Cascades

by NVIDIA

Ryan Geiss
Michael Thompson

Cascades

About the demo

- **Waterfalls flowing over procedural rock built on GPU**
- **Runs on Windows Vista, DirectX 10**
- **Heavily Utilizes:**
  - **Geometry Shaders**
  - **Stream Out**
  - **Render to 3D Texture**
  - **Pixel Shaders**
- **CPU virtually idle, even when generating new slices of rock.**

Demo

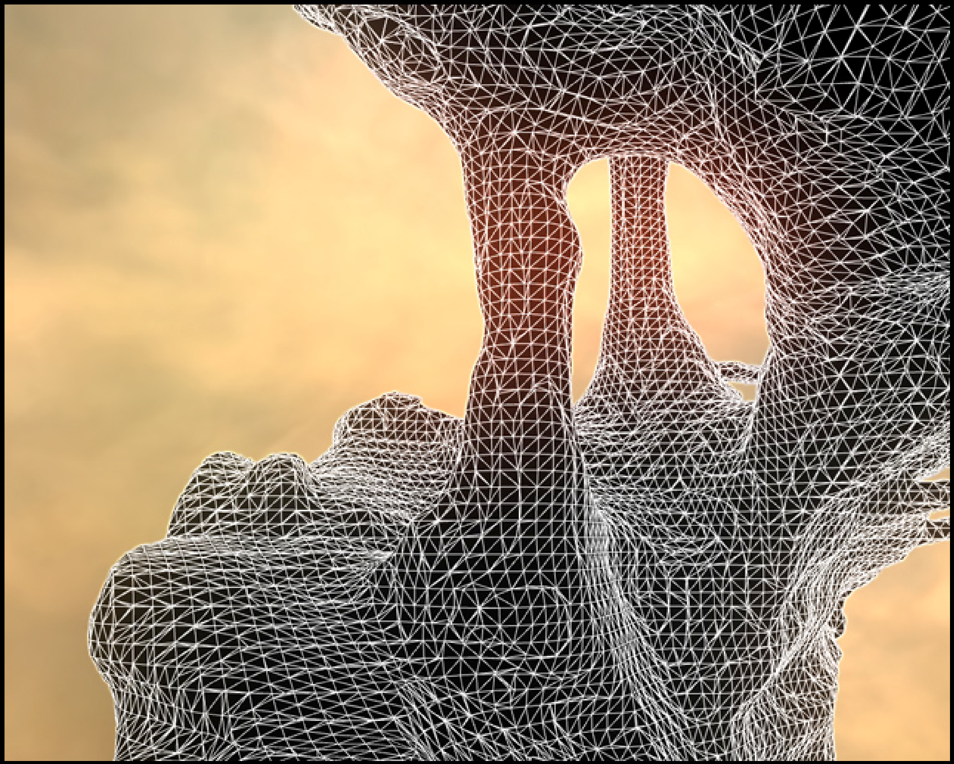
Cascades

What's the GPU doing here?

- *Building* complex procedural rock structures.
- Managing dynamic water particle system & physics (collisions with rock).
- Swarm of dragonflies buzzes around, avoiding the rock.
- Heavy-duty pixel shaders.

Main Topics to Cover

1. Rock Generation
2. Rock Rendering
3. Water (Particle System, Rendering)
4. Swarming Bugs



Rock Generation

Building the Rock: Overview

Step 1: Render to slices of a 3D texture

- Render a “density” value into each voxel.
- (+) values will become rock, (–) values, air.

Step 2: Precompute some lighting info.

- Compute normals
- Cast occlusion rays

Step 3: Generate & store polygons

- Use ‘Marching Cubes’ algorithm on each cell.

(…all on the GPU.)

Building the Rock

Step 1: Render to 3D (volume) texture

- **3D Texture:**

Format:	**DXGI_FORMAT_R16_FLOAT** (one 16-bit float)
Size:	**96 x 96 x 256**
Memory:	**< 5 MB**
Contents:	**Density values (positive ~ rock, negative ~ air)**

- **To generate slices of rock, we render "fullscreen quads" to 2D slices of the 3D texture.**
- **Heavy pixel shader math to figure out the density value at each pixel (voxel). (160 instructions)**

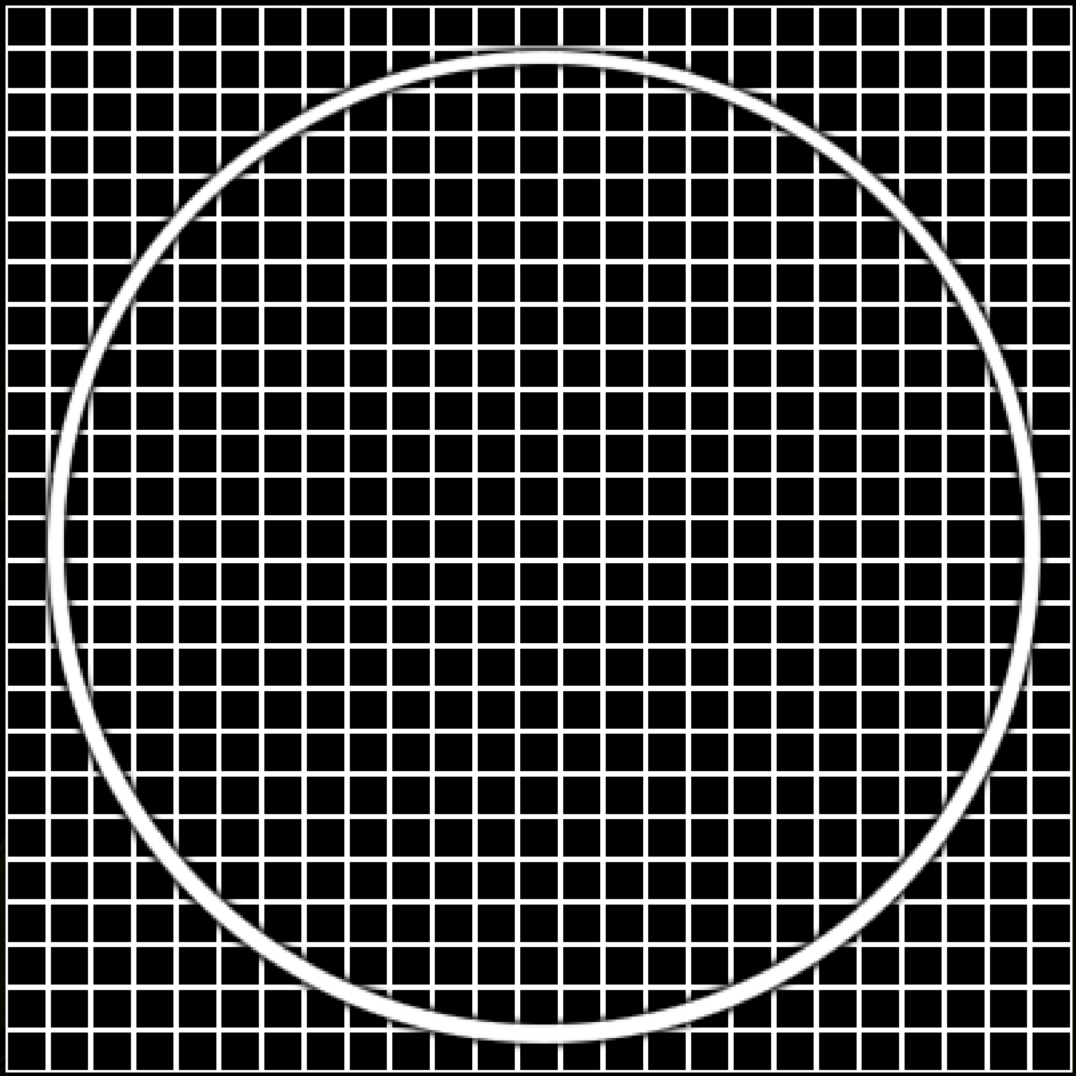
Building the Rock

Add several base shapes together:

1. Three roaming vertical pillars (cylinders) (+)
2. One negative pillar, to create open space in the center (–)
3. Shelves – a function of the Y coordinate only; periodically creates a shelf of rock. (+)
4. Helix – biases half of the space toward rock, half toward air. (+/–)
5. Noise – four octaves of random noise sampled from small 3D textures (+/–)

Looking at a Y-slice of rock:

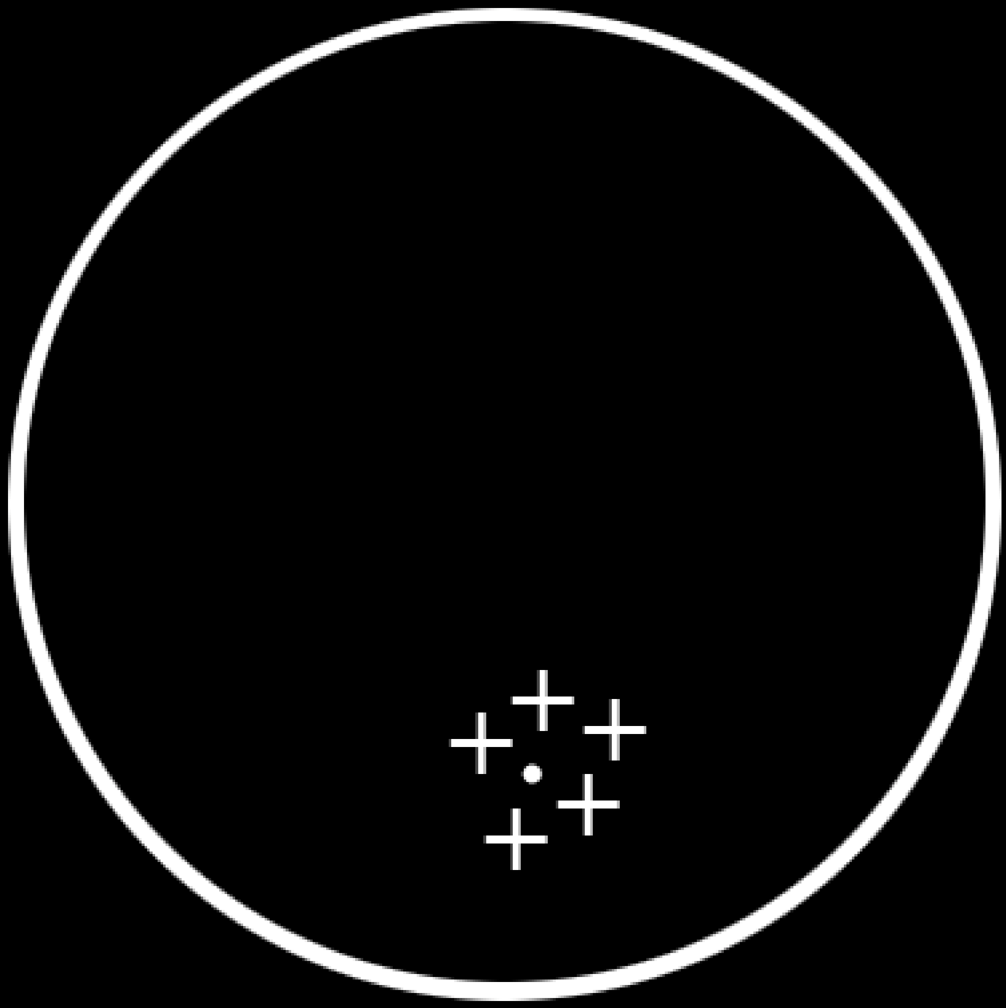
(…value starts at zero everywhere.)



```
float f = 0;
```

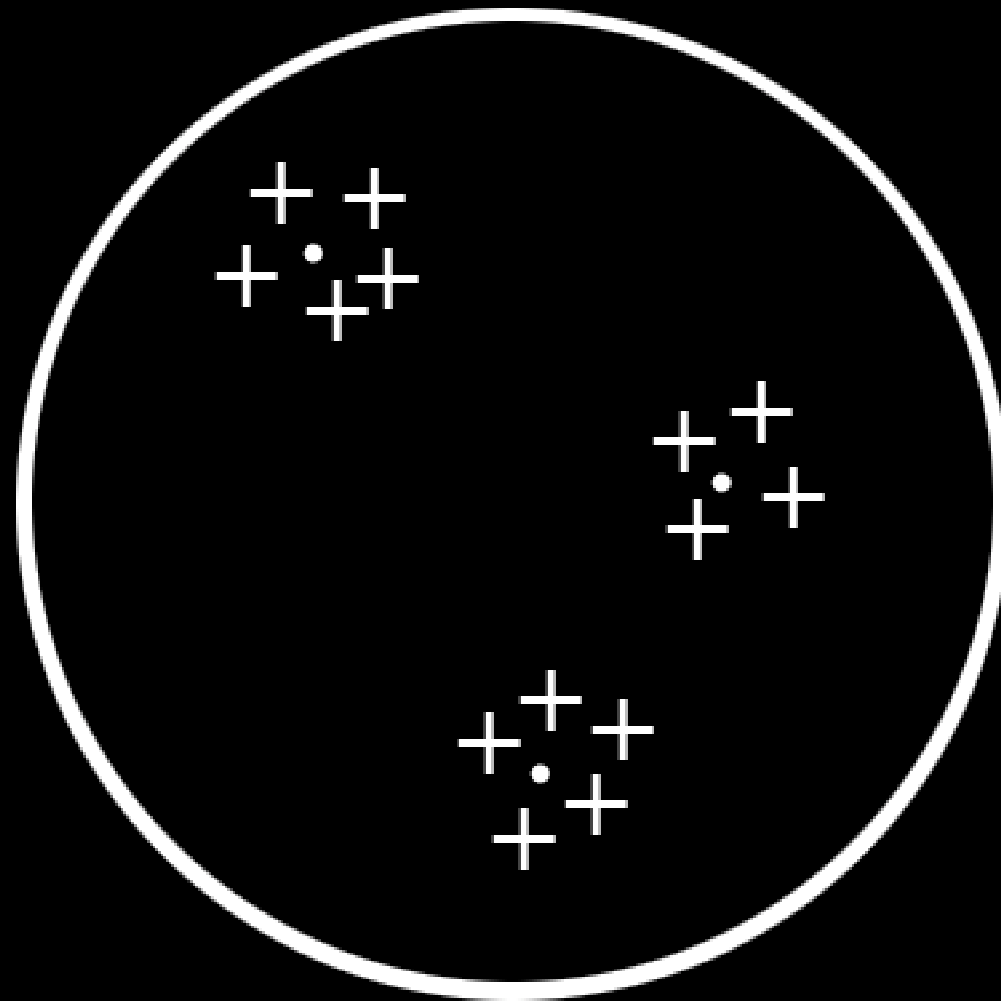
Add a pillar:

(…pillar center roams in XZ plane from slice to slice; stored in a constant buffer.)



```
f += 1 / length(ws.xz – pillar.xy) - 1;
```

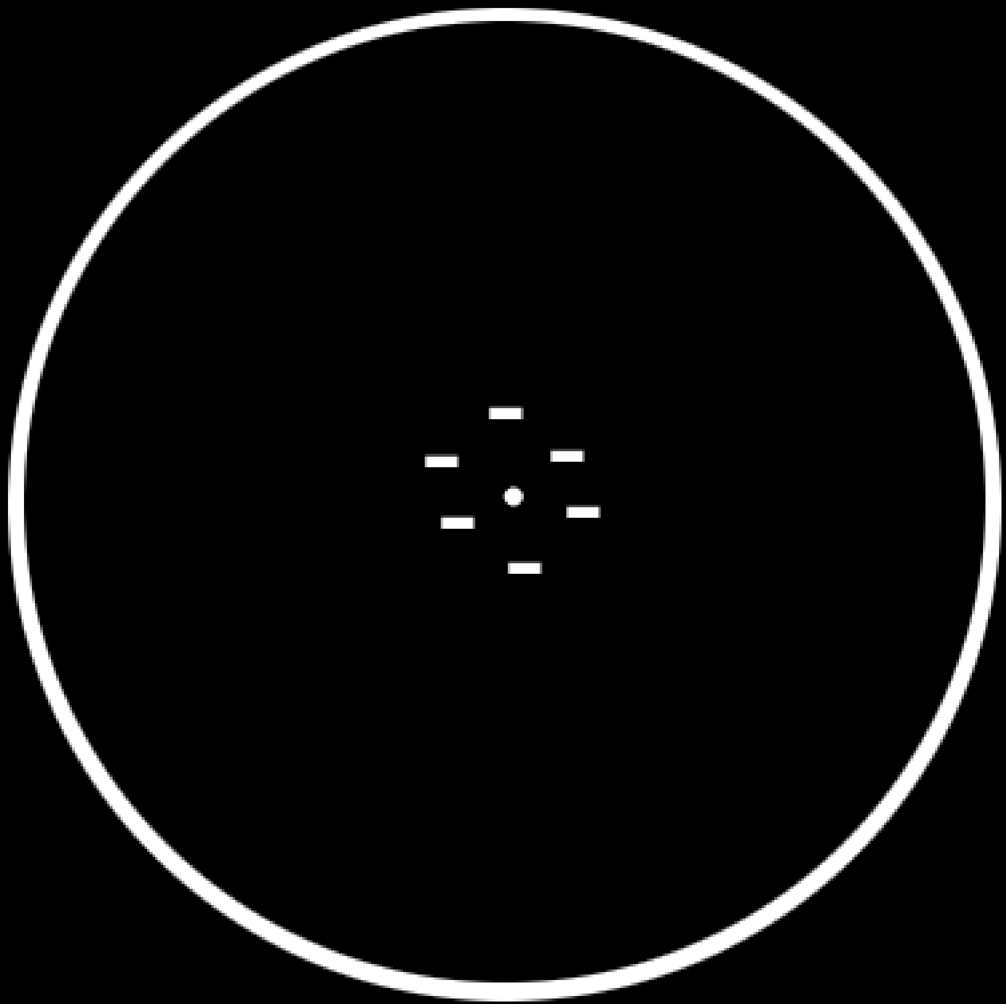
3-pillar version:



```
for k = 0,1,2
  f += 1 / length(ws.xz –  pillar[k].xy) - 1;
```

Add negative values going down the center:

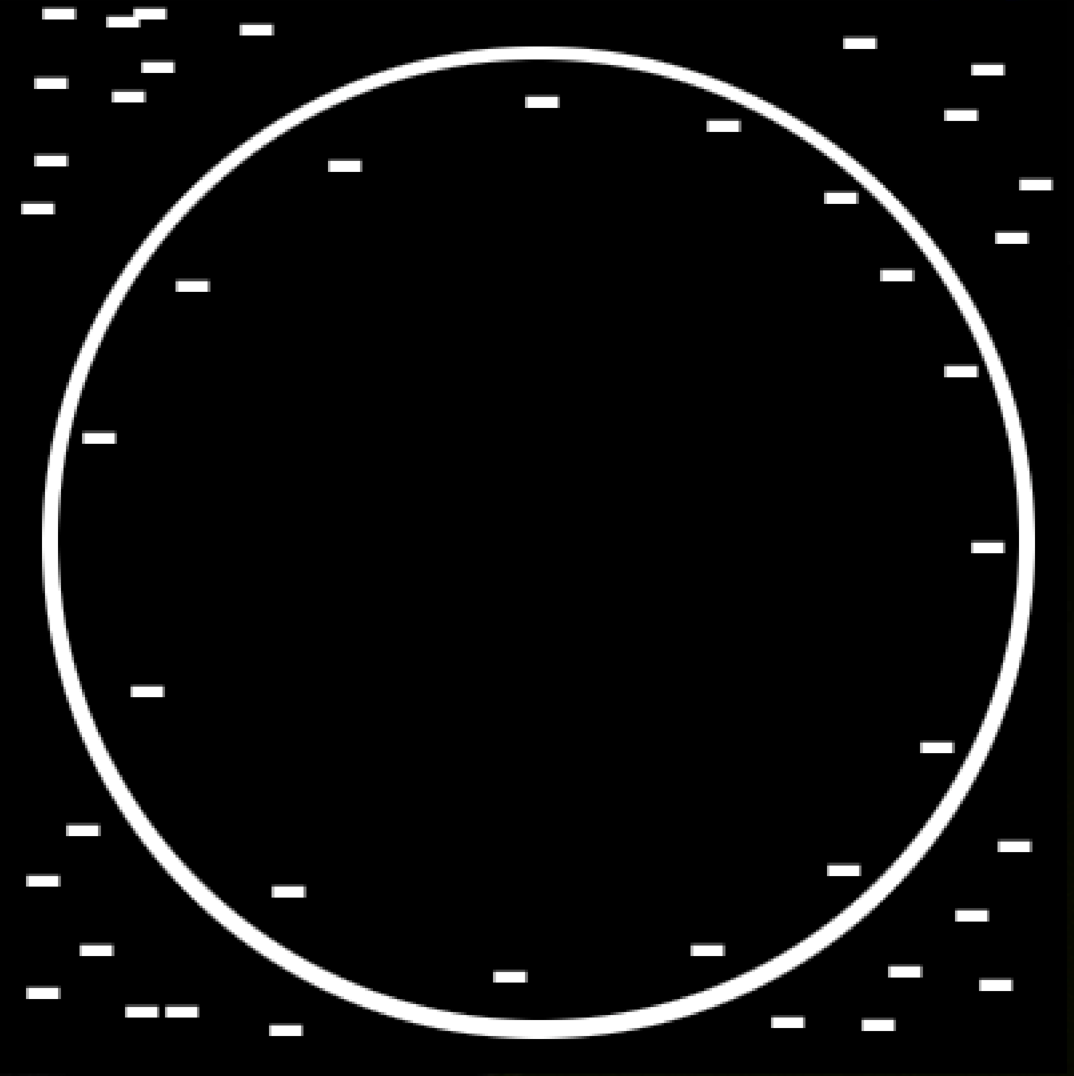
(water flow channel)



```
f -= 1 / length(ws.xz) - 1;
```

Add strong negative values at outer edge.

Keeps solid rock "in bounds".

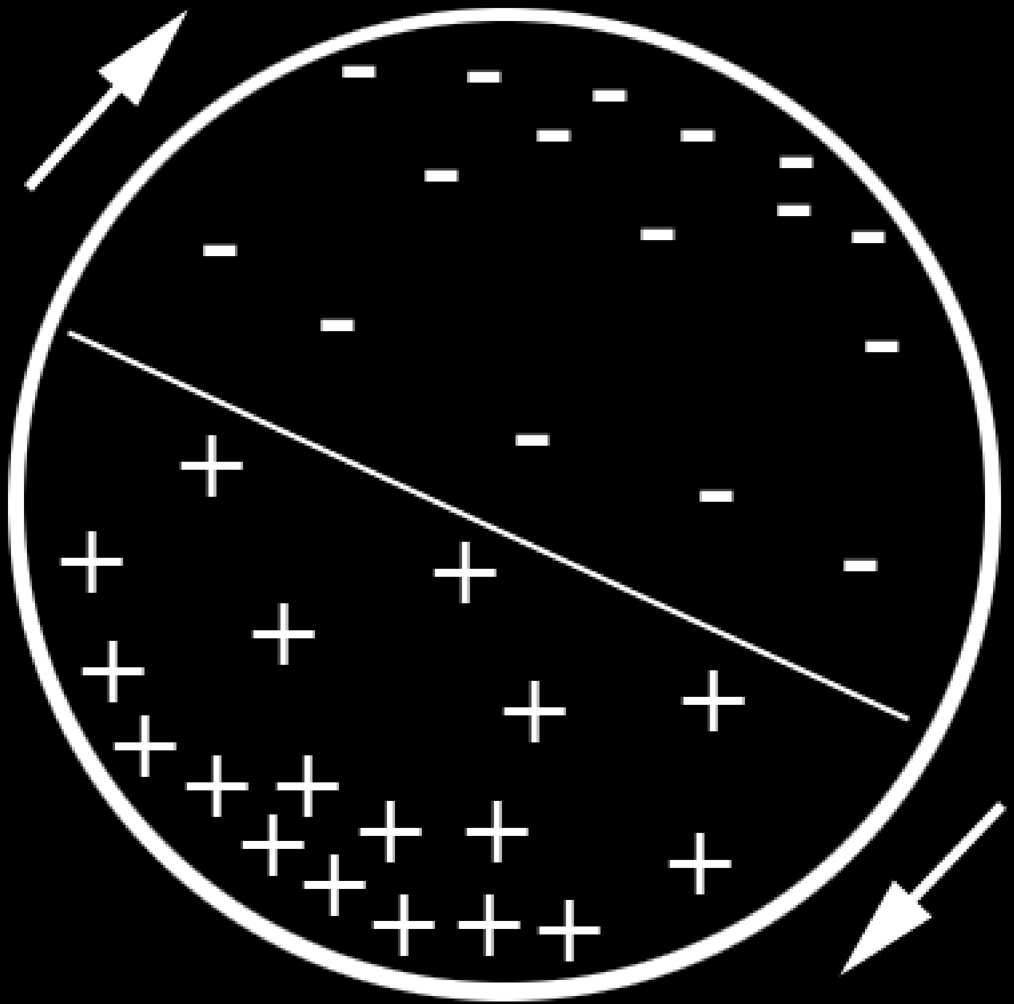


```
f = f - pow( length(ws.xz), 3 );
```

Helix:

Add + and – values on opposite sides.

