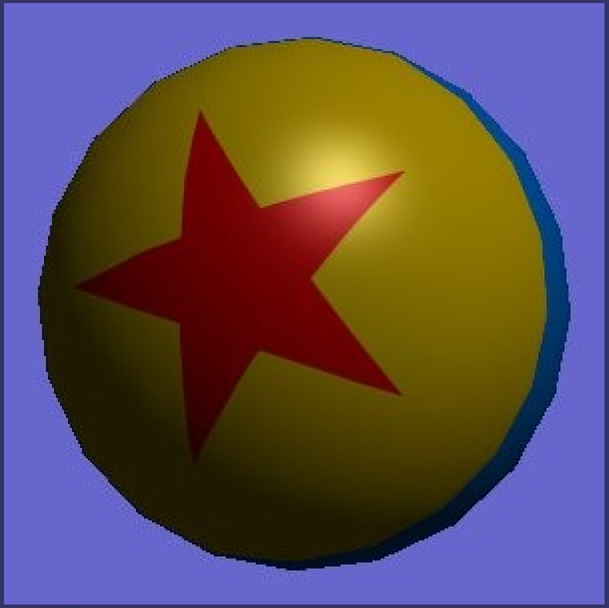
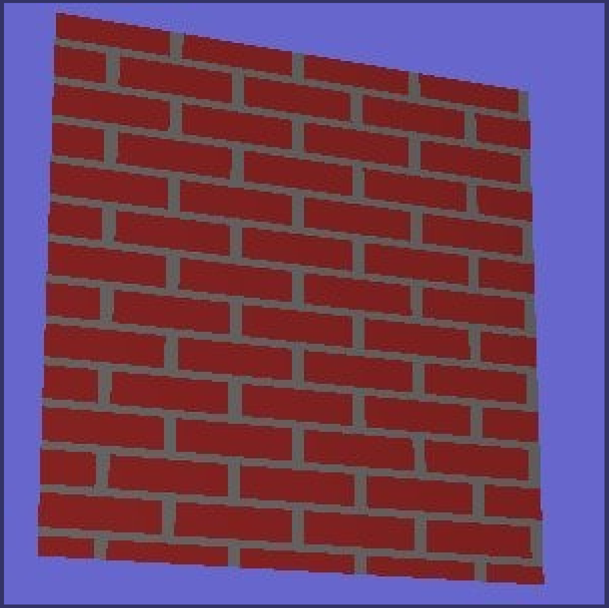
VGP352 – Week 2

- Agenda:
  - Procedural texturing and modeling
    - Rationale
    - Basic techniques / examples
    - Noise
    - Anti-aliasing

Procedural Graphics

- Generation of textures, models, or animation from *code* instead of *data*
  - Creation may happen at rendering-time *or* at application load-time





Procedural Graphics

- Why?
  - Less space!
  - Easier to add “random” variation
  - May be easier to describe than to draw
    - L-systems for trees
    - Fractals for whole worlds
    - etc.

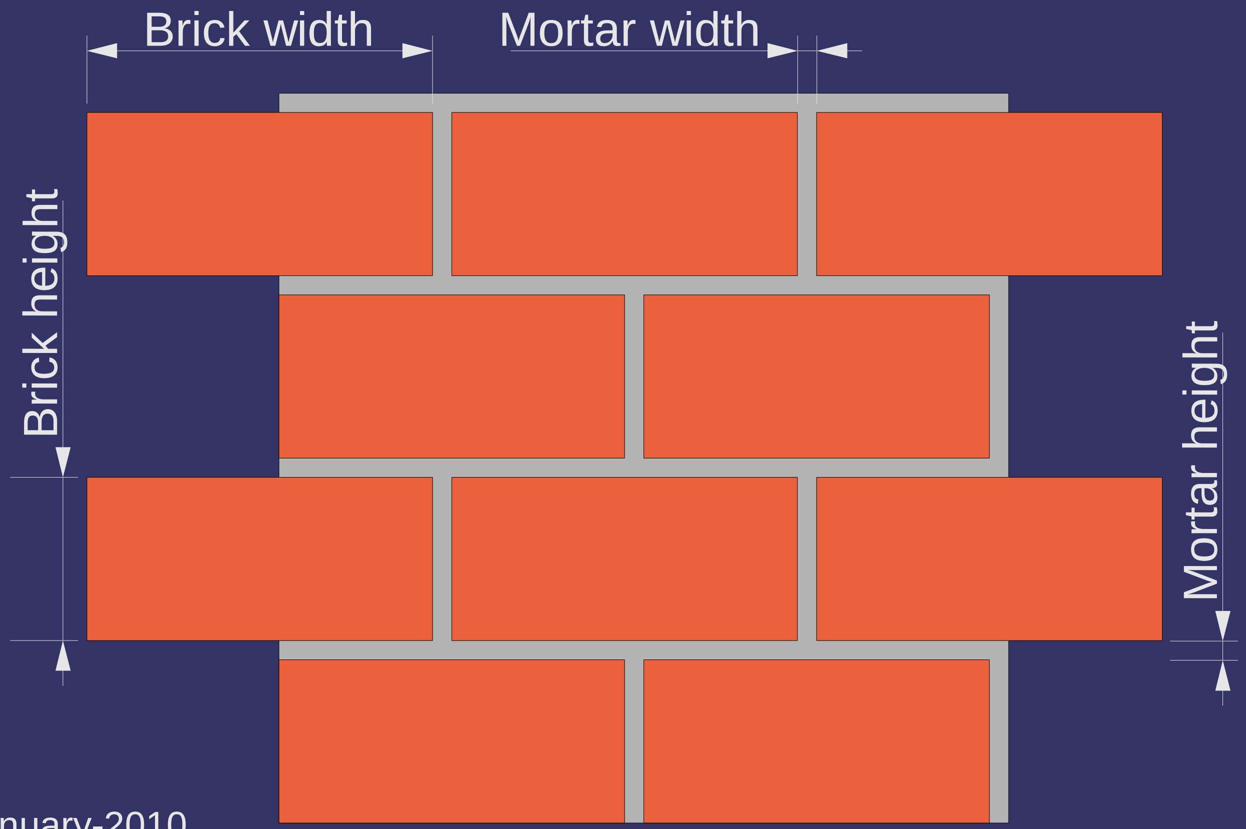
Procedural Graphics

- Example: “Debris” by Farbrausch
  - Entire demo is 181,248 bytes
  - This JPEG image is 166,059 bytes!

  - See http://scene.org/file.php?id=373930 or http://www.youtube.com/watch?v=wqu_IpkOYBg&fmt=22

