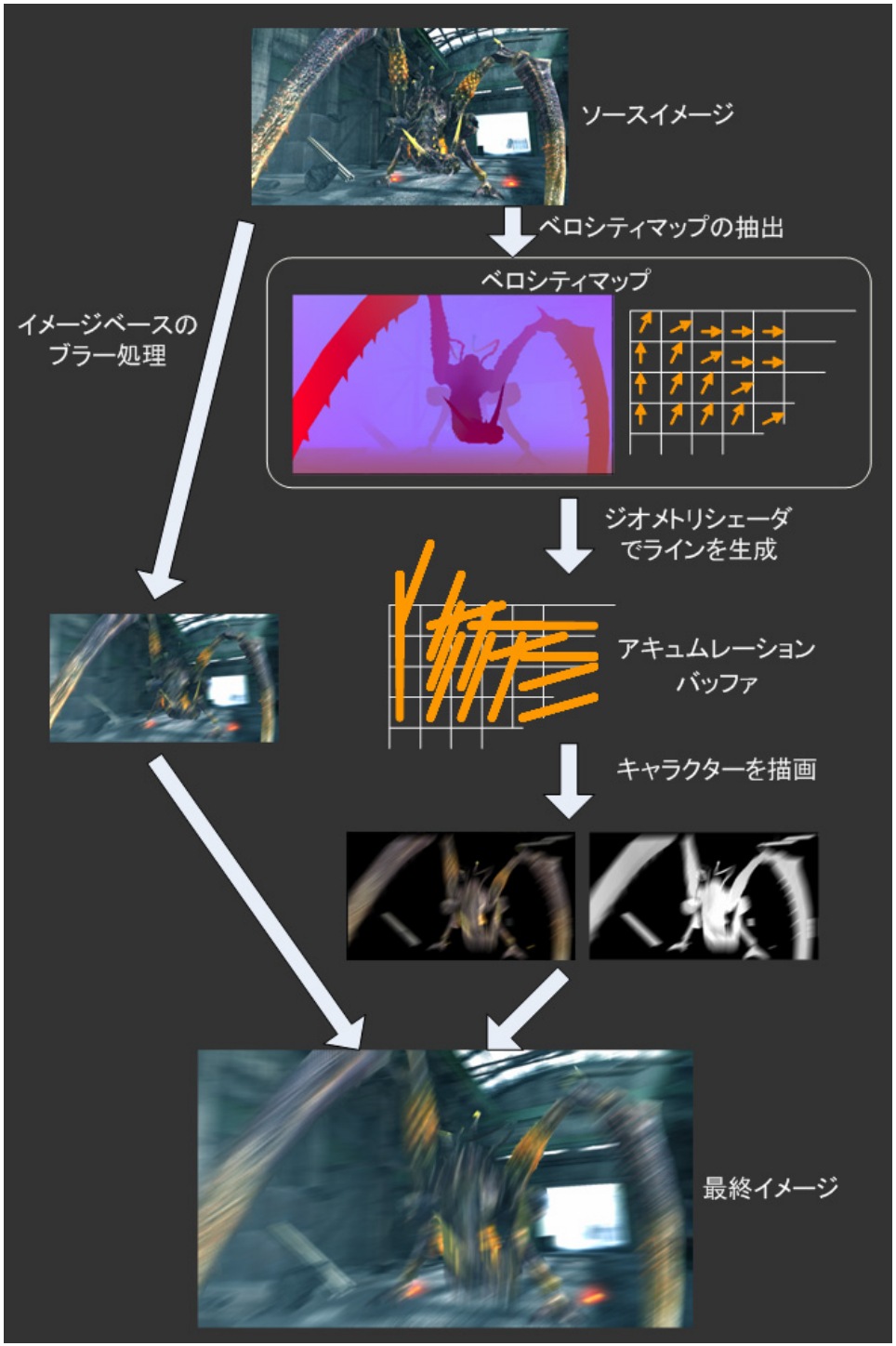
2.5D Motion Blur (cont.)