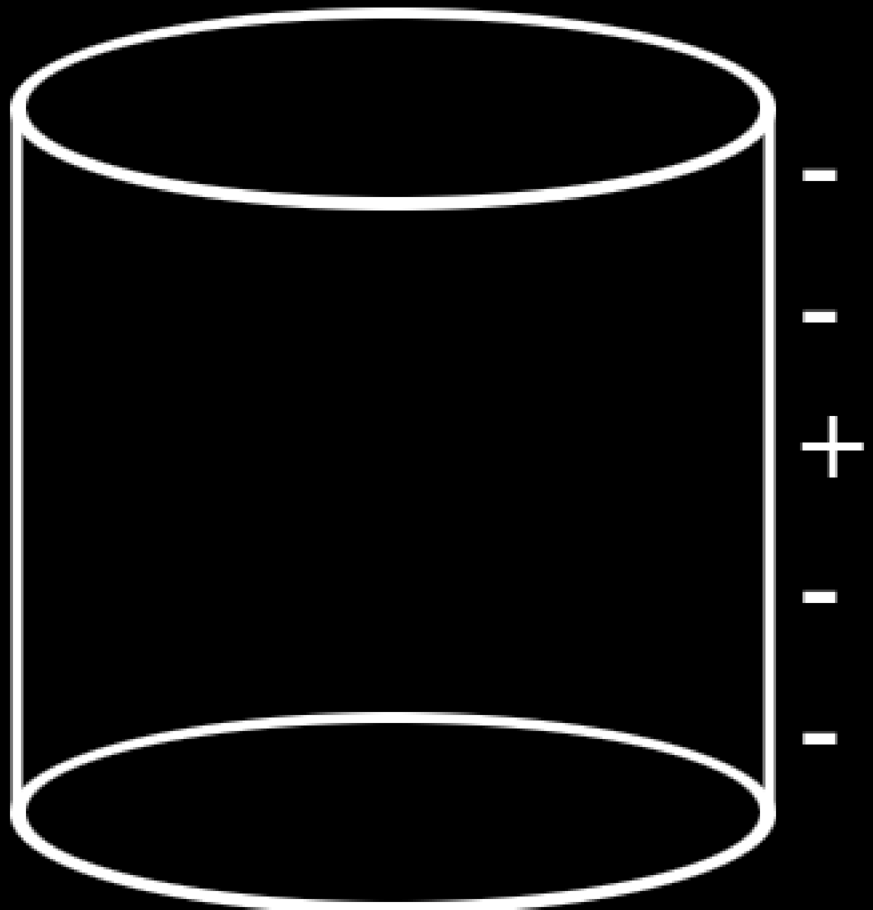
Rotate the values as the slice's Y coord changes.



```
float2 vec = float2(cos(ws.y), sin(ws.y));
f += dot(vec, ws.xz);
```

Shelves:

Periodically add positive values based on slice's Y coord.




```
f += cos( ws.y );
```

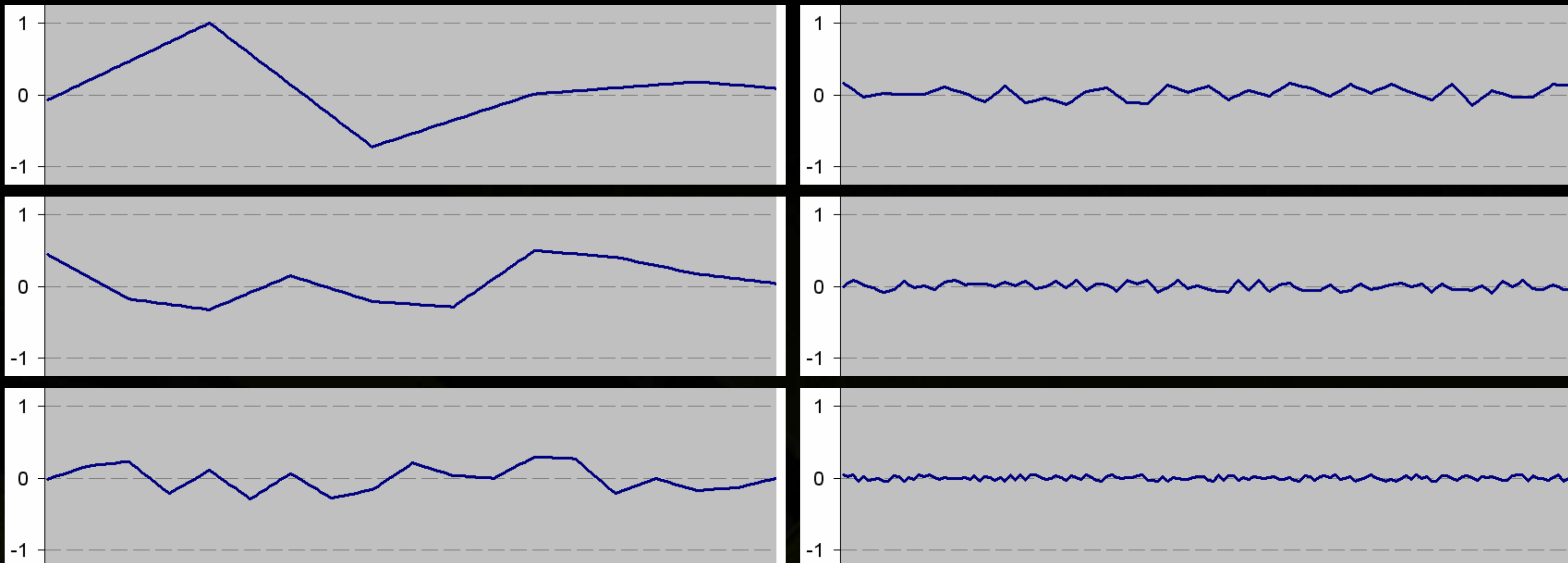
Building the Rock

- Next, add Noise for a more natural look.

- In 1D case, create noise by adding several octaves of random signals.

- Signal at each octave has:
  - half the amplitude
  - ~twice the frequency

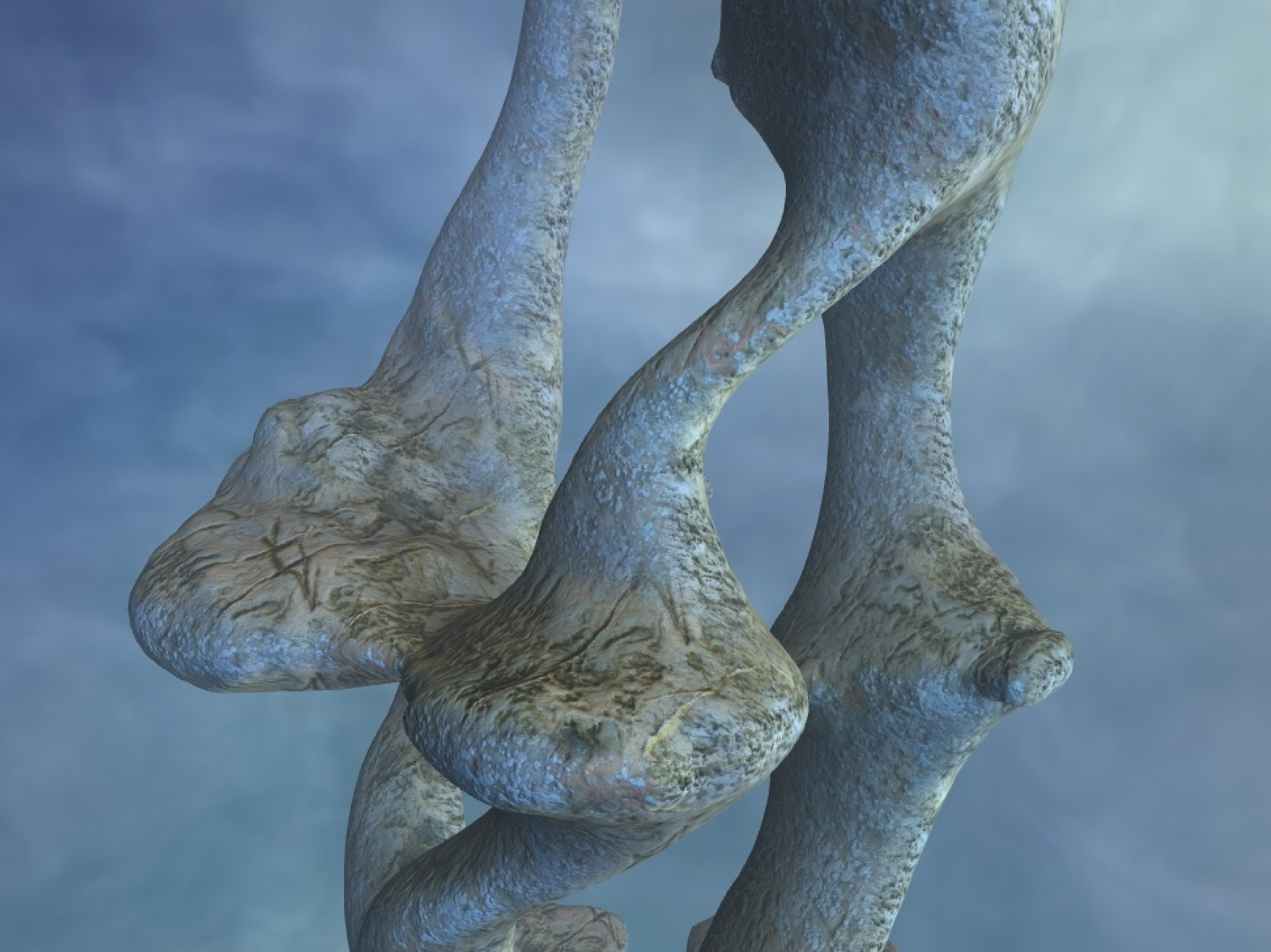
  of the previous octave.

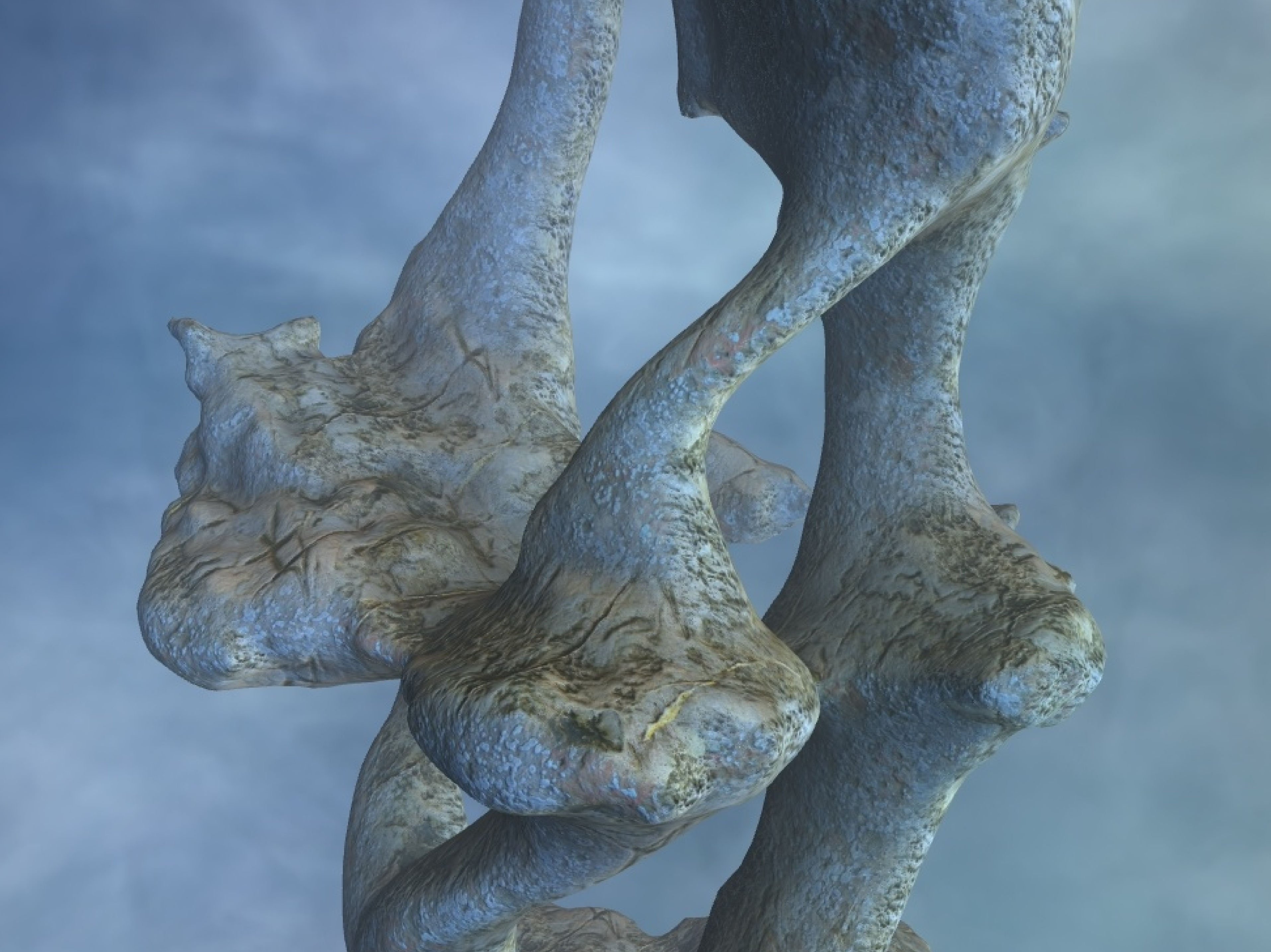


Add all of the above & you get… a mountain:

Building the Rock

- 1D noise ~ mountain silhouette.
- 2D noise ~ terrain height map.
- 3D noise ~ a bunch of +/– values in 3D space.
- When added to the simpler basis functions (cylinder, helix, etc) they add nice fractal detail to our rock’s shape.





Building the Rock

- Noise on the GPU:
  - Each octave is a 3D texture of random floats.
  - Size: 16 x 16 x 16
  - Range: [-1..1]
- Sample 4 octaves & sum the results.

- To avoid visual repetition:
  - Avoid lacunarity of exactly 2.0.
  - Randomly rotate input to each octave.
    - (each octave has own 3x3 rotation matrix)
  - Translation not necessary.

Building the Rock

- Advantages of noise-based geometry:
  - Yields visually rich & non-repeating “terrain”
  - Every little bit of geometry preserved.
  - Save your favorites.
  - Preset files (scenes) use only 3 kilobytes.

Normals & Occlusion

Step 2: Precompute lighting information.

- Render to slices of a second 3D texture.
  - This one will store lighting info.
- Use first 3D Texture (densities) to compute normal vector and ambient occlusion factor.
- Store both in a rgba8 volume texture
  - .xyz ← normal (packed)
  - .w ← occlusion

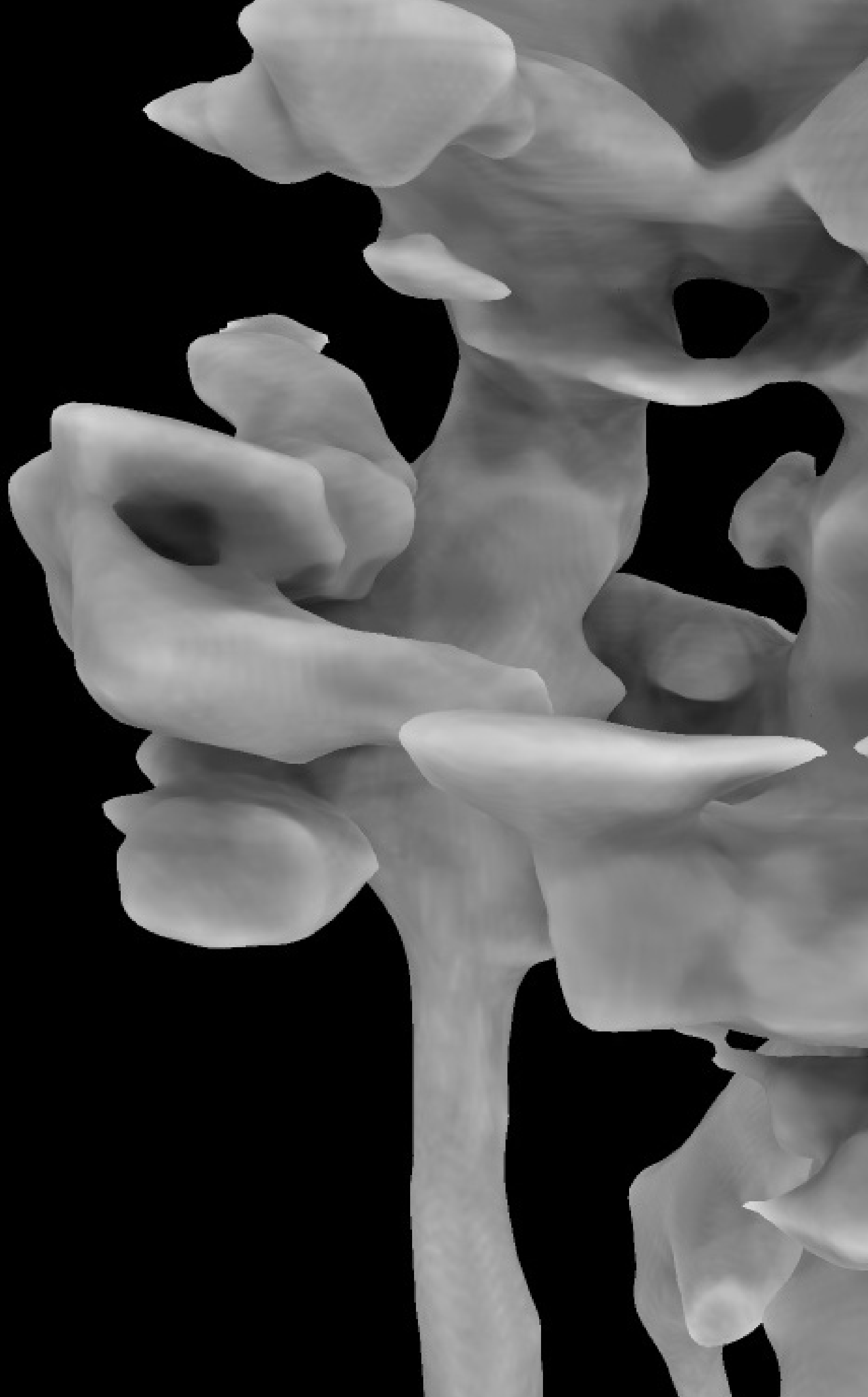
Normals & Occlusion

Normal vector is simply the gradient of the density values.

```
float3 ComputeNormal( Texture3D tex, SamplerState s,
                      float3 uvw) {
  float4 step = float4(inv_voxelDim, 0);
  float3 gradient = float3(
      tex.SampleLevel(s, uvw + step.xww, 0)
    - tex.SampleLevel(s, uvw - step.xww, 0),
      tex.SampleLevel(s, uvw + step.wwy, 0)
    - tex.SampleLevel(s, uvw - step.wwy, 0),
      tex.SampleLevel(s, uvw + step.wzw, 0)
    - tex.SampleLevel(s, uvw - step.wzw, 0)
  );
  return normalize(-gradient);
}
```

Normals & Occlusion

- **<u>Ambient occlusion</u>** factor tells us, at any point, what % of random rays cast out would hit the rock (vs. escaping into the environment).
- Used to shade the rock, so recesses appear darker.



Normals & Occlusion

- Occlusion factor generated by casting 32 rays, testing for collisions with the rock.
  - Sample the densities at each point along ray; ‘collision’ when a positive density is found.
  - The 32 rays are in a 3D poisson distribution.
  - Take 16 samples per ray.
  - Distance-wise, march through 20% of the width.

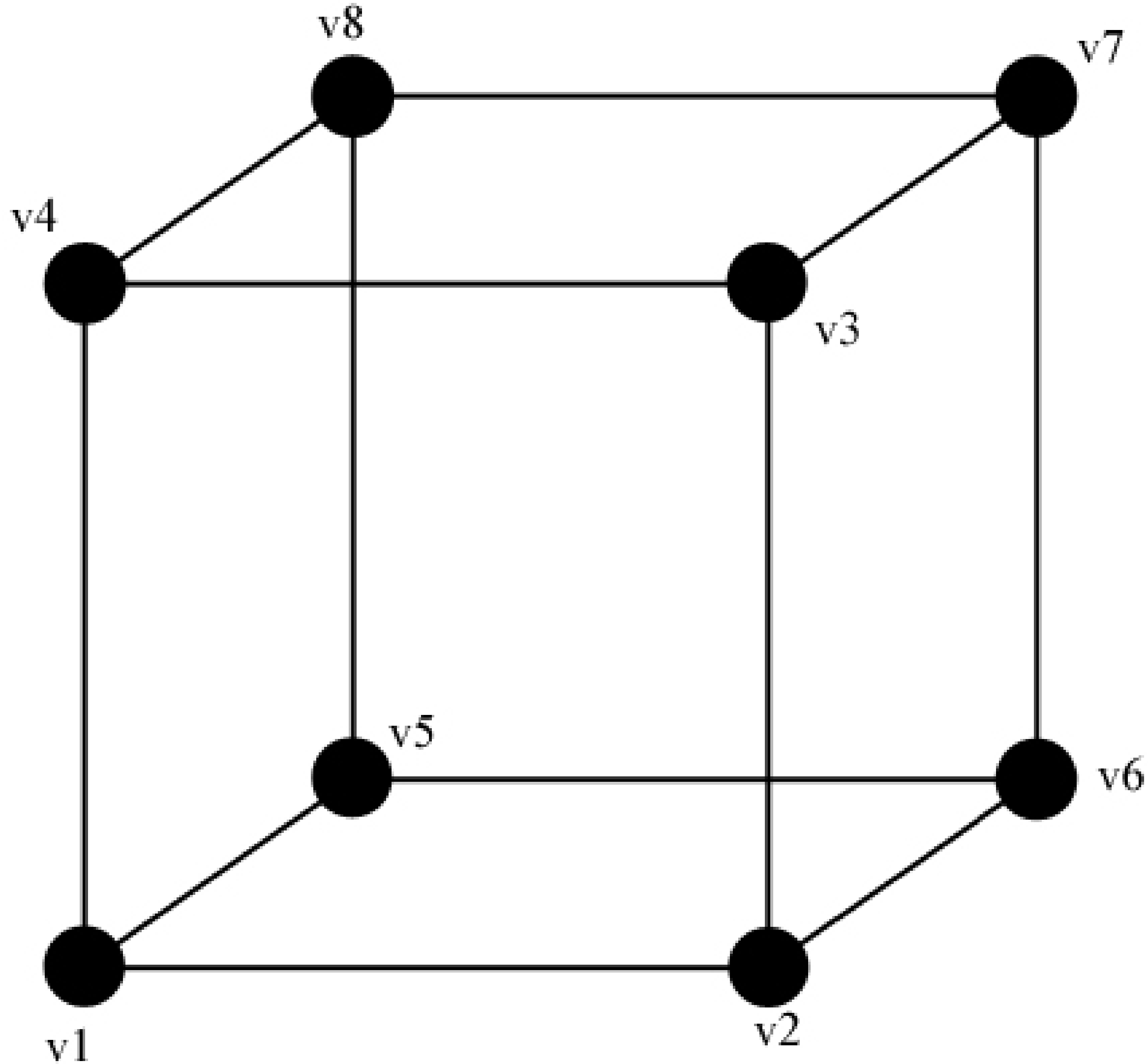
Normals & Occlusion

- Why do we need lighting data *everywhere?*
  - Why not just per vertex?
- Knowing occlusion data lets us light anything in the rock volume.
  - Dragonflies
  - Water
  - Vines (easter egg – see args.txt)
- Normals speed up water flow & vine-crawl calculations.