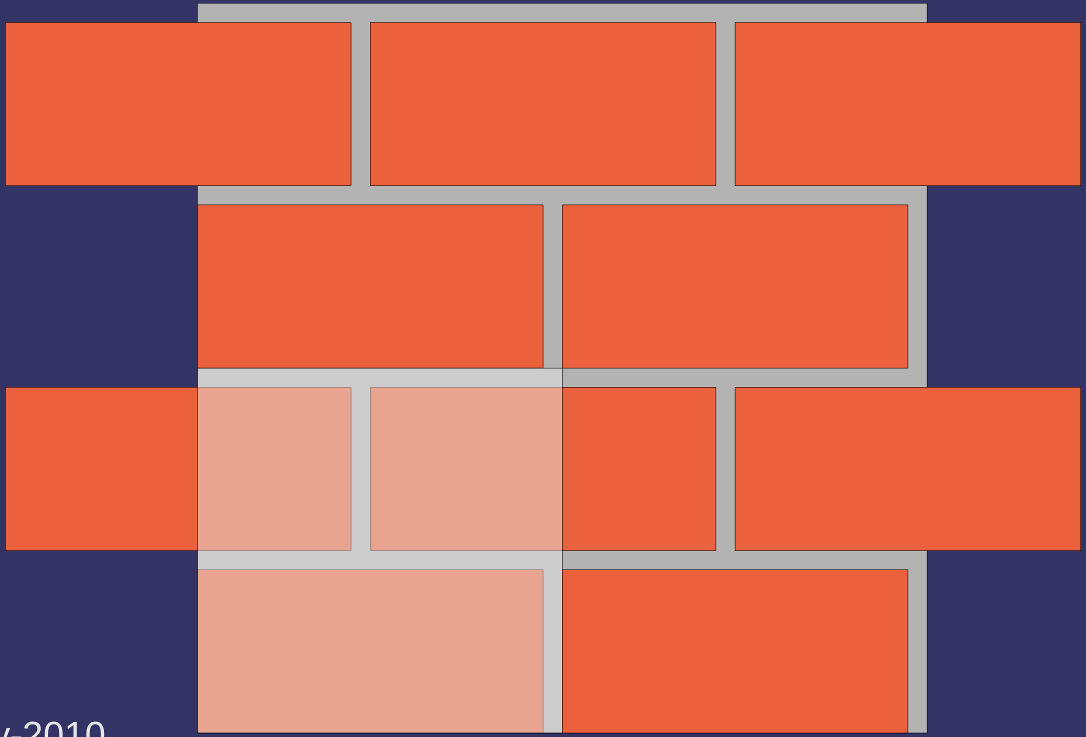
Brick Shader

- Given some parameters, generate an image that looks like bricks



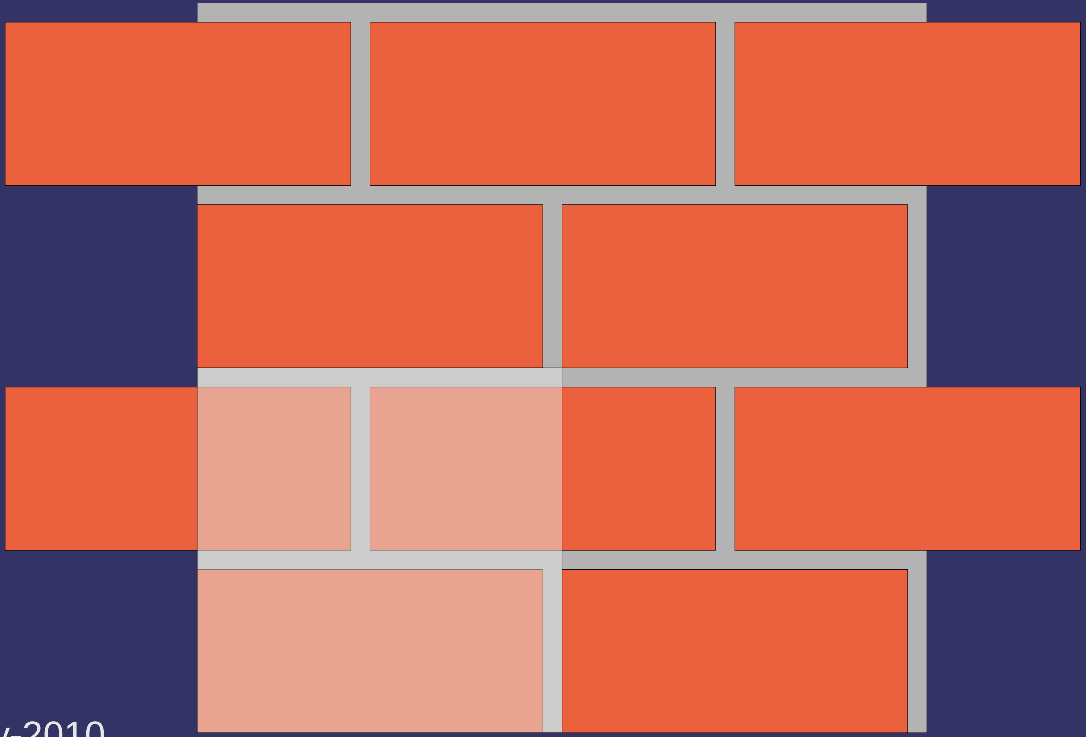
Brick Shader

- Given some parameters, generate an image that looks like bricks
  - Divide *shader-space* into cells
  - Each cell is conceptually a 1×1 unit



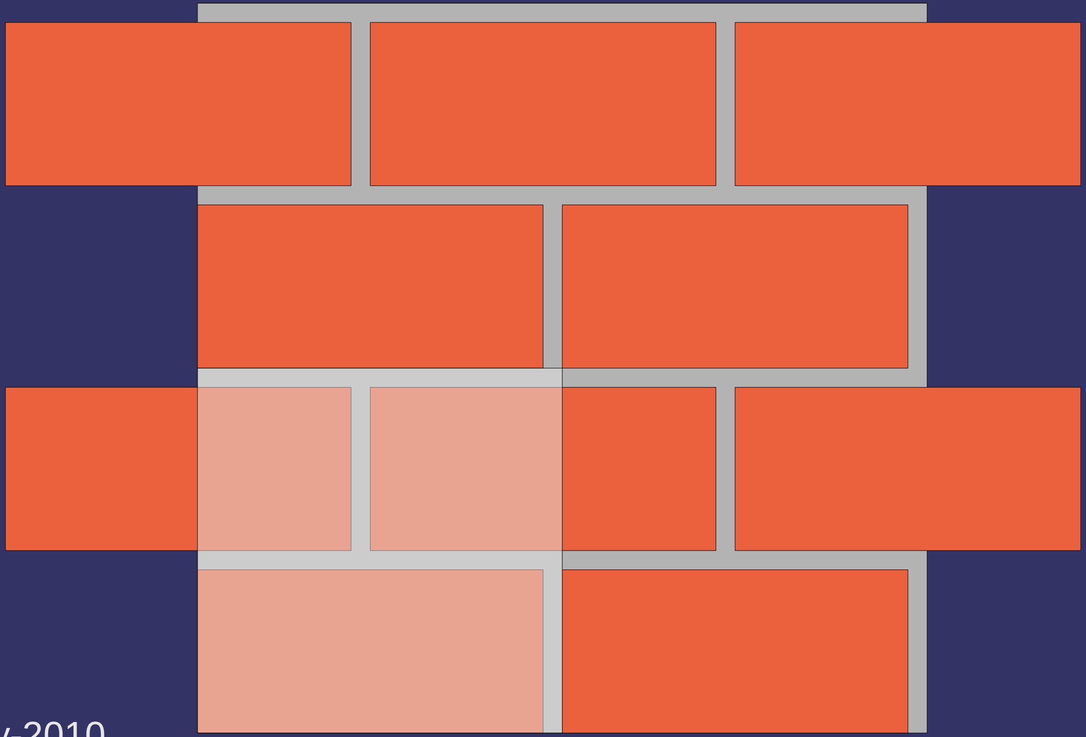
Brick Shader

- Bottom row is easy:
  - If `s` is less than `brick_width / (brick_width + mortar_width)`, the color is brick



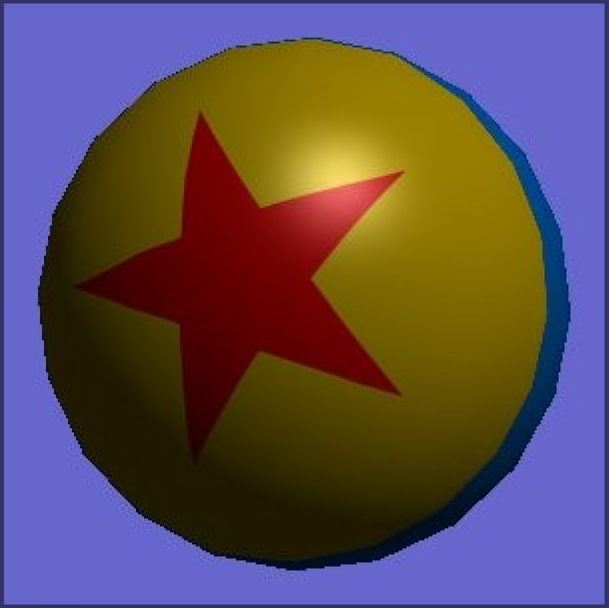
Brick Shader

- Top row is the bottom row with an offset
  - If `t` is greater than `brick_height / (brick_height + mortar_height)`, add 0.5 to `s`