- But there's a small problem!
  - Shaders only run for rasterized geometry, which is only the current geometry and positions
  - This means bad artifacts when you have moving objects within a scene
  - Ideally, you would want some kind of trail following the object
- Hacks to the rescue!
  - Use the previous modelview/projection transformations to stretch the geometry
  - Take dot product of normal with velocity vector; if it's positive, use current transform, if negative, use previous transformation

A More Recent Technique : Capcom

- Using Geometry shaders!
- Capcom Lost Planet (2007)
- Extract a velocity map as usual
- Use geometry shaders to draw lines based on that velocity into a separate frame buffer
- Filter/blur and combine with original image
- Reduces artifacts compared to ordinary 2.5D motion blur

see http://www.beyond3d.com/content/news/499 for more



Capcom's Lost Planet

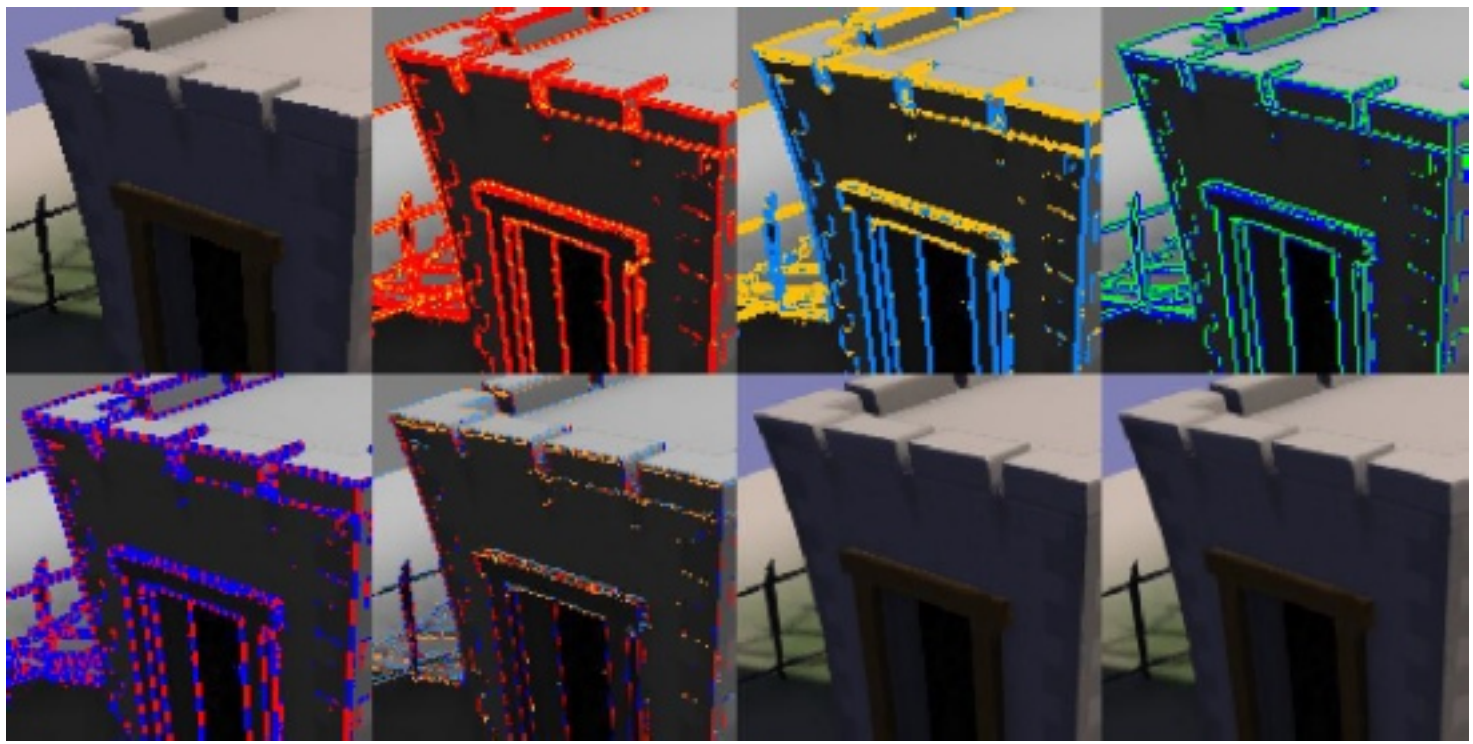
2.5D

Geometry Shader

A More Recent Technique : Valve

http://www.valvesoftware.com/publications/2008/GDC2008_PostProcessingInTheOrangeBox.pdf

Case Study: FXAA



FXAA

- Image Space Anti-Aliasing technique
- Alternative to traditional multisample techniques
  - Independent of scene complexity
  - Works for everything that is drawn, regardless of geometry, texture, transparency, post processing
  - Fast and easily integrated into existing projects (just a single fragment shader)

see http://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf for the original paper and http://timothylottes.blogspot.com for updates on algorithm and source code

Algorithm Overview

1. Convert to luminance
    - Mostly concerned with perception
2. Calculate local contrast to only run on edges
    - Saves compute time, avoids blurring in areas that don't need it
3. Find the direction of the edge (in terms of x and y)
4. Gather samples perpendicular to that direction to blur the edge
    - Recombine with some kind of weighted average/filter kernel

I didn't put a picture here because you probably wouldn't be able to tell the difference