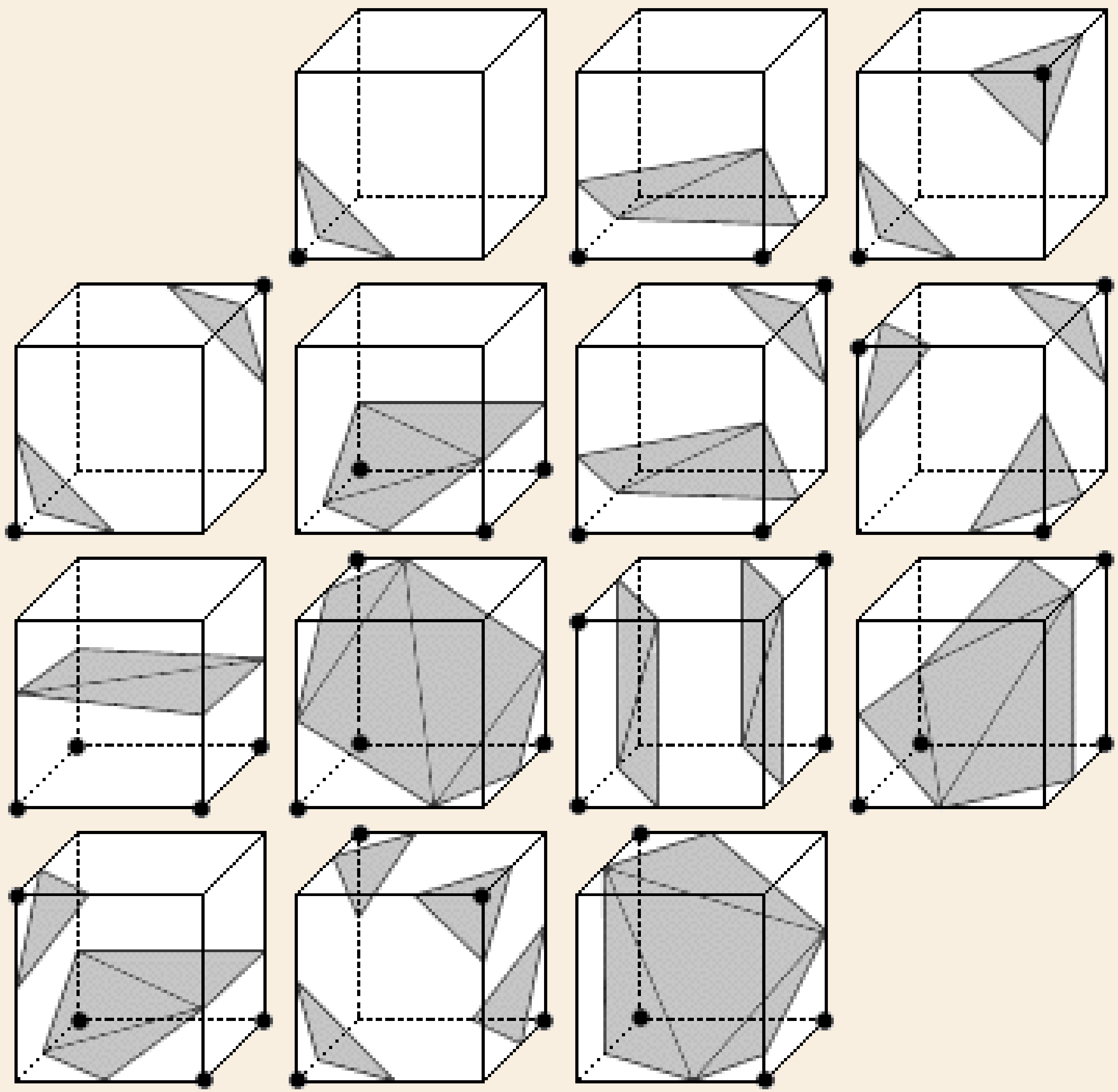
Generating Polygons

Step 3: Generate polygons via Marching Cubes

- Constructs a polygonal surface where densities equal zero.
- Works on one voxel (cell) at a time.
- INPUT: the density at each of the 8 corners
  - 8 corners “in”/”out” → 256 possible cases (2^8)
- OUTPUT: 0 to 5 polygons
- (Note: patent expired; free to use)



Case = v8|v7|v6|v5|v4|v3|v2|v1



Generating Polygons

To generate a slice of rock:

- “Draw” dummy vertex buffer of 96x96 points
  - Points have uv coords in [0..1] range.
- Pipeline: VS → GS → VB
  - Pixel shader disabled
- Vertex Shader:
  - Samples densities at 8 corners
  - Determines the MC case
  - Passes this data on to GS

- Geometry Shader uses:
  - Dynamic Indexing
  - Lookup Tables (constant buffers)
  - Dynamic Branching
  - Stream Output (variable # of primitives)

…to generate polygons.

- Output primitive type: Triangle List
- Appends 0, 3, 6, 9, 12, or 15 vertices to a VB.

Generating Polygons

More on the Geometry Shader (GS):

- Heavy use of lookup tables
- One translates case → # of polygons to output
- One tells you which cell edges to place the 3 verts on, for each triangle emitted.
- Resulting vertices (a triangle list) are streamed out to a vertex buffer (VB).
- We used one VB for every 12 slices (voxel layers) of the rock.*
- VB's are created at startup and never resized.
  - Memory footprint: we needed about 22 bytes of video memory for each voxel in the VB.

Generating Polygons

Notes on coordinate spaces:

- In <u>world space</u>…
  - …+Y is up, although in the 3D texture, that's +Z.
    - *(you can render to Z slices of a volume; but not to X or Y slices)*
  - …the rock spans [-1..1] in X and Z and can roam on Y.
  - …the rock bounding box size is 2.0 in X and Z, and 5.333 in Y.   (2.0 * 256/96)
- In <u>UVW (3D texture) space</u>…
  - Coordinates range from [0..1] in x, y, z.
  - "slices" are along Z (not Y!).
- Texels in the 3D texture correspond to cell *corners*. If a texture slice is 96x96, then there are *95x95* cells (or voxels).

Generating Polygons

Some handy global constants:

```
float  WorldSpaceVolumeHeight = 2.0*(256/96.0);
float3 voxelDim               = float3(96, 256, 96);
float3 voxelDimMinusOne = float3(95, 256, 95);
float3 wsVoxelSize            = 2.0/95.0;
float4 inv_voxelDim           = float4( 1.0/voxelDim, 0 );
float4 inv_voxelDimMinusOne
  = float4( 1.0/voxelDimMinusOne, 0 );
```

Generating Polygons

Most of this is easily borrowed from the demo…

1. Generate your own density values
2. Copy our 3 shaders for getting normals / occlusion.
3. Copy our 2 shaders for rock generation.
4. Also grab contents of a few constant buffers – see `models\sceneBS.nma` (or see notes this slide).

Marching Cubes Vertex Shader [1]

```
// This vertex shader is fed 95*95 points, one for each *cell* we'll run M.C. on.
// To generate > 1 slice in a single frame, we call DrawInstanced(N),
//   and it repeats it N times, each time setting nInstanceID to [0 .. N-1].

// per-vertex input attributes: [never change]
struct vertexInput {
  float2 uv          : POSITION;      // 0..1 range
  uint   nInstanceID : SV_InstanceID;
};

struct vsOutputGsInput {         // per-vertex outputs:
  float3 wsCoord     : POSITION; // coords for LOWER-LEFT corner of the cell
  float3 uvw         : TEX;
  float4 f0123       : TEX1;     // the density values
  float4 f4567       : TEX2;     //   at the 8 cell corners
  uint   mc_case     : TEX3;     // 0-255
};

Texture3D    tex; // our volume of density values. (+=rock, -=air)
SamplerState s;   // trilinear interpolation; clamps on XY, wraps on Z.

cbuffer SliceInfos {
  // Updated each frame.  To generate 5 slices this frame,
  // app has to put their world-space Y coords in slots [0..4] here.
  float slice_world_space_Y_coord[256];
}

// converts a point in world space to 3D texture space (for sampling the 3D texture):
#define WS_to_UVW(ws) (float3(ws.xz*0.5+0.5, ws.y*WorldSpaceVolumeHeight).xzy)
```

Marching Cubes Vertex Shader [2]

```
v2gConnector main(vertexInput vtx)
{
  // get world-space coordinates & UVW coords of lower-left corner of this cell
  float3 wsCoord;
  wsCoord.xz = vtx.uv.xy*2-1;
  wsCoord.y  = slice_world_space_Y_coord[ vtx.nInstanceID ];
  float3 uvw = WS_to_UVW( wsCoord );

  // sample the 3D texture to get the density values at the 8 corners
  float2 step = float2(worldSpaceVoxelSize, 0);
  float4 f0123 = float4(  tex.SampleLevel(s, uvw + step.yyy, 0).x,
                          tex.SampleLevel(s, uvw + step.yyx, 0).x,
                          tex.SampleLevel(s, uvw + step.xyx, 0).x,
                          tex.SampleLevel(s, uvw + step.xyy, 0).x  );
  float4 f4567 = float4(  tex.SampleLevel(s, uvw + step.yxy, 0).x,
                          tex.SampleLevel(s, uvw + step.yxx, 0).x,
                          tex.SampleLevel(s, uvw + step.xxx, 0).x,
                          tex.SampleLevel(s, uvw + step.xxy, 0).x  );

  // determine which of the 256 marching cubes cases we have for this cell:
  uint4 n0123 = (uint4)saturate(f0123*99999);
  uint4 n4567 = (uint4)saturate(f4567*99999);
  uint mc_case = (n0123.x     ) | (n0123.y << 1) | (n0123.z << 2) | (n0123.w << 3)
             | (n4567.x << 4) | (n4567.y << 5) | (n4567.z << 6) | (n4567.w << 7);

  ...
  // fill out return struct using these values, then on to the Geometry Shader.
}
```

Marching Cubes Vertex Shader

```
// sample the iso-value at the 8 corners
float2 step = float2(worldSpaceVoxelSize, 0);
float4 f0123 = float4( tex.SampleLevel(s, uvw + step.yyy, 0).x,
                       tex.SampleLevel(s, uvw + step.yyx, 0).x,
                       tex.SampleLevel(s, uvw + step.xyx, 0).x,
                       tex.SampleLevel(s, uvw + step.xyy, 0).x);
float4 f4567 = float4( tex.SampleLevel(s, uvw + step.yxy, 0).x,
                       tex.SampleLevel(s, uvw + step.yxx, 0).x,
                       tex.SampleLevel(s, uvw + step.xxx, 0).x,
                       tex.SampleLevel(s, uvw + step.xxy, 0).x);

// determine which of the 256 marching cubes cases for this cell:
uint4 n0123 = (uint4)saturate( f0123 * 99999 );
uint4 n4567 = (uint4)saturate( f4567 * 99999 );
uint  mc_case =  (n0123.x     ) | (n4567.x << 4)
               | (n0123.y << 1) | (n4567.y << 5)
               | (n0123.z << 2) | (n4567.z << 6)
               | (n0123.w << 3) | (n4567.w << 7);
```

Marching Cubes Geom. Shader

```
// GEOMETRY SHADER INPUTS:

struct vsOutputGsInput {
  float4 wsCoord : POSITION;
  float3 uvw     : TEX;
  float4 f0123   : TEX1;  // the density values
  float4 f4567   : TEX2;  //   at the corners
  uint   mc_case : TEX3;  // 0-255
};

struct GSOutput {
  // Stream out to a VB & save for reuse!
  // .xyz = wsCoord, .w = occlusion
  float4 wsCoord_Ambo : POSITION;
  float3 wsNormal     : NORMAL;
};

// our volume of density values.
Texture3D    tex;

// .xyz = low-quality normal; .w = occlusion
Texture3D    grad_ambo_tex;

// trilinear interp; clamps on XY, wraps on Z.
SamplerState s;
```

```
cbuffer g_mc_lut1 {
  uint
    case_to_numpolys[256];
  float4 cornerAmask0123[12];
  float4 cornerAmask4567[12];
  float4 cornerBmask0123[12];
  float4 cornerBmask4567[12];
  float3 vec_start[12];
  float3 vec_dir  [12];
};

cbuffer g_mc_lut2 {
  int4 g_triTable[1280];
    //5*256
};
```

Marching Cubes Geom. Shader

```
[maxvertexcount (15)]
void main( inout TriangleStream<GSOutput> Stream,
           point vsOutputGsInput input[1] )
{
  GSOutput output;
  uint num_polys = case_to_numpolys[ input[0].mc_case ];
  uint table_pos = mc_case*5;
  for (uint p=0; p<num_polys; p++) {
    int4 polydata = g_triTable[ table_pos++ ];
    output = PlaceVertOnEdge( input[0], polydata.x );
    Stream.Append(output);
    output = PlaceVertOnEdge( input[0], polydata.y );
    Stream.Append(output);
    output = PlaceVertOnEdge( input[0], polydata.z );
    Stream.Append(output);
    Stream.RestartStrip();
  }
}
```

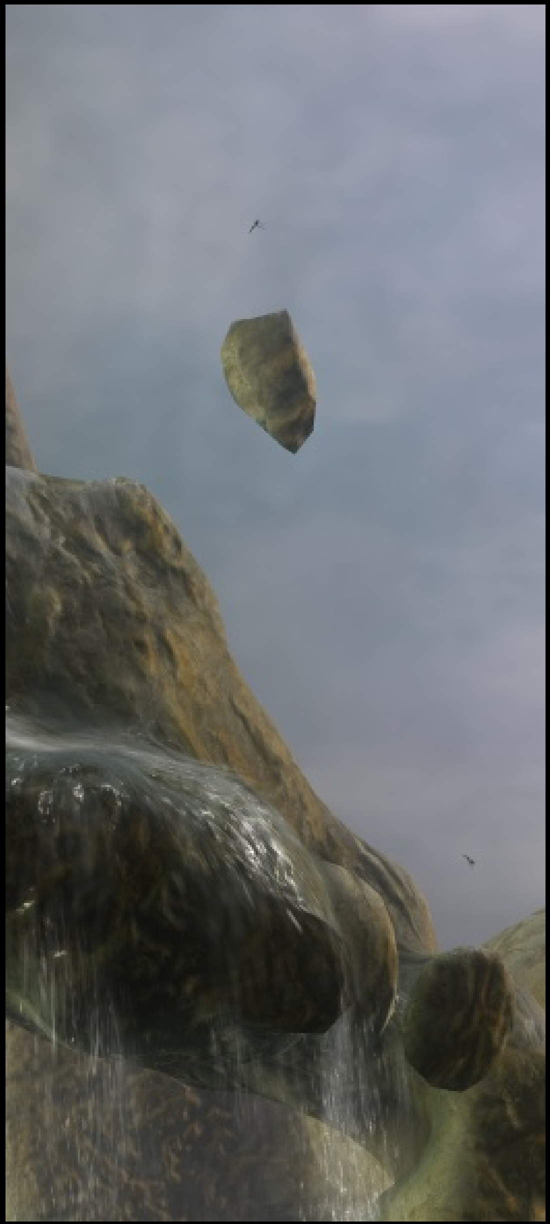
Marching Cubes Geom. Shader

```
GSOutput PlaceVertOnEdge( vsOutputGsInput input, int edgeNum )
{
  // Along this cell edge, where does the density value hit zero?
  float str0 = dot(cornerAmask0123[edgeNum], input.field0123) +
               dot(cornerAmask4567[edgeNum], input.field4567);
  float str1 = dot(cornerBmask0123[edgeNum], input.field0123) +
               dot(cornerBmask4567[edgeNum], input.field4567);
  float t = saturate( str0/(str0 - str1) );  //0..1

  // use that to get wsCoord and uvw coords
  float3 pos_within_cell = vec_start[edgeNum]
                         + t * vec_dir[edgeNum]; //[0..1]
  float3 wsCoord = input.wsCoord.xyz
                 + pos_within_cell.xyz * wsVoxelSize;
  float3 uvw = input.uvw + ( pos_within_cell *
                             inv_voxelDimMinusOne ).xzy;

  GSOutput output;
  output.wsCoord_Ambo.xyz = wsCoord;
  output.wsCoord_Ambo.w   = grad_ambo_tex.SampleLevel(s, uvw, 0).w;
  output.wsNormal         = ComputeNormal(tex, s, uvw);
  return output;
}
```

- "<u>Floaters</u>": annoying chunks of levitating rock.
- When generating 2D height maps from noise, you get small islands – no problem.
- In 3D, you get floating rocks…



Floaters

- Difficult to reliably kill polygons on small floaters.
- How does an ant know if he’s on a $1\ meter^3$ rock or a $10\ meter^3$ rock?

Floaters

<u>The Floater Test</u>: for each voxel in which we generate polygons…

- Cast out a bunch of rays.
- Track longest distance a ray could go *without exiting the rock*.
- Gives a good estimate of the size of the rock.

- If "parent rock" small, don't generate polygons for this voxel.
- Fast dynamic branching very helpful.
- Second pass can help too. *(See notes)*

Shading

Shading

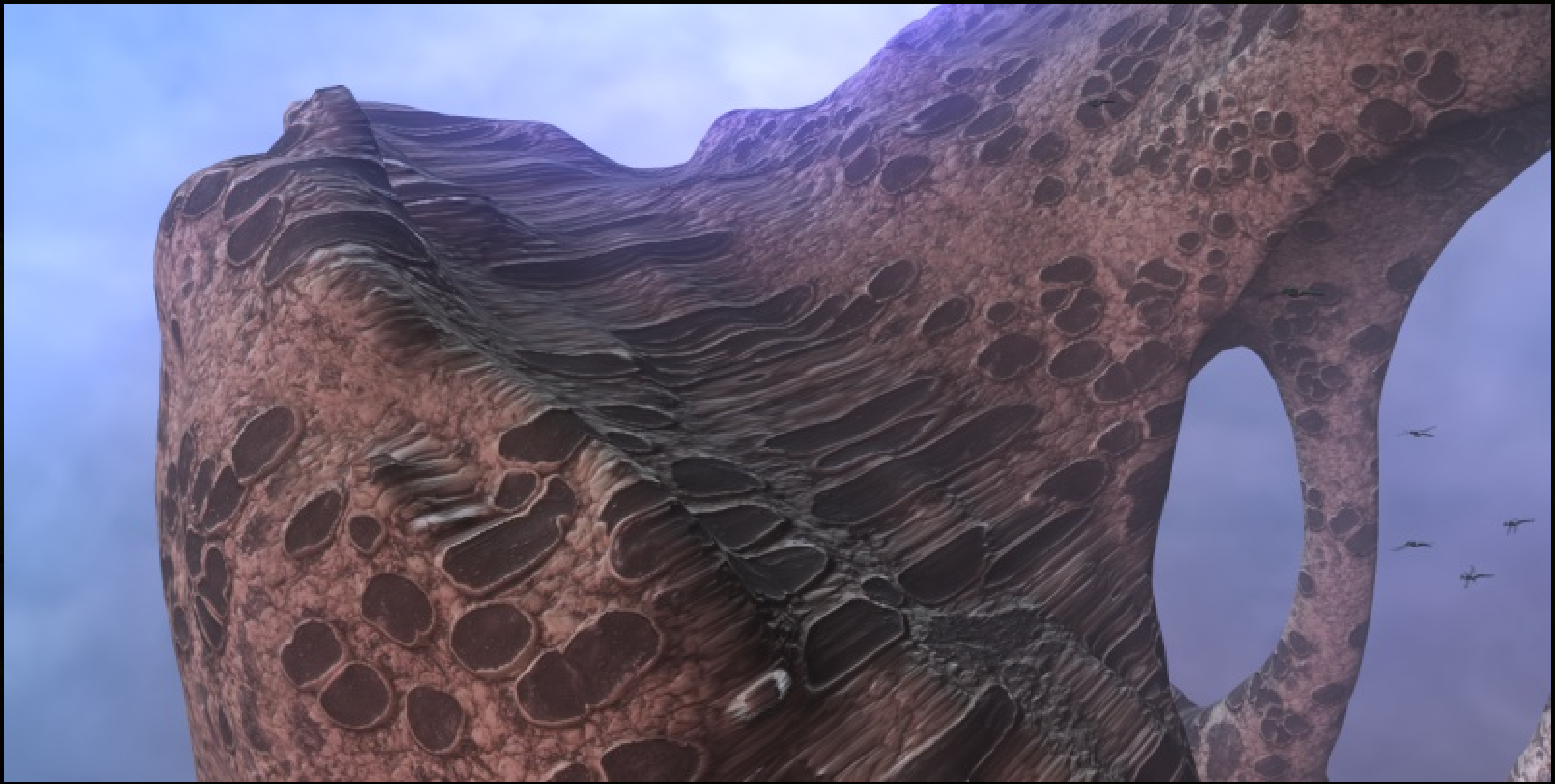
- Rock rendered in one pass with one big pixel shader
- 398 instructions, *not counting loops.*
- Shading topics to cover:
  - Texture coordinate generation
  - Lighting
  - ‘Wet Rock’ effects
  - Detail maps
  - Displacement Mapping