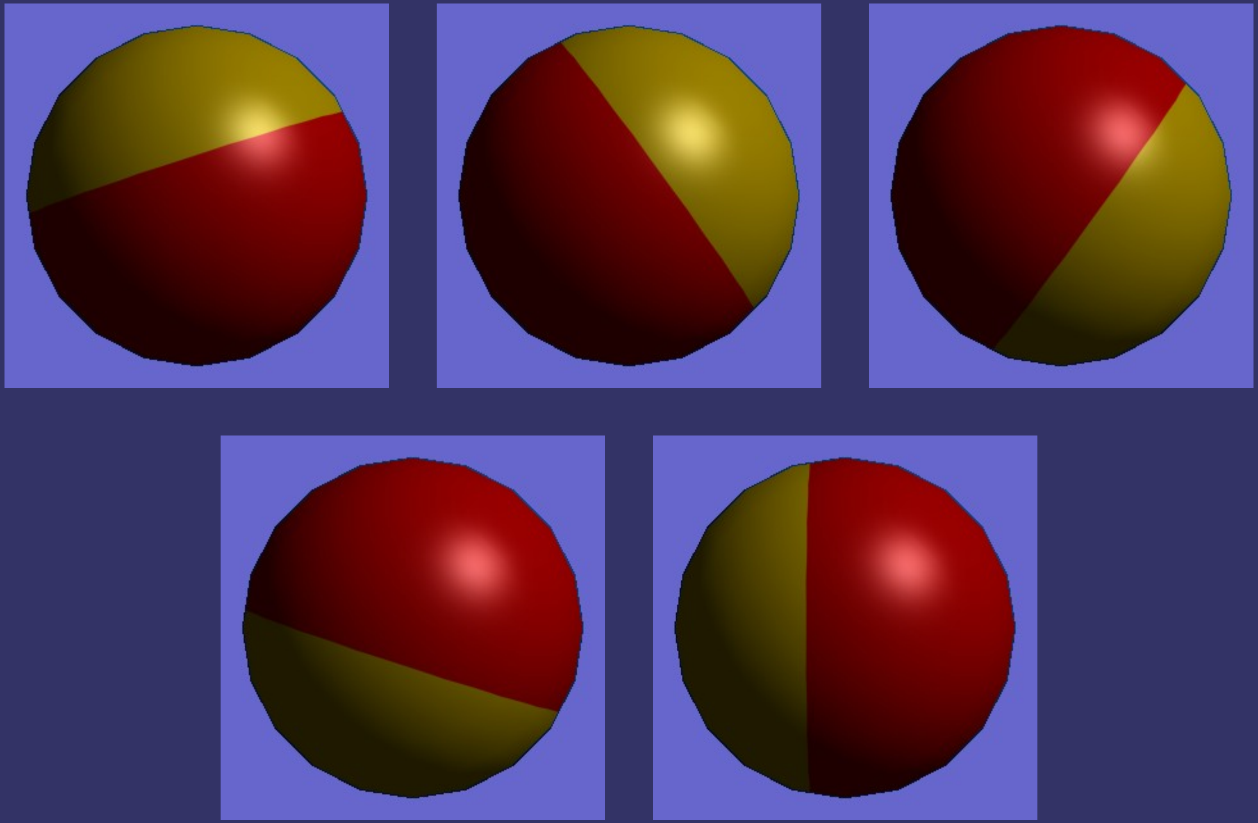
Toy Ball

- Texture consists of a complex shape
  - Can't use simple compares to determine which region a point is in
  - All of the boundaries are straight lines



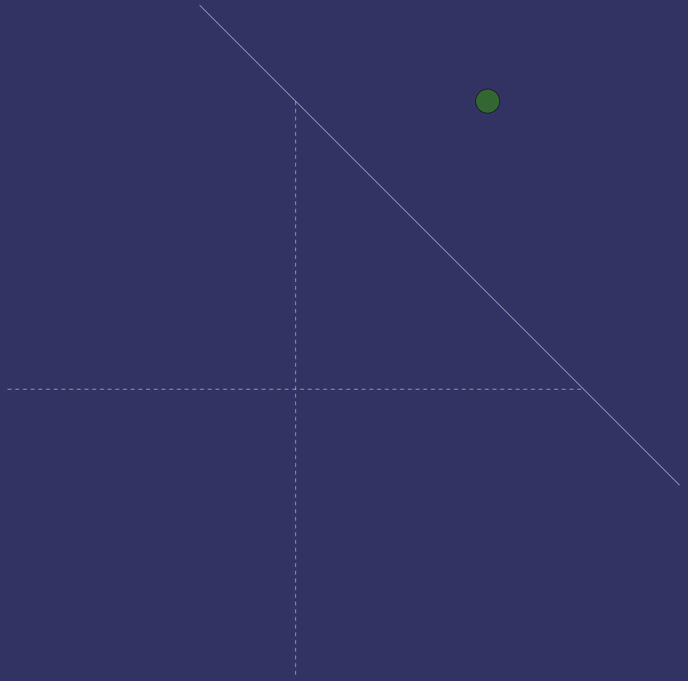
Toy Ball

- Divide shader space into regions called *half spaces*



Toy Ball

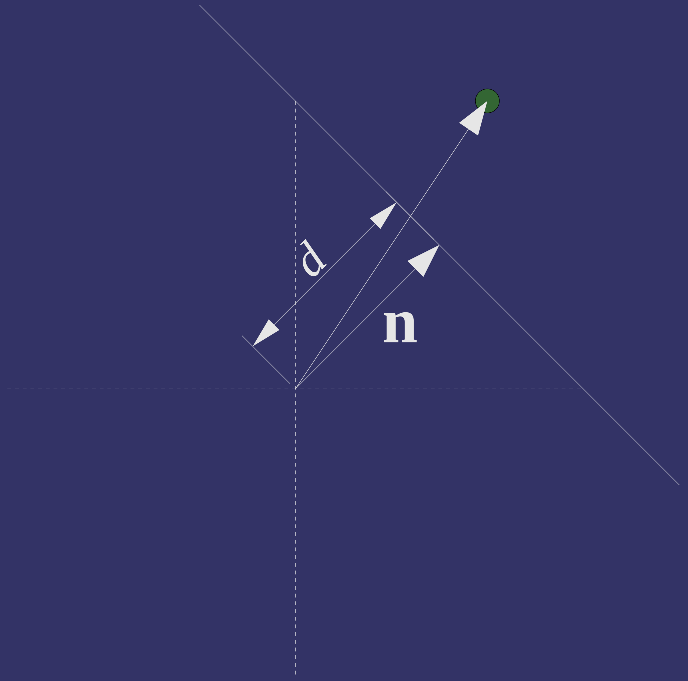
- If we draw a line through 2D space, how do we determine which side of that line a point is on?



Toy Ball

- If we draw a line through 2D space, how do we determine which side of that line a point is on?
  - Use the parametric definition of a line
  - Use *x* and *y* from the point
    - If the result is less than 0, the point is “inside”
    - If the result is equal to 0, the point is on the line
    - If the result is greater than 0, the point is “outside”

$$\mathbf{o} = ax + by - d$$



Toy Ball

- What does this look like?

$$a x + b y - d$$

Toy Ball

- What does this look like?

$$a x + b y - d$$

- Our friend, the dot-product:

$$\begin{bmatrix} a & b & -d \end{bmatrix} \cdot \begin{bmatrix} x & y & \mathbf{q} \end{bmatrix}$$

Toy Ball

- We want a binary answer whether the point is inside or outside

```
dist = dot(p, half_space);
in_or_out = (dist < 0.0) ? 0.0 : 1.0;
```

  - A more succinct way in GLSL uses the `step` function:

```
dist = dot(p, half_space);
in_or_out = step(0.0, dist);
```

Toy Ball

- We want a binary answer whether the point is inside or outside of all 5 half-spaces

```
dist.x = dot(p, half_space0);
dist.y = dot(p, half_space1);
dist.z = dot(p, half_space2);
dist.w = dot(p, half_space3);

dist.x = step(dot(dist, vec4(1.0))) +
    step(0.0, dot(p, half_space4));

in_or_out = dist.x > 4.0;
color = mix(ball_color, star_color, in_or_out);
```

References

http://www.wired.com/gaming/gamingreviews/magazine/16-08/pl_games

http://people.freedesktop.org/~idr/GLSL_presentation/GLSL-Portland-Bill.PPT

Wang Tiles

- Goal: we want to create an infinite, non-repeating texture for things like grass, sand, etc.

Wang Tiles

- Goal: we want to create an infinite, non-repeating texture for things like grass, sand, etc.
  - Even a 2048x2048 texture will show tiling artifacts
  - *And* it will use 16MB of texture memory...yuck!

Wang Tiles

- Goal: we want to create an infinite, non-repeating texture for things like grass, sand, etc.
  - Even a 2048x2048 texture will show tiling artifacts
  - *And* it will use 16MB of texture memory...yuck!
- Create a “mosaic” from small a few small “tiles”

Wang Tiles

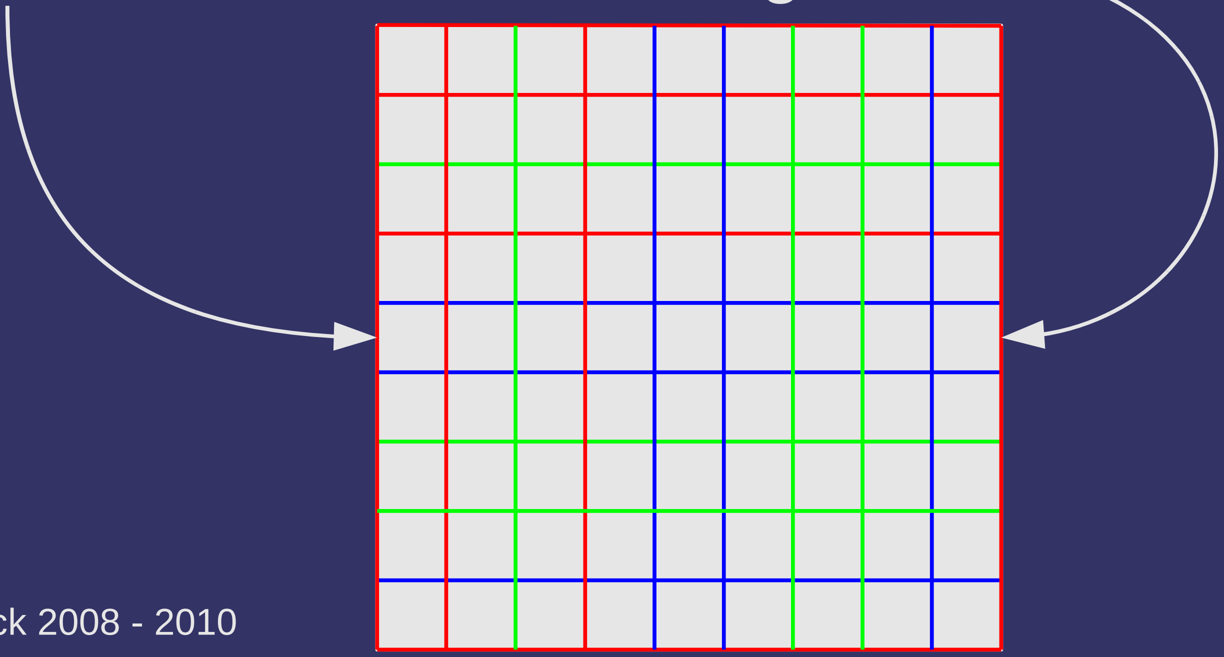
- Goal: we want to create an infinite, non-repeating texture for things like grass, sand, etc.
  - Even a 2048x2048 texture will show tiling artifacts
  - *And* it will use 16MB of texture memory...yuck!
- Create a “mosaic” from small a few small “tiles”
  - If the tile selection is pseudo-random, as few as 32 tiles can have a *very* large repeat period
  - Unlike mosaic tiles, texture tiles have to match at the edges
    - Either all tiles edges have to match or the selection algorithm has to pick a tile that will match edges with its neighbors

Wang Tiles – Edge Coloring

- Name the four tile edges N, E, S, W
  - The N/S edges can have one of K_v edge “colors”
  - The E/W edges can have one of K_h edge “colors”
    - A tile with an N edge of color X must be south of a tile with an S edge of color X
  - A tile with each possible combination of edge colors must exist
    - There must be at least $K_v^2 \times K_h^2$ tiles

Wang Tiles – Tile Arrangement

- Assuming we have a set of tiles...
  - Generating tiles from a sample source image is a larger topic than we have time for
- Arrange tiles in a $K_v^2 \times K_h^2$ pattern in texture atlas
  - Neighboring tiles must obey edge coloring rules...even neighbors across border edges!



Wang Tiles – Tile Arrangement

- Given a pair of edge colors, the following placement algorithm is use:

$$Index(e_1, e_2) = \begin{cases} 0 & if\ e_1 = e_2 = 0 \\ e_1^2 + 2e_2 - 1 & if\ e_1 > e_2 > 0 \\ e_2^2 + 2e_1 & if\ e_2 > e_1 \geq 0 \\ (e_2 + 1)^2 - 2 & if\ e_1 = e_2 > 0 \\ (e_1 + 1)^2 - 1 & if\ e_1 > e_2 = 0 \end{cases}$$

Wang Tiles – Tile Selection

- Given texture coordinate (s, t):
  - Calculate tile index
    - $O_h = t / T_h$
    - $O_v = s / T_v$
  - Hash tile index to calculate edge colors
    - $C_s = H(H(O_h) + O_v) \% K_v$
    - $C_n = H(H(O_h) + O_v + 1) \% K_v$
    - $C_w = H(O_h + H(O_v * 2)) \% K_h$
    - $C_e = H(O_h + 1 + H(O_v * 2)) \% K_h$
    - Notice that $C_e(x, y) = C_w(x + 1, y)$
  - Convert edge colors to row / column indexes

Wang Tiles – Tile Selection

- Given texture coordinate (s, t):
  - Calculate row / column position in texture
    - $t_{base} = I_h * T_h$
    - $s_{base} = I_v * T_v$
  - Calculate texel offset within tile
    - $t_{offset} = t \% T_h$
    - $s_{offset} = s \% T_v$
  - Sample the texture!
    - Final coordinate is ($s_{base} + s_{offset}$, $t_{base} + t_{offset}$)

Wang Tiles – Hash Function

- Implement as a permutation table
  - Use a texture rectangle that is 1 texel tall
    - Use roughly 4x entries in table as possible edge colors
  - More recent hardware can use uniform arrays
    - Geforce 6 or Radeon X800

Wang Tiles – Filtering Gotchas

- Mipmap filtering can be a problem...
  - The 1x1 level blends all of the tiles together...bad!!!
    - Need to clamp the minimum LOD to the level lowest level that doesn't blur across tile boundaries
    - The tile map is just a big texture atlas
  - This is *much* easier with texture arrays

References

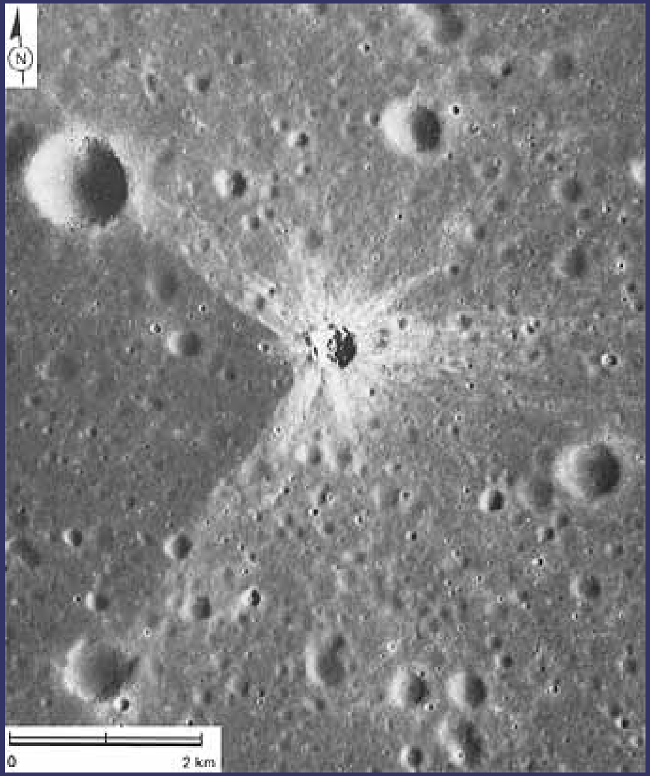
http://en.wikipedia.org/wiki/Wang_tile

Wei, L. “Tile-based texture mapping on graphics hardware.” In *ACM SIGGRAPH 2004 Sketches* (Los Angeles, California, August 08 - 12, 2004). R. Barzel, Ed. SIGGRAPH '04. ACM, New York, NY, 67. http://graphics.stanford.edu/papers/tile_mapping_gh2004/