Texture Coordinate Woes

Texture mapping: UV Layout

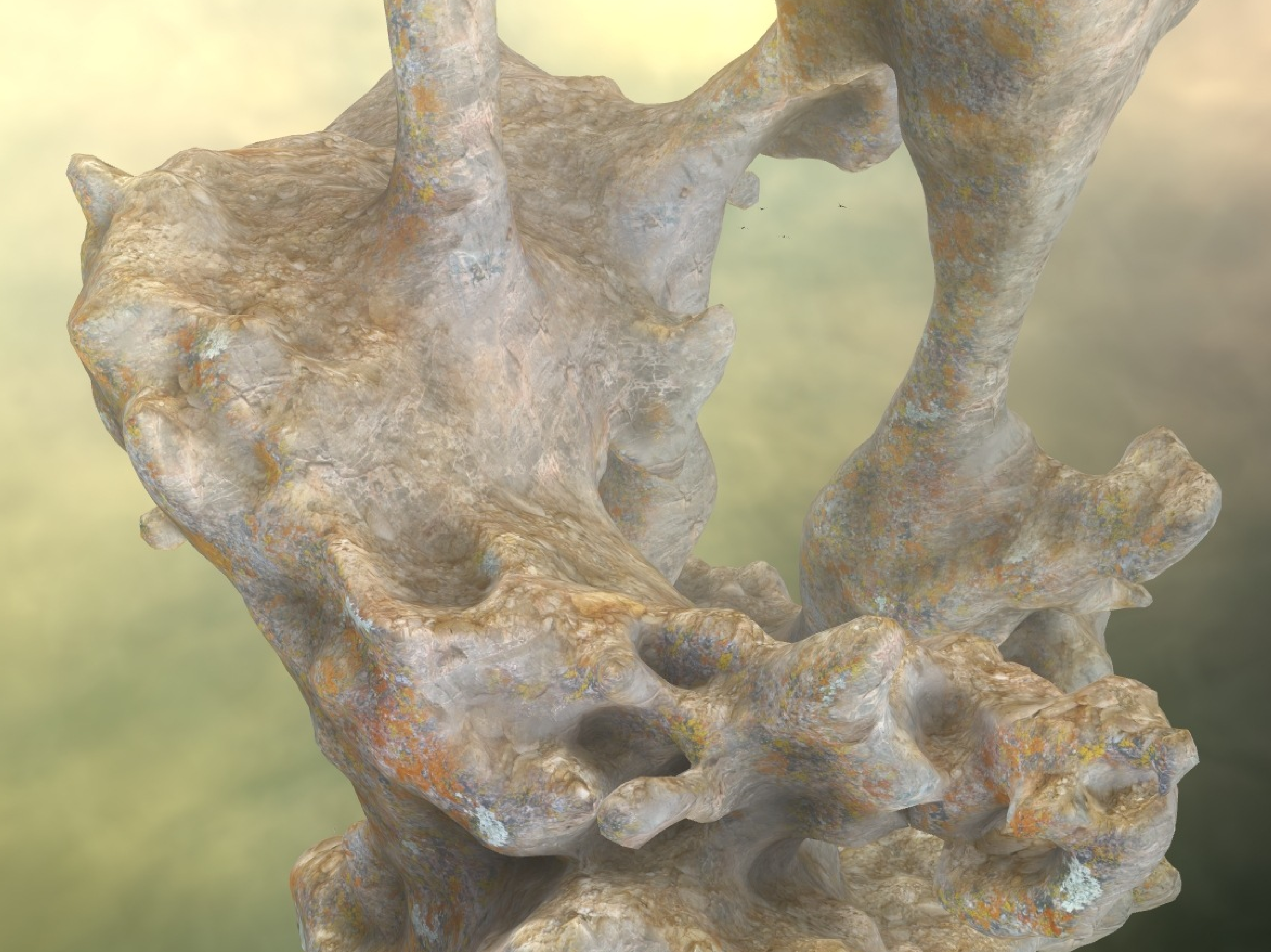
- Games: models have manually-created UV layouts for texture mapping.
- No good for procedural geometry with *arbitrary topology.*

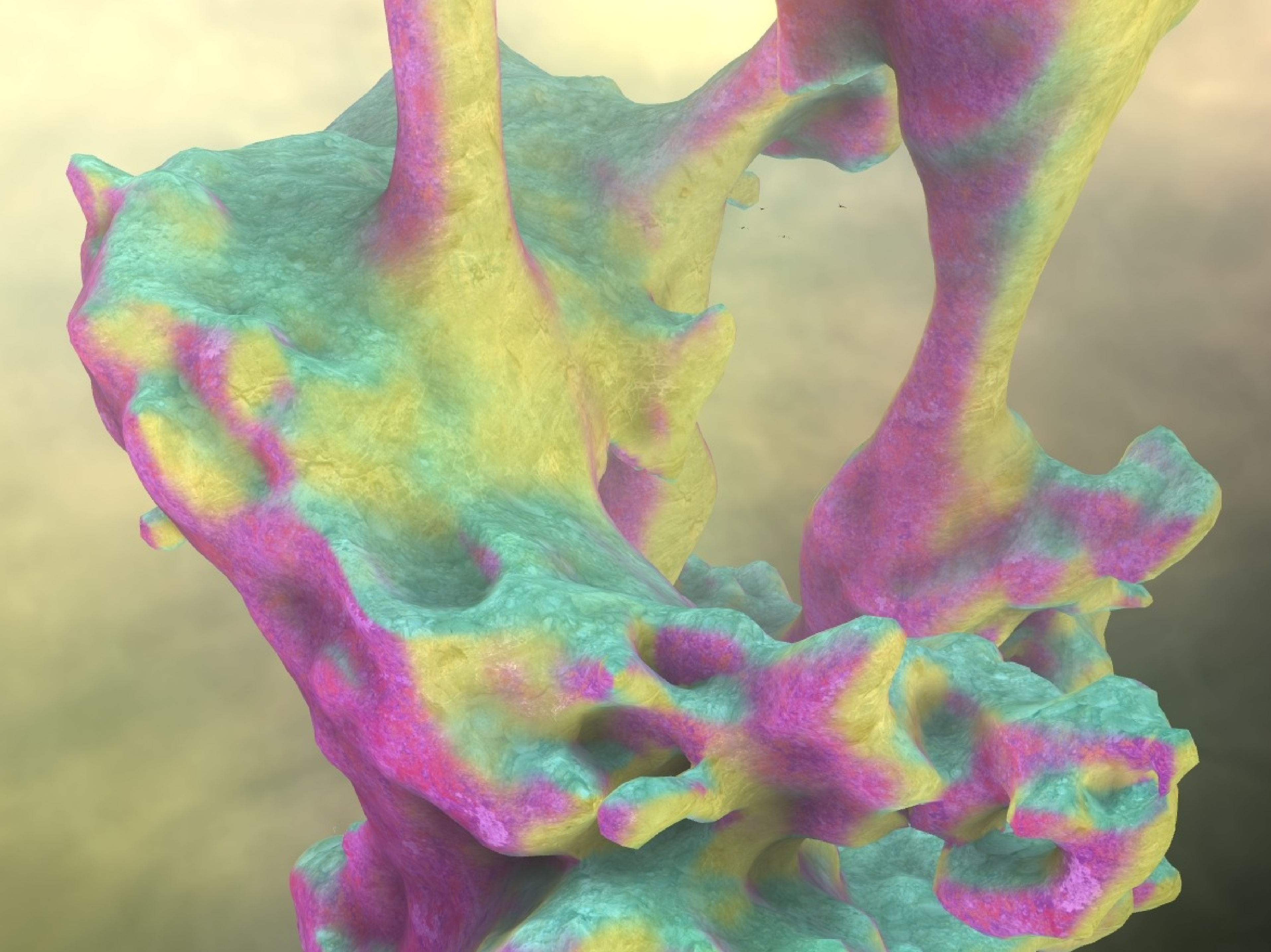
Planar projection along one axis:

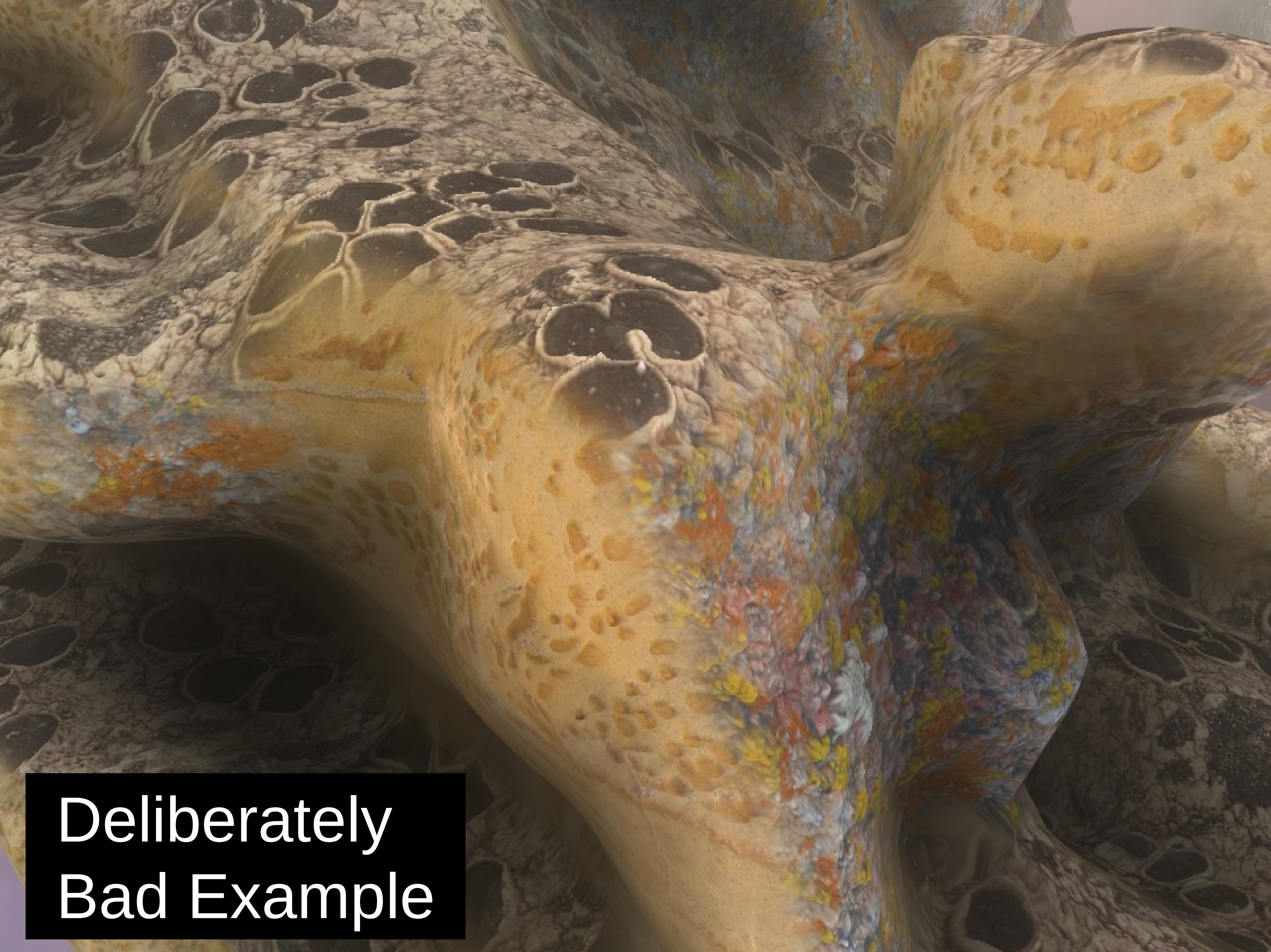


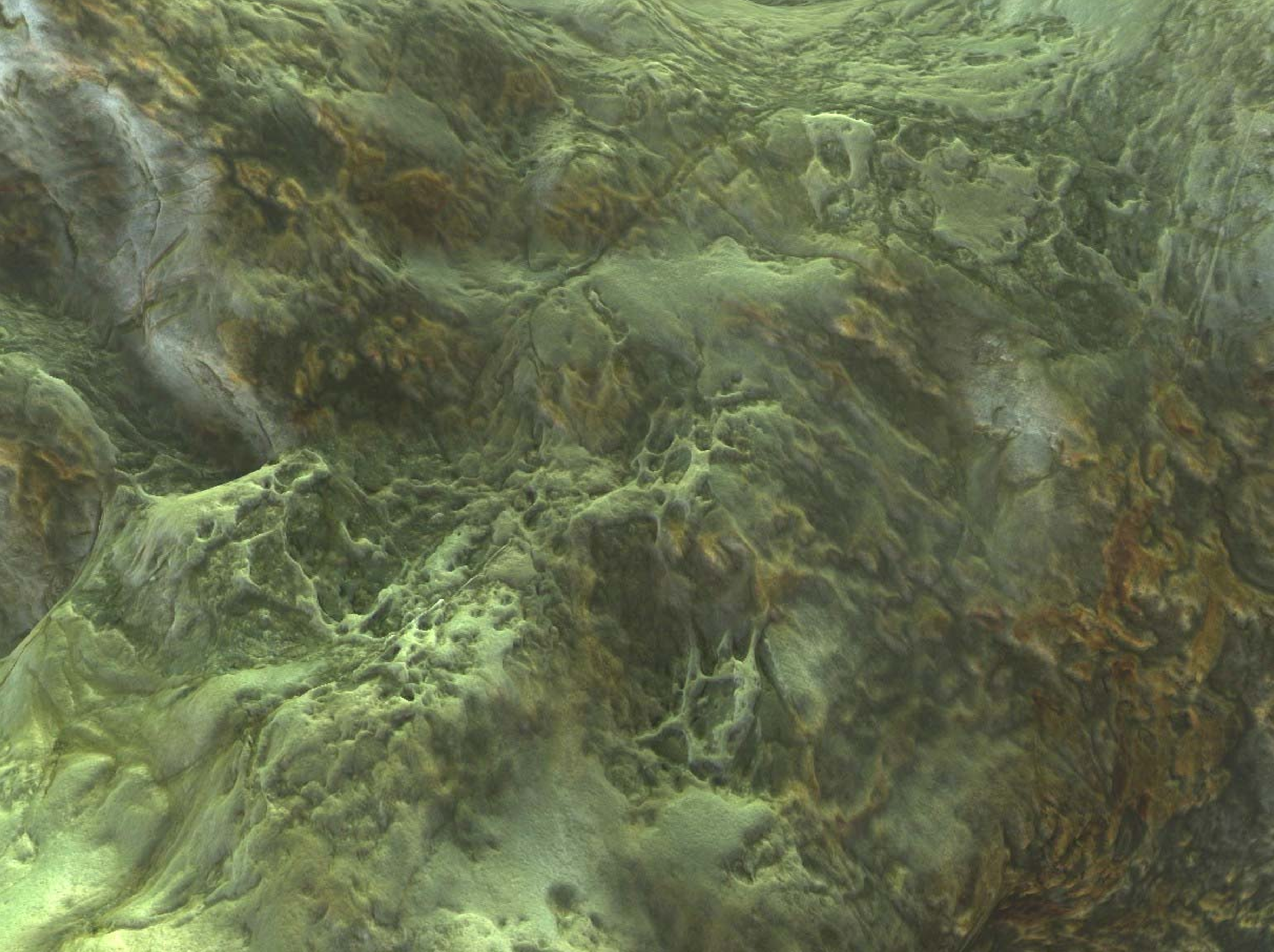
Tri-Planar Texturing

- Solution: <u>Tri-Planar Texturing</u>
- Project 3 different (repeating) 2D textures along X, Y, and Z axes; blend between them based on surface normal.
- For surface points facing mostly in +X or –X, use the YZ projection… etc.
  - Minimizes stretching / streaking.






Deliberately
Bad Example



Tri-planar Texturing

For each pixel:

1. <u>For each projection</u> (X, Y, Z):
   a. project (convert wsCoord to UV coord)
   b. determine surface <u>color</u> & <u>normal</u> based on that projection

2. Blend between the 3 colors & normals based on the original (unbumped / vertex) normal.
   [next slide]

Blending the 3 together...

- Blending amount based on abs(normal), but blend 'zone' is narrowed via a scale & bias.

```
float3 blend_weights = abs(N_orig) - 0.2;
blend_weights *= 7;
blend_weights = pow(blend_weights, 3);
blend_weights = max(0, blend_weights);
// and so they sum to 1.0:
blend_weights /= dot(blend_weights, 1);
```

Low-frequency Color Noise

- Repeating textures can get dull…
- Reduce monotony by sampling a 3D noise texture at a low frequency, and using that to vary the surface color.

```
const float freq = 0.17;
float3 noiseVal = noiseTex3D.Sample(
   LinearRepeat, wsCoord*freq ).xyz;
moss_color.xyz *= 0.9 + 0.1*noiseVal;
```

No colorization

Colorization (exaggerated)

Lighting

- 3 directional lights, no shadows.
- Typical diffuse and phong specular lighting.
- To save math, lighting is (dynamically) baked into two float4_16 cube maps:

	Equation	Face size
1. Diffuse light cube	(N-dot-L)	16x16
2. Specular light cube	(R-dot-L)^64	32x32

- Lighting influenced by bump vectors & ambient occlusion values from rock generation process.

Lighting

- Diffuse light modulated by `occlusion` value as-is.

- Specular light falls off more quickly.
  - Spec is modulated by:

    ```
    saturate((occlusion – 0.2) / 0.2)
    ```

  - Makes specular highlights fall off very quickly in recesses.

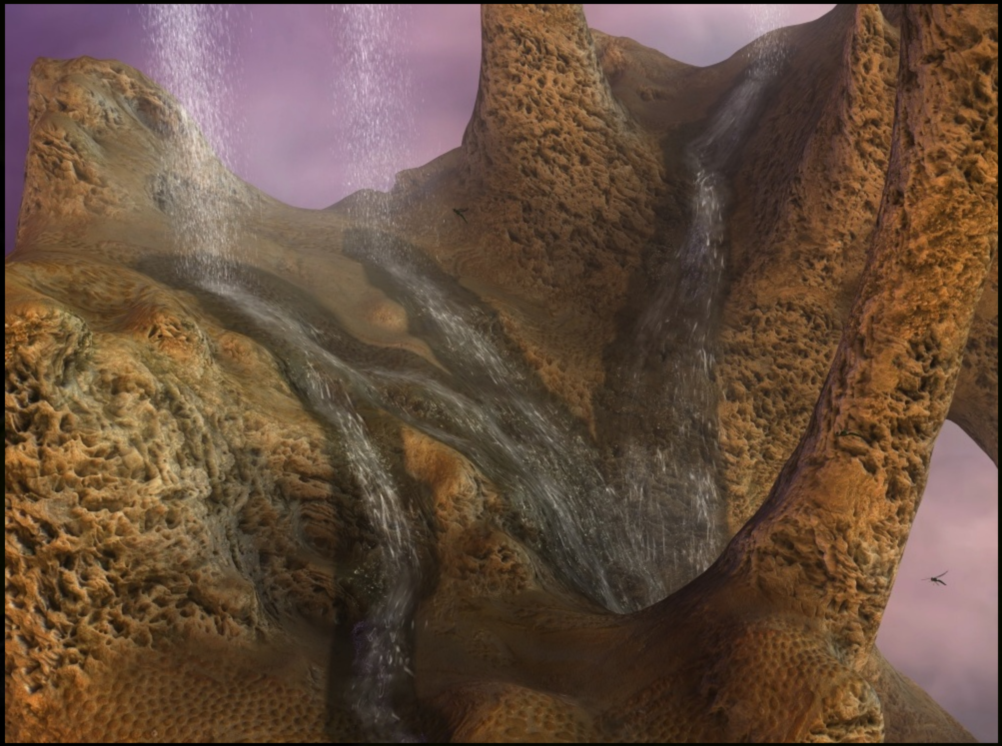

No ambient occlusion


Occlusion reducing
diffuse light only

Occlusion reducing
diffuse and specular light

Wet Rock

- Rock gets wet when water flows nearby.
- Tiny water ripples also visible, flowing down the rock.
- Remove waterfalls → wet rock dries up over time.




Demo

Wet Rock

- Wetness Map, a 3D texture, tracks where water has been flowing.
  - ½ resolution (48 x 48 x 128)
- When shading rock, sample wetness map to know how wet a pixel should be.

Wet Rock

- Making the rock look wet:
  1. Darken diffuse color
  2. Increase specular reflectivity
  3. Perturb normal using 3 octaves of <u>animated noise</u>.

Animated ‘Wetness’ Noise

- **Just barely perturbs the normal**
  - **…only visible through dancing specular highlights.**
- **Just add it once (after tri-planar projection)**
- **For each octave:**
  1. Start with world-space coord (but at varying scales/swizzles)
  2. Add current Time value to .Y – creates downward flow
  3. Sample the noise volume.
- **Use mipmap bias of +1 (slightly blurry).**

Displacement Mapping

Review of Pixel Shader techniques:

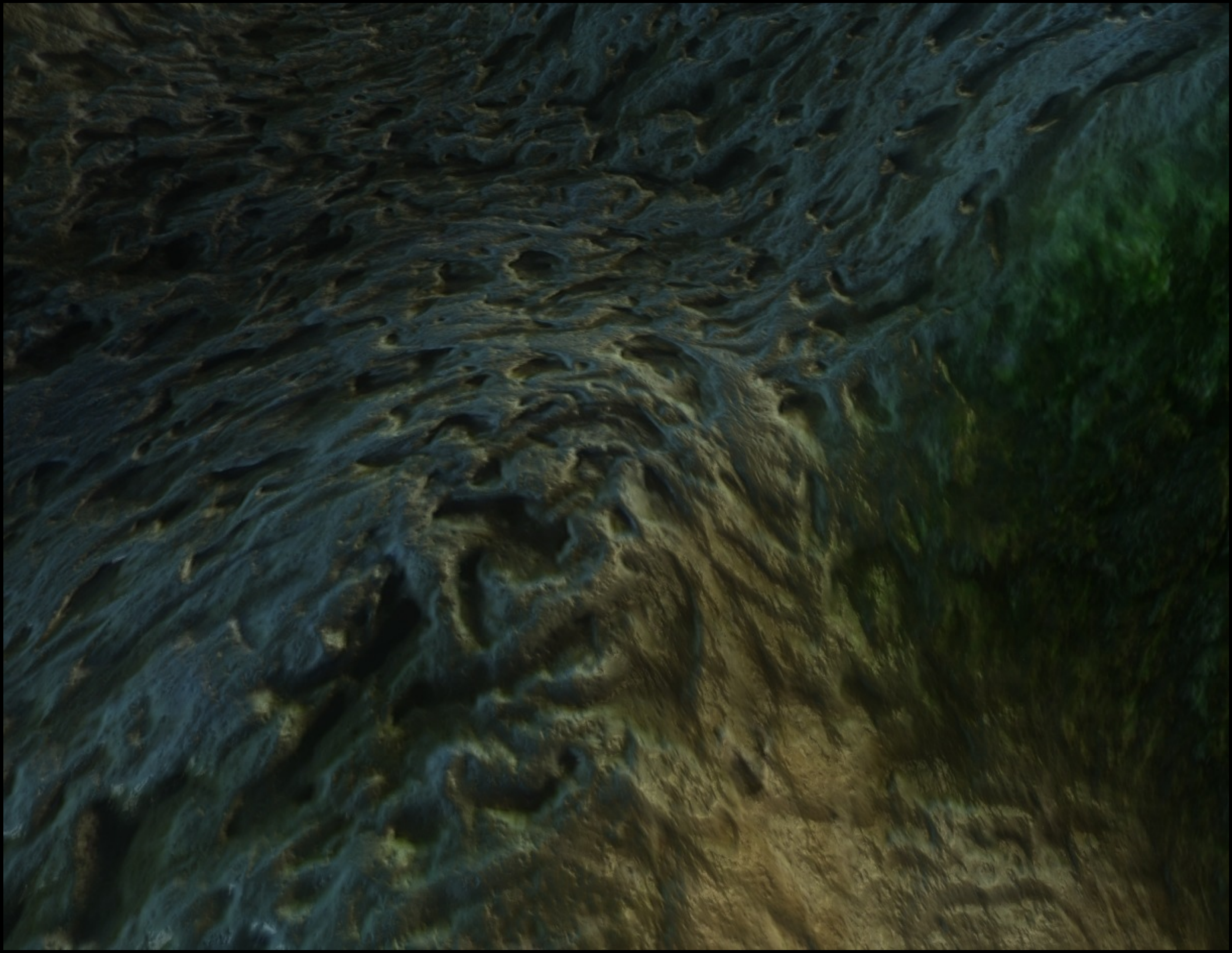
- Bump Mapping / Normal Mapping plays with the normal, and hence the light, so geometrically flat surfaces can appear “wrinkled.”
- Parallax Mapping virtually pushes texels “underneath” the polygon surface, at varying depths, creating the illusion of parallax as the camera angle changes.
- Pixel Shader Displacement Mapping adds the ability for texels to *occlude* each other.