Wei, L. “Tile-Based Texture Mapping.” In GPU Gems 2. Ed. Matt Pharr. Upper Saddle River, NJ: Pearson Education, Inc., April 2005. http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter12.html

Crater Shader

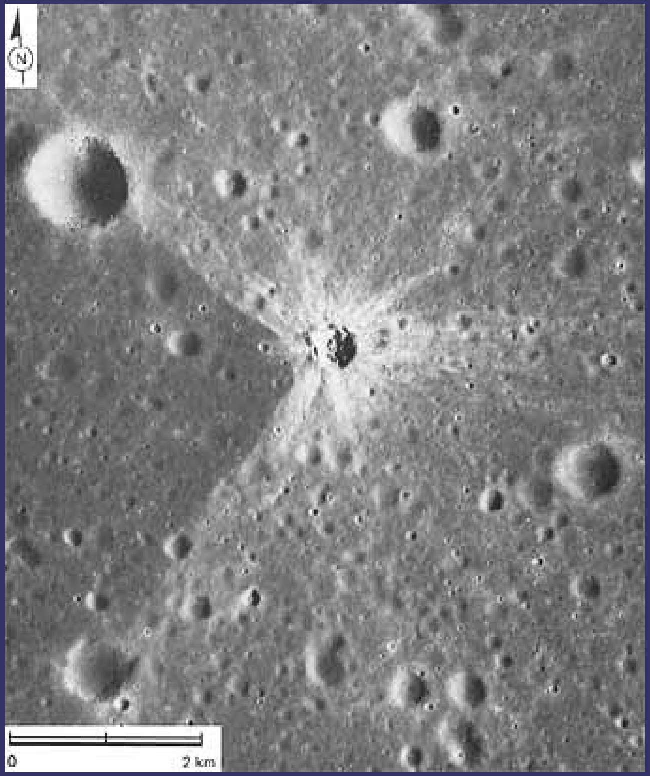
- Task: create a procedural texture for impact craters on, for example, the moon



Original image from http://www.hq.nasa.gov/office/pao/History/SP-362/ch5.2.htm

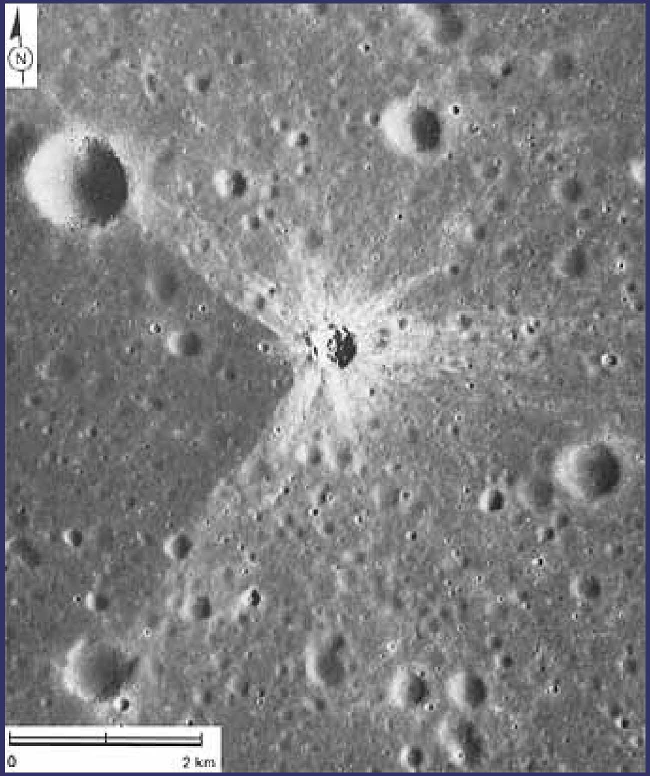
Crater Shader

- Two parts to this shader



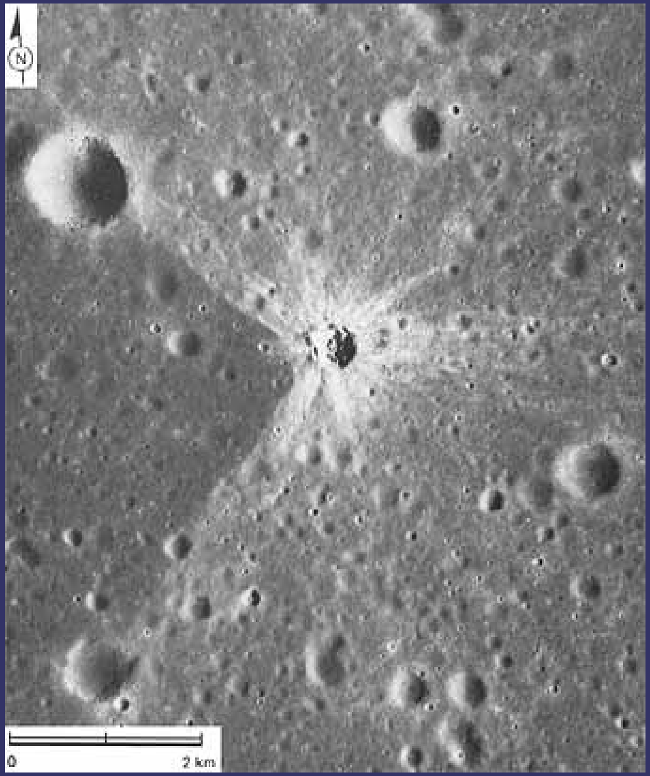
Crater Shader

- Two parts to this shader
  - Height / normal
  - Color



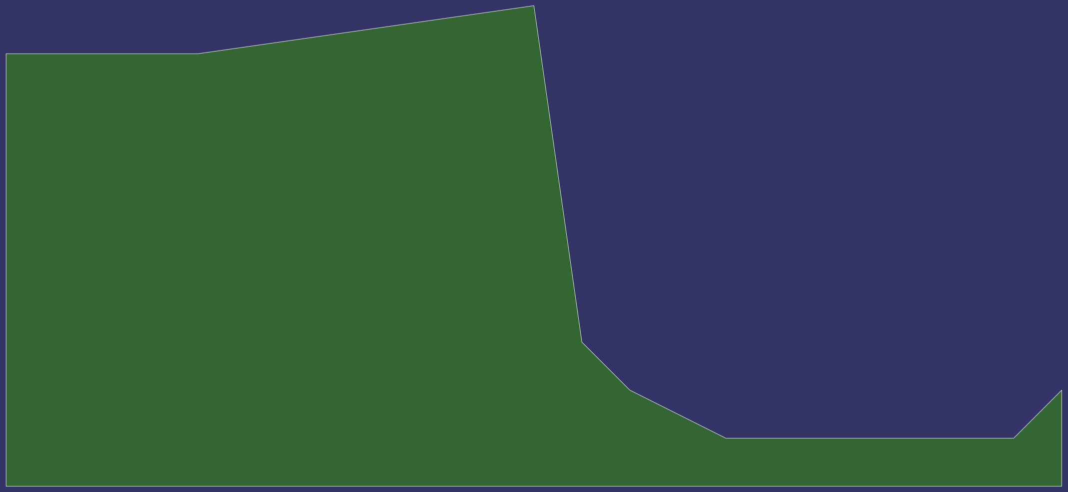
Crater Shader

- Two parts to this shader
  - Height / normal
  - Color
  - Attack each separately, then try to unify



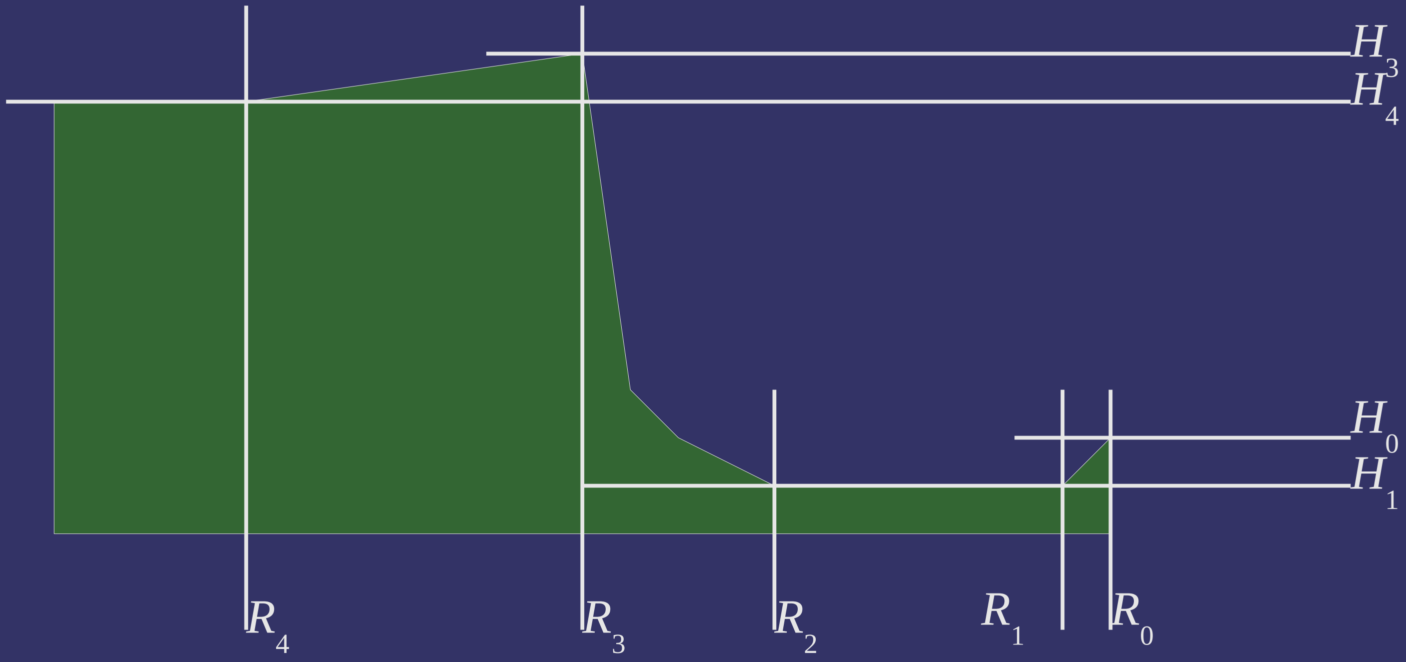
Crater Shader – Height

- Craters are generally circular
  - Height varies with distance from center



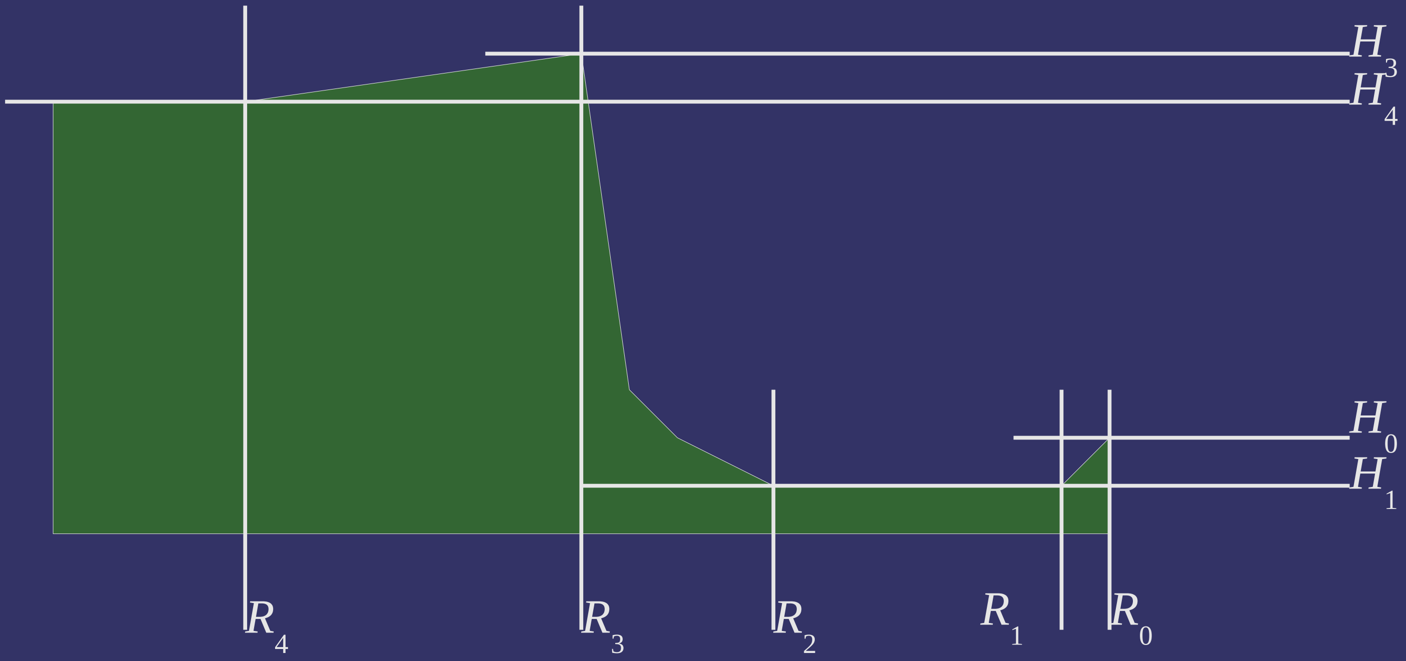
Crater Shader – Height

- Craters are generally circular
  - Height varies with distance from center
  - Associate a height with each distance where there is a change



Crater Shader – Height

- Select an interpolation scheme between each region
  - R_0 to R_1 and R_1 to R_2 could be linear, R_2 to R_3 and R_3 to R_4 could be exponential, etc.



Crater Shader – Height

- In shader:
  - Determine fragment distance from center

```
r = length(position – center);
```

Crater Shader – Height

- In shader:
  - Determine fragment distance from center

```
r = length(position – center);
```

  - Determine which region contains the fragment

```
if (r < crater_param[1].x) {
    ...
} else if (r < crater_param[2].x) {
    ...
} else ...
```

Crater Shader – Height

- In shader:
  - Determine fragment distance from center

```
r = length(position – center);
```

  - Determine which region contains the fragment

```
if (r < crater_param[1].x) {
    ...
} else if (r < crater_param[2].x) {
    ...
} else ...
```

  - Determine fragment location in region

```
t = (r – crater_param[n].x)
 / (crater_param[n+1].x - crater_param[n].x);
```

Crater Shader – Height

- In shader:
  - Determine fragment distance from center

```
r = length(position – center);
```

  - Determine which region contains the fragment

```
if (r < crater_param[1].x) {
    ...
} else if (r < crater_param[2].x) {
    ...
} else ...
```

  - Determine fragment location in region

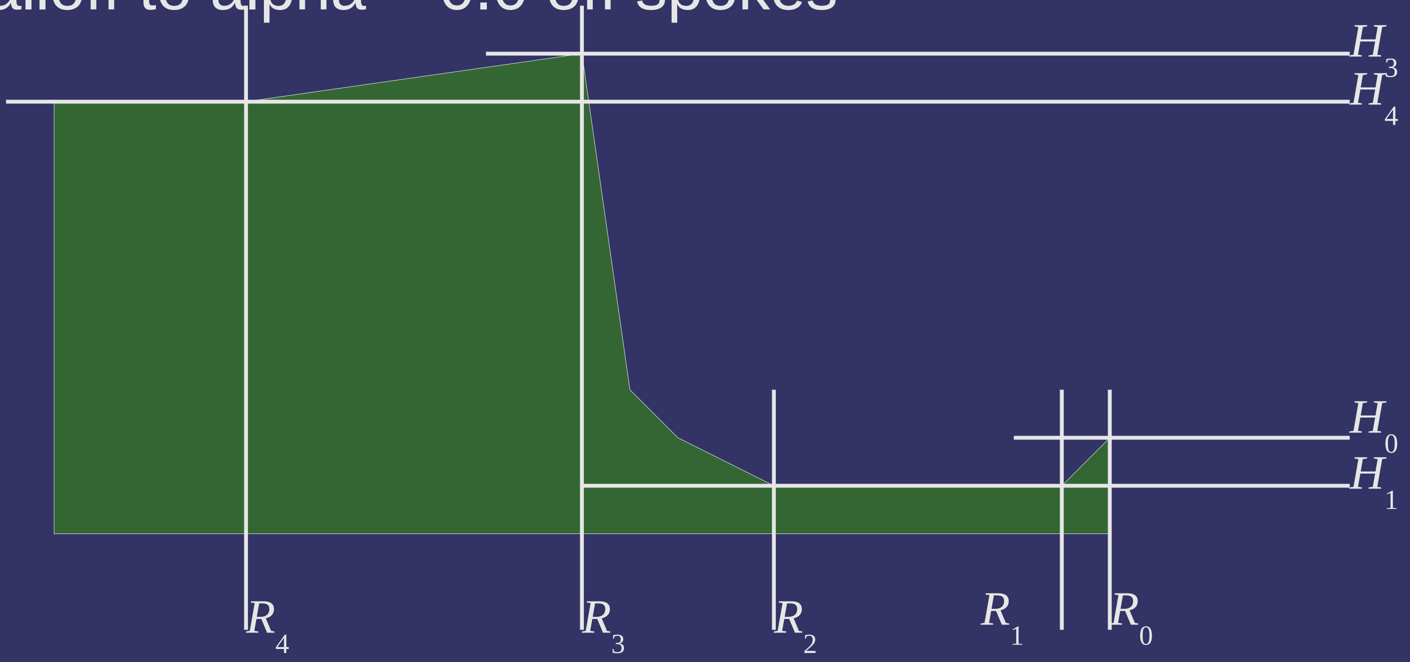
```
t = (r – crater_param[n].x)
  / (crater_param[n+1].x - crater_param[n].x);
```