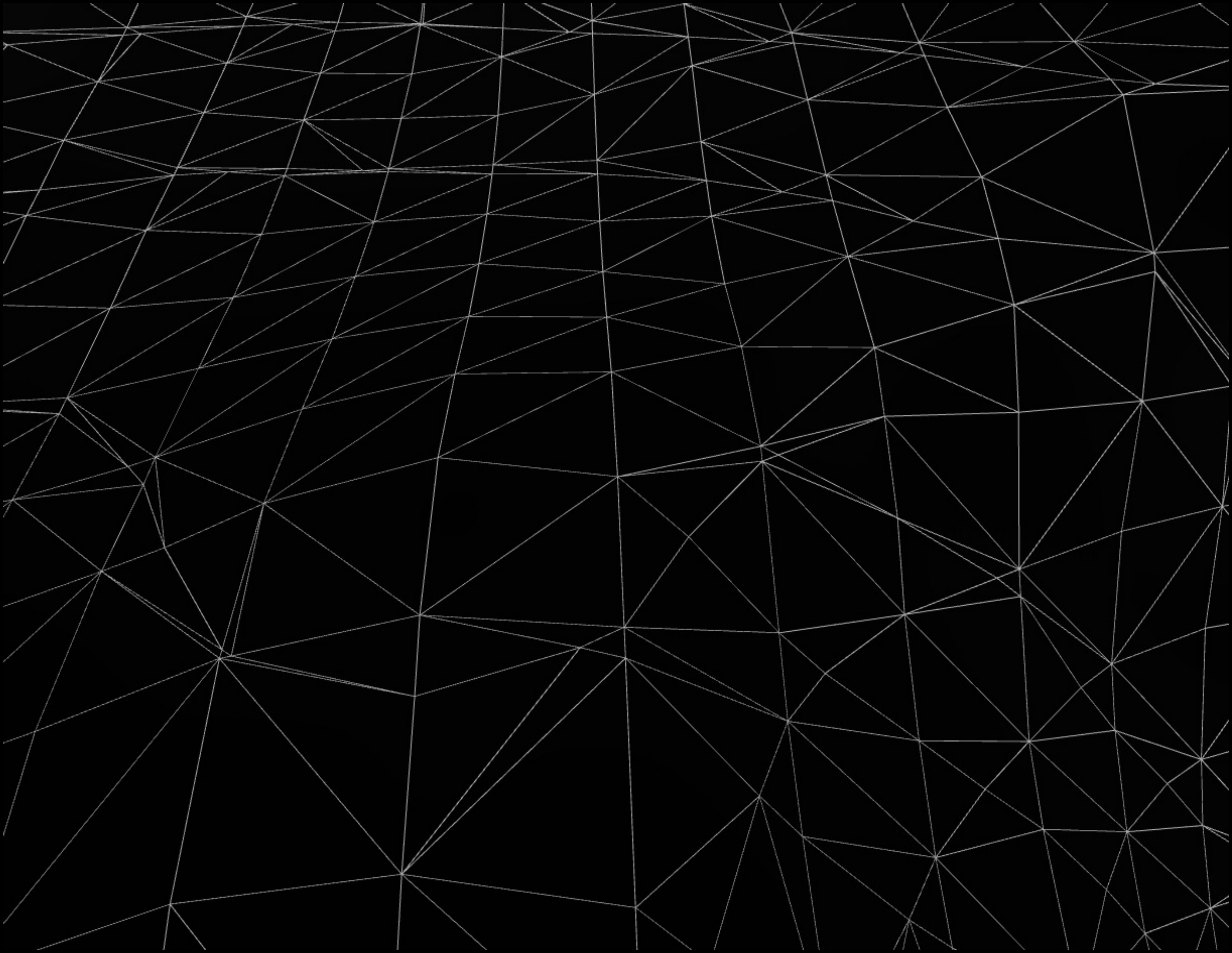
Displacement Mapping

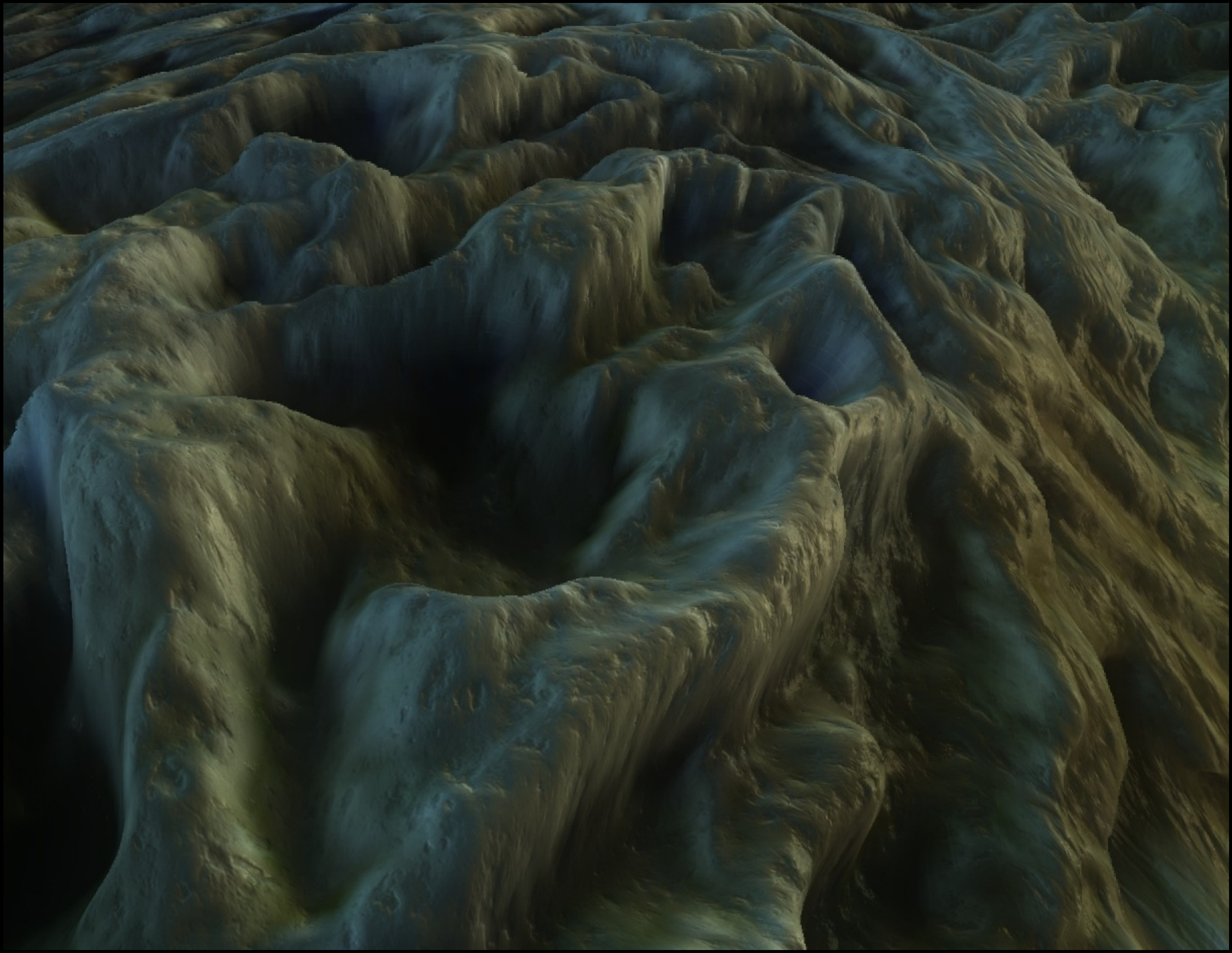
Our displacement technique:

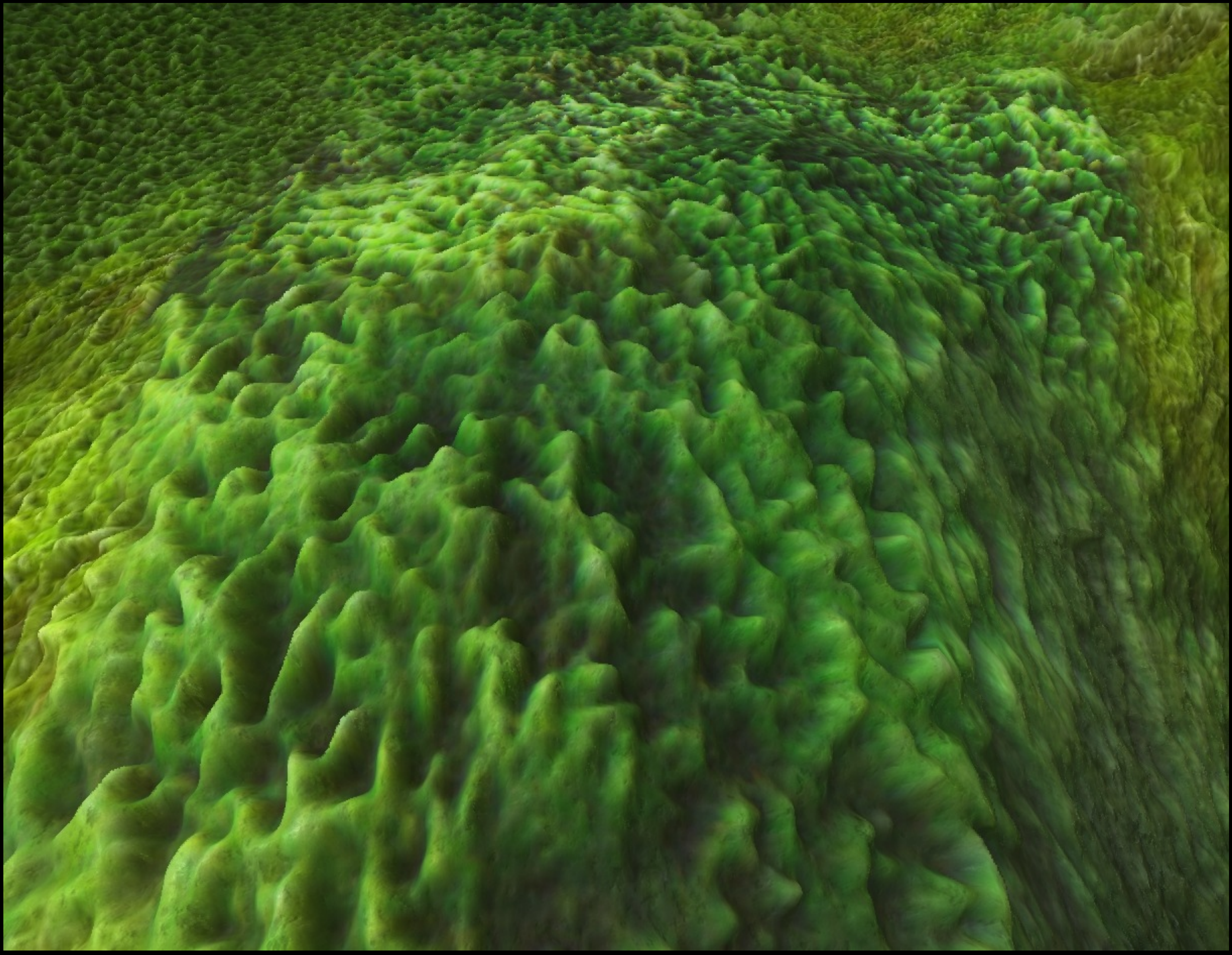
- A height map, matched to the color / bump map, “sinks” texels to various depths below the polygon surface.
- Brute-force ray cast.
- Uses simple height map.
  - (no precomputed cone maps, etc.)
- Works with tri-planar texturing.

- Demo -









Displacement Mapping

For each pixel…

For each planar projection…

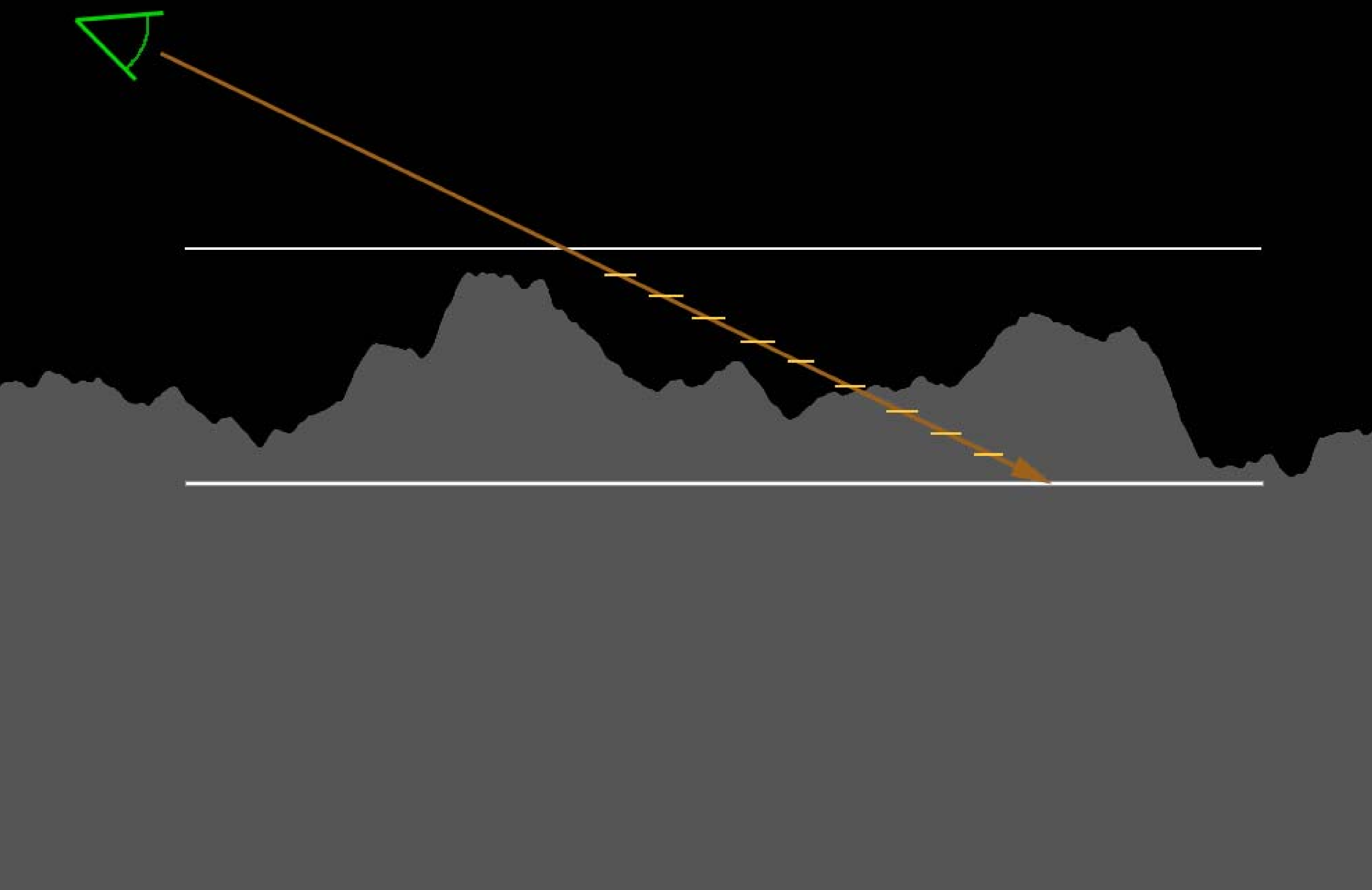
- Start with the original UV coordinates.
- Run the displacement algorithm; you end up with <u>modified UV coordinates</u> (UV’).
- Use UV’ for the final color / bump texture lookups for this projection.

Displacement Mapping

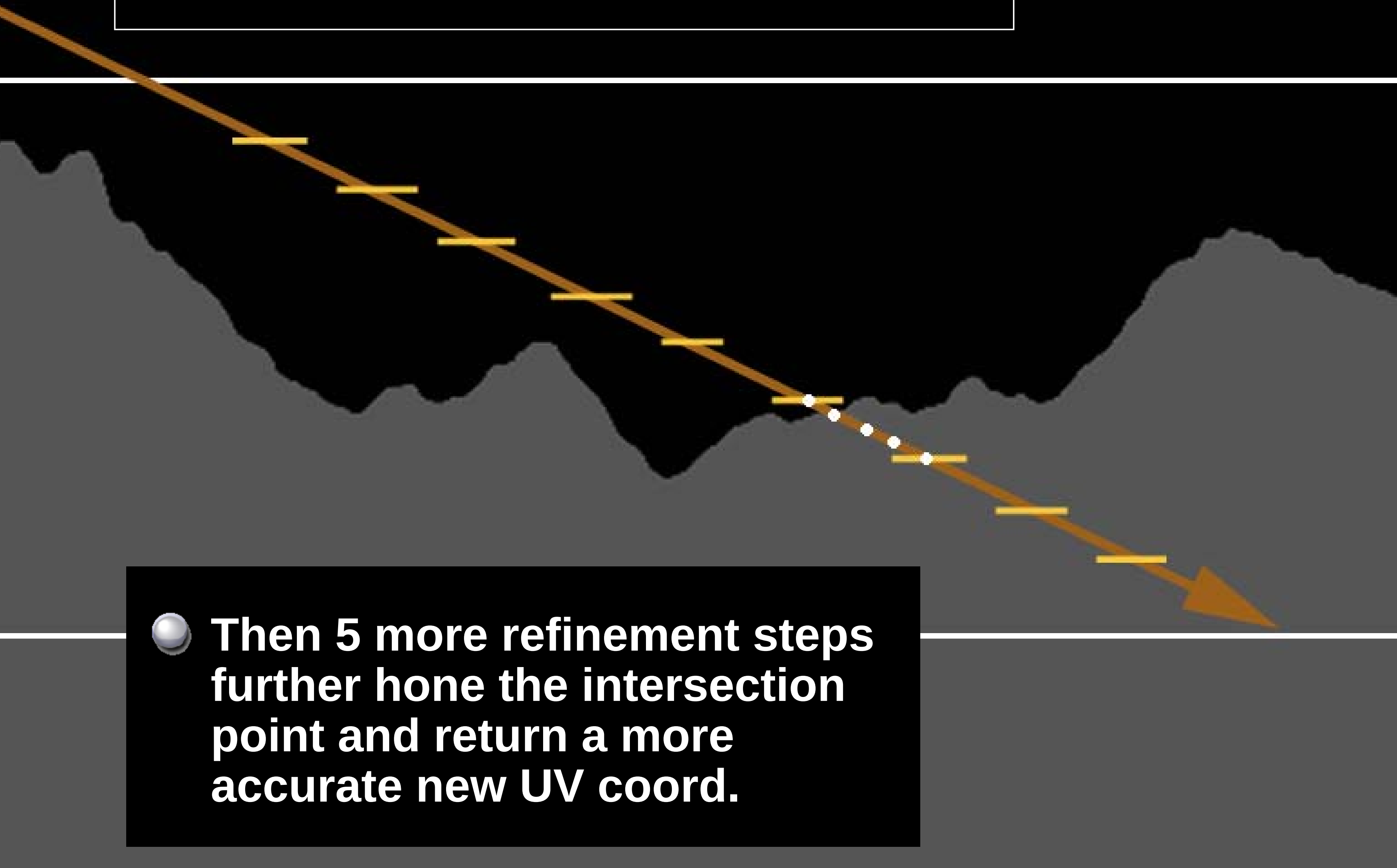
Finding the modified UV coordinates:

- March along the eye ray - 10 steps - until you hit the virtual [sunken] surface.
- At each step…

  1. get ray depth below surface
  2. get UV coord (re-project)
  3. sample height map at UV coord
  4. first time ray depth exceeds that of sample,
     we hit the rock; hang on to those UV coords.



- The first 10 steps determine the inter-texel occlusion silhouette.



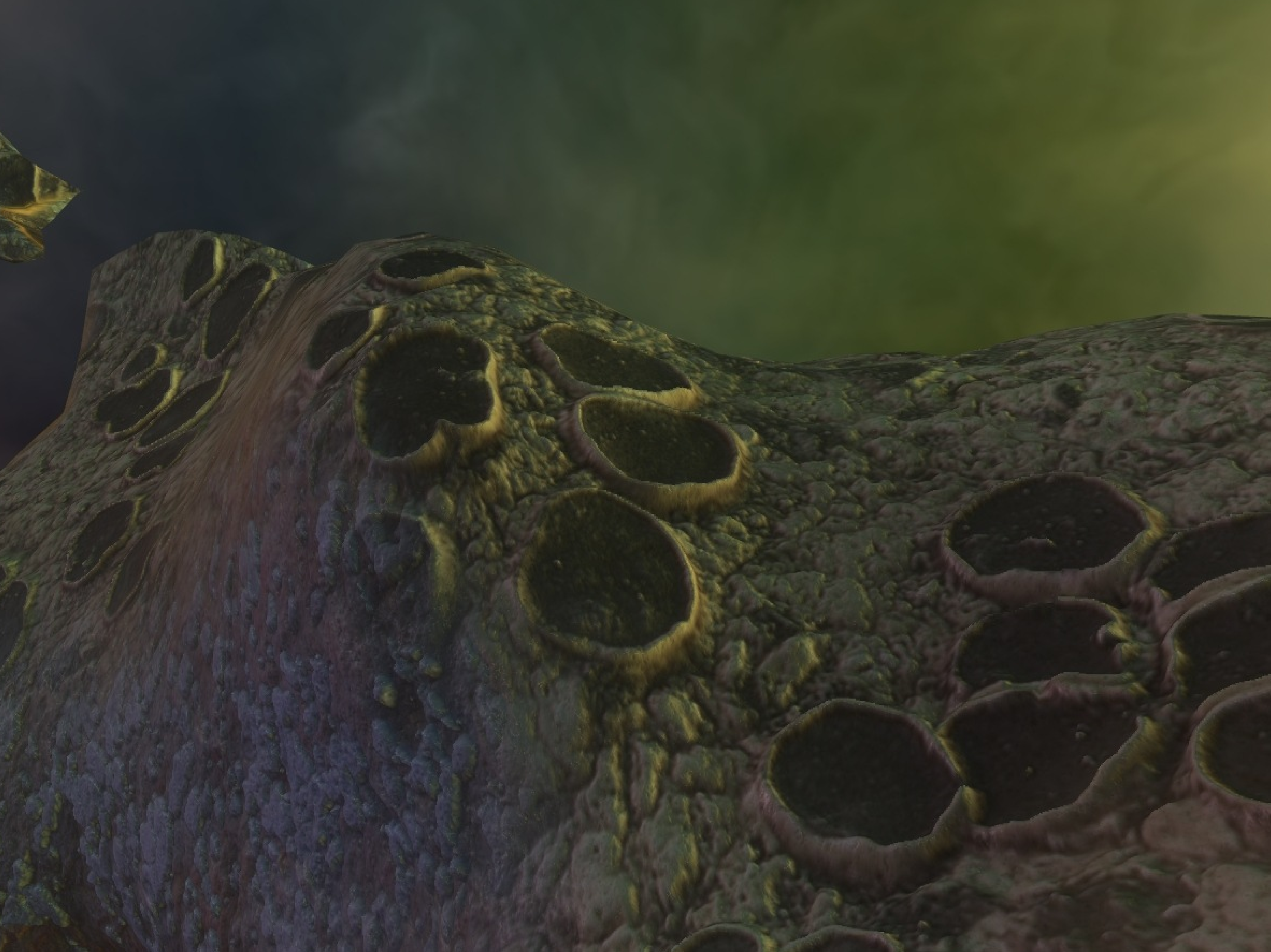
- Then 5 more refinement steps further hone the intersection point and return a more accurate new UV coord.