  - Perform interpolation

```
h = mix(crater_param[n+1].y, crater_param[n].y, t);
```

Crater Shader – Color

- Color works in a similar manner
  - Use one color inside the crater with alpha set to 1.0
  - Use another color outside the crater
    - Set alpha to 1.0 in “spokes” from crater
    - Falloff to alpha = 0.0 off spokes



Crater Shader – Color

- Selecting crater interior color is trivial
  - If r is less than R_3, use interior color

Crater Shader – Color

- Selecting crater interior color is trivial
    - If r is less than R_3, use interior color
- Selecting spoke color is more complex

Crater Shader – Color

- Selecting crater interior color is trivial
  - If r is less than R_3, use interior color
- Selecting spoke color is more complex
  - Need to know distance from center *and* angle (i.e., polar coordinates)

Crater Shader – Color

- Selecting crater interior color is trivial
  - If r is less than R_3, use interior color
- Selecting spoke color is more complex
  - Need to know distance from center *and* angle (i.e., polar coordinates)
  - Place spokes separated by fixed angles
    - Spokes are determined by a cosine wave in polar coordinates
    - $r_{spoke} = \cos(\alpha \times frequency)$

Crater Shader – Color

- Selecting crater interior color is trivial
  - If r is less than R_3, use interior color
- Selecting spoke color is more complex
  - Need to know distance from center *and* angle (i.e., polar coordinates)
  - Place spokes separated by fixed angles
    - Spokes are determined by a cosine wave in polar coordinates
    - $r_{spoke} = \cos(\alpha \times frequency)$
  - Select random length and thickness for each spoke
    - Noise to the rescue
    - Thickness is determined by raising $(r_{spoke} \times amplitude)$ to a power

References

Ebert, David, et. al., *Texturing and Modeling: A Procedural Approach*, second edition, Morgan-Kaufmann, 1998. pp. 315 – 318.

- This section provided the inspiration for the crater shader.

Brief history of noise

- Developed by Ken Perlin in the early 80s
  - Ken worked on the revolutionary graphics for the movie *Tron*
  - Frustrated that *Tron's* graphics looked so “machine-like,” he wanted to escape the "machine-look ghetto."
- *Tron* was *not nominated* for the Academy Award for Special Effects
  - It “cheated” by using computers
  - What movie won?

Brief history of noise

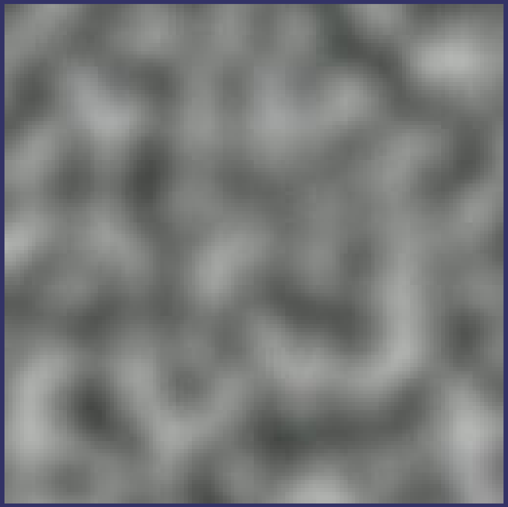
- Developed by Ken Perlin in the early 80s
  - Ken worked on the revolutionary graphics for the movie *Tron*
  - Frustrated that *Tron's* graphics looked so “machine-like,” he wanted to escape the "machine-look ghetto."
- *Tron* was *<u>not nominated</u>* for the Academy Award for Special Effects
  - It “cheated” by using computers
  - What movie won?
    - *E.T. the Extra Terrestrial* won, defeating *Blade Runner* and *Poltergeist*

Brief history of noise

- In 1983 Perlin worked on creating a space filling, apparently random signal function
  - Appear random
  - Be controllable
  - All features to be approximately the same size
  - All the features to be roughly isotropic
  - Have a range [-1, 1]
- First presented as a course at SIGGRAPH '84
  - The paper followed at SIGGRAPH '85
  - The Academy Award for Technical Achievement followed in 1997

Using Noise

- In Perlin's words, “noise is salt for graphics.”
  - Salt by itself is boring
  - Without salt, food is boring too



Original image from http://en.wikipedia.org/wiki/Perlin_noise

Using Noise

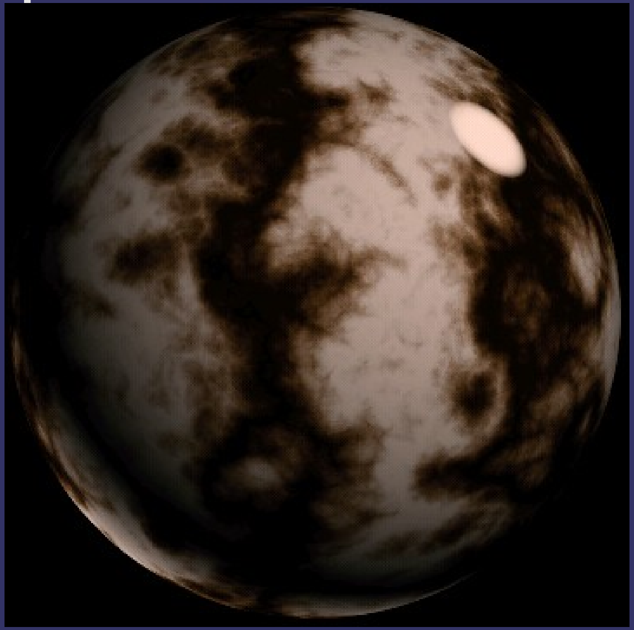
- Noise is typically used in multiple frequencies
  - Each frequency band is called an *octave*
  - As octave frequency increases, the amplitude decreases

$$NOISE(p) = \sum_{i=0}^{N-1} \frac{noise(f_i p)}{a_i}$$

Using Noise

- Add noise to boring functions or textures to make them interesting
  - Marble is the *classic* example

$$\sin\left(x+\left|NOISE\left(y\right)\right|\right)$$



Original image from http://www.noisemachine.com/talk1/23.html, copyright Ken Perlin

Implementing Noise

- Use GLSL noise function
  - Most (all?) implementations are *really* bad
  - Some just return a constant value for all inputs!