Displacement Mapping

```
float2 dUV = -tsEyeVec.xy * 0.08;  //~displm't depth
float  prev_hits = 0;
float  hit_h = 0;                  // THE OUTPUT
for (int it=0; it<10; it++) {
  h  -= 0.1f;
  uv += dUV;
  float h_tex = HeightMap.SampleLevel(samp,uv,0).x;
  float is_first_hit = saturate(
                 (h_tex – h - prev_hits)*4999999 );
  hit_h += is_first_hit * h;
  prev_hits += is_first_hit;
}
```

Displacement Mapping

- <u>Dynamic Branching</u> helps immensely
  - Usually skip 1-2 projections based on the normal
  - Skip all 3 if pixel far away!
- Not covered here: “basis fix” to make the displacement extrude in the direction of the actual polygon face.




16 / 8


10 / 5


7 / 3


4 / 2

Displacement Mapping

- Height maps get sampled OFTEN…
- Therefore:
  - Keep separate (don’t pack into color alpha channel!) for happy caching
  - Use DXGI_FORMAT_R8_UNORM (mono, 8 bits) or DXGI_FORMAT_BC4_UNORM (mono, 4 bits).
  - Photoshop Tips:
    - Gaussian blur (high frequencies bad)
    - Hi-pass filter (keeps displacement “happening”)

1) original color map

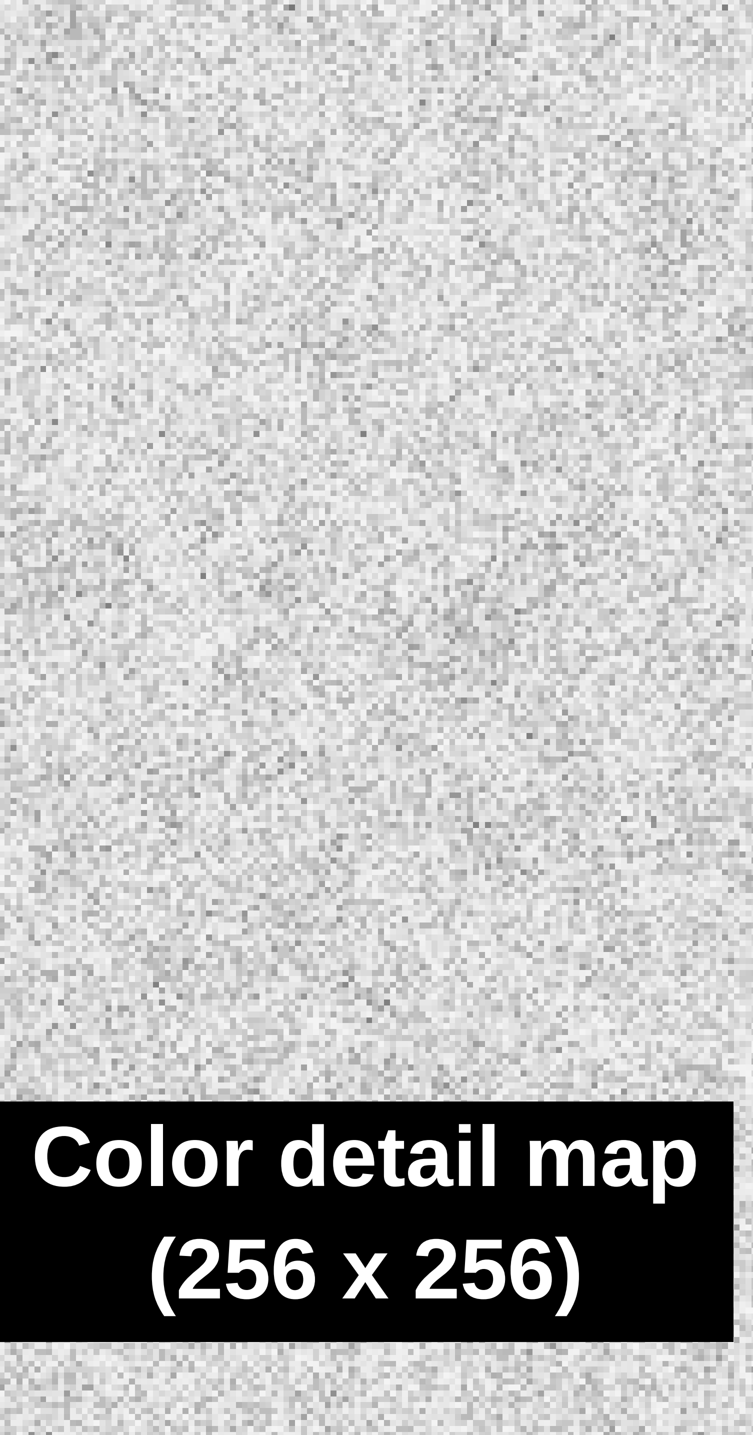
2) green ch + auto levels

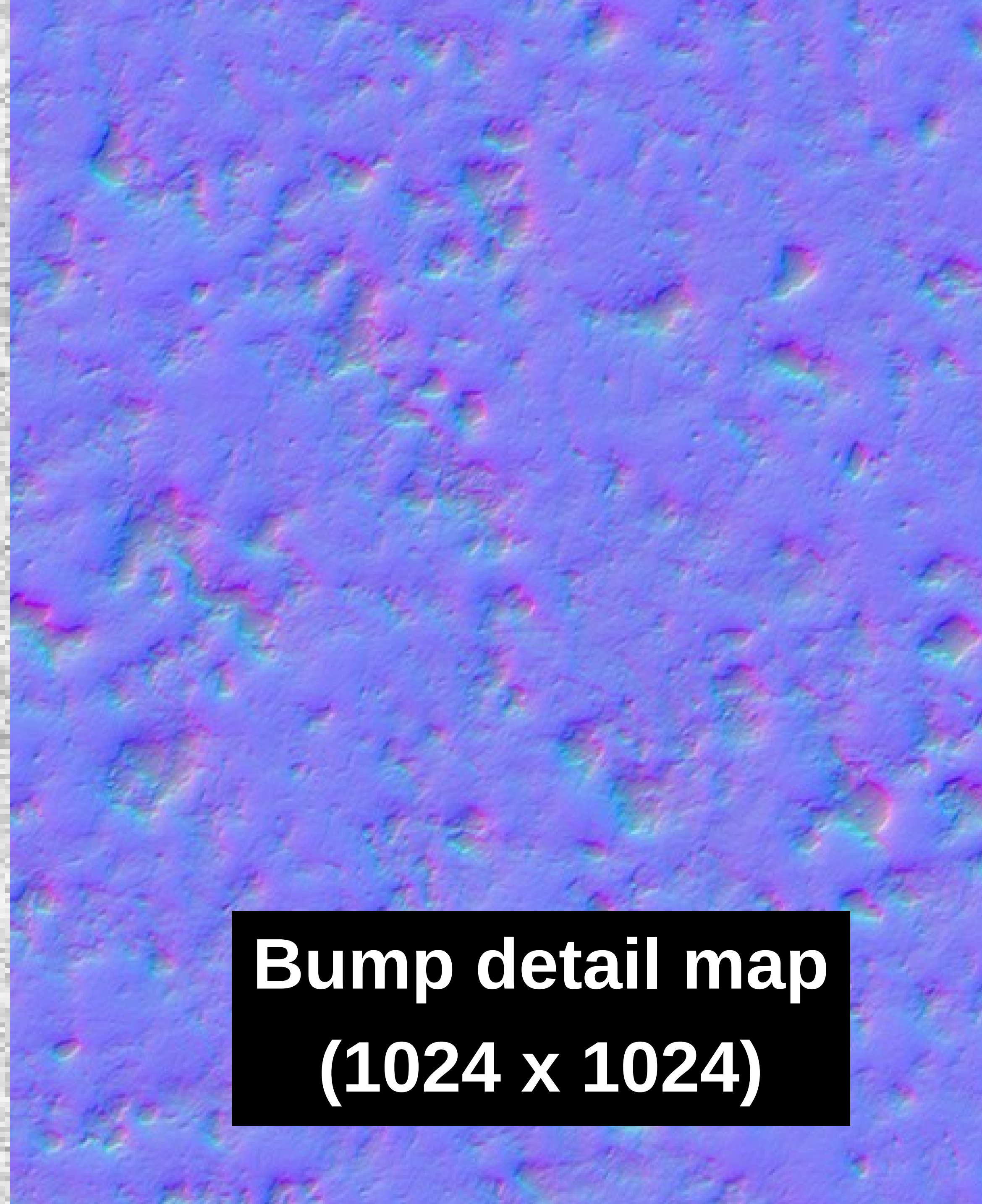
3) HPF @ 16

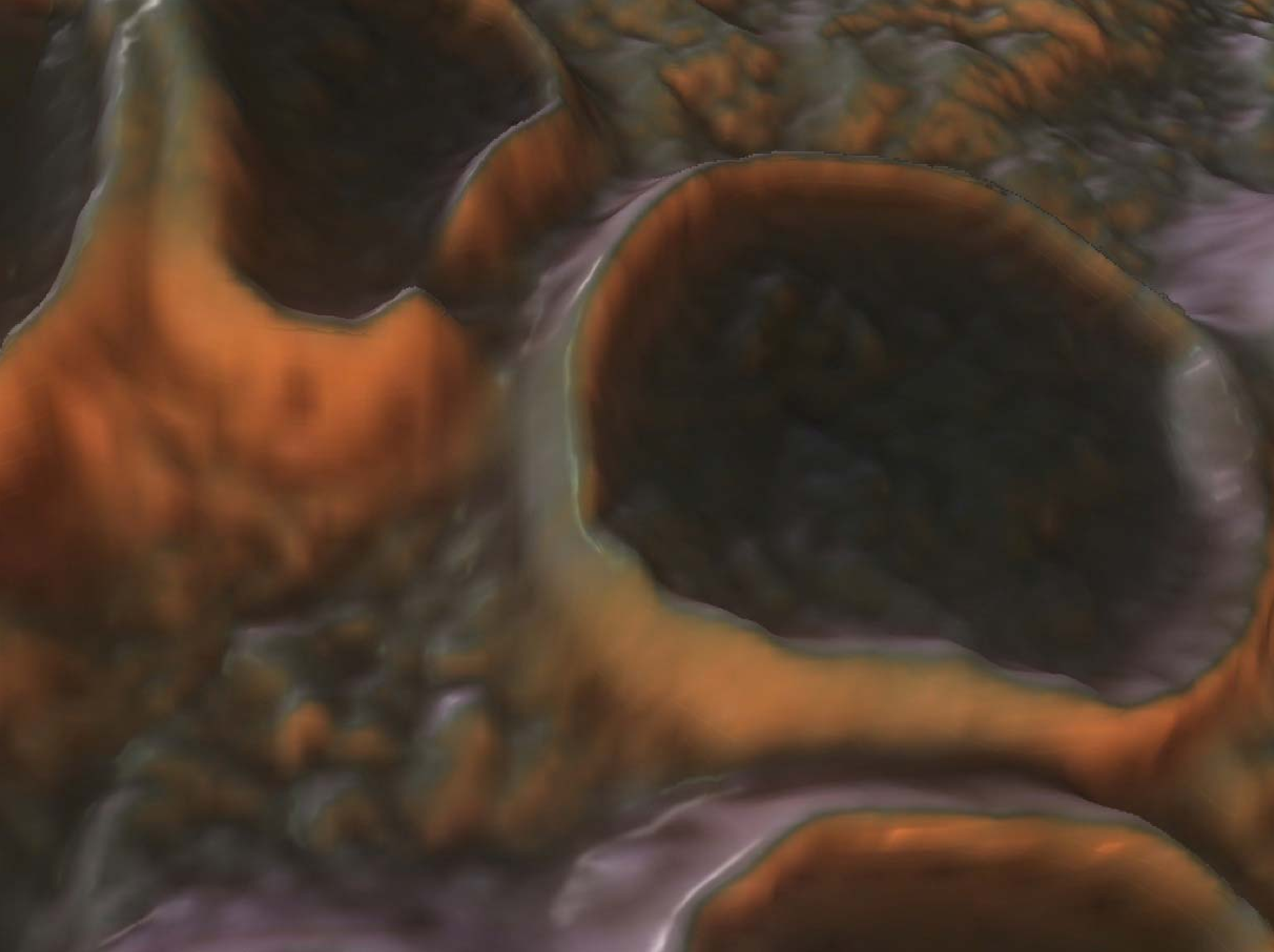
4) HPF @ 4

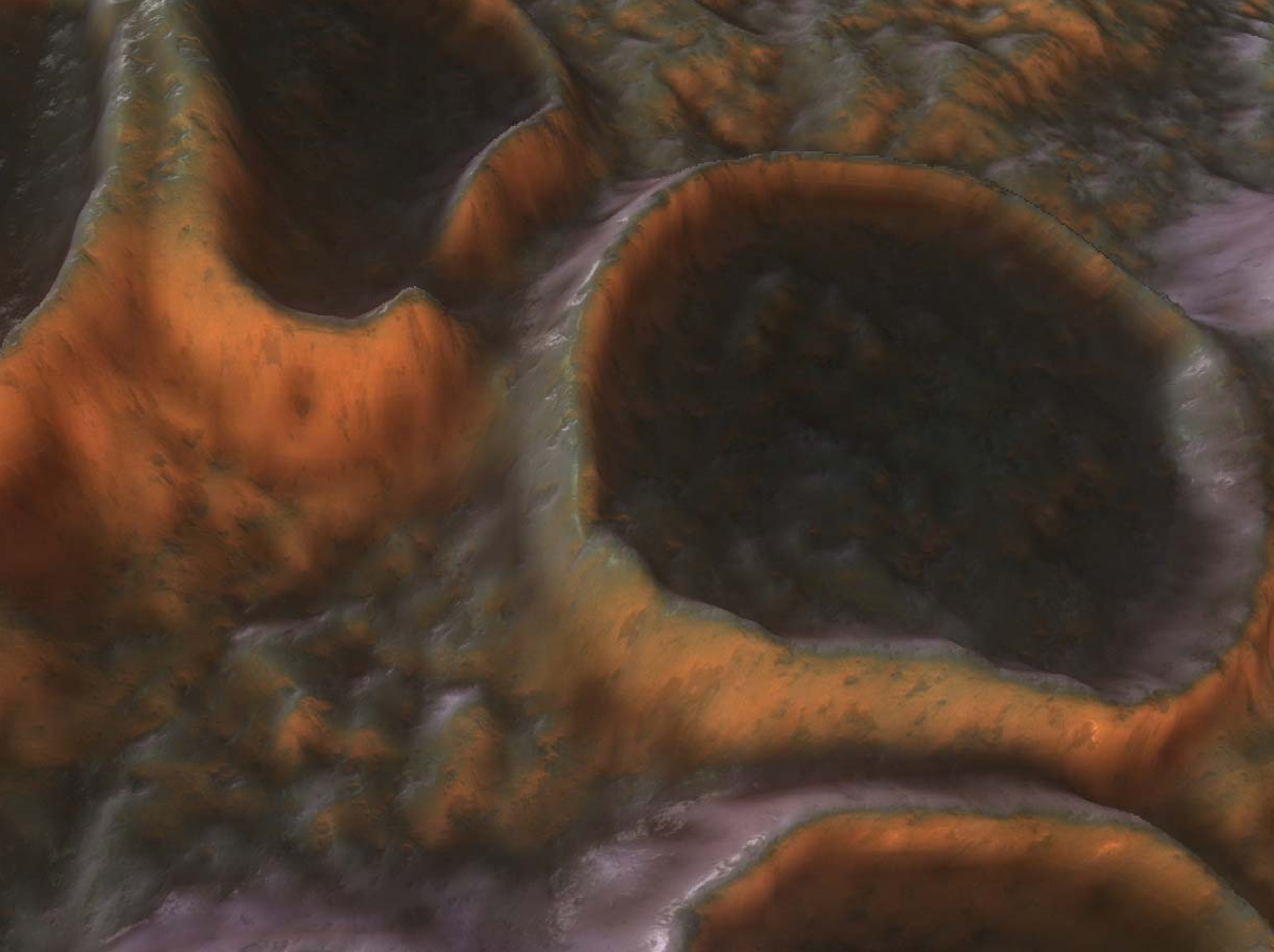
Detail Maps

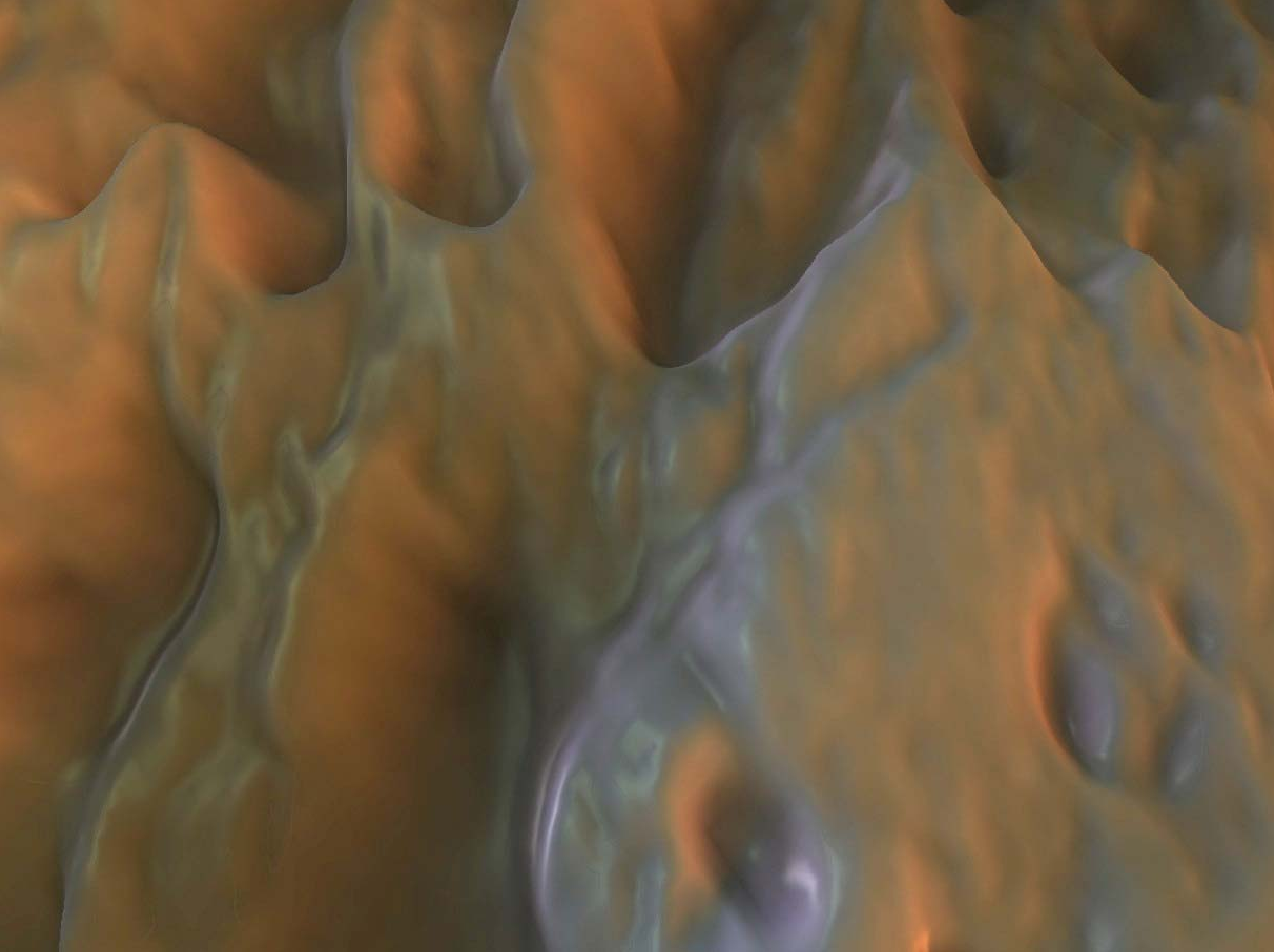
- Detail Maps enhance textures when viewer very close to surface.
  - Otherwise we see large, ugly, bilinearly-interpolated texels.
- Just one set of detail textures for the whole demo.
  - one <u>color detail</u> texture (~sandy noise)
  - one <u>bump detail</u> map (~divots, creases).