Implementing Noise

- Implement noise in C, generate noise texture
  - Tiling artifacts
  - Consumes texture resources

Implementing Noise

- Implement noise in GLSL code
  - Several implementations exist:

    Green, Simon. “Implementing Improved Perlin Noise.” GPU Gems 2. Ed. Matt Pharr. Upper Saddle River, NJ: Pearson Education, Inc., April 2005. http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter26.html

    Olano, Marc. “Modified Noise for Evaluation on Graphics Hardware.” Proceedings of Graphics Hardware 2005, Eurographics/ACM SIGGRAPH, July 2005. http://www.cs.umbc.edu/~olano/papers/mNoise.pdf
  - Most use several textures for tables
  - Use 60 – 80 GPU instructions

References

Perlin, K. 1999. Making Noise. Presented at GDCHardCore. http://www.noisemachine.com/talk1/

Perlin, K. 2002. Improving noise. In *Proceedings of the 29th Annual Conference on Computer Graphics and interactive Techniques* (San Antonio, Texas, July 23 - 26, 2002). SIGGRAPH '02. ACM, New York, NY, 681-682. http://mrl.nyu.edu/~perlin/noise/

Zucker, Matt. 2001. The Perlin noise math FAQ. http://www.cs.cmu.edu/~mzucker/code/perlin-noise-math-faq.html

http://freespace.virgin.net/hugo.elias/models/m_perlin.htm

Anti-aliasing Procedural Textures

- How can we control aliasing in procedural textures?
  - No magic mipmapping for procedural textures!
- Three common solutions:
  - Supersampling – expensive!
  - Analytical anti-aliasing – difficult!
  - Render to a texture, use mipmapping – sets an upper bound on texture resolution, may consume a lot of memory

Anti-aliasing – Supersampling

- Determine the size / shape of the sample area
  - The GLSL functions `dFdx()`, `dFdy()`, and `fwidth()` provide this information
    - These are called *partial derivatives*
    - Not available in unextended OpenGL ES 2.0
      - Added by `GL_OES_standard_derivatives`
  - Roughly the same information used by the texture filtering hardware

Anti-aliasing – Supersampling

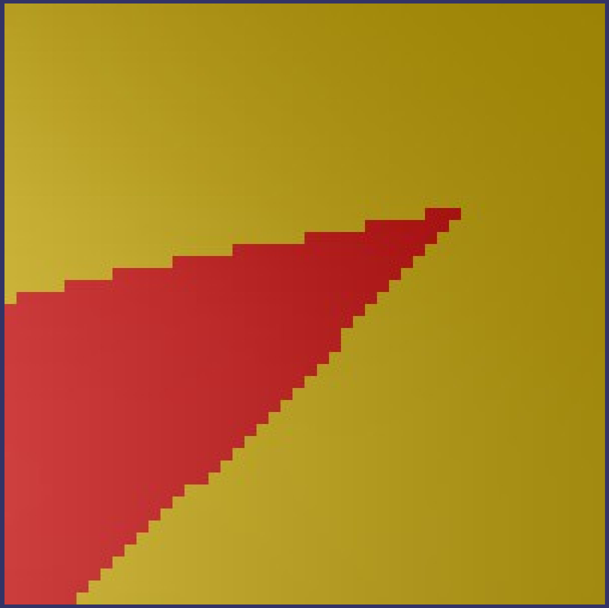
- Perform multiple texture calculations within the sample area
  - A rectangle based on `dFdx()` and `dFdy()` should be sufficient
  - Filter (average) the results

Anti-aliasing – Analytical

- Formulate the shader to calculate the average color over an area
  - Usually ranges from difficult to nearly impossible

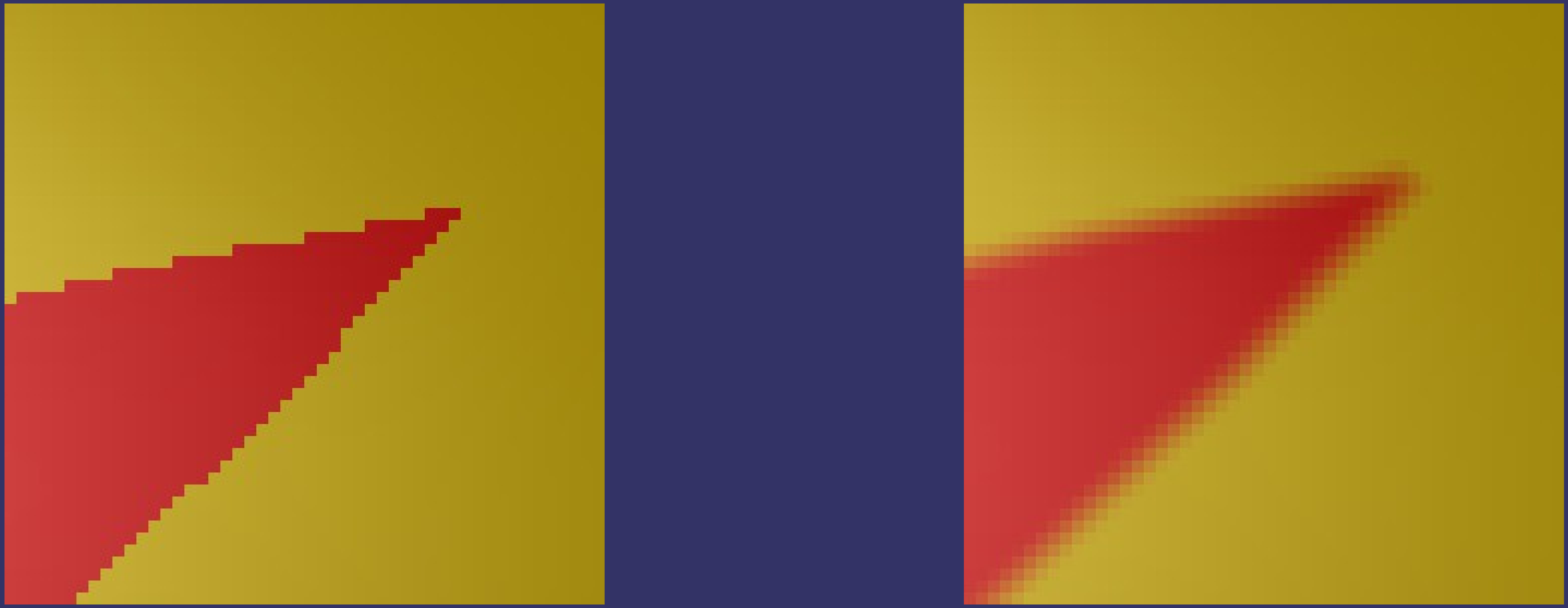
Anti-aliasing – Index Aliasing

- Sometimes the boundary function causes aliasing
  - Remember the toy ball shader:



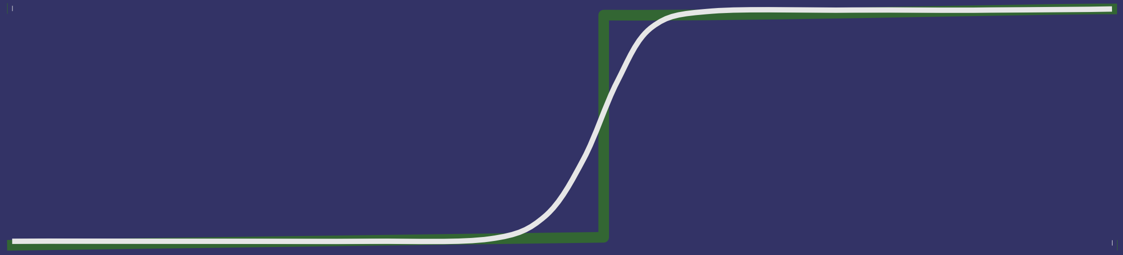
Anti-aliasing – Index Aliasing

- Sometimes the boundary function causes aliasing
  - Remember the toy ball shader:



Anti-aliasing – Index Aliasing

- `step` function adds unnecessary high frequency components
  - Instead use `smoothstep` based on the width of the sample area



  - Calculates: $-2t^3 + 3t^2$, $t \in [0, 1]$

References

Ebert, D. S., Musgrave, F. K., Peachey, D., Perlin, K., and Worley, S. *Texturing and Modeling: a Procedural Approach.* 3rd Ed. Morgan Kaufmann Publishers Inc., 2002.

Next week...

- And by “next week” I mean *tomorrow*...
- Quiz #1
- Render-to-texture
- Improving the lighting model
  - Environment maps as lights
  - Fresnel reflection

Legal Statement

This work represents the view of the authors and does not necessarily represent the view of Intel or the Art Institute of Portland.

OpenGL is a trademark of Silicon Graphics, Inc. in the United States, other countries, or both.

Khronos and OpenGL ES are trademarks of the Khronos Group.

Other company, product, and service names may be trademarks or service marks of others.