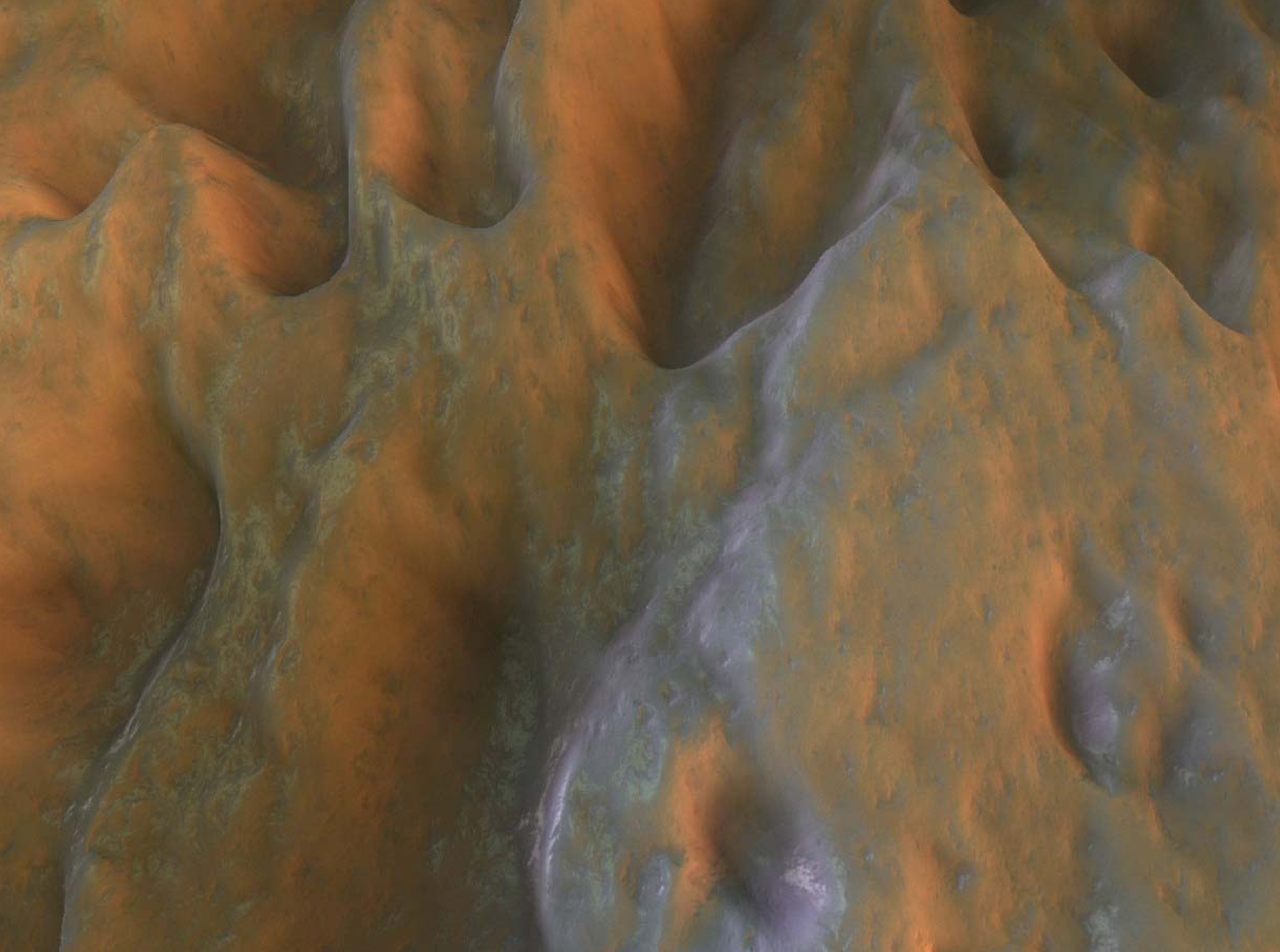

Color detail map
(256 x 256)


Bump detail map
(1024 x 1024)









Texture Creation

- 19 rock texture sets
- Each has 3 coordinated maps:
  - Color map (1536x1536, 4 ch)
  - Bump map (1536x1536, 2 ch)
  - Displacement height map (1 ch, half-size)
- Looks terrible if they don't match up well… so height maps (for bump, displacement) derived from color maps.
  - Usually from green channel. (?)
  - High-pass filters (radius ~96 pix)

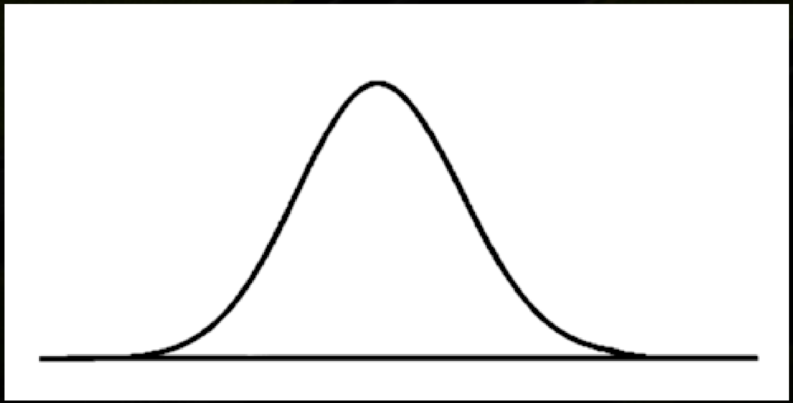
Texture Creation

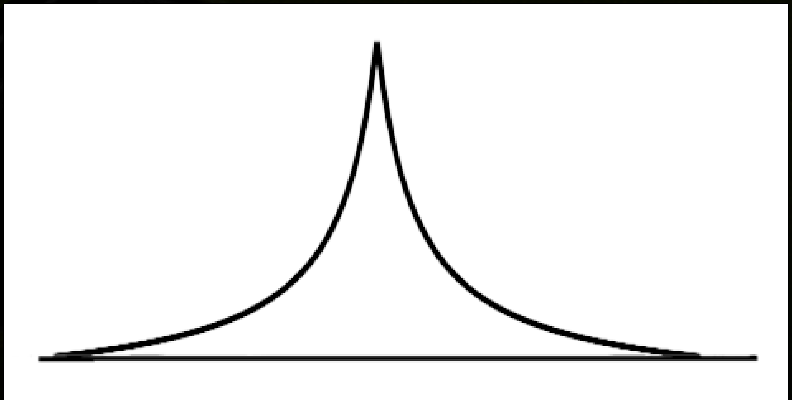
- Most color maps were made from photos.
- Ideally want *evenly lit* rock surface color…
  - Morning fog
  - Or sun perpendicular to rock surface

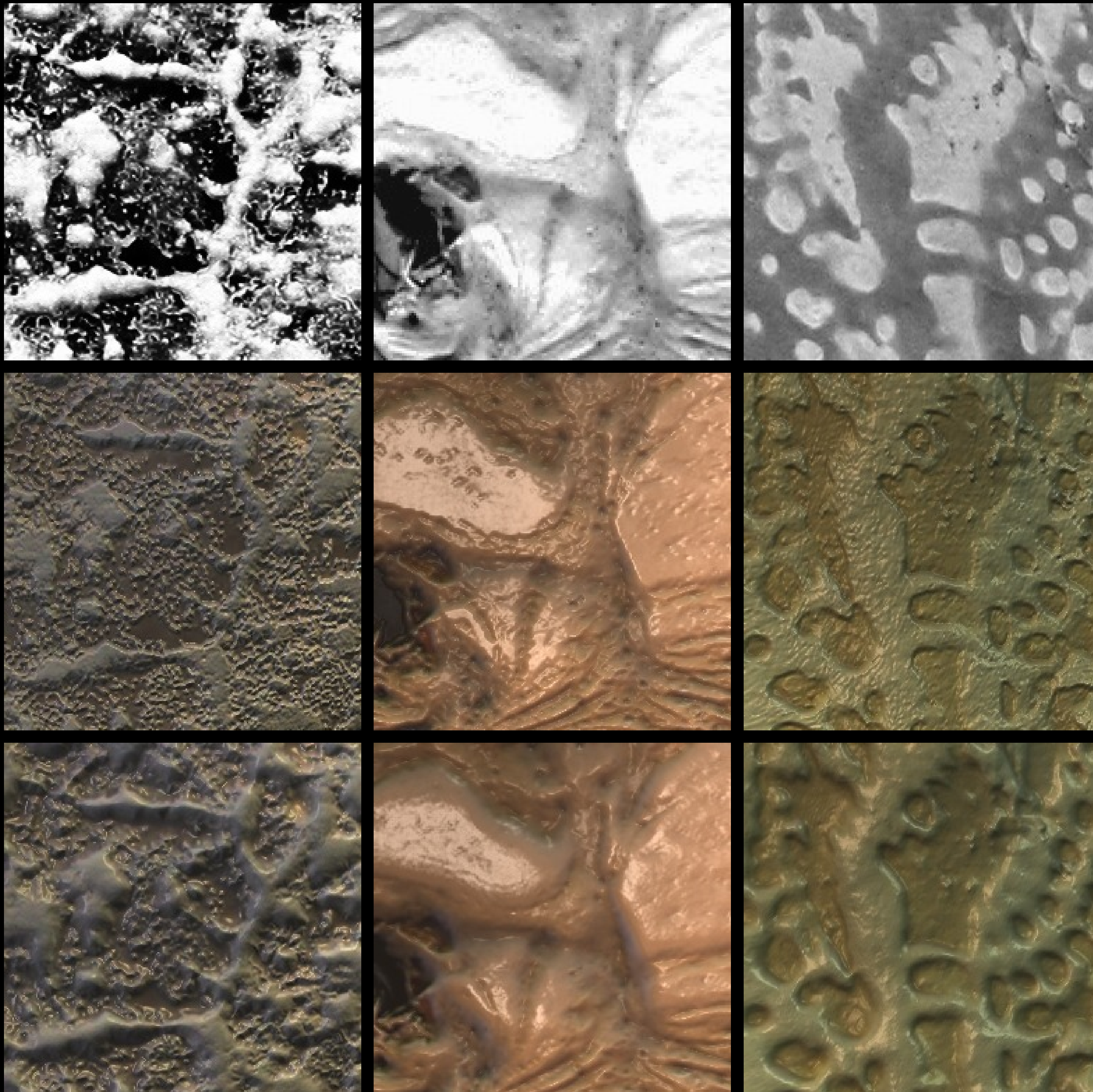


“1/R” Height Map Filtering

- Height maps were run through a special blur kernel before being used to create bump maps.
- Makes resulting bump maps look more organic / less flat.
- Like a gaussian blur, but kernel shape different.
  - Approximated by weighted sum of 4 gaussians of varying radii. See slide notes.







Original height map

Using bump map created from *original* height map

Using bump map created from *1/R-filtered* height map

Water

Water

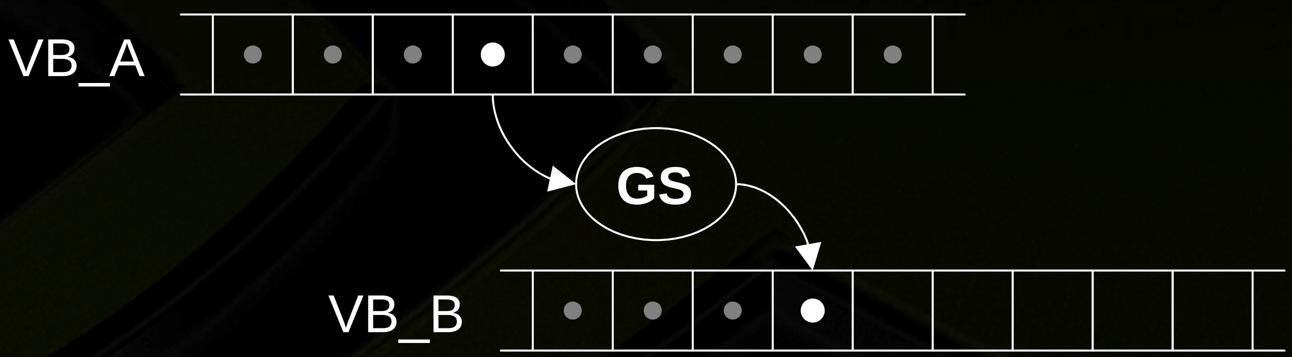
Demo

Structure

- Water is a particle system on the GPU
  - Dynamically flows over arbitrary rock
  - Interactive placement by user
- Stored in a Vertex Buffer
  - Each particle is a vertex
  - Geometry Shader's variable output allows the number of particles to rise and fall

Updating the Particles

- Water VB is double buffered
  - Set up one VB as input
  - Process vertices (particles) in the shader
  - Stream out updated particles to the other VB
  - Next frame, swap VB’s



- Geometry Shader allows variable output
  - A single emitter particle spawns many output particles
  - Expired particles are discarded in the GS


Particle Types
Emitter
Collision Mist
Sliding Water
Falling Water
Falling Mist

Water Particle Types

- Five particles types, in three categories
  - Emitter
  - Water (two types)
  - Mist (two types)
- Particles of all types are stored in the same VB and processed by the same GS
  - Particles can change types
  - Particles can spawn other types of particles
- Dynamic Branching in the shaders enables their different behaviors

Update: Emitter Particles

- In the shader, each input emitter outputs itself plus several new water particles.
- Each waterfall actually has several emitter particles at the same location
  - Parallelize the work of creating new water particles
  - GS performs better with fewer/smaller outputs

Update: Water Particles

- Sliding Water
  - Subject to gravity and sliding friction
  - Sticks to the rock surface
  - Changes back to falling water when it goes over an edge
- Falling Water
  - Subject to gravity and air resistance
  - Handles collisions with rock
  - Turns into sliding water or mist

Water-Rock Interaction

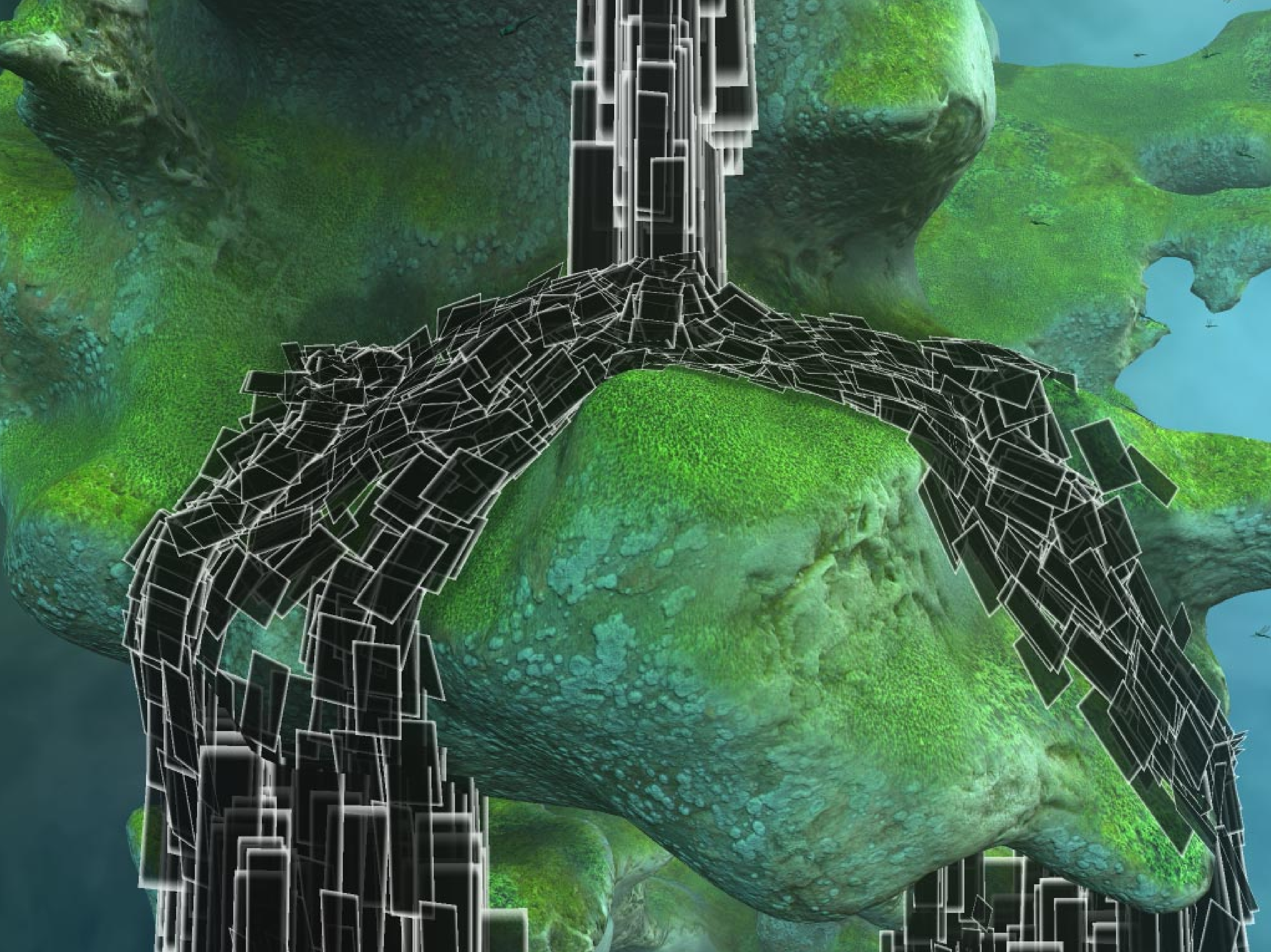
- Rock is fully described by 3D textures
- Use density texture to test for collisions (rock vs. air)
- Use surface normal texture to move the sliding water

Update: Mist Particles

- Falling Mist
  - Created randomly from falling water
  - Water particles live longer than mist particles
- Collision Mist
  - Created sometimes when falling particles collide with rock
- (Both Mist Types)
  - Move like Falling Water
  - Cannot change back to being Water

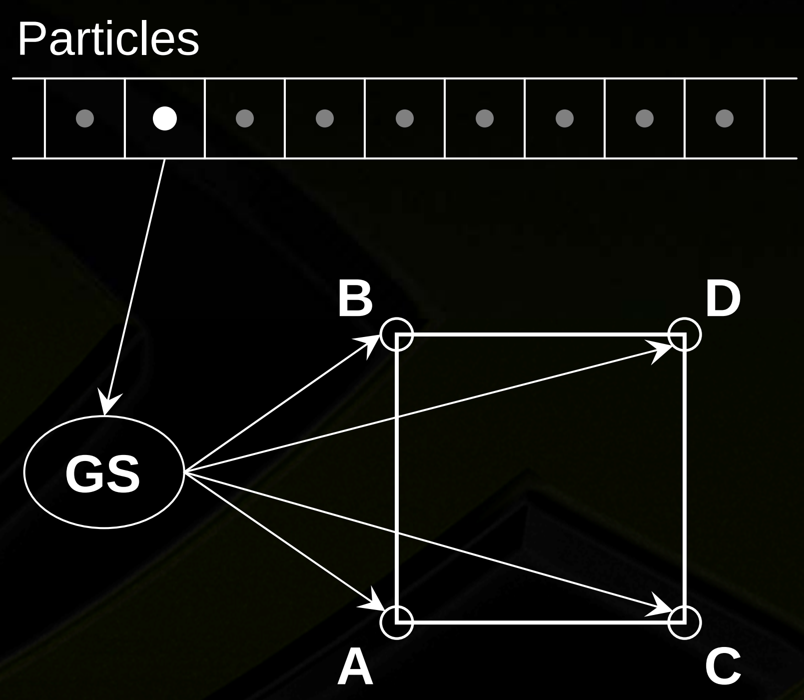
Drawing the Water

- Water particles drawn using quads
- Sliding water quads are parallel to rock
- Falling water and mist face the screen
  - Smooth transition between sliding and falling



Sliding to Falling Transition

Billboarding: Obvious Approach

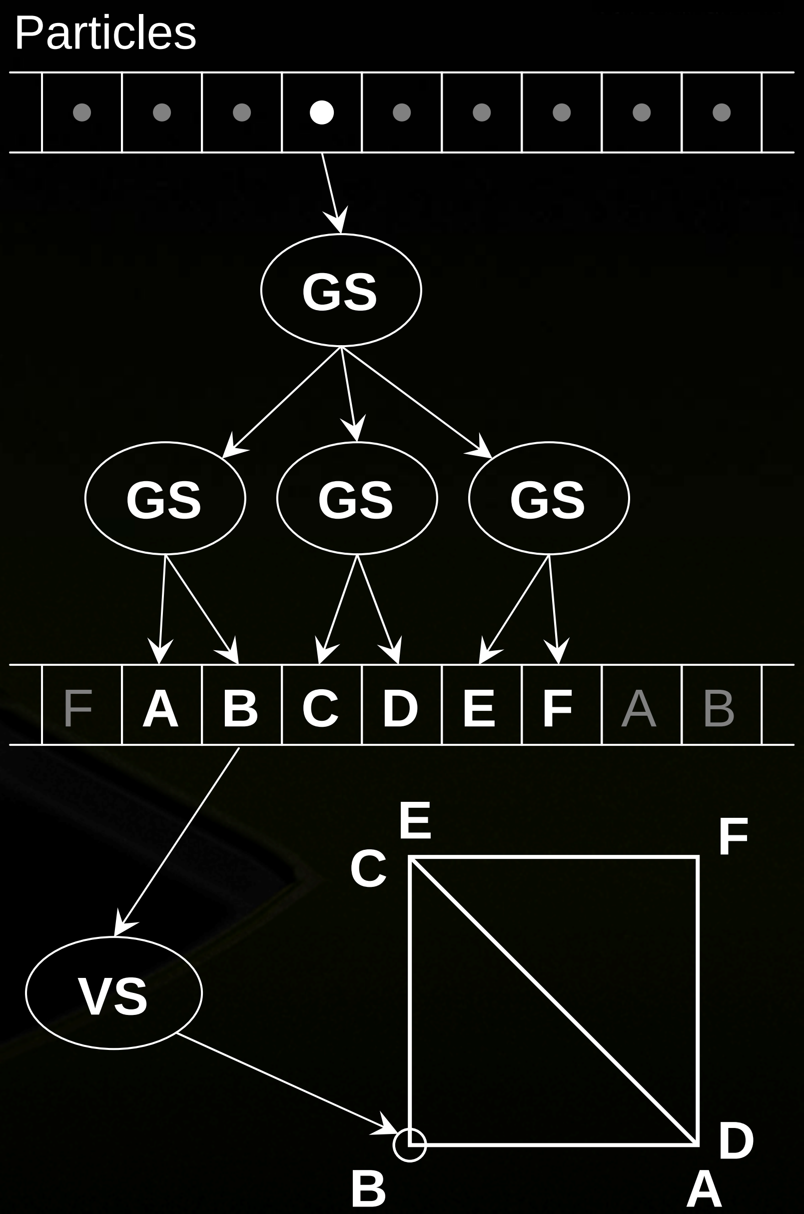


GS Performance

- GS performance improves when output size is small (either few vertices, or few attributes per vertex)
  - These vertices have many attributes used for shading
  - 25 floats per vertex * 4 vertices = 100 outputted
- In general, it's better to spread heavy workloads over many threads to ensure maximum parallelism
  - Calculating these positions is not trivial
    - Different particle types
    - Smooth transitions

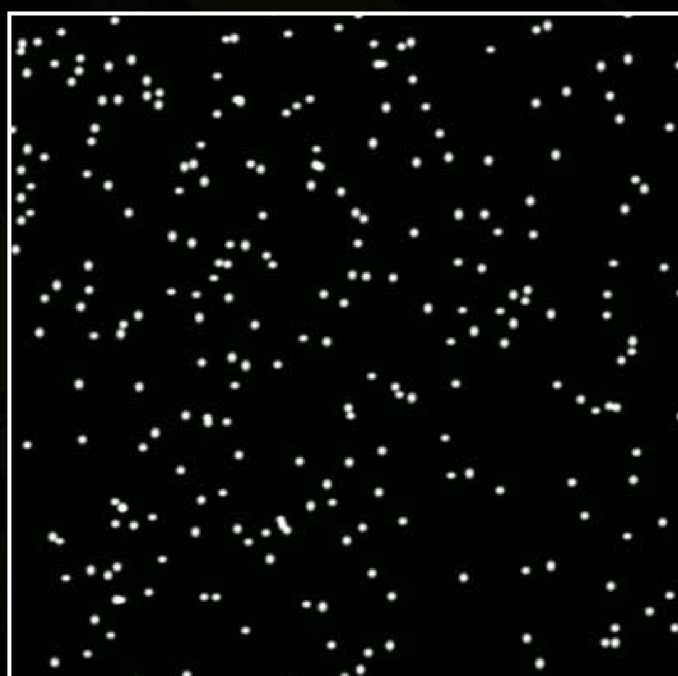
A Faster Way

- Each particle is duplicated 6 times (enough vertices for two triangles) by the GS
  - 12 floats per particle * 3 = 36 outputted (max)
- In the VS, `SV_VertexID%6` is used to index a Constant Buffer
  - 2 floats per vertex for xy offset
  - 2 floats per vertex for texCoord
- The VS moves the vertex to the billboard's corner and assigns its texture coordinate



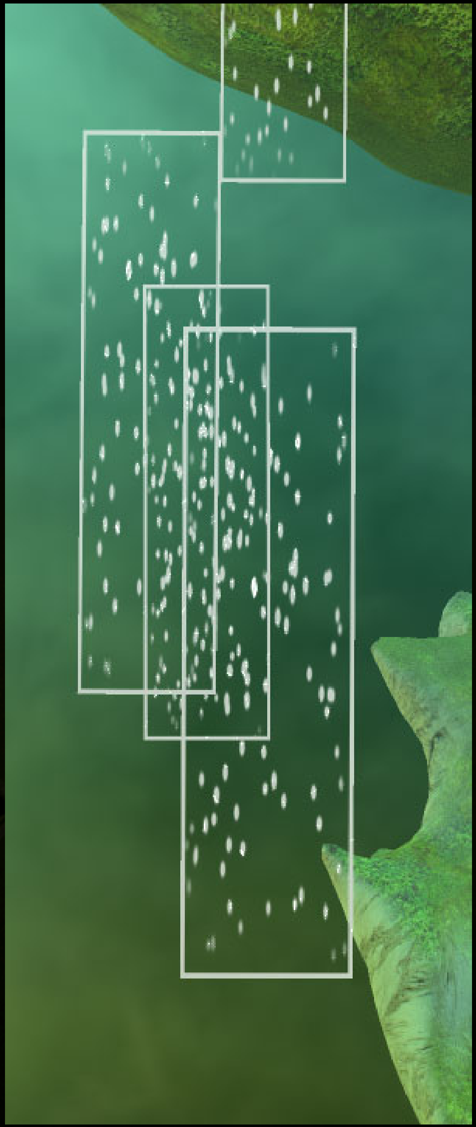
Texturing the Billboards

- Every frame, 256 small water drops are drawn into a small render target
  - The droplets wiggle around independently on a sum of different frequency sine waves



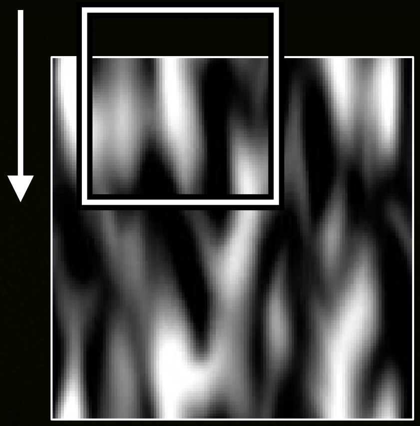
Texturing the Billboards

- Falling water uses a small sub-rectangle of this dynamic “droplets” texture
- Result: Each simulated particle’s billboard looks like many independently-moving water droplets.
  - Even though they all use the same texture, every billboard looks different, because of their unique sub-rectangle



Texturing the Billboards

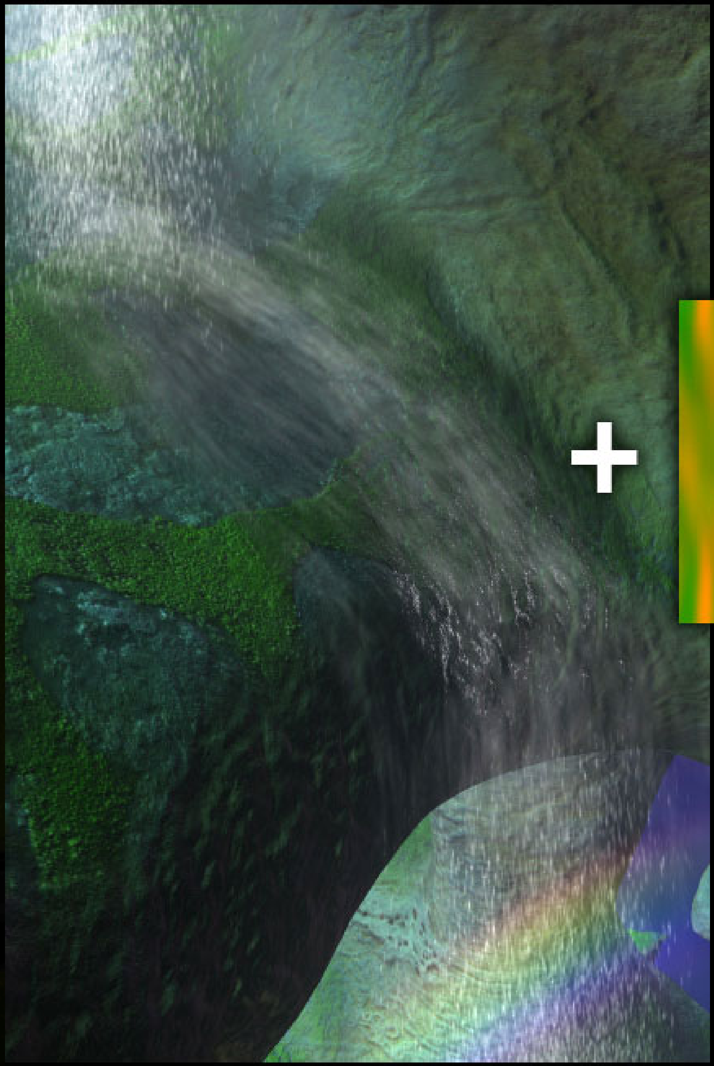
- Sliding water uses a moving window over a static texture
  - Texture wraps seamlessly
- X coordinates within the texture differ between particles
- Y coordinates constantly slide upwards over time
  - Features of the texture appear to be flowing faster than the particle is actually moving
  - Makes it harder to identify individual quads with your eye



Specular Highlights

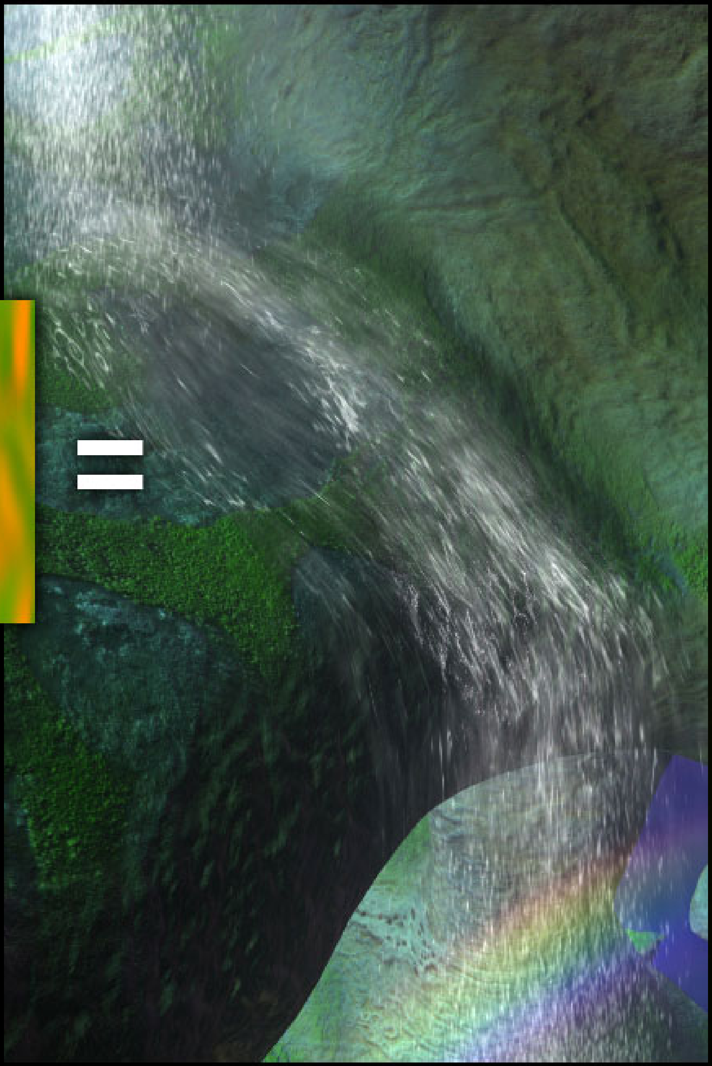
- Normal vector needed
- Sliding water is parallel to the rock
  - Surface normal of the rock is modified by a bump map
- Falling water quads all face the screen; No normal
  - Make it up!
  - Use any old normal map to compute spec
  - Mask it with the droplets texture as a spec map

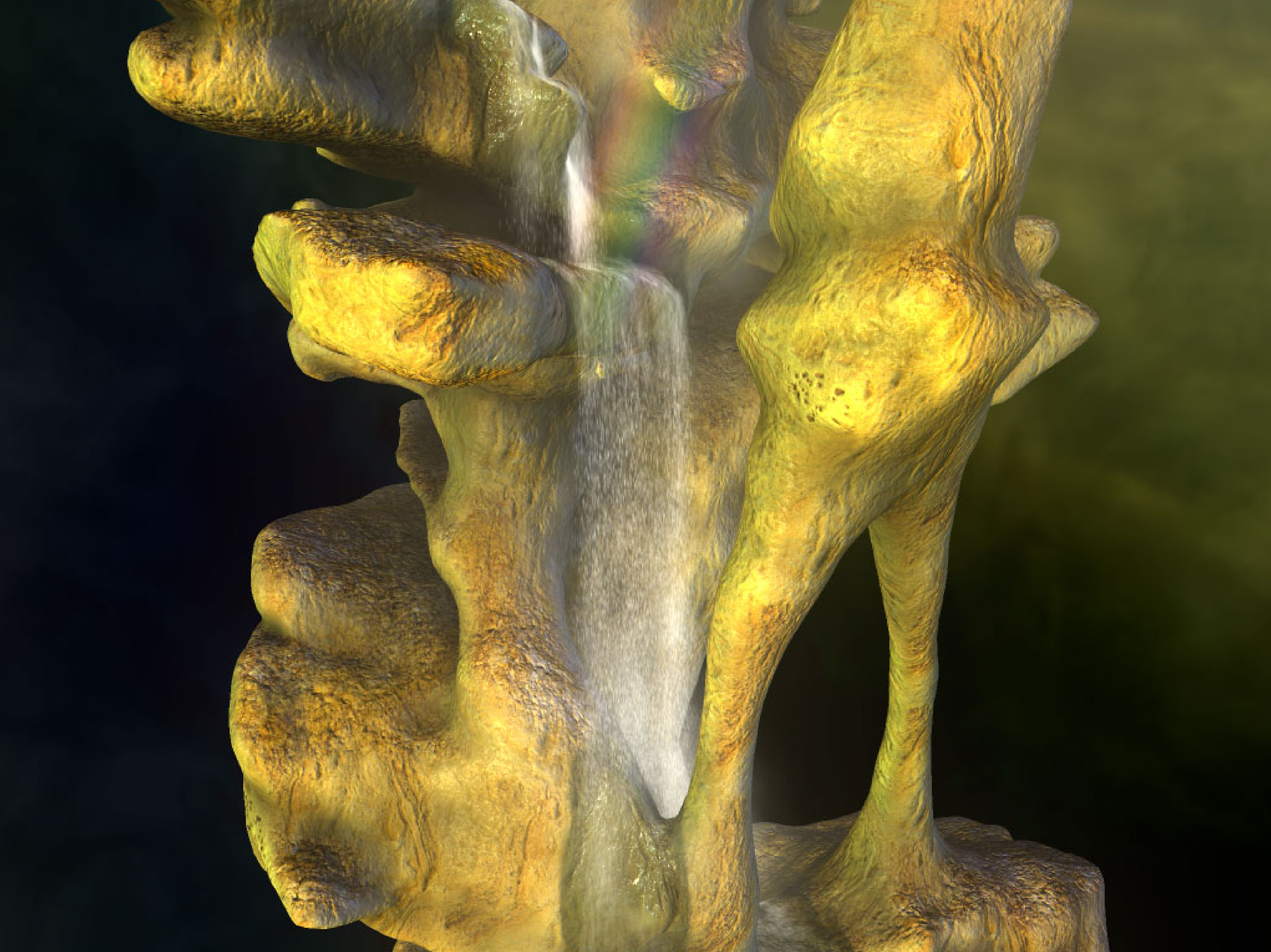
Sliding Spec

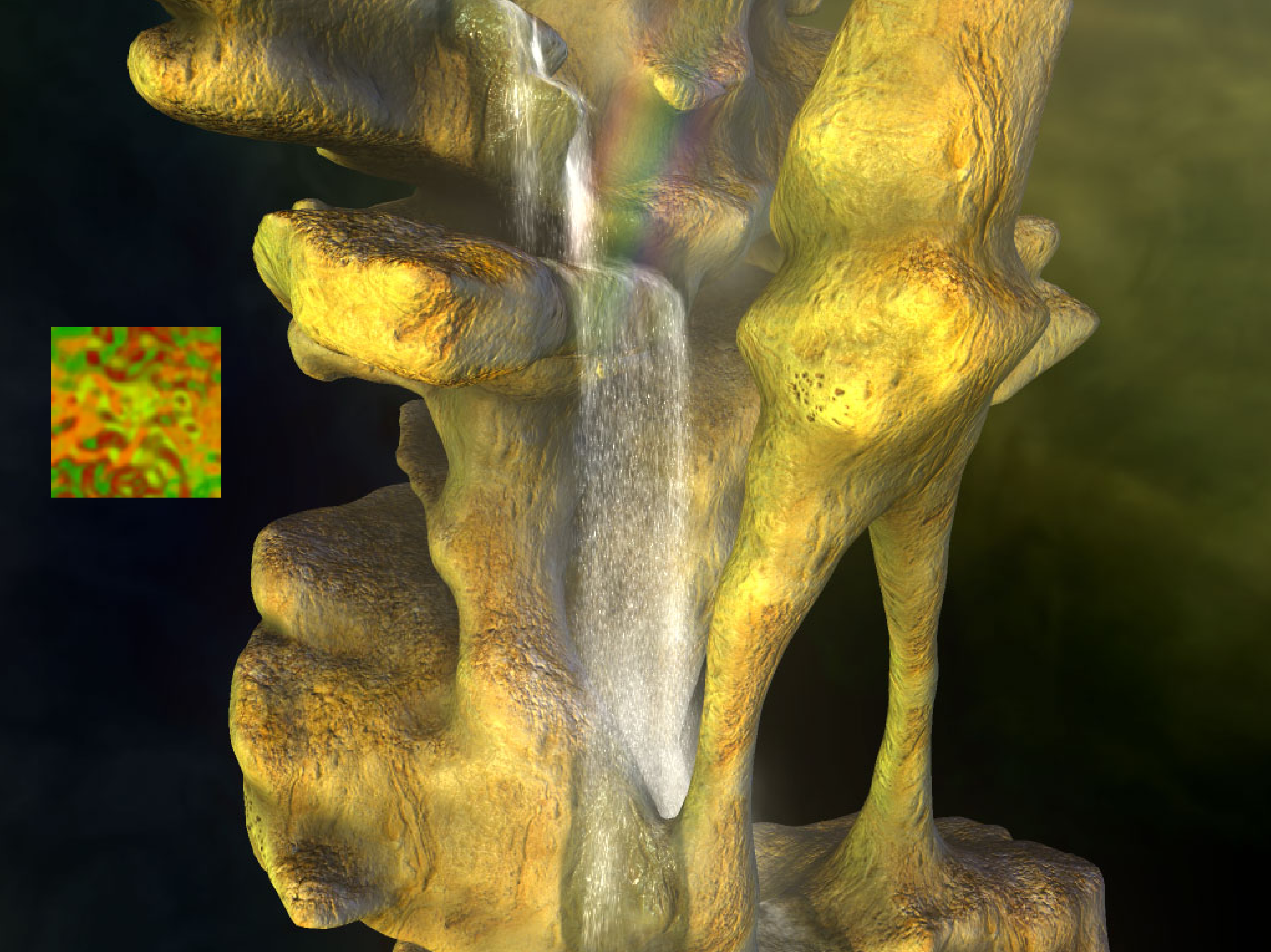


+

=





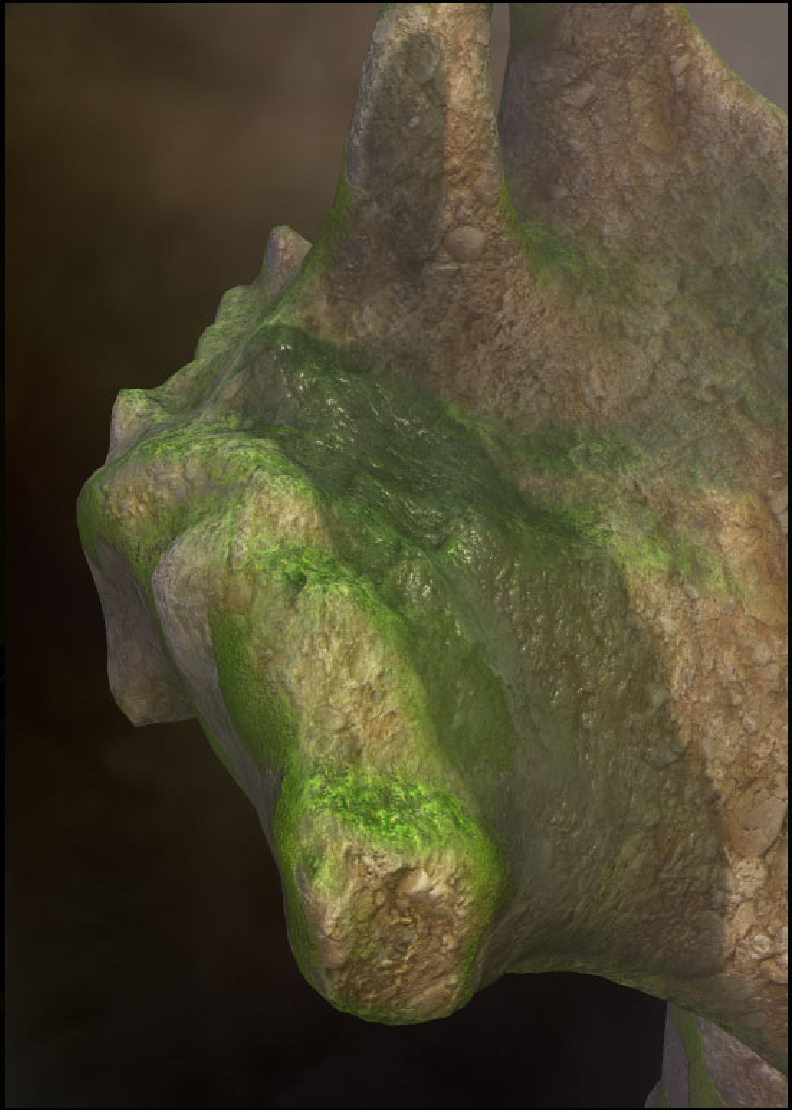


Wet Rock

- Water particles render themselves as points to a 3D “wetness” texture
- Additive blending sums many particles’ wetness contributions
- Values sampled from the wetness texture are used to shade the rock

Wet Rock Drying

- Each frame, large quads are drawn to each slice of the 3D wetness texture
- Subtractive blending reduces wetness
- An 8 bit UNORM DXGI texture format offers free clamping of values to [0,1]
  - Floats would require double-buffering with blending and clamping computed by a shader



Introducing Variation

- Every particle has a unique, fixed number that influences:
  - Movement (speed and direction)
  - Likelihood of turning to mist
  - Size of billboard
  - Texture coordinates for drawing
- Shaders need a random number generator
  - Update a seed in a CB from the application every frame
  - Multiply it by the Vertex_ID before using it

Random Numbers

```
cbuffer RandomCB {
   float randomSeed;
}

void seedRandomNumberGenerator(const float seed) {
   // randomSeed is changed by the app
   // at the beginning of every frame
   randomSeed *= frac(3.14159265358979 * seed);
}

float urand() {
   randomSeed = (((7271.2651132 * (randomSeed +
   0.12345678945687)) % 671.0) + 1.0) / 671.0;
   return randomSeed;
}
```


Flocking

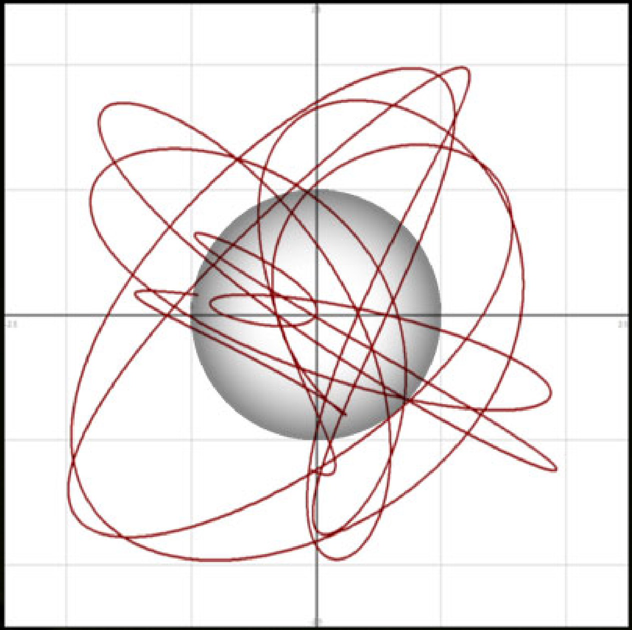
Dragonflies

- Behavior is calculated on the GPU
  - Including collision avoidance
- Each dragonfly is stored as a vertex
- Vertex Buffer is double-buffered
- Shader updates a dragonfly’s vertex
- Results are Streamed Out to other VB

(Just like the water particles!)

Where Are They Going?

- Two invisible, moving attractors are updated by the application every frame, stored in a CB
  - Attractors move on a sum of sine waves
  - In the shader, each dragonfly is drawn to the closer of the two attractors
  - This is what makes the dragonflies move together as a flock (or two flocks)



- Random up and down wandering
  - Each dragonfly has a different frequency for a sine wave

Look Out For The Rock

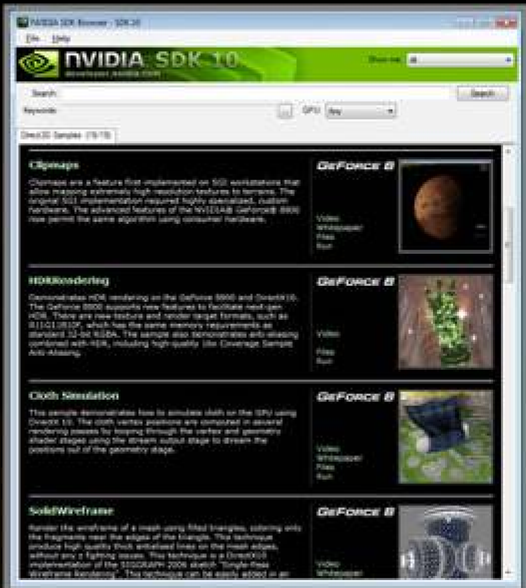
- **Dragonflies are able to sample the Rock 3D texture to avoid flying into the rock**
  - **The shader tests several random directions ahead of the dragonfly for the existence of rock**
  - **Much stronger influence than the attractors, to allow sharper turns**

Drawing the Dragonflies

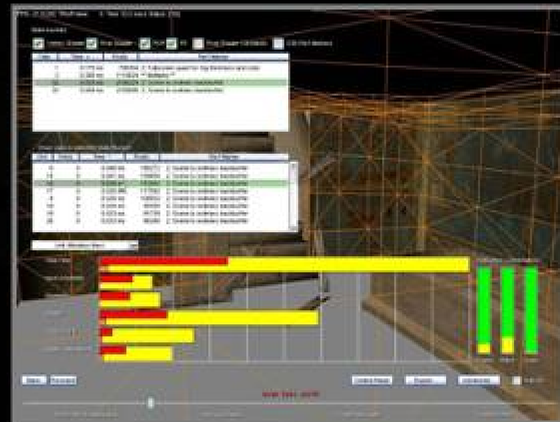
- Three different models for the dragonfly (LOD)
- Positions and velocities are read back to the CPU
  - But double-buffered to avoid a stall
- Distance from camera determines the LOD for each
- Three Instanced draw calls are made, to draw all LODs


The End.

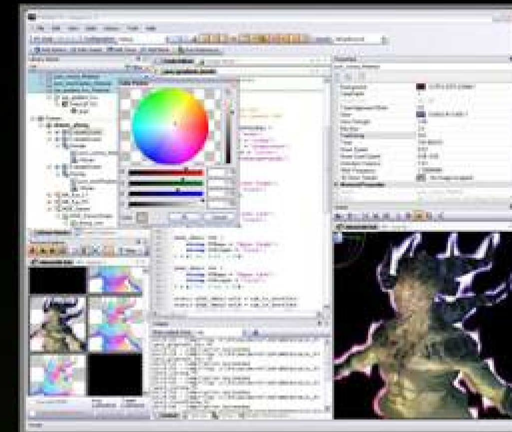
New Developer Tools at GDC 02007



SDK 10



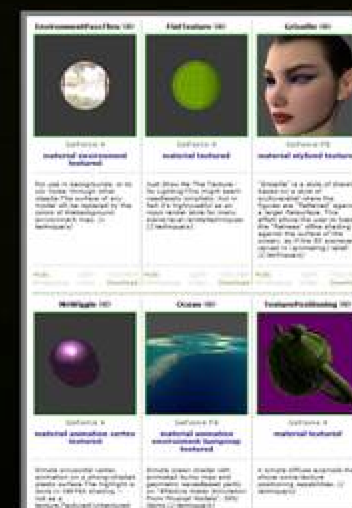
PerfKit 5



FX Composer 2

GPU-Accelerated Texture Tools

ShaderPerf 2



Shader Library